Руководство разработчика

C3D Labs

2026

Оглавление

1 Общие сведения **1**
- 1.1 C3D Toolkit . . . 1
- 1.2 Структура и отличительные особенности . . . 1
- 1.3 C3D API . . . 2
- 1.4 Выполняемые функции . . . 3
- 1.5 Теоретические основы . . . 4

2 Быстрый старт **5**
- 2.1 Комплектация . . . 5
- 2.2 Тестовое приложение . . . 5
- 2.3 Разработка в среде .NET . . . 8

3 C3D Modeler **9**
- 3.1 МЕТОДЫ ПОСТРОЕНИЯ ТЕЛ . . . 9
- 3.2 ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НАД ТЕЛАМИ . . . 104
- 3.3 МЕТОДЫ ПОСТРОЕНИЯ ДВУМЕРНЫХ КРИВЫХ . . . 239
- 3.4 МЕТОДЫ ПОСТРОЕНИЯ КРИВЫХ . . . 257
- 3.5 МЕТОДЫ ПОСТРОЕНИЯ ПОВЕРХНОСТЕЙ . . . 271
- 3.6 МЕТОДЫ ПРЯМОГО МОДЕЛИРОВАНИЯ . . . 287
- 3.7 МЕТОДЫ ПОСТРОЕНИЯ ТЕЛ ИЗ ЛИСТОВОГО МЕТАЛЛА . . . 317
- 3.8 ВСПОМОГАТЕЛЬНЫЕ МЕТОДЫ . . . 330
- 3.9 ЭЛЕМЕНТАРНЫЕ ОБЪЕКТЫ . . . 339
- 3.10 ГЕОМЕТРИЧЕСКИЕ ОБЪЕКТЫ . . . 348
- 3.11 КРИВЫЕ ДВУМЕРНОГО ПРОСТРАНСТВА . . . 355
- 3.12 КРИВЫЕ . . . 360
- 3.13 ПОВЕРХНОСТИ . . . 395
- 3.14 СПЕЦИАЛЬНЫЕ ОБЪЕКТЫ . . . 436
- 3.15 ТОПОЛОГИЧЕСКИЕ ОБЪЕКТЫ . . . 448
- 3.16 ОБЪЕКТЫ ГЕОМЕТРИЧЕСКОЙ МОДЕЛИ . . . 467
- 3.17 МНОГОПОТОЧНОСТЬ . . . 488
- 3.18 ФОРМАТ C3D . . . 496
- 3.19 АТРИБУТЫ . . . 513
- 3.20 ПОСТРОЕНИЕ ТРИАНГУЛЯЦИИ . . . 519
- 3.21 ПОСТРОЕНИЕ ПЛОСКИХ ПРОЕКЦИЙ . . . 530
- 3.22 ВЫЧИСЛЕНИЕ ИНЕРЦИОННЫХ ХАРАКТЕРИСТИК . . . 538

4 C3D B-Shaper **547**

4.1 Автоматический режим распознавания оболочки по полигональной сетке . . 548
4.2 Класс MbMeshProcessor – создание оболочки 550
4.3 Управление точностью распознавания 551
4.4 Редактирование сегментации полигональной сетки 551
4.5 Реконструкция поверхности на сегменте 552

5 C3D PolyShaper 555
5.1 Особенности реализации двусвязного списка рёбер 555
5.2 Сеточные итераторы 558
5.3 Сегментация и сегментационные итераторы 563
5.4 Топология полигонального объекта MbPolymeshTopo 567
5.5 Атрибуты сетки и сегментации 583
5.6 Реброзависимые атрибуты 584

6 C3D Collision Detection 587
6.1 Определение пересечения двух тел 587
6.2 Определение столкновений в наборе тел 588
6.3 Запросы на поиск столкновений 591
6.4 Настройка запроса на поиск столкновений 592
6.5 Группировка тел из контрольного набора 594

7 C3D Solver 595
7.1 ДВУМЕРНЫЙ ГЕОМЕТРИЧЕСКИЙ РЕШАТЕЛЬ 595
7.2 ДВУМЕРНЫЕ РАЗМЕРЫ 612
7.3 ДВУМЕРНЫЕ ЛОГИЧЕСКИЕ ОГРАНИЧЕНИЯ 624
7.4 ВЫЧИСЛЕНИЕ И ДИАГНОСТИКА ДВУМЕРНЫХ ОГРАНИЧЕНИЙ 634
7.5 ДВУМЕРНЫЕ СПЛАЙНЫ И ПАРАМЕТРИЧЕСКИЕ КРИВЫЕ 639
7.6 ТРЕХМЕРНЫЙ ГЕОМЕТРИЧЕСКИЙ РЕШАТЕЛЬ 640

8 C3D Converter 667
8.1 Принципы работы конвертера 668
8.2 Порядок работы с конвертером 668
8.3 Особенности обменных форматов 675
8.4 Свойство конвертера IConvertorProperty3D 675
8.5 Документ модели ItModelDocument 678
8.6 Индикатор процесса выполнения IProgressIndicator 680
8.7 Архитектура дерева модели 681
8.8 Атрибуты изделия 686
8.9 Импорт конфигураций модели 687
8.10 Плагины C3D Converter 688
8.11 Импорт IFC 689
8.12 Импорт метаданных строительных форматов 689
8.13 Вспомогательные методы конвертеров 690
8.14 Создание тела по пользовательским объектам 691

9 Справочные материалы 693
9.1 Коды результата операции 693
9.2 Способы построения поверхности сопряжения (скругления или фаски) 701

ГЛАВА 1

Общие сведения

1.1 C3D Toolkit

C3D Toolkit представляет собой программный продукт, включающий геометрическое ядро, параметрическое ядро, конвертер, модули визуализации и преобразования геометрических моделей.

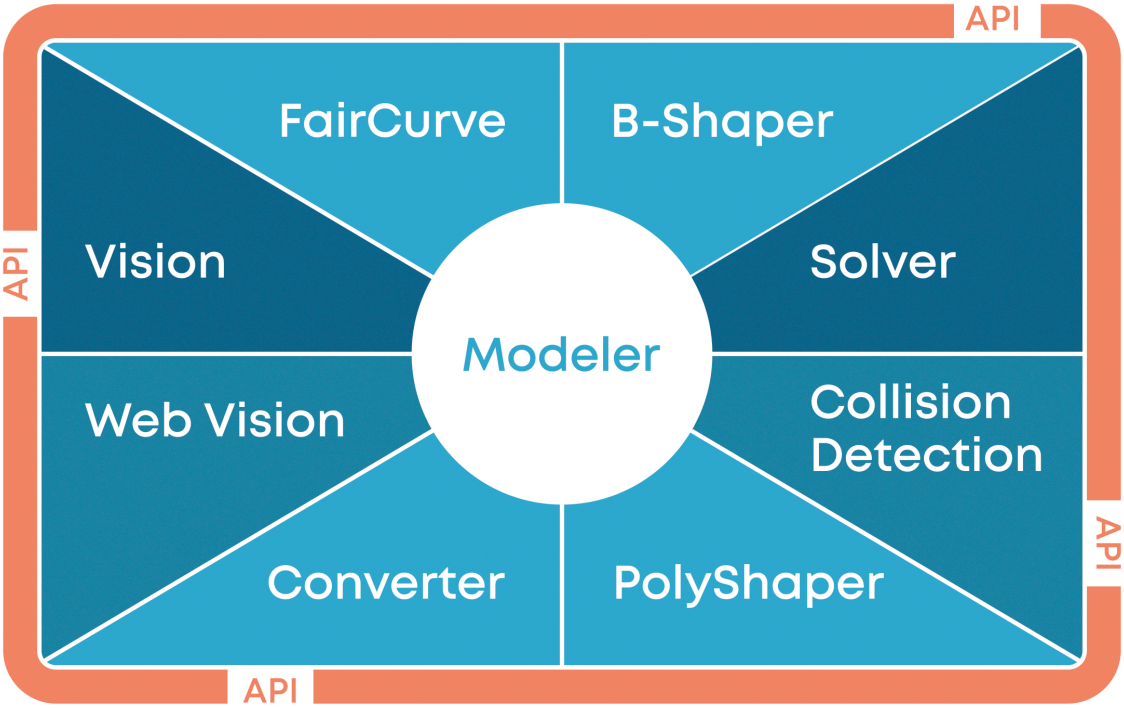
C3D Toolkit предназначен для использования в системах автоматизированного проектирования в качестве программного компонента.

C3D Toolkit сосредотачивает в себе программную реализацию математических методов построения численных моделей геометрии реальных и воображаемых объектов, а также математических методов обслуживания этих моделей. Численные модели используются в системах, выполняющих проектирование (Computer-Aided Design), расчёты (Computer-Aided Engineering) и производство (Computer-Aided Manufacturing) моделируемых объектов. Численные модели геометрии реальных и воображаемых объектов называют геометрическими моделями.

Геометрическая модель содержит описание формы моделируемого объекта и описание связей элементов модели. Кроме того, в геометрическую модель включают историю её построения, хранящую способы и последовательность построения модели, а элементы геометрической модели наделяют атрибутами, несущими информацию о физических, технологических и других свойствах элементов.

1.2 Структура и отличительные особенности

C3D Toolkit состоит из пяти модулей, приведённых на рисунке: геометрического ядра C3D Modeler, параметрического ядра C3D Solver, модуля обмена данными C3D Converter, модуля визуализации C3D Vision и C3D B-Shaper, осуществляющего преобразование полигональных моделей в твердотельные с граничным представлением (B-Rep).



Геометрическое ядро C3D Modeler выполняет построение геометрической модели, редактирование модели путем изменения ее внутренних данных, построение триангуляции, вычисление инерционных характеристик модели, построение плоских проекций модели, определение столкновений элементов модели.

Параметрическое ядро C3D Solver обеспечивает взаимосвязь элементов геометрической модели, что позволяет редактировать модель, синхронно изменяя ее элементы, строить подобные модели, моделировать механизмы.

C3D Converter осуществляет обмен информацией о геометрической модели с другими системами.

Модуль визуализации C3D Vision осуществляет качественную визуализацию геометрической модели и обеспечивает взаимодействие с интерфейсом инженерного ПО.

C3D B-Shaper преобразует полигональные модели в твердотельные с граничным представлением.

Наличие в одном программном продукте геометрического ядра, параметрического ядра и конвертера является одной из отличительных особенностей C3D.

1.3 C3D API

Все модули C3D предоставляют программные интерфейсы (API) для использования их функциональности в инженерном приложении, которые состоят из констант, перечислений, функций, структур, классов.

Особенностями C3D API являются открытость и расширяемость. C3D предоставляет прямой доступ к объектам, что позволяет расширить функциональность за счет наследования от объектов C3D.

Кроме того, важной особенностью C3D API являются его стабильность и обратная совместимость. Стабильность означает, что API не должен сильно меняться в течение как можно более длительного периода времени. Обратная совместимость означает, что код, написанный с использованием предыдущей версии, функционально работает так же, как и со следующей версией API.

Для случаев, когда возникает необходимость модифицировать существующий интерфейс, существует четкая процедура замены его на новый интерфейс, направленная в первую очередь на поддержание удобства использования API пользователями.

Интерфейс, который необходимо изменить, объявляется устаревшим с помощью макросов DEPRECATE_DECLARE и DEPRECATE_DECLARE_REPLACE, определенных в файле *math_x.h*. Эти макросы отображают предупреждающее сообщение при компиляции кода, использующего устаревшую функцию. Кроме того, на платформе Windows выданное предупреждение содержит информацию о сроках удаления устаревшего интерфейса, а макрос DEPRECATE_DECLARE_REPLACE дополнительно указывает интерфейс, который заменяет устаревшую функцию.

1.4 Выполняемые функции

Геометрическое ядро C3D Modeler строит геометрическую модель, которая содержит описание формы моделируемого объекта, описание связей элементов геометрической модели, историю построения модели, атрибуты элементов геометрической модели.

Для описания формы моделируемого объекта геометрическое ядро C3D Modeler использует граничное представление (Boundary Representation). Геометрическое ядро поддерживает также полигональное представление (Polygonal Representation). Для построения геометрической модели используются методы твердотельного моделирования (Solid Modeling), методы поверхностного моделирования (Surface Modeling), методы прямого моделирования (Direct Modeling) и методы моделирования из листового металла (Sheet Metal Modeling).

Используемые для построения модели методы, их последовательность и необходимые исходные данные хранятся в журнале построения. Журнал построения позволяет редактировать геометрическую модель путем полного перестроения модели с новыми параметрами.

Дополнительная информация об элементах геометрической модели хранится в атрибутах. Атрибутами снабжены объекты геометрической модели, а также отдельные элементы этих объектов.

Геометрическое ядро C3D Modeler выполняет построение полигональной модели по ее граничному представлению. Полигональная модель строится путем триангуляции элементов геометрической модели и используется для визуализации и расчетов. Кроме того, геометрическое ядро выполняет построение плоских проекций геометрической модели (Mapping), вычисление инерционных характеристик геометрической модели (Inertia Properties), определение столкновений элементов модели (Collision Detection).

Параметрическое ядро C3D Solver налагает вариационные связи на элементы геометрической модели. Эти связи называют геометрическими ограничениями (Geometric Constraints). Геометрические ограничения работают с трехмерными и двумерными объектами геометрической модели. Геометрические ограничения представляют собой условия, наложенные на элементы модели и выраженные с помощью уравнений. Геометрические ограничения позволяют создавать сборочные единицы путем наложения зависимостей на составляющие ее детали. Размеры одной или нескольких деталей можно связать уравнениями, что дает возможность синхронно редактировать элементы геометрической модели. Если связать почти все степени свободы геометрической модели и описать закон

изменения для оставшихся степеней свободы, то можно привести в движение геометрическую модель и смоделировать механизм.

Модуль C3D Converter обеспечивает обмен данными с другими системами путем чтения и записи геометрической модели в форматах ACIS, IGES, STEP, X_T, VRML, OBJ, SAT, X_B, STL, JT, 3MF, SAB. Форматы ACIS, IGES, STEP, X_T, SAT, X_B, SAB передают информацию о геометрической форме моделируемого объекта с помощью граничного представления. Форматы STL, VRML, OBJ, 3MF для передачи геометрической формы моделируемого объекта используют полигональное представление. Формат JT может содержать модель как в граничном, так и в полигональном представлении.

Модуль визуализации C3D Vision отвечает за визуальное отображение геометрических моделей и функционирование графического интерфейса инженерного приложения. Компонент управляет качеством отрисовки геометрических моделей, используя математические, программные и аппаратные средства, вследствие чего повышается скорость работы программного обеспечения с большими сборками. Применение C3D Vision в разработке открывает ряд новых возможностей для управления трехмерными сценами, позволяет задействовать готовое дерево построения 3D-моделей, анимацию, интерактивные средства манипуляции со сценой, действующие в режиме «пользователь-компьютер», а также виртуальные устройства, являющиеся неотъемлемой частью современного интерфейса инженерного ПО.

Модуль C3D B-Shaper преобразует полигональные модели в твердотельные с граничным представлением. В результате выполнения алгоритмов пользователь может редактировать полученную модель привычными инструментами CAD-системы, значительно уменьшив сложность и объем данных.

1.5 Теоретические основы

Граничное представление, с которым работает C3D, дает точное описание геометрической формы моделируемого объекта. Для описания геометрической формы C3D использует набор граней, проходящих по границе, отделяющей внутреннее пространство моделируемого объекта от остальной части пространства. Грани представляют собой криволинейные поверхности, стыкующиеся друг с другом по своим краям. Края граней могут иметь сложную форму. Формирование и стыковка граней выполняются во время построения модели. Это обеспечивают методы построения модели и организация данных в C3D.

Геометрические ограничения, описывающие связи элементов модели и прочие условия, формулируются в виде уравнений. Для поиска решения, удовлетворяющего уравнениям, параметрическое ядро C3D Solver использует вариационный подход. Вариационный подход обеспечивает равноправие всех геометрических ограничений.

С помощью триангуляции граничное представление позволяет построить полигональное представление модели, которое используется для визуализации и геометрических расчетов. Полигональные объекты состоят из треугольных и четырехугольных пластин, аппроксимирующих грани, и ломаных, аппроксимирующих ребра. Триангуляция в геометрическом ядре C3D Modeler выполняется по принципу Делоне в плоскости параметров поверхностей.

Для кривых и поверхностей геометрическое ядро C3D Modeler может создать NURBS (Non-Uniform Rational B-Spline) копии. NURBS-объекты используются для прямого моделирования и для обмена данными, когда отсутствует прямое соответствие между объектами C3D и объектами обменных форматов.

Применяемые в C3D математические объекты, методы и алгоритмы приведены в книге Голованова Н. Н. «Геометрическое моделирование», 2-е изд. — Москва: ДМК Пресс, 2024.

ГЛАВА 2

Быстрый старт

2.1 Комплектация

В комплект поставки C3D входят библиотечные файлы *c3d.lib*, *c3d.dll*, *libc3d.so*, *libc3d.dylib* и набор заголовочных файлов *Include*.h*. В операционной системе Windows библиотечные файлы собраны в конфигурациях 32bit/64bit, Unicode/Multibyte, Debug/Release в средах разработки Visual Studio 2019 и Visual Studio 2022. В операционной системе Linux библиотечные файлы собраны компиляторами GCC и Clang в конфигурациях 64bit, Unicode, Debug/Release. В операционной системе macOS библиотечные файлы собраны компилятором Clang в конфигурациях 64bit, Unicode, Debug/Release. В операционной системе FreeBSD файлы библиотеки собраны компилятором Clang в конфигурациях amd64, Unicode/Multibyte, Release/Debug.

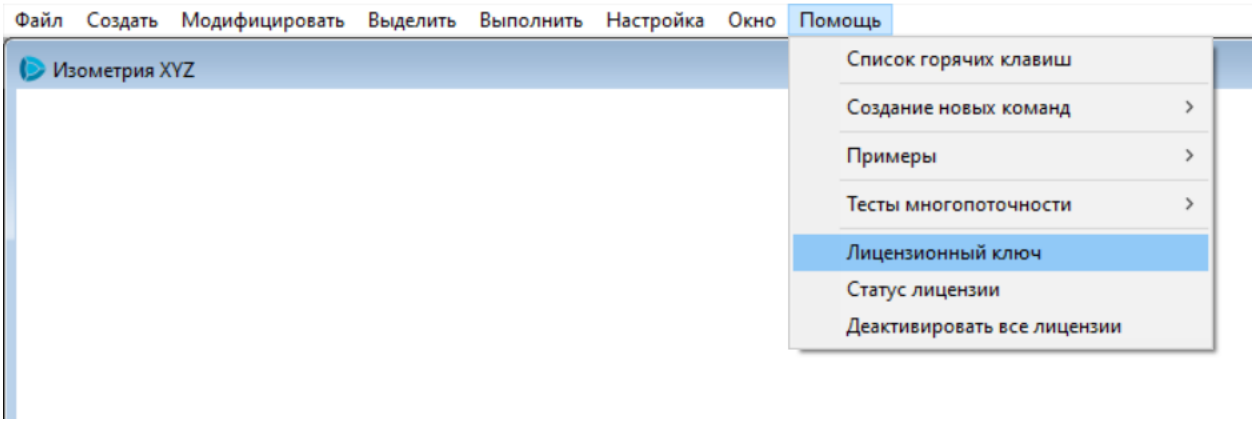
По заголовочным файлам *Include*.h* сгенерирована документация C3D. Заголовочные файлы содержат описание объектов и методов C3D на русском и английском языках. Описание объектов и методов C3D также приведено в данном руководстве. В файле *Changes.txt* приведены изменения интерфейса программного продукта.

В комплект поставки входит обертка для C3D, позволяющая использовать технологию .NET при разработке приложений на языке C#.

Вместе с C3D поставляется приложение *test.exe* для системы Windows, демонстрирующее возможности C3D, исходные тексты этого приложения, файл *CMakeLists.txt* для генерации проекта приложения и набор файлов с моделями.

2.2 Тестовое приложение

В каталоге *Example\Demo* находится готовое для использования в системе Windows тестовое приложение *test.exe*, демонстрирующее возможности геометрического ядра *c3d.dll*. После запуска приложения *test.exe* необходимо ввести ключ и подпись, выбрав пункт меню «Помощь -> Лицензионный ключ»:

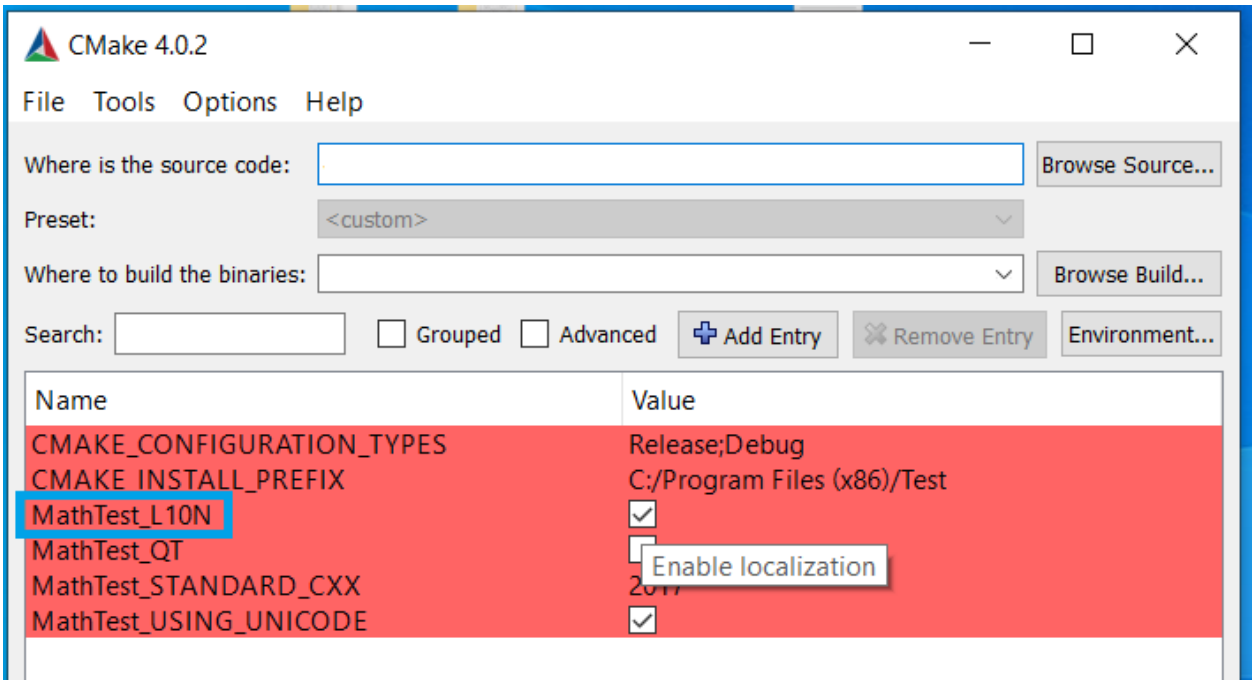


В каталоге *Example\Models* находятся примеры моделей в виде файлов *.c3d*, *.stp*, *.x_t*, *.stl*, с которыми может работать тестовое приложение *test.exe*.

В каталоге *Example\Source* находятся исходные тексты тестового приложения *test.exe*, а также файл *CMakeLists.txt* для генерации проекта/решения тестового приложения.

Для компиляции тестового приложения геометрического ядра C3D выполните следующие действия:

1. Создайте в удобном для работы месте новый каталог, например «TestApp».
2. В каталоге <C3D> выберите архив, соответствующий среде разработки.
3. Скопируйте из выбранного архива каталоги <Include>, <Debug> и <Release> в новый каталог.
4. Скопируйте из архива <Example> каталог <Source> в новый каталог.
5. Убедитесь, что в новом каталоге находятся четыре папки: <Debug>, <Include>, <Release> и <Source>.
6. Установите CMake, выбрав в процессе установки опцию «Add CMake to the system PATH for all users».
7. Создайте проект/решение тестового приложения, выполнив следующие действия:
   - Запустите CMake версии не ниже 3.24.
   - Для «Where is the source code» укажите каталог <path_to_testapp>\TestApp\Source.
   - Для «Where to build the binaries» укажите каталог <path_to_testapp>\TestApp\Build.
   - Нажмите кнопку «Configure» для конфигурирования проекта/решения.
   - На запрос «Create Directory» подтвердите создание каталога <path_to_testapp>\TestApp\Build.
   - По запросу «Specify the generator for this project» укажите соответствующую версии C3D конфигурацию среды разработки.
   - Установите флаг «MathTest_L_10N» для генерации приложения с русскоязычным интерфейсом:



- Нажмите кнопку «Generate» для генерации файлов проекта/решения.

Примечание: если в PATH добавлен каталог с дистрибутивом Qt5, то, задав в CMake опцию «MathTest_QT», можно создать проект, использующий для создания окна приложения библиотеку Qt5 вместо Win32 API.

8. Под Windows запустите созданное решение TestApp\Build\Test.sln тестового приложения в вашей среде разработки.
9. Под Linux можно использовать команды:

```
cmake -S Source -B Build -DCMAKE_BUILD_TYPE=Release  # опционально, эквивалентно␣
↪пункту 7
cmake --build Build -- -j4 # пакеты с Qt5 должны быть предустановлены, например,␣
↪для Ubuntu командой "sudo apt install qtbase5-dev qt5-qmake"
```

10. Если при использовании CMake возникли проблемы, можно воспользоваться готовым проектом Test.vcxproj, расположенным в каталоге Example (проект подготовлен для MS Visual Studio, конфигурация Unicode, MS Windows). Для сборки проекта скопируйте файлы проекта Test.vcxproj, Test.vcxproj.filters в каталог, созданный на шаге 1, и запустите проект Test.vcxproj тестового приложения в вашей среде разработки.
11. Для активации геометрического ядра C3D необходимо прописать полученный лицензионный ключ в файл *c3d.lic*. Для Windows файл расположен в каталоге USERPROFILE (C:\Users\user_name). Для UNIX-систем файл расположен в каталоге HOME (/home/user_name).
12. После компиляции запустите созданное тестовое приложение *test.exe* из каталога \Debug или \Release.
13. Если есть необходимость сборки тестового приложения с использованием Conan-пакетов ядра и примеров к нему (пакеты c3d и c3d-examples), можно воспользоваться краткой инструкцией внутри файла *conanfile-examples-usage.py* в каталоге \Source. Для сборки потребуется указать номер версии пакета ядра.

2.3 Разработка в среде .NET

C3D может работать в среде .NET. Для разработки приложений в среде .NET следует использовать обертку, входящую в комплект поставки C3D.

Обертка для C3D представляет собой dll-файл *NetC3D.dll*, собираемый на платформе .NET Framework 4.5.2 в конфигурациях 32bit/64bit, Debug/Release и средах разработки Visual Studio 2019, Visual Studio 2022. Библиотека скомпилирована с поддержкой подписи «Strong Name».

Для использования C3D в приложениях, разрабатываемых на C#, нужно выполнить следующие действия:

1. Выбрать из комплекта C3D файл *NetC3D.dll* нужной конфигурации: 32bit/64bit, Debug/Release в одной из сред разработки: Visual Studio 2019, Visual Studio 2022.
2. Положить в один каталог с *NetC3D.dll* файл *c3d.dll* из этого же комплекта, той же конфигурации и среды разработки: 32bit/64bit, Debug/Release, Visual Studio 2019, Visual Studio 2022.
3. Добавить файл *NetC3D.dll* в разрабатываемый проект: *References -> Add Reference -> Browse...*, далее выбрать файл *NetC3D.dll*.
4. Для дальнейшей работы перед вызовом функций из *NetC3D.dll* нужно ввести лицензионный ключ и сигнатуру. Способ ввода ключа и сигнатуры может быть следующим:

```
System.String key = Environment.GetEnvironmentVariable("C3Dkey");
System.String signature = Environment.GetEnvironmentVariable("C3Dsignature");
NetC3D.ToolEnabler.EnableMathModules(key, signature);
```

где *C3Dkey* и *C3Dsignature* — переменные среды, содержащие ключ и сигнатуру.

ГЛАВА 3

C3D Modeler

Модуль C3D Modeler выполняет все геометрические расчеты, необходимые для построения 2D-эскизов и 3D-моделей. Для описания формы моделируемого объекта в C3D Modeler используется граничное представление геометрии, при этом модель строится из трехмерных тел, которые создаются с помощью поверхностей и кривых. Далее тела группируются в сборочные единицы, из которых строятся сборочные единицы следующего уровня.

Вместе с граничным представлением модели поддерживается полигональное представление геометрии. Элементами полигональных моделей являются аппроксимационные объекты, построенные из пластин и полигонов. Такие модели используются для расчётов и визуализации.

Объекты геометрической модели, такие как B-Rep тела, полигональные объекты и каркасы, снабжены журналами построения, в которых хранятся методы, их последовательность и необходимые исходные данные для построения объектов.

Журнал построения позволяет редактировать геометрическую модель и перестраивать модель с новыми параметрами. Информация о геометрии модели в виде поверхностей, кривых и точек хранится в топологических элементах объектов модели, таких как грани, ребра и вершины.

Дополнительная информация для каждого элемента объекта модели и самого объекта в целом может храниться в них в виде атрибутов.

Для активации лицензии на модуль необходимо вызвать функцию EnableMathModules() с соответствующими параметрами.

3.1 МЕТОДЫ ПОСТРОЕНИЯ ТЕЛ

Основными элементами геометрической модели служат тела. Геометрическое ядро C3D строит тела, описывающие всю поверхность или часть поверхности моделируемого объекта.

Тела могут быть замкнутыми и незамкнутыми. Замкнутое тело не содержит краевых ребер и описывает всю поверхность моделируемого объекта, а также множество его внутренних точек. Незамкнутое тело содержит краевые ребра и описывает только часть поверхности моделируемого объекта. Многие тела имеют простую форму и строятся на основе точек, кривых и поверхностей.

3.1.1 Построение элементарного тела

Функция

```
MbResultType ElementarySolid( const MbElementarySolidParams  & params,
                                    MbElementarySolidResults & results );
```

выполняет построение элементарного тела по заданным точкам и типу создаваемого тела (см. таблицу 1), а также по элементарной поверхности (сфера, тор, конус, цилиндр).

Входным параметром является объект **params** класса MbElementarySolidParams (см. определение в *op_shell_parameter.h*), который содержит исходные данные для операции создания тела.

Выходным параметром функции является объект **results** класса MbElementarySolidResults, наследника класса MbOperationResults (см. определение в *op_shell_parameter.h*), содержащий построенное тело и обобщенные результаты выполнения функции.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Построение элементарного тела по контрольным точкам

При построении тел по контрольным точкам для инициализации объекта **params** с параметрами операции в классе MbElementarySolidParams определен следующий конструктор (см. *op_shell_parameter.h*):

```
MbElementarySolidParams( const ElementaryShellType      & solidType,
                         const c3d::SpacePointsVector   & points,
                         const MbSNameMaker             & operNames );
```

с параметрами:

- **solidType** — тип создаваемого тела;
- **points** — множество точек;
- **operNames** — именователь операции.

Для разных типов создаваемого тела требуется разное количество контрольных точек. В таблице 1 приведено количество контрольных точек, необходимое для создания тела типа **solidType**:

Таблица 1 - Количество контрольных точек для различных типов элементарных тел

solidType	Тип тела	Количество контрольных точек
et_Sphere	шар	3 точки
et_Torus	тор	3 точки
et_Cylinder	цилиндр	3 точки
et_Cone	конус	3 точки
et_Cone	усеченный конус	4 точки
et_Block	блок	4 точки
et_Wedge	клин	4 точки
et_Prism	призма	количество вершин основания + 1 точка
et_Pyramid	пирамида	количество вершин основания + 1 точка
et_Plate	плита	4 точки
et_Icosahedron	икосаэдр	3 точки
et_Polyhedron	многогранник	3 точки
et_Tetrapipe	тетра-труба	3 точки
et_Octapipe	окта-труба	3 точки

При построении сферы точка **points**[0] определяет центр сферы, точка **points**[1] определяет направление оси **axisZ** локальной системы координат сферы, точка **points**[2] вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат сферы. Расстояние между точками **points**[0] и **points**[2] определяет радиус сферы (см. рис. 1):

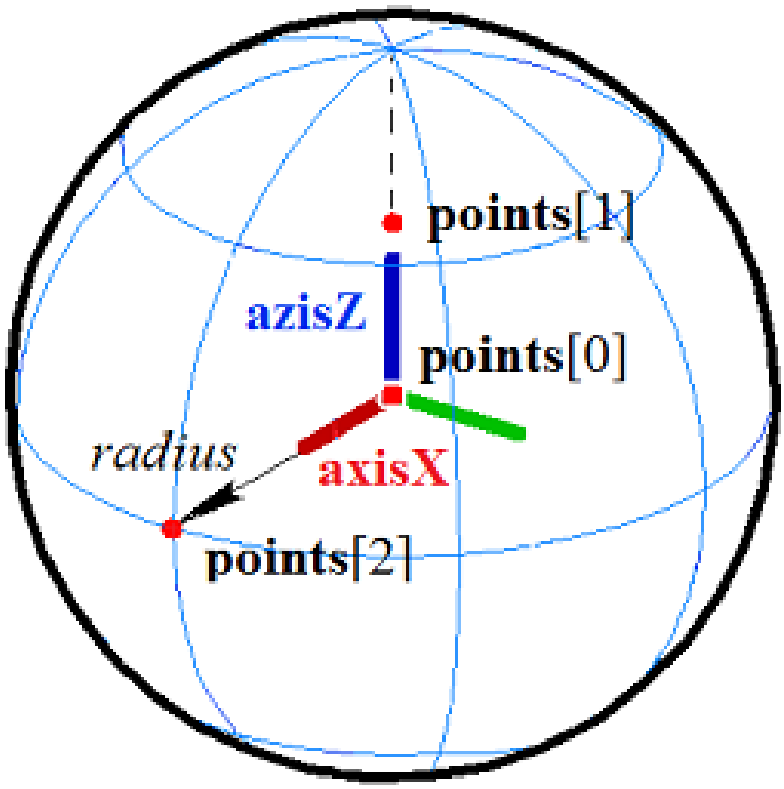


Рис. 1

При построении тора точка **points**[0] определяет центр тора, точка **points**[1] определя-

ет направление оси **axisX** локальной системы координат тора, точка **points**[2] вместе с предыдущими точками определяет плоскость, в которой располагаются оси **axisX** и **axisZ** локальной системы координат тора. Расстояние между точками **points**[0] и **points**[1] определяет больший радиус тора, расстояние между точками **points**[1] и **points**[2] определяет меньший радиус тора (рис. 2):

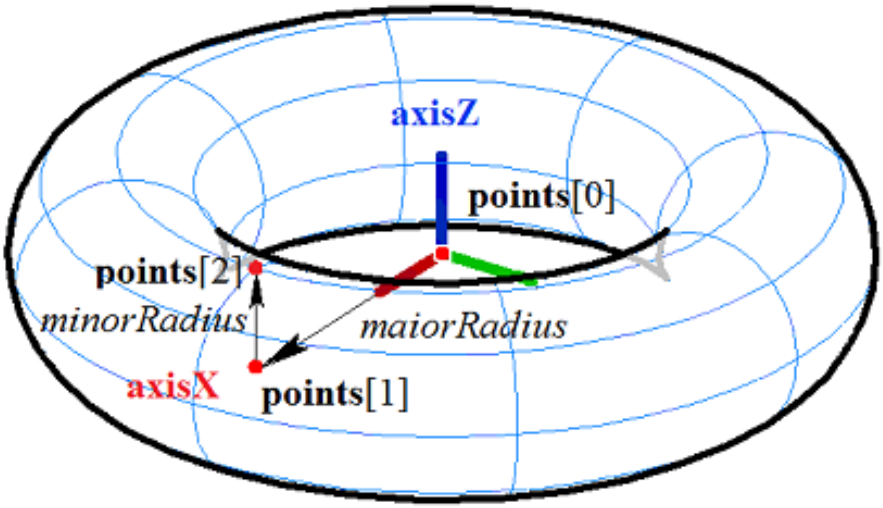


Рис. 2

При построении цилиндра точка множества **points**[0] определяет центр нижнего основания цилиндра, точка **points**[1] определяет центр верхнего основания цилиндра и направление оси **axisZ** локальной системы координат цилиндра, точка **points**[2] вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат цилиндра. Расстояние между точками **points**[0] и **points**[1] определяет высоту цилиндра, расстояние от оси **axisZ** до точки **points**[2] определяет радиус цилиндра (рис. 3):

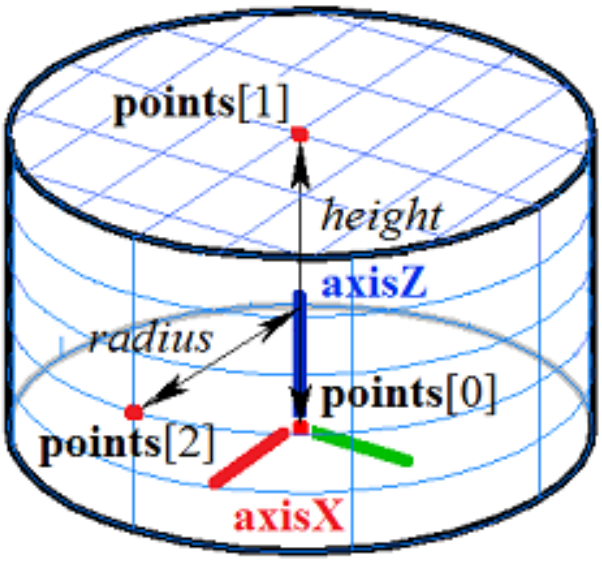


Рис. 3

При построении конуса точка множества **points**[0] определяет вершину конуса, точка **points**[1] определяет центр основания конуса и направление оси **axisZ** локальной системы координат конуса, точка **points**[2] вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат конуса. Расстояние между точками **points**[0] и **points**[1] определяет высоту конуса, угол конуса определяется из условия, что точка **points**[2] лежит на боковой поверхности конуса (рис. 4):

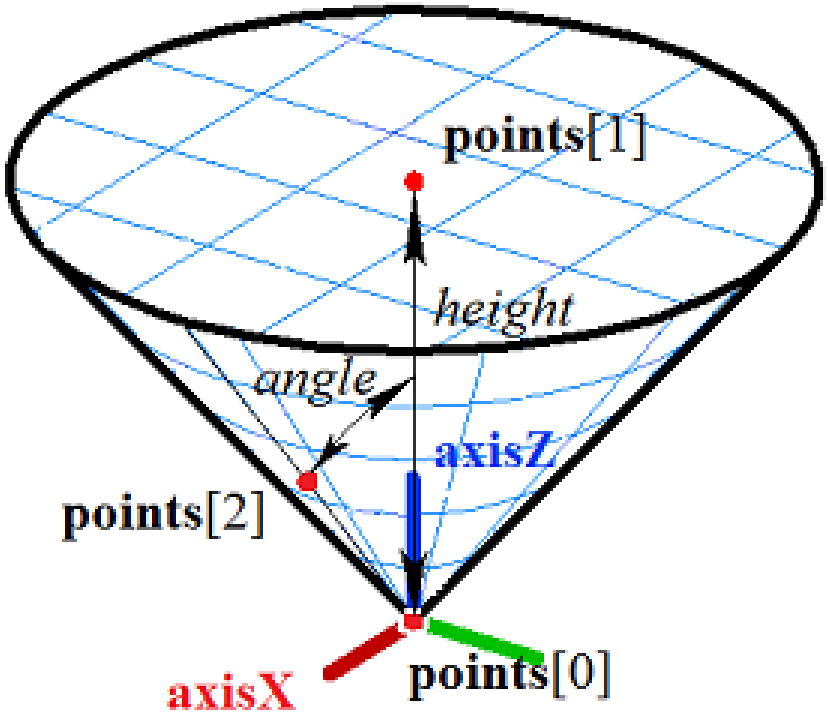


Рис. 4

При построении усеченного конуса точка множества **points**[0] определяет вершину конуса, точка **points**[1] определяет центр большего основания конуса и направление оси **axisZ** локальной системы координат конуса, точка **points**[2] определяет центр меньшего основания конуса, точка **points**[3] вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат конуса. Расстояние между точками **points**[1] и **points**[2] определяет высоту усеченного конуса, угол конуса определяется из условия, что точка **points**[3] лежит на боковой поверхности конуса (рис. 5):

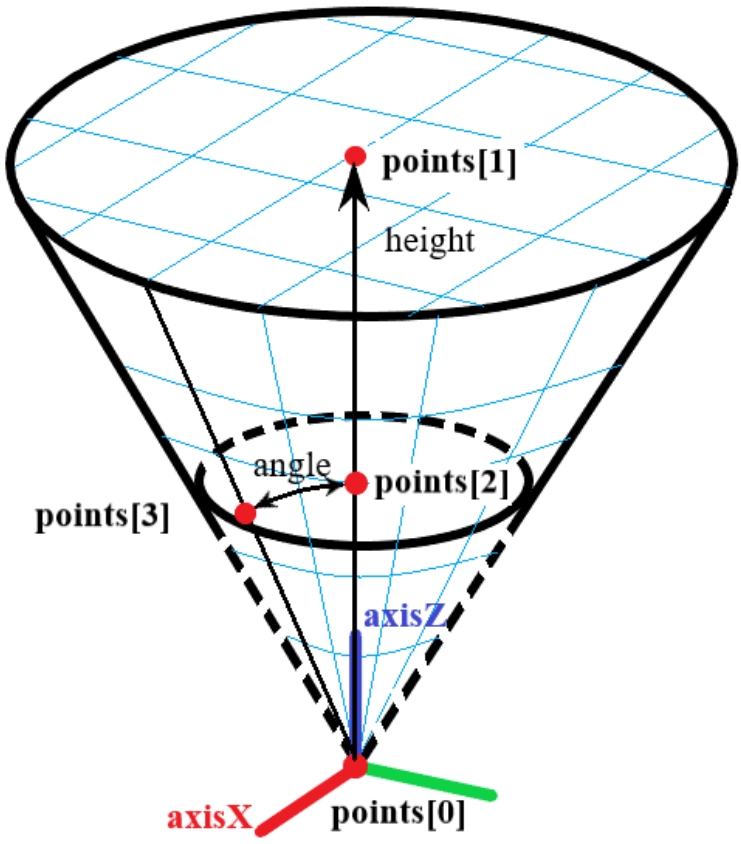


Рис. 5

При построении прямоугольного блока точки множества **points**[0] и **points**[1] определяют ребро и две вершины блока, точка **points**[2] вместе с предыдущими точками определяет плоскость нижнего основания блока. Через точку **points**[2] проходит ребро блока, параллельное ребру **points**[0]– **points**[1]. Точка **points**[3] определяет плоскость верхнего основания блока (рис. 6):

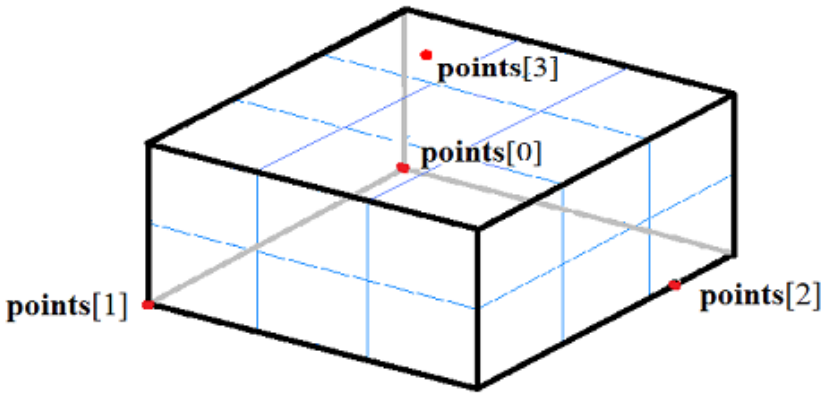


Рис. 6

При построении прямоугольного клина точки множества **points**[0] и **points**[1] определяют

ребро и две вершины клина, точка **points**[2] вместе с предыдущими точками определяет плоскость нижнего основания клина и его вершину. Через точку **points**[2] проходит ребро клина, параллельное ребру **points**[0]– **points**[1]. Точка **points**[3] определяет плоскость верхнего основания клина (рис. 7):

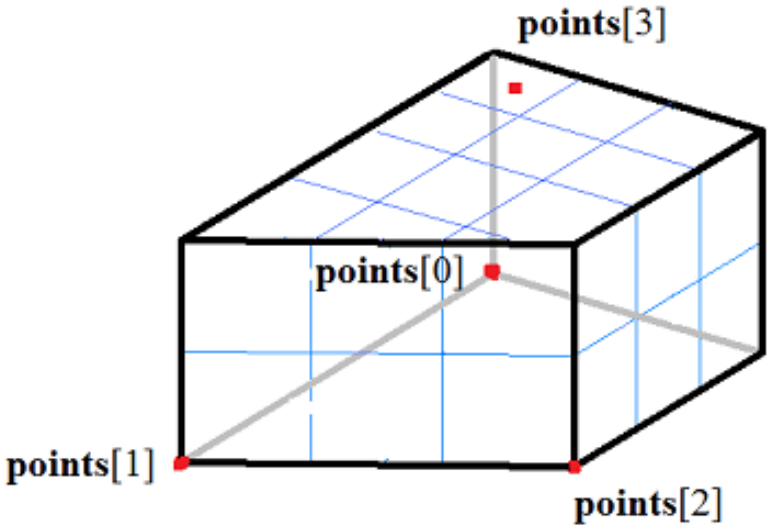


Рис. 7

При построении плиты с цилиндрическими торцами точки множества **points**[0] и **points**[1] определяют ребро и две вершины плиты, точка **points**[2] вместе с предыдущими точками определяет плоскость нижнего основания пластины. Через точку **points**[2] проходит ребро плиты, параллельное ребру **points**[0]– **points**[1]. Точка **points**[3] определяет плоскость верхнего основания пластины (рис. 8):

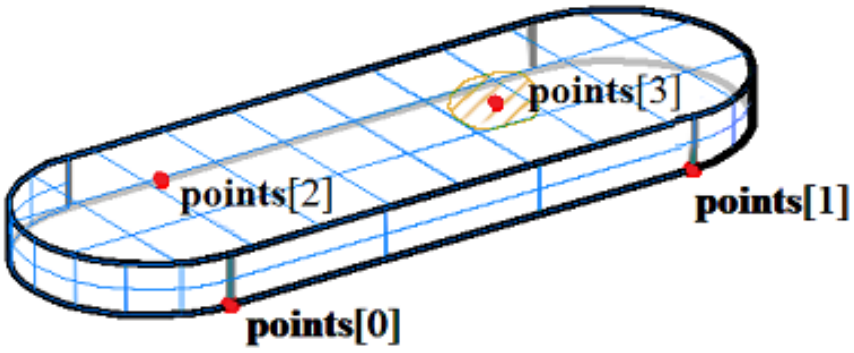


Рис. 8

При построении прямой призмы точки **points**[0], **points**[1], ..., **points**[n-1] определяют многоугольник основания. Высоту призмы определяет расстояние от плоскости нижнего основания до последней точки **points**[n]. На рис. 9 приведена прямая призма с пятиугольным основанием:

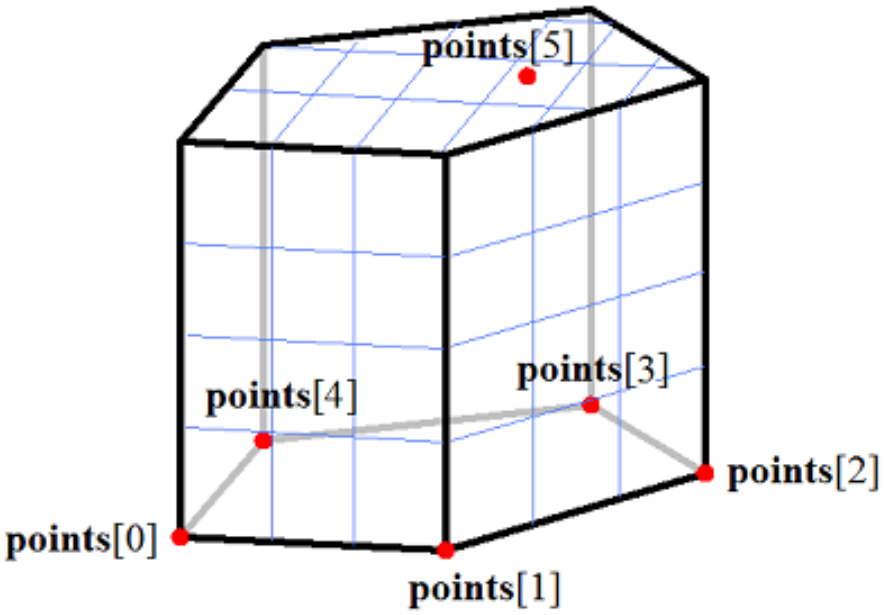


Рис. 9

При построении пирамиды с многоугольником в основании точки **points**[0], **points**[1], ..., **points**[n-1] определяют многоугольник основания, последняя точка **points**[n] определяет вершину пирамиды. На рис. 10 приведена пирамида с пятиугольным основанием:

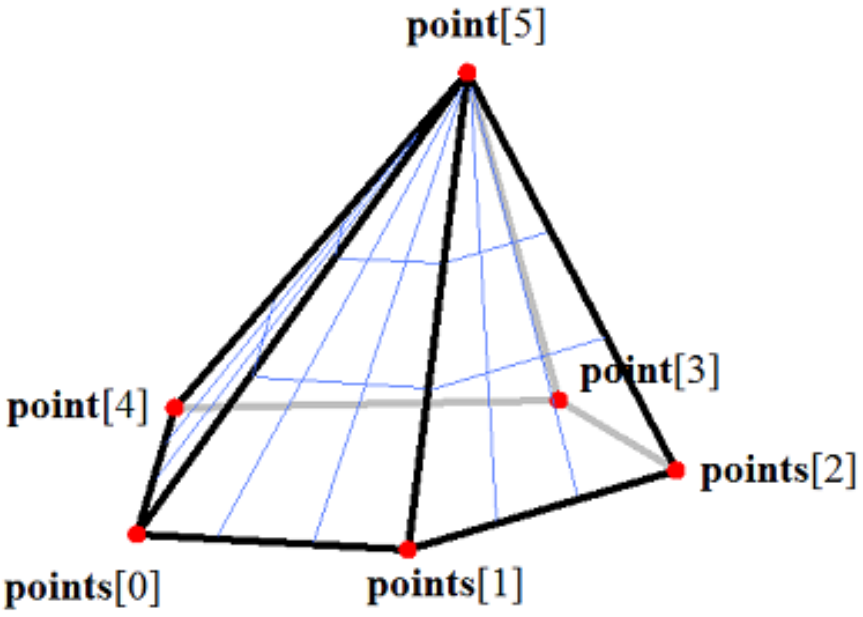


Рис. 10

Многоугольник основания призмы или пирамиды может быть произвольным.

При построении икосаэдра точки **points**[0], **points**[1], **points**[2] определяют вершины и плоскости всех граней фигуры (рис. 11):

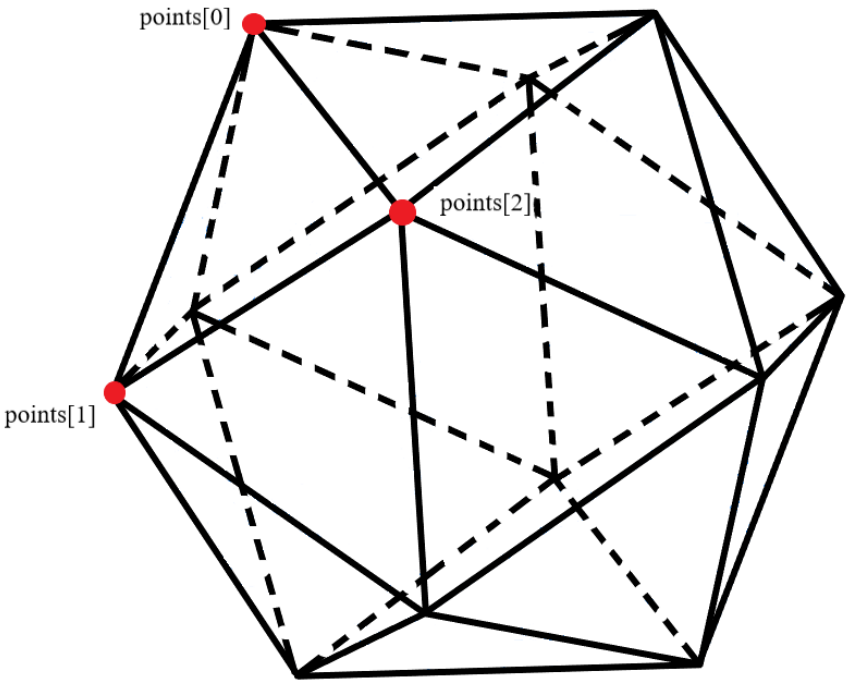


Рис. 11

Построение элементарного тела по заданной поверхности

Для создания тел по элементарной поверхности необходимо использовать функцию *ElementarySolid()* с использованием следующего конструктора для инициализации объекта **params** с параметрами операции (см. *op_shell_parameter.h*):

```
MbElementarySolidParams( const SPtr<const MbElementarySurface> & surface,
                         const MbSNameMaker                     & operNames );
```

с параметрами:

- **surface** — элементарная поверхность (сфера, тор, цилиндр, конус);
- **operNames** — именователь операции.

Элементарной поверхностью может быть сфера MbSphere, поверхность тора MbTorus, цилиндрическая поверхность MbCylinder, коническая поверхность MbCone. На рис. 12 приведена сферическая поверхность и построенное по ней тело:

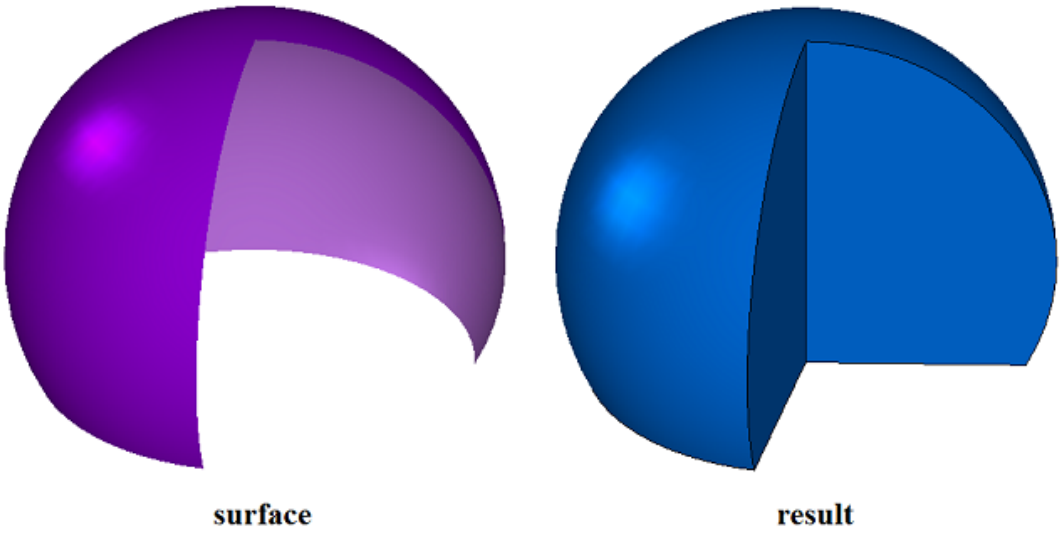


Рис. 12

На рис. 13 приведена поверхность тора и построенное по ней тело:

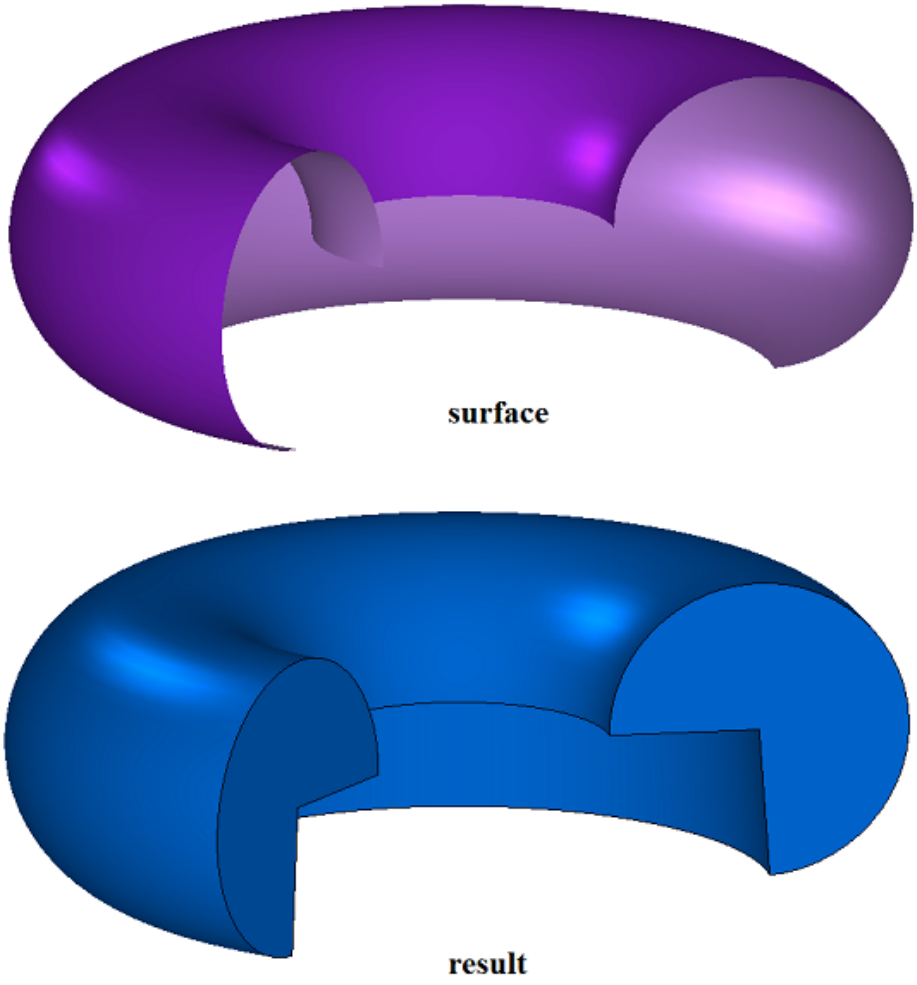


Рис. 13

На рис. 14 приведена цилиндрическая поверхность и построенное по ней тело:

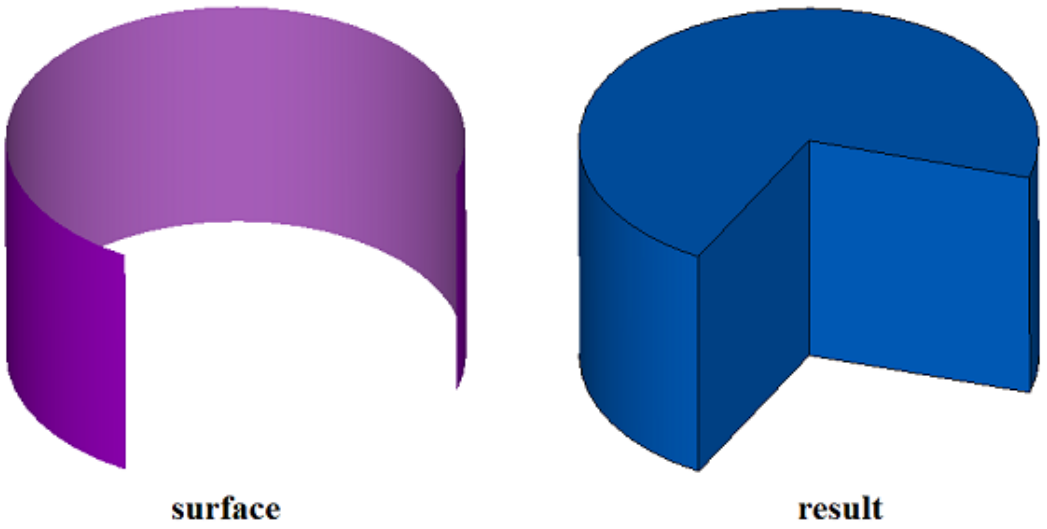


Рис. 14

На рис. 15 приведена коническая поверхность и построенное по ней тело:

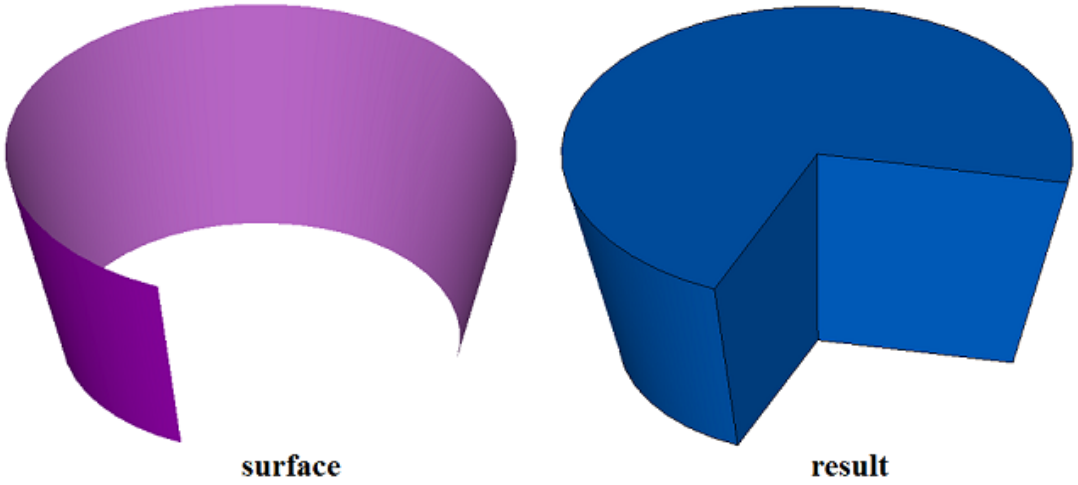


Рис. 15

Если перечисленные поверхности являются циклически замкнутыми, то построенные по ним тела будут иметь соответствующую форму. Если поверхность не является ни одной из перечисленных элементарных поверхностей, функция возвращает код ошибки *rt_Error*.

Функция ElementarySolid() добавляет в журнал построенного тела builder MbElementarySolid (объявлен в файле *cr_elementary_solid.h*).

В тестовом приложении *test.exe* построение элементарного тела выполняется командами меню «Создать→Тело→По точкам» и «Создать→Тело→На базе поверхности→По элементарной поверхности».

3.1.2 Построение тела выдавливания

Функция

```
MbSweptData ExtrusionSolid( const MbSweptData &           sweptData,
                            const MbVector3D &            direction,
                            const MbSolid *               solid1,
                            const MbSolid *               solid2,
                                  bool                    checkIntersection,
                            const ExtrusionValues &       params,
                            const MbSNameMaker &          operNames,
                                  PArray<MbSNameMaker> &  contoursNames,
                                  MbSolid *&              result );
```

выполняет построение тела выдавливания.

Входными параметрами функции являются:

- **sweptData** – параметр, содержащий данные об образующей кривой;
- **direction** – направление выдавливания;
- **solid1** – используется при выборе опции MbSweptWay::sw_shell, для выдавливания до ближайших граней заданного тела в прямом направлении;
- **solid2** – используется при выборе опции MbSweptWay::sw_shell, для выдавливания до ближайших граней заданного тела в обратном направлении;
- **checkIntersection** – флаг для объединения тел **solid1** и **solid2** с проверкой пересечения;
- **params** – объект класса ExtrusionValues, который содержит информацию о том, как задается величина выдавливания в каждую из сторон (на расстояние, до поверхности, до тела);
- **operNames** – именователь операции;
- **contoursNames** – именователи сегментов образующего контура.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Тело выдавливания относится к разновидности тел движения, которые получают путем движения образующей кривой вдоль направляющей кривой. Направляющей кривой тела выдавливания служит отрезок прямой. Тело выдавливания строится путем движения одной или нескольких кривых вдоль отрезка, направление которого задает вектор **direction**.

Параметр **sweptData** содержит информацию об образующих кривых. Классы MbSweptData, ExtrusionValues описаны в файле *op_swept_parameter.h*.

Образующая кривая операции выдавливания, вращения или кинематической операции может включать в себя набор двумерных контуров, набор трехмерных контуров, проволочный каркас, тело.

Двумерные контуры на поверхности могут пересекаться. Замкнутые пересекающиеся контуры объединяются с помощью булевой операции над регионами (рис. 1):

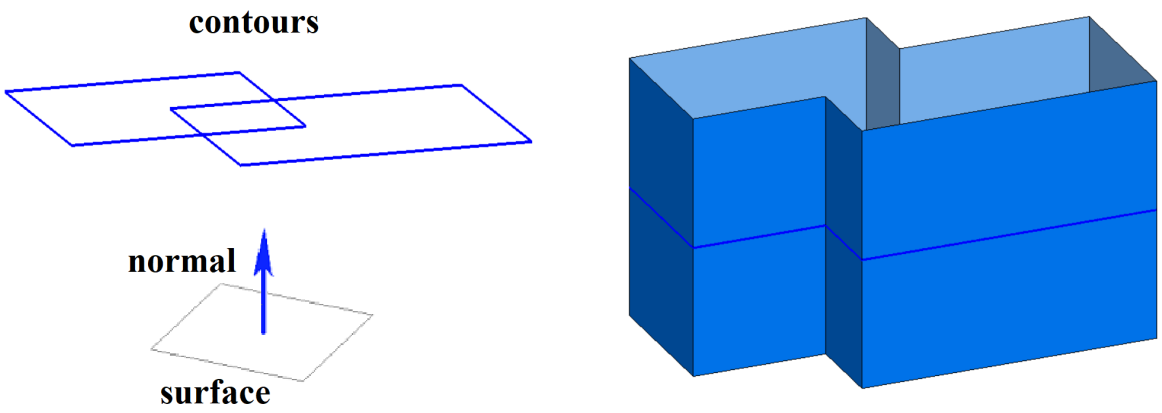


Рис. 1

В случае построения незамкнутой оболочки, если в число пересекающихся контуров входит разомкнутый контур, то будет построена оболочка с пересечением частей и функция вернет ошибку.

В случае построения замкнутой оболочки, пересечения оболочек, полученных по разомкнутым контурам, обрабатываются булевой операцией над оболочками.

Для набора двумерных контуров на поверхности существуют следующие ограничения:

- может быть один или несколько контуров;
- если контуров несколько, они должны быть либо все замкнуты, либо все разомкнуты;
- если контуры замкнуты, они могут быть вложенными друг в друга, уровень вложенности не ограничивается.

Для двумерных контуров на неплоской поверхности есть дополнительное ограничение: все контуры должны быть замкнуты.

Построение операции по двумерным контурам на неплоской поверхности рассчитано на указание пользователем грани тела в качестве образующей. В этом случае данные для образующей можно получить с помощью метода грани MbFace::GetSurfaceCurvesData(). Этот метод возвращает копию базовой поверхности грани и набор двумерных контуров, созданных по циклам ребер.

Ограничение для трехмерных контуров – создание замкнутого тела путем выдавливания трехмерного контура возможно только для варианта с тонкой стенкой.

При указании тела и поверхности одновременно предполагается, что выполняется кинематическая операция над телом вдоль кривой на этой поверхности, причем движение согласовано с нормалью.

Построение тела может выполняться в прямом направлении относительно вектора **direction**, в обратном направлении относительно вектора **direction** и в обоих направлениях. Параметры построения в каждом направлении задают объекты MbSweptSide.

Параметр **params** содержит информацию о способе выдавливания в прямом направлении MbSweptSide **side1** и информацию о способе выдавливания в обратном направлении MbSweptSide **side2**. Выдавливание в каждую сторону может выполнятся тремя способами:

1. При **way** = *sw_scalarValue* выдавливание выполняется на длину *scalarValue* в направлении **side1** или **side2**.
2. При **way** = *sw_shell* выдавливание выполняется до ближайшего объекта **solid1** или **solid2**.
3. При **way** = *sw_surface* выдавливание выполняется до поверхности **side1**.*surface* или **side2**.*surface*, если **side1**.*distance* равно нулю или **side2**.*distance* равно нулю, соответственно.

При **way** = *sw_surface* и **distance** не равным нулю выдавливание выполняется до эквидистантной поверхности к **side1**.*surface* или до эквидистантной поверхности к **side2**.*surface*, соответственно.

Если **side1**.*rake* не равно нулю или **side2**.*rake* не равно нулю, то выдавливание выполняется с уклоном **side1**.*rake* или **side2**.*rake* в соответствующую сторону.

Параметр **params**.*thickness1* задает отступ наружу от образующей кривой, а параметр **params**.*thickness2* задает отступ внутрь от образующей кривой.

Параметр **params**.*shellClosed* управляет замкнутостью построенного тела.

Параметр **params**.*checkSelfInt* сообщает о необходимости проверки результата построения на самопересечение. По умолчанию **params**.*checkSelfInt* = *false*, проверка не выполняется.

Управляющие флаги **mergeFlags** управляют слиянием подобных граней или ребер (*mergeFaces* = *true* – сливать подобные грани; *mergeEdges* = *true* – сливать подобные ребра).

На рис. 2 приведены данные, используемые при построении, и схема наследования параметров построения тела выдавливания ExtrusionValues & **params**.

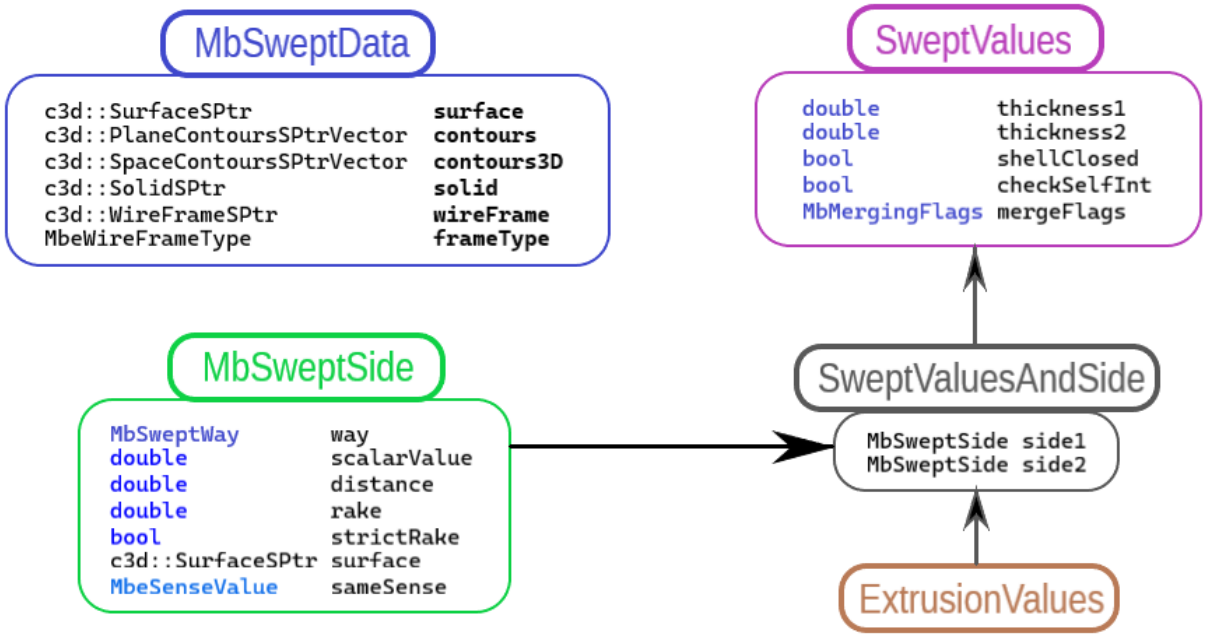


Рис. 2

Параметр **operNames** обеспечивает именование граней, ребер и вершин построенного

тела.

На рис. 3 приведен двумерный контур **contour** и плоская поверхность **surface** (MbPlane):

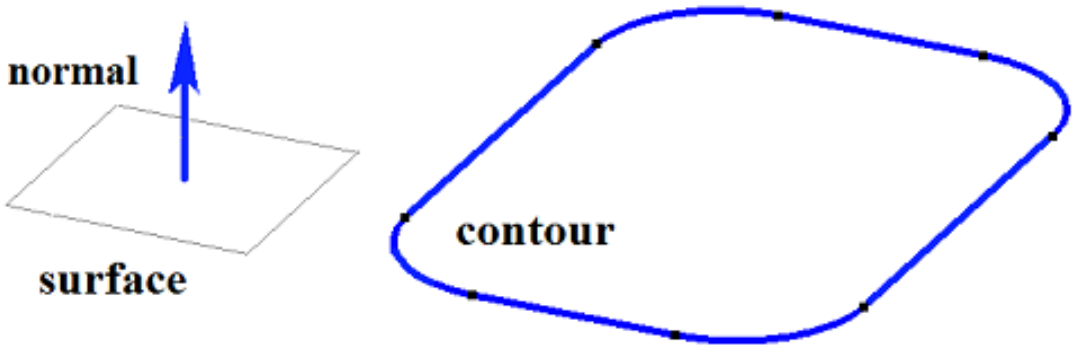


Рис. 3

На рис. 4 приведено замкнутое тело, построенное выдавливанием по заданным параметрам контура, показанного на рис. 3. Каждому сегменту контура соответствует грань тела, имя которой взято от соответствующего элемента генератора имен **contoursNames**[0].

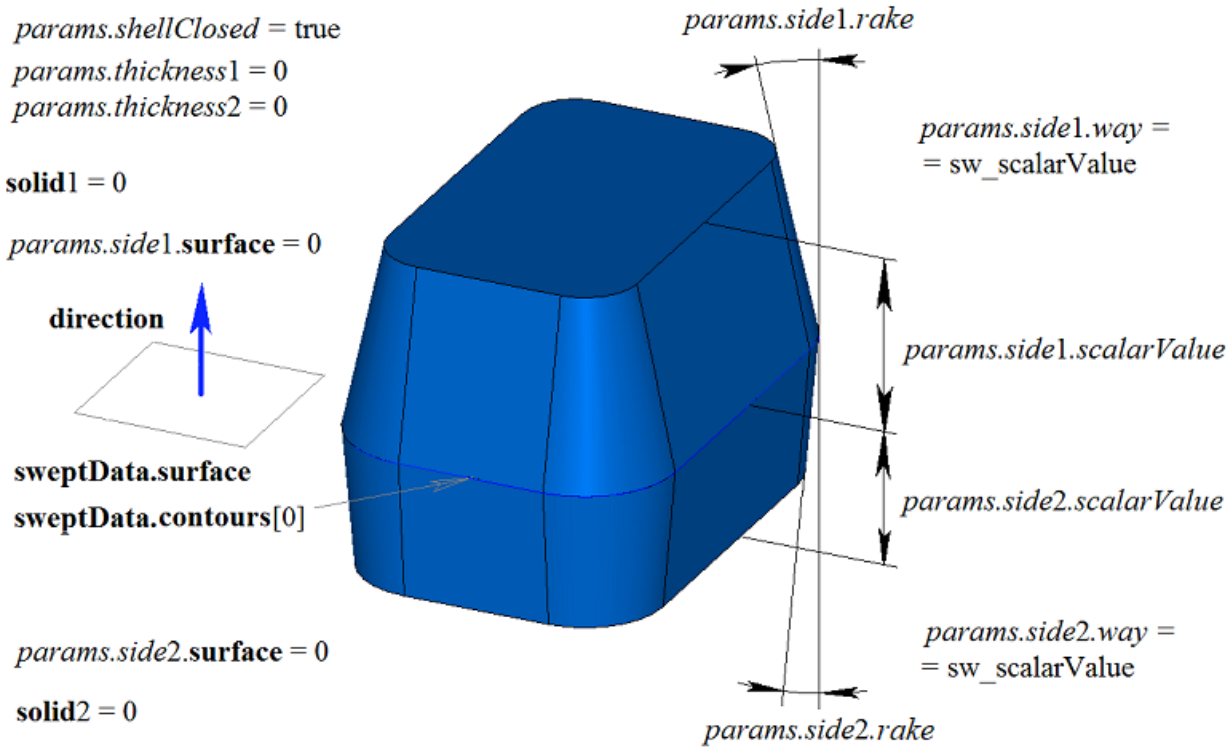


Рис. 4

На рис. 5 приведено замкнутое тонкостенное тело, построенное выдавливанием по за-

данным параметрам контура, показанного на рис. 3:

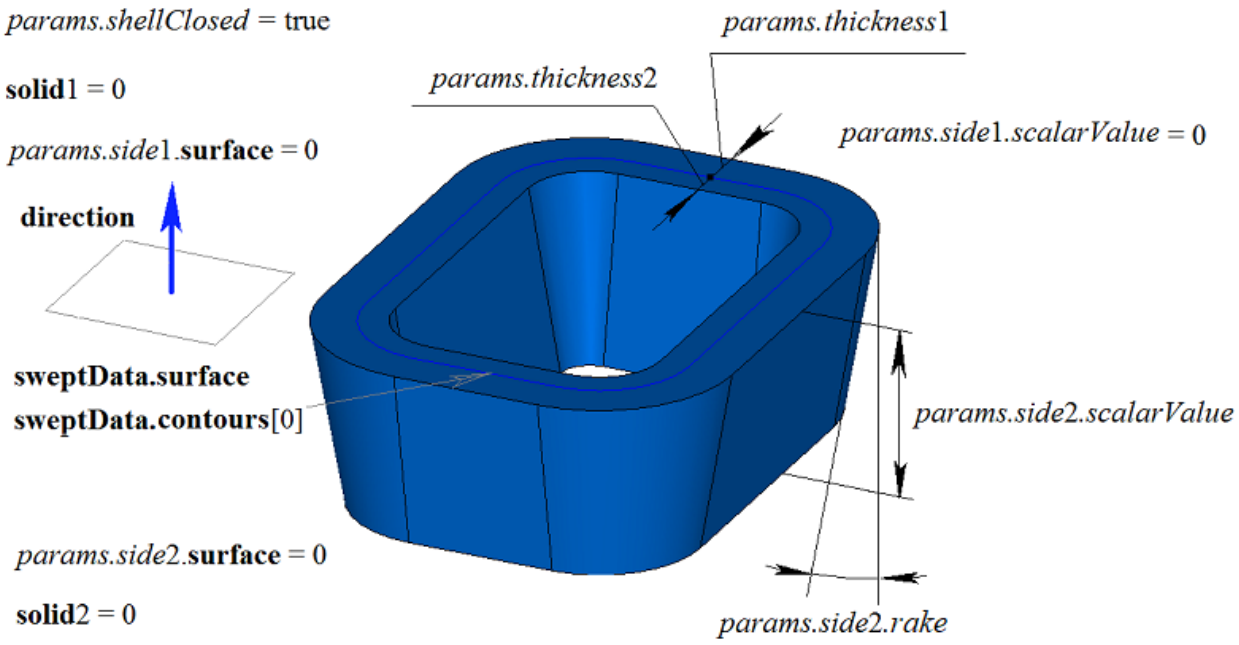


Рис. 5

На рис. 6 приведено незамкнутое тело, построенное выдавливанием по заданным параметрам контура, показанного на рис. 3. Параметры построения тела, показанного на рис. 4, отличаются от параметров построения тела, показанного на рис. 6, только величиной **params**.*shellClosed*:

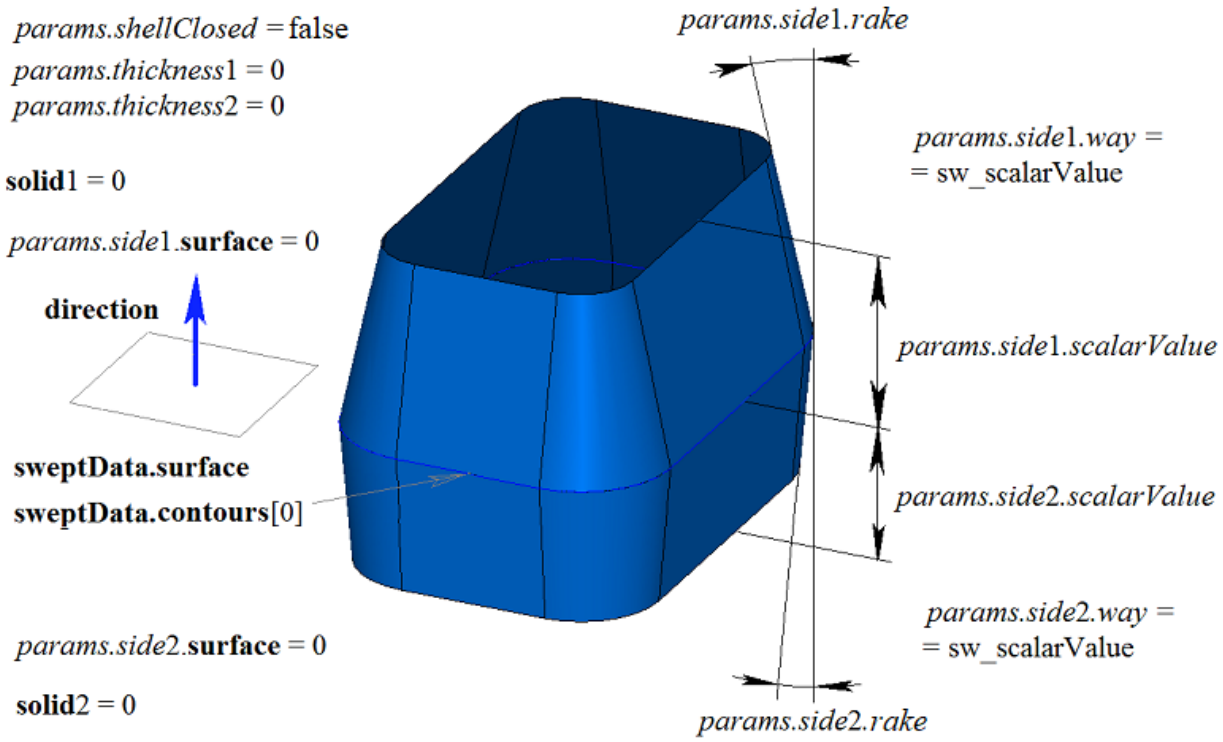


Рис. 6

На рис. 7 приведен двумерный контур **contour**, плоская поверхность **surface** (MbPlane) и два тела **solid1** и **solid2**, которые будут использоваться при построении тела выдавливания. Для построения необходимо, чтобы тела **solid1** и **solid2** полностью перекрывали путь движения контура в соответствующем направлении. При этом следует учитывать параметры **params**.*side1.rake*, **params**.*side2.rake*, **params**.*thickness1*, **params**.*thickness2*:

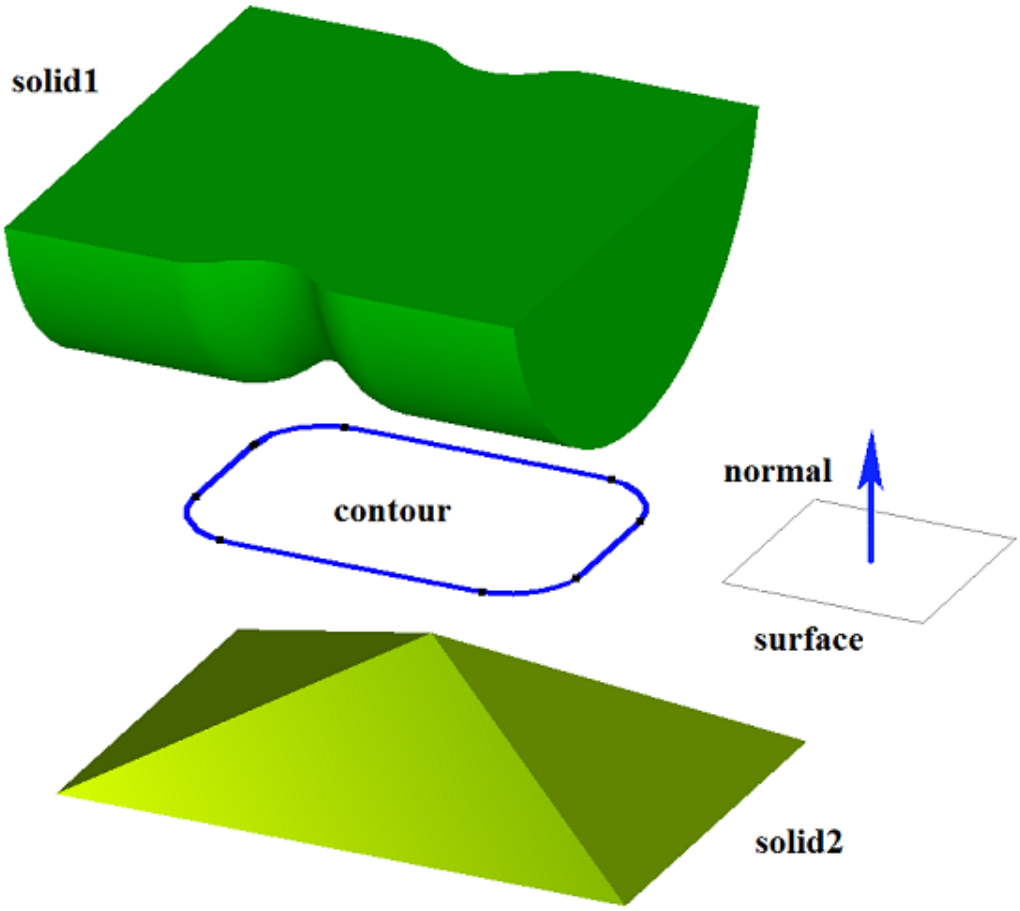


Рис. 7

Такое построение выполняется выдавливанием контура на длину, превосходящую максимальное расстояние до заданного тела, с последующим вычитанием заданного тела из построенного тела.

На рис. 8 приведено тело, построенное выдавливанием контура, показанного на рис. 7, с опциями «До ближайших объектов», в качестве которых заданы тела **solid1** и **solid2**:

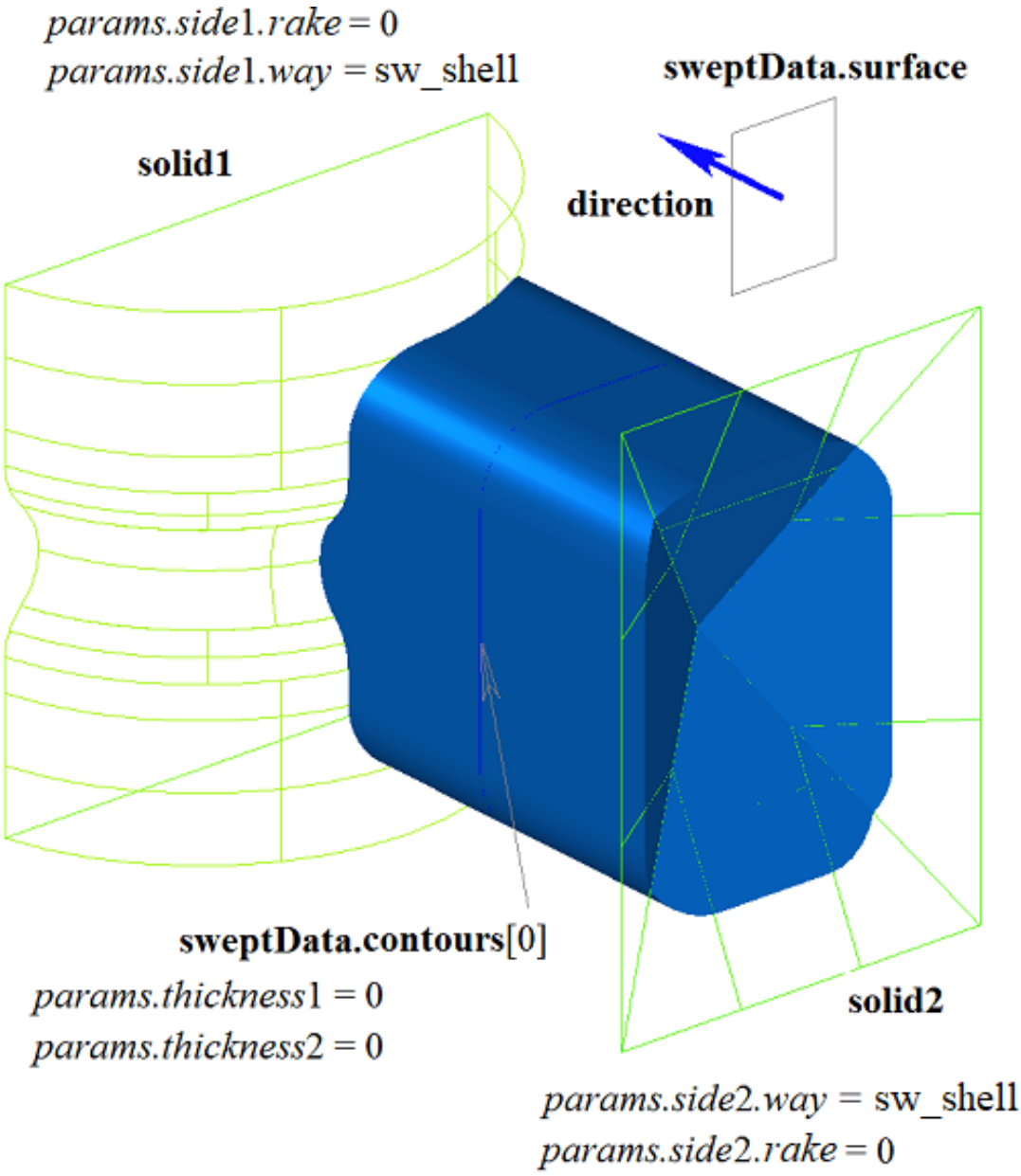


Рис. 8

На рис. 9 приведено тонкостенное тело с уклоном граней, построенное выдавливанием контура, показанного на рис. 7, с опциями «До ближайших объектов», в качестве которых заданы тела **solid1** и **solid2**:

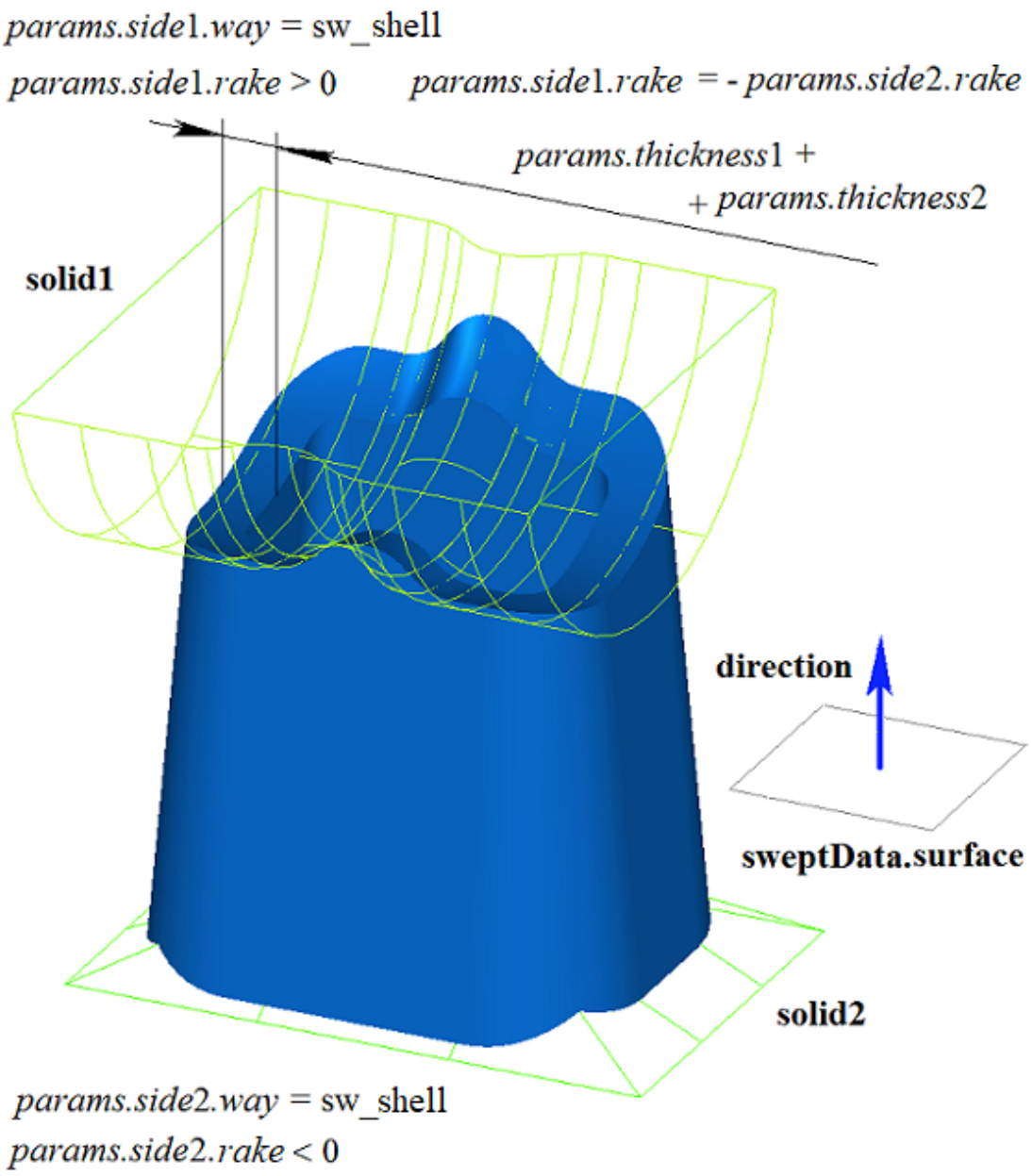


Рис. 9

На рис. 10 приведен двумерный контур **contour**, плоская поверхность **surface** (*MbPlane*) и две поверхности **surface1** и **surface2**, которые будут использоваться при построении тела выдавливания. Для построения необходимо, чтобы поверхности **surface1** и **surface2** полностью перекрывали путь движения контура в соответствующем направлении. При этом следует учитывать параметры **params**.*side1.rake*, **params**.*side2.rake*, **params**.*thickness1*, **params**.*thickness2*. Тело выдавливания обрезается заданными поверхностями или эквидистантными к ним поверхностями, если **params**.*side1.distance* или **params**.*side2.distance* не равны нулю.

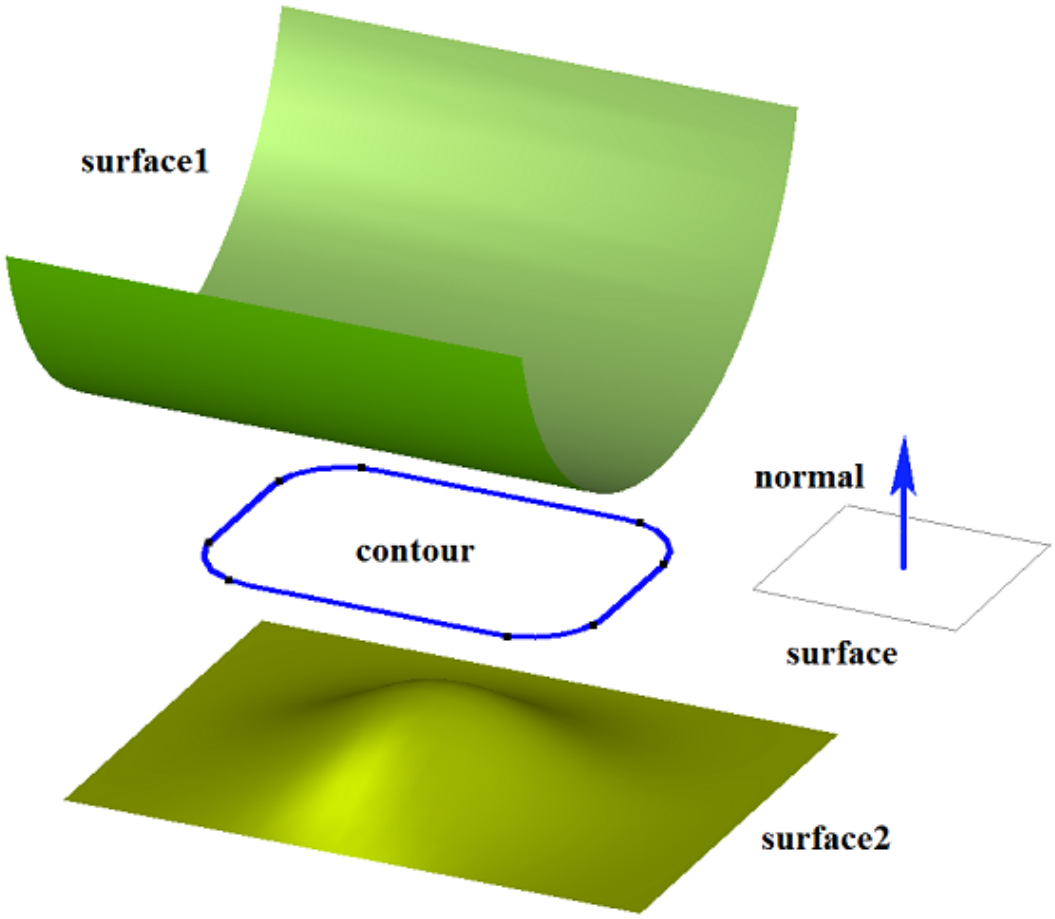


Рис. 10

На рис. 11 приведено тело, построенное выдавливанием контура, показанного на рис. 10, с опциями «До поверхности», в качестве которых заданы **surface1** и **surface2**.

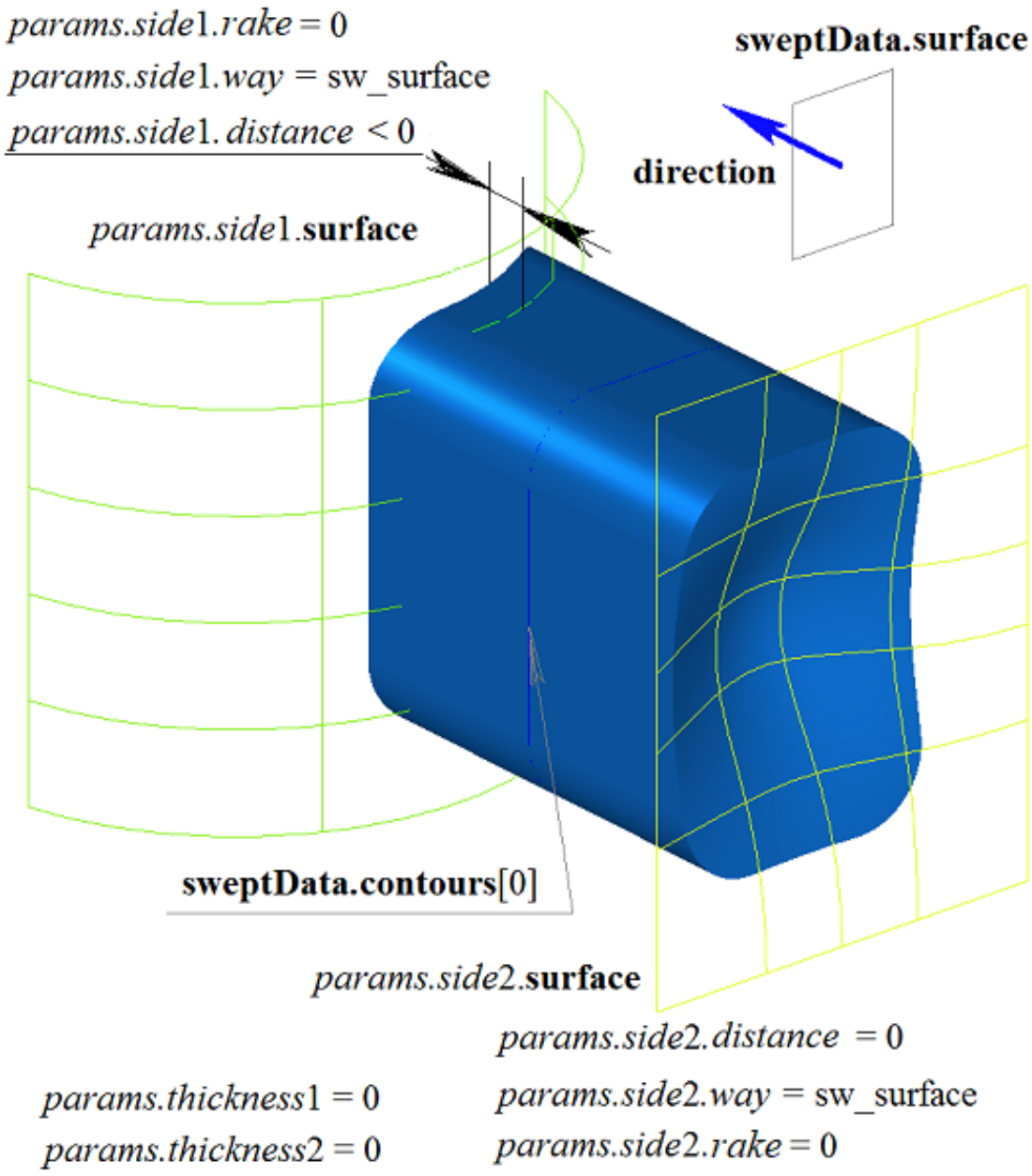


Рис. 11

На рис. 12 приведено тонкостенное тело с уклоном граней, построенное выдавливанием контура, показанного на рис. 10, с опциями «До поверхности», в качестве которых заданы **surface1** и **surface2**:

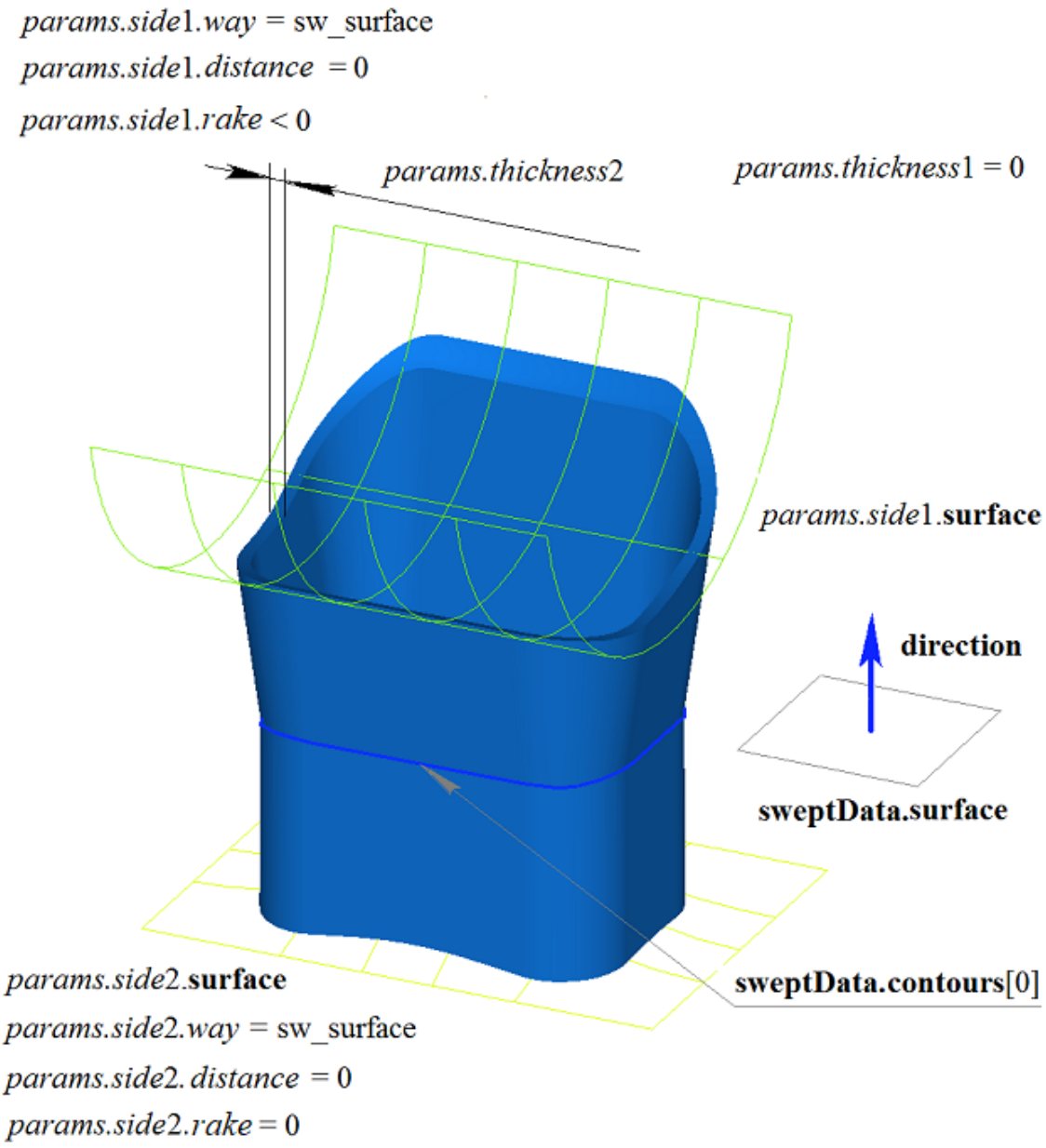


Рис. 12

Двумерный контур может располагаться на плоской или криволинейной поверхности. Например, тело можно построить выдавливанием контура на криволинейной поверхности, полученного от цикла одной из граней твердого тела, показанного на рис. 13:

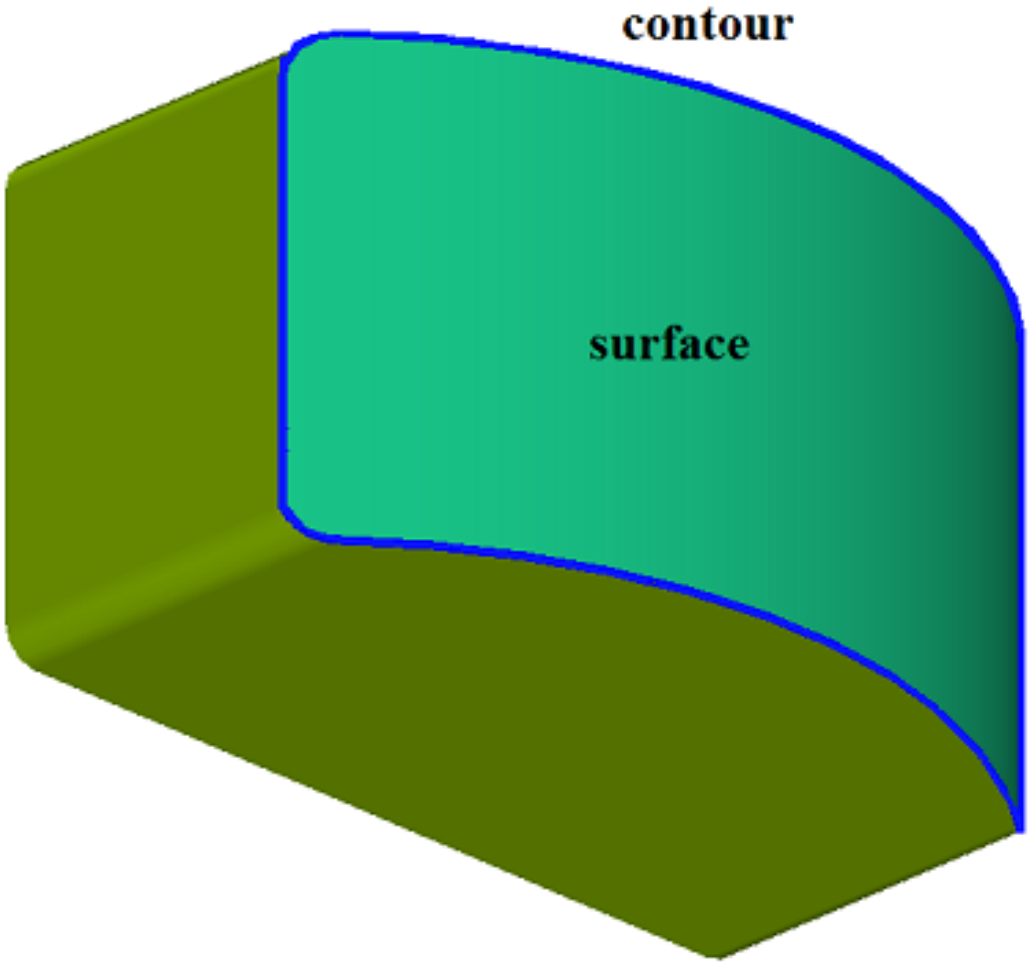


Рис. 13

На рис. 14 приведено тело, полученное выдавливанием контура на криволинейной поверхности, приведенной на рис. 13:

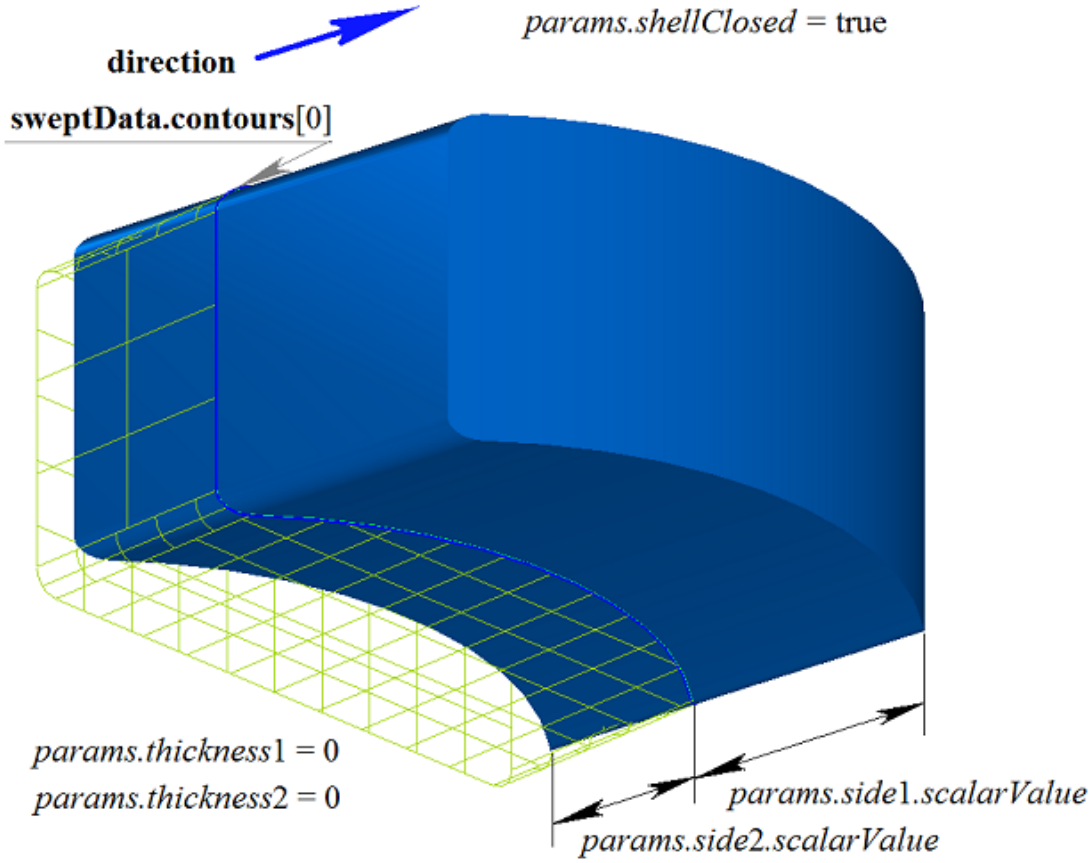


Рис. 14

На рис. 15 приведено тонкостенное тело, полученное выдавливанием контура на криволинейной поверхности, приведенной на рис. 13:

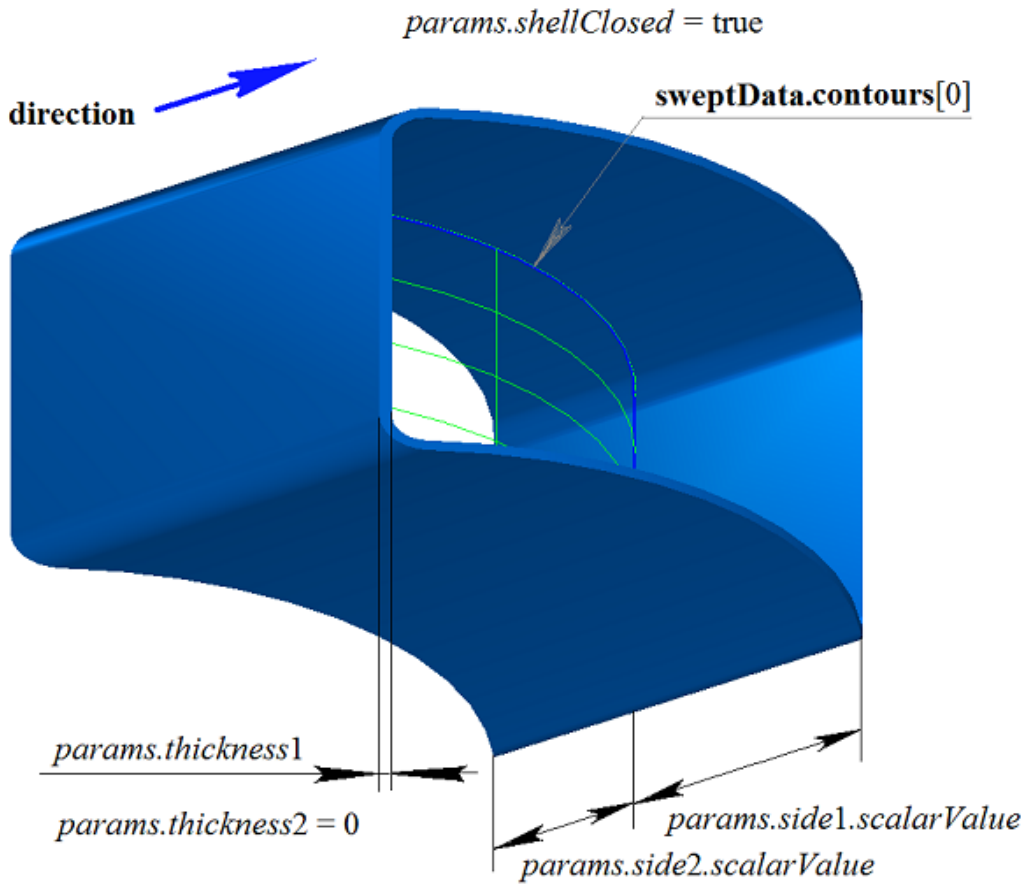


Рис. 15

На рис. 16 приведено незамкнутое тело, полученное выдавливанием контура на криволинейной поверхности, приведенной на рис. 13:

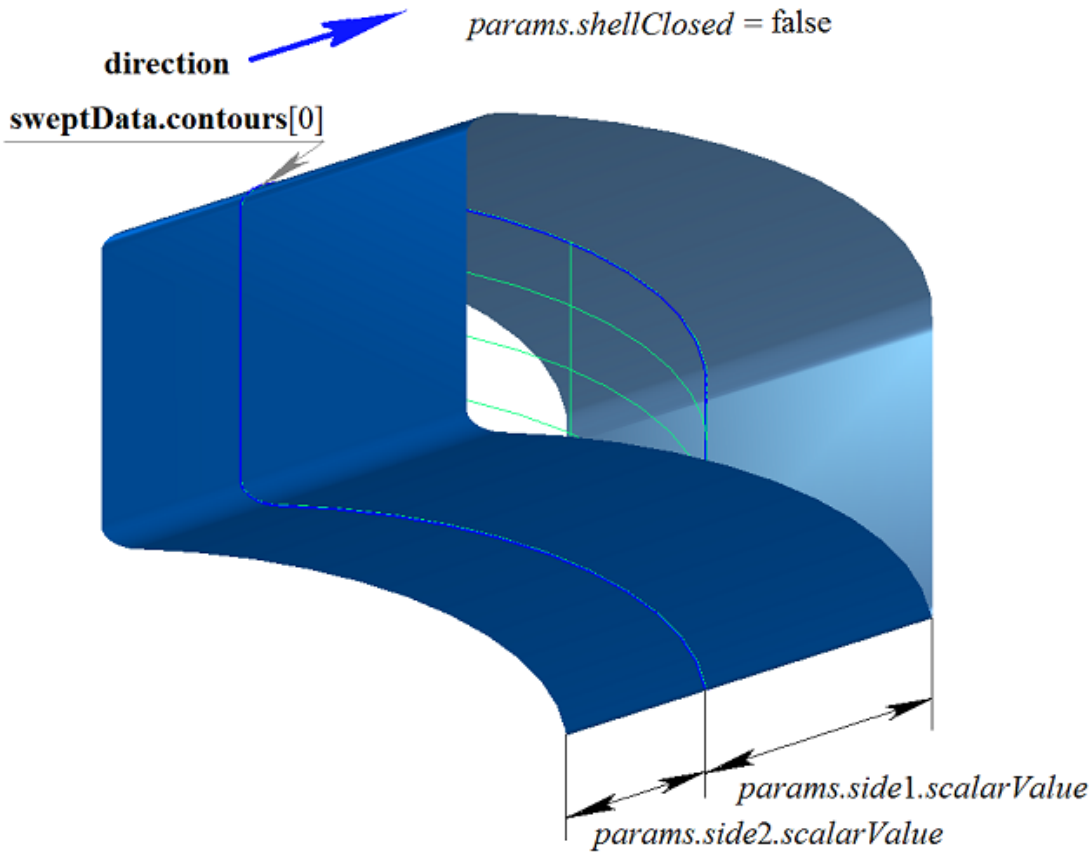


Рис. 16

Если на одной поверхности расположено множество не пересекающихся двумерных контуров, то рассматриваемая функция определяет внешние контуры и вложенные в них внутренние контуры, причем вложение может быть многократным. На рис. 17 приведено множество не пересекающихся двумерных контуров **contours** и плоская поверхность **surface** (MbPlane).

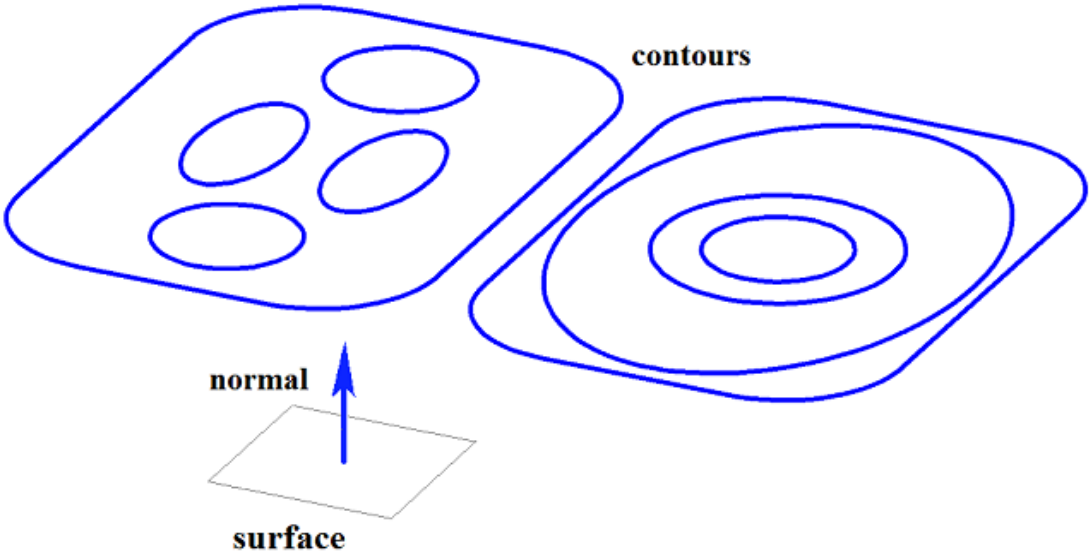


Рис. 17

На рис. 18 приведено замкнутое тело, состоящее из нескольких частей, построенное выдавливанием множества контуров, показанного на рис. 17:

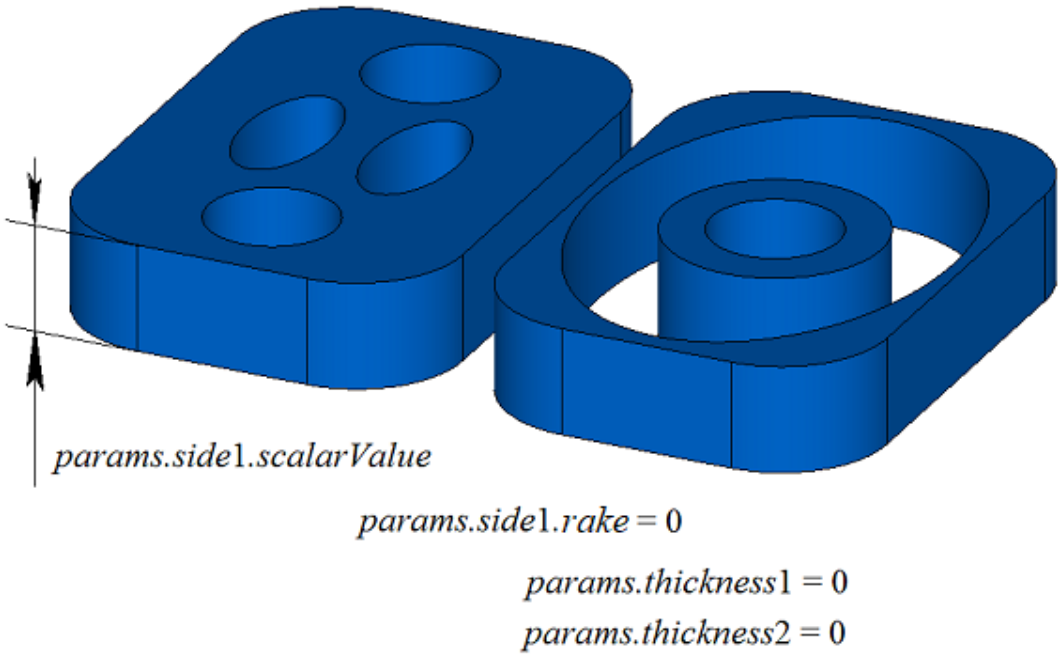


Рис. 18

На рис. 19 приведено замкнутое тело, состоящее из нескольких частей, построенное выдавливанием с уклоном множества контуров, показанного на рис. 17:

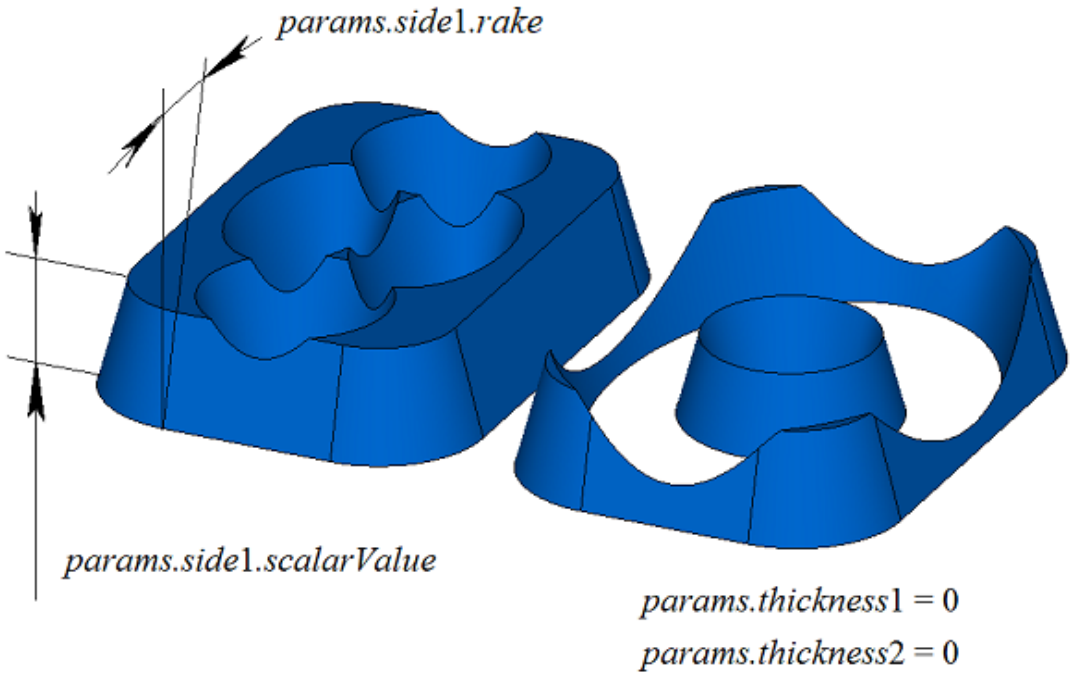


Рис. 19

На рис. 20 приведено замкнутое тонкостенное тело, состоящее из нескольких частей, построенное выдавливанием множества контуров, показанного на рис. 17:

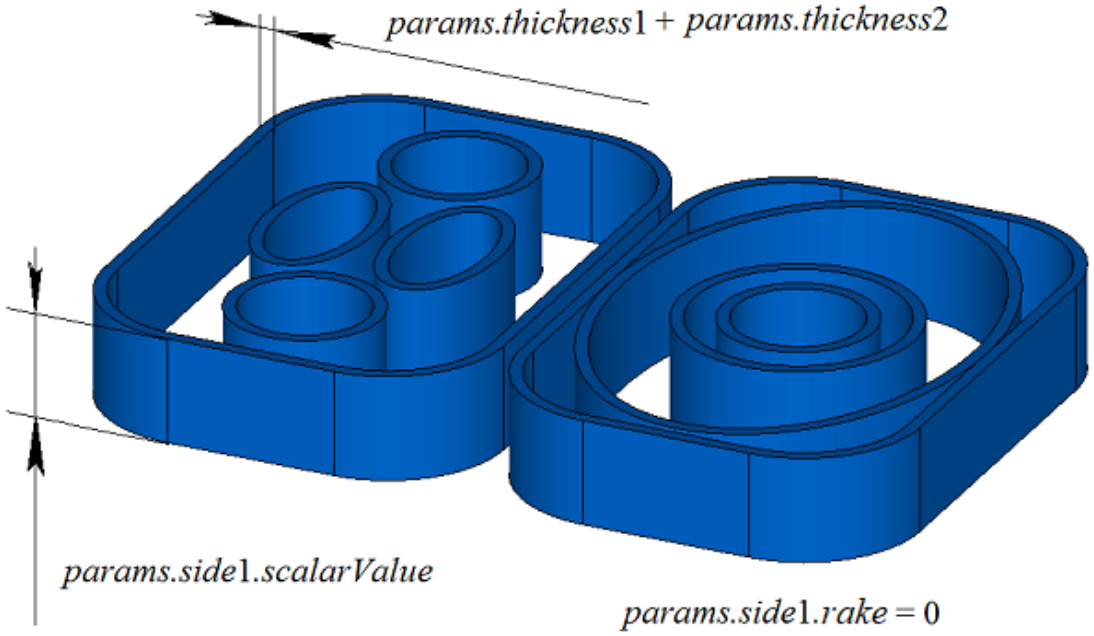


Рис. 20

На рис. 21 приведены два трехмерных контура:

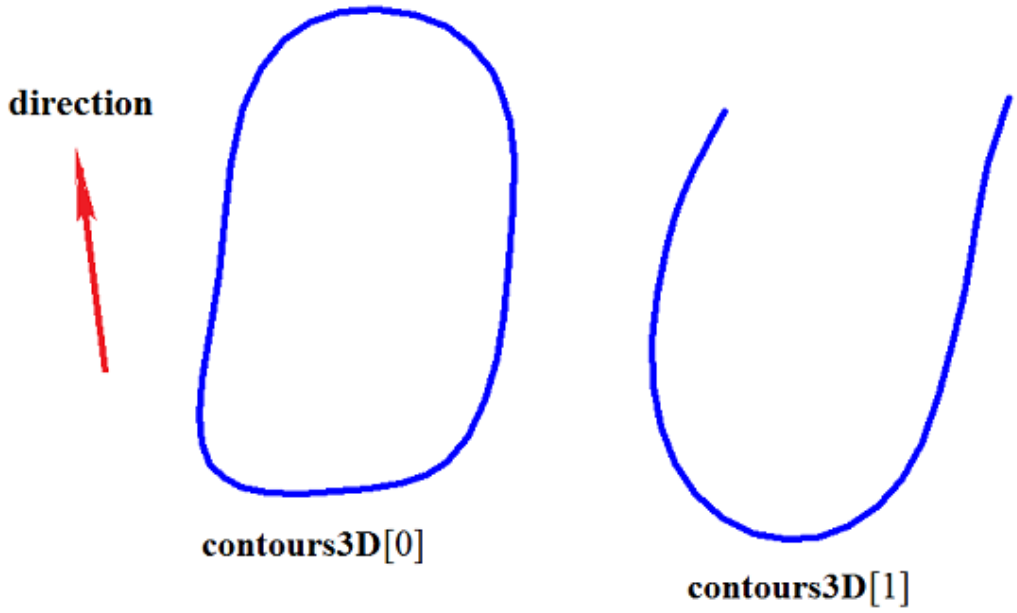


Рис. 21

На рис. 22 приведено двусвязное тонкостенное замкнутое тело, полученное выдавливанием трехмерных контуров, приведенных на рис. 21.

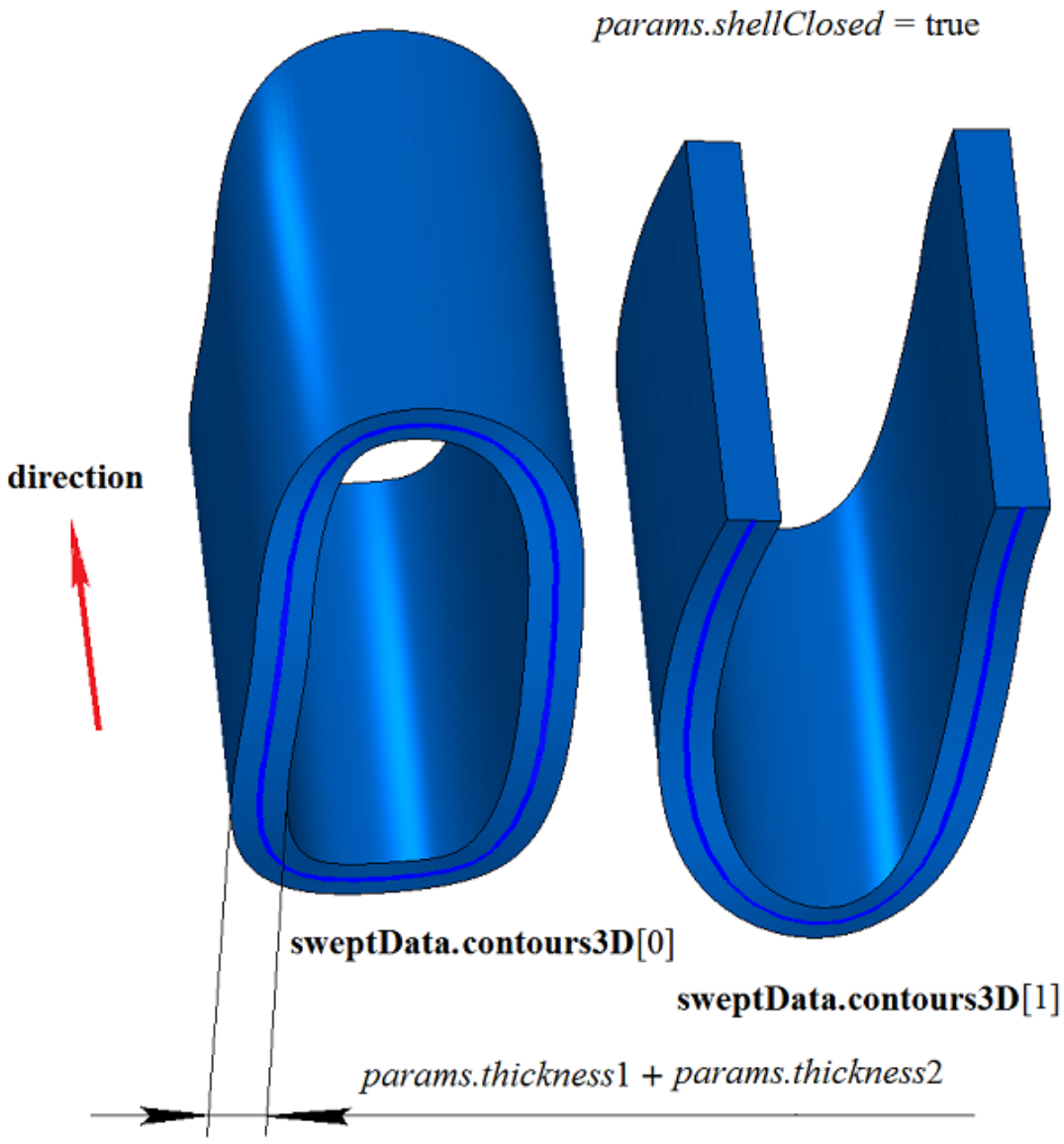


Рис. 22

На рис. 23 приведены два незамкнутых тела, полученных выдавливанием трехмерных контуров, приведенных на рис. 21. Построение выполнено для каждого контура отдельно:

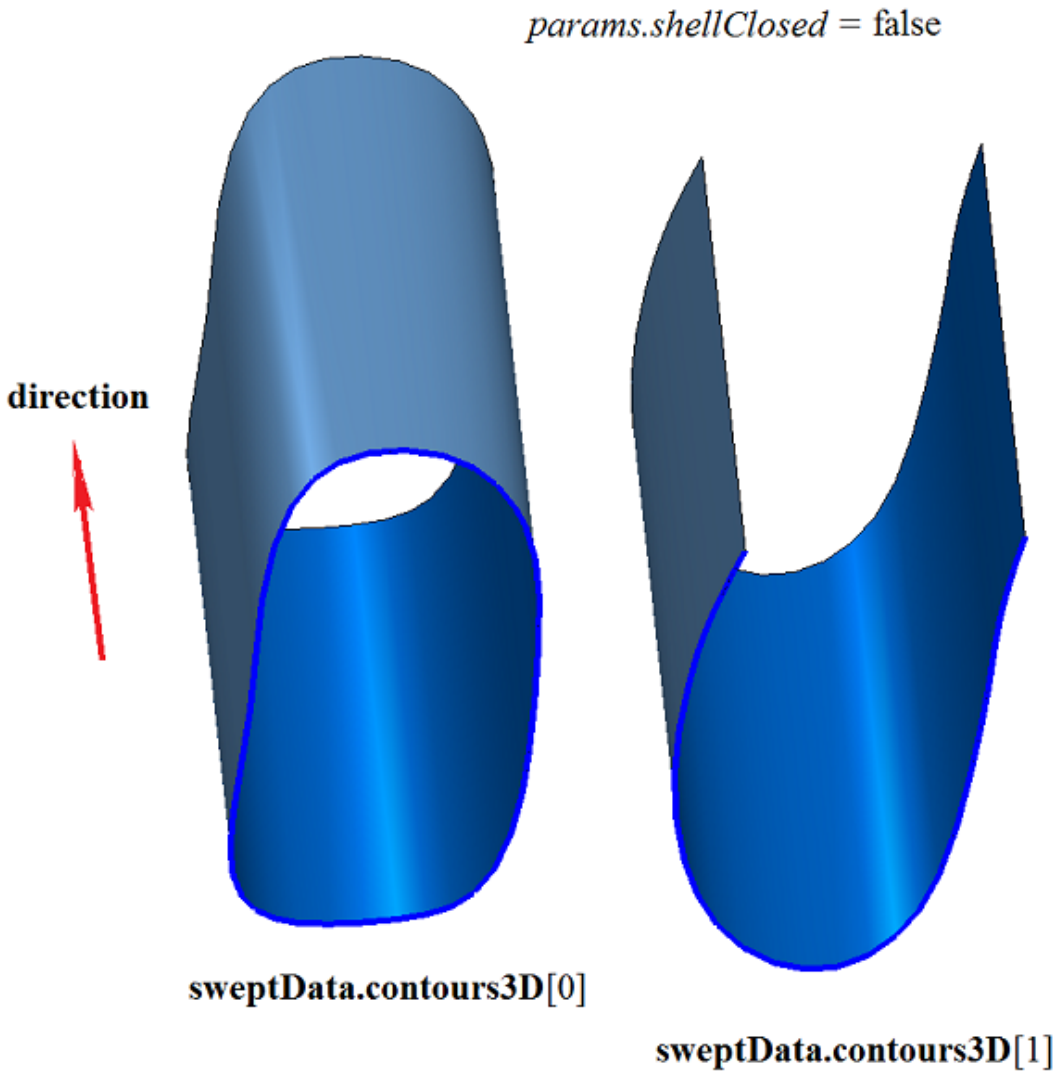


Рис. 23

Функция построения тела выдавливания ExtrusionSolid() добавляет в журнал построенного тела строитель MbCurveExtrusionSolid, который содержит все необходимые для построения тела данные. Строитель MbCurveExtrusionSolid объявлен в файле *cr_extrusion_solid.h*.

В тестовом приложении *test.exe* построение тела выдавливания выполняется командами меню «Создать->Тело->На базе кривых->Выдавливанием поверхностной кривой» и «Создать->Тело->На базе кривых->Выдавливанием трехмерной кривой».

3.1.3 Построение тела вращения

Функция

```
MbResultType RevolutionSolid( const MbSweptData &          sweptData,
                              const MbAxis3D &             axis,
                              const RevolutionValues &     params,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                                const MbSNameMaker &           operNames,
                                      RPArray<MbSNameMaker> & contoursNames,
                                      MbSolid *&              result );
```

выполняет построение тела вращения.

Входными параметрами функции являются:

- **sweptData** – данные об образующих кривых;
- **axis** – ось вращения;
- **params** – объект, содержащий параметры вращения;
- **operNames** – именователь операции;
- **contoursNames** – именователи сегментов образующего контура.

Выходным параметром функции является построенное тело (оболочка) **result**.

Функция возвращает код результата операции из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Тело вращения относится к разновидности тел движения, которые получаются путем движения образующей кривой вдоль направляющей кривой. Направляющей кривой тела вращения служит окружность или ее дуга. Тело вращения строится путем вращения одной или нескольких кривых вокруг оси **axis**.

Параметр **sweptData** содержит информацию об образующих кривых. Классы MbSweptData и RevolutionValues описаны в файле *op_swept_parameter.h*. Образующие кривые могут представлять собой двумерные контуры **contours** на поверхности **surface** или контуры в пространстве **contours3D**. В частном случае поверхность **surface** может представлять собой плоскость. Ориентация контуров **contours** может быть произвольной. Контуры **contours** могут быть вложены друг в друга.

Вращение кривых может выполняться в прямом направлении относительно оси **axis**, в обратном направлении относительно оси **axis** и в обоих направлениях. Вращение в прямом направлении выполняется против часовой стрелки при взгляде навстречу оси **axis**. Параметры построения в каждом направлении задают объекты MbSweptSide.

Параметр **params** содержит два объекта класса MbSweptSide – **side1** и **side2**, в которых находятся параметры вращения в прямом и обратном направлении.

Способ вращения задается в поле **way** объекта класса MbSweptSide:

- при **way** = *sw_scalarValue* вращение выполняется на угол **scalarValue** в направлении **side1** или **side2**. Угол вращения задается в радианах;
- при **way** = *sw_surface* вращение выполняется до поверхности **side1**.*surface* или **side2**.*surface* при **side1**.*distance* равном нулю или **side2**.*distance* равном нулю соответственно;
- при **way** = *sw_surface* и **distance** не равным нулю вращение выполняется до эквидистантной поверхности к **side1**.*surface* или до эквидистантной поверхности к **side2**.*surface* соответственно;
- при **way** = *sw_shell* вращение выполняется до ближайшего объекта (тела) в направлении **side1** или **side2**.

Параметры **params**.*thickness1* и **params**.*thickness2* определяют толщину стенки тонкостенного тела. Параметр **params**.*thickness1* задает отступ наружу от образующей кривой, а параметр **params**.*thickness2* задает отступ внутрь от образующей кривой.

Параметр **params**.*shellClosed* управляет замкнутостью построенного тела.

Параметр **params**.*checkSelfInt* сообщает о необходимости проверки результата построения на самопересечение. По умолчанию **params**.*checkSelfInt* = *false*, проверка не выполняется.

Параметр **params**.*shape* управляет формой построенного тела. При **params**.*shape* = 1 построенное тело имеет топологию тора, при **params**.*shape* = 0 построенное тело имеет топологию сферы. Если образующая — незамкнутая плоская кривая и ось вращения лежит в плоскости кривой, то возможно построение тела вращения с топологией типа сферы. В этом случае образующая достраивается до оси вращения.

На рис. 1 приведены данные, используемые при построении, и схема наследования параметров построения тела вращения RevolutionValues & **params**.

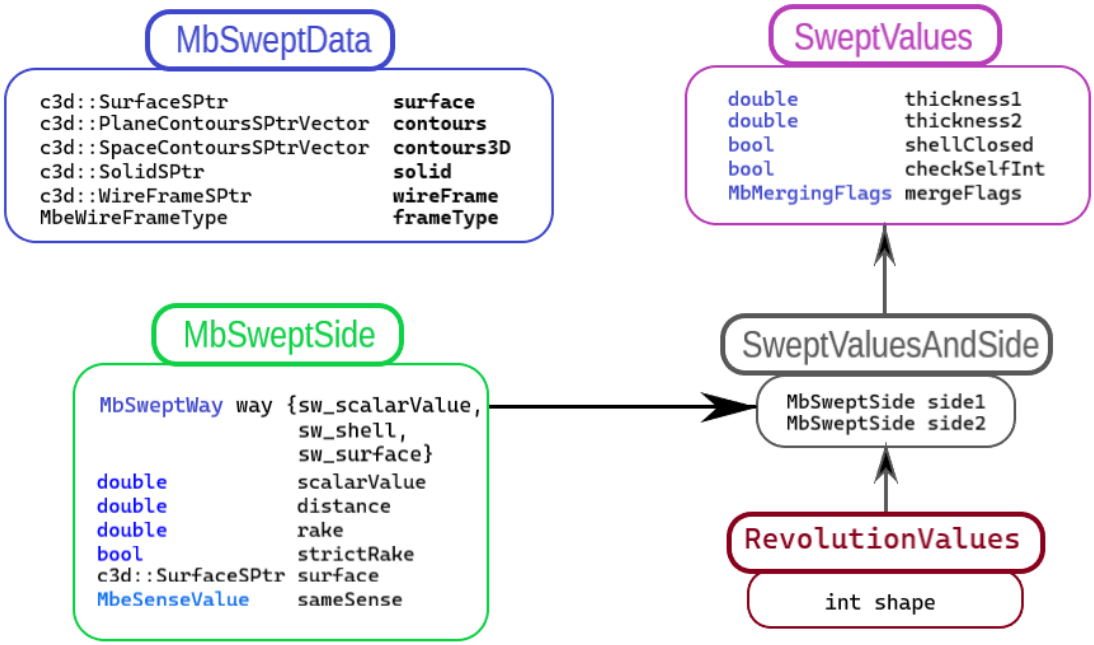


Рис. 1

Параметры **operNames** и **contoursNames** обеспечивают именование граней, ребер и вершин построенного тела.

На рис. 2 приведен двумерный контур **contour**, плоская поверхность **surface** (MbPlane) и ось вращения **axis**.

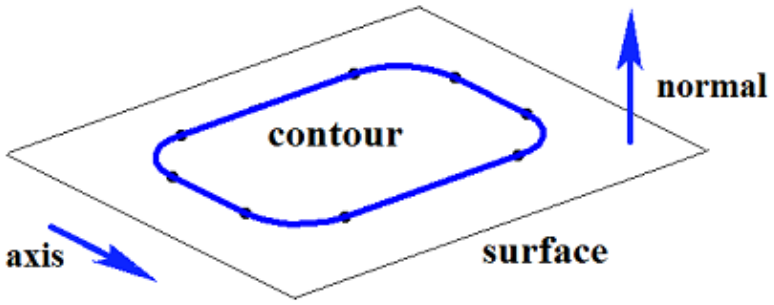


Рис. 2

На рис. 3 приведено замкнутое тело, построенное вращением по заданным параметрам контура, показанного на рис. 2.

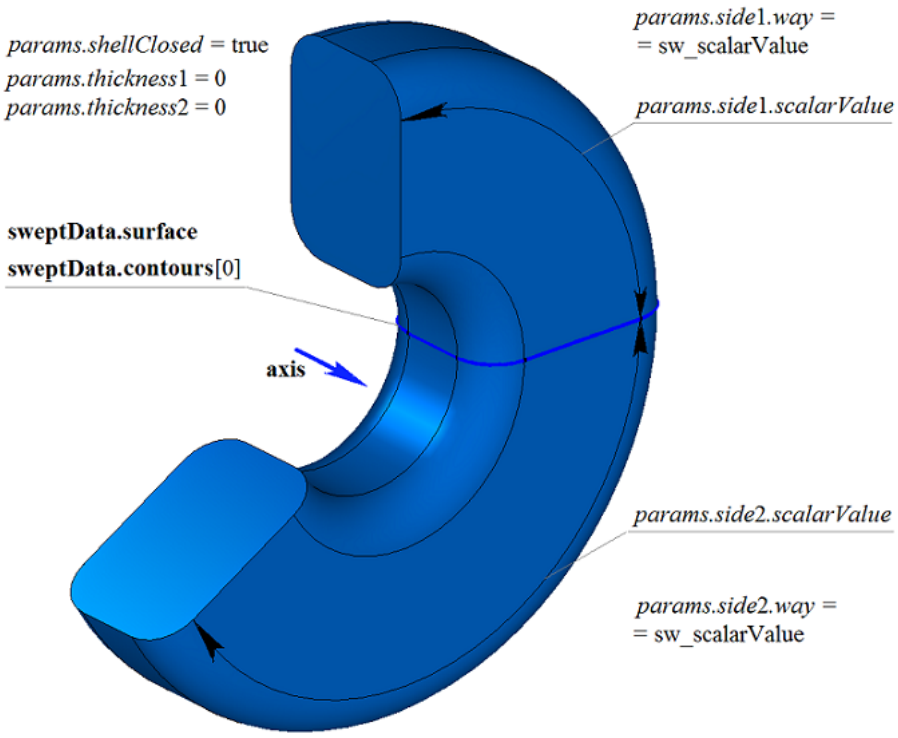


Рис. 3

На рис. 4 приведено замкнутое тонкостенное тело, построенное вращением по заданным параметрам контура, показанного на рис. 2.

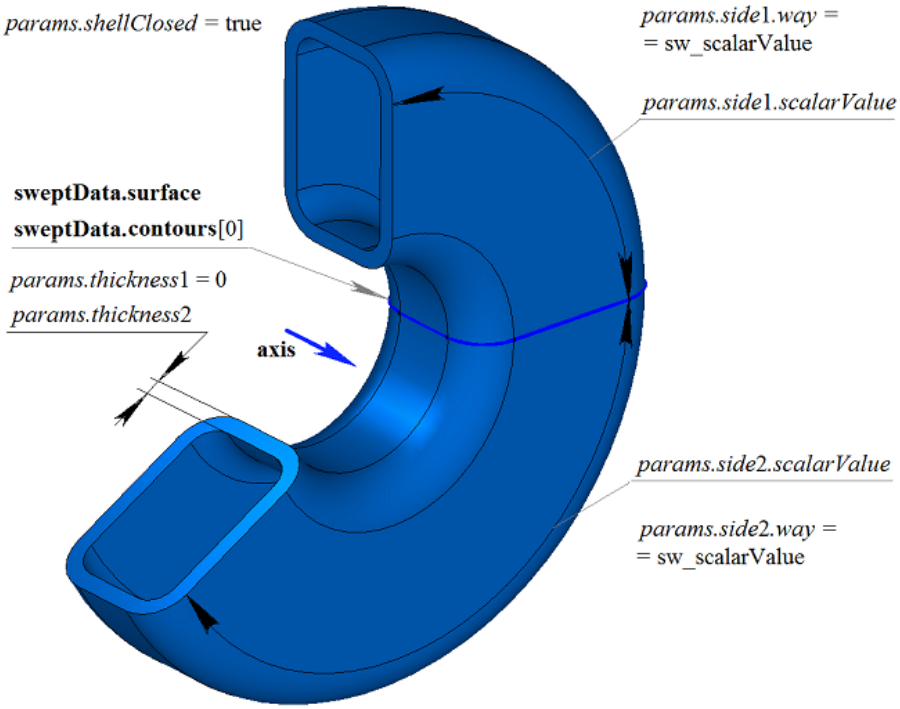


Рис. 4

На рис. 5 приведено незамкнутое тело, построенное вращением по заданным параметрам контура, показанного на рис. 2. Параметры построения тела, показанного на рис. 3, отличаются от параметров построения тела, показанного на рис. 5, только величиной **params**.*shellClosed*.

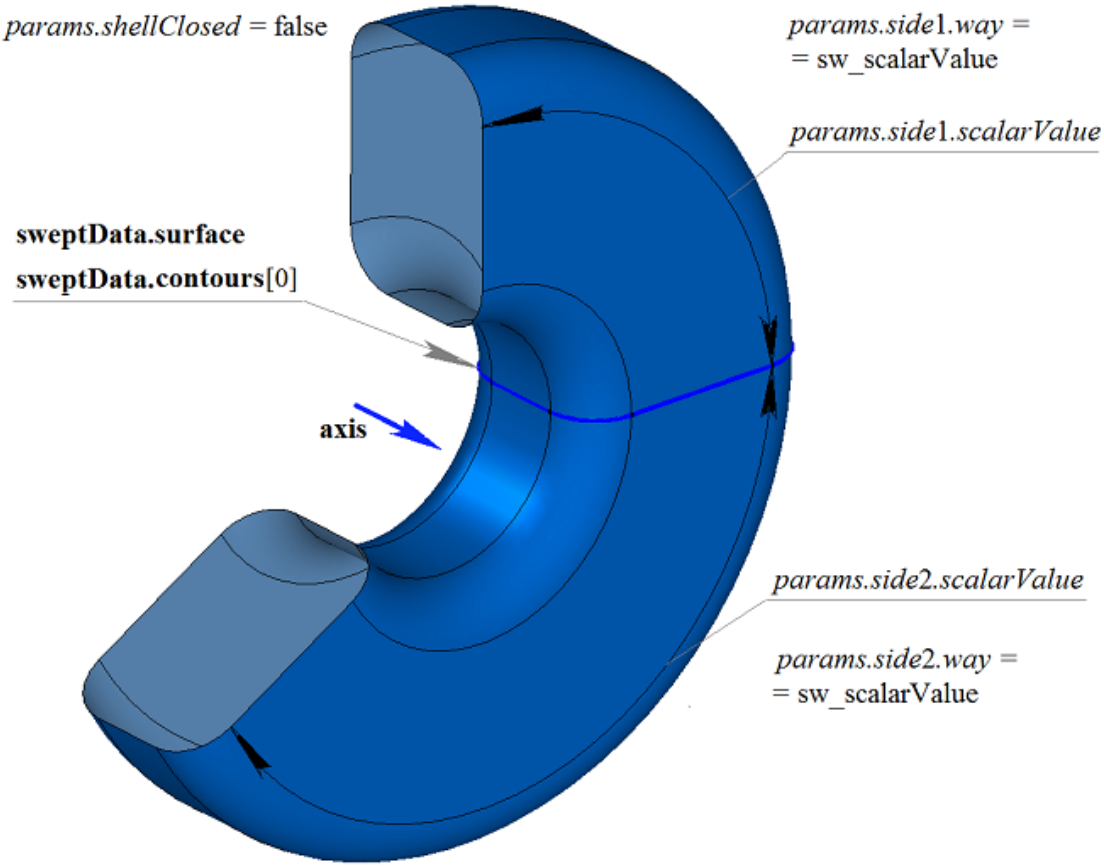


Рис. 5

На рис. 6 приведен двумерный контур **contour**, плоская поверхность **surface** (MbPlane) и две поверхности **surface1** и **surface2**, которые будут использоваться при построении тела вращения. Для построения необходимо, чтобы поверхности **surface1** и **surface2** полностью перекрывали путь движения контура в соответствующем направлении. При этом следует учитывать параметры **params**.*thickness1*, **params**.*thickness2*. Тело вращения обрезается заданными поверхностями (если **params**.*side1.distance* или **params**.*side2.distance* равны нулю) или эквидистантными к ним поверхностями (если **params**.*side1.distance* или **params**.*side2.distance* не равны нулю).

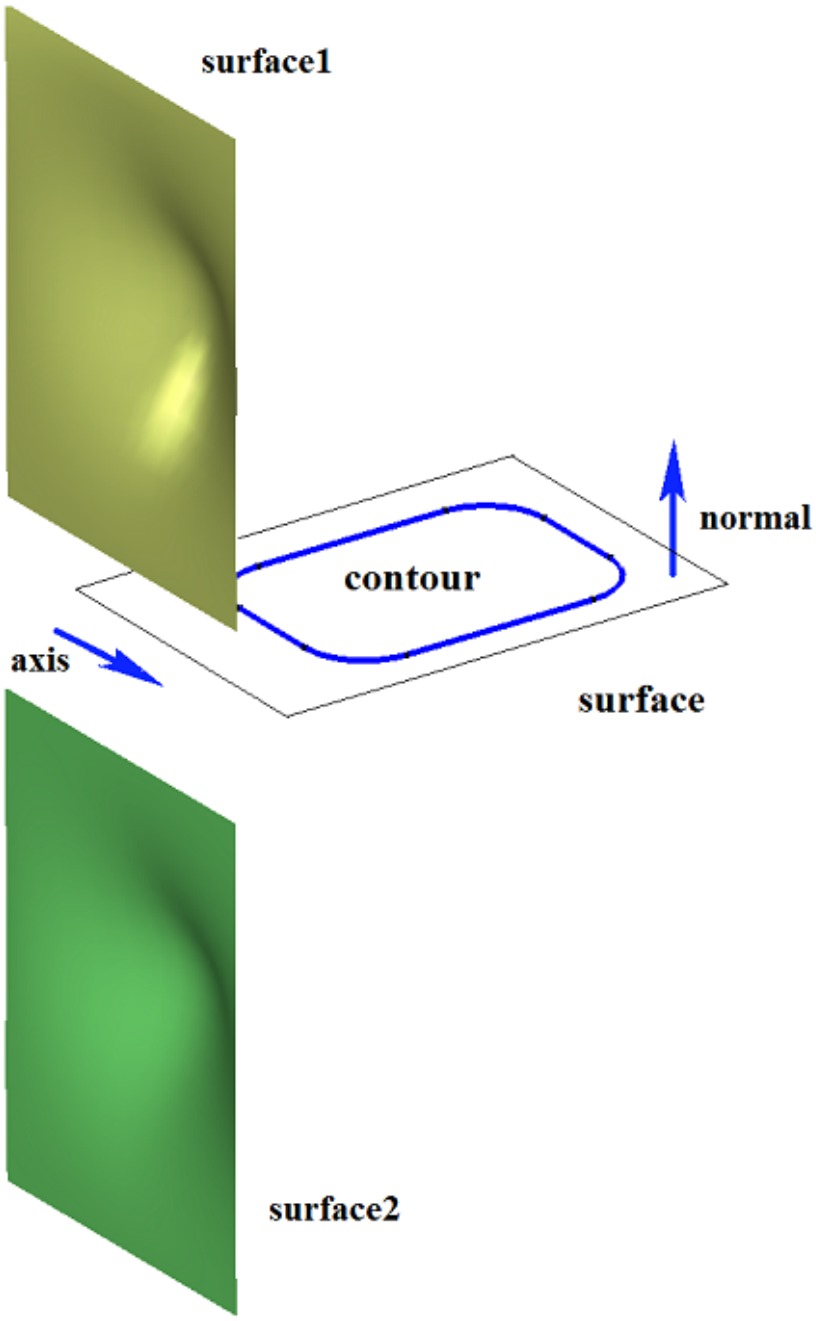


Рис. 6

На рис. 7 приведено тело, построенное вращением контура, показанного на рис. 6, с опциями «До поверхности» (**params**.*side1(2).way* = *sw_surface*), в качестве которых заданы **surface1** и **surface2**.

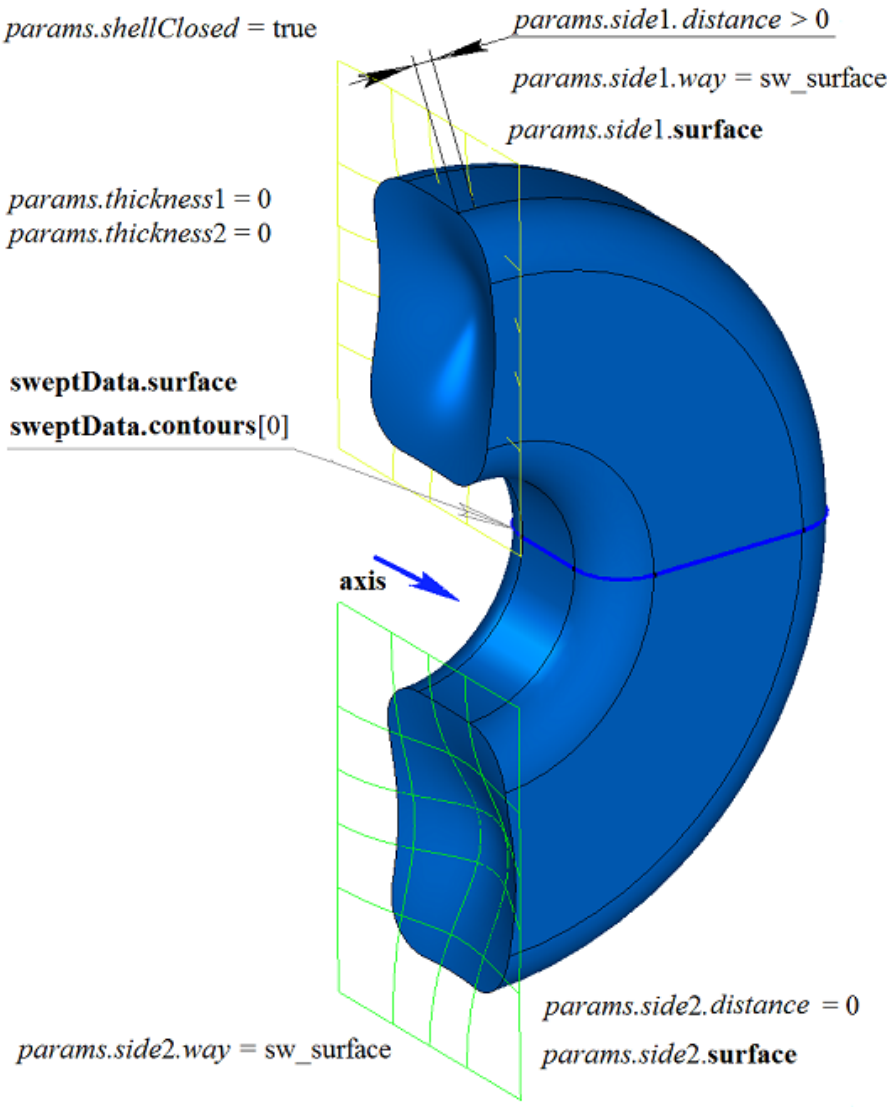


Рис. 7

На рис. 8 приведено тонкостенное тело, построенное вращением контура, показанного на рис. 6, с опциями «До поверхности», в качестве которых заданы **surface1** и **surface2**. Тело вращения обрезается заданными поверхностями.

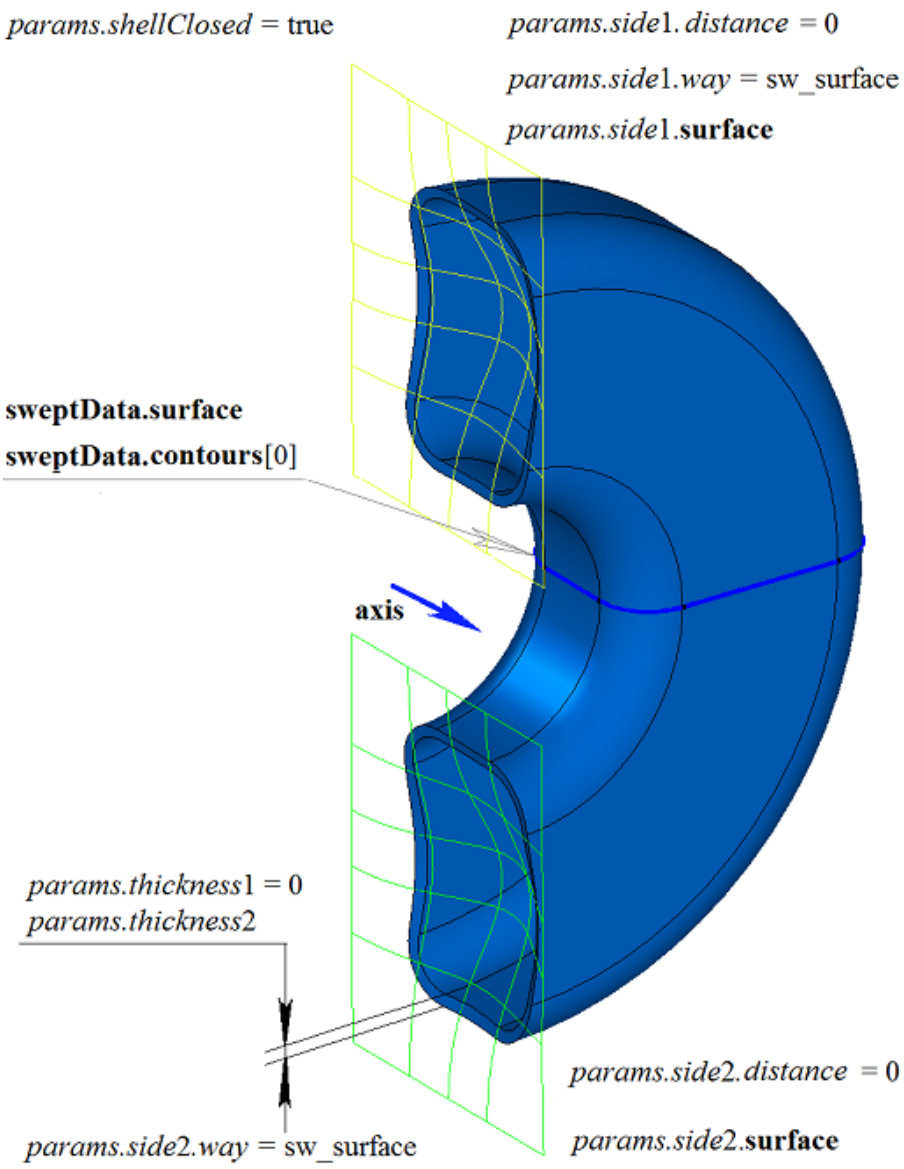


Рис. 8

Двумерный контур может располагаться на плоской или криволинейной поверхности. Например, тело можно построить вращением контура на криволинейной поверхности, полученного от цикла одной из граней твердого тела, показанного на рис. 9.

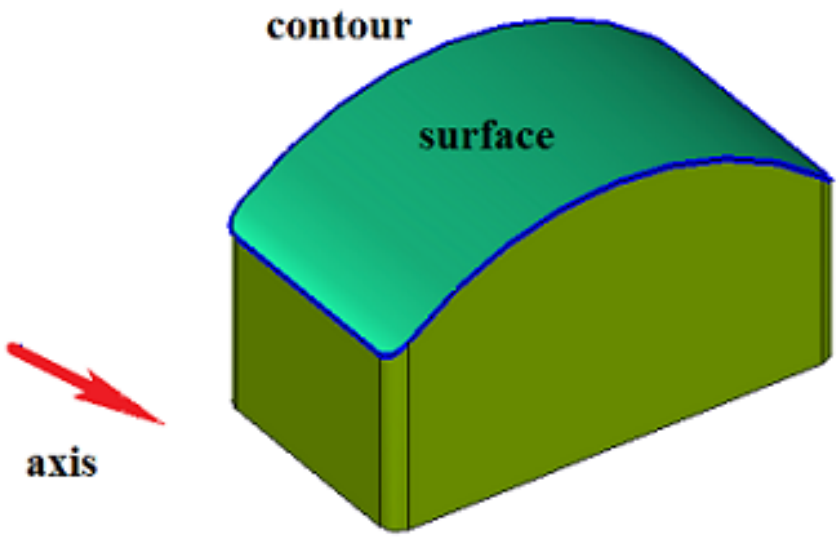


Рис. 9

На рис. 10 приведено замкнутое тело, полученное вращением контура на криволинейной поверхности, приведенной на рис. 9.

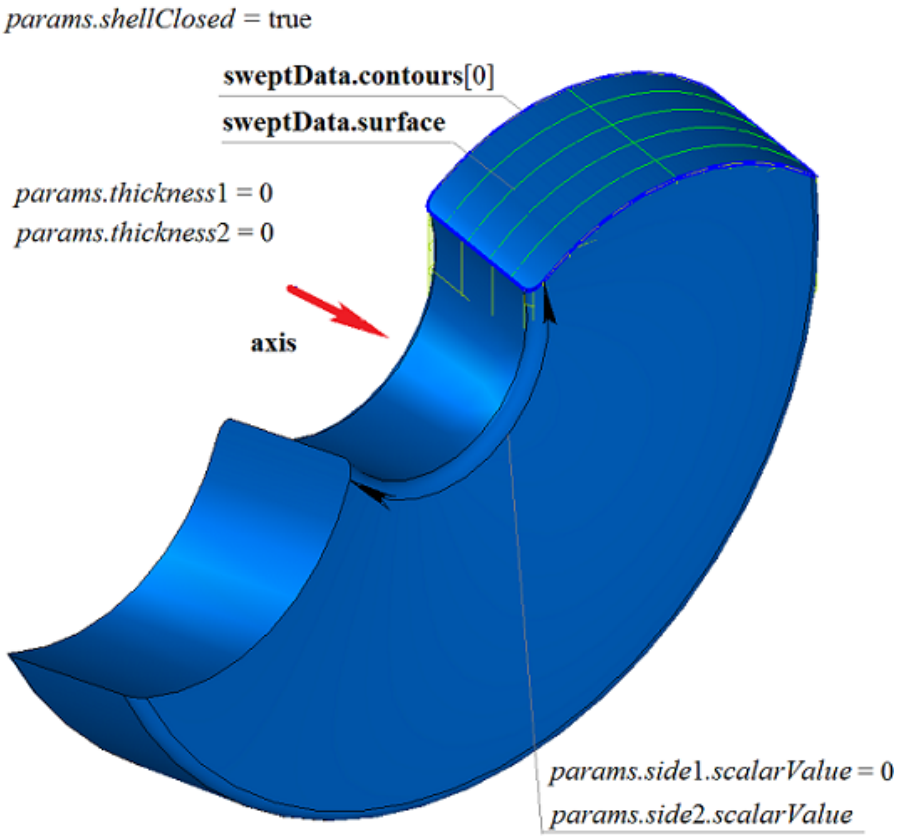


Рис. 10

На рис. 11 приведено замкнутое тонкостенное тело, полученное вращением контура на

криволинейной поверхности, приведенной на рис. 9.

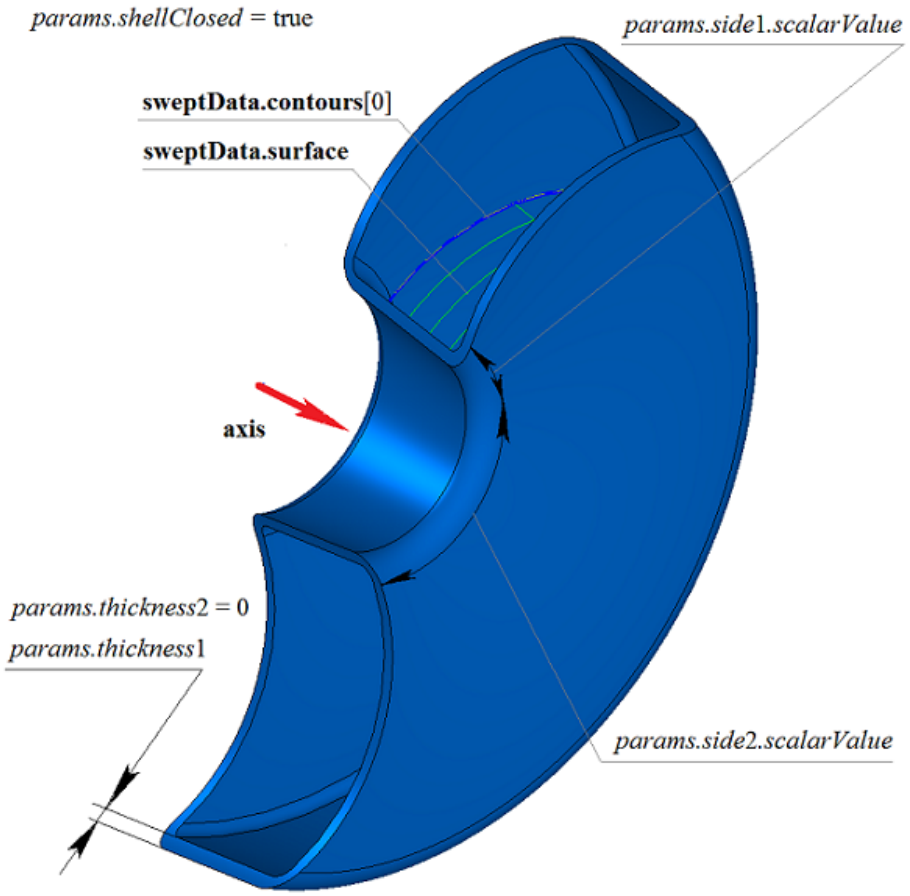


Рис. 11

На рис. 12 приведено незамкнутое тело, полученное вращением контура на криволинейной поверхности, приведенной на рис. 9.

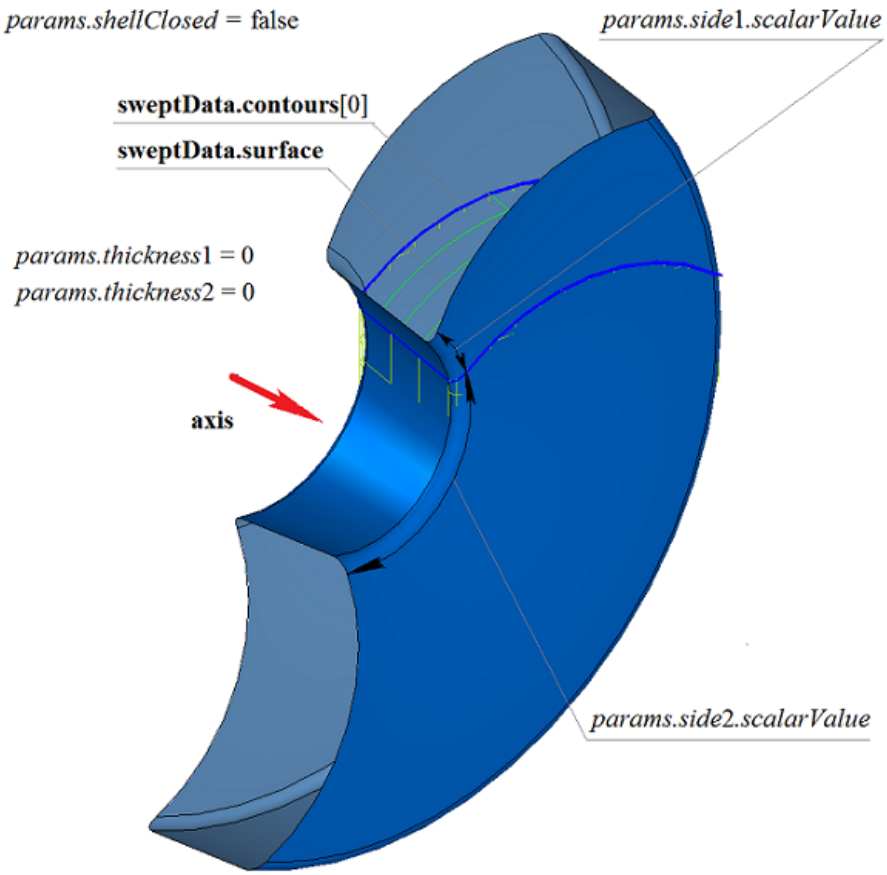


Рис. 12

Если на одной поверхности расположено множество не пересекающихся двумерных контуров, то рассматриваемая функция определяет внешние контуры и вложенные в них внутренние контуры, причем вложение может быть многократным. На рис. 13 приведено множество не пересекающихся двумерных контуров **contours** и плоская поверхность **surface** (MbPlane).

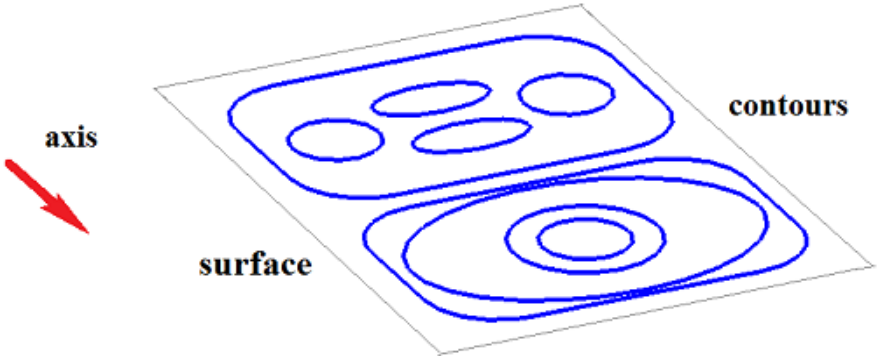


Рис. 13

На рис. 14 приведено замкнутое тело, состоящее из нескольких частей, построенное вращением множества контуров, показанного на рис. 13.

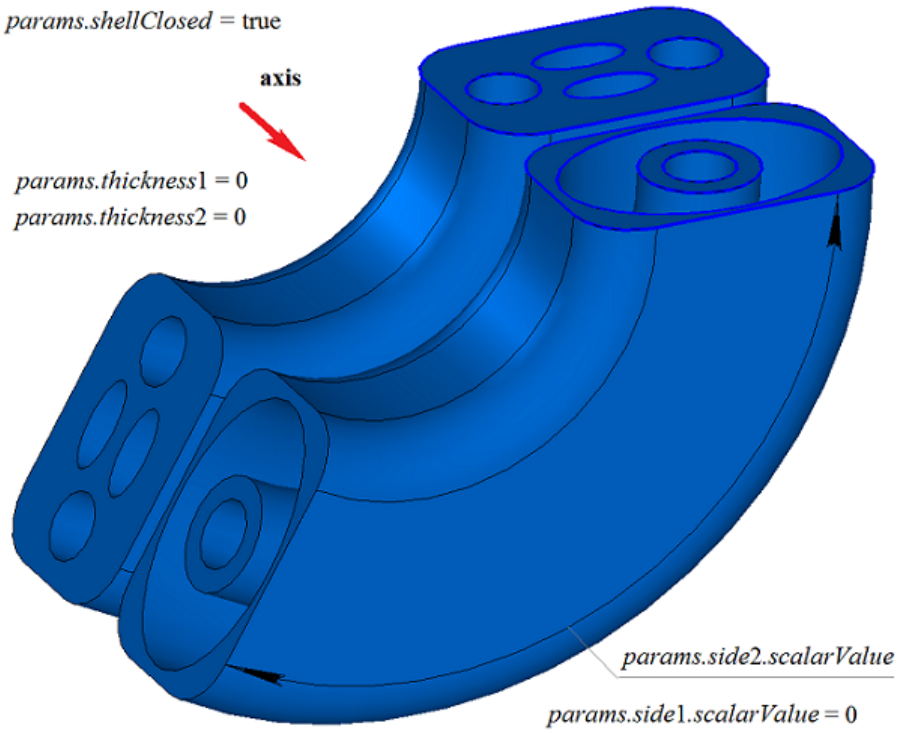


Рис. 14

На рис. 15 приведено замкнутое тонкостенное тело, состоящее из нескольких частей, построенное вращением множества контуров, показанного на рис. 13.

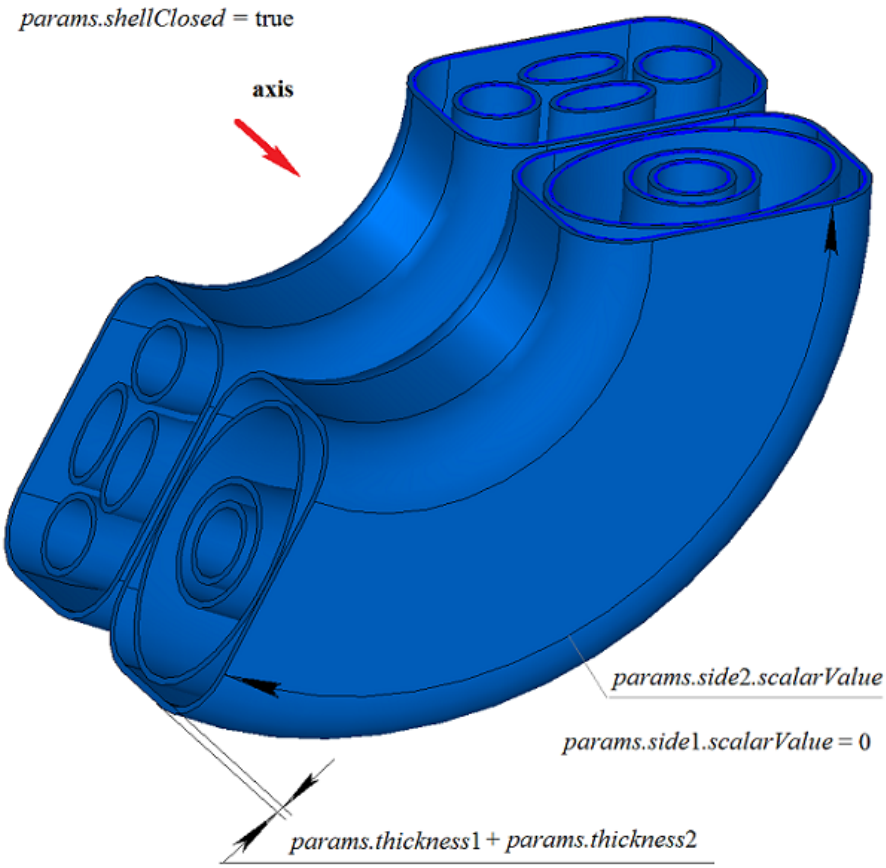


Рис. 15

На рис. 16 приведены два трехмерных контура.

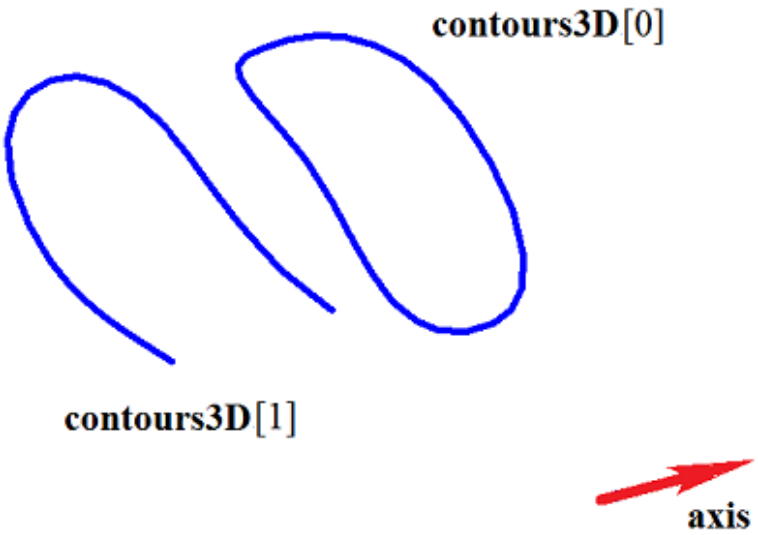


Рис. 16

На рис. 17 приведено двусвязное тонкостенное замкнутое тело, полученное вращением трехмерных контуров, приведенных на рис. 16.

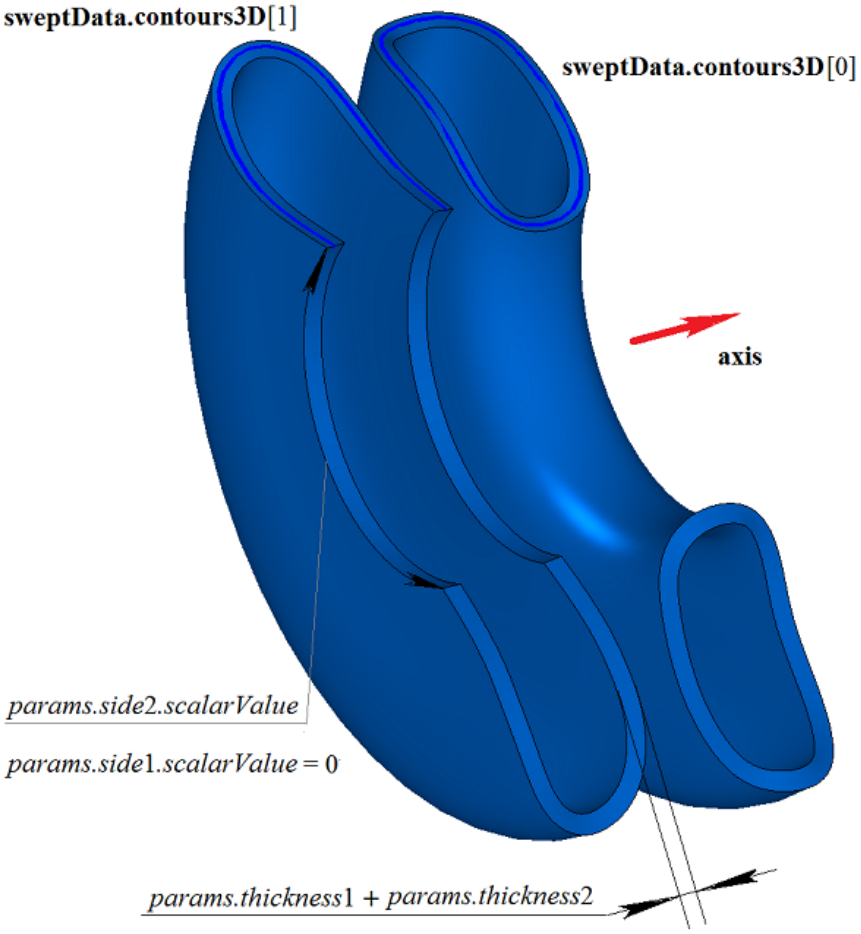


Рис. 17

На рис. 18 приведены два незамкнутых тела, полученных вращением трехмерных контуров, приведенных на рис. 16. Построение выполнено для каждого контура отдельно.

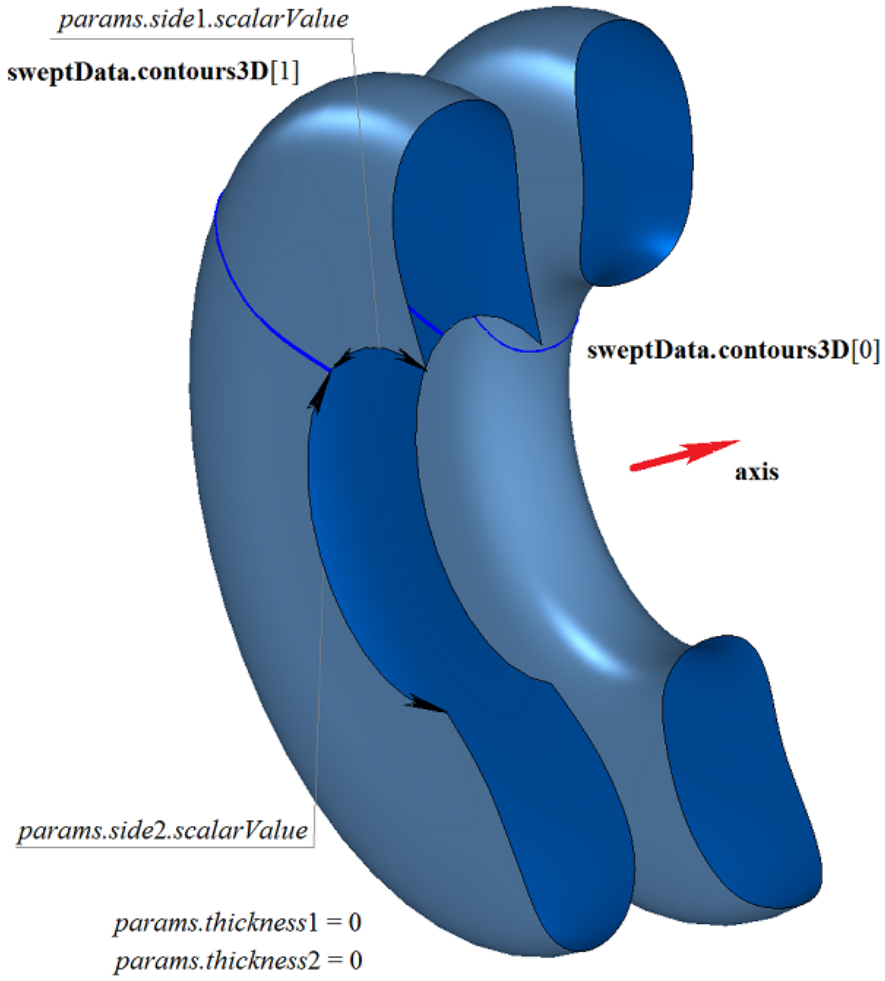


Рис. 18

При построении операции вращения до конструктивной плоскости с вырожденной областью определения оболочка будет построена до ближайшего пересечения с указанной плоскостью.

В случае, если плоскость имеет невырожденные границы, то часть плоскости, до которой будет построено вращение, определяется по положению центра тяжести контура, ограничивающего плоскость, относительно проекции оси вращения на плоскость.

Функция построения тела вращения RevolutionSolid() добавляет в журнал построенного тела строитель MbCurveRevolutionSolid, который содержит все необходимые для построения тела данные. Строитель MbCurveRevolutionSolid объявлен в файле *cr_revolution_solid.h*.

В тестовом приложении *test.exe* построение тела вращения выполняется командами меню «Создать->Тело->На базе кривых->Вращением поверхностной кривой» и «Создать->Тело->На базе кривых->Вращением трехмерной кривой».

3.1.4 Построение тела по сечениям

Функция

```
MbResultType LoftedSolid( const MbLoftedSolidParams & loftSolParams,
                                c3d::SolidSPtr &      result );
```

выполняет построение тела по пространственным (в том числе плоским) сечениям.

Входным параметром функции является объект **loftSolParams** с исходными данными для выполнения операции (определение класса MbLoftedSolidParams см. *op_shell_parameter.h*).

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Для инициализации объекта **loftSolParams** с параметрами построения в классе MbLoftedSolidParams заданы следующие конструкторы (см. *op_shell_parameter.h*):

```
MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );

MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );

MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const OperationType                   oType,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );

MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                    const c3d::ConstSpaceCurveSPtr &      spine,
                    const LoftedValues &                  params,
                    const OperationType                   oType,
                    const c3d::SpaceCurvesSPtrVector *    guideCurves,
                    const c3d::SpacePointsVector *        guidePoints,
                    const MbSNameMaker &                  operName,
                    const c3d::SNamesMakerSPtrVector &    contourNames,
                          IProgressIndicator *            prog = nullptr );
```

где

- **places** – множество систем координат образующих контуров;
- **planeConours** – множество образующих контуров;
- **surfaces** – множество поверхностей образующих контуров.
- **oType** – тип булевой операции;
- **spine** – осевая кривая (может быть nullptr);
- **params** – параметры операции;
- **guideCurves** – множество направляющих кривых, задающих траектории соответствующих точек контуров;
- **guidePoints** – множество точек на образующих контурах, задающее соответствующие точки (цепочки точек);
- **operName** – именователь операции;
- **contourNames** – именователи образующих контуров;
- **prog** – индикатор прогресса выполнения операции.

Построение тела по плоским сечениям

Поверхность построенного по плоским сечениям тела проходит по всем плоским кривым, определяющим тело. Множество **places** содержит локальные системы координат, в плоскости XY которых располагаются двумерные контуры **planeConours**.

Множества **places** и **planeConours** согласованы по индексу: **planeConours**[*i*] располагается в плоскости XY локальной системы координат **places**[*i*]. Ориентация контуров **planeConours** может быть произвольной. Если все контуры **planeConours** замкнуты, то для предотвращения перекручивания поверхностей положения начал контуров меняются так, чтобы начала располагались как можно ближе друг к другу. Изменить требуемым образом соответствие точек стыковки кривых множества контуров позволяют контрольные точки **guidePoints**. Если множество **guidePoints** не пустое, то оно должно быть согласовано с множествами **places** и **planeConours**. Для управления формой тела между сечениями могут использоваться направляющие кривые **guideCurves**. Направляющей кривой тела может служить произвольная кривая.

Параметр **params** содержит информацию о способе движения, наличии и толщине стенок тела, замкнутости построенного тела.

Параметры **params**.*thickness1* и **params**.*thickness2* определяют толщину стенки построенного тонкостенного тела. Параметр **params**.*thickness1* задает отступ наружу от образующей кривой, а параметр **params**.*thickness2* задает отступ внутрь от образующей кривой.

Параметр **params**.*shellClosed* управляет замкнутостью построенного тела (*true* – замкнутое тело; *false* – незамкнутое тело).

Параметр **params**.*checkSelfInt* сообщает о необходимости проверки результата построения на самопересечение. По умолчанию **params**.*checkSelfInt* = *false* – проверка не выполняется.

Управляющие флаги **mergeFlags** управляют слиянием подобных граней или ребер (**mergeFaces** = *true* – сливать подобные грани; **mergeEdges** = *true* – сливать подобные ребра).

Параметр **params**.*closed* управляет замкнутостью трубки сечений (наличием торцов у тела). При **params**.*closed* = *true* торцы тела отсутствуют и тело имеет топологию тора. По умолчанию **params**.*closed* = *false*.

Векторы **params**.*vector1* и **params**.*vector2* задают направление тела в районе начального и конечного торцов. Например, они позволяют задать направление тела в районе торцов ортогональным плоскостям торцов. По умолчанию векторы **params**.*vector1* и **params**.*vector2* равны нулю.

Параметры **params**.*setNormal1* и **params**.*setNormal2* устанавливают нормали в начальном и в конечном торцах соответственно, если начальное сечение точечное (по умолчанию *false*).

Параметры **params**.*derFactor1* и **params**.*derFactor2* устанавливают множители величин производных при установке нормалей в начальном и в конечном торцах соответственно.

Векторы **params**.*directSurf1* и **params**.*directSurf2* задают ось направления движения поверхности в начальном и в конечном торцах соответственно при установке нормали.

На рис. 1 приведены данные, используемые при построении, и схема наследования параметров построения тела по плоским сечениям MbLoftedSolidParams & **loftSolParams**:

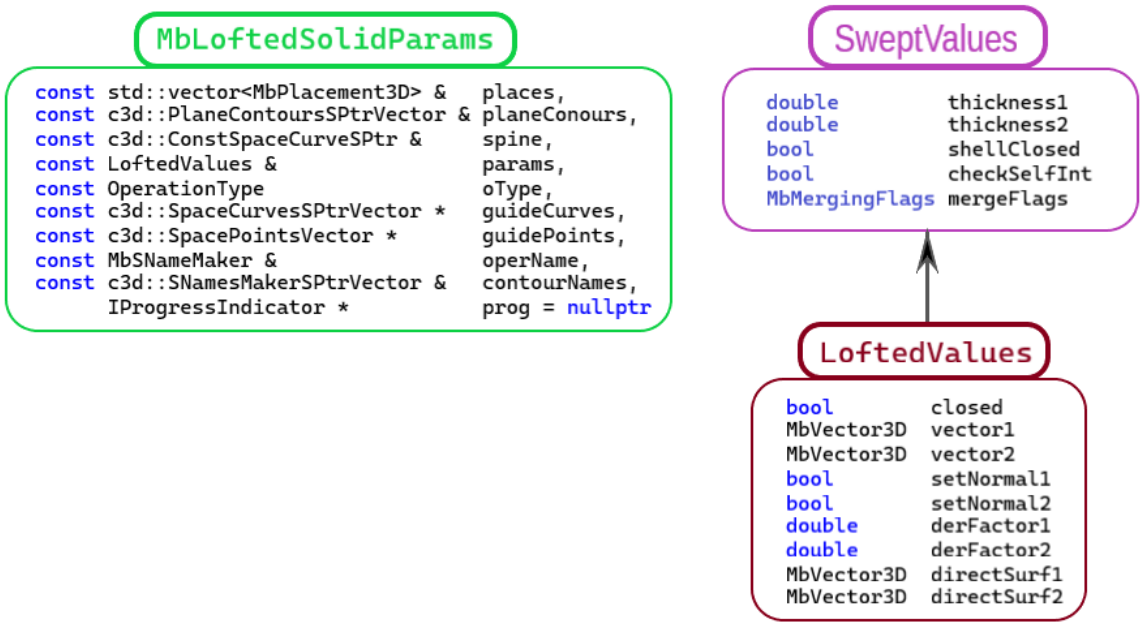


Рис. 1

Параметр **contourNames** обеспечивает именование граней построенного тела.

На рис. 2 приведено множество двумерных контуров **planeConours** и их локальные системы координат **places** (стрелками указаны направления нормалей локальных систем координат):

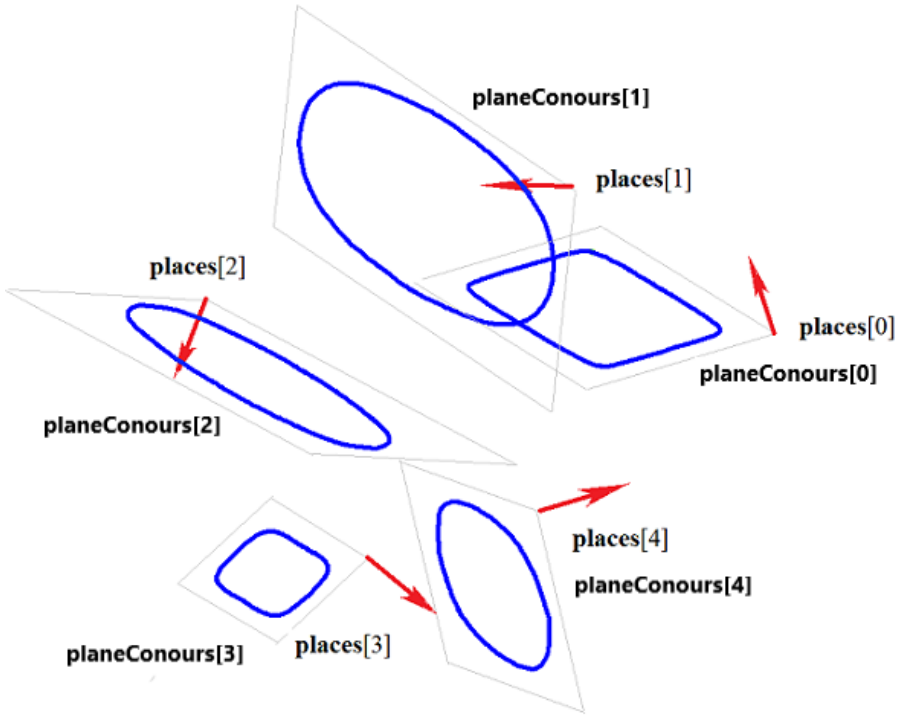


Рис. 2

На рис. 3 приведено замкнутое тело, построенное по плоским сечениям, приведенным на рис. 2, с заданными направлениями нормалей на торцах для значения параметра **params**.*closed* = *false*.

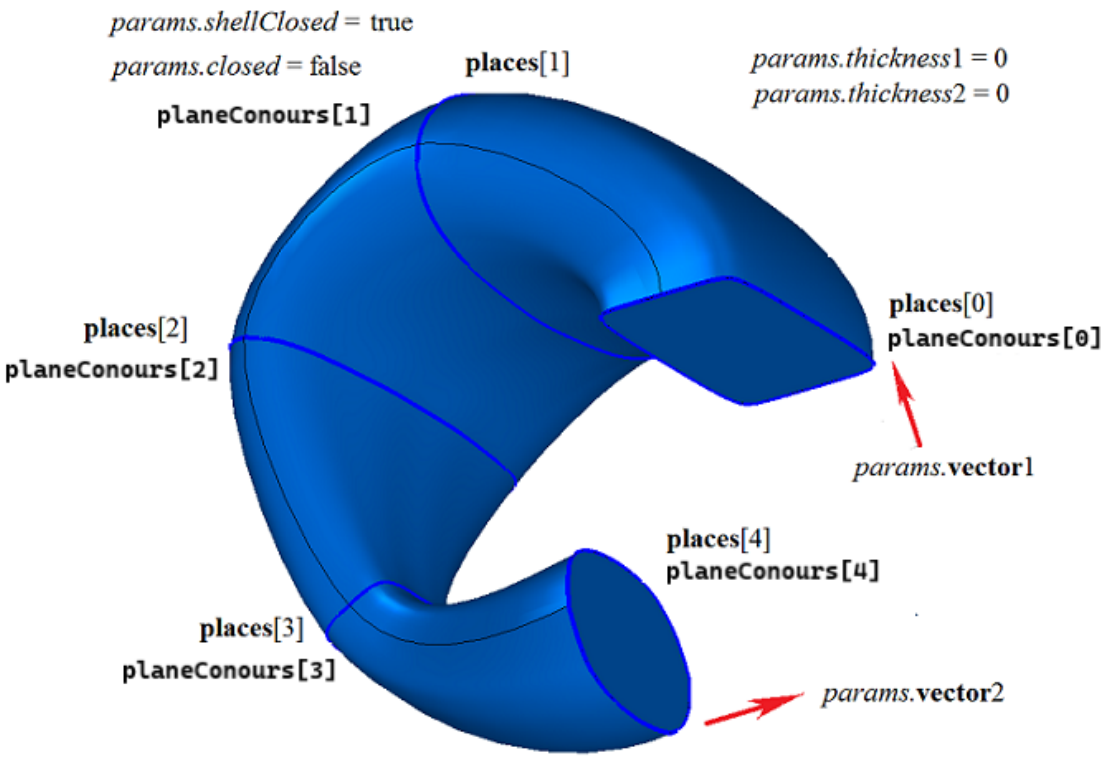


Рис. 3

На рис. 4 приведено замкнутое тело, построенное по плоским сечениям, приведенным на рис. 2, с параметром **params**.*closed* = *true*. Построенное тело имеет топологию тора и у него отсутствуют торцы.

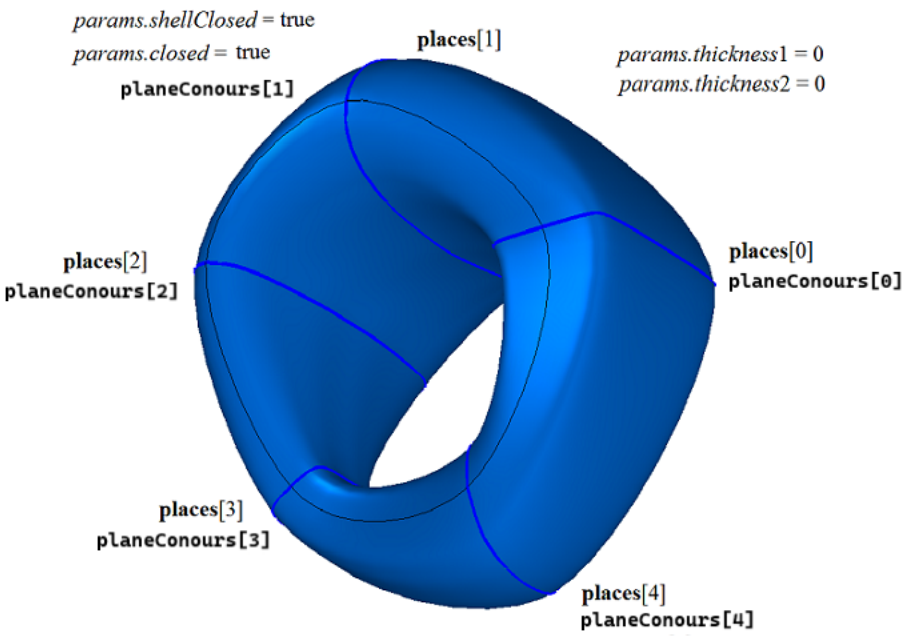


Рис. 4

На рис. 5 приведено замкнутое тонкостенное тело, построенное по плоским сечениям, приведенным на рис. 2, без задания нормалей на торцах для параметра **params**.*closed* = *false*:

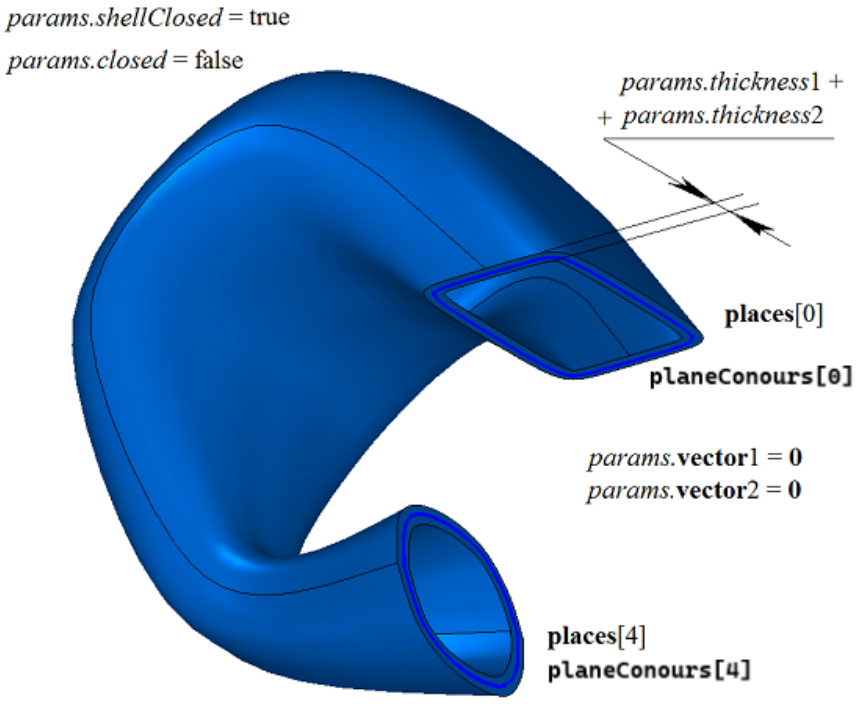


Рис. 5

На рис. 6 приведено замкнутое тонкостенное тело, построенное по плоским сечениям,

приведенным на рис. 2, с заданными нормалями на торцах для параметра **params**.*closed* = *false*:

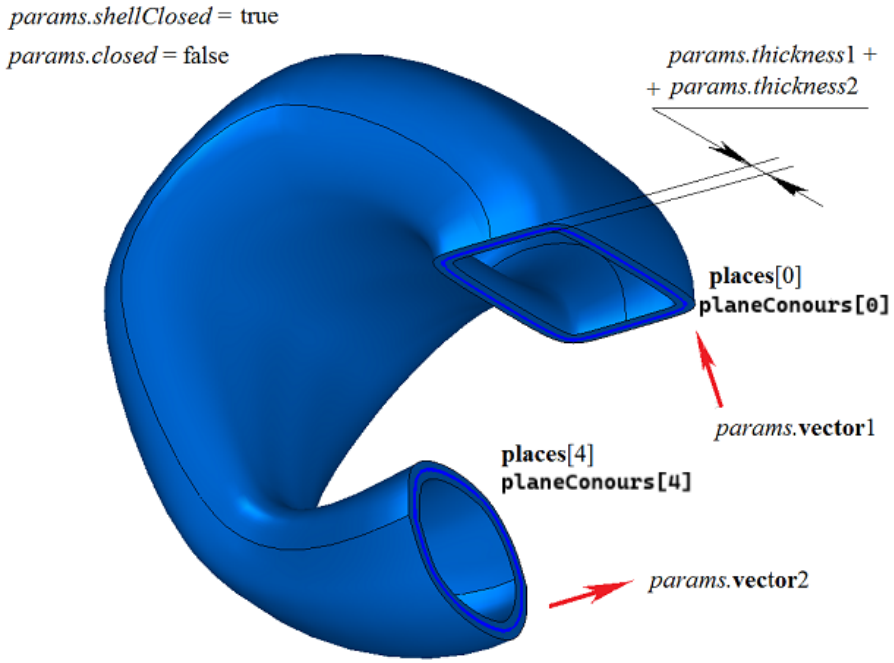


Рис. 6

На рис. 7 приведено замкнутое тонкостенное тело, построенное по плоским незамкнутым контурам с не заданными нормалями на торцах для параметра **params**.*closed* = *false*. Параметры **params**.*thickness1* и **params**.*thickness2* должны быть не нулевыми.

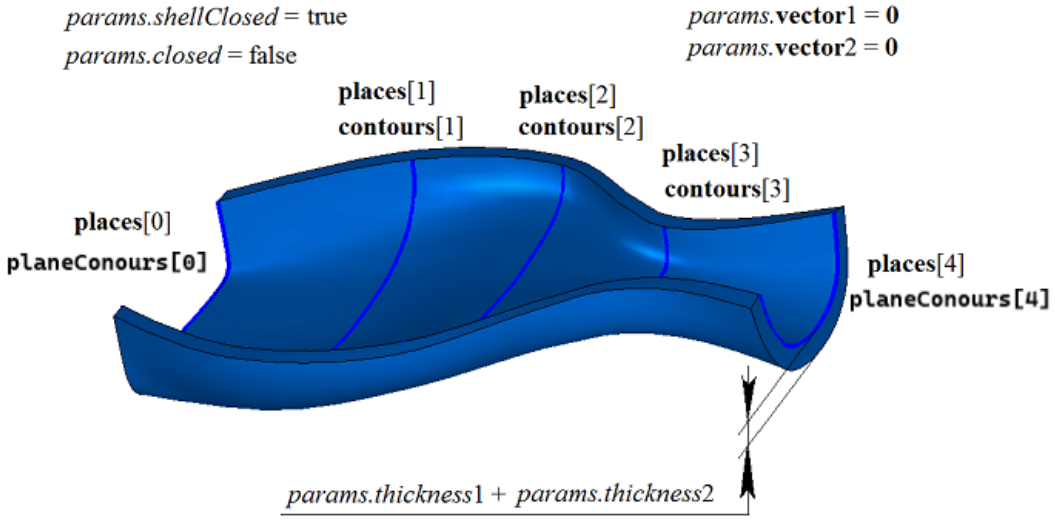


Рис. 7

На рис. 8 приведено незамкнутое тело, построенное по плоским незамкнутым контурам с не заданными нормалями на торцах для значения параметра **params**.*closed* = *false*. Параметры **params**.*thickness1* и **params**.*thickness2* могут быть нулевыми.

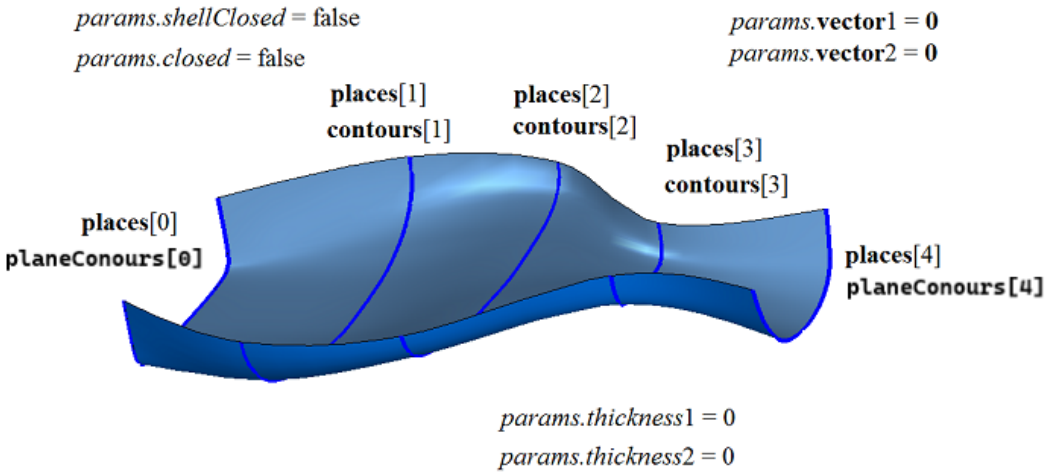


Рис. 8

При неодинаковом числе сегментов плоских контуров выполняется разбивка некоторых сегментов так, чтобы число сегментов в каждом контуре множества **planeConours** было равным. На рис. 9 приведены три контура с разным числом сегментов.

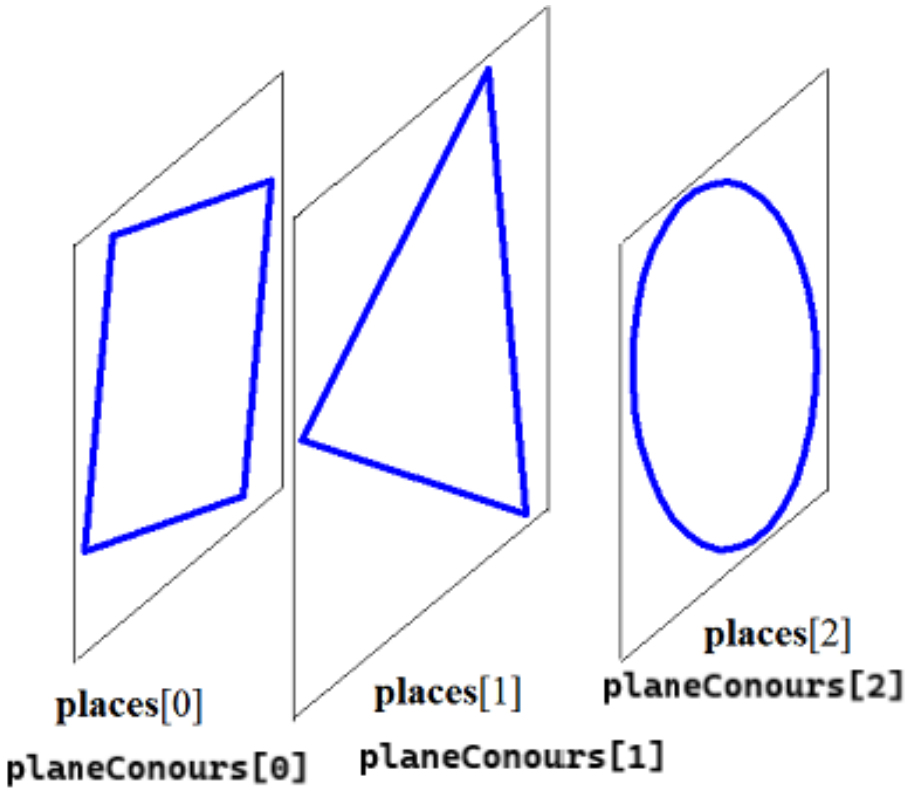


Рис. 9

На рис. 10 приведено построенное по этим контурам замкнутое тело, в котором один из сегментов треугольного контура разбит на два сегмента, а окружность разбита на четыре

дуги:

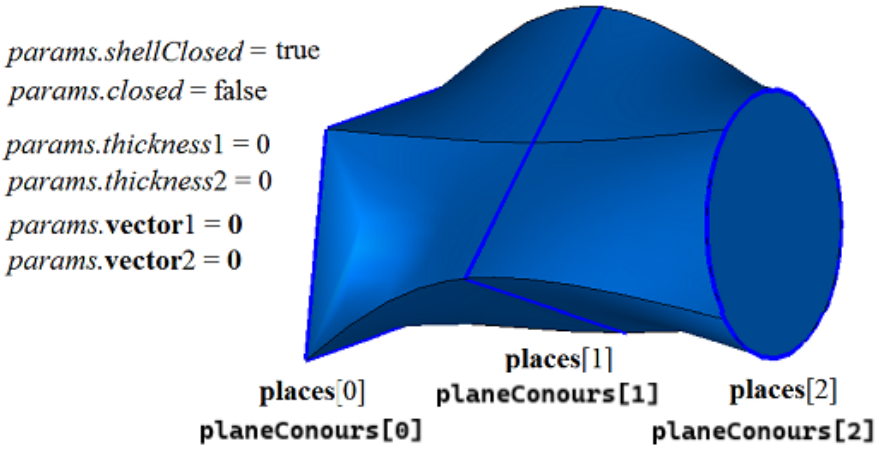


Рис. 10

Управлять положением ребер, соединяющих вершины разных контуров множества, можно с помощью контрольных точек **guidePoints**. Точки **guidePoints**[*i*] указывают положения стыков сегментов разных контуров множества, которые должны соединяться ребрами. Для демонстрации применения контрольных точек **guidePoints** построим тело по плоским сечениям, приведенным на рис. 11:

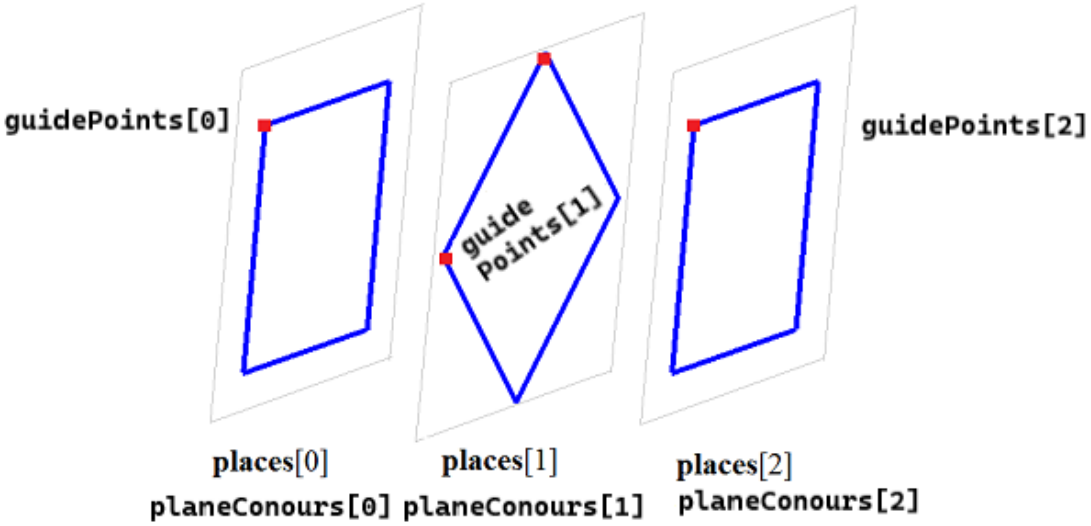


Рис. 11

На рис. 12 и 13 показаны тела, построенные по плоским сечениям, приведенным на рис. 11, с различным положением контрольных точек **guidePoints**.

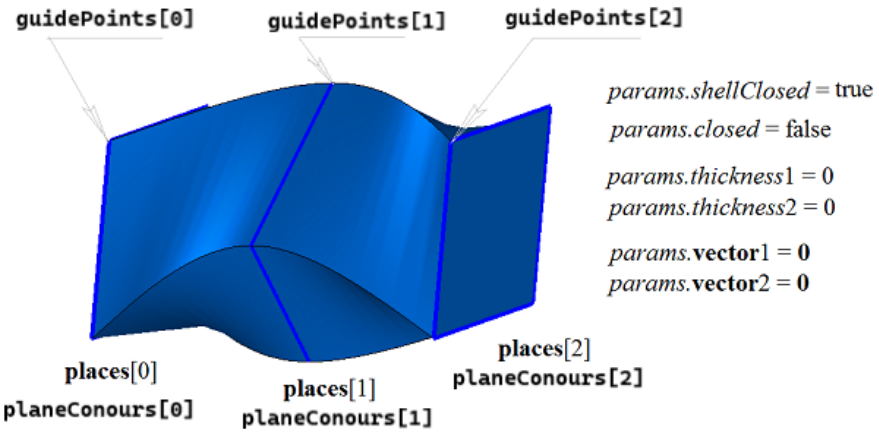


Рис. 12

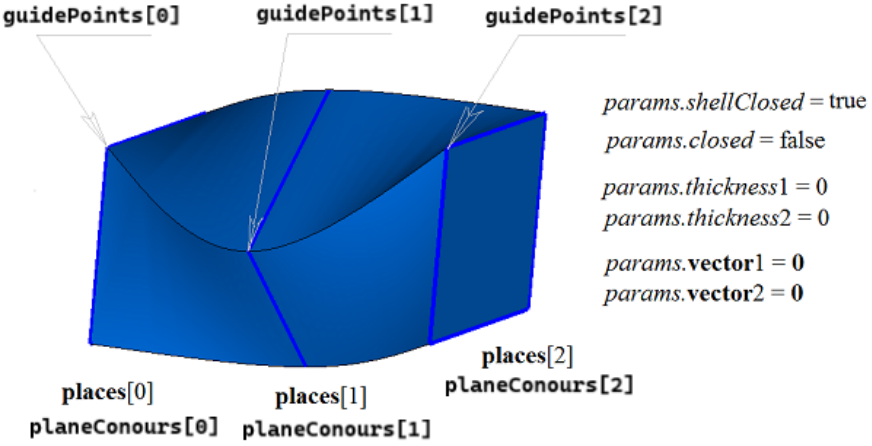


Рис. 13

На рис. 14 приведены два двумерных контура и осевая кривая **spine**, которая будет служить направляющей при построении тела по плоским сечениям:

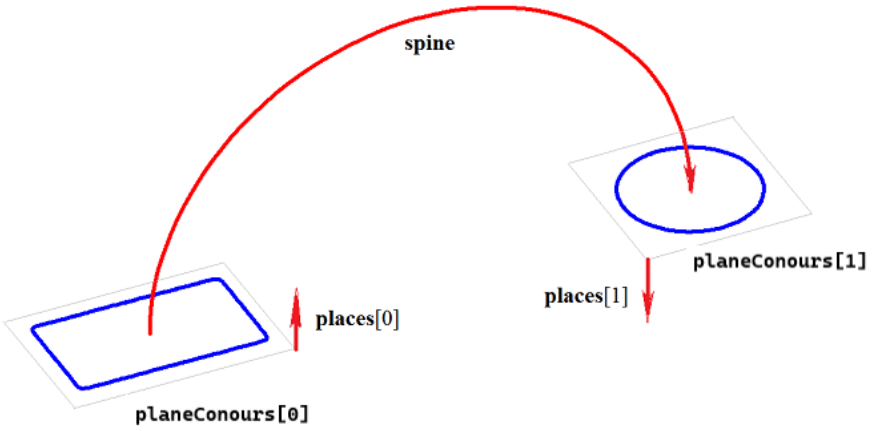


Рис. 14

На рис. 15 приведено замкнутое тело, построенное по плоским сечениям и осевой кривой, приведенным на рис. 14:

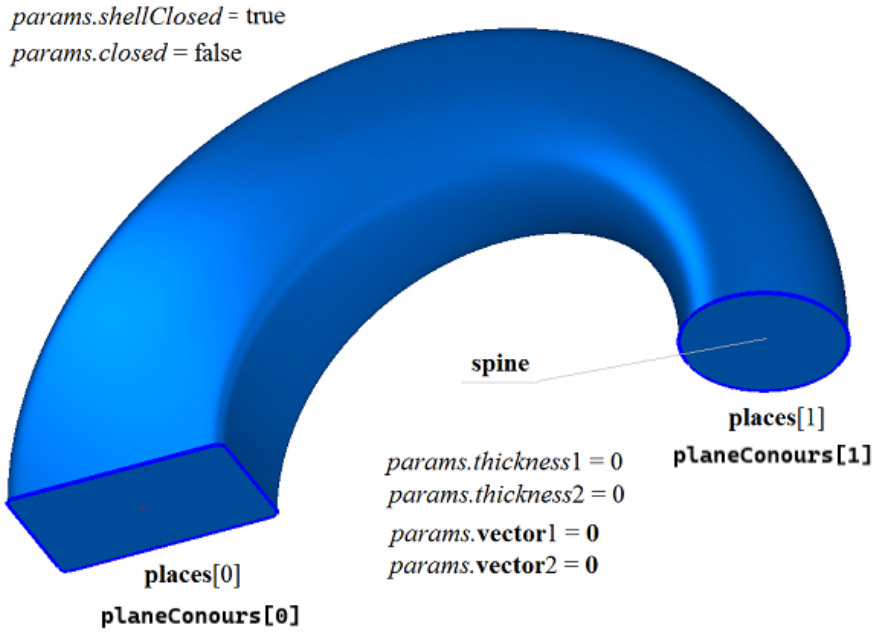


Рис. 15

На рис. 16 приведено замкнутое тонкостенное тело, построенное по плоским сечениям и осевой кривой, приведенным на рис. 14:

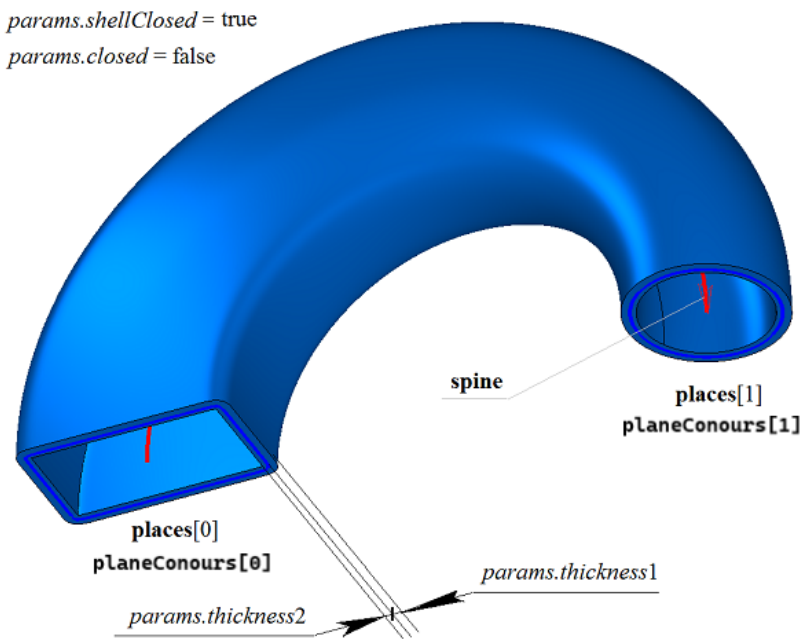


Рис. 16

На рис. 17 приведено незамкнутое тело, построенное по плоским сечениям и осевой кривой, приведенным на рис. 14:

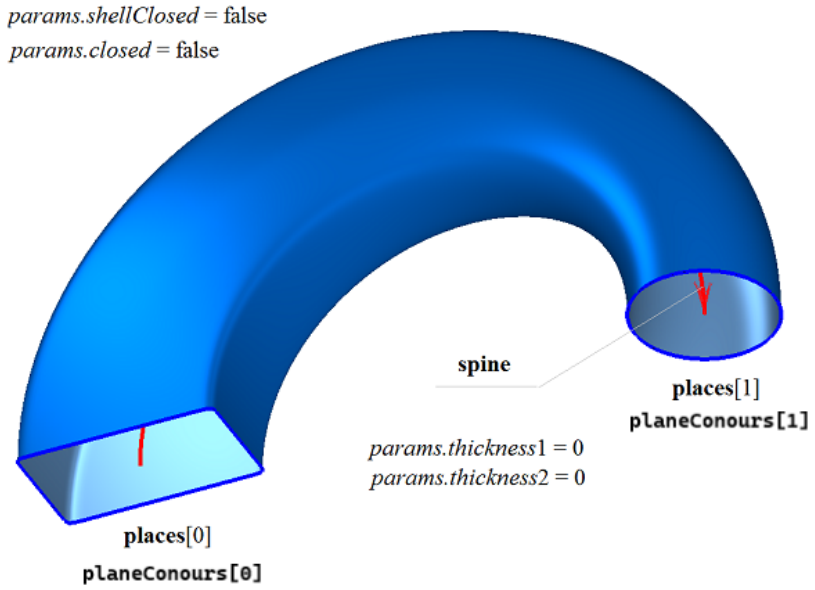


Рис. 17

Функция построения тела по сечениям LoftedSolid() добавляет в журнал построенного тела строитель MbCurveLoftedSolid, который содержит все необходимые для построения тела данные. Строитель MbCurveLoftedSolid объявлен в файле *cr_lofted_solid.h*.

В тестовом приложении *Test.exe* построение тела по плоским сечениям выполняется ко-

мандами меню «Создать→Тело→На базе кривых→По сечениям» и «Создать→Тело→На базе кривых→По сечениям, осевой линии и направляющим».

3.1.5 Создание тела по заданным множеству граней или оболочке

Функция

```
MbSolid * CreateSolid(       MbFaceShell &  shell,
                       const MbSNameMaker & names );
```

создаёт тело с заданными множеством граней или оболочкой без истории построения.

Функция принимает в качестве входного параметра объект **shell** класса MbFaceShell (см. определение в *topology_faceset.h*), содержащий параметры заданных множества граней или оболочки.

При удачной работе функция возвращает построенное тело без истории, в противном случае возвращает ноль.

Функция объявлена в файле *action_solid.h*.

Параметр **names** обеспечивает именование граней, ребер и вершин построенного тела.

Оболочка представляет собой набор граней MbFace, стыкующихся друг с другом по рёбрам MbCurveEdge.

В общем случае оболочка может быть многосвязной и описывать несколько не связанных между собой наборов граней.

Оболочка называется **замкнутой**, если она не имеет края, в противном случае оболочка называется **незамкнутой**.

Замкнутость оболочки указывает на возможность использования множества её внутренних точек в операциях над телами MbSolid.

Формально множество граней не ограничено никакими условиями, однако реально грани множества удовлетворяют некоторым условиям.

Чтобы подчеркнуть эту особенность оболочки, вводится понятие «Однородная оболочка». Оболочки, удовлетворяющие нижеперечисленным требованиям, называются **однородными**:

1. Оболочки являются конечными.
2. Оболочки не пересекают сами себя.
3. Оболочки являются двухсторонними (ориентируемыми).
4. В каждом ребре оболочки стыкуется не более двух граней. Две грани оболочки стыкуются так, что внешняя сторона одной грани переходит во внешнюю сторону другой грани.
5. При любом обходе любой вершины замкнутой оболочки по её поверхности обязательно посещаются все примыкающие к данной вершине грани и пересекаются все выходящие из неё рёбра.

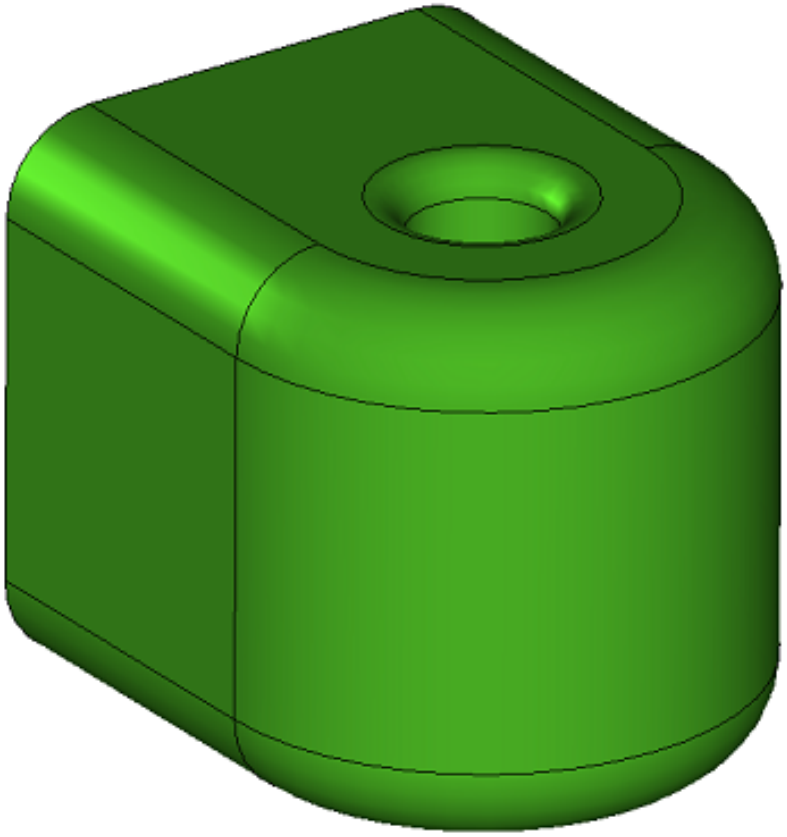
Так как внешняя сторона грани переходит во внешнюю сторону соседней грани, то однородная оболочка также имеет внешнюю и внутреннюю стороны.

Замкнутая однородная оболочка делит трёхмерное пространство на две части, одна из которых находится внутри оболочки.

Замкнутая однородная оболочка называется **внешней**, если её внешняя сторона направлена вне ограничиваемой оболочкой части пространства.

Замкнутая однородная оболочка называется **внутренней**, если её внешняя сторона направлена внутрь ограничиваемой оболочкой части пространства.

На рисунке приведено тело, построенное по множеству граней или оболочке:



shell

Функция выполняет именование неименованных граней, ребер, вершин и создаёт тело по заданной оболочке. Рассматриваемая функция не выполняет никаких проверок и построений. Если множество граней содержит краевые ребра, то функция построит незамкнутое тело. В журнал созданного тела функция CreateSolid() добавляет простой строитель MbSimpleCreator, который объявлен в файле *cr_simple_creator.h*.

3.1.6 Построение незамкнутого тела на базе поверхности

Функция

```
MbResultType SurfaceShell( const MbSurfaceShellParams & params,
                           c3d::SolidSPtr &       result );
```

выполняет построение тела, состоящего из одной грани на базе исходной поверхности. Поверхность должна быть без самопересечений, с корректной ориентацией ограничивающих кривых в случае поверхности MbCurveBoundedSurface.

Функция принимает в качестве входного параметра объект **params** класса MbSurfaceShellParams (см. определение в *op_shell_parameter.h*), содержащий параметры передаваемой поверхности:

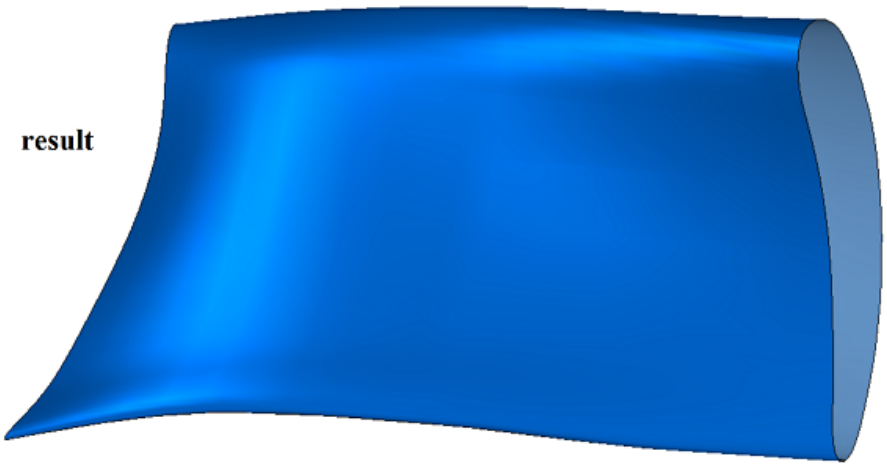
- **_surface** — передаваемая поверхность;
- **_operNames** — именователь операции;
- **_sameSurface** — флаг использования поверхности (true) или ее копии (false).

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_shell.h*.

На рисунке приведено тело, построенное по поверхности, циклически замкнутой по первому параметрическому направлению:



Рассматриваемая функция не выполняет никаких проверок и вычислений. Построенное тело состоит из одной грани. Грань от исходной поверхности отличается наличием ребер на краях и швах и наличием вершин в точках стыковки рёбер. Если исходная поверхность циклически замкнута по одному или обоим параметрическим направлениям, то построенное тело будет состоять из циклически замкнутой по соответствующим рёбрам грани. Если исходная поверхность имеет края, то функция построит незамкнутое тело. В журнал созданного тела функция добавляет простой строитель MbSimpleCreator, который объявлен в файле *cr_simple_creator.h*.

В тестовом приложении *test.exe* построить тело на базе поверхности можно с помощью команды меню «Создать→Тело→На базе поверхности».

3.1.7 Построение линейчатого незамкнутого тела

Функция

```
MbResultType RuledShell( const MbRuledShellParams & ruledParams,
                         MbRuledShellResult & outParams );
```

выполняет построение линейчатого незамкнутого тела по двум кривым, заданным в параметрах.

Функция принимает в качестве входного параметра объект **ruledParams** класса MbRuledShellParams (см. определение в *op_shell_parameter.h*), содержащий параметры построения:

- **pars** — параметры операции (геометрическая информация, необходимая для построения тела);
- **operNames** — именователь граней, ребер и вершин построенного тела;
- **isPhantom** — режим создания фантома.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_shell.h*.

Линейчатое тело строится по кривым **pars**.*curve0* и **pars**.*curve1*. Если кривые являются составными, то на каждой паре сегментов составных кривых будет построена линейчатая поверхность, а на каждой поверхности будет построена грань тела. Смежные грани будут иметь общие ребра.

Кривые **pars**.*curve0* и **pars**.*curve1* могут быть дополнительно разбиты на сегменты параметрами **pars**.*breaks0* и **pars**.*breaks1*.

Параметр **pars**.*joinByVertices* указывает, соединять ли контуры с одинаковым количеством сегментов через вершины. По умолчанию **pars**.*joinByVertices* = *false*.

Параметр **pars**.*checkSelfInt* указывает, проверять ли кривые на самопересечение. По умолчанию **pars**.*checkSelfInt* = *false*.

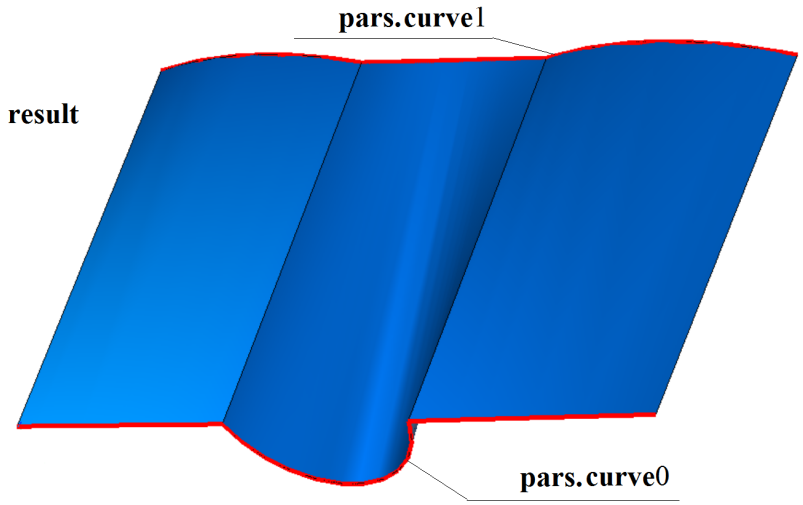
Параметр **pars**.*simplifyFaces* указывает, упрощать ли грани построенного тела.

Параметр **pars**.*proportional* инициирует пропорциональную натуральной параметризацию кривых.

Параметр **pars**.*mismatchMax* определяет допустимое несовпадение противолежащих точек стыковки сегментов.

Параметр **pars**.*segmentSplit* указывает, разделять ли оболочку на грани по сегментам контуров.

На рисунке приведено линейчатое тело, построенное по составным кривым, каждая из которых содержит три сегмента:



Построенное тело состоит из одной или нескольких граней. Линейчатое тело всегда имеет края, проходящие по исходным кривым. Рассматриваемая функция может выполнять проверку на самопересечение граней. Кривые **pars**.*curve0* и **pars**.*curve1* могут быть циклически замкнуты. В журнал созданного тела функция добавляет строитель MbRuledShell, который объявлен в файле *cr_ruled_shell.h*.

3.1.8 Построение тела по сети кривых

Функция

```
MbResultType MeshShell( const MbMeshShellParameters & params,
                              MbMeshShellResults &    result );
```

выполняет построение тела по сети кривых.

Функция принимает в качестве входного параметра объект **params** класса MbMeshShellParameters (см. определение в *op_shell_parameter.h*), содержащий параметры:

- **_params** – параметры построения;
- **_operNames** – именователь граней, ребер и вершин построенного тела;
- **_isFantom** – флаг создания фантомного тела (true – режим фантома).

Выходным параметром функции является объект класса MbMeshShellResults **result**, содержащий построенное тело и флаги направлений сопряжений.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_shell.h*.

Параметры построения **_params** содержат геометрическую информацию, необходимую для построения тела, рис. 1:

MeshSurfaceValues

```
RPArray<MbCurve3D> curvesU;
RPArray<MbCurve3D> curvesV;
RPArray<MbPolyline3D> chainsU;
RPArray<MbPolyline3D> chainsV;
bool                  uClosed;
bool                  vClosed;
bool                  checkSelfInt;
MbeMatingType         type0;
MbeMatingType         type1;
MbeMatingType         type2;
MbeMatingType         type3;

c3d::SurfacesVector surface0;
c3d::SurfacesVector surface1;
c3d::SurfacesVector surface2;
c3d::SurfacesVector surface3;

bool                  defaultDir0;
bool                  defaultDir1;
bool                  defaultDir2;
bool                  defaultDir3;
bool                  autoDirection;
bool                  tesselate;
bool                  g2Cont;
bool                  useProport;
std::map<size_t, bool> splitU;
std::map<size_t, bool> splitV;
```

Рис. 1

Тело строится по двум множествам кривых **_params.curvesU** и **_params.curvesV**. Кривые **_params.curvesU** расположены вдоль первого параметрического направления граней тела. Кривые **_params.curvesV** расположены вдоль второго параметрического направления граней тела.

Кривые по каждому направлению указываются в порядке их следования в сети. Каждая кривая одного направления должна иметь по одной общей точке с каждой кривой дру-

гого направления.

В качестве одной или обеих крайних кривых какого-либо одного направления могут быть указаны точечные кривые. В этом случае все кривые другого направления должны проходить через точку (точки), указанную для данного направления.

Замыкание поверхности по тому или иному направлению возможно, если все кривые данного направления замкнуты.

Кривые могут быть с изломами и в этом случае будет построена многогранная оболочка с ребрами по месту излома.

Наборы цепочек по параметрическим направлениям **_params.chainsU**, **_params.chainsV** задаются в виде ломаных кривых и служат для выравнивания параметризации поверхности. Каждая цепочка определяет дополнительные вершины, через которые будут автоматически достроены дополнительные кривые без участия пользователя.

Существует возможность создания поверхности только по одному семейству кривых. В этом случае флаги **defaultDir** задают направление вывернутости поверхности при задании сопряжения, т.к. для каждой границы возможны два варианта вывернутости.

Параметр **_params**.*uClosed* указывает на замкнутость поверхности тела вдоль первого параметрического направления, для этого все кривые множества **_params.curveU** должны быть циклически замкнуты.

Параметр **_params**.*vClosed* указывает на замкнутость поверхности тела вдоль второго параметрического направления, для этого все кривые множества **_params.curveV** должны быть циклически замкнуты.

С помощью параметров **_params**.*uClosed* и **_params**.*vClosed* возможно построение незамкнутых поверхностей на циклически замкнутых кривых.

Параметр **_params**.*checkSelfInt* указывает, проверять ли поверхность построенного тела на самопересечение. По умолчанию **_params**.*checkSelfInt = false*. При обнаружении самопересечений функция выдает ошибку.

Типы сопряжений на границах **_params**.*type0*, **_params**.*type1*, **_params**.*type2*, **_params**.*type3* совместно с соответствующими сопрягаемыми поверхностями **_params**.*surface0*, **_params**.*surface1*, **_params**.*surface2*, **_params**.*surface3* определяют поведение тела на краях при значениях параметров грани *v = vMin*, *u = uMin*, *v = vMax*, *u = uMax* соответственно.

Параметры сопряжения **_params**.*type0*, **_params**.*type1*, **_params**.*type2*, **_params**.*type3* могут принимать значения *trt_Position*, *trt_Tangent*, *trt_Normal*, *trt_SmoothG2* из перечисления MbeMatingType (см. *mb_enum.h*).

Значение **trt_Position** параметры принимают по умолчанию, оно означает прохождение поверхности тела по кривым. Поверхности **_params**.*surface0*, **_params**.*surface1*, **_params**.*surface2*, **_params**.*surface3* при этом могут быть равны нулю.

Значение **trt_Tangent** означает касание соответствующего края тела поверхности **_params**.*surface0*, **_params**.*surface1*, **_params**.*surface2* или **_params**.*surface3*.

Значение **trt_Normal** означает ортогональность соответствующего края тела поверхности **params**.*surface0*, **params**.*surface1*, **params**.*surface2* или **params**.*surface3*.

Значение **trt_SmoothG2** означает касательное сопряжение по кривизне, при котором поверхность по сети кривых и поверхность сопряжения имеют в каждой точке границы совпадающие нормали и равные кривизны в любом направлении.

Флаг **tesselate** = *true* позволяет дополнить пользовательскую сеть кривых дополнительными кривыми, которые в некоторых случаях могут улучшить геометрию.

Флаги **splitU** = *true*, **sPlitV** = *true* задают пользовательским кривым признак, по которому кривые становятся разделительными ребрами результирующей оболочки, даже если они проходят через гладкие кривые трансверсального направления.

Флаг **usePropopt** = *true* репараметризует исходные кривые пропорционально длине, что приводит к возможному улучшению геометрии поверхности.

Флаг **autoDirection** меняет логику выбора направления по умолчанию. Если **autoDirection** = *true* – используется геометрический критерий (перемещение границы сопряжения вместе с поверхностью сопряжения может привести к смене вывернутости); если **autoDirection** = *false* – вывернутость будет определяться жестко.

Флаг **g2Cont** задает степень гладкости между патчами поверхности по сети кривых (сразу для двух направлений). Ею можно управлять, однако нужно иметь ввиду, что повышение порядка гладкости может приводить к ухудшению формы поверхности. В случае отсутствия сопряжений на трансверсальных границах и одного патча, в рассматриваемом направлении используется линейная интерполяция. В случае наличия любого сопряжения на трансверсальных границах либо нескольких патчей, в рассматриваемом направлении используется кубическая интерполяция (обеспечивается гладкость G1). В случае наличия гладкого сопряжения на трансверсальных границах либо при включенной опции **g2Cont**, в рассматриваемом направлении используется интерполяция 5-й степени (обеспечивается гладкость G2).

На рис. 2 приведена сеть кривых, состоящая из четырех циклически замкнутых кривых множества **_params.curveU** и двух незамкнутых кривых множества **_params.curveV**.

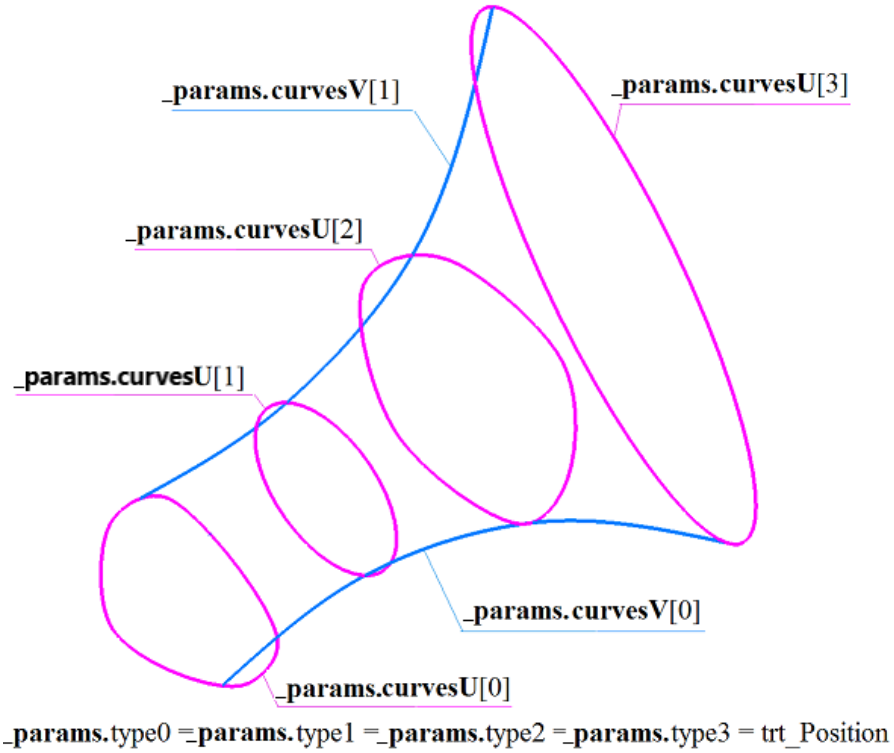


Рис. 2

На рис. 3 приведено тело, построенное по сети кривых, приведенных на рис. 2:

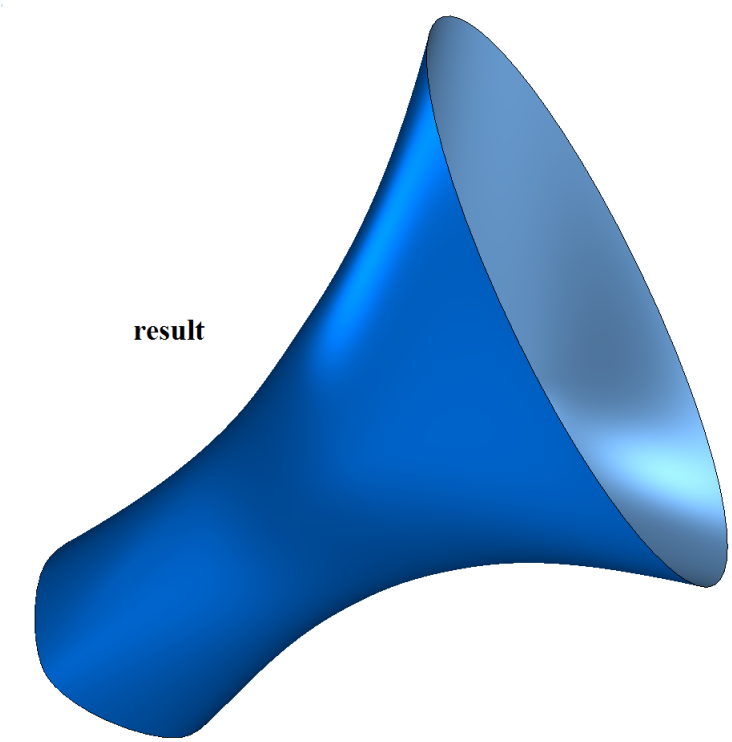


Рис. 3

На рис.4 приведена сеть кривых, состоящая из двух незамкнутых кривых множества **_params.curveU** и трех незамкнутых кривых множества **_params.curveV**, две крайние из которых вырождены в точки:

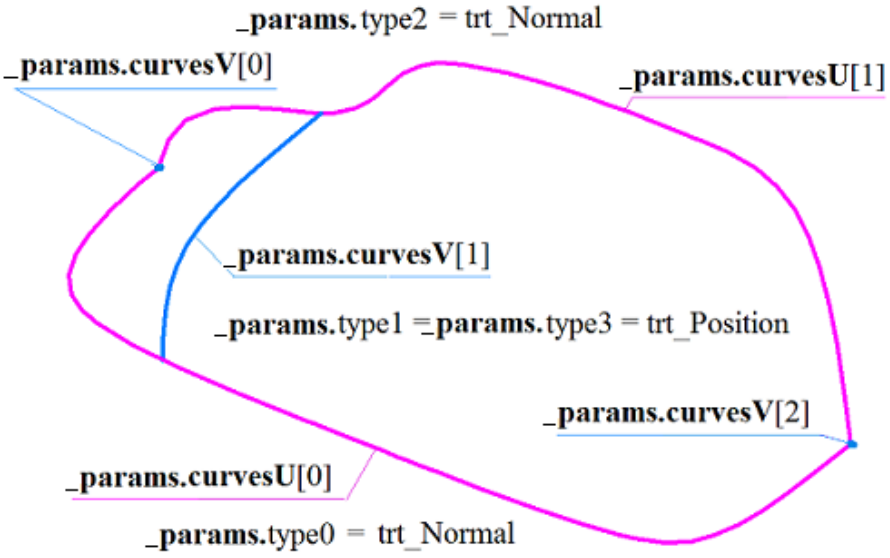


Рис. 4

На рис. 5 приведено тело, построенное по сети кривых, приведенных на рис. 4. Края построенного тела, соответствующие параметрам грани *v = vMin* и *v = vMax*, ортогональны плоским поверхностям **_params**.*surface0* и **_params**.*surface2*, так как параметры **_params**.*type0* и **_params**.*type2* имеют значения *trt_Normal*.

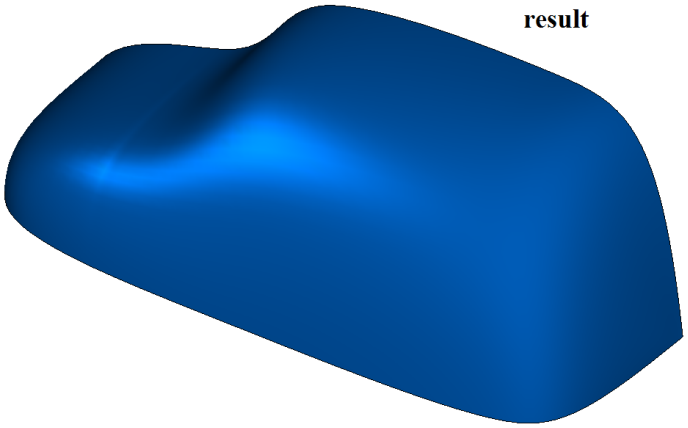


Рис. 5

В журнал созданного тела функция добавляет строитель MbMeshShell, который объявлен в файле *cr_mesh_shell.h*.

В тестовом приложении *test.exe* построить тело по сети кривых можно с помощью команды меню «Создать→Тело→На базе кривых».

3.1.9 Построение заплатки

```
MbResultType PatchShell( const MbPatchShellParams & params,
                               c3d::SolidSPtr &     result );
```

Функция выполняет построение заплатки по множеству кривых.

Функция принимает в качестве входного параметра объект **params** класса MbPatchShellParams, содержащий параметры, необходимые для операции (см. определение в *op_shell_parameter.h*):

- **initCurves** – набор кривых;
- **patchValues** – объект, содержащий параметры построения;
- **names** – именователь.

Класс MbPatchShellParams содержит 3 конструктора по набору кривых, параметрам и именователю:

```
MbPatchShellParams( const std::vector<SPtr<MbPatchCurve>> & initCurves,
                    const PatchValues &                     patchValues,
                    const MbSNameMaker &                    names );


MbPatchShellParams( const RPArray<MbPatchCurve> & initCurves,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                    const PatchValues &           patchValues,
                    const MbSNameMaker &          names );


MbPatchShellParams( const RPArray<MbCurve3D> & initCurves,
                    const PatchValues &        patchValues,
                    const MbSNameMaker &       names );
```

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_shell.h*.

Функция выполняет построение тела, состоящего из грани, края которой проходят по заданным кривым. Для построения корректного тела кривые множества **initCurves** должны стыковаться друг с другом и образовывать замкнутый контур.

Если кривые **initCurves** лежат в одной плоскости, то заплатка будет представлять собой часть плоскости, лежащую внутри контура, образованного кривыми.

Если все кривые **initCurves** лежат на одной и той же криволинейной поверхности, то заплатка будет представлять собой часть общей поверхности, ограниченную заданными кривыми.

Если множество **initCurves** состоит из одной кривой, то эта кривая будет разбита на две части, а поверхность заплатки будет линейчатой поверхностью, имеющей два полюса.

Аналогично строится заплатка, если множество **initCurves** содержит две кривые – поверхность заплатки будет линейчатой поверхностью, имеющей два полюса.

Если количество кривых **initCurves** равно трем, то поверхность заплатки будет поверхностью MbCornerSurface, построенной по трем кривым и имеющей полюс.

Если количество кривых **initCurves** равно четырем, то поверхность заплатки будет поверхностью MbCoverSurface, построенной по четырем кривым. В других случаях возможен отказ от построения.

Объект **patchValues** содержит информацию о типе заплатки и флаге проверки самопересечений. Для инициализации объекта в структуре PatchValues по умолчанию задан следующий конструктор:

```
PatchValues()
  : type               ( ts_none  )
  , checkSelfInt       ( false    )
  , mergeEdges         ( true     )
  , tolerantData       ( false    )
  , internalMatingType ( pmt_None )
{ }
```

где

- **type** – тип заплатки;
- **checkSelfInt** – флаг проверки самопересечений;
- **mergeEdges** – флаг слияния подобных ребер (*true* – сливать подобные ребра);
- **tolerantData** – флаг построения неточной заплатки по неточным входным данным;

- **internalMatingType** – тип сопряжения между частями заплатки. Используется при **type** = *ts_byCurves*.

Для объекта **patchValues** определены следующие типы заплатки (см. перечисление SurfaceType):

- *ts_tang* – по касательной;
- *ts_norm* – по нормали;
- *ts_none* – не определено;
- *ts_plane* – плоская заплатка;
- *ts_byCurves* – построение задается сопряжениями по каждой кривой.

Типы *ts_tang*, *ts_norm* задаются при построении заплатки по касательной или по нормали соответственно.

Если задан тип заплатки *ts_plane*, будет построена заплатка с плоскими гранями.

Тип *ts_byCurves* задается, если требуется построить заплатку с сопряжением по некоторым границам. В этом случае для нужной границы MbPatchCurve нужно задать сопряжение методом SetMating().

На рис. 1 приведено восемь кривых, лежащих на краю внутреннего выреза грани. На рис. 2 приведена заплатка, построенная по кривым, приведенным на рис. 1:

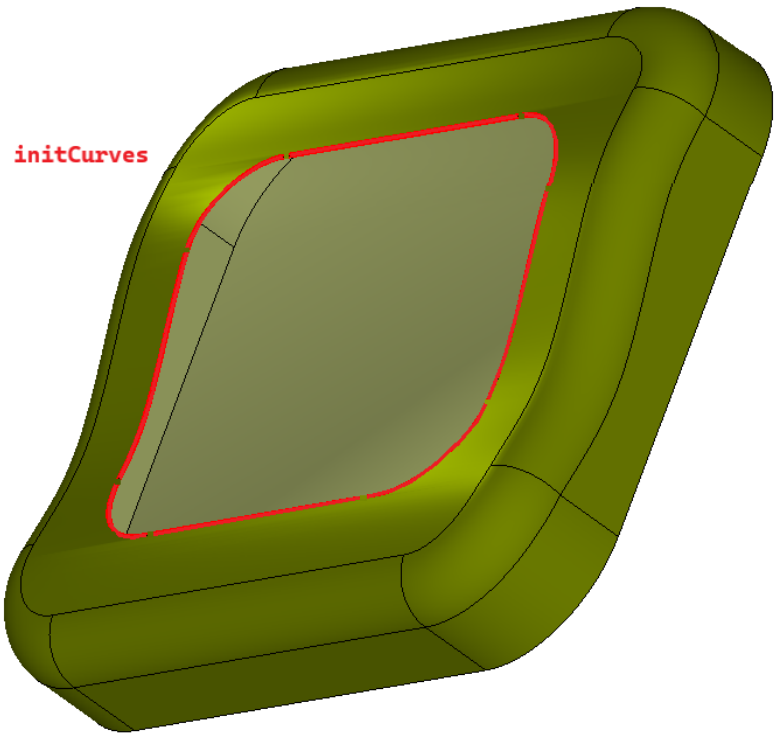


Рис. 1

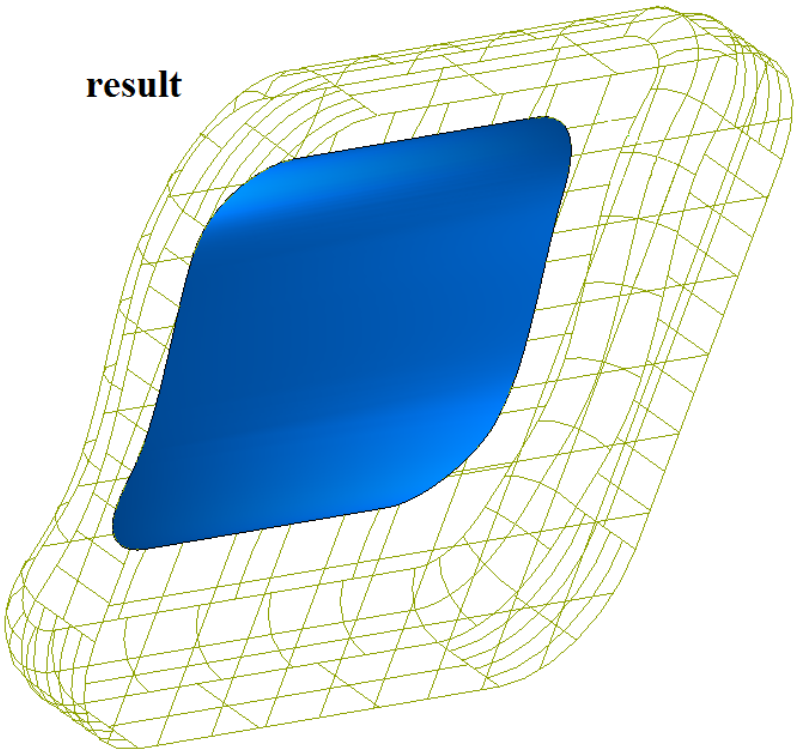


Рис. 2

На рис. 3 приведено шесть кривых, лежащих на краях разных граней, полученных путем резки граней общей поверхностью:

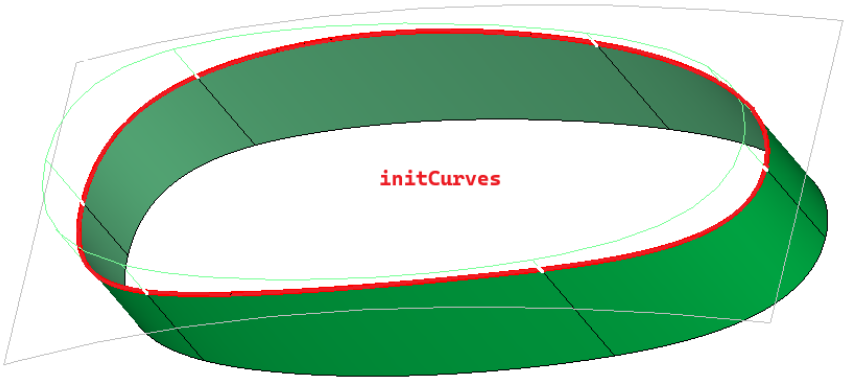


Рис. 3

Если после резки в краевых ребрах сохранилась информация о режущей поверхности, то при построении заплатки может быть использована режущая поверхность. На рис. 4 приведен пример построения заплатки в виде некоторой части режущей поверхности, которая образовала краевые ребра, приведенные на рис. 3:

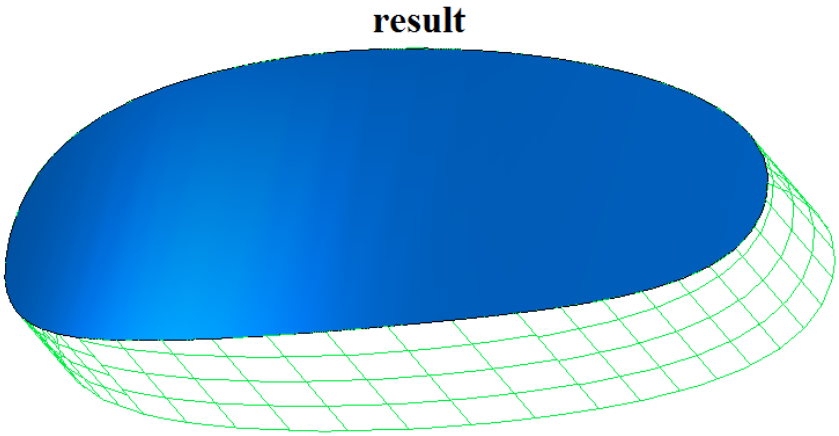


Рис. 4

Как правило, кривые для множества **initCurves** берутся от краевых ребер. После построения заплатку часто сшивают с другими телами, примыкающими к ее краям.

Функция PatchShell() добавляет в журнал построенного тела строитель MbPatchCreator. Строитель MbPatchCreator объявлен в файле *cr_patch_creator.h*.

В тестовом приложении *test.exe* построение заплатки можно реализовать командой меню «Создать→Оболочку→Заплатку».

3.1.10 Сшивка граней тел

Функция

```
MbeStitchResType StitchShells( const c3d::SolidsSPtrVector & initialSolids,
                               const MbShellStitchParams &   stitchParams,
                                     c3d::SolidSPtr &        resultSolid );
```

выполняет сшивку стыкующихся друг с другом граней нескольких тел в одно тело (ориентация граней может быть изменена). Функция объявлена в файле *action_shell.h*.

Входными параметрами функции являются:

initialSolids – множество тел для сшивки;

stitchParams – параметры сшивки оболочек, в которые входят следующие:

- **_nameMaker** – именователь операции;
- **_stitchAccuracy** – точность сшивки (точность поиска парных ребер);
- **_formSolidBody** – флаг формирования твердого тела из результирующей оболочки;
- **_mergeEdges** – слияние подобных ребер (true).

Выходным параметром функции является построенное тело или оболочка **resultSolid** (в зависимости от флага **_formSolidBody**). Если параметр **_formSolidBody** = *true*, то построенное тело **resultSolid** проверяется на отсутствие краевых ребер, выполняется поиск вложенных друг в друга оболочек и переориентация внутренних оболочек при необходимости. Если параметр **_formSolidBody** = *false*, то в построенном теле **resultSolid** допускается присутствие краевых ребер, а определение вложенности оболочек не выполняется.

Функция возвращает код результата операции из перечисления *MbeStitchResType* (см. *mb_operation_result.h*):

- **stch_Success** – операция выполнена успешно;
- **stch_PrepareError** – ошибка при подготовке операции сшивки;
- **stch_CoorientFaceError** – невозможно выставить согласованную ориентацию граней;
- **stch_SomeEdgesUnstitched** – некоторые рёбра остались несшитыми;
- **stch_OutwardOrientError** – не удалось установить нормали граней наружу тела;
- **stch_NoEdgeWasStitched** – не было сшито ни одного ребра;
- **stch_SeparatePartsResult** – после сшивки остались несвязанные между собой куски;
- **stch_EdgeStitchError** – ошибка сшивки ребра;
- **stch_InputTopologyError** – критические ошибки топологии во входных оболочках.

Функция выполняет поиск пар краевых ребер, соответствующих друг другу по длине и расположению, и строит кривую пересечения граней найденных ребер на заданном участке. После успешного построения кривой пересечения пара краевых ребер заменяется одним ребром, соединяющим смежные грани тел. При необходимости выполняется переориентация внешней стороны некоторых граней. Множество **initialSolids** может содержать любое конечное число тел.

На рис. 1 приведены два сшиваемых тела **solids** [0] и **solids** [1]. На рис. 2 приведено построенное тело **resultSolid**:

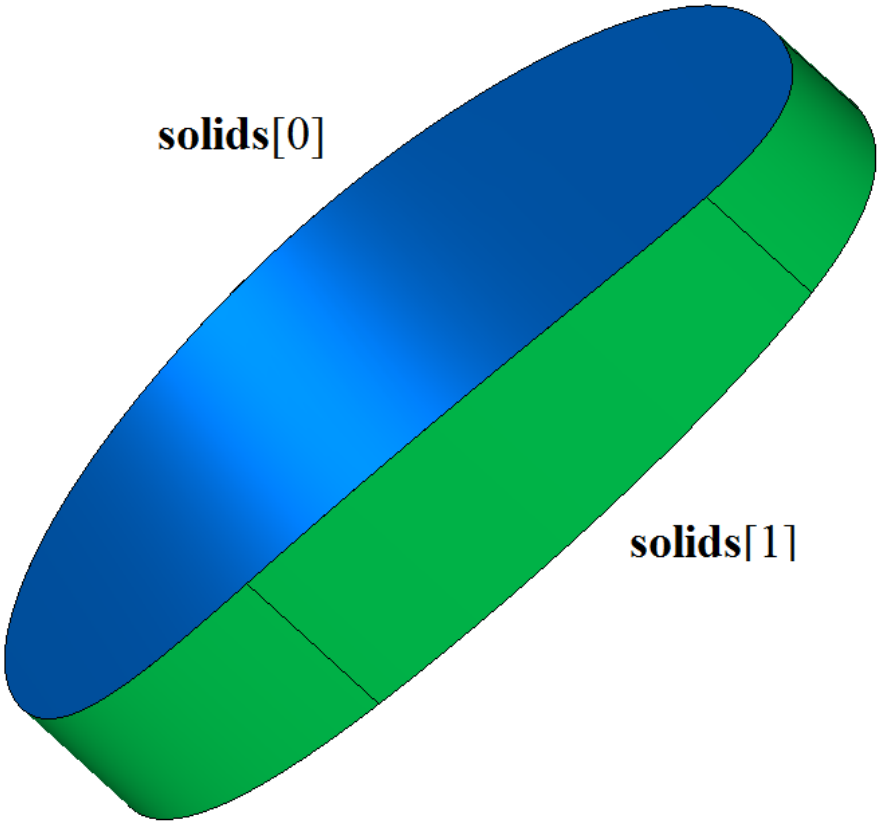


Рис. 1

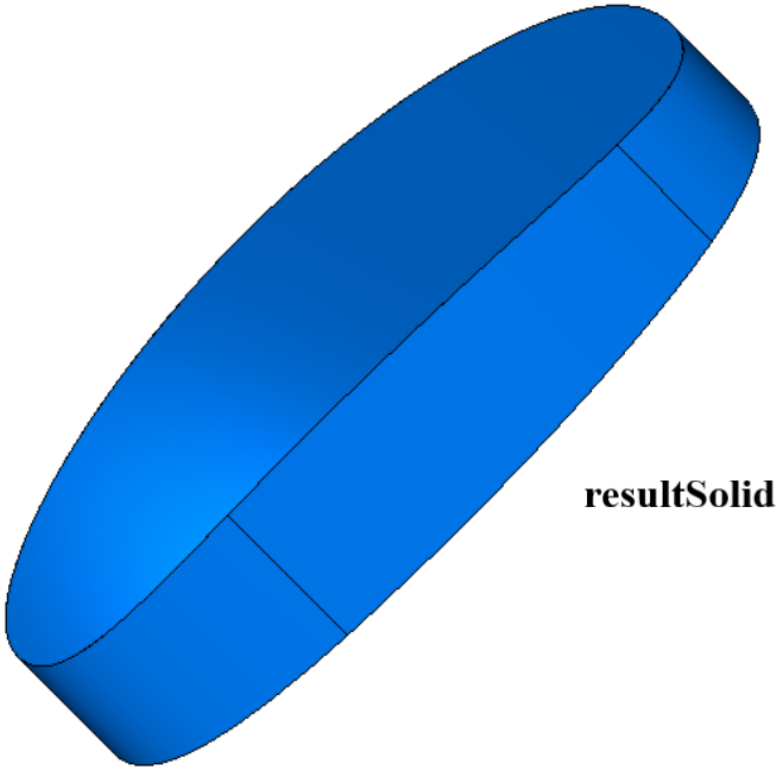


Рис. 2

Функция StitchShells() добавляет в журнал построенного тела строитель MbStitchedSolid, который содержит все необходимые данные для построения тела. Строитель MbStitchedSolid объявлен в файле *cr_stitch_solid.h*.

3.1.11 Построение тела по множеству кривых или групп точек

Построение тела по множеству кривых

Функция

```
MbResultType LoftedShell( const MbLoftedCurvesShellParams & params,
                                c3d::SolidSPtr &            result );
```

выполняет построение тела, состоящего из одной грани, поверхность которой проходит по семейству кривых.

Функция принимает в качестве входного параметра объект **params** класса MbLoftedCurvesShellParams, содержащий параметры, необходимые для операции (см. определение в *op_shell_parameter.h*):

- **curves** – набор кривых;
- **names** – именователь операции;
- **name** – идентификатор.

Выходным параметром функции является построенное тело **result**.

Для инициализации объекта **params** в классе MbLoftedCurvesShellParams определен конструктор:

```
template < class CurvesVector >
MbLoftedCurvesShellParams( const CurvesVector & curves,
                           const MbSNameMaker & names,
                           const SimpleName &   name );
```

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_shell.h*.

Поверхность построенного тела проходит по всем кривым **curves**. На форму поверхности тела влияет не только форма и взаимное расположение кривых, но и их ориентация и положение начал. Для предотвращения самопересечения и перекручивания поверхности тела соседние кривые множества **curves** должны иметь одинаковое направление, а начальные точки замкнутых кривых должны иметь близкое расположение. Если все кривые **curves** замкнуты, то поверхность построенного тела будет замкнута по одному из параметрических направлений.

Параметры **names** и **name** обеспечивают именование граней, ребер и вершин построенного тела.

На рис. 1 приведено множество кривых **curves**:

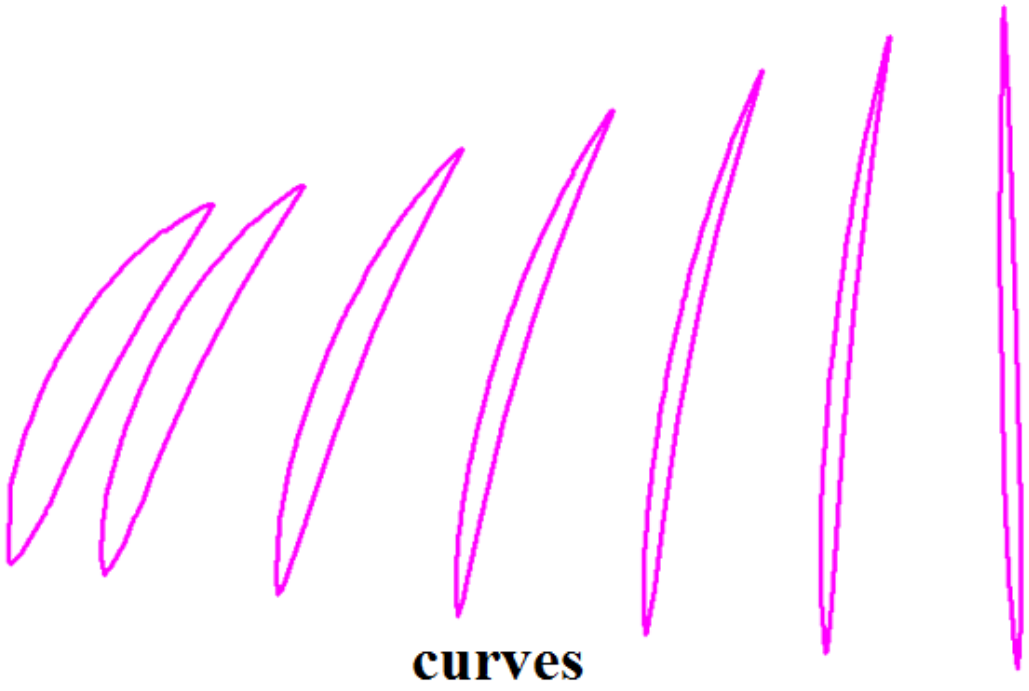


Рис. 1

На рис. 2 приведено тело, построенное по кривым, приведенным на рис. 1:

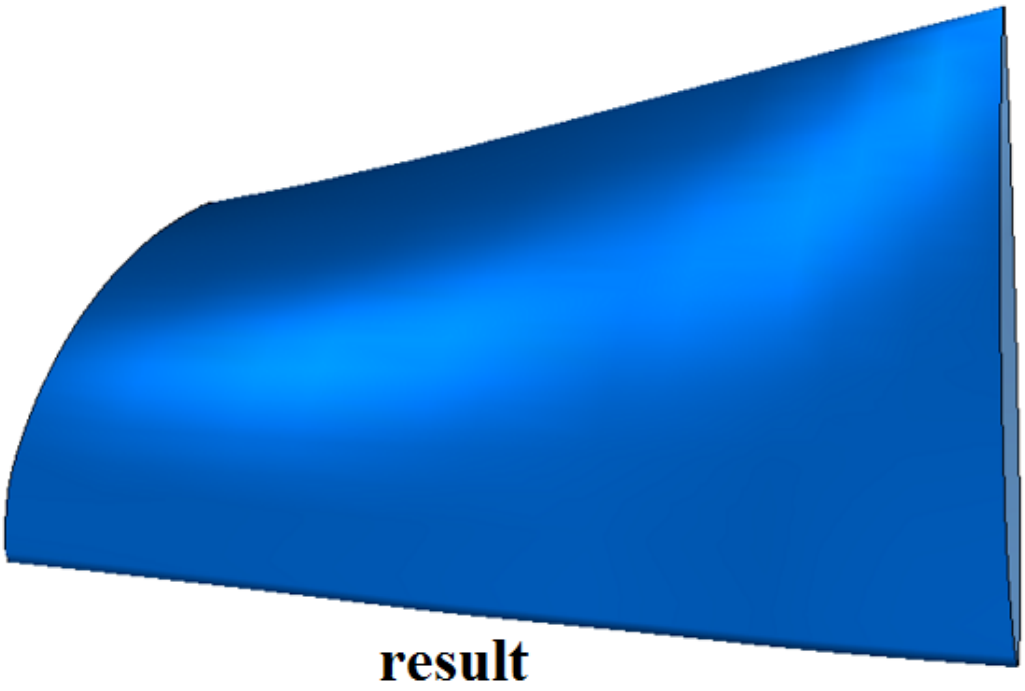


Рис. 2

Построение тела по множеству групп точек

Функция

```
MbResultType LoftedShell( const MbLoftedPointsShellParams & params,
                                c3d::SolidSPtr &            result );
```

выполняет построение незамкнутого тела из одной грани, поверхность которой проходит по контрольным точкам.

Функция принимает в качестве входного параметра объект **params** класса MbLoftedPointsShellParams, содержащий параметры, необходимые для операции (см. определение в *op_shell_parameter.h*):

- **points** – множество точек;
- **names** – именователь;
- **name** – идентификатор.

Выходным параметром функции является построенное тело **result**.

В классе MbLoftedPointsShellParams определены два конструктора по наборам точек, именователю и идентификатору:

```
MbLoftedPointsShellParams( const RPArray<SArray<MbCartPoint3D>> & points,
                           const MbSNameMaker &                   names,
                           const SimpleName &                     name );
```

```
MbLoftedPointsShellParams( const std::vector<c3d::SpacePointsVector> & points,
                           const MbSNameMaker &                        names,
                           const SimpleName &                          name );
```

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_shell.h*.

Контрольные точки сгруппированы по контейнерам *SArray* или в векторе *c3d::SpacePointsVector* в зависимости от типа используемого конструктора. Перед построением тела выполняется построение сплайновых кривых по множеству точек. Далее выполняется построение тела по семейству кривых, описанное в разделе «Построение тела по множеству кривых», функцией LoftedShell().

Поверхность построенного тела проходит по всем точкам **points**. На форму поверхности тела влияет не только взаимное расположение точек, но и их последовательность в группах. Для предотвращения самопересечения и перекручивания поверхности тела последовательности точек в соседних группах должны иметь одинаковое направление, а начальные точки в соседних группах должны иметь близкое расположение. Для построения замкнутой по одному из параметрических направлений поверхности первые и последние точки в каждом контейнере и векторе должны совпадать.

Параметры **names** и **name** обеспечивают именование граней, ребер и вершин построенного тела.

На рис. 3 приведено множество контрольных точек **points**, группы которых имеют разный цвет:

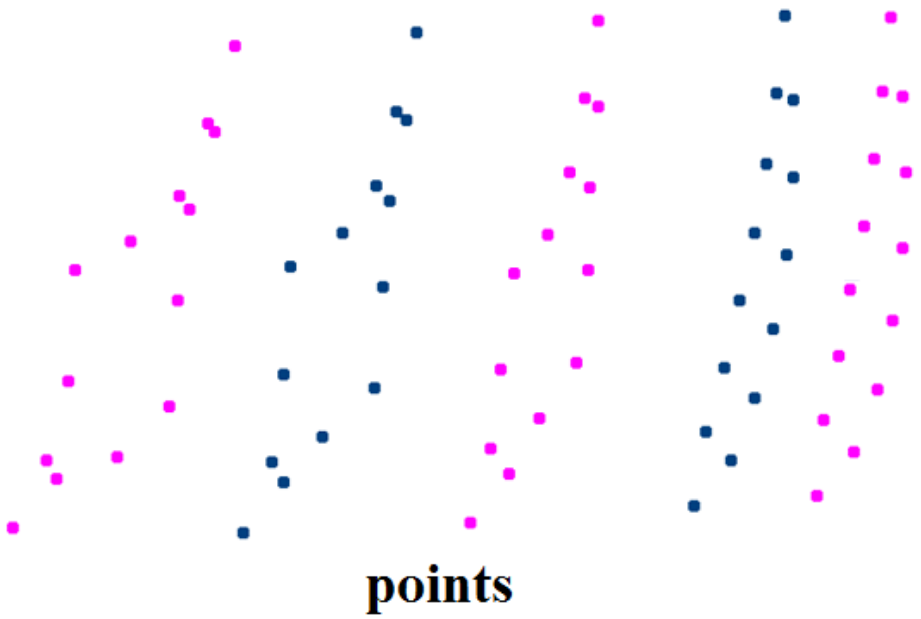


Рис. 3

На рис. 4 приведено тело и контрольные точки, по которым построено тело:

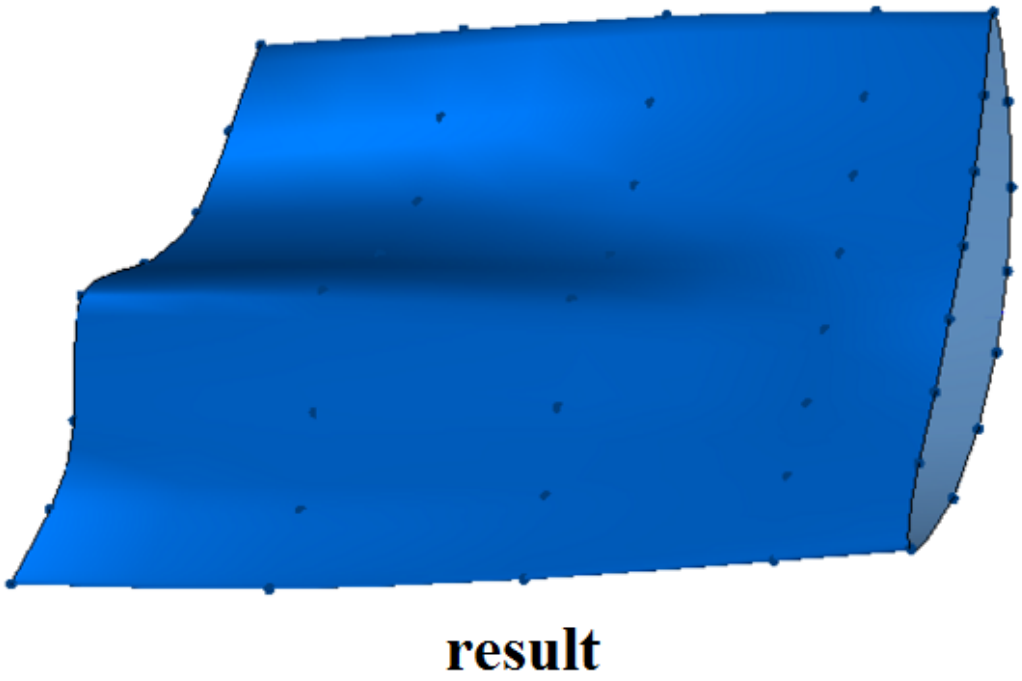


Рис. 4

Функция LoftedShell() добавляет в журнал построенного тела строитель MbThinShellCreator, который содержит все необходимые данные для построения тела. Строитель MbThinShellCreator объявлен в файле *cr_thin_sheet.h*.

3.1.12 Построение эквидистантного тела

Функция

```
MbResultType OffsetShell(       MbSolid &              solid,
                                MbeCopyMode            sameShell,
                                RPArray<MbFace> &      initFaces,
                                bool                   checkFacesConnection,
                                SweptValues &          p,
                          const MbSNameMaker &         operNames,
                                bool                   copyFaceAttrs,
                                MbSolid *&             result );
```

выполняет построение эквидистантного тела для множества граней.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – режим копирования исходного тела;
- **initFaces** – множество исходных граней тела, для которых выполняется операция построения эквидистантного тела;
- **checkFacesConnection** – флаг проверки связности выбранных граней:

true – операция проверяет, что все грани в **initFaces** образуют одну связную область, при нарушении связности возвращается код ошибки и эквидистантное тело не строится;

false – проверка не выполняется, что может привести к нескольким разрозненным участкам результата или к неуспеху операции;

- **p** – объект, содержащий параметры построения;
- **operNames** – именователь;
- **copyFaceAttrs** – флаг копирования атрибутов из исходных граней в эквидистантные.

Выходным параметром функции является построенное эквидистантное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType().

Функция объявлена в файле *action_shell.h*.

Функция строит тело, грани которого эквидистантны граням **initFaces** исходного тела **solid**.

Параметр **sameShell** управляет передачей граней, ребер и вершин от исходного тела **solid** построенному телу **result**. Данный параметр может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Детальное описание перечисления MbeCopyMode находится в параграфе *Копирование множества граней MbFaceShell*.

Параметр **p** содержит информацию о расстоянии между гранями построенного тела и гранями исходного тела. Расстояние может быть равным **p**.*thickness1* в положительном направлении или **p**.*thickness2* в отрицательном направлении нормали граней исходного тела.

Если **p**.*shellClosed* = *false*, то будет построено незамкнутое тело.

Параметр **p**.*checkSelfInt* сообщает о необходимости проверки результата построения на самопересечение. По умолчанию **p**.*checkSelfInt* = *false* – проверка не выполняется.

Флаги **p**.*mergeFlags* управляют слиянием подобных граней и ребер (*true* - сливать подобные элементы).

Параметр **operNames** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 1 приведено тело, на базе указанных граней которого будет построено эквидистантное тело:

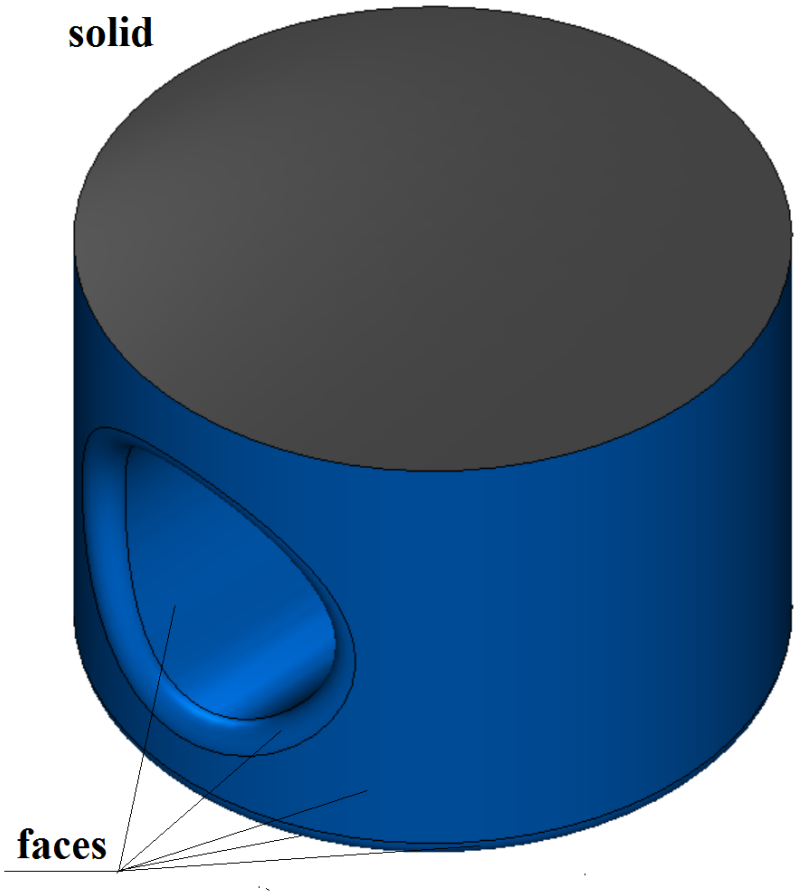


Рис. 1

На рис. 2 приведено построенное эквидистантное тело:

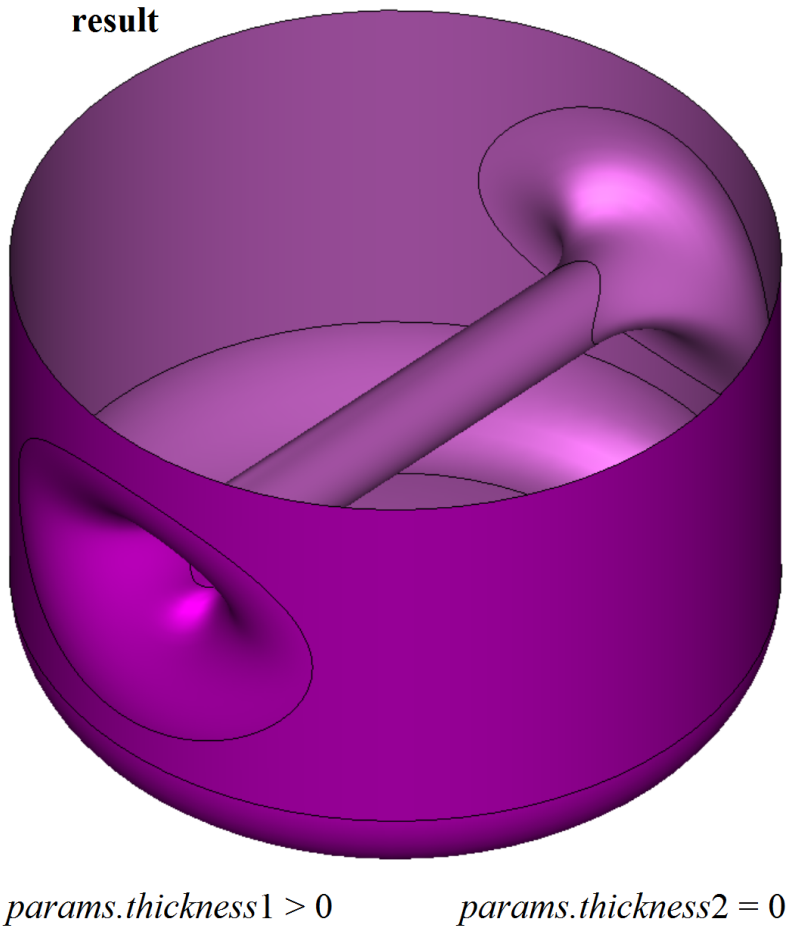


Рис. 2

Если множество граней **initFaces** будет содержать все грани исходного тела **solid**, то построенное тело **result** будет эквидистантным исходному телу **solid**.

Функция OffsetShell() добавляет в журнал построенного тела строитель MbShellSolid, который содержит все необходимые данные для построения тела. Строитель MbShellSolid объявлен в файле *cr_thin_shell_solid.h*.

В тестовом приложении *test.exe* построение эквидистантного незамкнутого тела осуществляется командой меню «Создать→Тело→На базе тела→Эквидистантное».

3.1.13 Продление грани тела

Функция

```
MbResultType ExtensionShell(       MbSolid &                  solid,
                                   MbeCopyMode                sameShell,
                             const MbExtensionShellParams &   params,
                                   c3d::SolidSPtr &           result );
```

выполняет построение продолжения незамкнутого тела путём продления краевых рёбер заданной грани тела.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – режим копирования исходного тела;
- **params** – объект, содержащий необходимые параметры для операции.

В классе MbExtensionShellParams для инициализации объектов **params** определены два конструктора по параметрам операции:

Конструктор параметров продолжения незамкнутого тела выдавливанием по заданному набору ребер:

```
MbExtensionShellParams( const c3d::EdgesSPtrVector & edges,
                        const ExtensionValues &      params,
                        const MbSNameMaker &         operNames );
```

Конструктор параметров продолжения незамкнутого тела выдавливанием без контейнера рёбер (ребра указываются через соответствующие методы MbExtensionShellParams::Add):

```
MbExtensionShellParams( const ExtensionValues & params,
                        const MbSNameMaker &    operNames );
```

- **edges** – множество краевых ребер, через которые выполняется продление;
- **params** – параметры операции;
- **operNames** – именователь.

Выходным параметром функции является построенное тело **result**. **result** всегда новое тело (не модификация исходного **solid**);

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_shell.h*.

Параметр **sameShell** управляет передачей граней, ребер и вершин от исходного тела **solid** построенному телу **result**. Параметр **sameShell** может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

Объект **params** класса ExtensionValues управляет способами продления граней и содержит данные, приведенные на рис. 1:

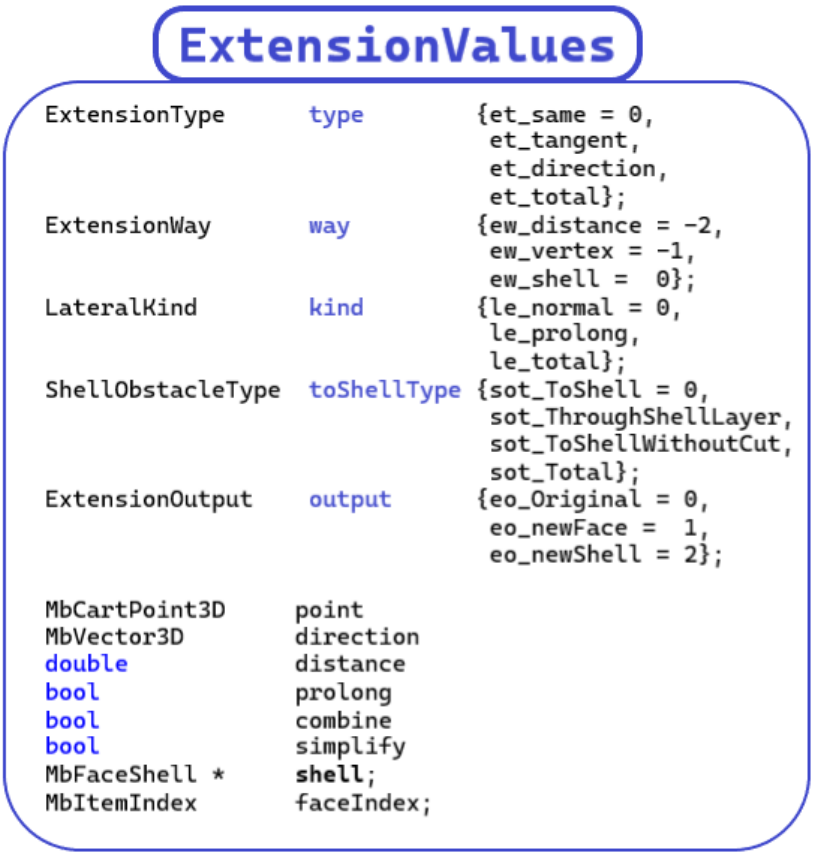


Рис. 1

Продление грани может быть трех типов. Тип продления определяет параметр **params**.*type*:

- При **params**.*type* = *et_same* продляется исходная грань путем расширения области определения параметров поверхности и смещения краевых ребер **edges**;
- При **params**.*type* = *et_tangent* к грани гладко пристыковываются новые грани через ребра **edges**;
- При **params**.*type* = *et_direction* к грани через ребра **edges** пристыковываются новые грани, полученные выдавливанием кривых ребер **edges** в направлении **params**.*direction*.

Параметр **params**.*way* определяет расстояние продления края грани:

- При **params**.*way* = *ew_distance* продление выполняется на расстояние **params**.*distance*;
- При **params**.*way* = *ew_vertex* продление выполняется до вершины, определяемой точкой **params**.*point*;
- При **params**.*way* = *ew_shell* продление выполняется до оболочки **params**.*shell*.

Если **params**.*way* ≠ *ew_shell*, то **params**.*shell* может быть равен нулю.

Параметр **params**.*faceIndex* задаёт индекс грани в оболочке **params**.*shell*, используемой при продлении до оболочки.

Параметр **params**.*kind* определяет способ обрезки краев граней продления в начале и в конце каждого ребра множества **edges**:

- Если **params**.*kind* = *le_normal*, то край продления грани обрезается по нормали к соответствующему ребру множества **edges**.
- Если **params**.*kind* = *le_prolong*, то край продления грани определяется соседними к ребрам множества **edges** ребрами.

Параметр **params**.*output* определяет варианты представления результирующего тела:

- Если **params**.*output* = *eo_Original*, то удлиняются исходные грани тела;
- Если **params**.*output* = *eo_newFace*, то результат продления формируется в виде новых граней;
- Если **params**.*output* = *eo_newShell*, то результат продления формируется в виде новой оболочки.

Параметр **params**.*toShellType* определяет способ продления до оболочки:

- Если **params**.*toShellType* = *sot_ToShell*, то продление осуществляется до оболочки;
- Если **params**.*toShellType* = *sot_ThroughShellLayer*, то продление осуществляется через первый слой оболочки (до первого выхода из оболочки);
- Если **params**.*toShellType* = *sot_ToShellWithoutCut*, то продление осуществляется до оболочки без подрезки.

Параметр **operNames** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 2 приведена грань тела, два краевых ребра которого будут продлены:

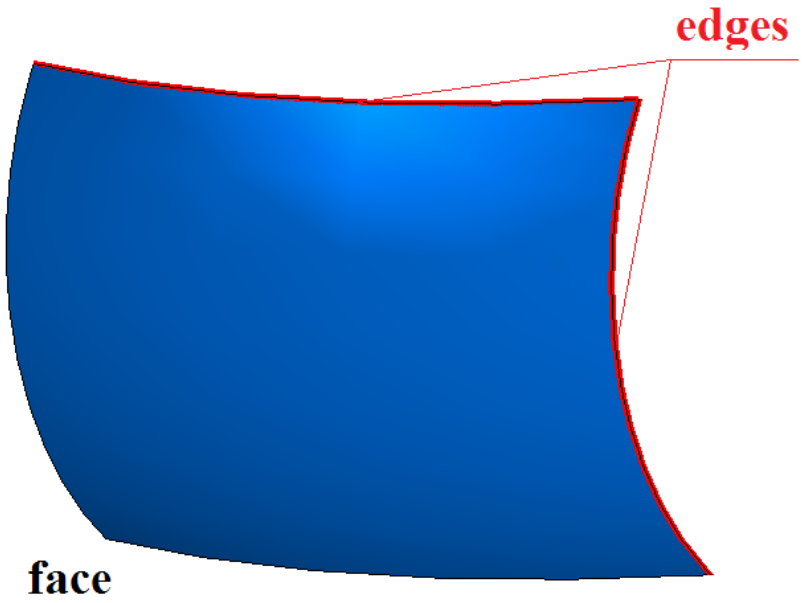


Рис. 2

На рис. 3 и 4 приведены результаты продления грани путем расширения области определения параметров поверхности с разными значениями параметра **params**.*kind*:

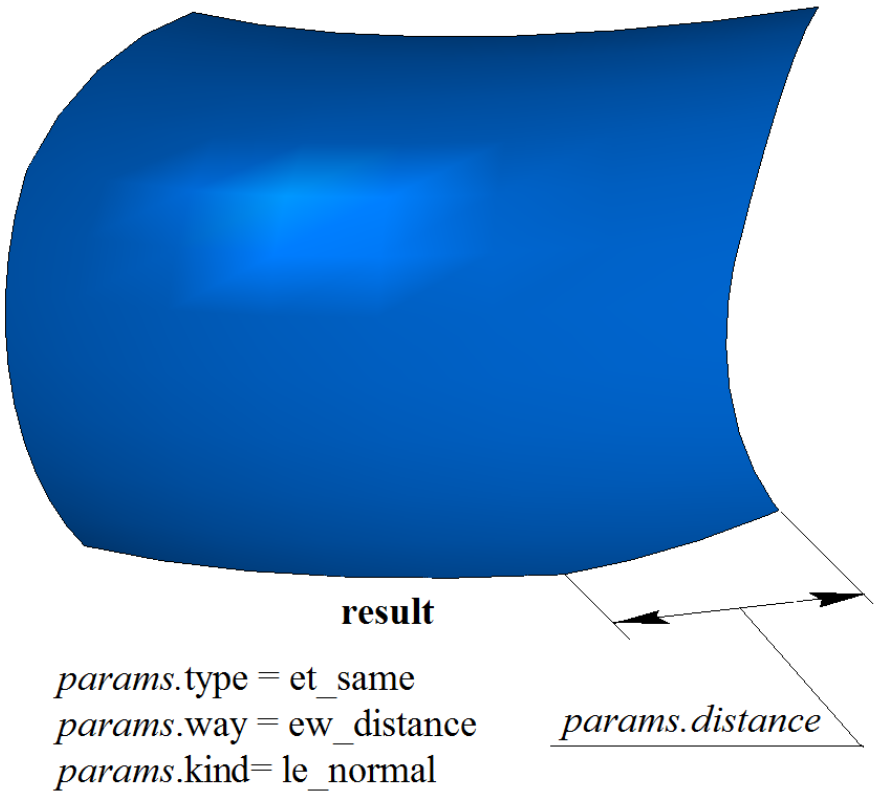


Рис. 3

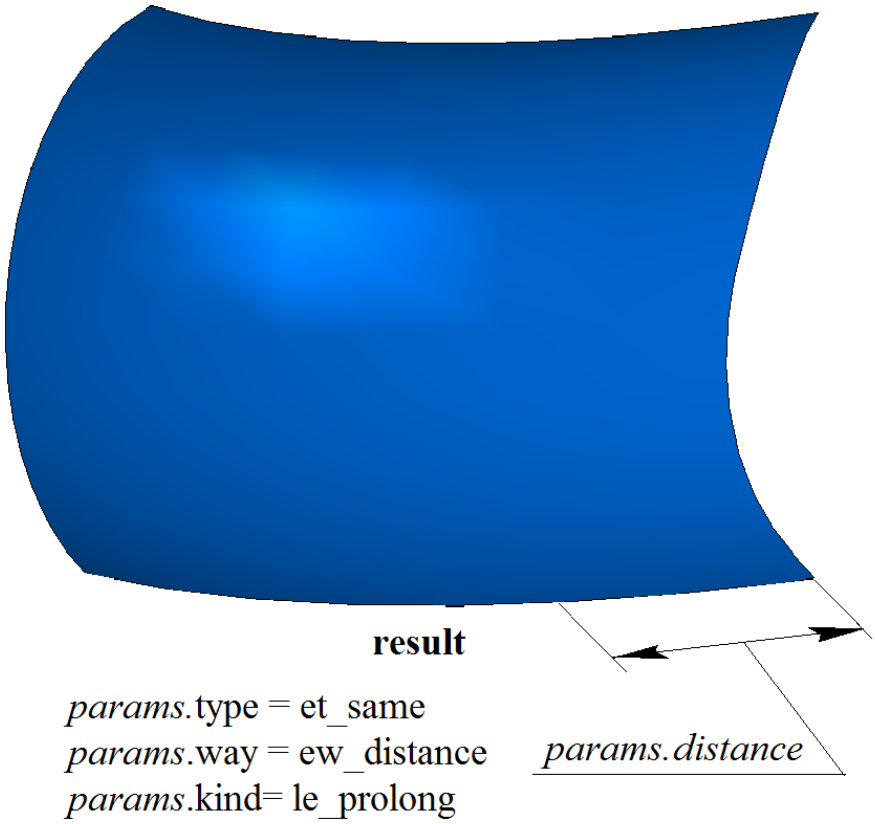


Рис. 4

На рис. 5 приведен результат продления грани **face** добавлением новых граней, гладко стыкующихся с гранью **face**. На рис. 6 приведен результат продления грани **face** в заданном направлении.

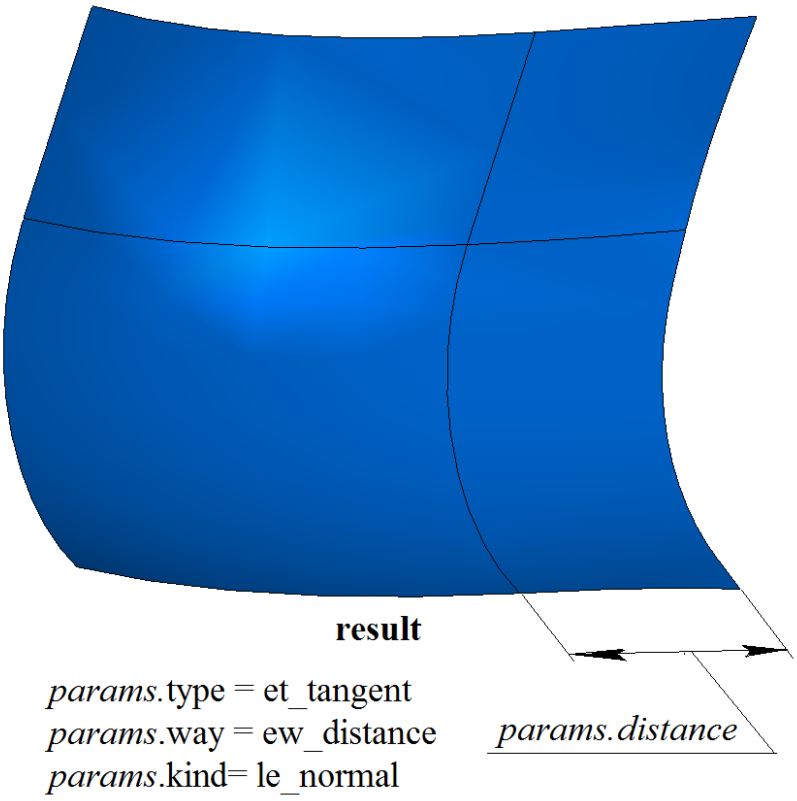

result
params.type = et_tangent
params.way = ew_distance
params.kind= le_normal
params.distance

Рис. 5

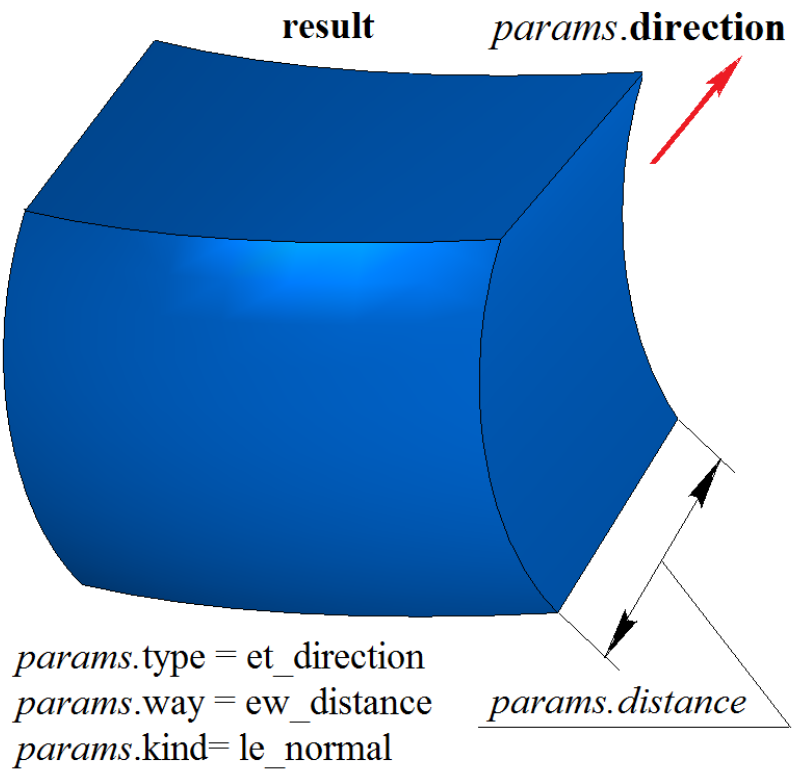


Рис. 6

Функция ExtensionShell() добавляет в журнал построенного тела строитель MbExtensionShell, который содержит все необходимые данные для построения тела. Строитель MbExtensionShell объявлен в файле *cr_extension_shell.h*.

3.1.14 Построение тела-слитка из заданных объектов

Функция

```
MbResultType IngotSolid(      bool                    makeCopy,
                        const MbIngotSolidParams &    params,
                              c3d::SolidSPtr &        resIngot );
```

выполняет построение тела-слитка из заданных объектов. Тело-слиток является упрощенным представлением заданного набора тел, предназначенным для ускорения расчётов, коллизий и т.п., и не предназначено для точной деталировки.

Заданными объектами могут быть тела (MbSolid), вставки тел (MbInstance) и сборки тел (MbAssembly). Из них строится одно тело, которое по внешнему виду совпадает с заданными телами и служит их упрощённым заменителем по внешним параметрам.

Входными параметрами функции являются:

- **makeCopy** – флаг копирования исходных тел;
- **params** – объект, содержащий параметры операции.

Выходным параметром функции является построенное тело **resIngot**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Для инициализации объекта **params** в классе MbIngotSolidParams определены конструкторы параметров создания тела-слитка из заданного набора тел (см. *op_shell_parameter.h*):

```
MbIngotSolidParams( const c3d::SolidSPtr &     solid,
                    const MbSNameMaker &       operNames,
                          IProgressIndicator * prog = nullptr );
```

```
MbIngotSolidParams( const c3d::ItemSPtr &      item,
                    const MbSNameMaker &       operNames,
                          IProgressIndicator * prog = nullptr );
```

```
MbIngotSolidParams( const c3d::SolidsSPtrVector & solids,
                    const MbSNameMaker &          operNames,
                          IProgressIndicator *    prog = nullptr );
```

```
MbIngotSolidParams( const c3d::ItemsSPtrVector & items,
                    const MbSNameMaker &         operNames,
                          IProgressIndicator *   prog = nullptr );
```

```
MbIngotSolidParams( const MbSNameMaker &         operNames,
                          IProgressIndicator *   prog = nullptr );
```

- **solid** – заданное тело;
- **item** – заданное тело (может быть объектом классов MbAssembly, MbInstance или MbSolid);
- **solids** – заданный контейнер тел;
- **items** – заданный контейнер тел (может содержать объекты MbAssembly, MbInstance и MbSolid);
- **operNames** – именователь операции;
- **prog** – индикатор прогресса выполнения операции.

Функция выполняет объединение множества тел MbSolid, вставок тел MbInstance или сборок тел MbAssembly в одно тело **resIngot**.

На рис. 1 приведен заданный контейнер тел **solids**:

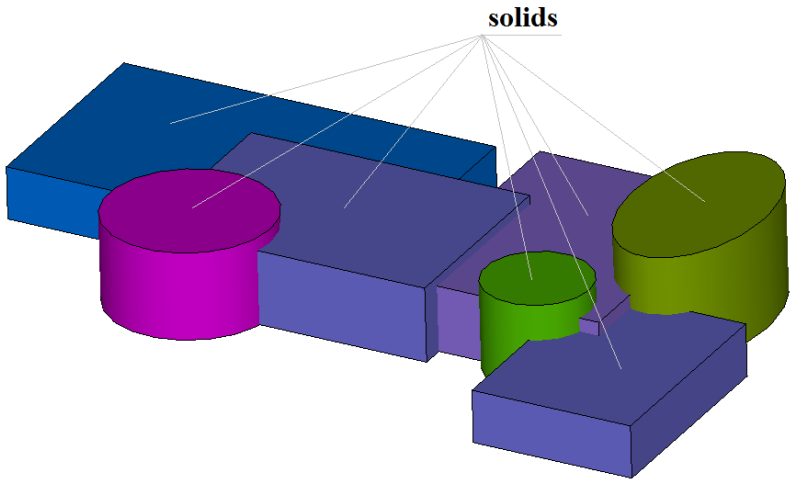


Рис. 1

На рис. 2 приведен результат **resIngot** объединения тел **solids**:

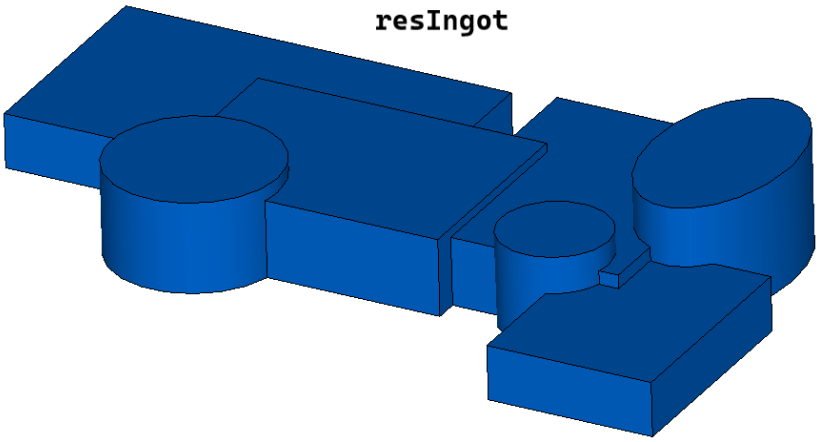


Рис. 2

В тестовом приложении *test.exe* построение тела-слитка выполняется командой меню «Создать → Тело → На базе тела → Набор тел».

3.1.15 Построение тела на основе полигональной модели

Функция

```
MbResultType MeshSolid( const MbMesh &                mesh,
                        const MbGridShellParams &     params,
                              c3d::SolidSPtr &        result );
```

выполняет построение тела MbSolid на основе полигональной модели MbMesh.

Входными параметрами функции являются:

- **mesh** – полигональная модель
- **params** – набор параметров операции (определение класса MbGridShellParams см. в *op_shell_parameter.h*)

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

В классе MbGridShellParams для инициализации объектов **params** определены два конструктора по параметрам:

```
MbGridShellParams( const MbSNameMaker &          operNames,
                         IProgressIndicator * & prog = nullptr )

: _sewGrids       ( true                   )
, _mergeFaces     ( false                  )
, _useGridSurface ( false                  )
, _copyAttributes ( false                  )
, _operNames      ( &operNames.Duplicate() )
, _progress       ( prog                   ){ },
```

```
MbGridShellParams(       bool sg,
                         bool mf,
                         bool ugs,
                         bool ca,
                   const MbSNameMaker &          operNames,
                         IProgressIndicator * & prog = nullptr )

: _sewGrids       ( sg                     )
, _mergeFaces     ( mf                     )
, _useGridSurface ( ugs                    )
, _copyAttributes ( ca                     )
, _operNames      ( &operNames.Duplicate() )
, _progress       ( prog                   ){ }.
```

- **_sewGrids** – флаг сшивки наборов граней от разных сеток триангуляции в единую оболочку (*true* – сшивать наборы граней от разных сеток триангуляции, *false* – каждая сетка определяет отдельную часть оболочки);
- Флаг **_mergeFaces** управляет слиянием *подобных* соседних граней для упрощения оболочки (*true* – сливать подобные грани);
- **_useGridSurface** – флаг использования поверхности на базе триангуляции (*true* – использовать поверхность на базе триангуляции, *false* – строить оболочку без создания поверхности на базе сетки);
- Флаг **_copyAttributes** управляет переносом атрибутов (цвета, имена, пользовательские данные) с граней MbMesh на грани результирующего тела (*true* – копировать атрибуты)
- **_operNames** – именователь создаваемых граней, рёбер и вершин в дереве построения.

Примечание

Подобными называются соседние грани, которые могут быть объединены без изменения геометрии оболочки. Например, подобными плоскими гранями являются грани, лежащие в одной плоскости и имеющие согласованную ориентацию нормалей.

На рисунках 1, 2 приведены полигональная модель **mesh** и тело **result**, построенное на её основе:

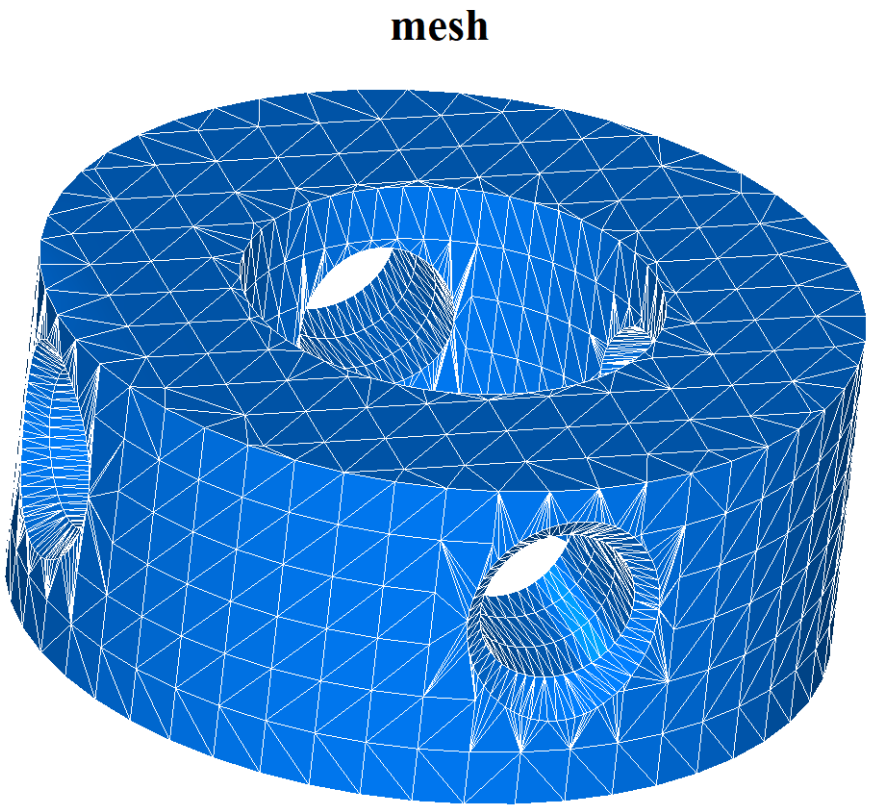


Рис. 1

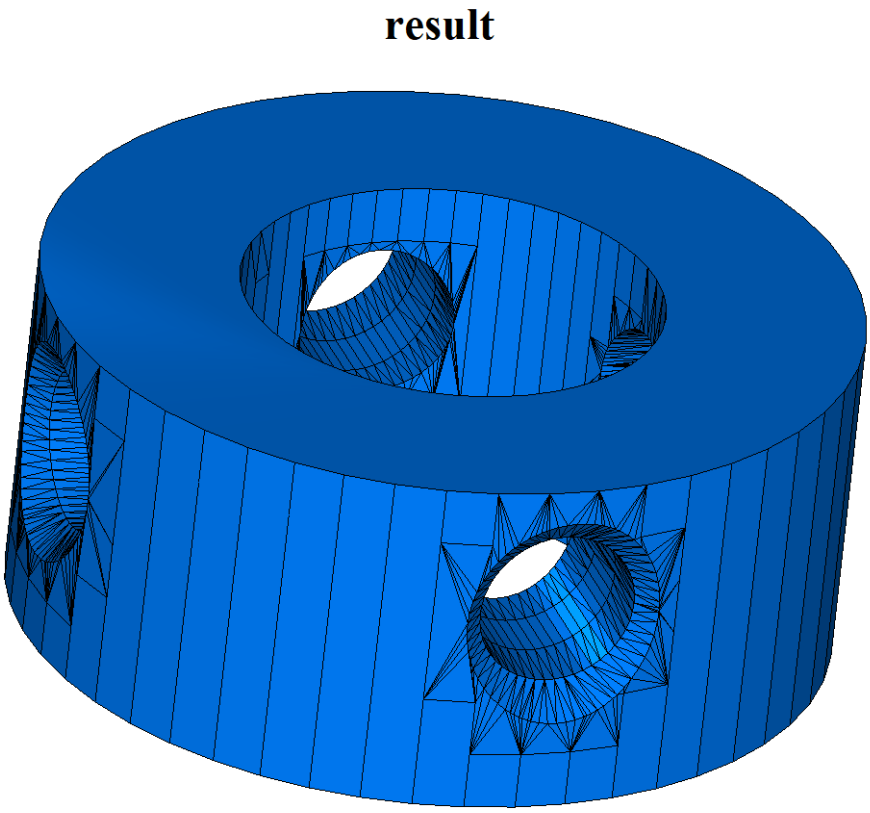


Рис. 2

Функция MeshSolid() добавляет в журнал построенного тела строитель MbSimpleMeshCreator, который содержит все необходимые данные для выполнения операции. Строитель MbSimpleMeshCreator объявлен в файле *cr_mesh_creator.h*.

В тестовом приложении *test.exe* построение тела на основе полигональной модели выполняется командой меню «Создать → Тело → На базе полигональной модели».

3.2 ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НАД ТЕЛАМИ

Один из подходов к построению тел в геометрическом моделировании похож на процесс изготовления моделируемого объекта. Сначала создаются тела простой формы, а далее выполняется набор действий, позволяющих из тел простой формы получить более сложные тела. Более сложные тела строятся путем выполнения операций над построенными телами. Операции фиксируются в журнале построения. Для замкнутых и незамкнутых тел одни и те же операции могут приводить к разным результатам.

3.2.1 Булева операция над телами

Функция

```
MbResultType BooleanResult( const c3d::SolidSPtr &          solid1,
                            MbeCopyMode                     sameShell1,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                              const c3d::SolidSPtr &           solid2,
                                    MbeCopyMode                sameShell2,
                              const MbBooleanOperationParams & params,
                                    c3d::SolidSPtr &           result );
```

выполняет построение нового тела путем булевой операции над двумя заданными телами с возможностью управления слиянием граней и рёбер.

Входными параметрами функции являются:

- **solid1** - первое тело для булевой операции;
- **sameShell1** - вариант копирования первого тела;
- **solid2** - второе тело для булевой операции;
- **sameShell2** - вариант копирования второго тела;
- **params** - параметры выполнения булевой операции.

Выходным параметром функции является объект **result**, содержащий построенное тело.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Функция выполняет операции объединения, пересечения и вычитания точек двух тел **solid1** и **solid2**.

Параметры **sameShell1** и **sameShell2** управляют передачей граней, ребер и вершин от исходных тел **solid1** и **solid2** построенному телу **result** и могут принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Детальное описание перечисления MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

Класс MbBooleanOperationParams содержит параметры выполнения булевой операции. При создании объекта класса MbBooleanOperationParams используется один из следующих конструкторов:

```
MbBooleanOperationParams(      OperationType        oType,
                         const MbBooleanFlags &     flags,
                         const MbSNameMaker &       operNames,
                               ProgressIndicator *  progress = nullptr );


MbBooleanOperationParams(      OperationType        oType,
                               bool                 closed,
                         const MbSNameMaker &       operNames,
                               IProgressIndicator * progress = nullptr );
```

где

- **oType** – тип булевой операции;
- **flags** – управляющие флаги булевой операции;
- **operNames** – именователь;
- **closed** – показатель замкнутости оболочек операндов;
- **progress** – индикатор прогресса выполнения операции.

Вариант объекта MbBooleanOperationParams с параметром **closed** работает только для незамкнутых тел и сообщает о необходимости выполнить проверку результата на замкнутость.

Параметр **oType** определяет тип булевой операции и может принимать три значения: *bo_Union*, *bo_Intersect*, *bo_Difference*:

- при **oType** = *bo_Union* рассматриваемая функция выполняет объединение тел **solid1** и **solid2**;
- при **oType** = *bo_Intersect* рассматриваемая функция выполняет пересечение тел **solid1** и **solid2**;
- при **oType** = *bo_Difference* рассматриваемая функция выполняет вычитание из тела **solid1** тела **solid2**.

Параметр **operNames** используется для именования результирующих объектов булевой операции.

Параметр **flags** содержит управляющие флаги булевой операции.

В объекте имеются следующие флаги, которые могут быть установлены в конструкторе при создании объекта, а также с помощью отдельных функций установки:

bool **mergeFaces** – сливать подобные грани;

bool **mergeEdges** – сливать подобные ребра;

bool **closed** – замкнутость оболочек операндов;

bool **enclosureCheck** – проверять оболочки на вложенность;

bool **allowNonIntersecting** – выдавать конечную оболочку, если нет пересечений;

bool **cutting** – флаг резки оболочки при построении разрезов и сечений;

bool **repairShellEdges** – флаг починки ребер исходных оболочек.

На рис. 1 приведены исходные тела **solid1** и **solid2**:

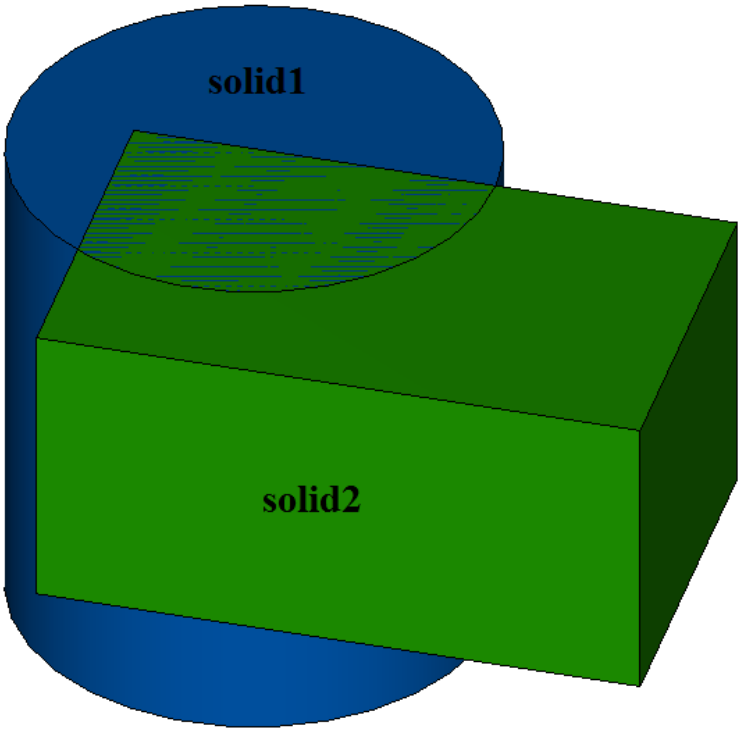
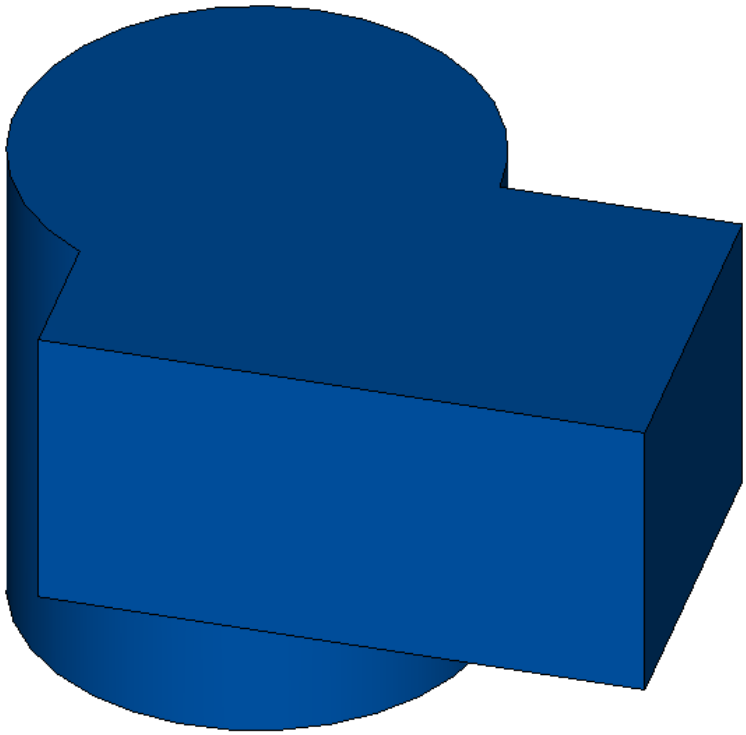


Рис. 1

На рис. 2 приведен результат булевой операции объединения тел **solid1** и **solid2**, показанных на рис. 1:

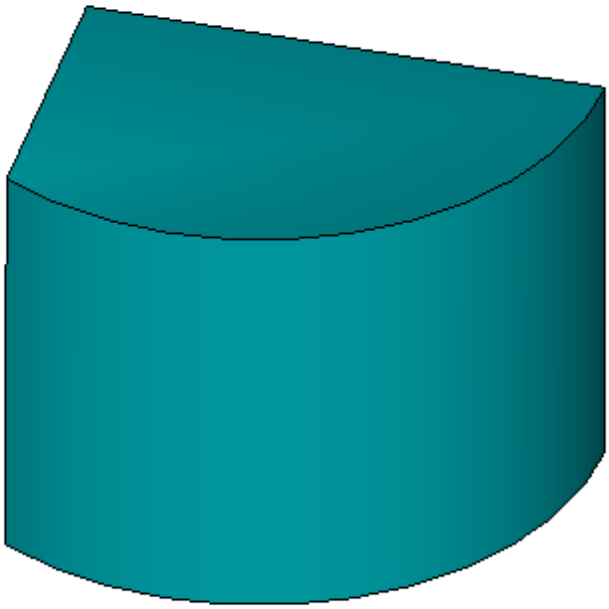


$oType = \text{bo\_Union}$

solid1 + solid2

Рис. 2

На рис. 3 приведен результат булевой операции пересечения тел **solid1** и **solid2**, показанных на рис. 1:

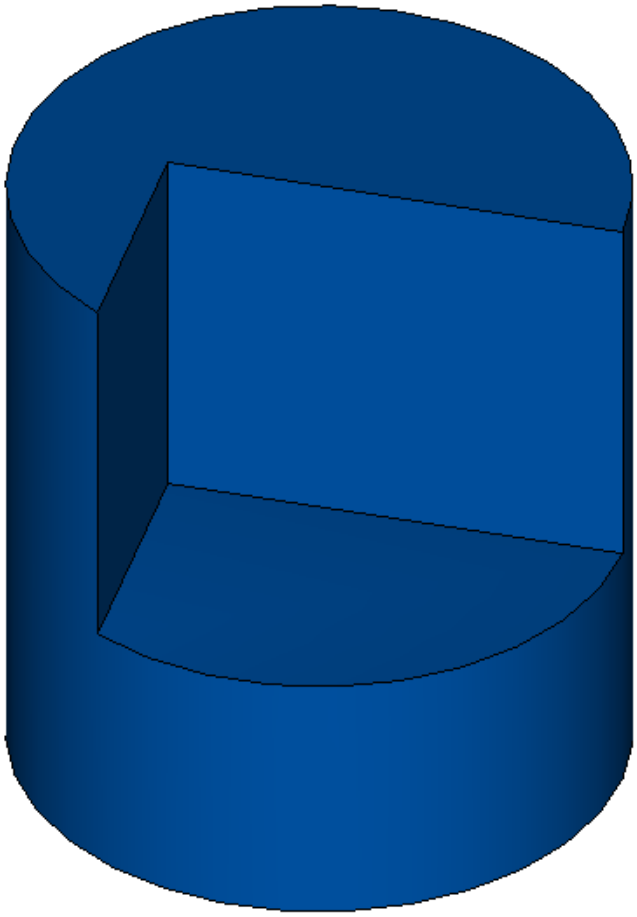


$oType = \text{bo\_Intersect}$

solid1&solid2

Рис. 3

На рис. 4 приведен результат булевой операции вычитания тела **solid2** из тела **solid1**, показанных на рис. 1:

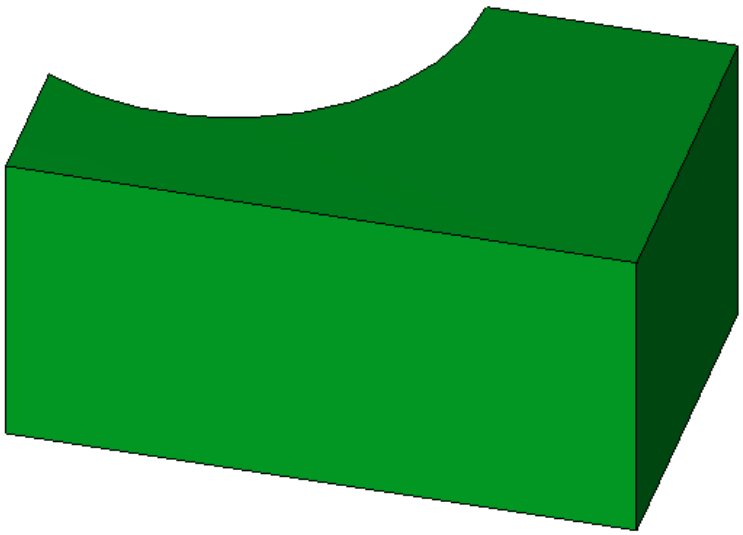


oType = bo_Difference

solid1 - solid2

Рис. 4

На рис. 5 приведен результат булевой операции вычитания тела **solid1** из тела **solid2**, показанных на рис. 1:



oType = bo_Difference

solid2 - solid1

Рис. 5

Для демонстрации работы функции с переданным объектом MbBooleanFlags с различными значениями флага *mergeFaces* рассмотрим булевы операции над исходными телами **solid1** и **solid2**, приведенными на рис. 6:

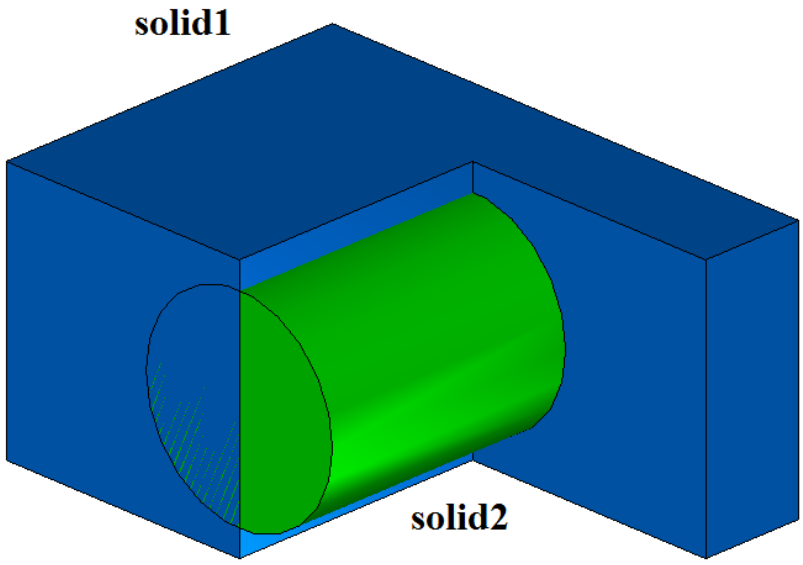


Рис. 6

На рис. 7 приведен результат **result** объединения тел **solid1** и **solid2** при работе рассматриваемой функции с флагом *mergeFaces* = *true*:

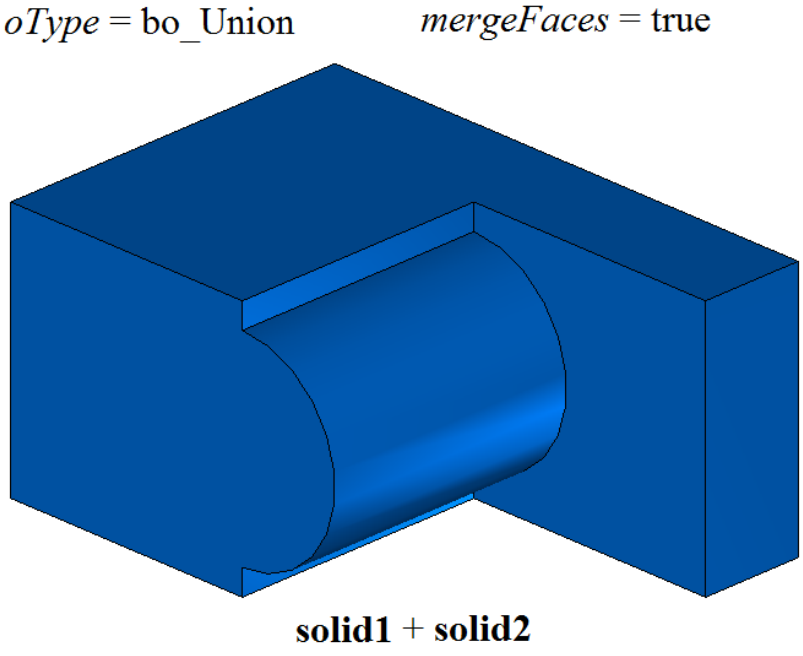


Рис. 7

На рис. 8 приведен результат **result** объединения тел **solid1** и **solid2** при работе рассмат-

риваемой функции с флагом *mergeFaces* = *false*. Совпадающие грани на рис. 8 не объединены:

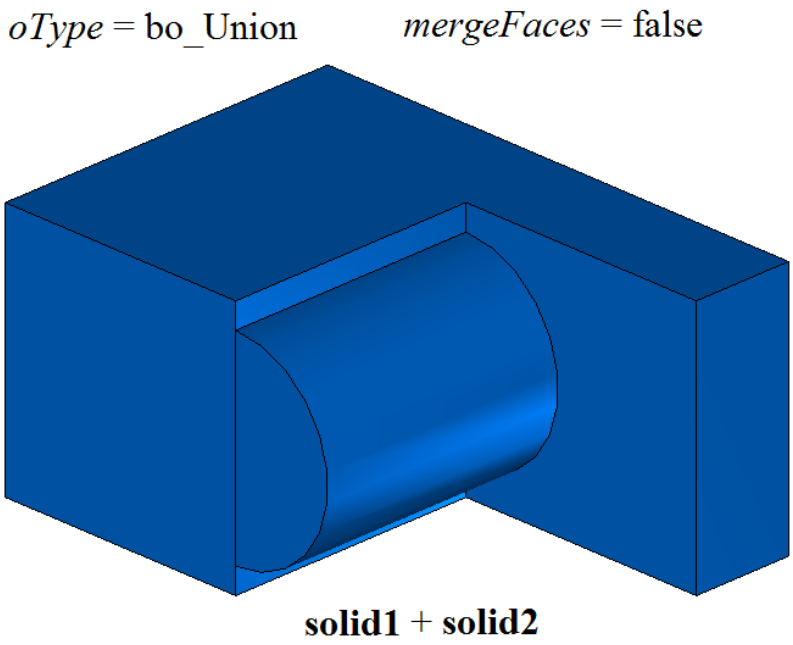


Рис. 8

На рис. 9 приведен результат **result** вычитания тела **solid2** из тела **solid1** при работе рассматриваемой функции с параметром *mergeFaces* = *true* (сливать подобные грани):

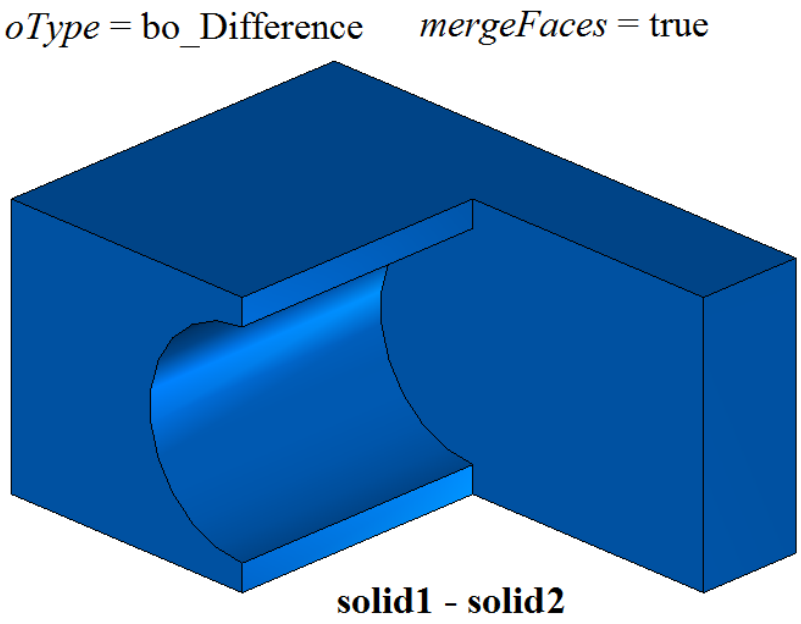


Рис. 9

На рис. 10 приведен результат **result** вычитания тела **solid2** из тела **solid1** при работе рассматриваемой функции с параметром *mergeFaces* = *false*. Совпадающие грани на рис. 10 не объединены.

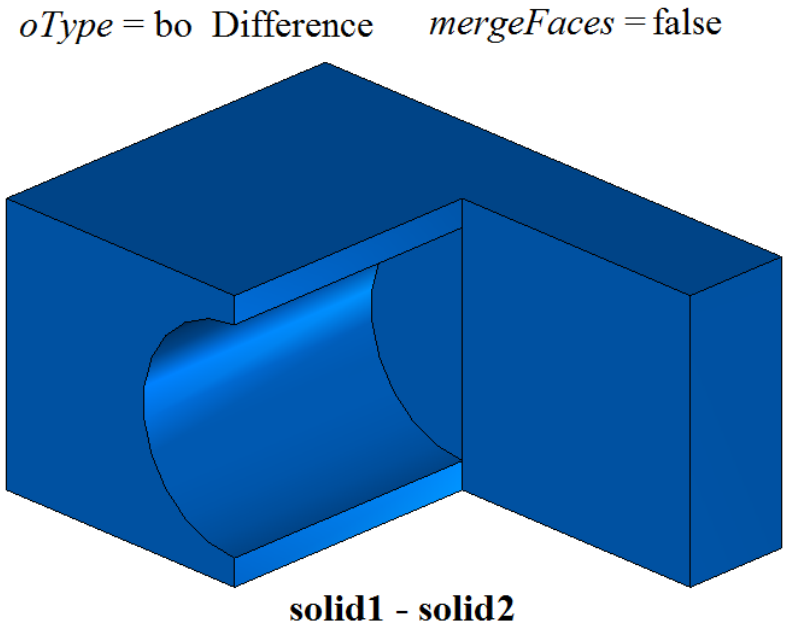


Рис. 10

Функция BooleanResult() добавляет в журнал построенного тела строитель MbBooleanSolid, который содержит все необходимые данные для выполнения операции. Строитель MbBooleanSolid объявлен в файле *cr_boolean_solid.h*.

В тестовом приложении *test.exe* булевы операции над телами выполняются командами меню «Создать→Тело→Приклеиванием к телу→Тела», «Создать→Тело→Вырезанием из тела→Тела», «Создать→Тело→Пересечением с телом→Тела».

3.2.2 Булева операция с телом выдавливания

```
MbResultType ExtrusionResult(      MbSolid &                solid,
                                   MbeCopyMode              sameShell,
                             const MbSweptData &            sweptData,
                             const MbVector3D &             direction,
                             const ExtrusionValues &        params,
                                   OperationType            oType,
                             const MbSNameMaker &           operNames,
                             const RPArray<MbSNameMaker> &  contoursNames,
                                   MbSolid *&               result );
```

выполняет построение тела выдавливания и булеву операцию заданного тела с построенным телом.

Входными параметрами функции являются:

- **solid** – заданное тело для булевой операции;

- **sameShell** – режим копирования заданного тела;
- **sweptData** – данные об образующих кривых для построения тела выдавливания;
- **direction** – направление выдавливания;
- **params** – параметры построения, возвращают информацию для построения элементов массива операций до поверхности;
- **oType** – тип булевой операции:

  bo_Union – объединение тел,

  bo_Intersect – пересечение тел,

  bo_Difference – вычитание тел;
- **operNames** – именователь операции;
- **contoursNames** – именователи образующих кривых.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Функция представляет собой последовательное объединение двух функций: функции ExtrusionSolid(), выполняющей построение тела путем выдавливания кривых **sweptData** по заданным параметрам **params** в направлении **direction**, и функции BooleanResult(), выполняющей булеву операцию **oType** тела **solid** с построенным на предыдущем шаге телом.

Функция ExtrusionSolid() описана в параграфе *Построение тела выдавливания*, функция BooleanResult() описана в параграфе *Булева операция над телами*.

Параметр **sameShell** управляет передачей граней, ребер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **oType** определяет тип булевой операции и может принимать три значения: *bo_Union*, *bo_Intersect*, *bo_Difference*.

При **oType** = *bo_Union* рассматриваемая функция выполняет объединение тел **solid** и тела выдавливания.

При **oType** = *bo_Intersect* рассматриваемая функция выполняет пересечение тел **solid** и тела выдавливания.

При **oType** = *bo_Difference* рассматриваемая функция выполняет вычитание из тела **solid** тела выдавливания.

Параметры **operNames** и **contoursNames** обеспечивают именование граней, ребер и вершин построенного тела.

Функция ExtrusionResult() при построении тела путем выдавливания кривых имеет те же возможности, что и функция ExtrusionSolid(): выдавливаемые кривые могут располагаться на плоскости, криволинейной поверхности или в пространстве. Выдавливание может выполняться в прямом, обратном и в обоих направлениях, с уклоном и без уклона граней. Тело может полностью заполнять замкнутые кривые или иметь тонкую стенку.

Остановимся только на некоторых возможностях рассматриваемой функции, связанных с булевыми операциями.

На рис. 1 показаны тело **solid**, образующая кривая, входящая в данные **sweptData**, и направление выдавливания **direction**:

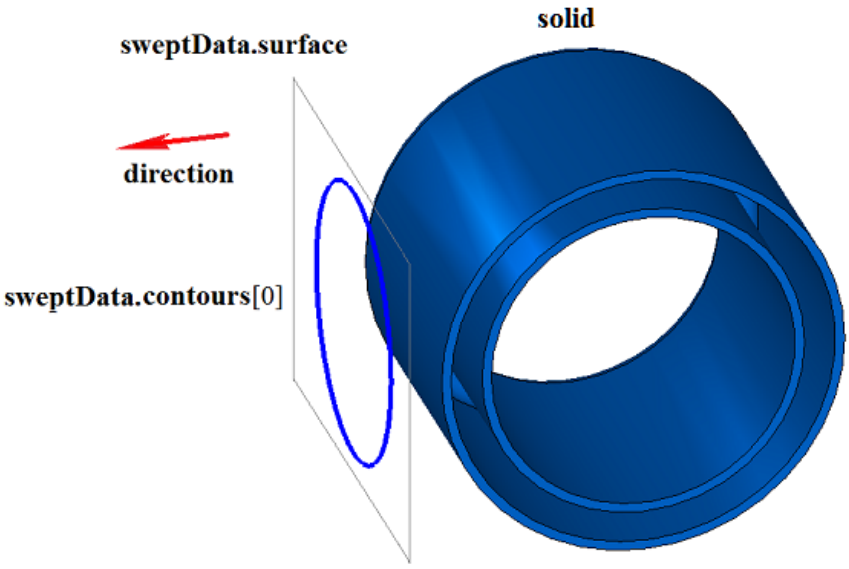


Рис. 1

На рис. 2 приведен результат булевой операции объединения тела **solid** и тонкостенного тела, полученного выдавливанием образующей кривой **sweptData** по направлению **direction**, показанных на рис. 1, на заданное расстояние:

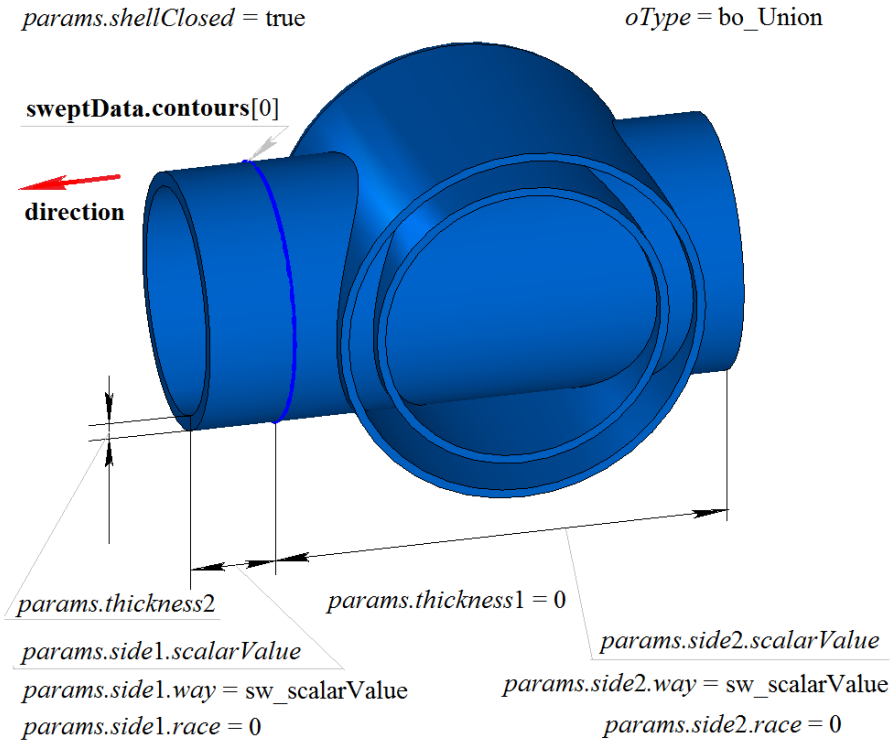


Рис. 2

На рис. 3 приведен результат булевой операции объединения тела **solid** и тела, полученного выдавливанием образующей кривой **sweptData** по направлению **direction**, показанных на рис. 1. Выдавливание образующей кривой выполнено в обратном направлении без уклона с опцией «До ближайших объектов» (*params.side2.way* = sw_shell).

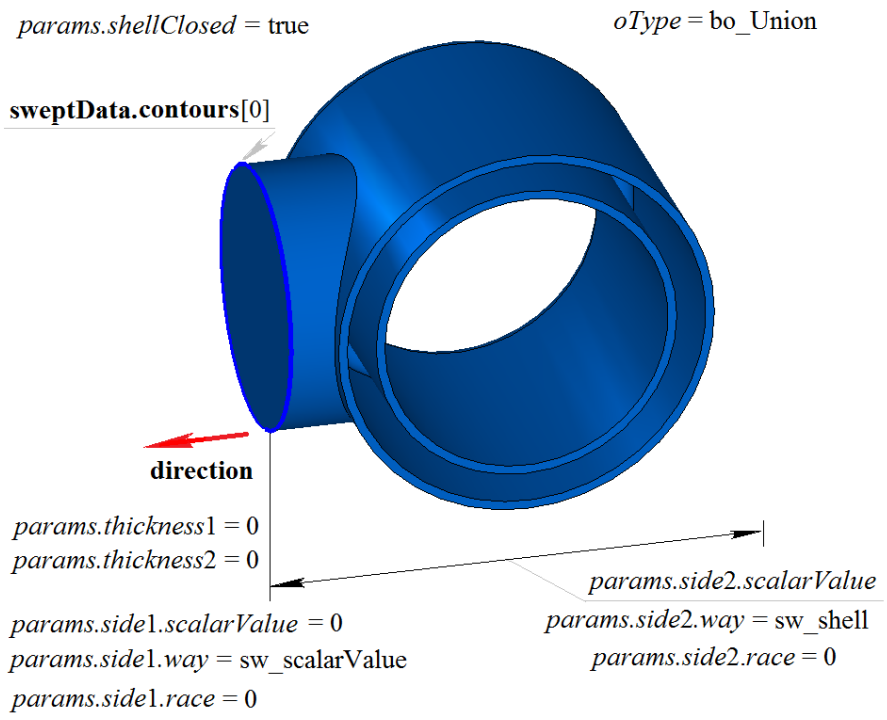


Рис. 3

На рис. 4 приведен результат булевой операции вычитания из тела **solid** тела, полученного выдавливанием образующей кривой **sweptData** по направлению **direction**, показанных на рис. 1:

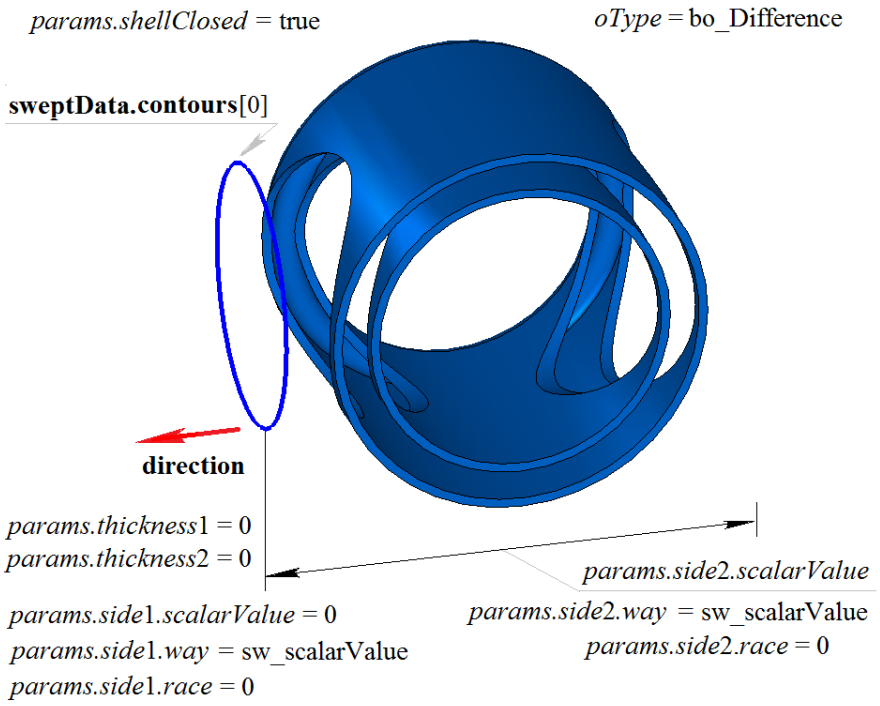


Рис. 4

На рис. 5 приведен результат булевой операции вычитания из тела **solid** тела, полученного выдавливанием образующей кривой **sweptData** по направлению **direction**, показанных на рис. 1. Выдавливание образующей кривой выполнено в обратном направлении без уклона с опцией «До ближайших объектов» (*params.side2.way* = sw_shell):

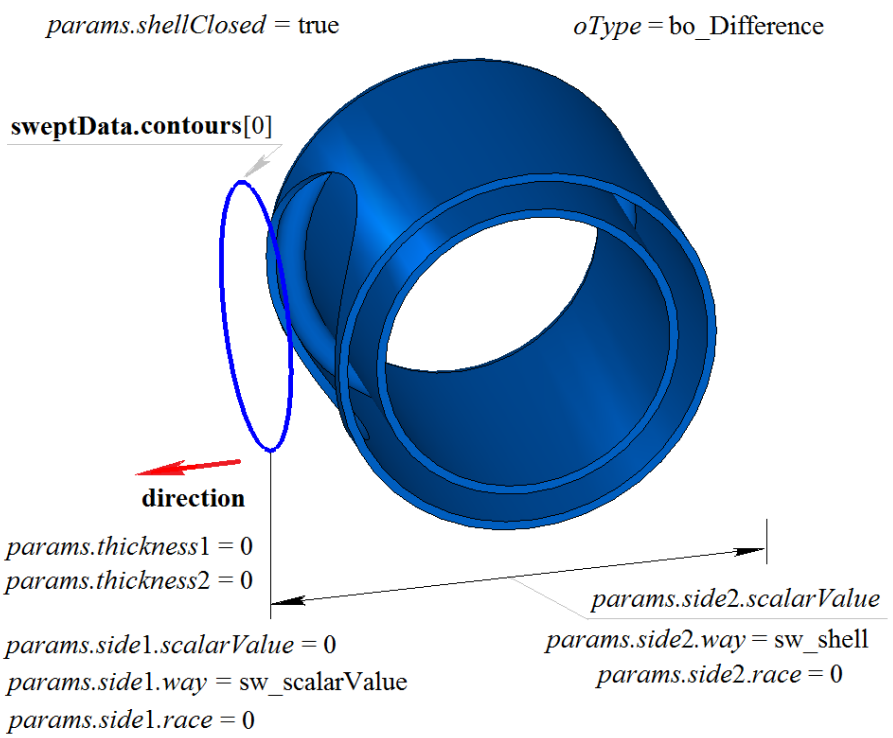


Рис. 5

На рис. 6 приведен результат булевой операции пересечения тела **solid** и тела, полученного выдавливанием образующей кривой **sweptData** по направлению **direction**, показанных на рис. 1:

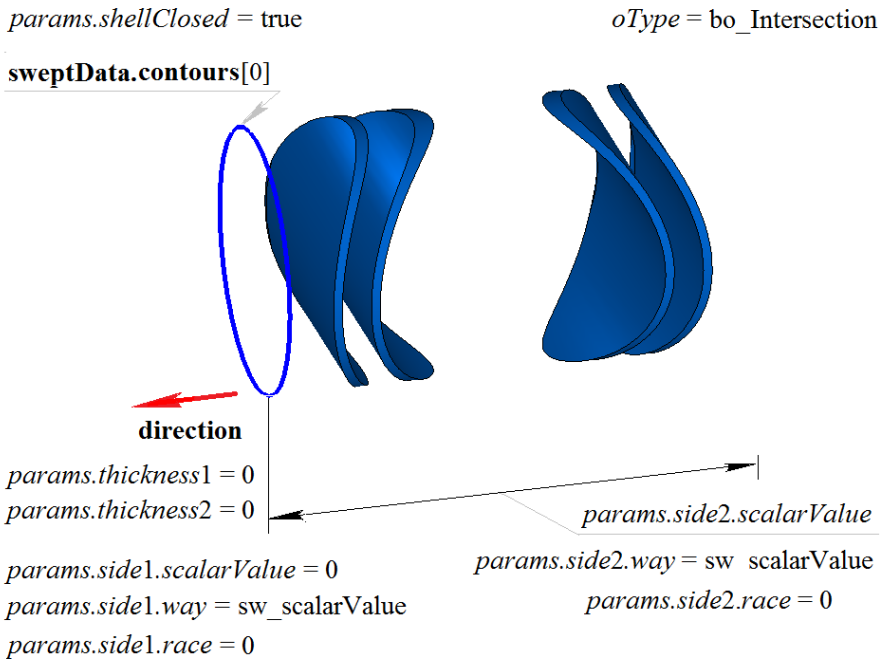


Рис. 6

На рис. 7 приведен результат булевой операции пересечения тела **solid** и тела, полученного выдавливанием образующей кривой **sweptData** по направлению **direction**, показанных на рис. 1. Выдавливание образующей кривой выполнено в обратном направлении без уклона с опцией «До ближайших объектов».

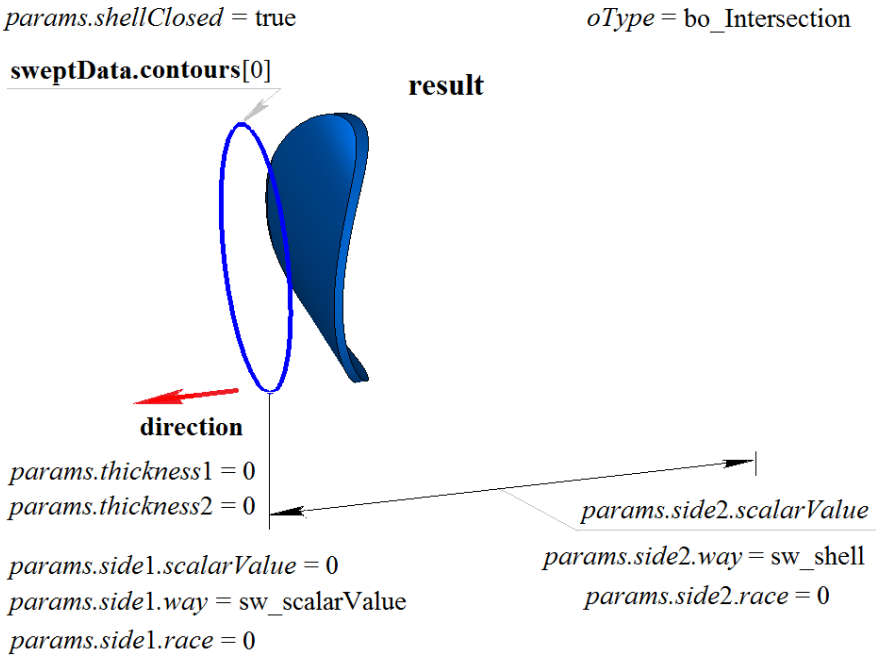


Рис. 7

Функция ExtrusionResult() добавляет в журнал построенного тела строитель MbCurveExtrusionSolid, который содержит все необходимые данные для выполнения операции. Строитель MbCurveExtrusionSolid объявлен в файле *cr_extrusion_solid.h*.

В тестовом приложении *test.exe* булева операция с построенным телом выдавливания выполняется командами меню «Создать→Тело→Приклеиванием к телу→Выдавливания кривой», «Создать→Тело→Вырезанием из тела→Выдавливания кривой» и «Создать→Тело→Пересечением с телом→Выдавливания кривой».

3.2.3 Булева операция с телом вращения

Функция

```
MbResultType RevolutionResult(       MbSolid &                solid,
                                     MbeCopyMode              sameShell,
                               const MbSweptData &            sweptData,
                               const MbAxis3D &               axis,
                               const RevolutionValues &       params,
                                     OperationType            oType,
                               const MbSNameMaker &           operNames,
                               const RPArray<MbSNameMaker> &  contoursNames,
                                     MbSolid *&               result );
```

выполняет построение тела вращения по данным **sweptData** и булеву операцию заданного тела **solid** с построенным телом вращения.

Входными параметрами функции являются:

- **solid** – заданное тело для булевой операции;
- **sameShell** – вариант копирования заданного тела;

- **sweptData** – данные об образующих кривых для построения тела вращения;
- **axis** – ось вращения;
- **params** – параметры построения;
- **oType** – тип булевой операции:

  bo_Union – объединение тел,

  bo_Intersect – пересечение тел,

  bo_Difference – вычитание тел;
- **operNames** – именователь операции;
- **contoursNames** – именователи сегментов образующих кривых (контуров), используемых при построении тела вращения.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Функция представляет собой последовательное объединение двух функций: функции RevolutionSolid(), выполняющей построение тела путем вращения кривых **sweptData** по заданным параметрам **params** вокруг оси **axis**, и функции BooleanResult(), выполняющей булеву операцию **oType** тела **solid** с построенным на предыдущем шаге телом. Функция RevolutionSolid() описана в параграфе *Построение тела вращения*, функция BooleanResult() описана в параграфе *Булева операция над телами*.

Параметр **sameShell** управляет передачей граней, ребер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметры вращения и структуры результата задаются полями **params**: направление и углы вращения – через **params**.*side1* и **params**.*side2*, толщина стенки – через **params**.*thickness1* и **params**.*thickness2*, замкнутость оболочки – через **params**.*shellClosed*, слияние сходных граней и рёбер – через флаги **params**.MergingFlags(), топология (сферическая, тороидальная) – через **params**.*shape*.

Параметр **oType** определяет тип булевой операции и может принимать три значения: *bo_Union*, *bo_Intersect*, *bo_Difference*.

При **oType** = *bo_Union* выполняется объединение тел **solid** и тела вращения.

При **oType** = *bo_Intersect* выполняется пересечение тел **solid** и тела вращения.

При **oType** = *bo_Difference* выполняется вычитание из тела **solid** тела вращения.

Параметры **operNames** и **contoursNames** обеспечивают именование граней, ребер и вершин построенного тела.

Функция RevolutionResult() при построении тела путем вращения кривых имеет те же возможности, что и функция RevolutionSolid(): вращаемые кривые могут располагаться на плоскости, криволинейной поверхности или в пространстве. Вращение может выполняться в прямом, обратном и в обоих направлениях. Тело может полностью заполнять замкнутые кривые или иметь тонкую стенку.

Остановимся на некоторых возможностях функции, связанных с булевыми операциями.

На рис. 1 показаны тело **solid**, образующая кривая, входящая в данные **sweptData**, и ось вращения **axis**:

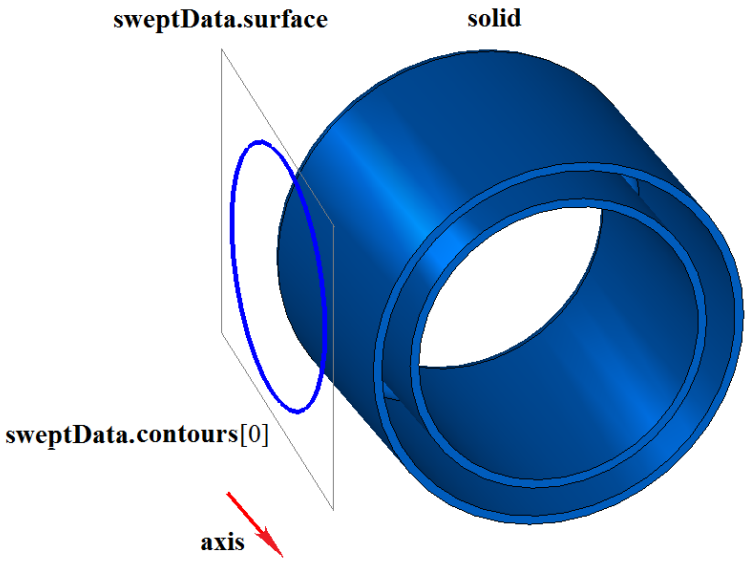


Рис. 1

На рис. 2 приведен результат булевой операции объединения тела **solid** и тонкостенного тела, полученного вращением образующей кривой **sweptData** вокруг оси **axis**, показанных на рис. 1:

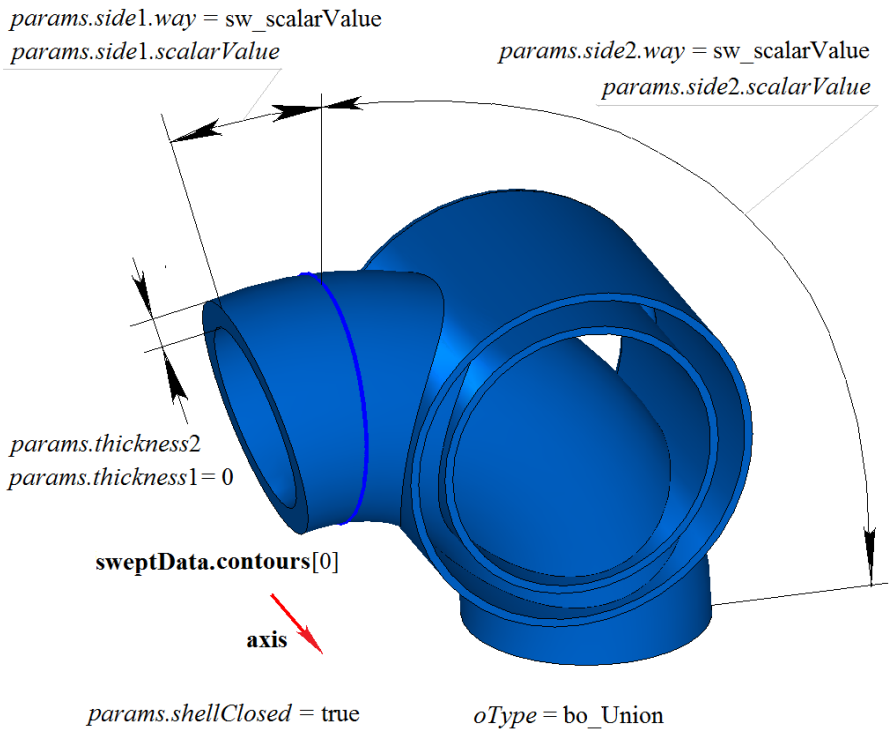


Рис. 2

На рис. 3 приведен результат булевой операции объединения тела **solid** и тонкостенного тела, полученного вращением образующей кривой **sweptData** вокруг оси **axis**, показанных на рис. 1. Вращение образующей кривой выполнено в обратном направлении с опцией «До поверхности» (*params.side2.way* = sw_surface).

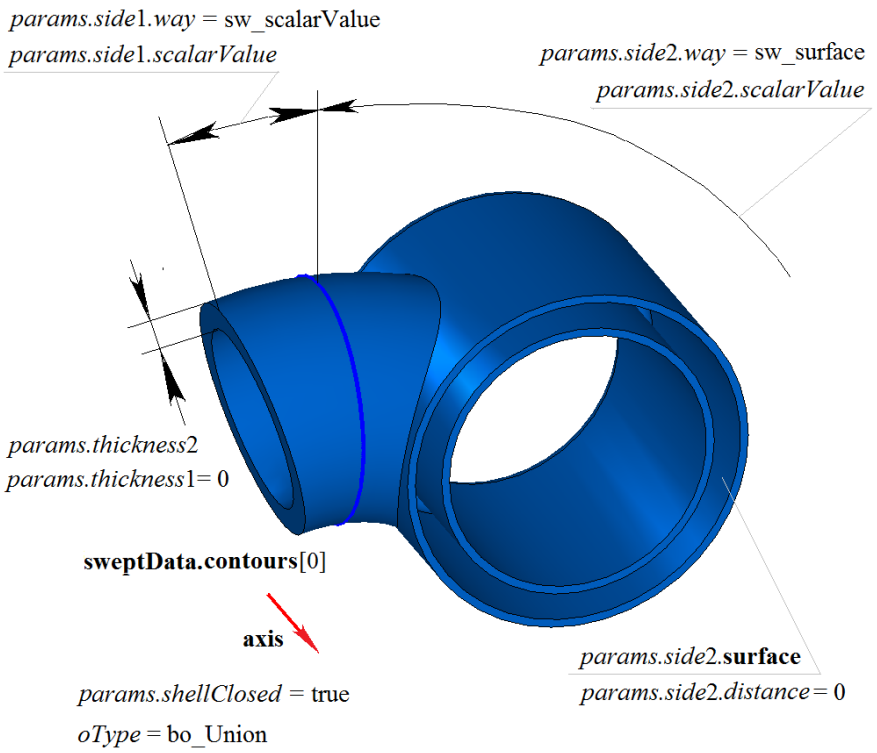


Рис. 3

На рис. 4 приведен результат булевой операции вычитания из тела **solid** тела, полученного вращением образующей кривой **sweptData** вокруг оси **axis**, показанных на рис. 1:

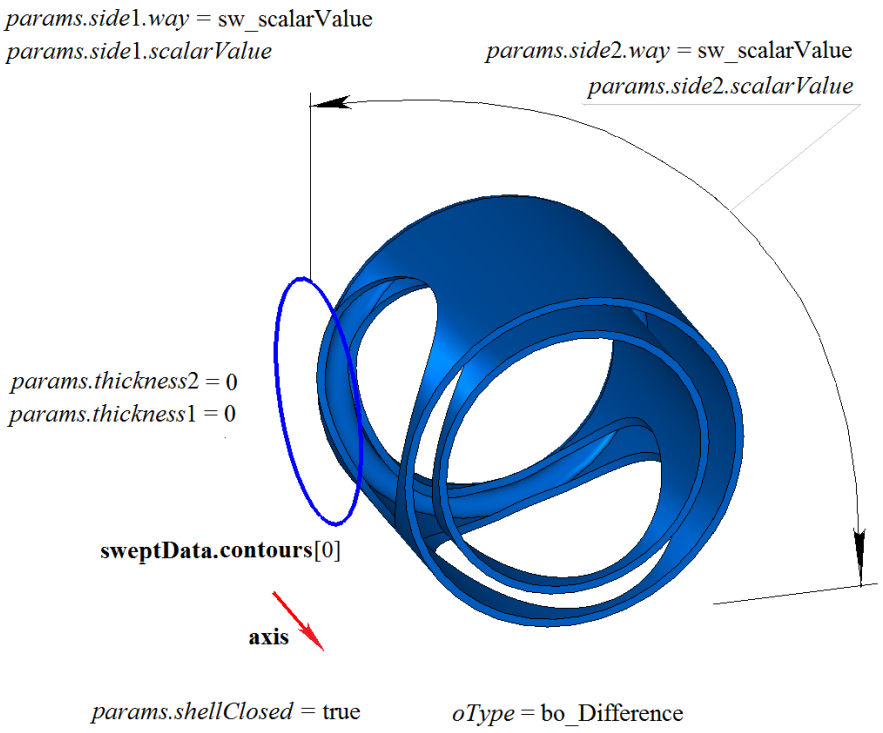


Рис. 4

На рис. 5 приведен результат булевой операции вычитания из тела **solid** тела, полученного вращением образующей кривой **sweptData** вокруг оси **axis**, показанных на рис. 1. Вращение образующей кривой выполнено в обратном направлении с опцией «До поверхности» (*params.side2.way* = sw_surface).

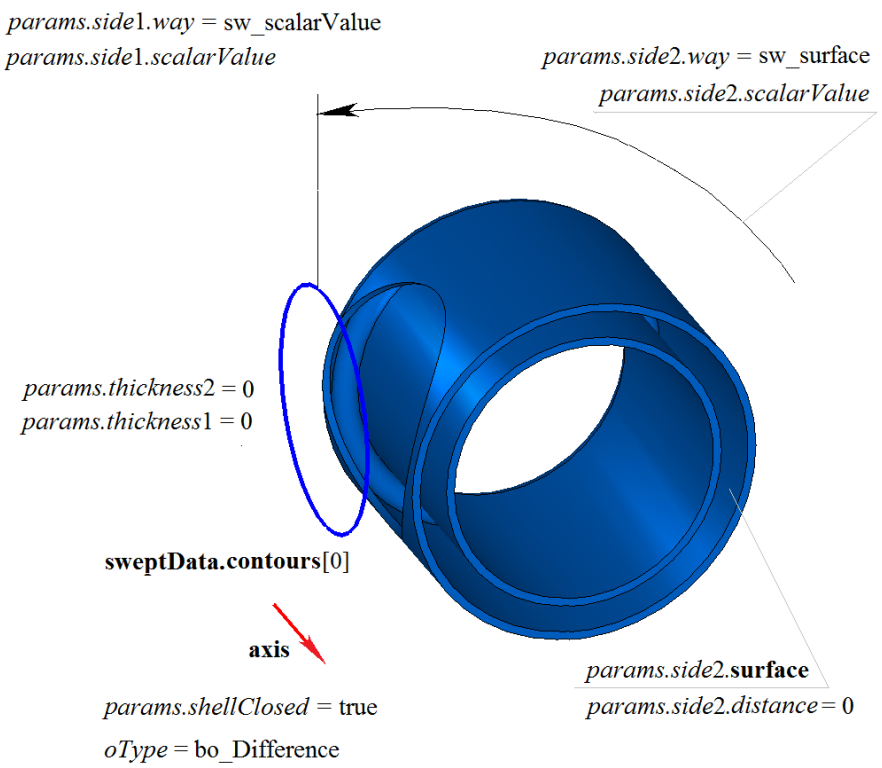


Рис. 5

На рис. 6 приведен результат булевой операции пересечения тела **solid** и тела, полученного вращением образующей кривой **sweptData** вокруг оси **axis**, показанных на рис. 1:

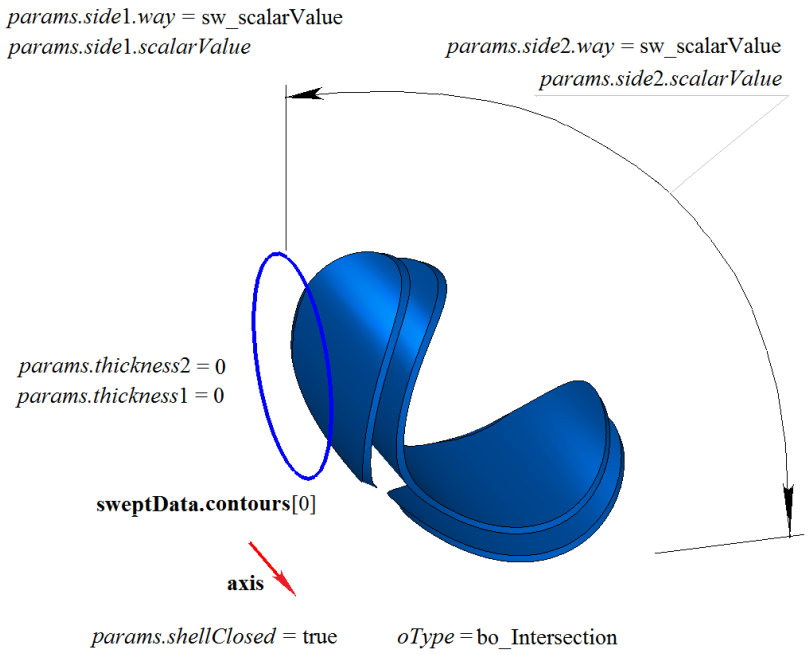


Рис. 6

На рис. 7 приведен результат булевой операции пересечения тела **solid** и тела, полученного вращением образующей кривой **sweptData** вокруг оси **axis**, показанных на рис. 1. Вращение образующей кривой выполнено в обратном направлении с опцией «До поверхности» (*params.side2.way* = sw_surface).

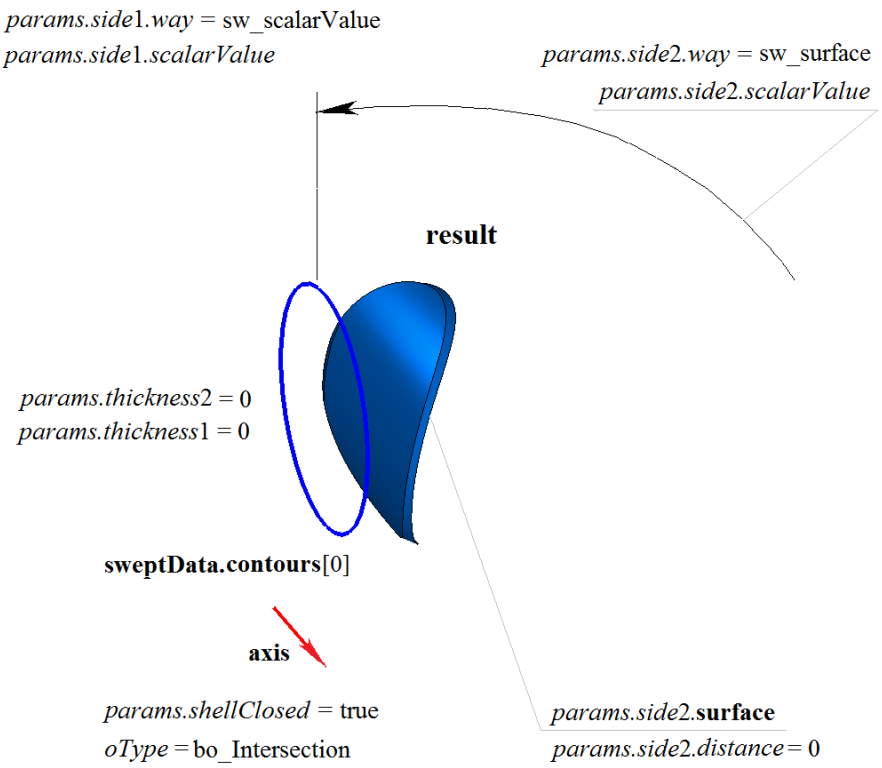


Рис. 7

Функция RevolutionResult() добавляет в журнал построенного тела строитель MbCurveRevolutionSolid, который содержит все необходимые данные для выполнения операции. Строитель MbCurveRevolutionSolid объявлен в файле *cr_revolution_solid.h*.

В тестовом приложении *test.exe* булева операция с построенным телом вращения выполняется командами меню «Создать→Тело→Приклеиванием к телу→Вращения кривой», «Создать→Тело→Вырезанием из тела→Вращения кривой» и «Создать→Тело→Пересечением с телом→Вращения кривой».

3.2.4 Булева операция с кинематическим телом

Функция

```
MbResultType EvolutionResult( const c3d::SolidSPtr &          srcSolid,
                              const MbeCopyMode               copyMode,
                                    OperationType             oType,
                              const MbEvolutionShellParams &  evShellParams,
                                    c3d::SolidSPtr &          result );
```

выполняет построение кинематического тела и булеву операцию заданного тела с построенным телом.

Входными параметрами функции являются:

- **srcSolid** – заданное тело для булевой операции;
- **copyMode** – режим копирования заданного тела;

- **oType** – тип булевой операции:

  bo_Union – объединение тел,

  bo_Intersect – пересечение тел,

  bo_Difference – вычитание тел;

- **evShellParams** – исходные данные для создания кинематического тела.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

При создании объекта **evShellParams** с исходными данными для построения используется один из следующих конструкторов (см. *op_shell_parameter.h*):

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbCurve3D &              spineCurve,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                              IProgressIndicator *     prog = nullptr );
```

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbWireFrame &            spineFrame,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                              IProgressIndicator *     prog = nullptr );
```

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbSpine &                spine,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                              IProgressIndicator *     prog = nullptr );
```

```
MbEvolutionShellParams( const MbSweptData &            sweptData,
                        const MbSpine &                spine,
                        const EvolutionValues &        evolParams,
                        const MbSNameMaker &           operName,
                        const RPArray<MbSNameMaker> &  contoursNames,
                        const MbSNameMaker &           spineName,
                        const VariableSectionValues &  varSectVals,
                              IProgressIndicator *     prog = nullptr );
```

где

- **sweptData** – данные об образующей;
- **spineCurve** – направляющая кривая;

- **spineFrame** – направляющая кривая в виде проволочного каркаса;
- **spine** – направляющая кривая с дополнительной информацией;
- **evolParams** – параметры операции;
- **operName** – именователь операции;
- **contoursNames** – именователь контуров образующей;
- **spineName** – именователь направляющей;
- **varSectVals** – данные, необходимые для построения элемента по траектории с изменяемым сечением;
- **prog** – индикатор прогресса выполнения операции.

Функция представляет собой последовательное объединение двух функций: функции EvolutionShell(), выполняющей построение тела путем движения образующих кривых **sweptData** вдоль направляющей кривой **spine** (**spineCurve**, **spineFrame**), и функции BooleanResult(), выполняющего булеву операцию **oType** тела **srcSolid** с построенным на предыдущем шаге кинематическим телом. Функция EvolutionShell() описана в параграфе Построение кинематического тела, функция BooleanResult() описана в параграфе *Булева операция над телами*.

Параметр **copyMode** управляет передачей граней, ребер и вершин от исходного тела **srcSolid** построенному телу **result** и может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **oType** определяет тип булевой операции и может принимать три значения: *bo_Union*, *bo_Intersect*, *bo_Difference*.

При **oType** = *bo_Union* функция выполняет объединение тела **srcSolid** и кинематического тела.

При **oType** = *bo_Intersect* функция выполняет пересечение тел **srcSolid** и кинематического тела.

При **oType** = *bo_Difference* функция выполняет вычитание из тела **srcSolid** кинематического тела.

Параметры **operName** и **contoursNames** обеспечивают именование граней, ребер и вершин построенного тела.

Функция EvolutionResult() при построении тела путем движения кривых имеет те же возможности, что и функция EvolutionShell(): направляющие кривые могут располагаться на плоскости, криволинейной поверхности или в пространстве, тело может полностью заполнять замкнутые кривые или иметь тонкую стенку. Остановимся на некоторых возможностях рассматриваемой функции, связанных с булевыми операциями.

На рис. 1 показаны тело **srcSolid**, образующая кривая **sweptData** и направляющая кривая **spine**:

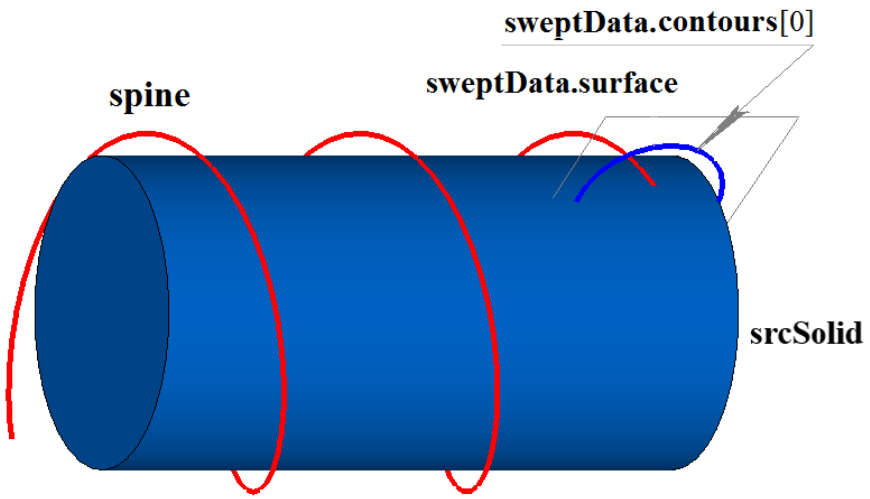


Рис. 1

На рис. 2 приведен результат булевой операции объединения тела **srcSolid** и тела, полученного движением образующей кривой **sweptData** вдоль направляющей кривой **spine**, показанных на рис. 1:

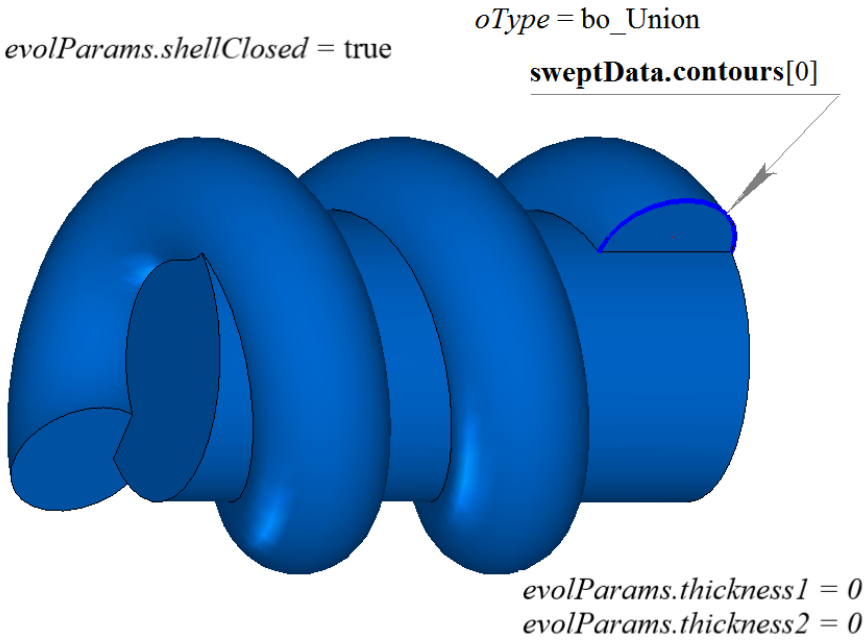


Рис. 2

На рис. 3 приведен результат булевой операции вычитания из тела **srcSolid** тела, полученного движением образующей кривой **sweptData** вдоль направляющей кривой **spine**,

показанных на рис. 1:

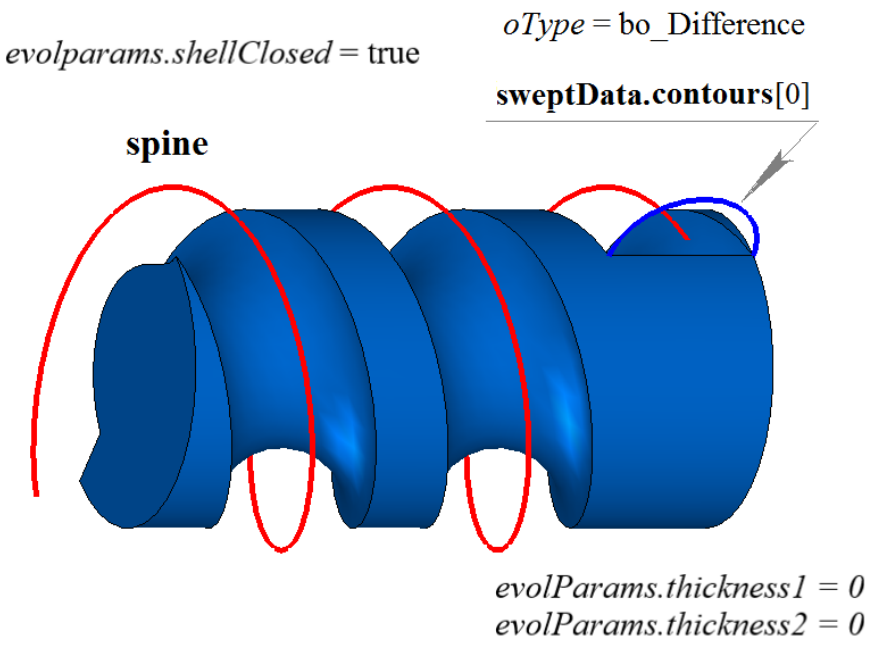


Рис. 3

На рис. 4 приведен результат булевой операции пересечения тела **srcSolid** и тела, полученного движением образующей кривой **sweptData** вдоль направляющей кривой **spine**, показанных на рис. 1:

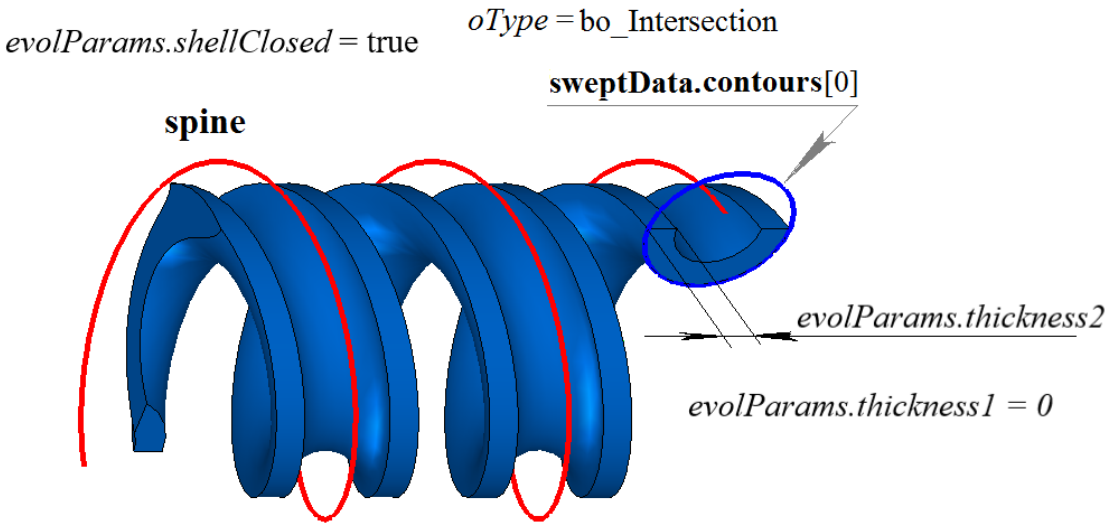


Рис. 4

В тестовом приложении *test.exe* булева операция с построенным кинематическим телом выполняется командами меню «Создать→Тело→Приклеиванием к телу→Движения кривой», «Создать→Тело→Вырезанием из тела→Движения кривой» и «Создать→Тело→Пересечением с телом→Движения кривой».

3.2.5 Булева операция с телом, построенным по плоским сечениям

Функция

```
MbResultType LoftedResult( const c3d::SolidSPtr &         srcSolid,
                           const MbeCopyMode              copyMode,
                           const MbLoftedSolidParams &    loftSolParams,
                                 c3d::SolidSPtr &         result );
```

выполняет построение тела по плоским сечениям и булеву операцию заданного тела с построенным телом.

Входными параметрами функции являются:

- **srcSolid** – заданное тело для булевой операции;
- **copyMode** – режим копирования заданного тела;
- **loftSolParams** – исходные данные для выполнения операции.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

При создании объекта **loftSolParams** с исходными данными для построения используется один из следующих конструкторов (см. *op_shell_parameter.h*):

```
MbLoftedSolidParams( const std::vector<MbPlacement3D> &    places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );
```

```
MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );
```

```
MbLoftedSolidParams( const std::vector<MbPlacement3D> &   places,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                      const c3d::ConstSpaceCurveSPtr &      spine,
                      const LoftedValues &                  params,
                      const OperationType                   oType,
                      const c3d::SpaceCurvesSPtrVector *    guideCurves,
                      const c3d::SpacePointsVector *        guidePoints,
                      const MbSNameMaker &                  operName,
                      const c3d::SNamesMakerSPtrVector &    contourNames,
                            IProgressIndicator *            prog = nullptr );
```

```
MbLoftedSolidParams( const c3d::SurfacesSPtrVector &       surfaces,
                     const c3d::PlaneContoursSPtrVector &  planeConours,
                     const c3d::ConstSpaceCurveSPtr &      spine,
                     const LoftedValues &                  params,
                     const OperationType                   oType,
                     const c3d::SpaceCurvesSPtrVector *    guideCurves,
                     const c3d::SpacePointsVector *        guidePoints,
                     const MbSNameMaker &                  operName,
                     const c3d::SNamesMakerSPtrVector &    contourNames,
                           IProgressIndicator *            prog = nullptr );
```

где

- **places** – множество систем координат образующих контуров;
- **surfaces** – множество поверхностей образующих контуров;
- **planeConours** – множество образующих контуров;
- **spine** – осевая кривая (может быть nullptr);
- **params** – параметры операции;
- **oType** – тип булевой операции;
- **guideCurves** – множество направляющих кривых, задающих траектории соответствующих точек контуров (опционально, может быть nullptr);
- **guidePoints** – множество точек на образующих контурах, задающее соответствующие точки (цепочки точек) (опционально, может быть nullptr);
- **operName** – именователь операции;
- **contourNames** – именователи образующих контуров.

Функция представляет собой последовательное объединение двух функций: функции LoftedSolid(), выполняющей построение тела по плоским сечениям **planeConours** на плоскостях **places** по заданным параметрам **params**, и функции BooleanResult(), выполняющей булеву операцию **oType** тела **srcSolid** с построенным на предыдущем шаге телом. Функция LoftedSolid() описана в параграфе *Построение тела по сечениям*, функция BooleanResult() описана в параграфе *Булева операция над телами*.

Параметр **copyMode** управляет передачей граней, ребер и вершин от исходного тела **srcSolid** построенному телу **result** и может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);

- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **oType** определяет тип булевой операции и может принимать три значения: *bo_Union*, *bo_Intersect*, *bo_Difference*.

При **oType** = *bo_Union* функция выполняет объединение тела **srcSolid** и построенного по сечениям тела.

При **oType** = *bo_Intersect* функция выполняет пересечение тел **srcSolid** и построенного по сечениям тела.

При **oType** = *bo_Difference* функция выполняет вычитание из тела **srcSolid** построенного по сечениям тела.

Параметры **operName** и **contoursNames** обеспечивают именование граней, ребер и вершин построенного тела.

Функция LoftedResult() при построении тела по плоским сечениям имеет те же возможности, что и функция LoftedSolid() – тело может строиться разомкнутым и замкнутым, тело в районе торцов может иметь разную форму, тело может полностью заполнять замкнутые кривые или иметь тонкую стенку, форма тела по сечениям может управляться направляющей. Остановимся на некоторых возможностях рассматриваемой функции, связанных с булевыми операциями.

На рис. 1 показаны тело **srcSolid** и плоские замкнутые кривые:

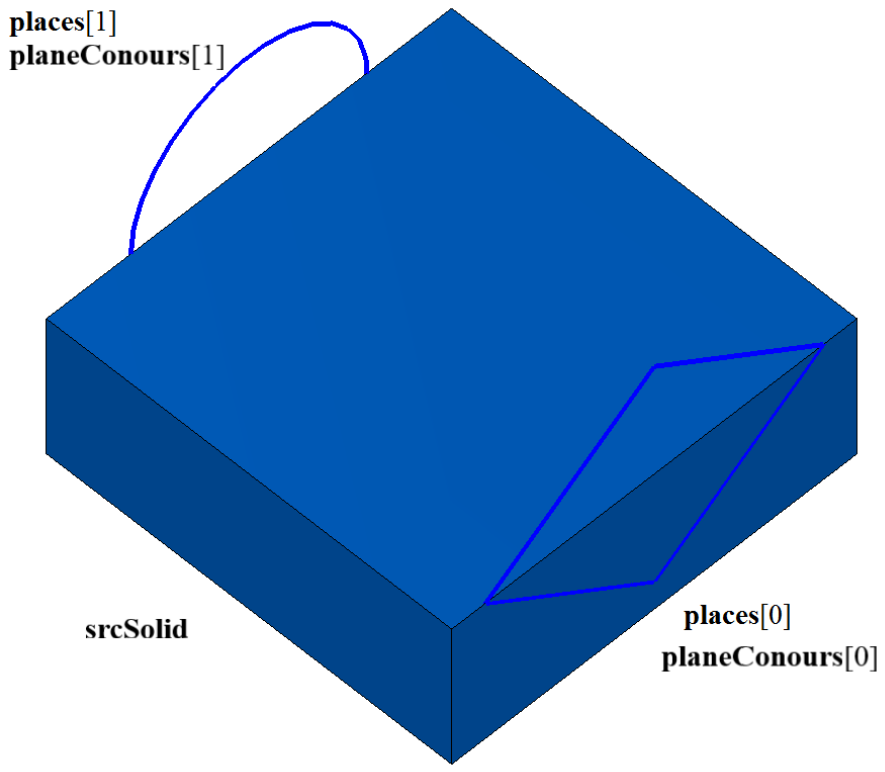


Рис. 1

На рис. 2 приведен результат булевой операции объединения тела **srcSolid** и тела, построенного по плоским сечениям **planeConours**, показанных на рис. 1:

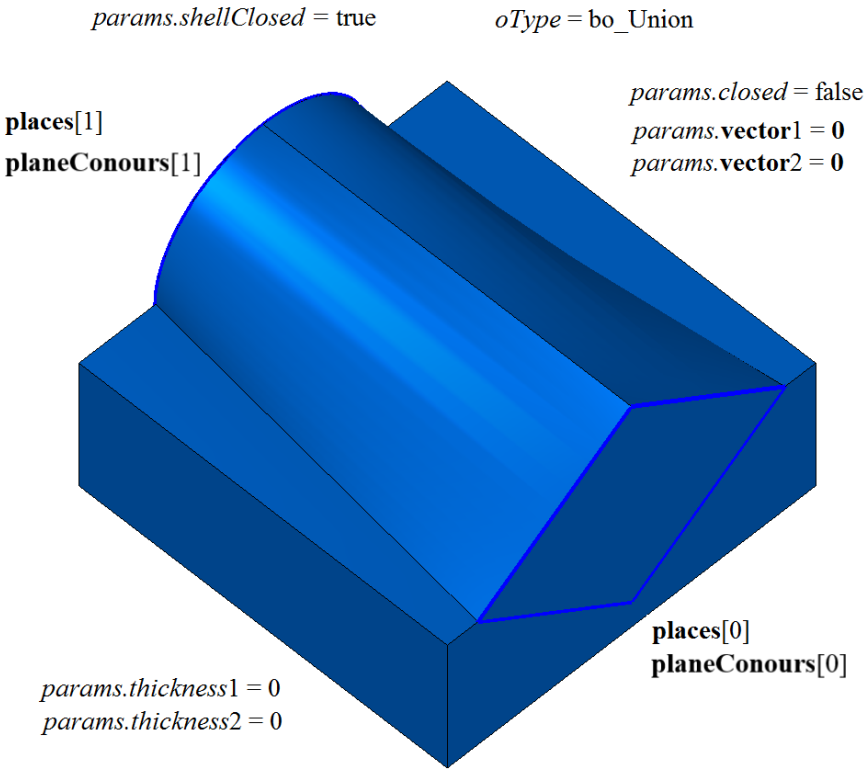


Рис. 2

На рис. 3 приведен результат булевой операции вычитания из тела **srcSolid** тела, построенного по плоским сечениям **planeConours**, показанных на рис. 1:

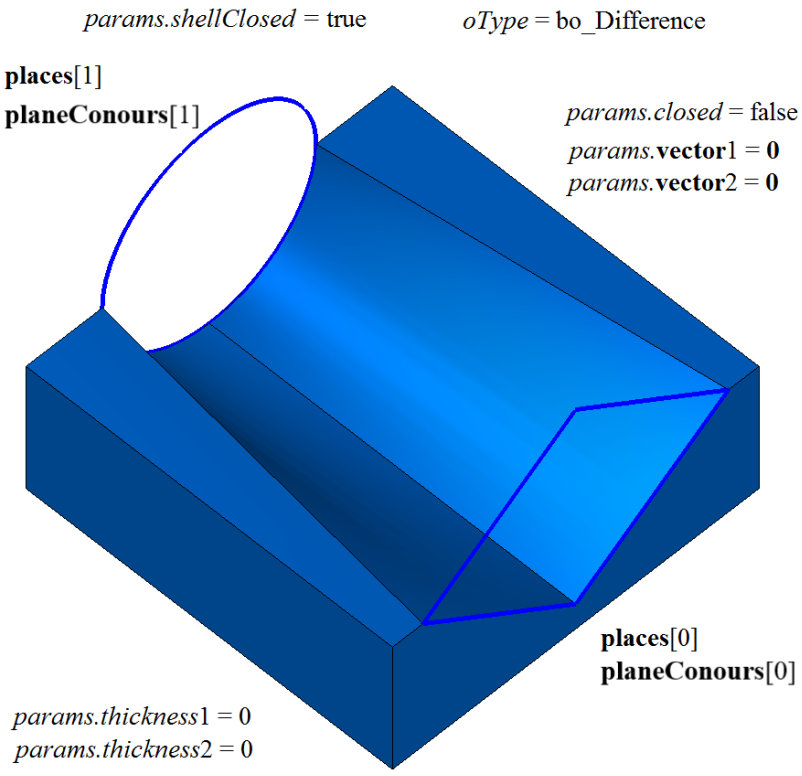


Рис. 3

На рис. 4 приведен результат булевой операции пересечения тела **srcSolid** и тела, построенного по плоским сечениям **planeConours**, показанных на рис. 1:

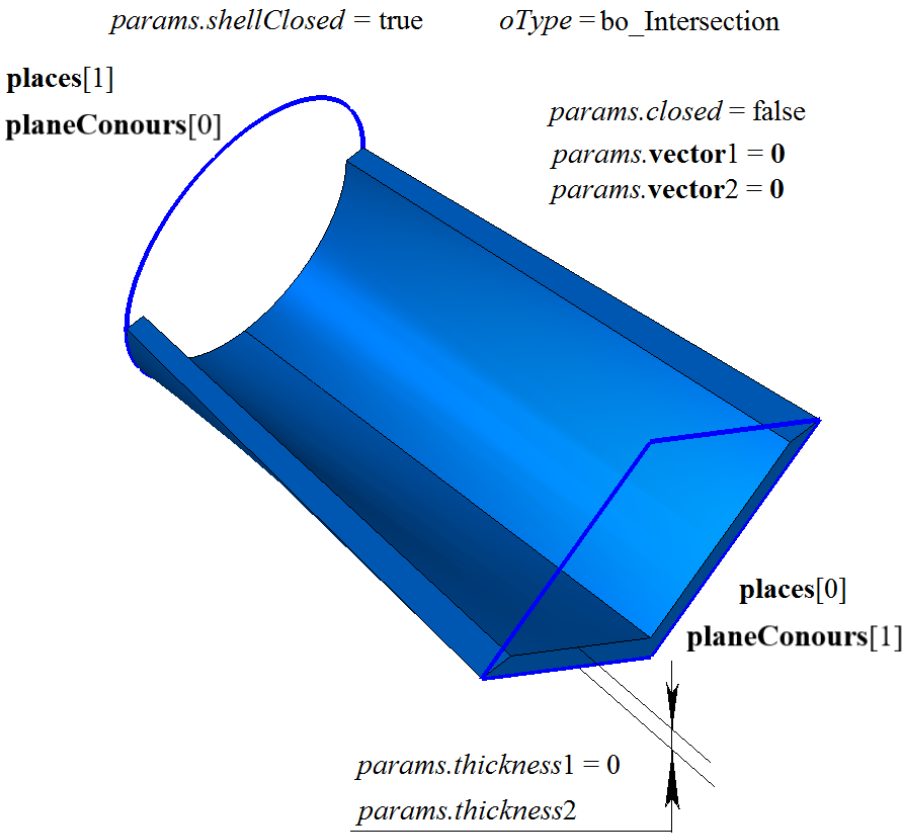


Рис. 4

Функция LoftedResult() добавляет в журнал построенного тела строитель MbCurveLoftedSolid, который содержит все необходимые данные для выполнения операции. Строитель MbCurveLoftedSolid объявлен в файле *cr_lofted_solid.h*.

В тестовом приложении *test.exe* булева операция с построенным телом, построенным по плоским сечениям, выполняется командами меню:

«Создать→Тело→Приклеиванием к телу→По сечениям»,

«Создать→Тело→Приклеиванием к телу→По сечениям и направляющей»,

«Создать→Тело→Вырезанием из тела→По сечениям»,

«Создать→Тело→Вырезанием из тела→По сечениям и направляющей»,

«Создать→Тело→Пересечением с телом→По сечениям»,

«Создать→Тело→Пересечением с телом→По сечениям и направляющей».

3.2.6 Резка тела на части

Функция

```
MbResultType SolidCutting( const c3d::SolidSPtr &        solid,
                                 MbeCopyMode             sameShell,
                           const MbShellCuttingParams &  cuttingParams,
                                 c3d::SolidsSPtrVector & results );
```

отрезает часть тела плоским контуром, поверхностью или оболочкой.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – режим копирования исходного тела;
- **cuttingParams** – параметры операции.

Выходным параметром функции являются построенные тела **results**.

Функция возвращает код результата операции из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_solid.h*.

Для инициализации объектов **cuttingParams** с параметрами операции в классе MbShellCuttingParams определен следующий конструктор по умолчанию:

```
MbShellCuttingParams(       int                  part,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr )

: cutterData  (                      )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

- **part** – сохраняемая часть исходной оболочки (+1, 0, -1), 0 – оставить обе части;
- **mergingFlags** – флаги слияния элементов оболочки;
- **cutAsClosed** – флаг построения замкнутой оболочки;
- **snMaker** – именователь операции.

Резка тела плоским контуром

Функция строит незамкнутую оболочку путем выдавливания в направлении **dir** двумерного контура **contour**, расположенного в плоскости XY локальной системы координат **place**, и выполняет булеву операцию пересечения исходного тела **solid** с незамкнутой оболочкой. При равенстве нулю вектора **dir** выдавливание контура выполняется вдоль вектора **place.axisZ**.

Для выполнения операции секущий контур должен полностью пересекать проекцию исходного тела на плоскость XY локальной системы координат **place** в направлении вектора выдавливания. Длина выдавливания контура рассчитывается так, чтобы незамкнутая оболочка полностью пересекала исходное тело.

Для инициализации объектов **cuttingParams** функции с параметрами операции в классе MbShellCuttingParams определены следующие конструкторы по контуру:

Конструктор с выбором сохраняемой части:

```
MbShellCuttingParams( const MbPlacement3D &      place,
                      const MbContour &          contour,
                            bool                 sameContour,
                      const MbVector3D &         dir,
                            int                  part,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr )

: cutterData  ( place, dir, contour, sameContour )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
  booleanFlags.InitCutting( cutAsClosed );
  booleanFlags.SetMerging( mergingFlags );
  SetRetainedPart( part );
}
```

Конструктор без выбора сохраняемой части (по умолчанию сохраняются все части):

```
MbShellCuttingParams( const MbPlacement3D &      place,
                      const MbContour &          contour,
                            bool                 sameContour,
                      const MbVector3D &         dir,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr )

: cutterData  ( place, dir, contour, sameContour )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
}
```

где

- **place** – локальная система координат, в плоскости XY которой расположен двумерный контур;
- **contour** – двумерный контур выдавливания расположен в плоскости XY локальной системы координат **place**;
- **sameContour** – флаг использования исходного контура (true) или его копии (false);
- **dir** – направление выдавливания контура;
- **part** – сохраняемая часть исходного тела (+1, 0, -1), 0 – оставить обе части;
- **mergingFlags** – флаги слияния элементов оболочки;
- **cutAsClosed** – флаг построения замкнутой оболочки;
- **snMaker** – именователь операции.

Параметр **part** определяет оставляемую часть исходного тела **solid**:

- при *part* = +1 сохраняется часть тела, расположенная справа от контура, если смотреть вдоль контура навстречу оси **place.axisZ**;
- при *part* = -1 сохраняется часть тела, расположенная слева от контура, если смотреть вдоль контура навстречу оси **place.axisZ**;
- при *part* = 0 сохраняются обе части тела.

Параметр **mergingFlags** управляет слиянием подобных граней и ребер результирующей оболочки.

Параметр **cutAsClosed** говорит о том, каким считать исходное тело **solid**: замкнутым или незамкнутым. При **cutAsClosed** = *true* операция выполняется над множеством точек, расположенным внутри и на поверхности тела. При **cutAsClosed** = *false* операция выполняется над множеством точек, расположенным на поверхности тела.

Параметр **snMaker** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 1 показаны исходное тело **solid**, режущий контур **contour** и плоскость XY локальной системы координат **place**.

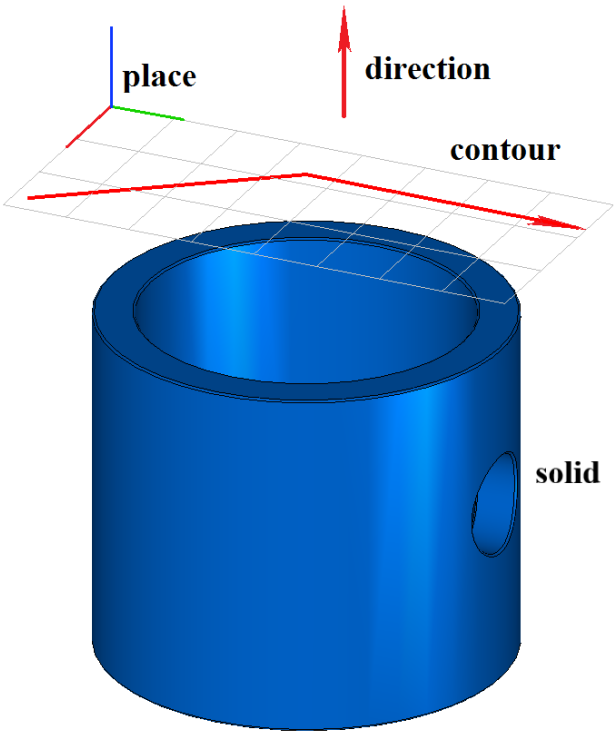
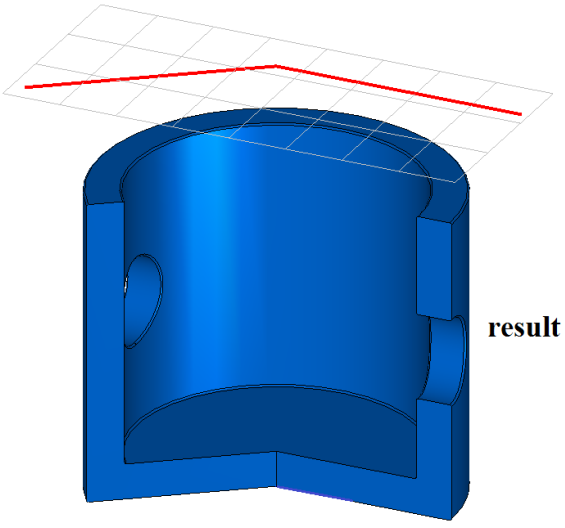


Рис. 1

На рис. 2 приведено построенное тело **result** при *part* = +1 и *cutAsClosed* = *true*. На рис. 3 приведено построенное тело **result** при *part* = -1 и *cutAsClosed* = *true*:



part = +1 *closed* = *true*

Рис. 2

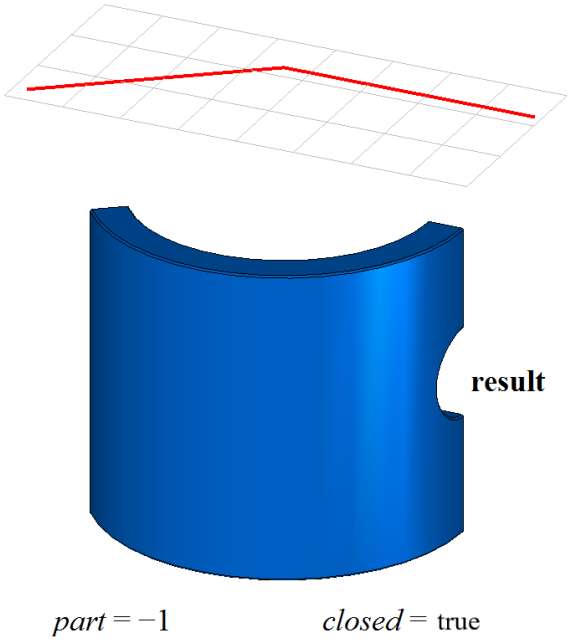


Рис. 3

На рис. 4 приведено построенное тело **result** при *part* = +1 и *cutAsClosed* = *false*:

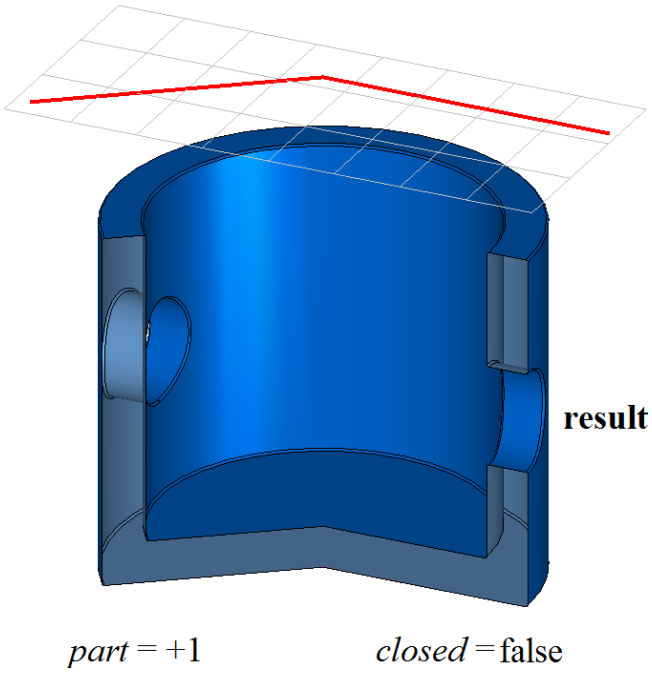


Рис. 4

При *part* = 0 рассматриваемая функция построит тело **result**, в которое войдут все части

исходного тела **solid**, полученные после резки.

Для разделения тела **result** на части следует воспользоваться методами DetachParts() или CreateParts(), описанным в параграфе Разделить тело на несвязанные части.

При использовании конструктора по контуру:

```
MbShellCuttingParams( const MbPlacement3D &      place,
                      const MbContour &          contour,
                            bool                 sameContour,
                      const MbVector3D &         dir,
                      const MbMergingFlags &     mergingFlags,
                            bool                 cutAsClosed,
                      const MbSNameMaker &       snMaker,
                            IProgressIndicator * prog = nullptr );
```

функция SolidCutting() режет тело поверхностью, полученной выдавливанием плоского контура, и из всех частей строит отдельные тела. На рис. 5 приведено построенное данным способом тело **result** при *cutAsClosed* = *true*:

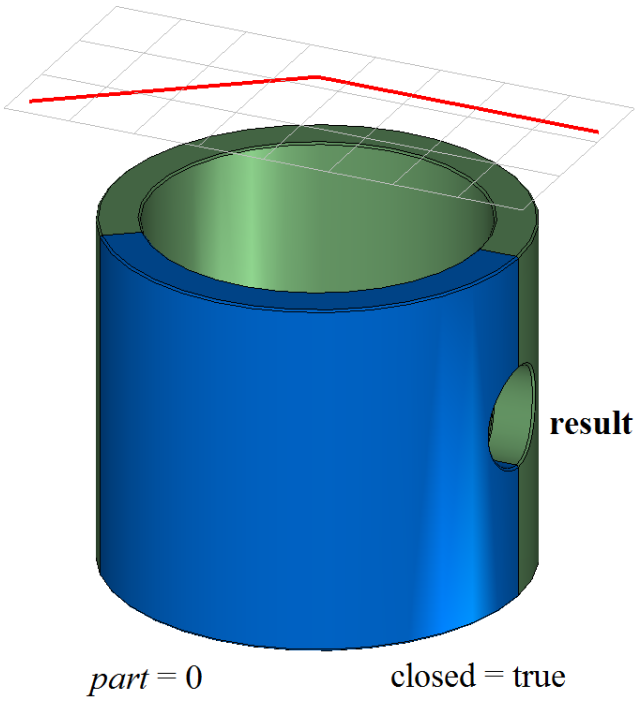


Рис. 5

Функция SolidCutting() добавляет в журнал построенного тела строитель MbCuttingSolid, который содержит все необходимые данные для выполнения операции. Строитель MbCuttingSolid объявлен в файле *cr_cutting_solid.h*.

В тестовом приложении *test.exe* резка тела плоским контуром выполняется командами меню «Создать→Тело→На базе тела→Разрезанное двумерной кривой» и «Создать→Оболочку→На основе оболочки→Разрезанную кривой».

Резка тела поверхностью

Функция строит незамкнутую оболочку с одной гранью на базе режущей поверхности **surface** и выполняет булеву операцию пересечения исходного тела **solid** с незамкнутой оболочкой. Для выполнения операции режущая поверхность должна полностью пересекать исходное тело.

Для инициализации объектов **cuttingParams** функции с параметрами операции в классе MbShellCuttingParams определены следующие конструкторы по поверхности:

```
MbShellCuttingParams( const MbSurface &          surface,
                      bool                       sameSurface,
                      int                        part,
                const MbMergingFlags &           mergingFlags,
                      bool                       cutAsClosed,
                const MbSNameMaker &             snMaker,
                      IProgressIndicator * prog = nullptr )

: cutterData  ( surface, sameSurface )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

```
MbShellCuttingParams( const MbSurface &          surface,
                      bool                       sameSurface,
                const MbMergingFlags &           mergingFlags,
                      bool                       cutAsClosed,
                const MbSNameMaker &             snMaker,
                      IProgressIndicator * prog = nullptr )

: cutterData  ( surface, sameSurface )
, booleanFlags(                      )
, nameMaker   ( &snMaker.Duplicate() )
, retainedPart( ts_neutral           )
, prolongState(                      )
, progress    ( prog                 )
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
}
```

- **surface** – режущая поверхность;
- **sameSurface** – флаг использования исходной поверхности (true) или её копии (false);
- **part** – сохраняемая часть исходного тела (+1, 0, -1), 0 – оставить обе части;
- **mergingFlags** – флаги слияния элементов оболочки;
- **cutAsClosed** – флаг построения замкнутой оболочки;

- **snMaker** – именователь операции.

Параметр **part** определяет сохраняемую часть исходного тела **solid**:

- при **part** = +1 сохраняется часть тела, расположенная над поверхностью (с той стороны, в которую направлена нормаль поверхности);
- при **part** = –1 сохраняется часть тела, расположенная под поверхностью;
- при **part** = 0 сохраняются обе части тела.

Параметр **cutAsClosed** говорит о том, каким считать исходное тело **solid**: замкнутым или незамкнутым:

- при **cutAsClosed** = *true* операция выполняется над множеством точек, расположенным внутри и на поверхности тела;
- при **cutAsClosed** = *false* операция выполняется над множеством точек, расположенным на поверхности тела.

Параметр **snMaker** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 6 показаны исходное тело **solid** и режущая поверхность **surface**.

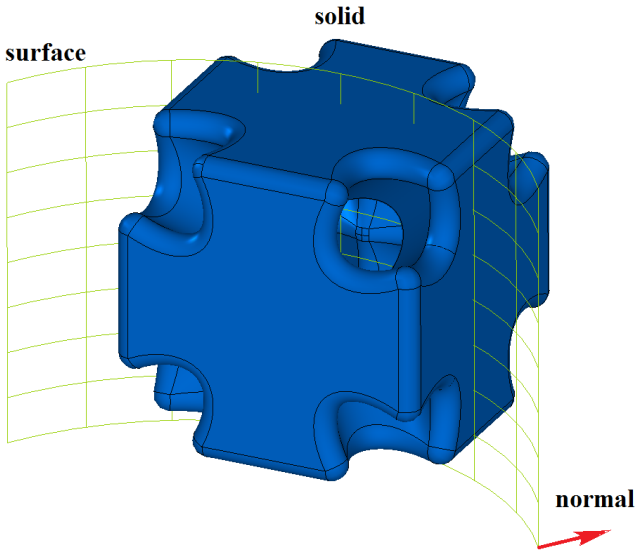


Рис. 6

На рис. 7 приведено построенное тело **result** при *part* = +1 и *cutAsClosed* = true. На рис. 8 приведено построенное тело **result** при *part* = -1 и *cutAsClosed* = true.

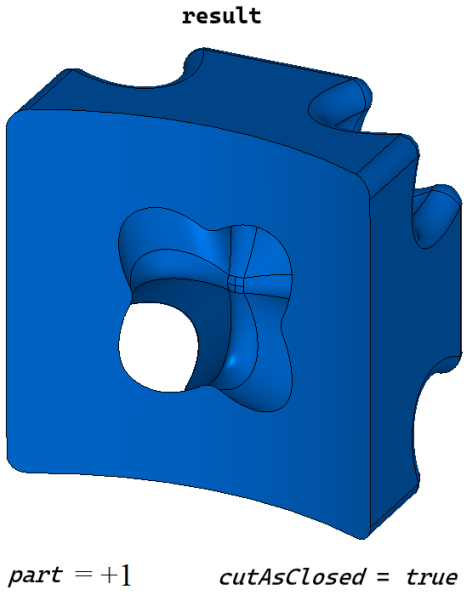


Рис. 7

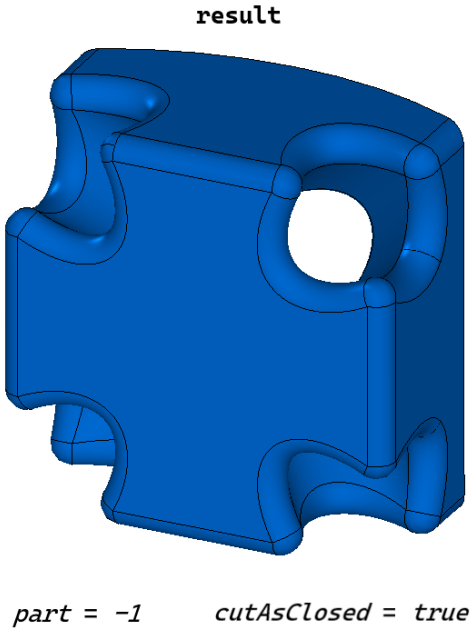


Рис. 8

На рис. 9 приведено построенное тело **result** при *part* = +1 и *cutAsClosed* = false:

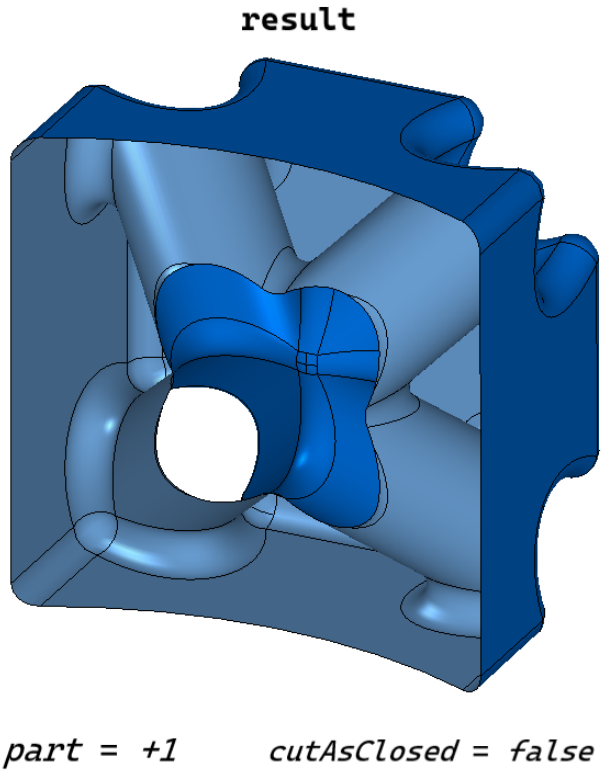


Рис. 9

При **part** = 0 функция построит тело **result**, в которое войдут все части исходного тела **solid**, полученные после резки.

Для разделения тела **result** на части следует воспользоваться функциями DetachParts() или CreateParts(), описанными в параграфе Разделить тело на несвязанные части.

При использовании конструктора по поверхности

```
MbShellCuttingParams( const MbSurface &          surface,
                      bool                       sameSurface,
                      const MbMergingFlags &     mergingFlags,
                      bool                       cutAsClosed,
                      const MbSNameMaker &       snMaker,
                      IProgressIndicator * prog = nullptr )
```

функция режет тело пересекающей его поверхностью и из всех частей строит отдельные тела. На рис. 10 приведено построенное данным методом тело **result** при *cutAsClosed* = *true*:

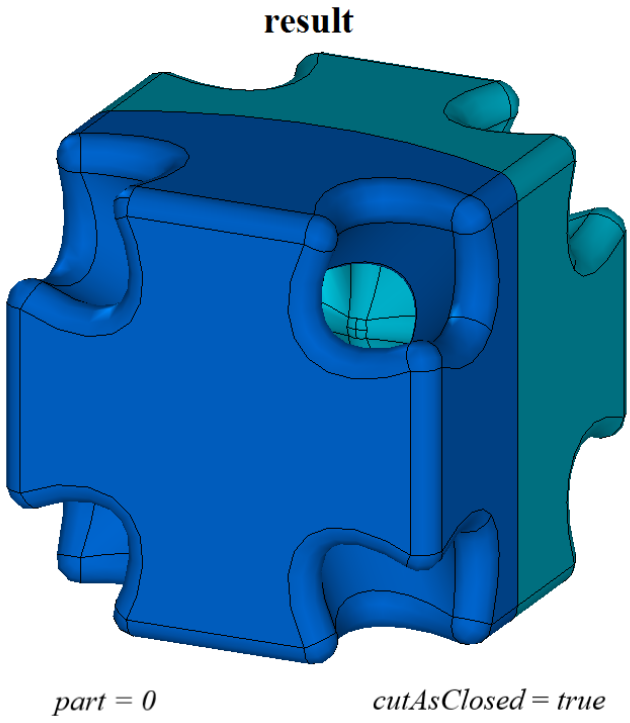


Рис. 10

В тестовом приложении *test.exe* резка тела поверхностью выполняется командами меню «Создать→Тело→На базе тела→Разрезанное поверхностью» и «Создать→Оболочку→На основе оболочки→Разрезанную поверхностью».

Резка тела оболочкой

Функция выполняет резку исходного тела **solid** пересекающей его оболочкой **cutSolid**. Для выполнения операции режущая оболочка **cutSolid** должна полностью пересекать исходное тело **solid**.

Для инициализации объектов **cuttingParams** функции с параметрами операции в классе MbShellCuttingParams определен следующий конструктор по оболочке:

```
MbShellCuttingParams( const MbSolid &            solid,
                      bool                       sameSolid,
                      int                        part,
                    const MbMergingFlags &       mergingFlags,
                      bool                       cutAsClosed,
                    const MbSNameMaker &         snMaker,
                      IProgressIndicator * prog = nullptr )

: cutterData  ( solid, sameSolid, true )
, booleanFlags(                         )
, nameMaker   ( &snMaker.Duplicate()    )
, retainedPart( ts_neutral              )
, prolongState(                         )
, progress    ( prog                    )
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
{
    booleanFlags.InitCutting( cutAsClosed );
    booleanFlags.SetMerging( mergingFlags );
    SetRetainedPart( part );
}
```

- **solid** – режущая оболочка;
- **sameSolid** – флаг использования исходного тела (*true*) или его копии (*false*);
- **part** – сохраняемая часть исходного тела (+1, 0, -1), 0 – оставить обе части;
- **mergingFlags** – флаги слияния элементов оболочки;
- **cutAsClosed** – флаг построения замкнутой оболочки;
- **snMaker** – именователь операции.

Параметр **part** определяет сохраняемую часть исходного тела **solid**:

- при **part** = +1 сохраняется часть тела, расположенная снаружи оболочки;
- при **part** = –1 сохраняется часть тела, расположенная внутри оболочки;
- при **part** = 0 сохраняются обе части тела.

Параметр **cutAsClosed** говорит о том, каким считать исходное тело **solid**: замкнутым или незамкнутым:

- при **cutAsClosed** = *true* операция выполняется над множеством точек, расположенным внутри и на поверхности тела;
- при **cutAsClosed** = *false* операция выполняется над множеством точек, расположенным на поверхности тела.

Параметр **snMaker** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 11 показаны исходное тело **solid** и режущая оболочка **cutSolid**.

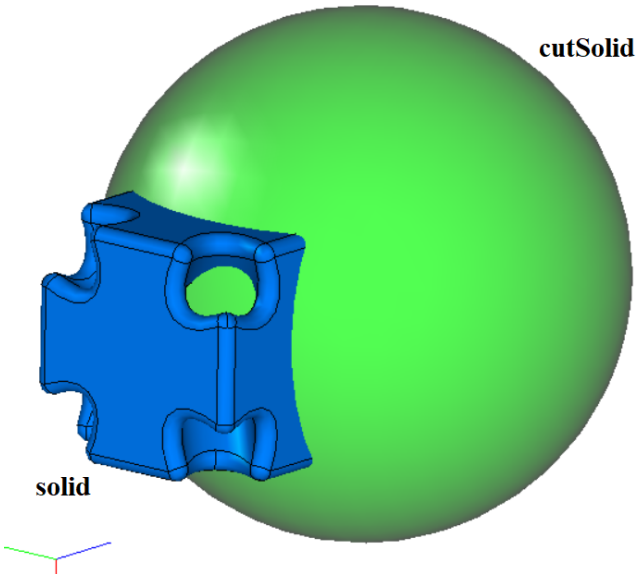


Рис. 11

На рис. 12 приведено построенное тело **result** при *part* = +1 и *cutAsClosed* = *true*. На рис. 13 приведено построенное тело **result** при *part* = -1 и *cutAsClosed* = *true*:

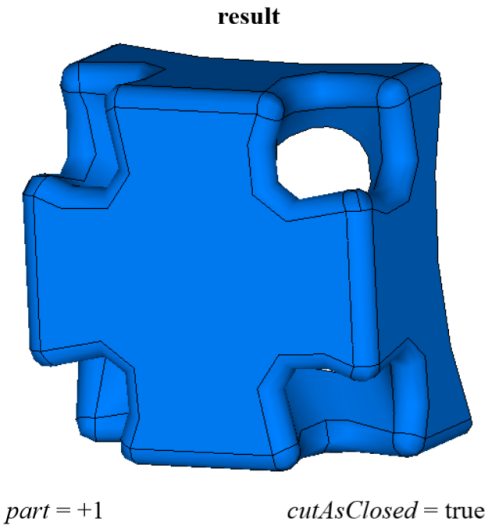


Рис. 12

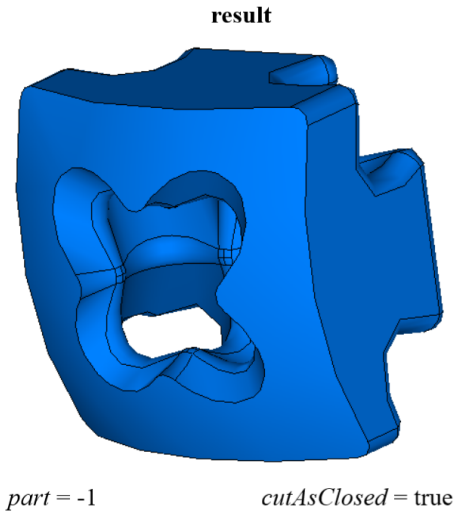


Рис. 13

При **part** = 0 рассматриваемая функция построит тело **result**, в которое войдут все части исходного тела **solid**, полученные после резки (рис. 14):

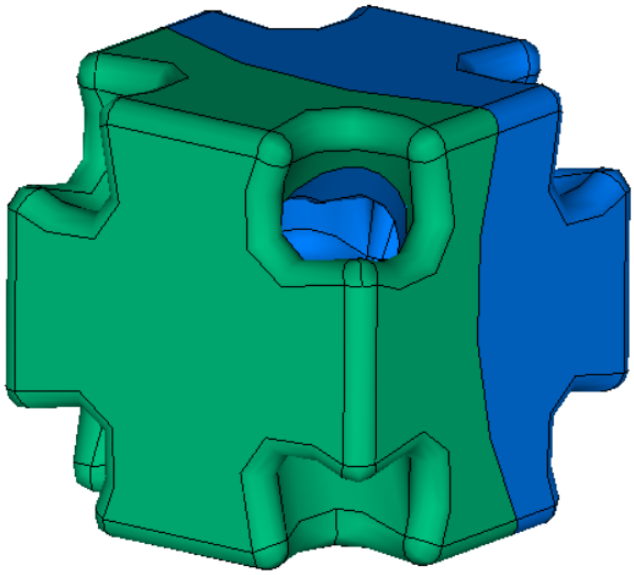


Рис. 14

В тестовом приложении *test.exe* резка тела оболочкой выполняется командами меню «Создать→Тело→На базе тела→Разрезанное оболочкой» и «Создать→Оболочку→На основе оболочки→Разрезанную оболочкой».

3.2.7 Построение симметричного тела

Функция

```
MbResultType SymmetrySolid(       MbSolid &                solid,
                                  MbeCopyMode              sameShell,
                            const MbSymmetrySolidParams &  symParams,
                                  c3d::SolidSPtr &         result );
```

выполняет построение симметричного тела с заданной плоскостью симметрии.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – режим копирования исходного тела;
- **symParams** – параметры операции.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Параметр **sameShell** может принимать одно из четырех значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

В классе MbSymmetrySolidParams для инициализации объекта **symParams** задан конструктор параметров создания симметричного тела относительно плоскости XY локальной системы координат (см. *op_shell_parameter.h*):

```
MbSymmetrySolidParams( const MbPlacement3D & place,
                       const MbSNameMaker &  operNames,
                             ThreeStates     side );
```

где

- **place** – система координат плоскости симметрии;
- **side** – сохраняемая часть исходной оболочки (+1, 0, -1);
- **operNames** – именователь операции.

Если параметр **symParams**.*side* = 1, то функция создаёт симметричное тело с заданной плоскостью симметрии следующим образом: исходное тело режется плоскостью XY локальной системы координат, берётся часть исходного тела, расположенная по положительному направлению оси Z от этой плоскости, строится зеркальная копия выбранной части исходного тела и объединяется с выбранной частью.

Если **symParams**.*side* = -1, то функция объединяет часть исходного тела, расположенную по отрицательному направлению оси Z от плоскости XY локальной системы координат, с зеркальной копией этой части относительно плоскости симметрии.

Параметр **operNames** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 1 показаны исходное тело **solid** и плоскость симметрии **place**:

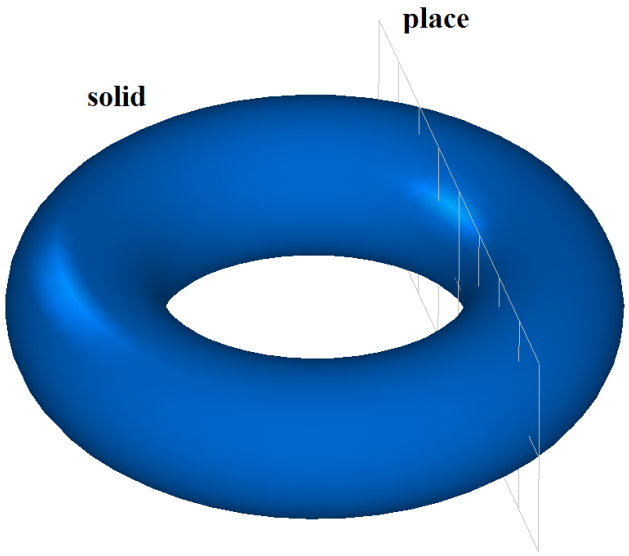


Рис. 1

На рис. 2 приведено построенное тело **result**:

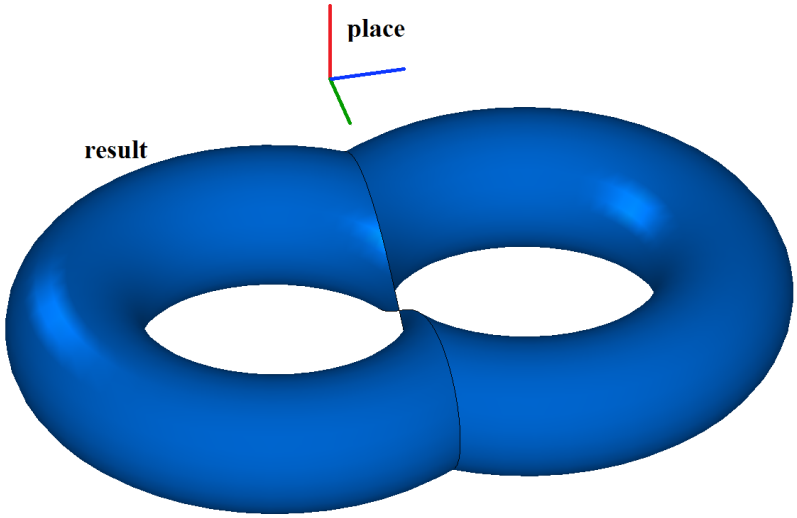


Рис. 2

На рис. 3 приведено построенное тело **result** для плоскости симметрии с противоположной нормалью:

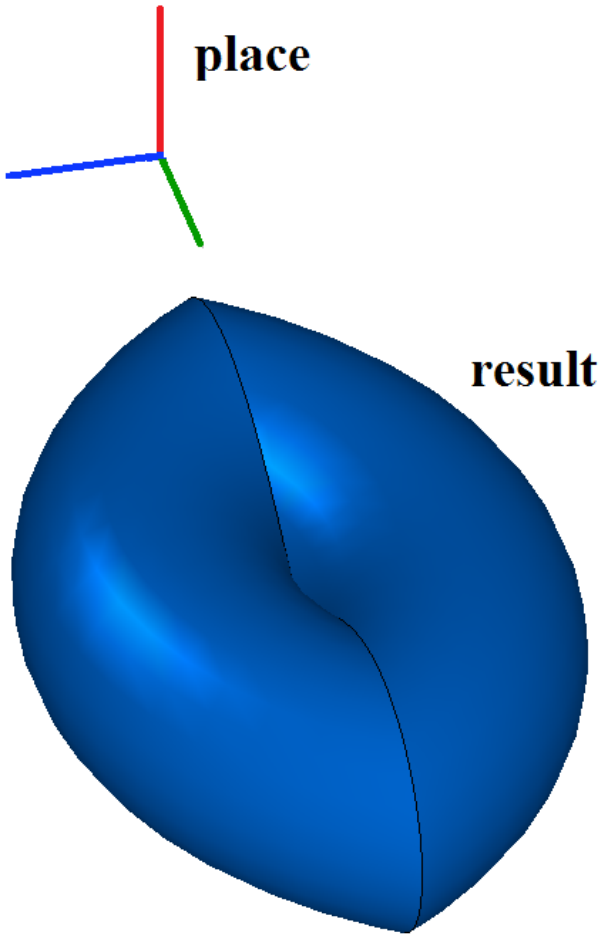


Рис. 3

На рис. 4 показаны исходное тело **solid** и плоскость симметрии **place**, не соприкасающиеся друг с другом:

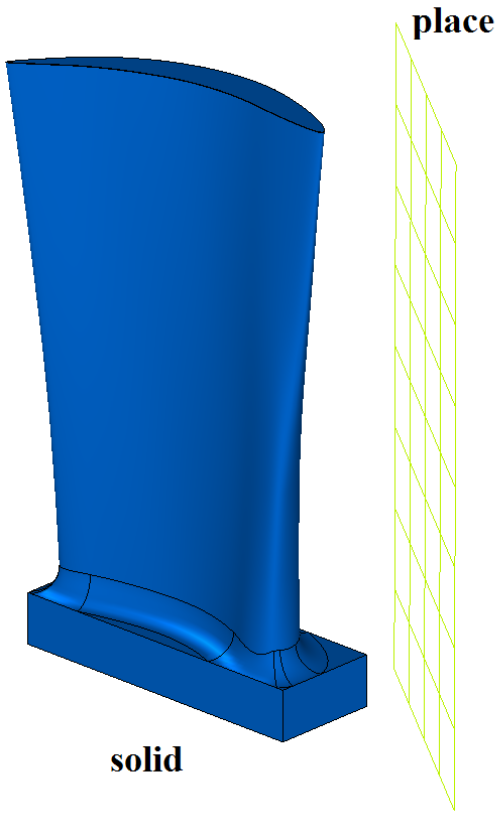


Рис. 4

На рис. 5 приведены исходное тело **solid** и построенное зеркальное тело **result**:

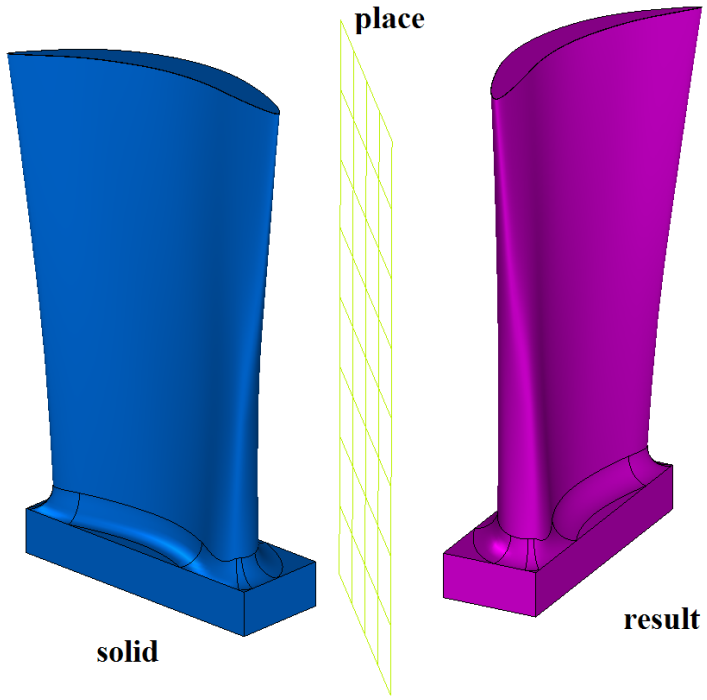


Рис. 5

Функция SymmetrySolid() добавляет в журнал построенного тела строитель MbSymmetrySolid, который содержит все необходимые данные для выполнения операции. Строитель MbSymmetrySolid объявлен в файле *cr_symmetry_solid.h*.

В тестовом приложении *test.exe* построение симметричного тела выполняется командой меню «Создать→Тело→На базе тела→Симметричное».

3.2.8 Скругление вершин и рёбер тела

Функция

```
MbResultType FilletSolid(       MbSolid &               solid,
                                MbeCopyMode             sameShell,
                          const MbShellFilletValues &   filletParams,
                                c3d::SolidSPtr &        result,
                                IProgressIndicator *    progInd = nullptr );
```

выполняет замену указанных рёбер исходного тела гранями, гладко сопрягающими смежные грани указанных рёбер. В поперечном сечении сопрягающие грани могут иметь форму дуги окружности, эллипса, гиперболы, параболы.

Способы скругления ребер тела:

- скругление указанных рёбер тела постоянным радиусом;
- скругление вершин и примыкающих к ним рёбер тела постоянным радиусом (в вершинах должно стыковаться три ребра);
- скругление указанных ребер тела переменным радиусом, задаваемым MbEdgeFunction.function.

Параметры поперечного сечения могут изменяться по заданному закону. Способ построения скругления определяется набором параметров в объекте **filletParams**.

Входными параметрами функции являются:

- **solid** – исходное тело, ребра которого подлежат обработке;
- **sameShell** – режим копирования исходного тела;
- **filletParams** – параметры построения.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Скругление ребер постоянным радиусом

При скруглении ребер постоянным радиусом для инициализации объектов **filletParams** с параметрами операции в классе MbShellFilletValues определен следующий конструктор:

```
MbShellFilletValues(       c3d::EdgesSPtrVector      initCurves,
                           c3d::FacesSPtrVector &    initBounds,
                    const  SmoothValues &            params,
                    const  MbSNameMaker &            names )
: MbShellChamferParams( initCurves, params, names )
, _initBounds( initBounds ) {}
```

где

- **initCurves** – множество скругляемых ребер тела;
- **initBounds** – множество граней для обрезки торцов;
- **params** – параметры скругления рёбер;
- **names** – именователь.

Параметр построения скруглений *params* содержит информацию о форме и способе сопряжения смежных граней обрабатываемых ребер, рис. 1. Класс SmoothValues описан в файле *op_shell_parameter.h*.

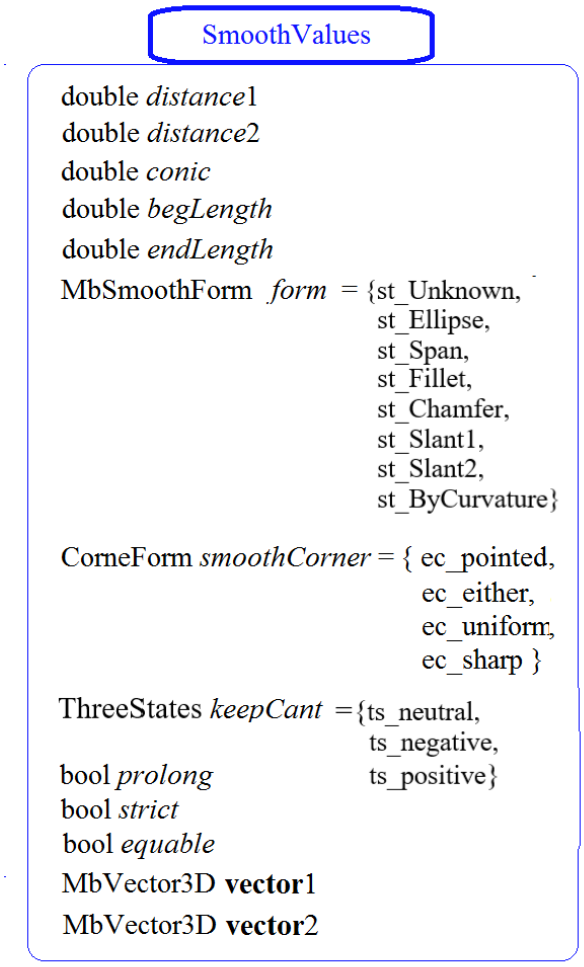


Рис. 1

- **distance1** – радиус кривизны на первой поверхности;
- **distance2** – радиус кривизны на второй поверхности;
- **conic** – коэффициент формы поверхности сопряжения, принимает значения 0, 0.05...0.95 (0 – дуга окружности или эллипса);
- **begLength** – расстояние от начала скругления до точки остановки (UNDEFINED_DBL – остановки нет);
- **endLength** – расстояние от конца скругления до точки остановки (UNDEFINED_DBL – остановки нет);
- **form** – тип сопряжения (скругления) из перечисления MbeSmoothForm;
- **smoothCorner** – способ обработки углов стыковки трёх рёбер;
- **prolong** – флаг продолжения скругления по касательным рёбрам;
- **keepCant** – автоопределение сохранения кромки (*ts_neutral*), сохранения поверхности (*ts_negative*), сохранения кромки (*ts_positive*);
- **strict** – флаг строгости построения: при *false* скруглить хотя бы то, что возможно;

- **equable** – флаг вставки тороидальной поверхности в углах сочленения поверхности сопряжения (для штамповки листового тела);
- **vector1** – вектор нормали к плоскости, по которой выполняется усечение скругления в начале цепочки;
- **vector2** – вектор нормали к плоскости, по которой выполняется усечение скругления в конце цепочки.

Типом скругления управляет параметр **form**. Для скругления ребер используются значения параметра **form** равные *st_Ellipse*, *st_Fillet* и *st_Span*.

При значении **form** = *st_Fillet* функция строит поверхность скругления с заданными радиусами, которые определяются параметрами **distance1** и **distance2**. На рис. 2 приведено скругление с заданными радиусами ребра, соединяющего две цилиндрические поверхности:

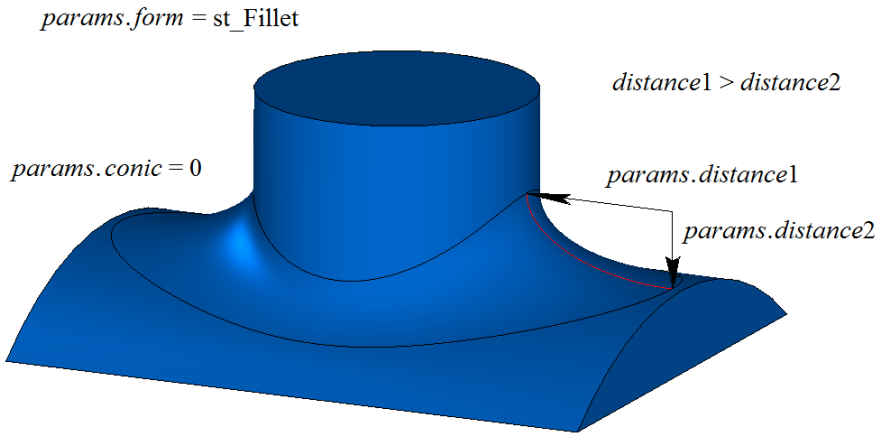


Рис. 2

При **distance1** = **distance2** и **conic** = 0 поверхность скругления строится путем движения сферы, касающейся двух смежных для скругляемого ребра граней. Опорные края грани сопряжения проходят по точкам касания сферы и соответствующей смежной грани. Поперечное сечение грани сопряжения представляет собой дугу окружности. На рис. 3 приведено скругление с заданными одинаковыми радиусами ребра, соединяющего две цилиндрические поверхности:

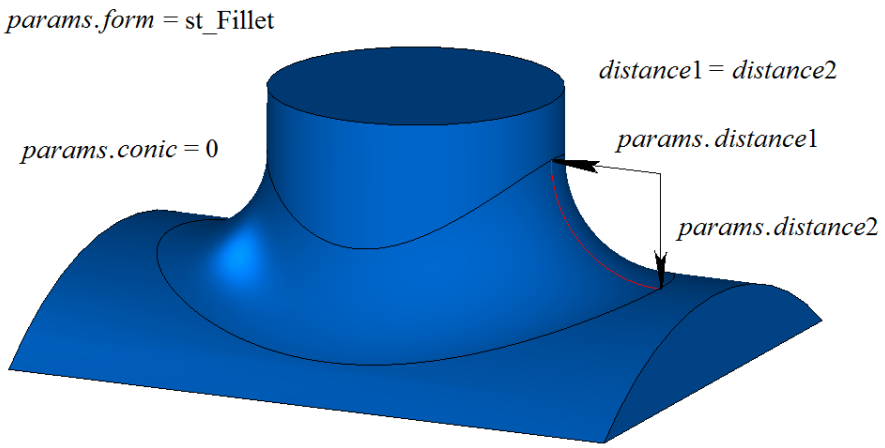


Рис. 3

Коэффициент **conic** управляет формой поверхности скругления. При **conic** = 0 (макрос _ARC_) сечение поверхности сопряжения имеет форму дуги окружности или эллипса с заданными радиусами. Кроме нулевого значения коэффициент формы может принимать значения от 0.05 до 0.95.

При **conic** = 0.5 поперечное сечение грани скругления представляет собой дугу параболы.

При **conic** < 0.5 поперечное сечение грани скругления представляет собой дугу эллипса.

При **conic** > 0.5 поперечное сечение грани скругления представляет собой дугу гиперболы.

На рис. 4, 5 приведены скругления одинаковыми радиусами ребра, соединяющего две цилиндрические поверхности, с разными коэффициентами формы:

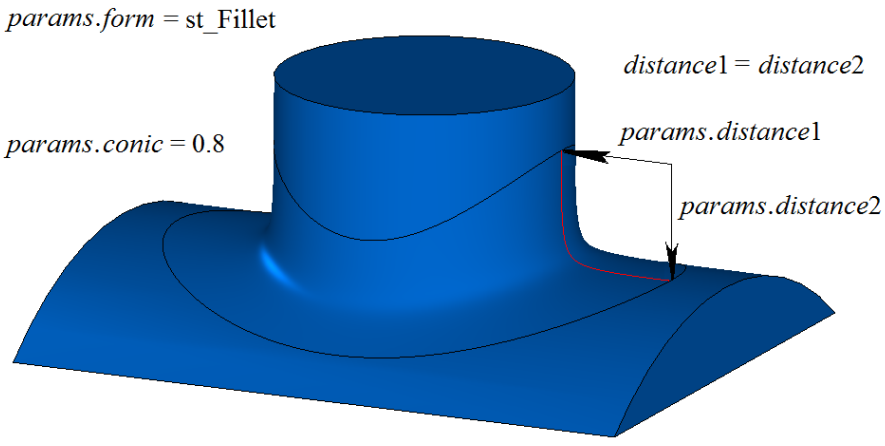


Рис. 4

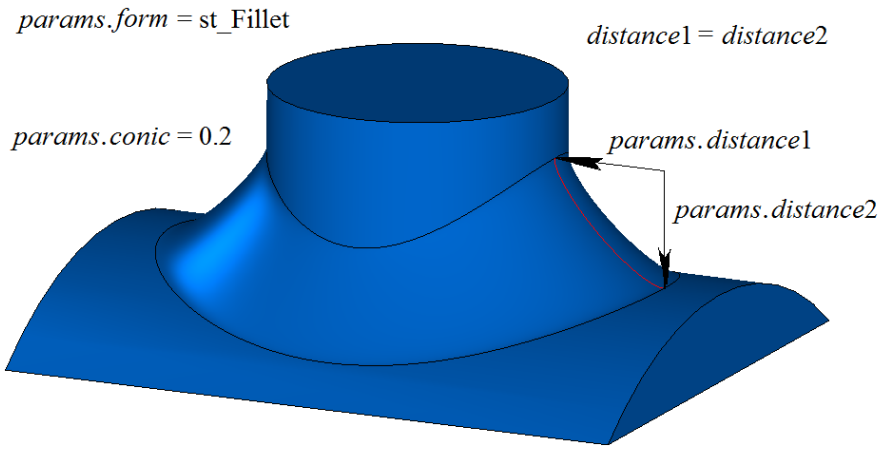


Рис. 5

При значении **form** = *st_Span* функция строит поверхность скругления с заданной хордой. Параметры **distance1** и **distance2** равны и определяют расстояние между опорными краями грани сопряжения. Поперечное сечение грани скругления представляет собой дугу окружности. В общем случае в каждом поперечном сечении грани скругления радиус дуги различный, а параметры **distance1** и **distance2** равны хорде дуги окружности. На рис. 6 приведено скругление с заданной хордой ребра, соединяющего две цилиндрические поверхности:

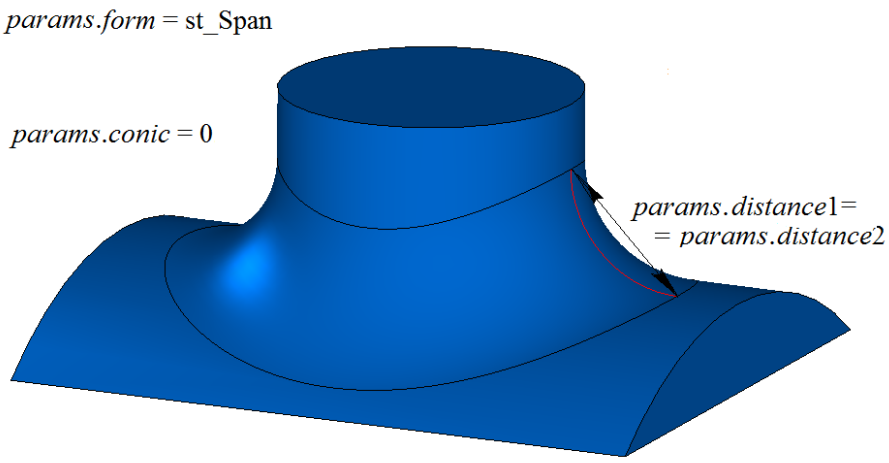


Рис. 6

На рис. 7 приведено скругление с заданной хордой ребра, соединяющего две цилиндрические поверхности, с ненулевым коэффициентом формы:

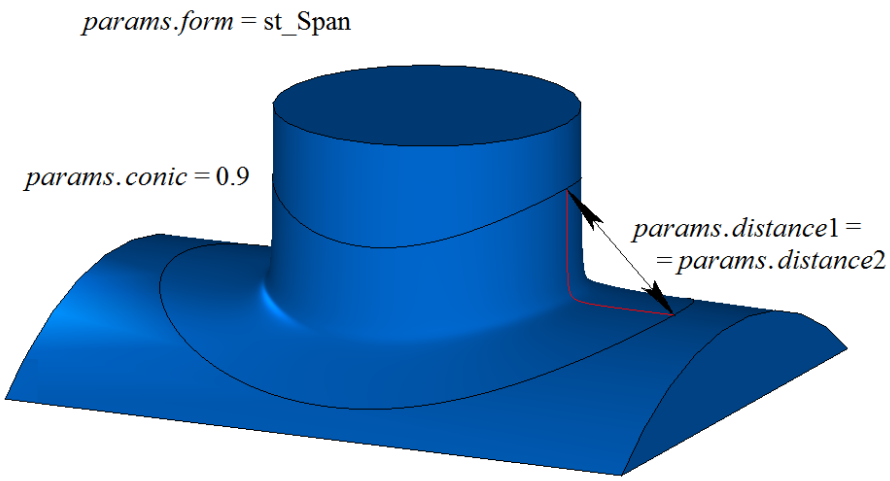


Рис. 7

На рис. 8 приведен пример остановки скругления на расстоянии **begLength** от начальной вершины и на расстоянии **endLength** от конечной вершины обрабатываемого ребра. Если не требуется останавливать сопряжение, то расстояния **begLength** и **endLength** должны принимать значение *UNDEFINED_DBL*. По умолчанию остановка скругления выполняется гранями, перпендикулярными скругляемому ребру. Можно переопределить поведение остановки скругления, используя вектор **vector1** как нормаль останавливающей грани в начале сопряжения и вектор **vector2** как нормаль останавливающей грани в конце сопряжения.

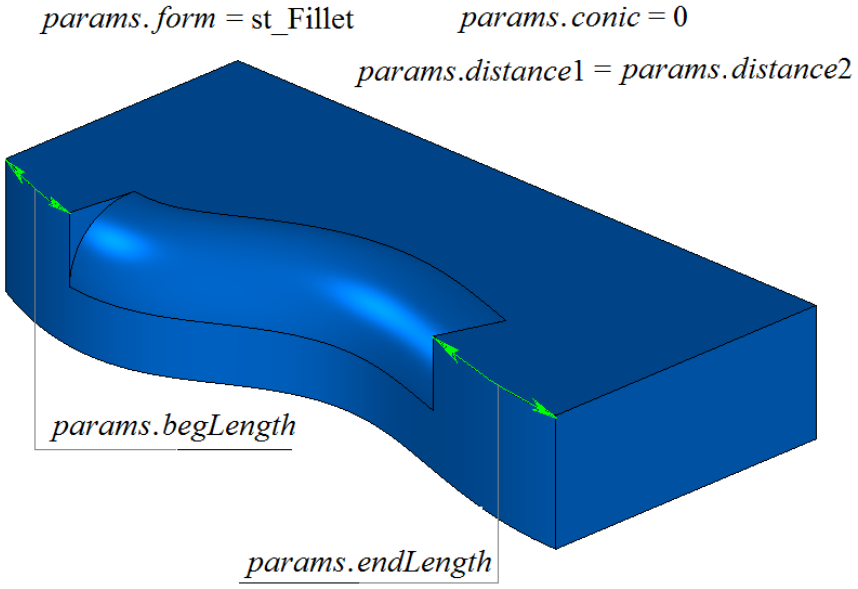


Рис. 8

На примере тела, приведенного на рис. 9, продемонстрируем работу флагов *prolong* и *keepCant* при скруглении выделенного на рисунке ребра.

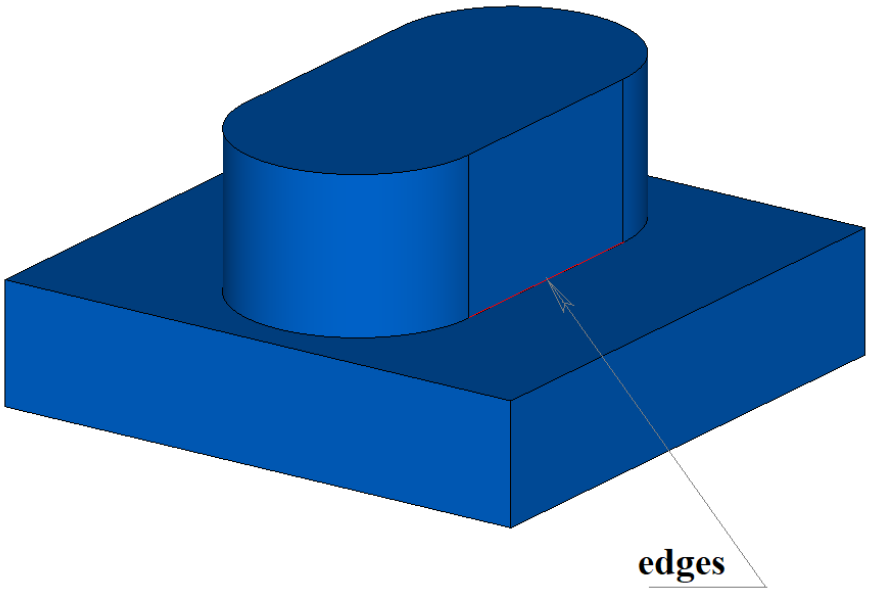


Рис. 9

Флаг *prolong* указывает на то, какие ребра тела должны быть обработаны. Если *prolong* = *false*, то обработке подлежат только те ребра, которые указаны в контейнере **initCurves**, рис. 10.

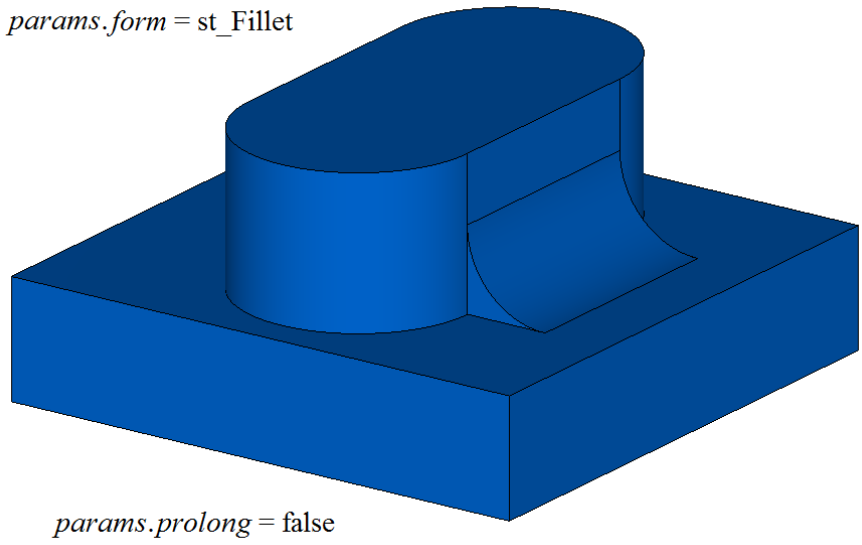


Рис. 10

Если *prolong* = true, то обработке подлежат ребра, указанные в контейнере **initCurves**, а также ребра, гладко стыкующиеся с ними, рис. 11:

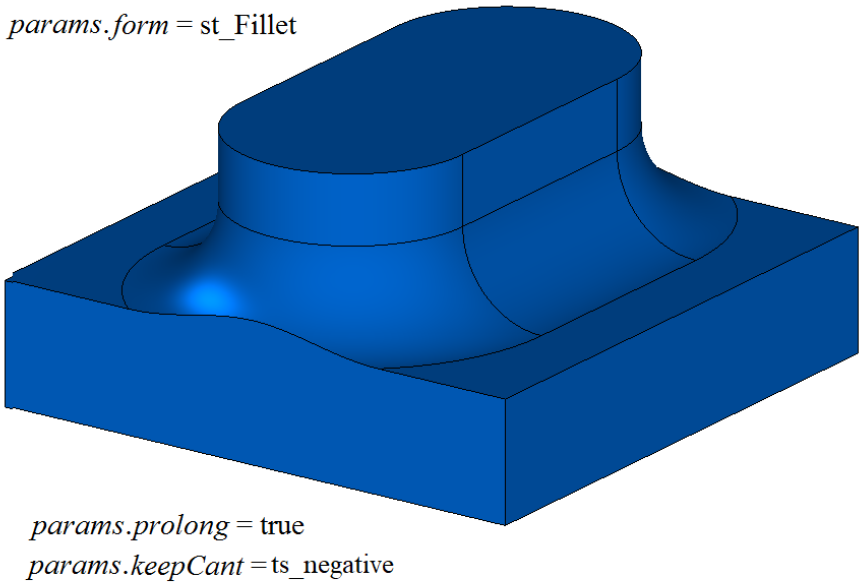


Рис. 11

Параметр **keepCant** управляет обработкой ситуаций, когда опорные края грани сопряжения выходят за границу смежной грани.

Если *keepCant* = *ts_negative*, то при выходе опорного края грани сопряжения за границу смежной грани через острое ребро грань сопряжения не меняет своей формы, а «подрезается» соседней гранью, как приведено на рис. 11.

Если *keepCant* = *ts_positive*, то при выходе опорного края грани сопряжения за границу смежной грани через острое ребро грань сопряжения меняет свою форму и проходит опорным краем по границе, сохраняя ее неизменной, как приведено на рис. 12:

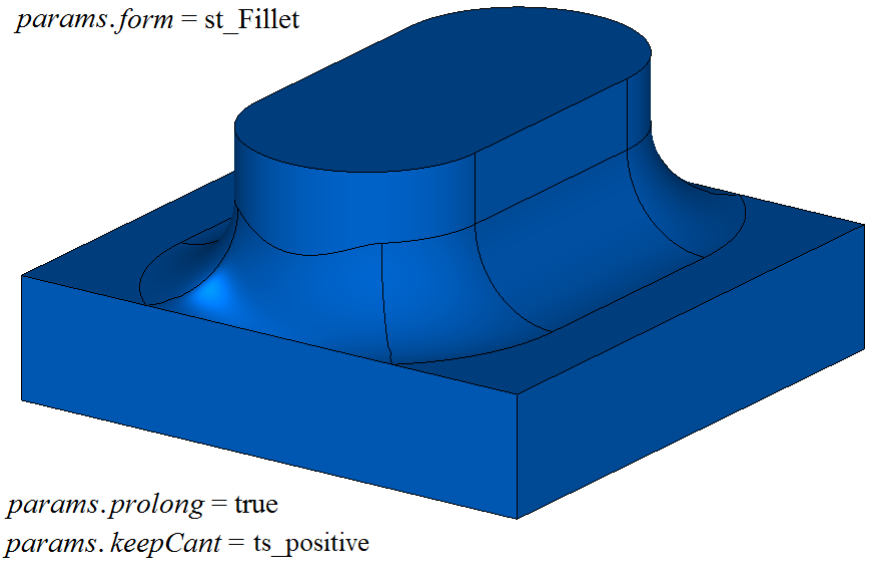


Рис. 12

Если *keepCant* = *ts_neutral*, то при выходе опорного края грани сопряжения за границу смежной грани через гладкое ребро грань сопряжения заменяет смежную грань на соседнюю, изменяя форму на данном участке, рис. 13:

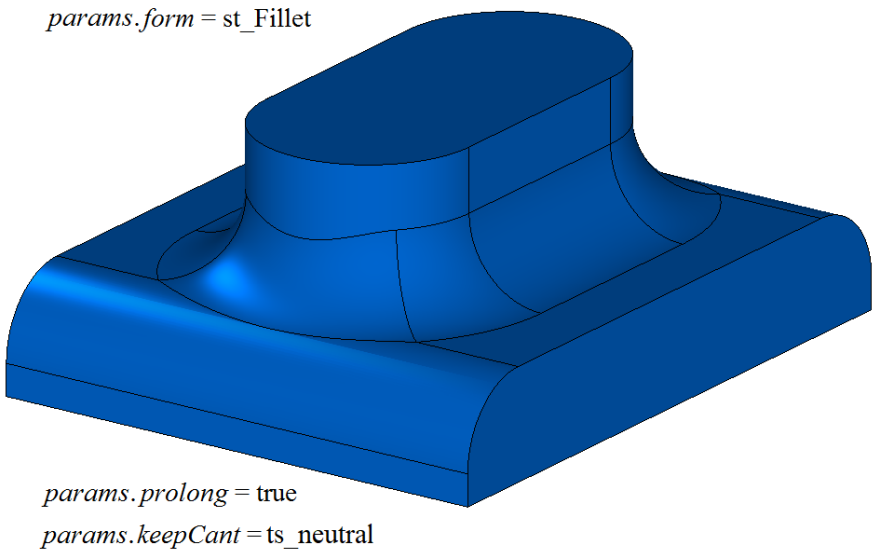


Рис. 13

На рис. 14 приведено скругление четырех ребер, выполненное одним вызовом функции с флагом **equable** = *false* (в углах сочленения тороидальная поверхность отсутствует):

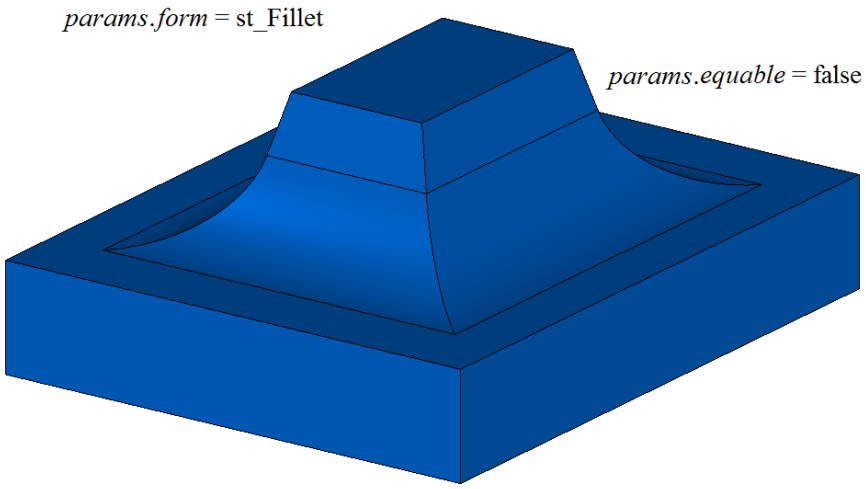


Рис. 14

На рис. 15 приведено скругление четырех ребер, выполненное одним вызовом функции с флагом **equable** = *true*, указывающим на необходимость вставлять тороидальную поверхность в углы сочленения поверхностей сопряжения:

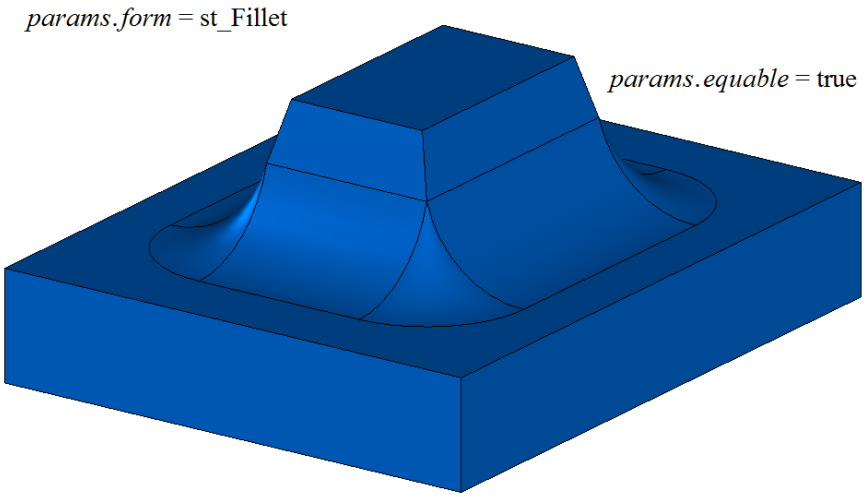


Рис. 15

При построении скругления трех ребер, стыкующихся в одной вершине, параметр **smoothCorner** определяет способ обработки скругления чемоданных углов. Если **smoothCorner** = *ec_pointed*, то обработка углов, в которых стыкуются три ребра одинаковой выпуклости, отсутствует (рис. 16):

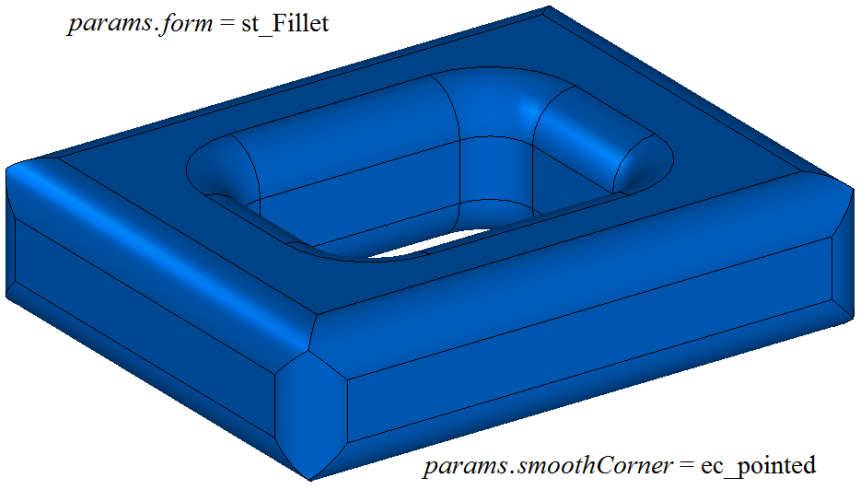


Рис. 16

Если **smoothCorner** = *ec_uniform*, то углы, в которых стыкуются три ребра различной выпуклости, обрабатываются одинаковым образом – если в точке стыкуются два выпуклых (вогнутых) и одно вогнутое (выпуклое) ребро, то первыми обрабатываются два выпуклых (вогнутых) ребра (рис. 17):

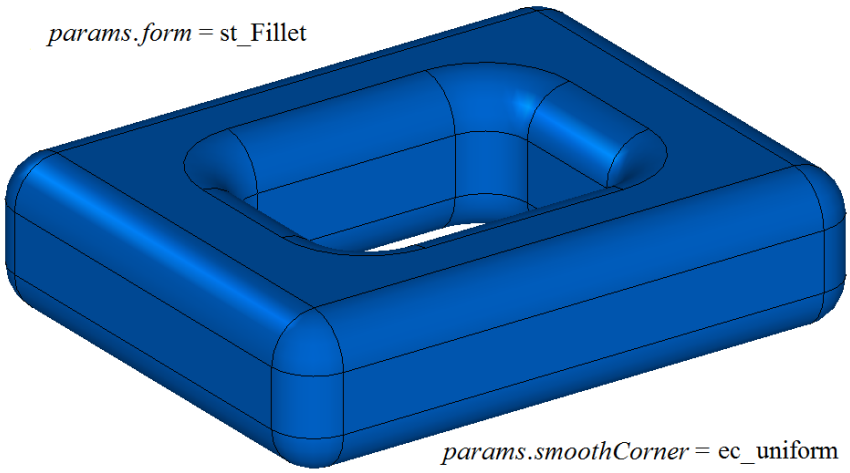


Рис. 17

Если **smoothCorner** = *ec_sharp*, то углы, в которых стыкуются три ребра различной выпуклости, обрабатываются одинаковым образом, как показано на рис. 18. Если в точке стыкуются два выпуклых (вогнутых) и одно вогнутое (выпуклое) ребро, то первыми обрабатываются выпуклые (вогнутые) ребра.

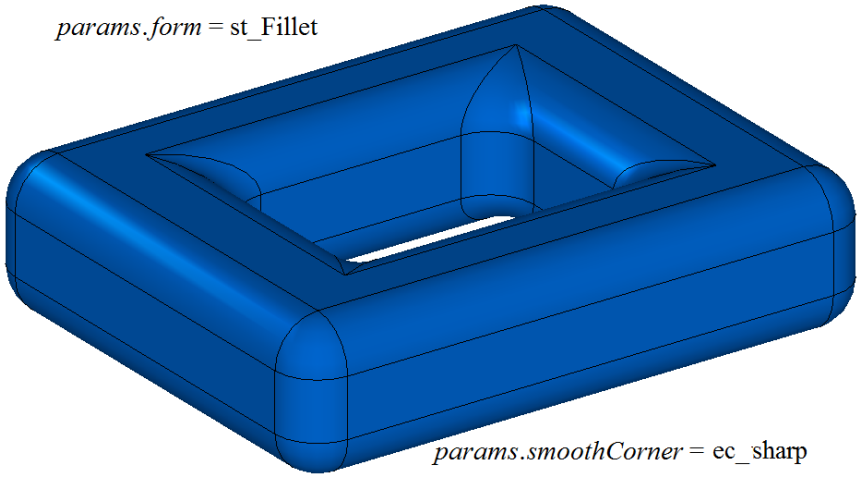


Рис. 18

Если **smoothCorner** = *ec_either*, то углы, в которых стыкуются три ребра различной выпуклости, могут быть обработаны разным образом – стыкующиеся в одной точке три ребра обрабатываются в порядке внутренней нумерации ребер без учета выпуклости и вогнутости.

В неоднозначной ситуации на торцах грани сопряжения может быть задействован параметр **initBounds** с гранями тела **solid**, которые следует использовать для обрезки граней сопряжения. Пример использования параметра **initBounds** приведен на рис. 19 и 20:

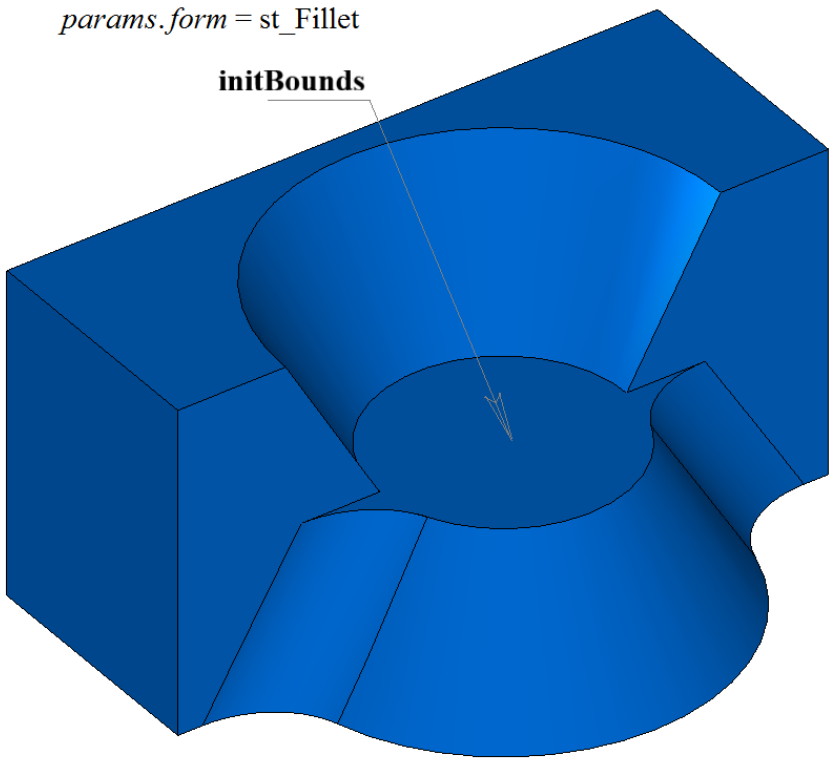


Рис. 19

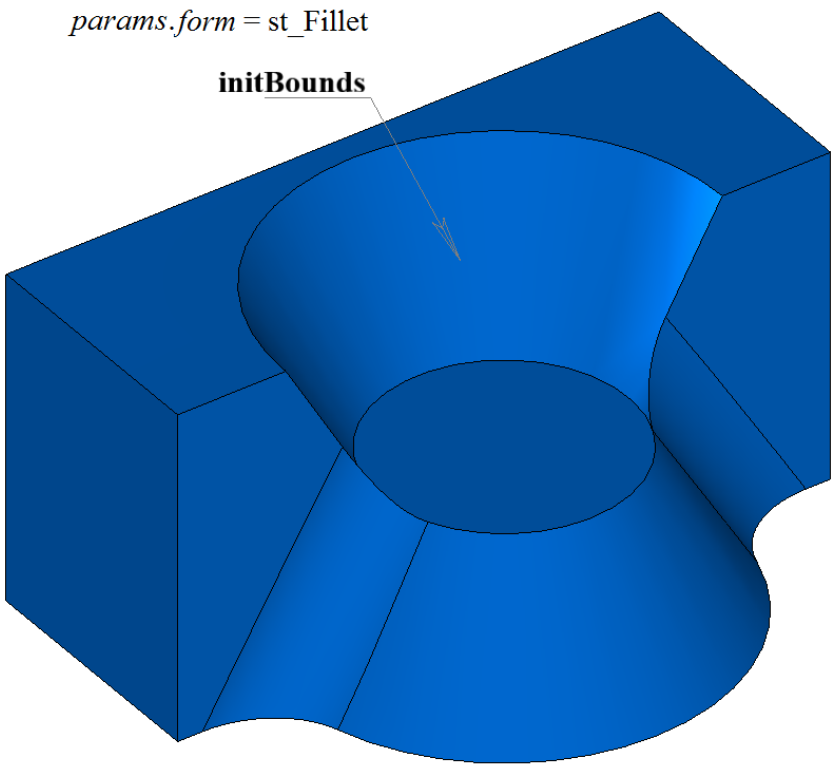


Рис. 20

Параметр **initBounds** может использоваться для остановки граней сопряжения в начале и конце. При этом грани, определяемые параметром **initBounds**, должны принадлежать исходному телу **solid**.

На рис. 21 приведена модель, у которой требуется построить скругления указанных ребер, полностью охватывающие отверстие и выступ:

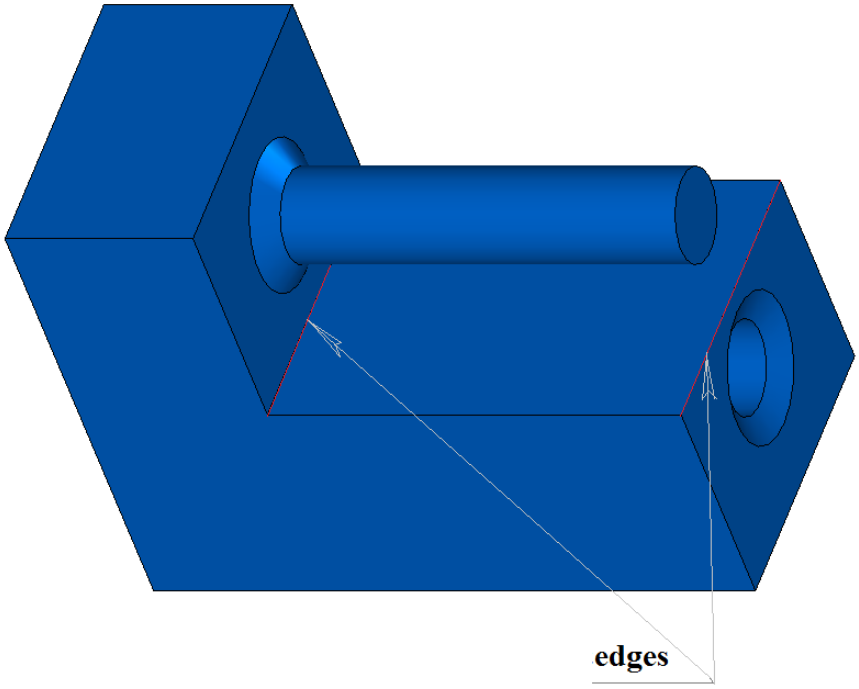


Рис. 21

Результат построения скругления с обходом препятствий приведен на рис. 22:

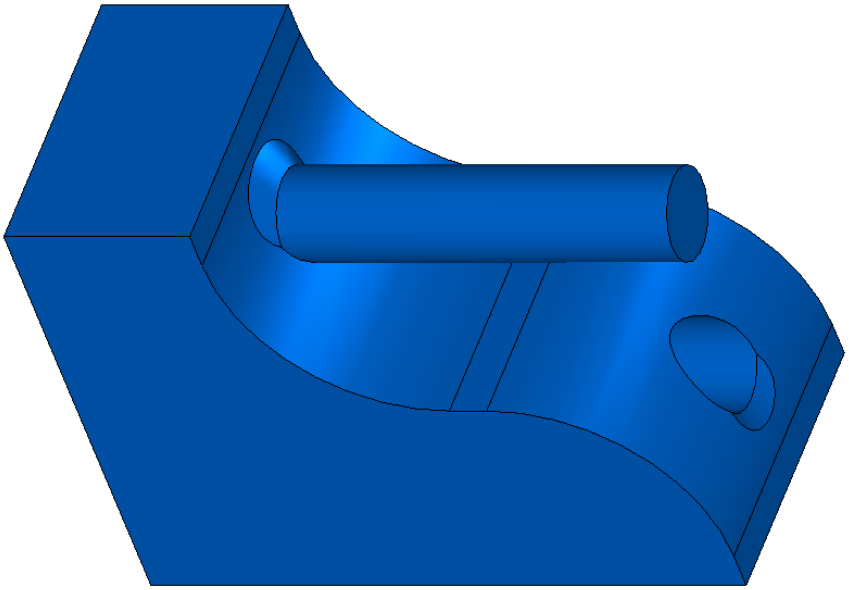


Рис. 22

На рис. 23 приведен пример одновременного скругления группы из шести ребер, стыкующихся в общей вершине:

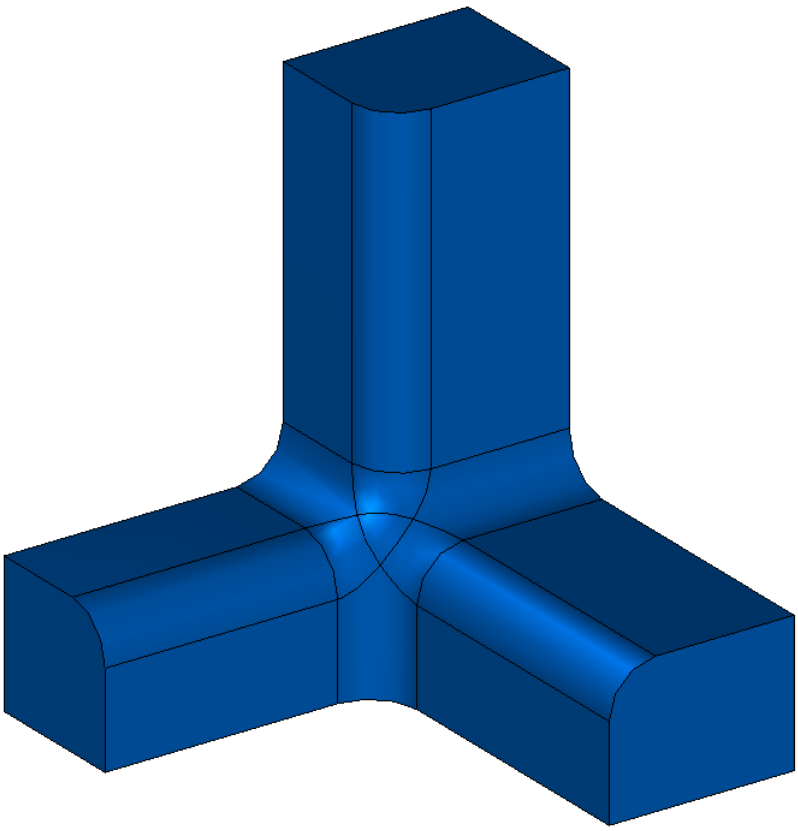


Рис. 23

На рис. 24 приведен пример скругления нескольких групп из четырех ребер, стыкующихся в общих вершинах. Особенностью скругления служит то обстоятельство, что группы связаны между собой и могут обрабатываться только одновременно. Исходное тело, для показанного на рис. тела, получено путем вычитания из куба четырех цилиндров, оси которых совпадают с диагоналями куба.

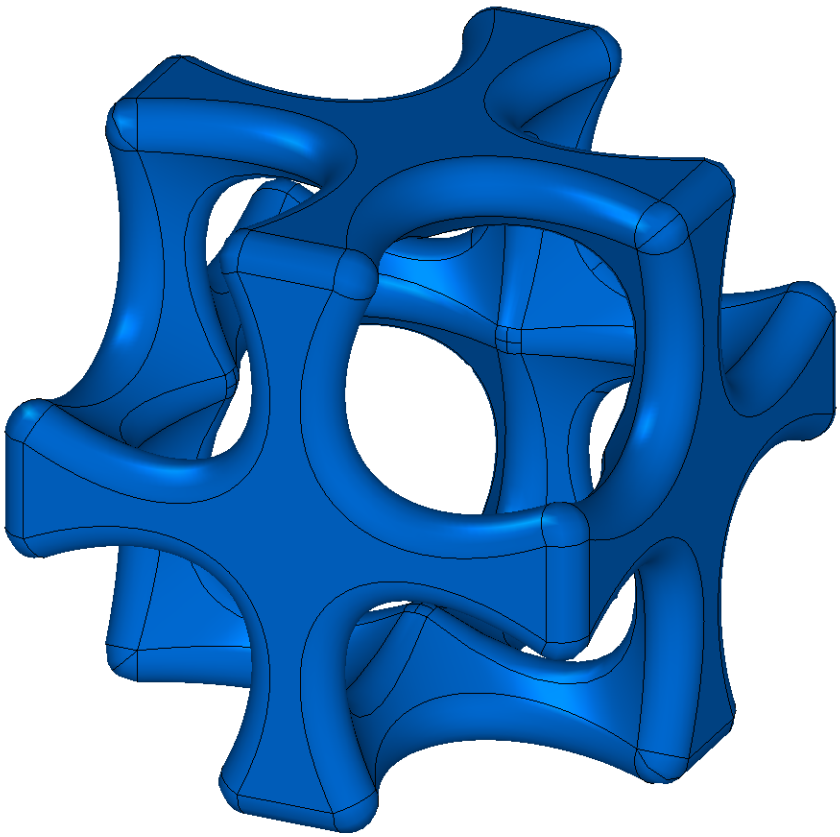


Рис. 24

Когда в методах построения скругления параметры *distance1* и *distance2* не равны и для обработки заданы несколько ребер, то трудно разобраться, что чему соответствует. В такой ситуации предлагается использовать вспомогательные методы: SmoothPhantom(), SmoothSequence(), SmoothPositionData(), объявленные в файле *action_phantom.h*.

Функция SmoothPhantom() строит упрощенные поверхности для имитации будущих скруглений.

Функция SmoothSequence() строит последовательности ребер, к которым по желанию могут быть добавлены гладко стыкующиеся ребра, обрабатываемые совместно.

Функция SmoothPositionData() вычисляет тройку опорных точек скругления, которые позволяют оценить заданные параметры скругления.

Обычно с помощью указанных функций рисуется некий фантом, по которому можно понять, что получится в результате операции, и дается возможность поменять параметры при необходимости.

Перед построением скруглений обрабатываемые ребра сортируются, при необходимости или по требованию к ним добавляются гладко стыкующиеся ребра, и из ребер создаются последовательности гладко стыкующихся ребер.

Параметры **distance1** и **distance2** привязаны к первой и второй поверхностям и, соответственно, к кривой пересечения первого ребра в последовательности гладко стыкующихся ребер.

Поверхности можно получить функциями кривой пересечения GetCurveOneSurface() и GetCurveTwoSurface().

По **distance1** и **distance2** первого ребра определяются параметры для остальных ребер каждой последовательности. Так, если для первого ребра указатель edge→GetIntersectionCurve().GetSurfaceOne() равен указателю **edge**→GetFacePlus()→GetSurface().GetSurface(), то при построении скругления **distance1** будет соответствовать радиусу для грани **facePlus**, а **distance2** будет соответствовать радиусу для грани **faceMinus** первого ребра последовательности.

При построении скругления ребер рассматриваемая функция добавляет в журнал построенного тела строитель MbFilletSolid, который объявлен в файле *cr_fillet_solid.h*.

В тестовом приложении *test.exe* обработка ребер тела выполняется командами меню «Создать→Тело→Обработкой ребер→Скругление по радиусу», «Создать→Тело→Обработкой ребер→Скругление по хорде».

Скругление ребер переменным радиусом

Функция аналогична описанной в предыдущем параграфе, и отличается от нее параметром **initFunctions**. При скруглении ребер сопрягающие грани в поперечном сечении могут иметь форму дуги окружности переменного радиуса.

При скруглении ребер переменным радиусом для инициализации объектов **filletParams** с параметрами операции в классе MbShellFilletValues определен следующий конструктор:

```
MbShellFilletValues( const std::vector<MbEdgeFunction> &        initFunctions,
                           c3d::FacesSPtrVector &               initBounds,
                     const MbSNameMaker &                       names,
                     const SmoothValues &                       params )
: MbShellChamferParams( c3d::EdgesSPtrVector(), params, names )
, _initFunctions( initFunctions )
, _initBounds( initBounds ) {}
```

где

- **initFunctions** – множество скругляемых ребер тела с функциями изменения радиуса;
- **initBounds** – множество граней для обрезки торцов;
- **params** – параметры скругления рёбер;
- **names** – именователь.

Параметр **initFunctions** содержит обрабатываемые ребра тела **solid** и функции изменения радиуса вдоль ребра. Каждый элемент множества **initFunctions** состоит из указателя на ребро и указателя на скалярную функцию, на значения которой будут умножены радиусы скругления **distance1** и **distance2**, рис. 25:

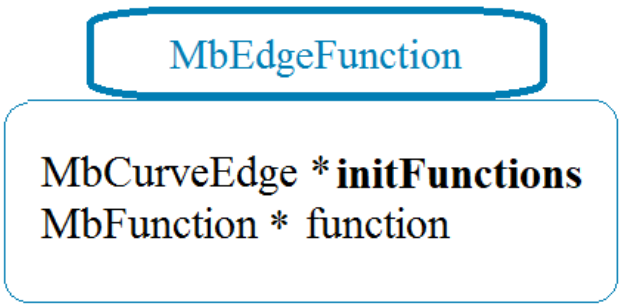


Рис. 25

Для скругления ребер переменного радиуса используется значение параметра **form** равное *st_Fillet*. Для каждой точки обрабатываемого ребра **initFunctions**[i].**initFunctions**→Point(*t*, **point**) радиусы поверхности скругления равны параметрам **distance1** и **distance2**, умноженным на значение функции ребра **initFunctions**[i].function→Value(*t*). На рис. 26 приведено скругление с переменными радиусами ребра прямоугольной призмы. При **distance1** = **distance2** и **conic** = 0 поверхность скругления строится путем движения сферы переменного радиуса, касающейся двух смежных для скругляемого ребра граней. Опорные края грани сопряжения проходят по точкам касания сферы и соответствующей смежной грани. Поперечное сечение грани сопряжения представляет собой дугу окружности.

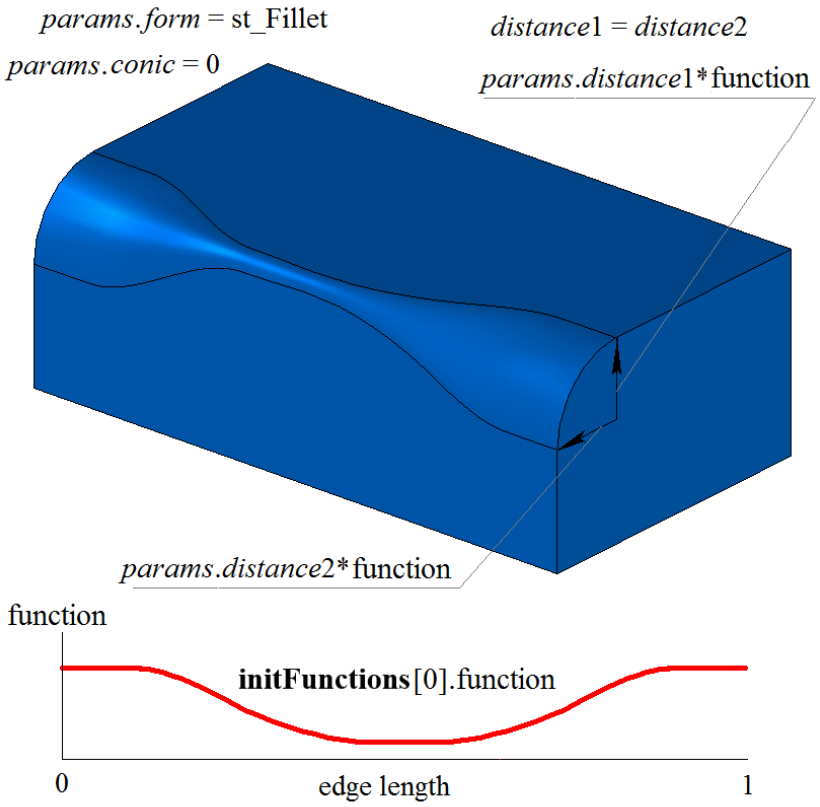


Рис. 26

На рис. 27 приведено эллиптическое скругление с переменными радиусами ребра прямоугольной призмы:

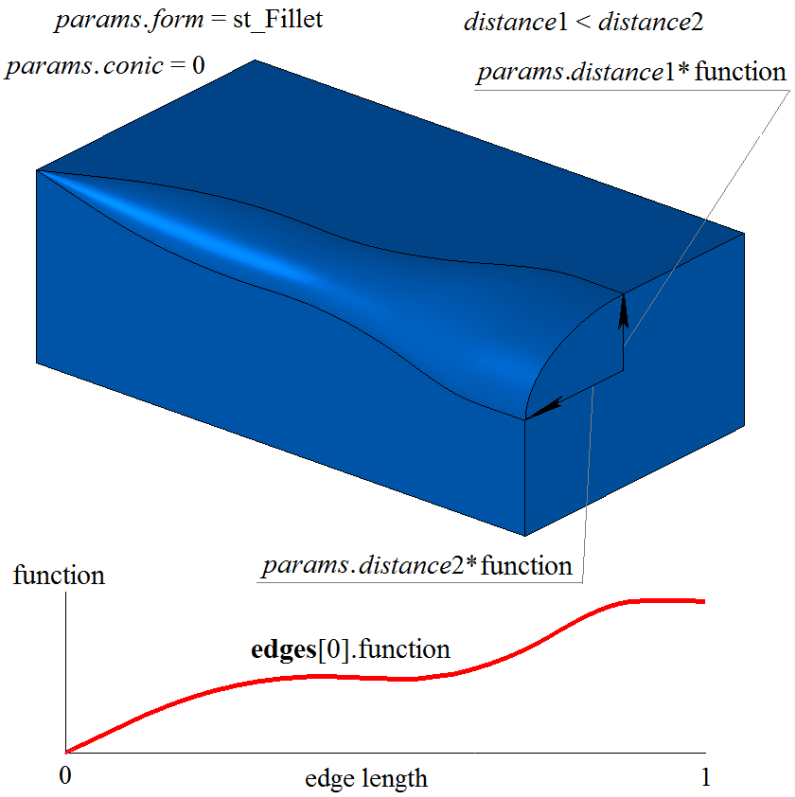


Рис. 27

На рис. 28, 29 приведены скругления с переменными радиусами ребра прямоугольной призмы с разными коэффициентами формы **conic**:

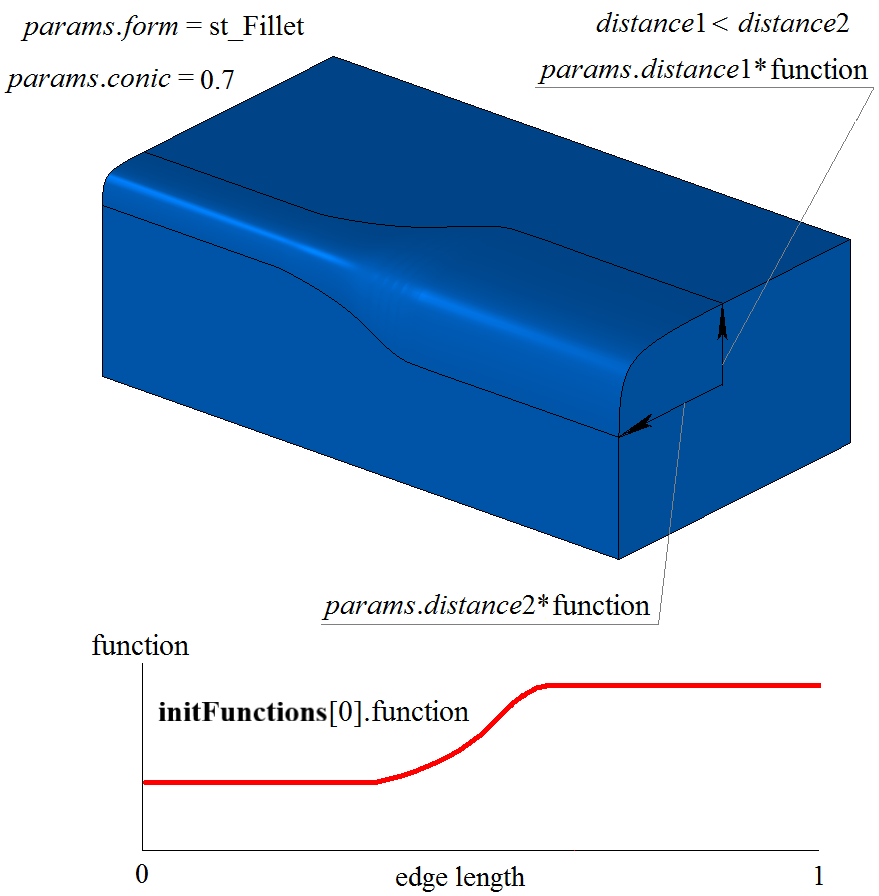

params.form = st_Fillet
params.conic = 0.7
distance1 < distance2
params.distance1*function
params.distance2*function
function
initFunctions[0].function
0
edge length
1

Рис. 28

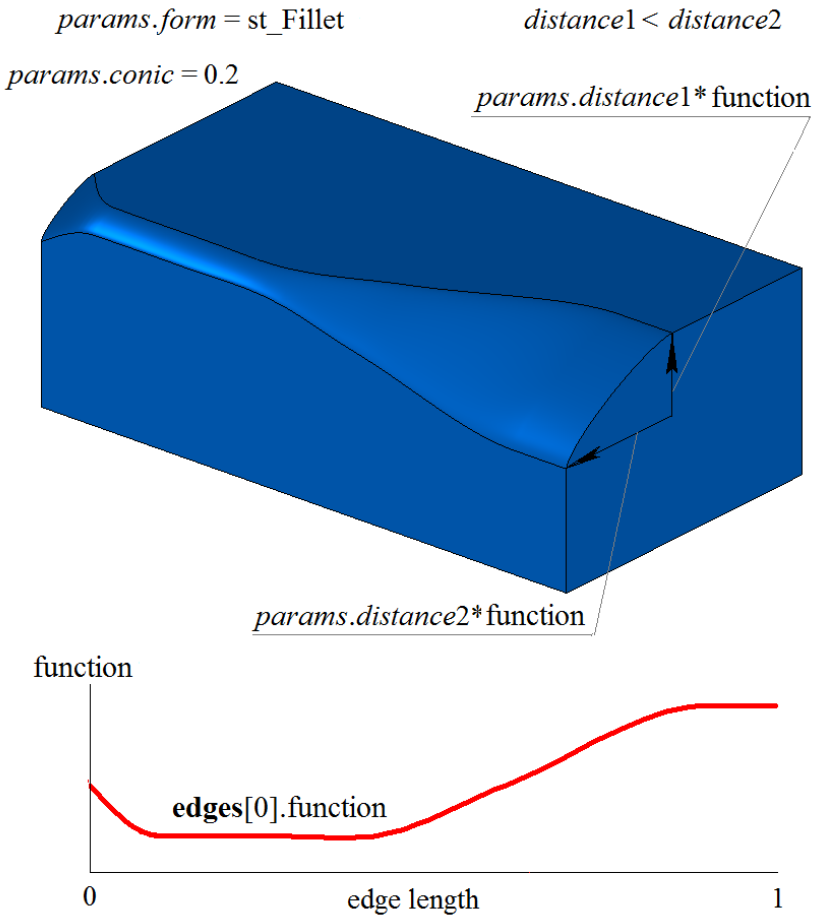


Рис. 29

На рис. 30 приведен пример остановки скругления на расстоянии **begLength** от начальной вершины и на расстоянии **endLength** от конечной вершины обрабатываемого ребра. При задании параметров остановки скругления метод изменения радиусов скругления задается для всего ребра. Если не требуется останавливать сопряжение, то расстояния **begLength** и **endLength** должны принимать значение *UNDEFINED_DBL*.

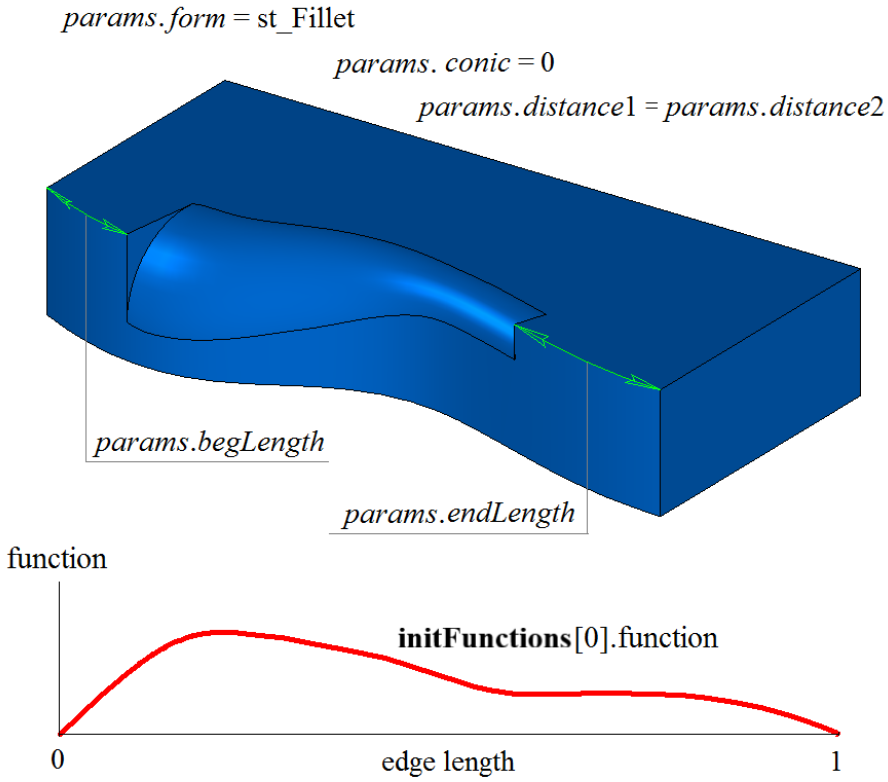


Рис. 30

Для ребер продолжения скругления функция изменения радиуса принимает константное значение, равное значению функции на краю предыдущего гладко стыкующегося ребра. На рис. 31 приведено исходное тело, ребра, подлежащие скруглению, и функция изменения радиуса для указанных ребер:

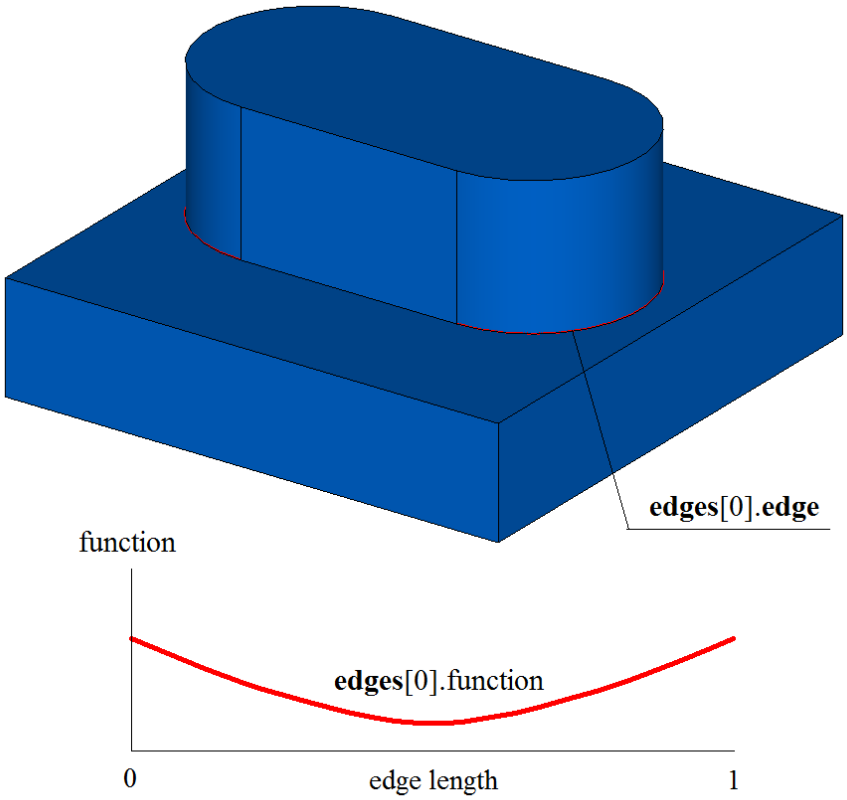


Рис. 31

На рис. 32 приведен результат работы функции для исходного тела. На рисунке видно, что на краях функция изменения радиуса была подобрана так, чтобы обеспечить гладкое сопряжение с соседней гранью скругления:

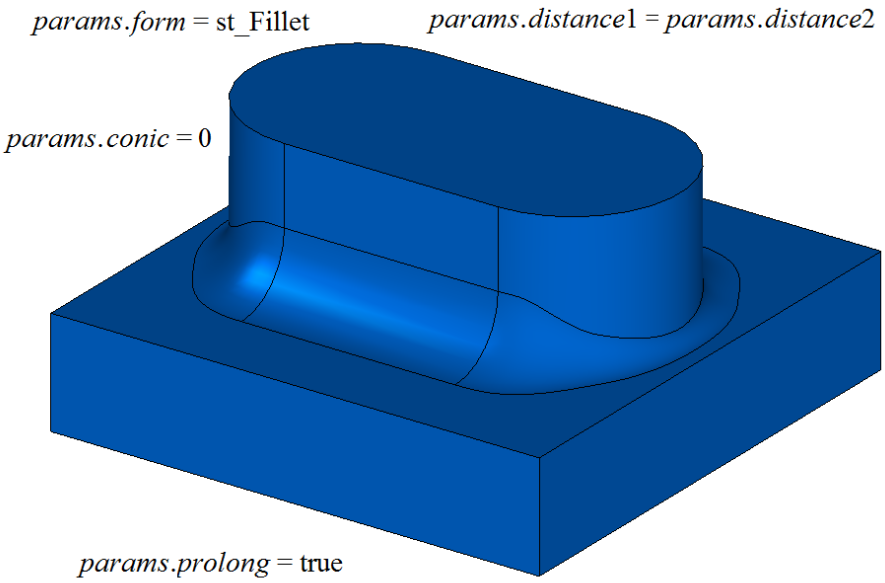


Рис. 32

В тестовом приложении *Test.exe* обработка ребер тела переменным радиусом выполняется командой меню «Создать→Тело→Обработкой ребер→Скругление переменное».

Скругление вершин и примыкающих к ним рёбер постоянным радиусом

При скруглении вершин и примыкающих к ним рёбер постоянным радиусом для инициализации объектов **filletParams** с параметрами операции в классе MbShellFilletValues определен следующий конструктор:

```
MbShellFilletValues(       c3d::EdgesSPtrVector &    initCurves,
                           c3d::FacesSPtrVector &    initBounds,
                           c3d::VerticesSPtrVector & initVertices,
                    const CornerValues &             cornerData,
                    const SmoothValues &             params,
                    const MbSNameMaker &             names )
: MbShellChamferParams( initCurves, params, names )
, _initBounds  ( initBounds )
, _initVertices( initVertices )
, _cornerData  ( cornerData ) {}
```

где

- **initCurves** – множество скругляемых ребер тела;
- **initBounds** – множество граней для обрезки торцов;
- **initVertices** – множество скругляемых вершин;
- **cornerData** – параметры скругления вершин;
- **params** – параметры скругления рёбер;
- **names** – именователь.

Параметр построения скруглений **params** содержит информацию о форме и способе сопряжения смежных граней обрабатываемых ребер (см. *Скругление ребер постоянным радиусом*).

Параметры скругления вершины **cornerData**, в которой стыкуются три ребра, содержат информацию, необходимую для выполнения операции, рис. 33. Класс CornerValues описан в файле *op_shell_parameter.h*.

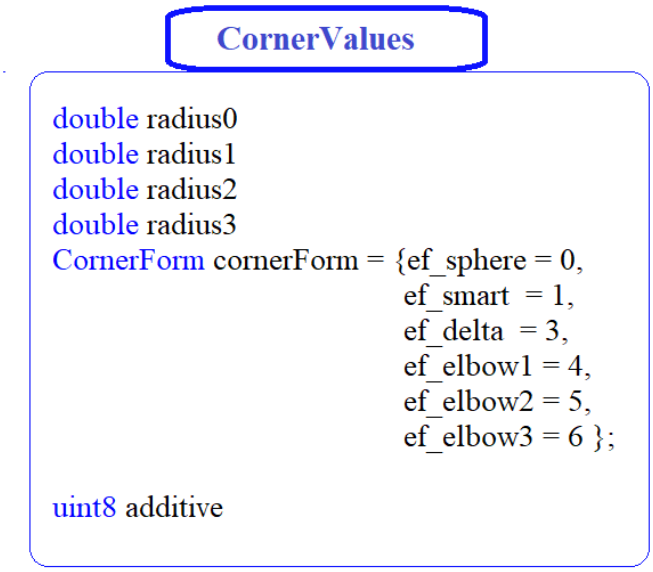


Рис. 33

- **radius0** – радиус сферы в вершине;
- **radius1** – радиус первого ребра вершины;
- **radius2** – радиус второго ребра вершины;
- **radius3** – радиус третьего ребра вершины;
- **cornerForm** – способ скругления вершины стыковки трёх рёбер;
- **additive** – сдвиг в нумерации рёбер вершины (добавка к номеру ребра).

Типом скругления вершины управляет параметр **cornerForm**. Для скругления вершин используются следующие значения параметра **cornerForm**:

ef_sphere – скругление вершины сферической поверхностью;

ef_smart – скругление вершины гладкой поверхностью;

ef_delta – скругление вершины треугольной поверхностью;

ef_elbow1 – скругление вершины четырёхугольной поверхностью (четвёртая сторона располагается напротив первого ребра вершины);

ef_elbow2 – скругление вершины четырёхугольной поверхностью (четвёртая сторона располагается напротив второго ребра вершины);

ef_elbow3 – скругление вершины четырёхугольной поверхностью (четвёртая сторона располагается напротив третьего ребра вершины).

При значении **cornerForm** = *ef_sphere* вершины, в которых стыкуются три ребра различной выпуклости, скругляются сферической поверхностью как показано на рис. 34:

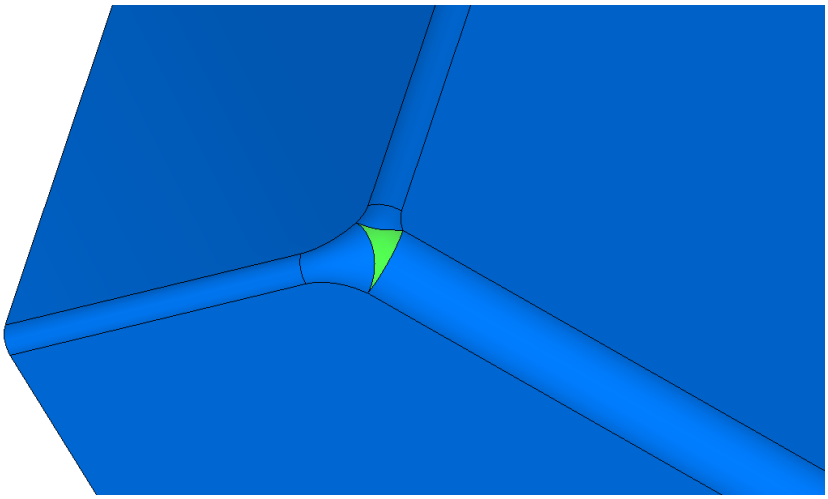


Рис. 34

При значении **cornerForm** = *ef_smart* вершины, в которых стыкуются три ребра различной выпуклости, скругляются гладкой поверхностью как показано на рис. 35:

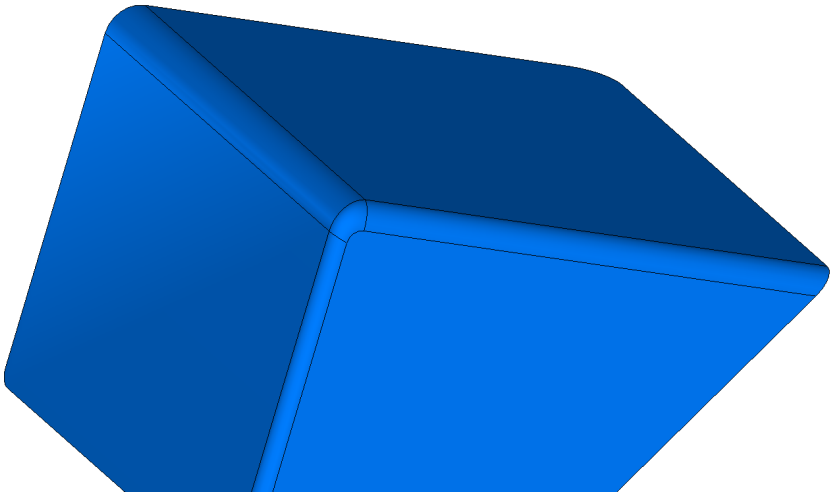


Рис. 35

При значении **cornerForm** = *ef_delta* вершины, в которых стыкуются три ребра различной выпуклости, скругляются треугольной поверхностью как показано на рис. 36:

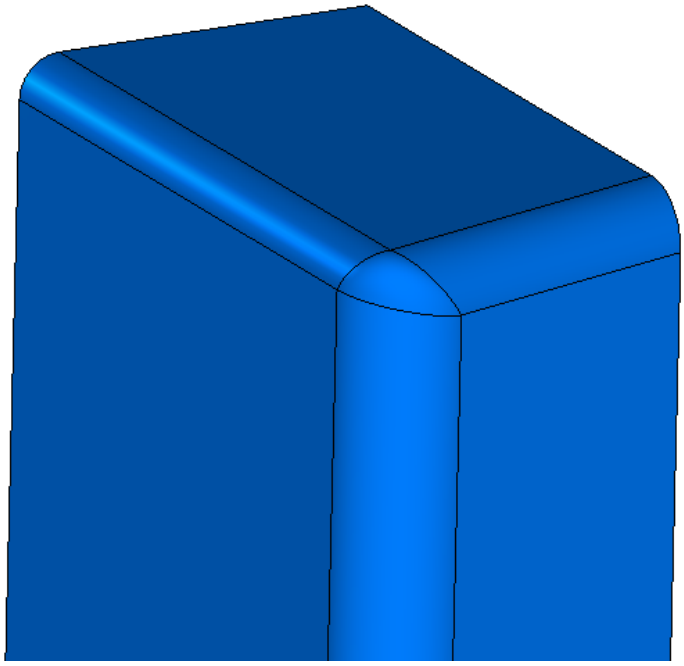


Рис. 36

При значении **cornerForm** = *ef_elbow2* вершины, в которых стыкуются три ребра различной выпуклости, скругляются четырёхугольной поверхностью с расположением четвёртой стороны напротив второго ребра вершины как показано на рис. 37:

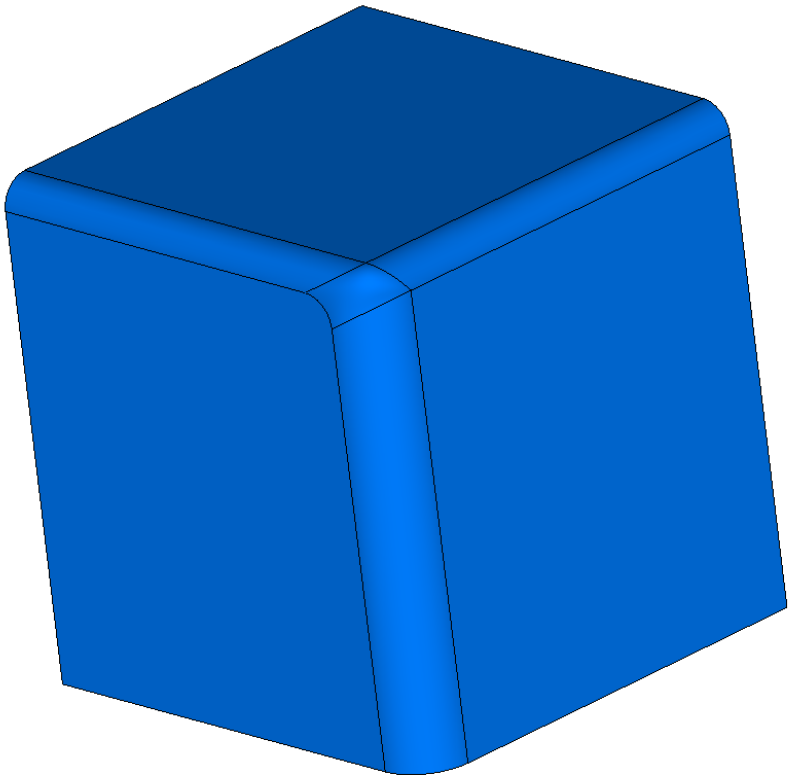


Рис. 37

В тестовом приложении *test.exe* скругление вершин и примыкающих к ним рёбер постоянным радиусом выполняется командами меню «Создать→Тело→Обработкой вершин→Скруглением вершин».

3.2.9 Построение тела с фасками рёбер

Функция

```
MbResultType ChamferSolid(       MbSolid &              solid,
                                 MbeCopyMode            sameShell,
                           const MbShellChamferParams & chamferParams,
                                 c3d::SolidSPtr &       result );
```

выполняет построение фасок указанных рёбер исходного тела.

Входными параметрами функции являются:

- **solid** — исходное тело;
- **sameShell** — режим копирования исходного тела;
- **chamferParams** — параметры построения.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

При построении тел с фасками рёбер для инициализации объектов **chamferParams** с параметрами операции в классе MbShellChamferParams определён следующий конструктор:

```
MbShellChamferParams( const c3d::EdgesSPtrVector &  initCurves,
                      const SmoothValues &          params,
                      const MbSNameMaker &          names )
: _initCurves( initCurves )
, _nameMaker( &names.Duplicate() )
, _params( params ){}
```

где

- **initCurves** — массив обрабатываемых рёбер тела;
- **params** — объект с параметрами построения фаски;
- **names** — именователь.

Параметр **solid** содержит исходное тело, рёбра которого подлежат обработке.

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** к построенному телу **result** и может принимать одно из четырёх значений:

- *cm_Same = 0* (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Для построения фасок рёбер используются те же параметры SmoothValues & **params**, что и при построении скруглений рёбер. Класс SmoothValues описан в файле *op_shell_parameter.h*. Объект **params** содержит информацию о форме и способе сопряжения смежных граней обрабатываемых рёбер. Для построения фасок используются следующие данные параметра **params***:

- **distance1** — первый катет фаски;
- **distance2** — второй катет фаски;
- **begLength** — расстояние от начальной вершины до точки остановки сопряжения (отрицательное значение означает отсутствие остановки, значение должно быть равно *UNDEFINED_DBL*, если не требуется задавать остановку);
- **endLength** — расстояние от конечной вершины до точки остановки сопряжения (отрицательное значение означает отсутствие остановки, значение должно быть равно *UNDEFINED_DBL*, если не требуется задавать остановку);
- **form** — тип сопряжения из перечисления MbeSmoothForm;
- **smoothCorner** — способ скругления чемоданных углов;
- **prolong** — флаг продолжения скругления по касательным рёбрам;
- **vector1** — вектор нормали плоскости остановки сопряжения в начале;
- **vector2** — вектор нормали плоскости остановки сопряжения в конце.

* Все открытые поля будут перенесены в закрытую секцию. Для задания параметров фаски следует использовать специализированные методы инициализации:

```
// инициализация фаски с равными отступами
bool InitChamfer( double dist );

// инициализация фаски с разными отступами
bool InitChamfer( double dist1, double dist2 );

// инициализация фаски по катету и углу
bool InitOneSideChamfer( bool firstSide, double dist, double ang );
```

Дополнительные свойства операции задаются методом SetProperties(), аргументами которого являются:

- **c** — коэффициент формы, изменяется от 0.05 до 0.95 (при 0 — дуга окружности);
- **pro** — флаг продолжения по касательной;
- **cor** — способ скругления «чемоданных» углов;
- **autoS** — автоопределение сохранения кромки/поверхности;
- **keep** — сохранять кромку (true) или сохранять поверхность скругления/фаски (false);
- **str** — строгое скругление. Если false, скруглить хотя бы то, что возможно;
- **equ** — в углах сочленения вставлять тороидальную поверхность.

Для задания остановок определены методы SetBegLength()/SetEndLength() (по умолчанию остановка не задана).

Способом сопряжения фаски управляет параметр **form**. Для построения фасок рёбер используются значения параметра **form** равные *st_Chamfer*, *st_Slant1* и *st_Slant2*.

При значении **form** = *st_Chamfer* функция строит поверхность фаски с заданными катетами, которые определяются параметрами **distance1** и **distance2**, рис. 1:

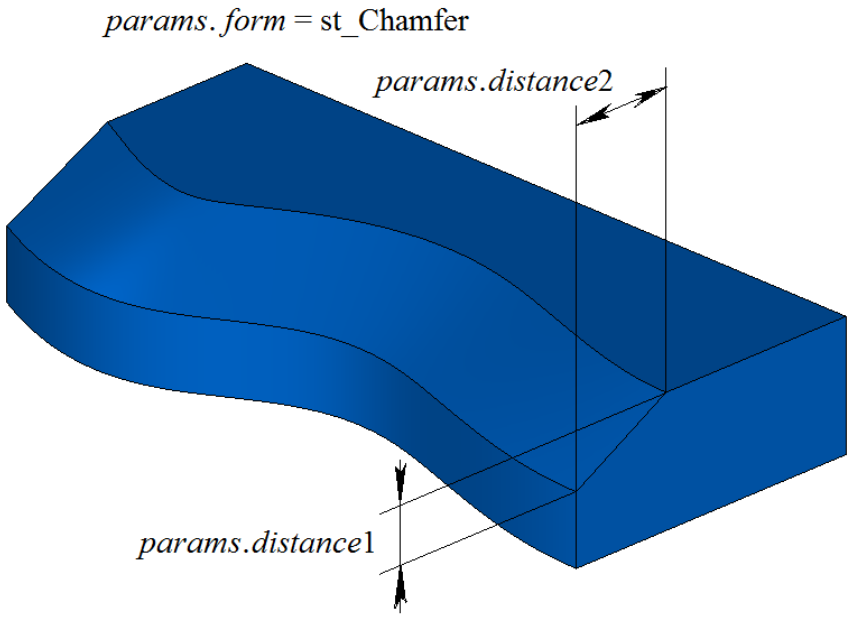


Рис. 1

При значении **form** = *st_Slant1* функция строит поверхность фаски с заданным катетом **distance1** и прилегающим к нему углом, определяемым катетом **distance2**, рис. 2:

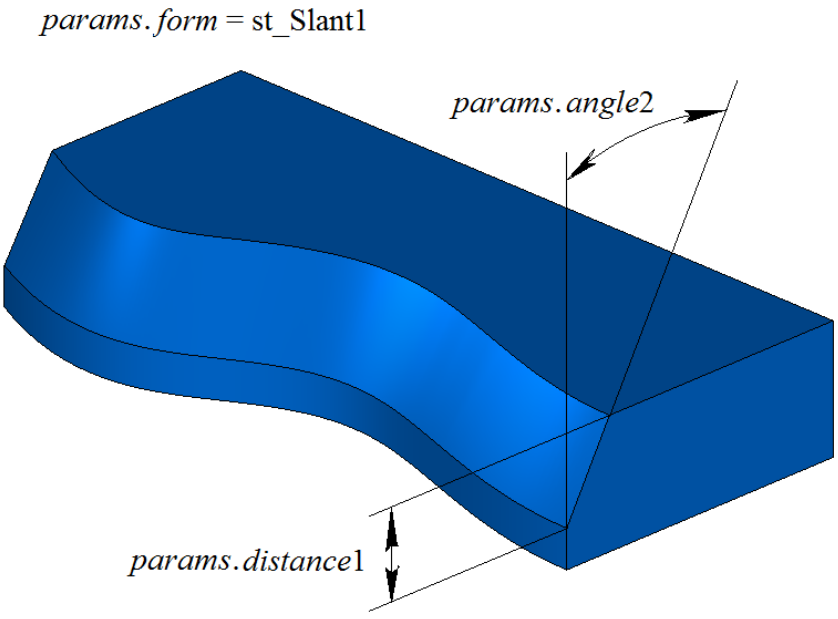


Рис. 2

При значении **form** = *st_Slant2* функция строит поверхность фаски с заданным углом и прилегающим к нему катетом. Параметр **distance1** соответствует катету, обеспечивающему заданный угол, а **distance2** определяет прилежащий катет, рис. 3:

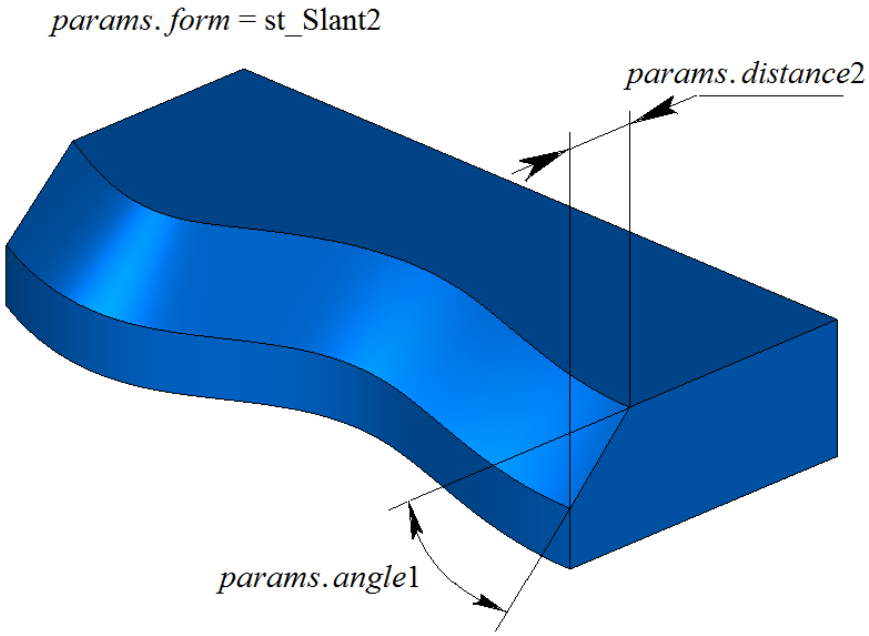


Рис. 3

На рис. 4 приведён пример остановки фаски на расстоянии **begLength** от начальной вершины и на расстоянии **endLength** от конечной вершины обрабатываемого ребра. Если не требуется останавливать сопряжение, то расстояния **begLength** и **endLength** должны принимать отрицательные значения.

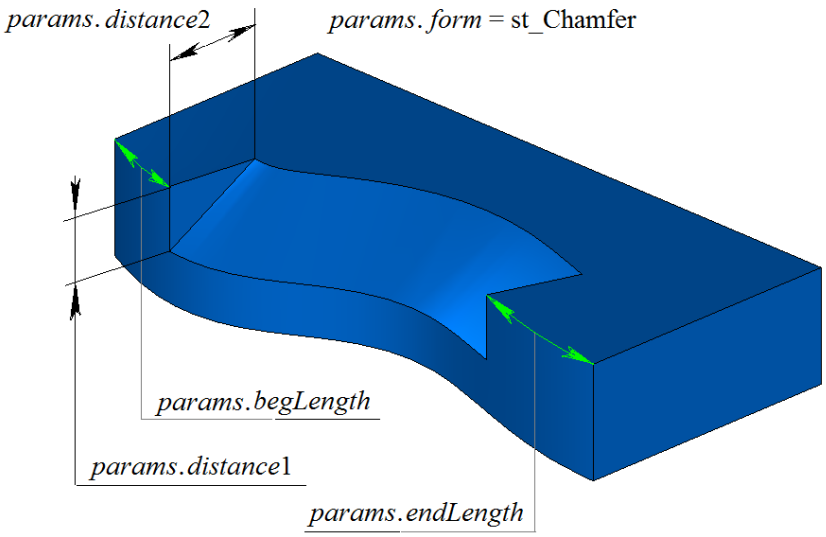


Рис. 4

На примере тела, приведённого на рис. 5, продемонстрируем работу флага **prolong** при построении фаски выделенного ребра:

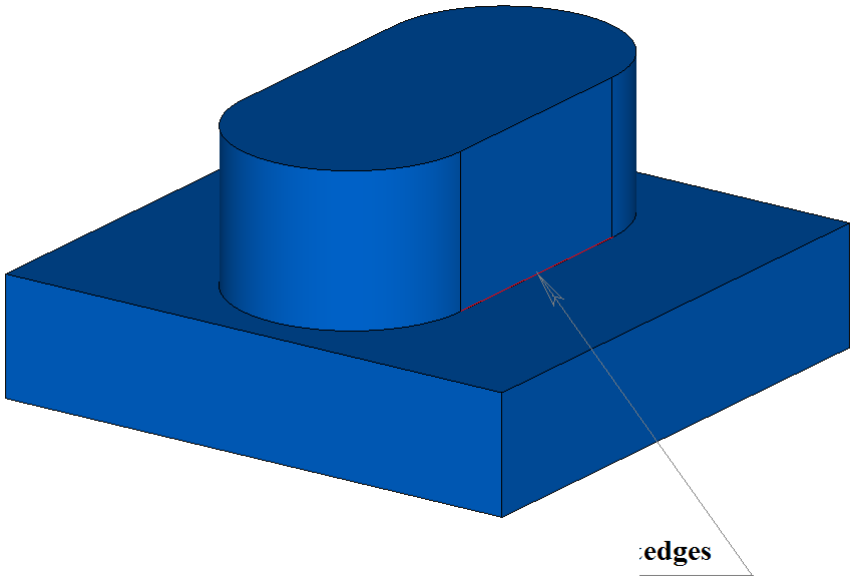


Рис. 5

Флаг **prolong** указывает на то, какие рёбра тела должны быть обработаны. Если **prolong** = *false*, то обработке подлежат только те рёбра, которые указаны в контейнере **initCurves**, рис. 6:

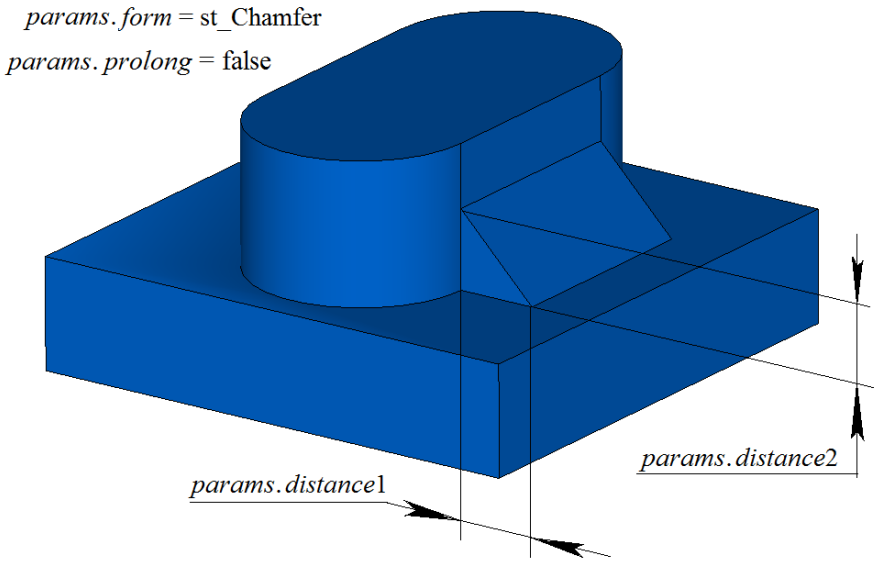


Рис. 6

Если **prolong** = *true*, то обработке подлежат рёбра, указанные в контейнере **initCurves**, а также рёбра, гладко стыкующиеся с ними, рис. 7:

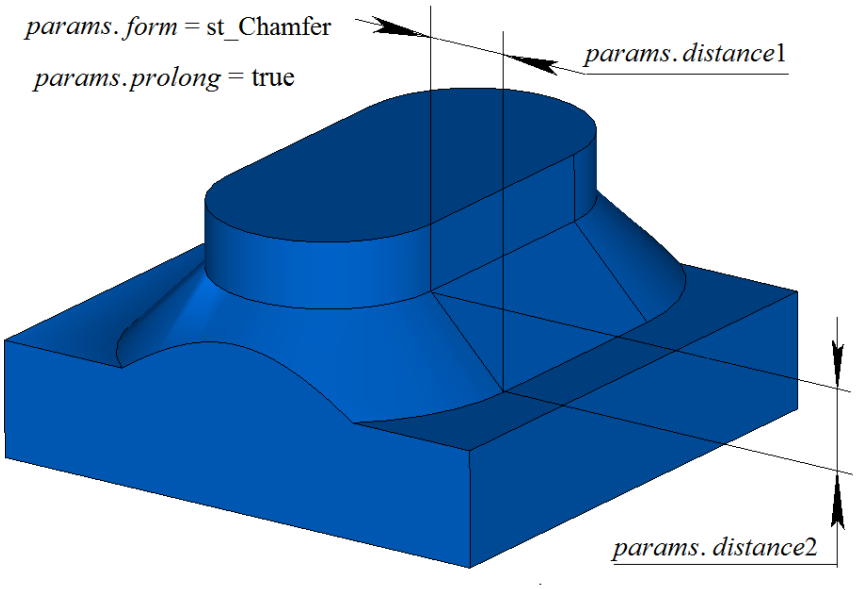


Рис. 7

При построении фаски трёх рёбер, стыкующихся в одной вершине, параметр **smoothCorner** определяет способ обработки чемоданных углов. На примере тела, приведённого на рис. 8, покажем работу параметра **smoothCorner** при построении фасок на всех рёбрах тела.

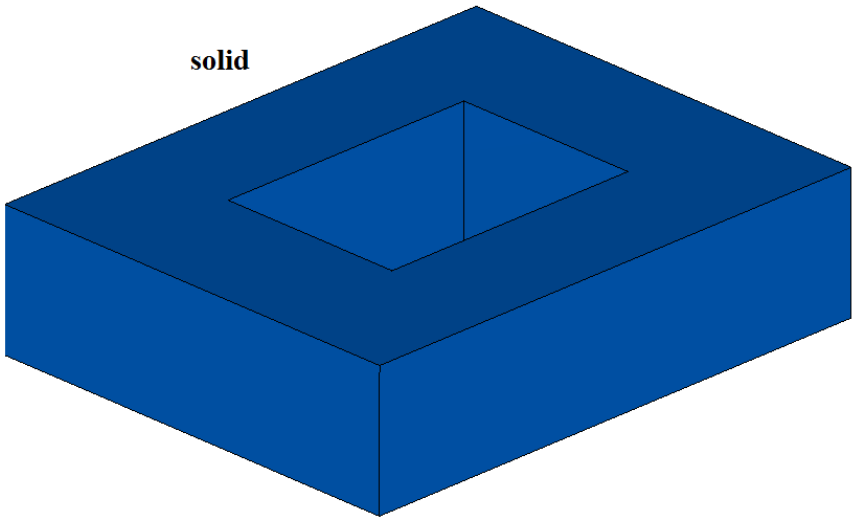


Рис. 8

Если **smoothCorner** = *ec_pointed*, то обработка углов, в которых стыкуются три ребра одинаковой выпуклости, отсутствует, и в построенном теле присутствует точка, в которой стыкуются три грани фаски, рис. 9:

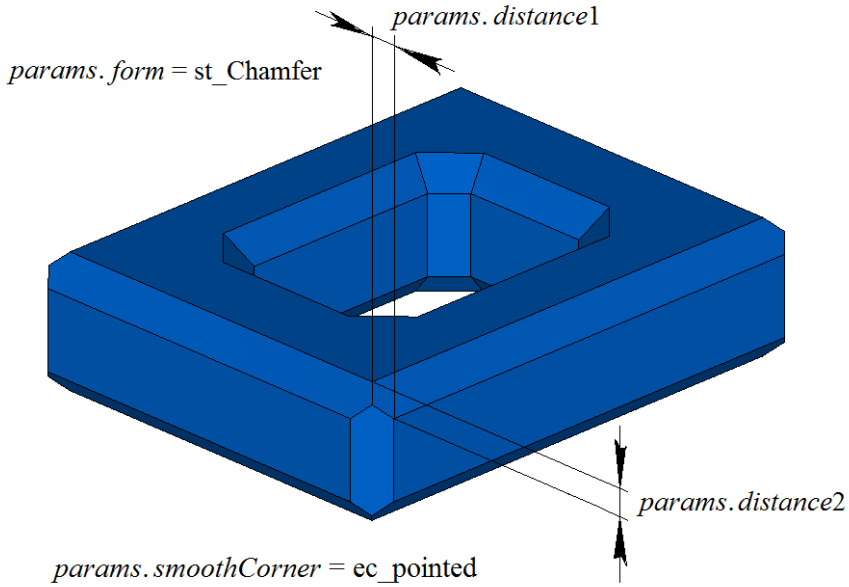


Рис. 9

При всех других значениях параметра **smoothCorner** углы, в которых стыкуются три ребра одинаковой выпуклости, обрабатываются путём построения дополнительной грани, как показано на рис. 10:

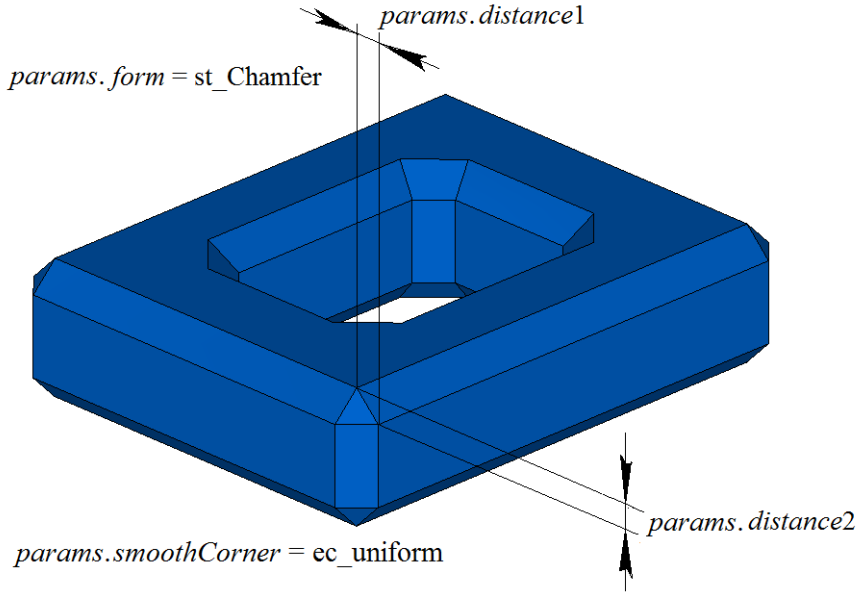


Рис. 10

На примере тела в форме пирамиды, приведённого на рис. 11, покажем работу рассматриваемой функции при построении фаски в частных случаях. Результат построения при симметричной конфигурации рёбер и симметричной фаске приведён на рис. 12:

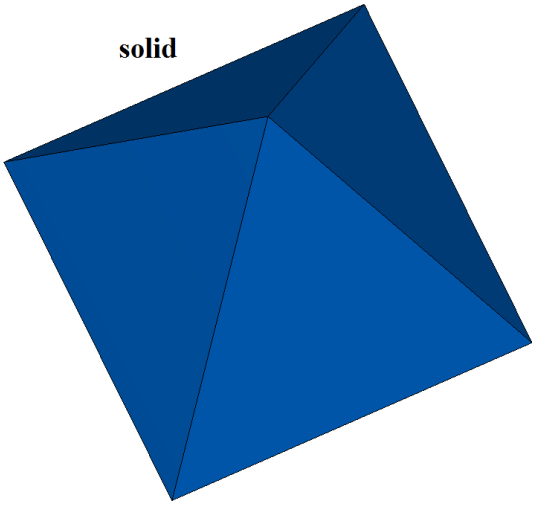


Рис. 11

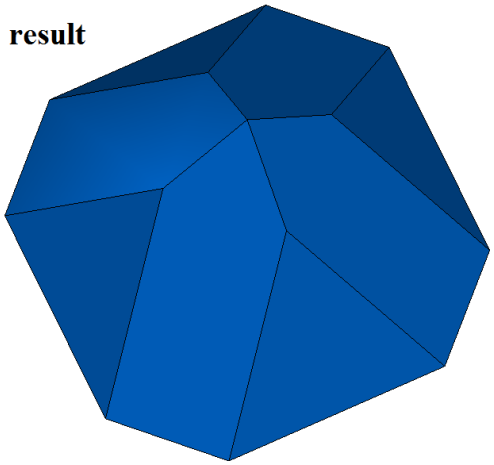


Рис. 12

Следует заметить, что для построения фаски стыкующихся в одной вершине четырёх и более рёбер необходимо пересечение всех поверхностей фаски в одной точке. Пример симметричной фаски семи рёбер, стыкующихся в общей вершине, приведён на рис. 13:

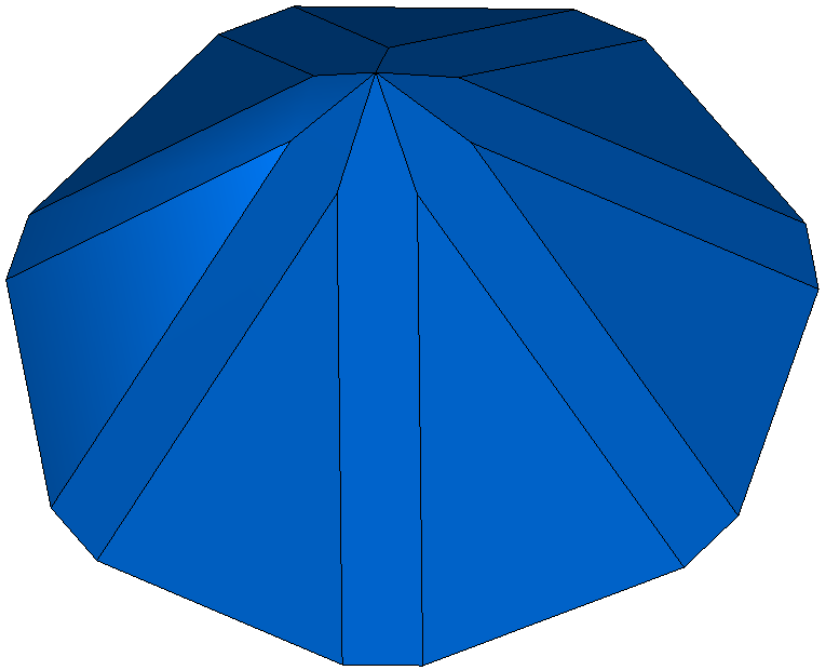


Рис. 13

Когда в функции построения фаски параметры **distance1** и **distance2** не равны и для обработки заданы несколько рёбер, то трудно разобраться, что чему соответствует. В такой ситуации предлагается использовать вспомогательные функции: SmoothPhantom(), SmoothSequence(), SmoothPositionData(), объявленные в файле *action_phantom.h*.

Функция SmoothPhantom() строит упрощённые поверхности для имитации будущих фасок.

Функция SmoothSequence() строит последовательности рёбер, к которым по желанию могут быть добавлены гладко стыкующиеся рёбра, обрабатываемые совместно.

Функция SmoothPositionData() вычисляет по три точки для обрабатываемых рёбер, которые используются для фантомных размеров катетов и углов. Обычно с помощью указанных функций рисуется некий фантом, по которому можно понять, что получится в результате операции, и даётся возможность поменять параметры при необходимости.

Перед построением фасок обрабатываемые рёбра сортируются, при необходимости или по требованию к ним добавляются гладко стыкующиеся рёбра, и из рёбер создаются последовательности гладко стыкующихся рёбер. Параметры **distance1** и **distance2** привязаны к первой и второй поверхностям соответственно, кривой пересечения первого ребра в последовательности гладко стыкующихся рёбер.

Поверхности можно получить методами кривой пересечения GetCurveOneSurface() и GetCurveTwoSurface().

По **distance1** и **distance2** первого ребра определяются параметры для остальных рёбер каждой последовательности.

При построении фаски рёбер рассматриваемая функция добавляет в журнал построенного тела строитель MbChamferSolid, который объявлен в файле *cr_chamfer_solid.h*.

В тестовом приложении *test.exe* построение тела с фасками выполняется командами меню «Создать → Тело → Обработкой рёбер → Фаска катет-катет», «Создать → Тело → Обработкой рёбер → Фаска катет-угол», «Создать → Тело → Обработкой рёбер → Фаска угол-катет».

3.2.10 Построение тонкостенного тела

Функция

```
MbResultType ThinSolid(       MbSolid &           solid,
                              MbeCopyMode         sameShell,
                              RPArray<MbFace> &   outFaces,
                              SweptValues &       params,
                        const MbSNameMaker &      names,
                              bool                copyFaceAttrs,
                              MbSolid *&          result );
```

выполняет построение тонкостенного тела исключением указанных граней исходного тела с опциональным приданием одинаковой толщины оставшимся граням.

Входными параметрами функции являются:

- **solid** — исходное тело;
- **sameShell** — режим копирования исходного тела;
- **outFaces** — вскрываемые грани тела;
- **params** — параметры операции;
- **names** — именователь;
- **copyFaceAttrs** — флаг копирования атрибутов из исходных граней в эквидистантные.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

Функция удаляет из исходного тела **solid** указанные грани **outFaces**, а оставшимся граням «придаёт заданную толщину».

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырёх значений:

- *cm_Same = 0* (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode детально описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **params** содержит информацию о толщине стенки сохраняемых граней и замкнутости построенного тела **result** (см. *op_swept_parameter.h*).

Толщина стенки сохраняемых граней может быть равна **params**.*thickness1* в положительном направлении нормали грани или **params**.*thickness2* в отрицательном направлении нормали грани.

Если **params**.*shellClosed* = *false*, то будет построено незамкнутое тело.

Параметр **params**.*checkSelfInt* сообщает о необходимости проверки результата построения на самопересечение. По умолчанию **params**.*checkSelfInt* = *false* (проверка не выполняется).

Параметр **params**.*mergeFlags* управляет слиянием подобных граней или рёбер (mergeFaces = *true* — сливать подобные грани; mergeEdges = *true* — сливать подобные рёбра).

Параметр **names** обеспечивает именование граней, рёбер и вершин построенного тела.

Для выполнения операции удаляемые грани **outFaces** по общему периметру не должны иметь гладких рёбер стыковки с сохраняемыми гранями исходного тела.

Параметр **copyFaceAttrs** управляет копированием атрибутов из исходных граней в эквидистантные.

На рис. 1 показаны исходное тело **solid** и удаляемые грани **outFaces**:

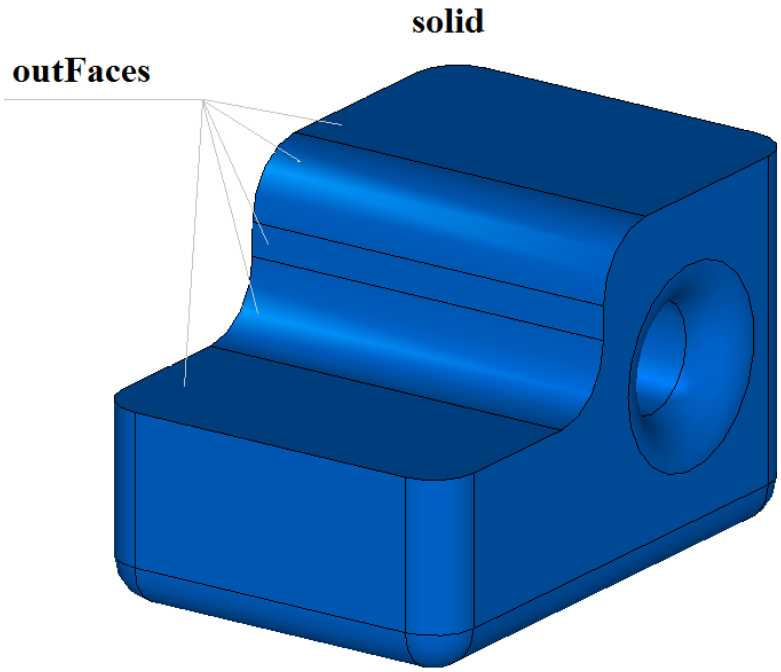


Рис. 1

На рис. 2 приведено построенное тонкостенное тело **result** с утолщением сохранённых граней внутрь исходного тела:

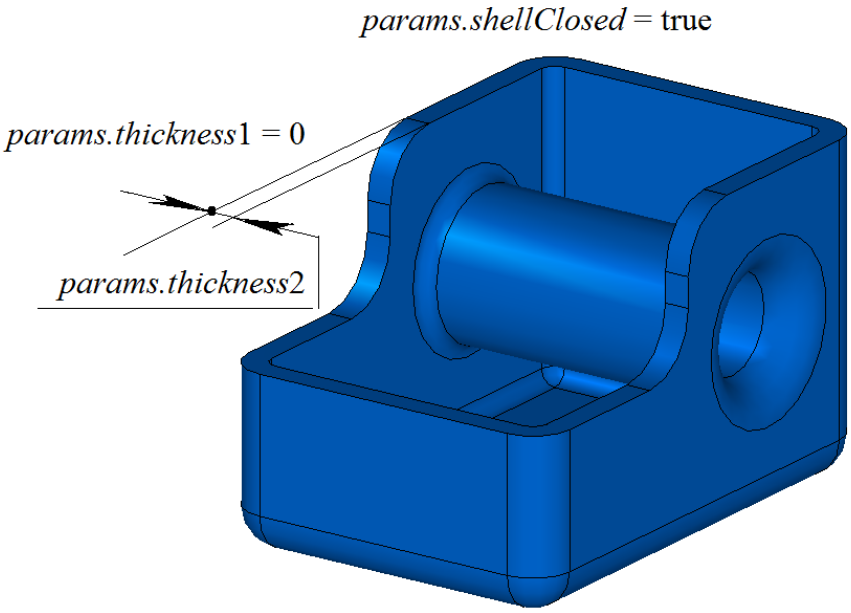


Рис. 2

На рис. 3 приведено построенное тонкостенное тело **result** с утолщением сохранённых граней наружу исходного тела:

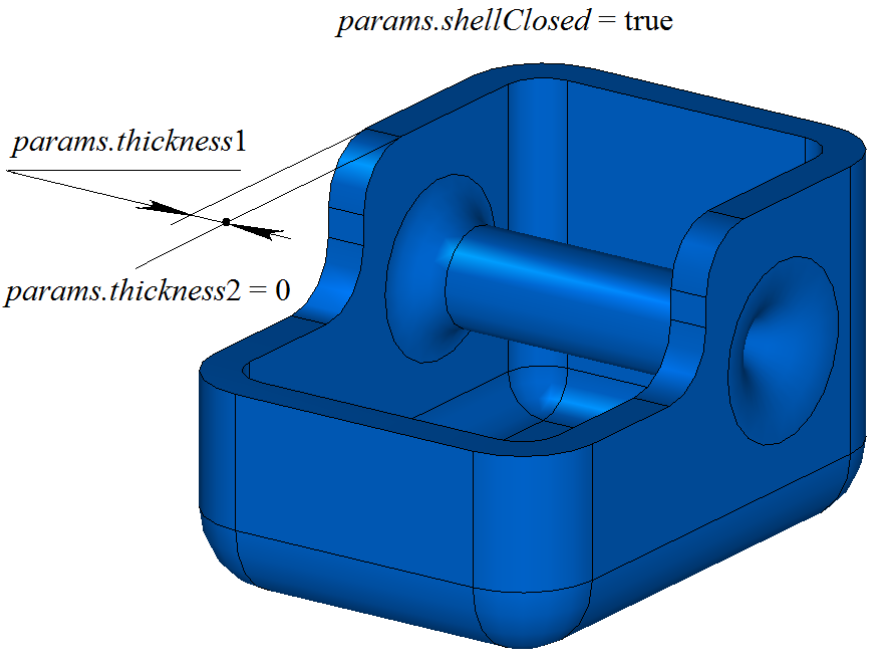


Рис. 3

На рис. 4 приведено построенное незамкнутое тело **result**:

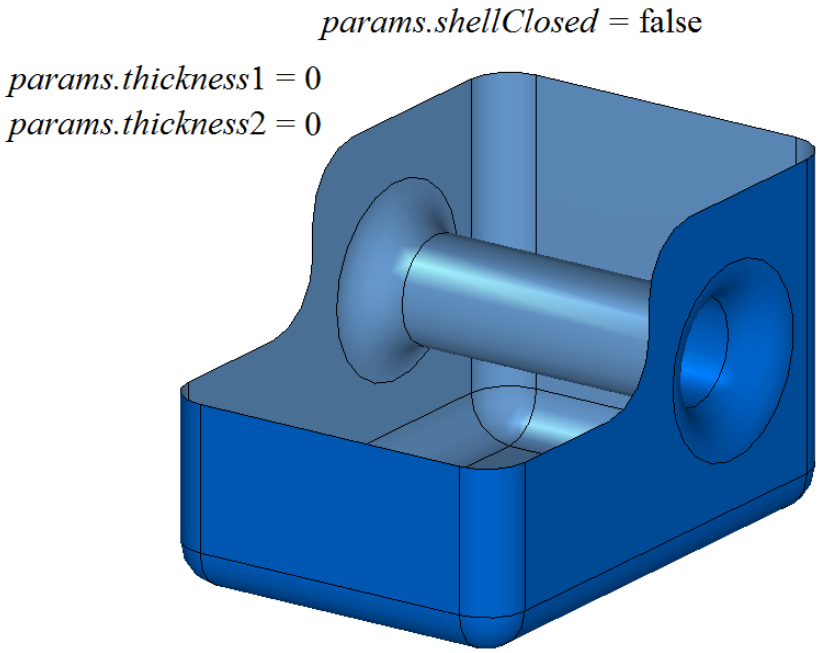


Рис. 4

На рис. 5 приведено тонкостенное тело, построенное при пустом множестве удаляемых граней:

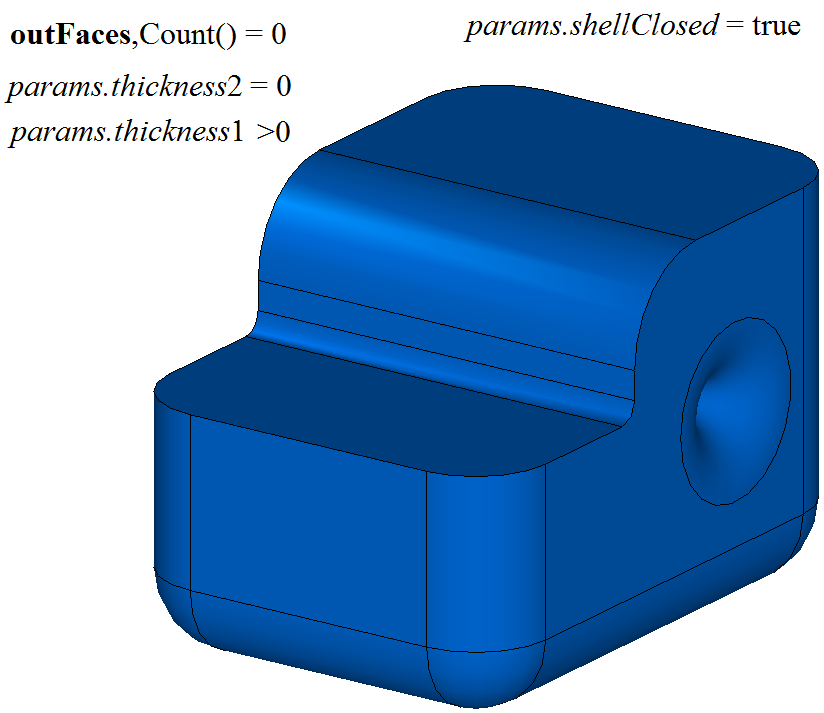


Рис. 5

Функция ThinSolid() добавляет в журнал построенного тела строитель MbShellSolid, который содержит все необходимые данные для выполнения операции. Строитель MbShellSolid объявлен в файле *cr_thin_shell_solid.h*.

В тестовом приложении *test.exe* построение тонкостенного тела выполняется командой меню «Создать → Тело → Обработкой граней → Тонкостенное одной толщины».

3.2.11 Построение тонкостенного тела с различной толщиной стенки

Функция

```
MbResultType ThinSolid(       MbSolid &             solid,
                              MbeCopyMode           sameShell,
                              RPArray<MbFace> &     outFaces,
                              RPArray<MbFace> &     offFaces,
                              SArray<double> &      offDists,
                              SweptValues &         params,
                        const MbSNameMaker &        names,
                              bool                  copyFaceAttrs,
                              MbSolid *&            result );
```

выполняет построение тонкостенного тела исключением указанных граней и приданием различной толщины оставшимся граням исходного тела.

Входными параметрами функции являются:

- **solid** — исходное тело;
- **sameShell** — режим копирования исходного тела;

- **outFaces** — исключаемые грани тела;
- **offFaces** — множество граней, для которых заданы индивидуальные значения толщин;
- **offDists** — множество индивидуальных значений толщин (должен быть синхронизирован с массивом offFaces);
- **params** — параметры операции;
- **names** — именователь;
- **copyFaceAttrs** — флаг копирования атрибутов из исходных граней в эквидистантные.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

Функция удаляет из исходного тела **solid** указанные грани **outFaces**, а оставшимся граням «придаёт заданную толщину», причём толщина для разных граней может отличаться.

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырёх значений:

- cm_Same = 0 (оболочка не копируется);
- cm_KeepHistory (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- cm_KeepSurface (исходная оболочка и её копия имеют общие базовые поверхности);
- cm_Copy (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **offFaces** содержит грани, которым заданы индивидуальные значения толщин **offDists**.

Грани **offFaces**[*i*] будет «придана толщина» **offDists**[*i*]. Остальным граням будет задана толщина, определяемая параметром **params**. **params** содержит информацию о замкнутости построенного тела **result** и толщине стенки сохраняемых граней, не входящих в множество **offFaces**.

Толщина стенки сохраняемых граней может быть равна или **params**.*thickness1* в положительном направлении нормали грани или **params**.*thickness2* в отрицательном направлении нормали грани.

Параметр **names** обеспечивает именование граней, рёбер и вершин построенного тела.

Для выполнения операции удаляемые грани **outFaces** по общему периметру не должны иметь гладких рёбер стыковки с сохраняемыми гранями исходного тела.

На рис. 1 показаны исходное тело **solid**, удаляемые грани **outFaces** и грани **offFaces**, которым заданы индивидуальные значения толщин **offDists**:

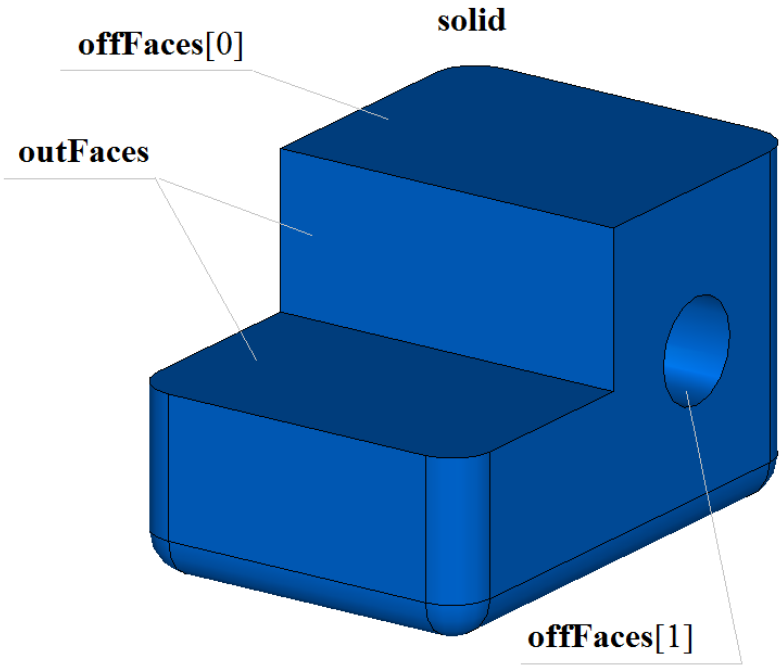


Рис. 1

На рис. 2 приведено построенное тело **result** с сохранёнными и вновь построенными гранями:

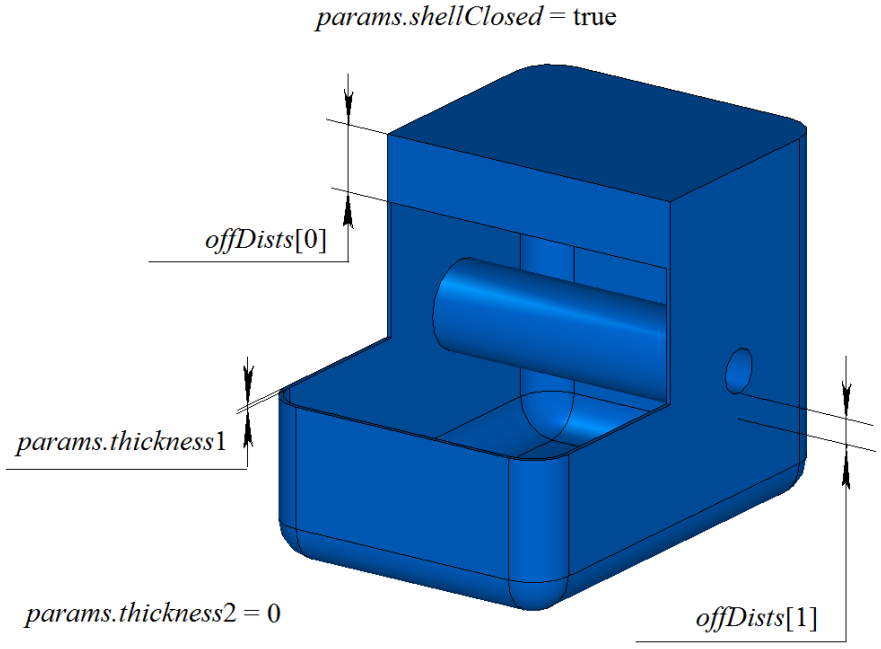


Рис. 2

Для выполнения операции каждая из граней **offFaces** по общему периметру не должна иметь гладких рёбер стыковки с гранями исходного тела другой толщины.

Функция ThinSolid() добавляет в журнал построенного тела строитель MbShellSolid, который содержит все необходимые данные для выполнения операции. Строитель MbShellSolid объявлен в файле *cr_thin_shell_solid.h*.

В тестовом приложении *test.exe* построение тонкостенного тела с различной толщиной стенки выполняется командой меню «Создать → Тело → Обработкой граней → Тонкостенное разных толщин».

3.2.12 Построение тела приданием толщины поверхности

Функция

```
MbResultType ThinSolid( const MbSurface &       surface,
                              bool              faceSense,
                              SweptValues &     params,
                        const MbSNameMaker &    names,
                              SimpleName        name,
                              MbSolid *&        result )
```

выполняет построение тела путём придания толщины заданной поверхности.

Входными параметрами функции являются:

- **surface** — заданная поверхность;
- **faceSense** — ориентация нормали поверхности в грани построенного тела (*true* означает совпадение нормали поверхности и грани, *false* — противоположное направление);
- **params** — параметры построения;
- **names** — именователь операции;
- **name** — имя операции.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

Функция строит грань на основе поверхности **surface** и далее строит тело путём придания этой грани заданной толщины.

Параметр **faceSense** указывает, совпадает ли направление нормали поверхности **surface** с направлением нормали грани. Грани будет придана толщина, определяемая параметром **params**.

Параметр **params** содержит информацию о толщине стенки построенного тела **result**. Толщина стенки может быть равна или **params**.*thickness1* в положительном направлении нормали грани или **params**.*thickness2* в отрицательном направлении нормали грани.

Параметры **names** и **name** обеспечивают именование граней, рёбер и вершин построенного тела.

На рис. 1 показана исходная поверхность **surface**:

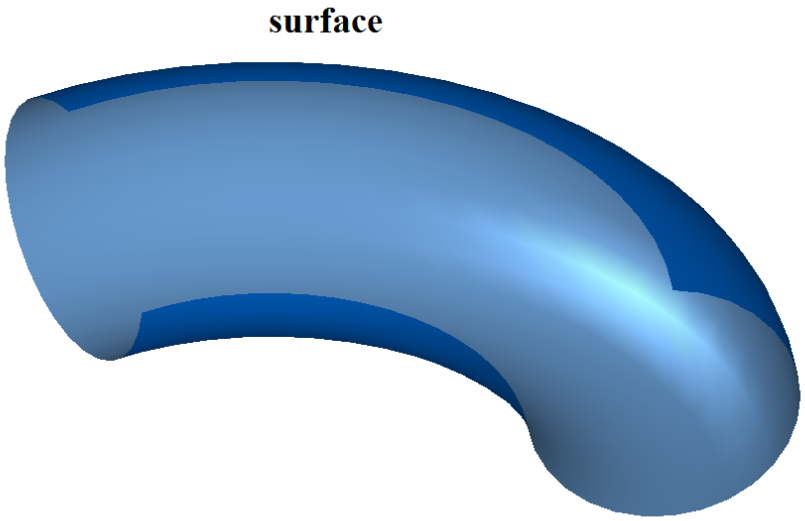


Рис. 1

На рис. 2 приведено построенное тело **result**:

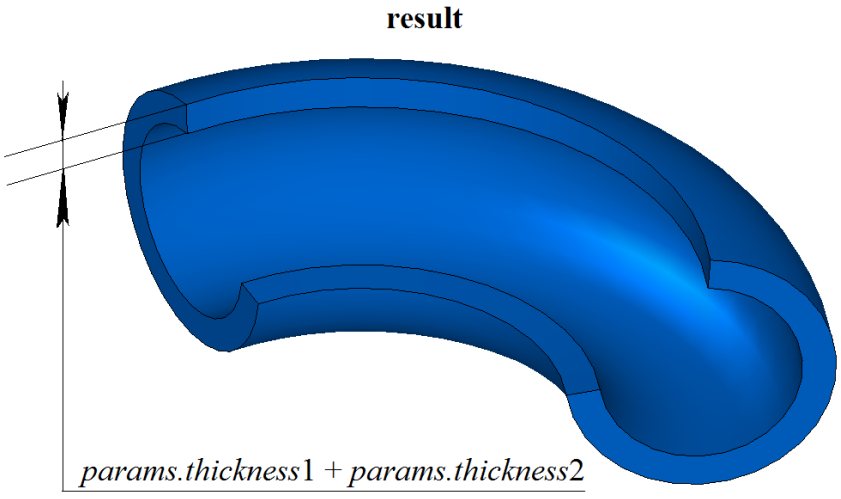


Рис. 2

Функция ThinSolid() добавляет в журнал построенного тела строитель MbShellSolid, который содержит все необходимые данные для выполнения операции. Строитель MbShellSolid объявлен в файле *cr_thin_shell_solid.h*.

В тестовом приложении *test.exe* построение тонкостенного тела выполняется командой меню «Создать → Тело → На базе поверхности → Приданием толщины».

3.2.13 Булева операция тела и множества тел

Функция

```
MbResultType UnionResult(       MbSolid *               solid,
                                MbeCopyMode             sameShell,
                                RPArray<MbSolid> &      solids,
                                MbeCopyMode             sameShells,
                                OperationType           oType,
                                bool                    checkIntersect,
                          const MbMergingFlags          mergeFlags,
                          const MbSNameMaker &          names,
                                bool                    isArray,
                                MbSolid *&              result,
                                RPArray<MbSolid> *      notGluedSolids = NULL )
```

выполняет объединение тел заданного множества и заданную булеву операцию с исходным телом, если оно задано.

Входными параметрами функции являются:

- **solid** – исходное тело (может быть ноль);
- **sameShell** – вариант копирования исходного тела;
- **solids** – множество тел;
- **sameShells** – режим копирования тел множества;
- **oType** – тип булевой операции между телом и массивом тел:
  - *bo_Union* – объединение тел,
  - *bo_Intersect* – пересечение тел,
  - *bo_Difference* – вычитание тел;
- **checkIntersect** – флаг проверки пересечения тел множества (*false* — не проверять);
- **mergeFlags** – управляющие флаги слияния элементов оболочки;
- **names** – именователь;
- **isArray** – флаг регулярности множества тел.

Выходным параметром функции является построенное тело **result** и множество тел **notGluedSolids**, которые оказались не используемыми в операции (может быть ноль).

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

Функция представляет собой разновидность булевой операции BooleanSolid() и ускоряет работу, когда требуется выполнить одну и ту же булеву операцию тела **solid** с большим количеством других тел. Сначала функция выполняет объединение множества тел **solids** в общее промежуточное тело, а затем выполняет указанную булеву операцию *oType* тела **solid** с промежуточным телом. Тела **solids** могут не пересекаться друг с другом.

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырёх значений:

- cm_Same = 0 (оболочка не копируется);
- cm_KeepHistory (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);

- cm_KeepSurface (исходная оболочка и её копия имеют общие базовые поверхности);
- cm_Copy (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **oType** определяет тип булевой операции и принимает три значения: *bo_Union*, *bo_Intersect*, *bo_Difference*:

- при **oType** = *bo_Union* выполняется объединение тел **solid** и множества тел **solids**;
- при **oType** = *bo_Intersect* выполняется пересечение тел **solid** и множества тел **solids**;
- при **oType** = *bo_Difference* выполняется вычитание из тела **solid** множества тел **solids**.

Параметры **checkIntersect** и **isArray** управляют построением общего промежуточного тела для множества тел **solids**.

Параметр **checkIntersect** даёт команду на проверку пересечения тел множества **solids** между собой. Если параметр **checkIntersect** == *true*, то при построении общего промежуточного тела выполняется булева операция объединения всех пересекающихся тел множества **solids** в одно тело. В противном случае объединение тел заданного множества выполняется простым перекладыванием граней всех тел в одно новое тело. Вне зависимости от значения параметра **checkIntersect** все не пересекающиеся тела множества **solids** передают свои грани общему промежуточному телу.

Параметр **isArray** работает только при **checkIntersect** == *true* и сообщает о регулярности множества тел **solids**. Если **isArray** == *true*, то тела множества расположены в узлах прямоугольной или круговой сетки и позиции тел заданы в именах граней.

Параметр **mergeFlags** позволяет объединять подобные грани и рёбра в построенном теле **result** или оставлять их разделёнными. При **mergeFlags**.*mergeFaces* == *false* подобные грани не объединяются, при **mergeFlags**.*mergeEdges* == *false* подобные рёбра не объединяются.

Параметр **names** обеспечивает именование граней, рёбер и вершин построенного тела.

Параметр **notGluedSolids** содержит тела, которые оказались не используемыми в операции из-за невозможности объединения их с общим промежуточным телом.

На рис. 1 приведены исходное тело **solid** и множество тел **solids**:

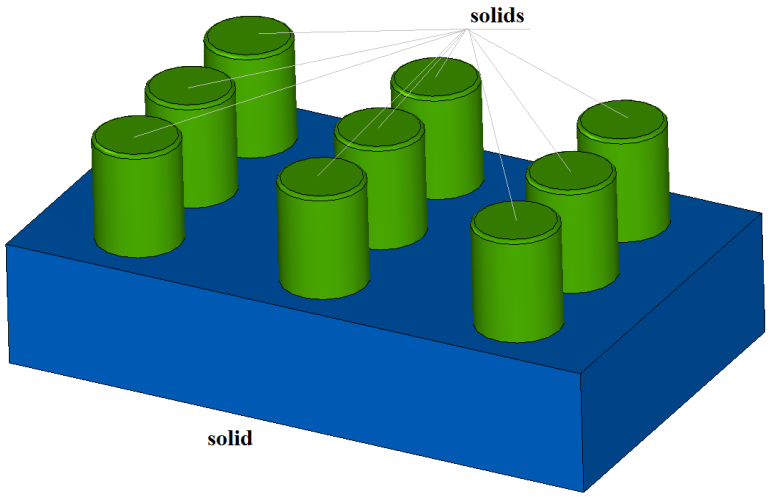


Рис. 1

На рис. 2 приведён результат **result** приклеивания тел **solids** к телу **solid**. В данном случае параметр **checkIntersect** можно положить равным *false*, так как тела **solids** не пересекаются друг с другом:

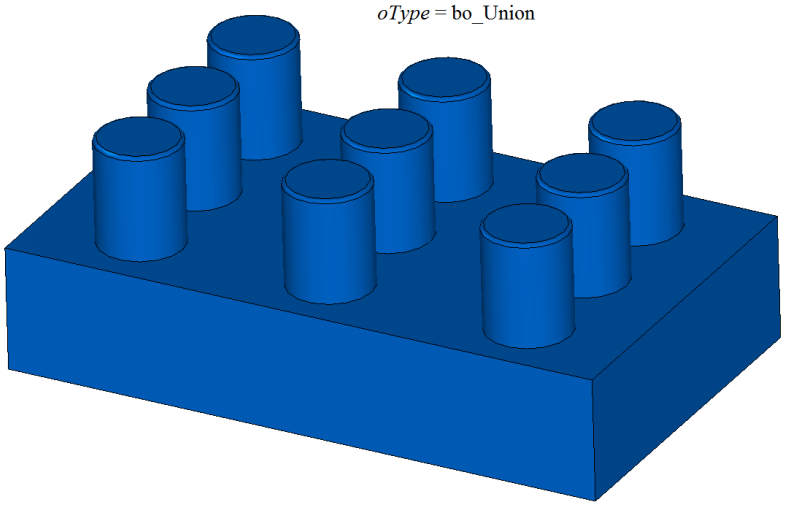


Рис. 2

На рис. 3 приведены исходное тело **solid** и множество тел **solids**:

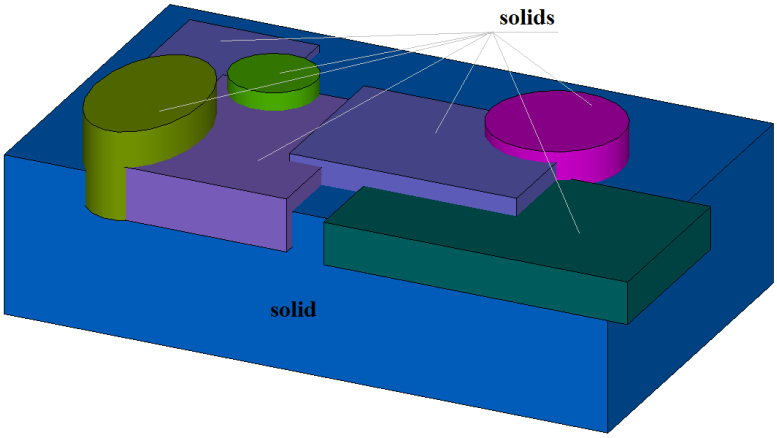


Рис. 3

На рис. 4 приведён результат **result** вычитания тел **solids** из тела **solid** при работе рассматриваемой функции с параметром **mergeFlags**.*mergeFaces* == *true*. В данном случае параметр **checkIntersect** необходимо положить равным *true*, так как тела **solids** пересекаются друг с другом:

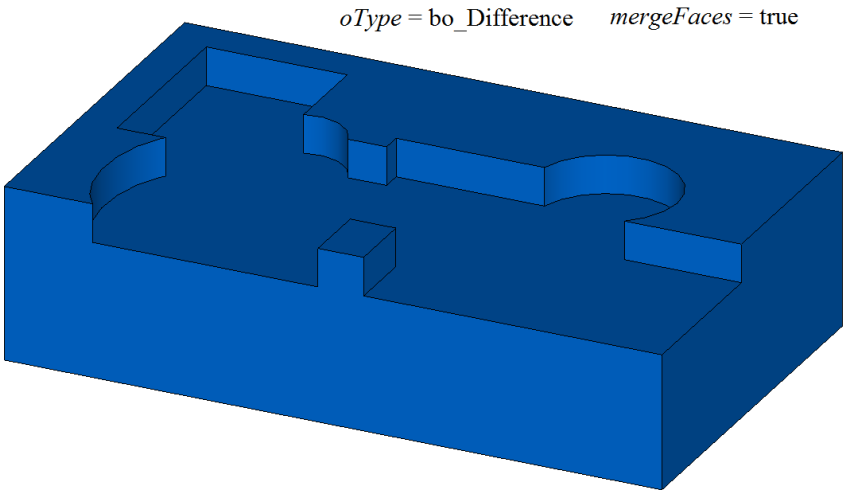


Рис. 4

На рис. 5 приведён результат **result** вычитания тел **solids** из тела **solid** с параметром **mergeFlags**.*mergeFaces* == *false*:

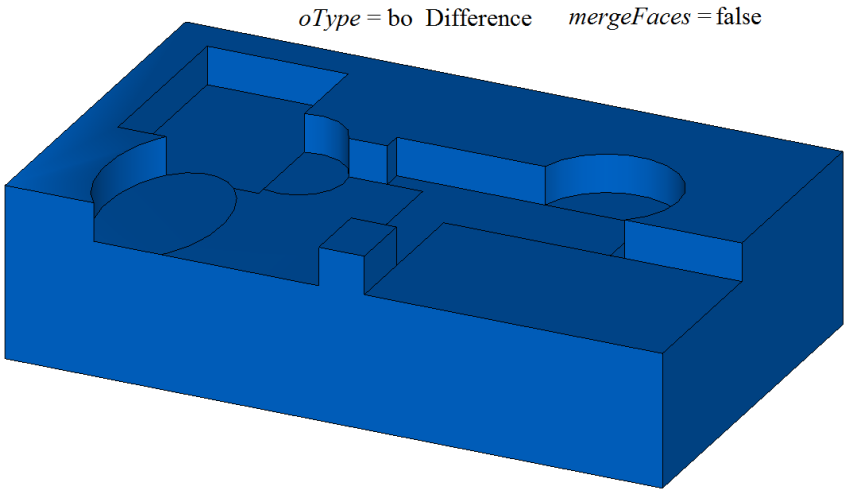


Рис. 5

На рис. 6 для наглядности грани результата вычитания тел **solids** из тела **solid**, приведённого на рис. 3, раскрашены в цвета исходных тел. По форме граней можно определить последовательность включения тел **solids** в промежуточное тело: тело оставляет более полный отпечаток, если оно включено в промежуточное тело раньше остальных:

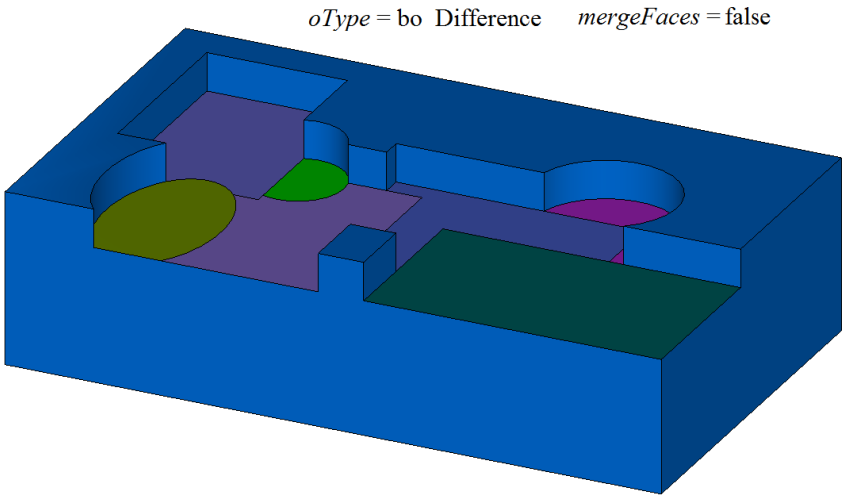


Рис. 6

Функция UnionResult() добавляет в журнал построенного тела строитель MbUnionSolid, который содержит все необходимые данные для выполнения операции. Строитель MbUnionSolid объявлен в файле *cr_union_solid.h*.

В тестовом приложении *test.exe* булева операция тела с множеством тел выполняется командами меню:

- «Создать → Тело → Приклеиванием к телу → Набора тел»;
- «Создать → Тело → Вырезанием из тела → Набора тел»;

- «Создать → Тело → Пересечением с телом → Набора тел».

3.2.14 Объединение множества тел

Функция

```
MbResultType UnionSolid(       RPArray<MbSolid> &  solids,
                               MbeCopyMode         sameShells,
                               bool                checkIntersect,
                         const MbSNameMaker &      names,
                               bool                isArray,
                               MbSolid *&          result,
                               RPArray<MbSolid> *  notGluedSolids = NULL );
```

выполняет объединение тел заданного множества в одно тело.

Входными параметрами функции являются:

- **solids** — множество тел;
- **sameShells** — вариант копирования тел множества;
- **checkIntersect** — флаг проверки пересечения тел множества (false — не проверять);
- **names** — именователь операции;
- **isArray** — флаг регулярности множества тел.

Выходными параметрами функции являются построенное тело **result** и множество тел **notGluedSolids**, которые оказались не используемыми в операции (указатель может быть равен нулю).

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Рассматриваемая функция работает так же, как функция UnionResult() при значениях параметров последнего **solid** = 0, **sameShell** = *cm_Same*, **oType** = *bo_Union*, **mergeFlags**.*mergeFaces* = *true*. Функция выполняет объединение множества тел **solids** в одно тело **result**, при этом тела могут не пересекаться друг с другом.

Параметр **sameShells** управляет передачей граней, ребер и вершин от тел **solids** построенному телу **result** и может принимать одно из четырёх значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode детально описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **checkIntersect** даёт команду на проверку пересечения тел множества **solids** между собой. Если параметр **checkIntersect** == *true*, то при построении результата выполняется булева операция объединения всех пересекающихся тел множества **solids**. В противном случае объединение тел заданного множества выполняется простым перекладыванием граней всех тел в построенное тело. Вне зависимости от значения параметра **checkIntersect** все не пересекающиеся тела множества **solids** передают свои грани общему промежуточному телу.

Параметр **isArray** работает только при **checkIntersect** == *true* и сообщает о регулярности множества тел **solids**. Если **isArray** == *true*, то тела множества расположены в узлах прямоугольной или круговой сетки и позиции тел заданы в именах граней.

Параметр **names** обеспечивает именование граней, ребер и вершин построенного тела.

Параметр **notGluedSolids** содержит тела, которые оказались не используемыми в операции из-за невозможности их объединения.

На рис. 1 приведены исходные тела **solids**:

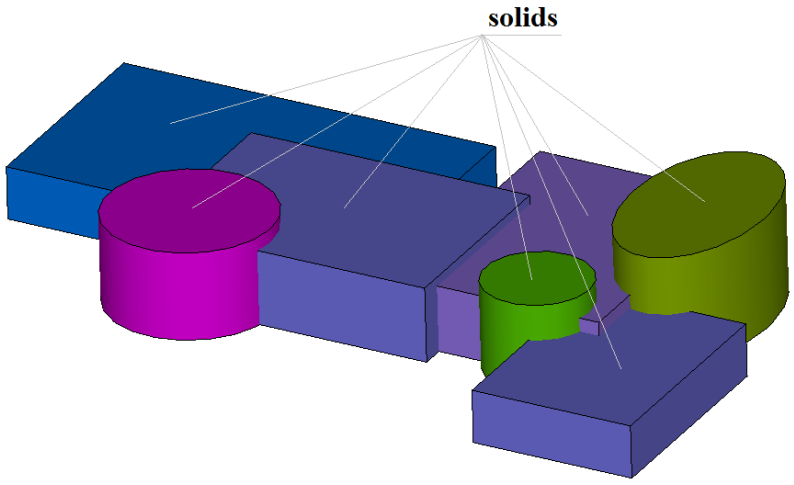


Рис. 1

На рис. 2 приведён результат **result** объединения тел **solids**. В данном случае параметр **checkIntersect** необходимо положить равным *true*, так как тела **solids** пересекаются друг с другом.

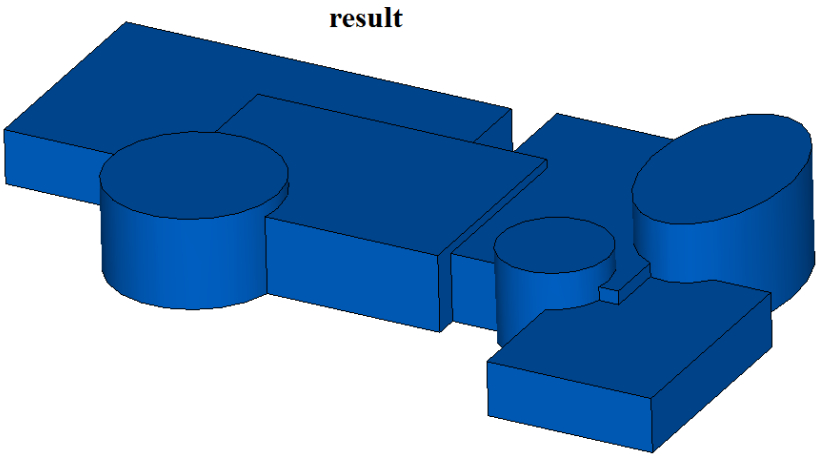


Рис. 2

Функции

```
MbResultType UnionSolid( const RPArray<MbSolid> & solids,
                         const MbSNameMaker &     names,
                               MbSolid *&         result ),
```

```
MbResultType UnionSolid( const c3d::SolidsVector & solids,
                         const MbSNameMaker &      names,
                               MbSolid *&          result ),
```

```
MbResultType UnionSolid( const c3d::SolidsSPtrVector & solids,
                         const MbSNameMaker &          names,
                               MbSolid *&              result )
```

являются упрощёнными вариантами рассматриваемой ранее одноимённой функции, объединяя тела заданного множества простым перекладыванием граней всех тел в одно новое тело. Таким образом, построенное тело и исходные тела имеют одни и те же грани.

Функции UnionSolid() добавляют в журнал построенного тела строитель MbUnionSolid, который содержит все необходимые данные для выполнения операции. Строитель MbUnionSolid объявлен в файле *cr_union_solid.h*.

В тестовом приложении *test.exe* булева операция тела с множеством тел выполняется командой меню «Создать → Тело → На базе тела → Набор тел».

3.2.15 Разбиение граней тела

Функция

```
MbResultType SplitSolid(       MbSolid &                solid,
                               MbeCopyMode              sameShell,
                         const RPArray<MbSpaceItem> &   spItems,
                               bool                     spSame,
                               RPArray<MbFace> &        selFaces,
                         const MbMergingFlags &         flags,
                         const MbSNameMaker &           names,
                               MbSolid *&               result );
```

выполняет разбиение указанных граней тела пространственными кривыми, поверхностями и оболочками. Пространственные элементы разбиения не должны иметь полных или частичных наложений и самопересечений.

Входными параметрами функции являются:

- **solid** — исходное тело;
- **sameShell** — режим копирования исходного тела;
- **spItems** — пространственные элементы разбиения граней;
- **spSame** — флаг использования пространственных элементов разбиения;
- **selFaces** — множество разбиваемых граней;
- **flags** — флаги слияния элементов оболочки;
- **names** — именователь.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Функция разбивает указанные грани **selFaces** исходного тела **solid** пространственными объектами **spItems**, если указанные грани пересекаются с объектами **spItems**. Объектами **spItems** могут служить кривые, поверхности или тело.

Для выполнения операции режущие объекты должны полностью пересекать указанные грани исходного тела.

Параметр **sameShell** управляет передачей граней, ребер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырёх значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **spSame** управляет копированием пространственных элементов разбиения (*true* — использовать оригиналы, *false* — использовать копии).

Параметр **flags** управляет слиянием элементов оболочки после разбиения:

- **flags**.*mergeFaces* = *true* — объединить геометрически подобные грани, возникшие как результат разбиения;
- **flags**.*mergeEdges* = *true* — объединить геометрически подобные рёбра.

Параметр **names** обеспечивает именование граней, ребер и вершин построенного тела.

На рис. 1 показаны исходное тело **solid**, разрезаемые грани **selFaces** и режущая поверхность **spItems**[0].

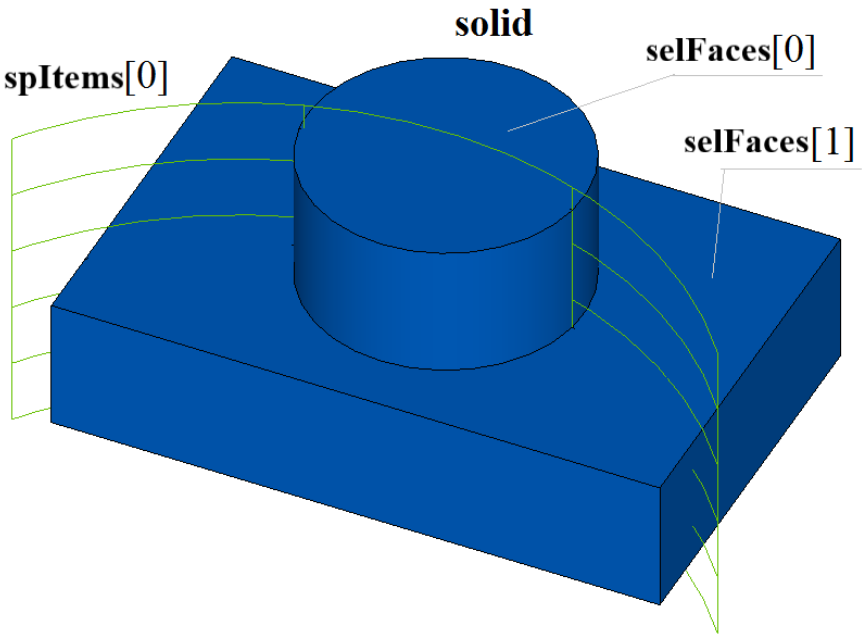


Рис. 1

На рис. 2 приведено построенное тело **result** с разбитыми на части указанными гранями. Новые рёбра на запрос MbEdge::IsSplit() возвращают *true*.

На рис. 3 разбитые грани построенного тела раскрашены в разные цвета:

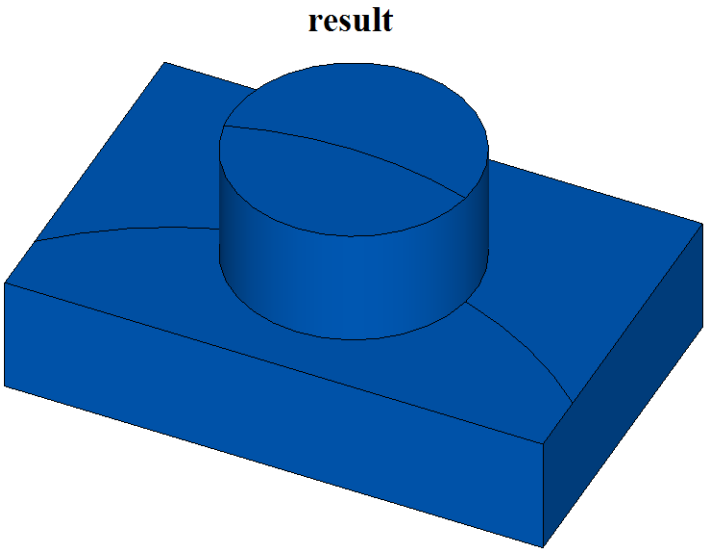


Рис. 2

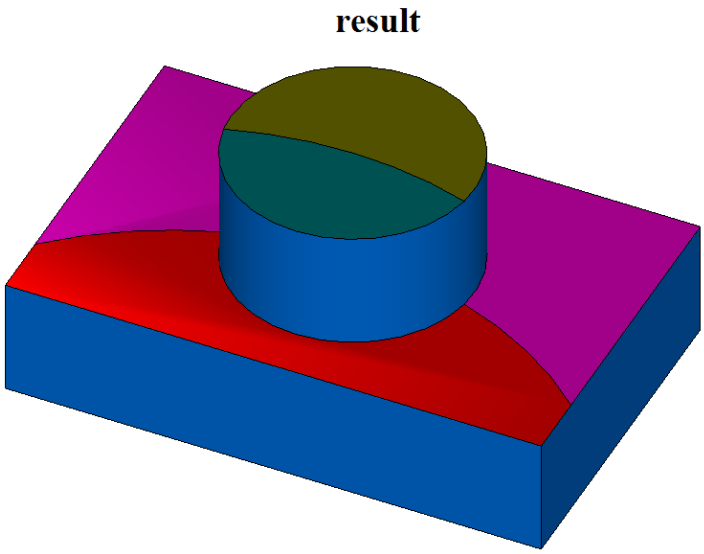


Рис. 3

Функция

```
MbResultType SplitSolid(       MbSolid &              solid,
                               MbeCopyMode            sameShell,
                         const MbPlacement3D &        spPlace,
                               MbeSenseValue          spType,
                         const RPArray<MbContour> &   spContours,
                               bool                   spSame,
                               RPArray<MbFace> &      selFaces,
                         const MbMergingFlags &       flags,
                         const MbSNameMaker &         names,
                               MbSolid *&             result );
```

выполняет те же действия, что и рассмотренная выше функция, с той разницей, что вместо разбивающих объектов **spItems** разбиение граней тела **solid** выполняют поверхности, построенные выдавливанием двумерных контуров **spContours**, расположенных в плоскости XY локальной системы координат **spPlace**. Выдавливание контуров выполняется в направлении оси **spPlace.axisZ** локальной системы координат контуров на длину, обеспечивающую полное пересечение исходного тела.

Функции SplitSolid() добавляют в журнал построенного тела строитель MbSplitShell, который содержит все необходимые данные для выполнения операции. Строитель MbSplitShell объявлен в файле *cr_split_shell.h*.

В тестовом приложении *test.exe* разбиение указанных граней тела выполняется командой меню «Создать → Тело → Обработкой граней → Разбивкой грани».

3.2.16 Построение отверстия, кармана или паза в теле

Функция

```
MbResultType HoleSolid( const c3d::SolidSPtr &      solid,
                              MbeCopyMode           sameShell,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                        const MbHoleSolidParams &   params,
                              c3d::SolidSPtr &      result );
```

выполняет построение отверстия, кармана или фигурного паза в теле.

Входными параметрами функции являются:

- **solid** - исходное тело (может быть ноль);
- **sameShell** - вариант копирования тела;
- **params** - параметры построения.

Для инициализации объекта **params** используется следующий конструктор (см. *op_shell_parameter.h*):

```
MbHoleSolidParams( const MbPlacement3D &   place,
                   const HoleValues &      params,
                   const MbSNameMaker &    operNames )
: _place     ( place                  )
, _parameters( params                 )
, _operNames ( &operNames.Duplicate() ){}
```

где

- **place** - локальная система координат;
- **params** - параметры построения;
- **operNames** - именователь.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

Функция строит вспомогательное тело в форме удаляемого объема для отверстия, кармана или паза.

Если исходное тело **solid** задано, функция возвращает разность исходного тела и вспомогательного тела.

Если исходное тело не задано (**solid** = 0), то функция возвращает вспомогательное тело (сверло, бобышка).

Для выполнения операции вспомогательное тело должно пересекать исходное тело.

Параметр **sameShell** управляет передачей граней, ребер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырех значений:

- *cm_Same = 0* (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **params** задает форму отверстия, кармана или паза.

Построения выполняются в локальной системе координат **place** с учетом углов поворота **params**.*placeAngle* и **params**.*azimuthAngle*.

Параметр **params**.*placeAngle* определяет угол поворота локальной системы координат **place** относительно оси **place**.**axisY**.

Параметр **params**.*azimuthAngle* определяет угол поворота локальной системы координат **place** относительно оси **place**.**axisZ**.

Поверхность **params**.*surface* может быть не задана. Если **params**.*surface* не равна нулю, то она служит для правильной обработки входа в отверстие, карман или паз.

На рис. 1 приведены данные, используемые при построении, и схема наследования параметров построения от абстрактного класса HoleValues.

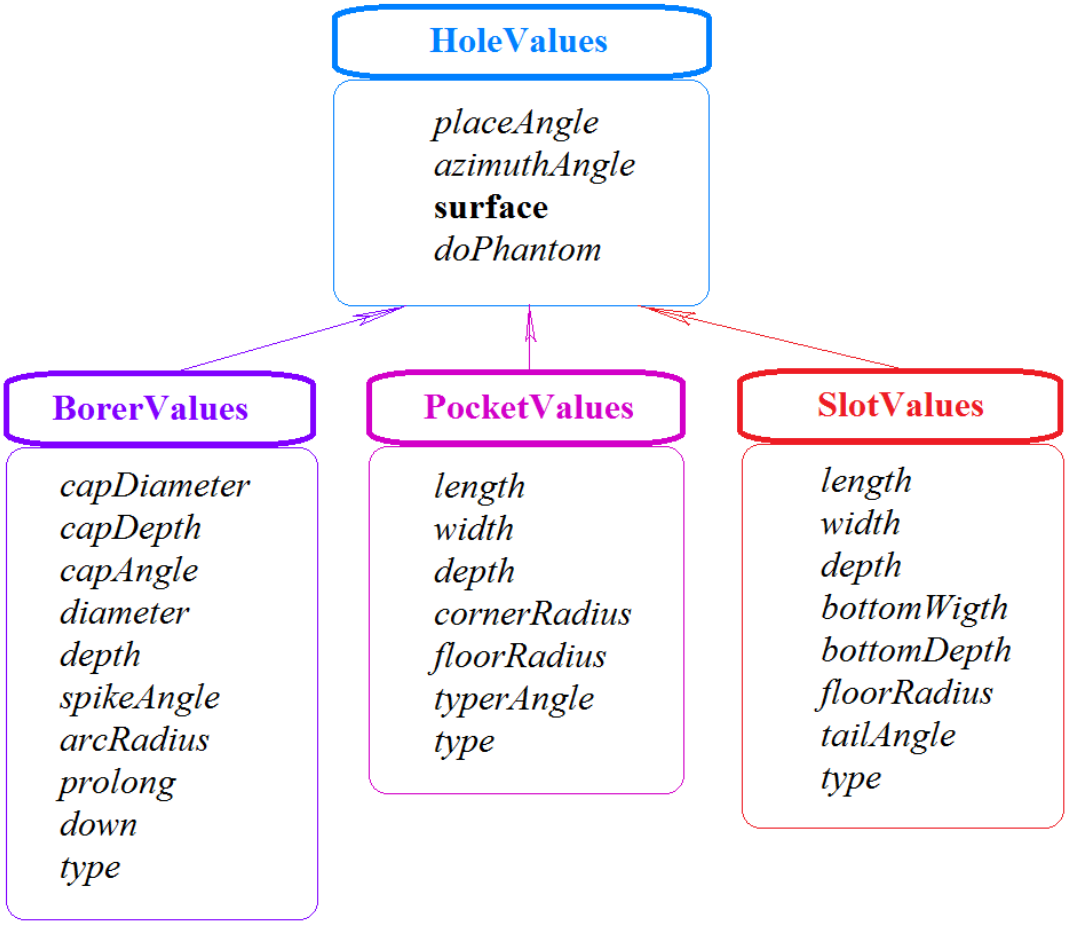


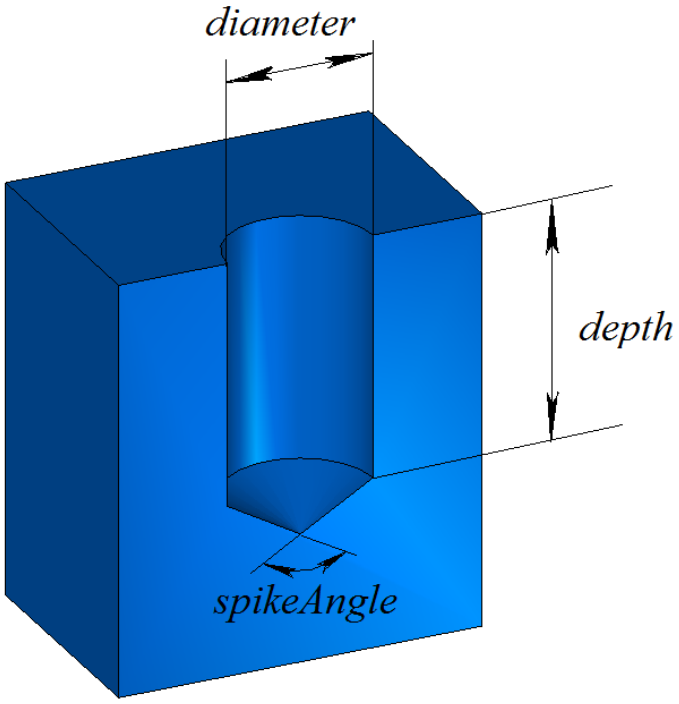
Рис. 1

Для построения отверстия следует использовать параметры BorerValues. Поддерживаются следующие типы отверстий, которые определяются параметром BorerValues::*type*, принимающим значения:

- bt_SimpleCylinder;
- bt_TwofoldCylinder;

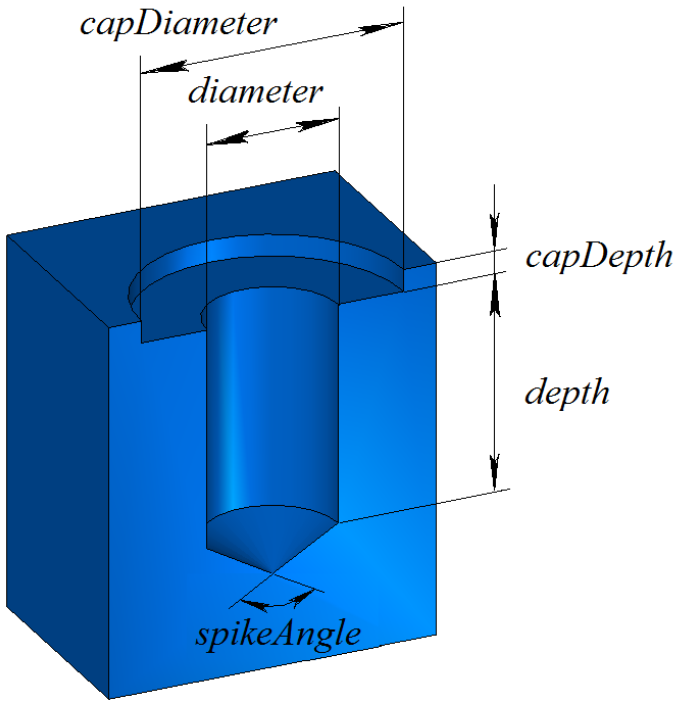
- bt_ChamferCylinder;
- bt_ComplexCylinder;
- bt_SimpleCone;
- bt_ArcCylinder;
- bt_DoubleCylinder.

На рис. 2...8 приведены тела с отверстиями различной формы:



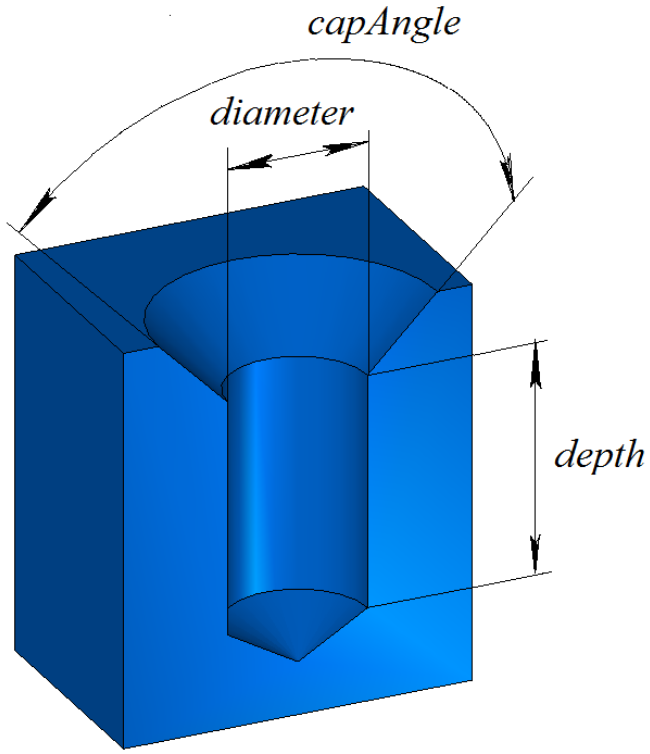
BorerValues::*type* = bt_SimpleCylinder

Рис. 2



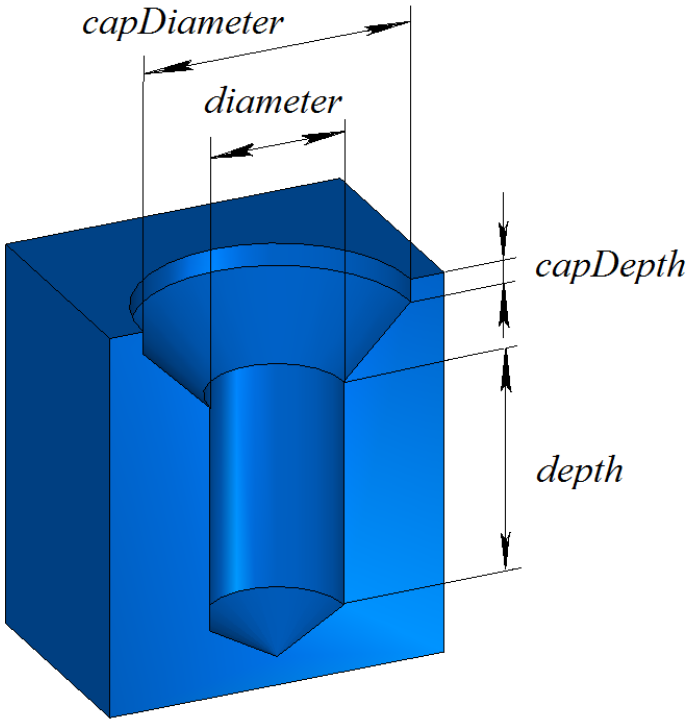
BorerValues::*type* = bt_TwofoldCylinder

Рис. 3



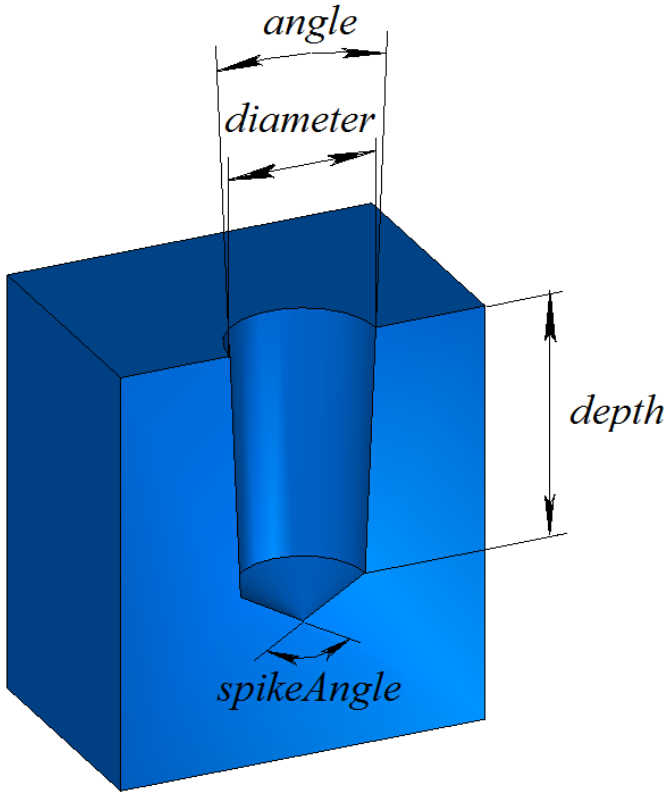
BorerValues::*type* = bt_ChamferCylinder

Рис. 4



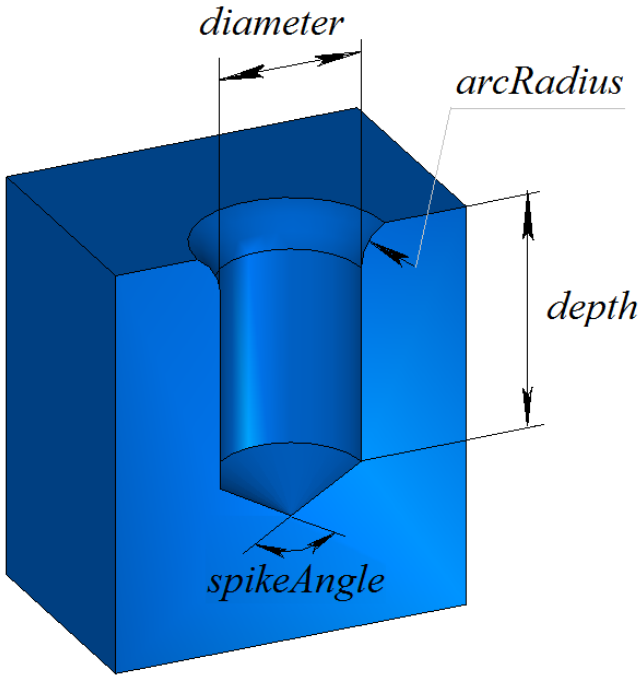
BorerValues::*type* = bt_ComplexCylinder

Рис. 5



BorerValues::*type* = bt_SimpleCone

Рис. 6



BorerValues::*type* = bt_ArcCylinder

Рис. 7

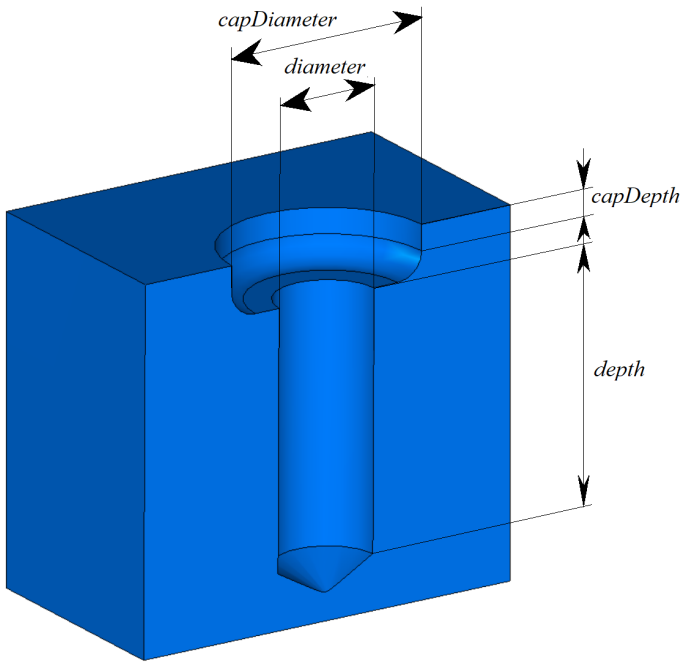


Рис. 8

Для построения кармана или бобышки следует использовать параметры PocketValues.

При PocketValues::*type* = *false* по заданным параметрам **params** строится карман, при PocketValues::*type* = *true* по заданным параметрам **params** строится бобышка.

На рис. 9 приведено тело с карманом прямоугольной формы без уклона боковых граней.

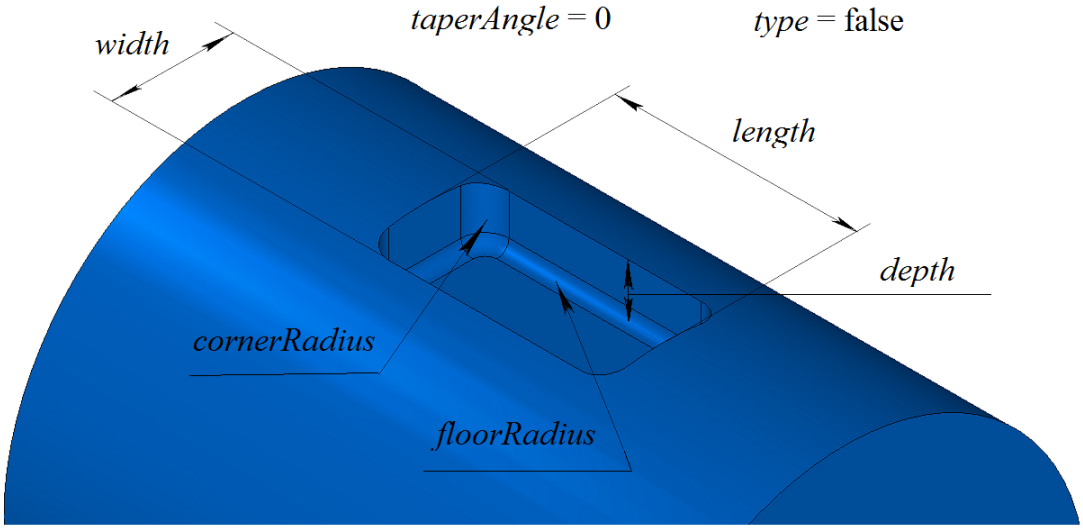


Рис. 9

Для построения паза следует использовать параметры SlotValues. Поддерживаются четыре типа пазов, которые определяются параметром SlotValues::*type*:

- st_BallEnd - цилиндрический в донной части;
- st_Rectangular - прямоугольный;
- st_TShaped - Т-образный;
- st_DoveTail - «Ласточкин хвост».

Функция HoleSolid() добавляет в журнал построенного тела строитель MbHoleSolid, который содержит все необходимые данные для выполнения операции. Строитель MbHoleSolid объявлен в файле *cr_hole_solid.h*.

В тестовом приложении *test.exe* построение отверстий, карманов или пазов в теле выполняется командами меню:

- «Создать → Тело → На базе тела → С отверстием»;
- «Создать → Тело → На базе тела → С карманом или бобышкой»;
- «Создать → Тело → На базе тела → С пазом».

3.2.17 Построение тела с ребром жёсткости

Функция

```
MbResultType RibSolid(       MbSolid &              solid,
                             MbeCopyMode            sameShell,
                       const MbRibSolidParameters & pars,
                             c3d::SolidSPtr &       result );
```

по заданному контуру строит ребро жёсткости и объединяет его с исходным телом. Сегмент контура с указанным номером устанавливает вектор уклона боковых граней ребра жёсткости.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – режим копирования исходного тела;
- **pars** – параметры ребра жёсткости.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** к построенному телу **result** и может принимать одно из четырёх значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode описано в параграфе *Копирование множества граней MbFaceShell*.

В классе MbRibSolidParameters для инициализации объекта **pars** задан конструктор (см. *op_shell_parameter.h*):

```
MbRibSolidParameters( const MbPlacement3D &  place,
                      const MbContour &      contour,
                            size_t           index,
                      const RibValues &      pars,
                      const MbSNameMaker &   names );
```

где

- **place** – локальная система координат, на плоскости XY которой расположен образующий контур. Ось Z этой системы определяет основное направление выдавливания ребра (до применения параметров *side* и углов уклона).
- **contour** – формообразующий контур на плоскости XY системы координат **place**;
- **index** – номер сегмента контура (начиная с 0), который определяет вектор уклона боковых граней ребра жёсткости. Направление нормали этого сегмента задаёт направление уклона. Если указан некорректный номер сегмента (за пределами количества сегментов контура), функция вернёт код ошибки;
- **pars** – параметры формообразования ребра жёсткости;
- **names** – именователь операции.

Объект **pars** класса RibValues содержит данные, приведённые на рис. 1.

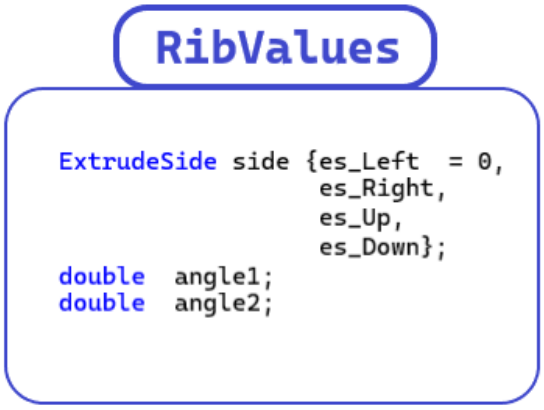


Рис. 1

Структура RibValues определена в файле *swept_parameter.h*.

Параметр **pars**.*side* определяет сторону заполнения пространства телом ребра:

1. При **pars**.*side* = *es_Left* ребро жёсткости выдавливается в левую сторону от кривой вдоль плоскости XY.
2. При **pars**.*side* = *es_Right* ребро жёсткости выдавливается в правую сторону от кривой вдоль плоскости XY.

3. При **pars**.*side* = *es_Up* ребро жёсткости выдавливается в сторону нормали плоскости XY.
4. При **pars**.*side* = *es_Down* ребро жёсткости выдавливается в сторону, противоположную нормали плоскости XY.

Параметры **pars**.*angle1* и **pars**.*angle2* определяют углы уклона плоскости ребра в прямом и обратном направлениях соответственно.

На рис. 2 показаны исходное тело **solid**, локальная система координат **place**, в плоскости XY которой расположен контур **contour**:

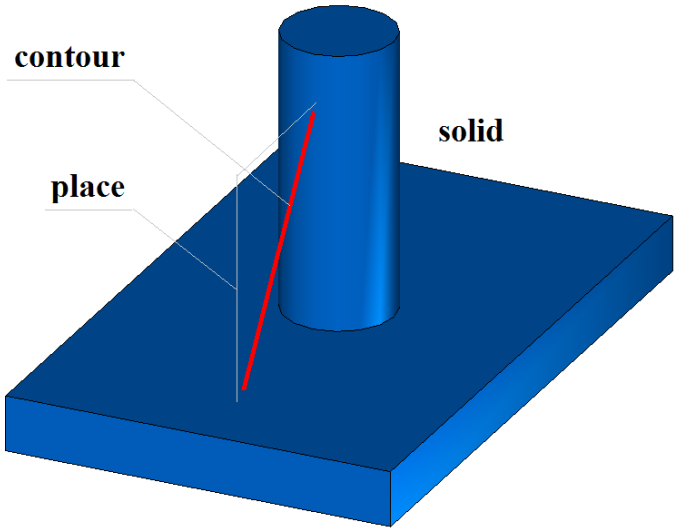


Рис. 2

На рис. 3 приведён результат построения ребра жёсткости без уклона боковых граней ребра жёсткости:

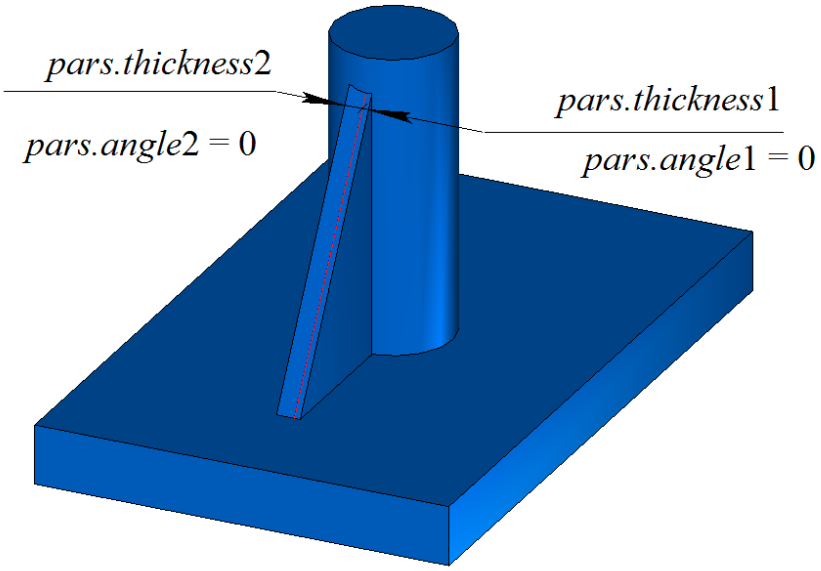


Рис. 3

На рис. 4 приведён результат построения ребра жёсткости с уклоном боковых граней ребра жёсткости:

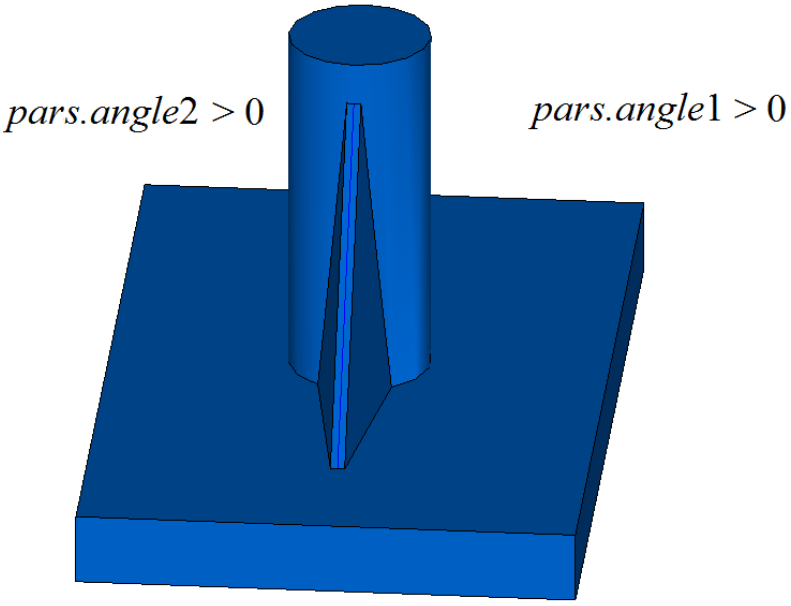


Рис. 4

Функция RibSolid() добавляет в журнал построенного тела строитель MbRibSolid, который содержит все необходимые данные для выполнения операции. Строитель MbRibSolid объявлен в файле *cr_rib_solid.h*.

Для создания отдельного ребра без объединения с исходным телом рекомендуется использовать функцию RibElement(). Она имеет аналогичные с RibSolid() параметры, но возвращает только тело ребра жёсткости без булевой операции объединения; это полезно, когда ребро необходимо обработать отдельно перед интеграцией в модель.

В тестовом приложении *test.exe* выполнить построение ребра жёсткости можно с помощью команды меню «Создать → Тело → На базе тела → С ребром жёсткости».

3.2.18 Уклонение граней тела

Функция

```
MbResultType DraftSolid(       MbSolid &             solid,
                               MbeCopyMode           sameShell,
                         const MbDraftSolidParams  & draftParams,
                               MbDraftSolidResults & result );
```

выполняет построение тела с уклонением указанных граней тела от нейтральной изоплоскости или нейтральных рёбер на заданный угол.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – режим копирования исходного тела;
- **draftParams** – параметры уклона.

Выходным параметром функции является объект **result** класса MbDraftSolidResults, содержащий построенное тело уклона и атрибут хот-точки уклона.

Функция возвращает код результата операции из перечисления MbResultType.

Функция объявлена в файле *action_solid.h*.

Функция строит тело, грани которого уклонены относительно их положения в исходном теле **solid**.

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырёх значений:

- *cm_Same* = 0 (оболочка не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходная оболочка и её копия имеют общие базовые поверхности);
- *cm_Copy* (исходная оболочка и её копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **draftParams** задаёт параметры построения.

В классе MbDraftSolidParams для инициализации объекта **draftParams** с параметрами уклона используются следующие конструкторы (см. *op_shell_parameter.h*):

Конструкторы по нейтральной плоскости и именователю операции:

```
MbDraftSolidParams( const c3d::FacesSPtrVector & faces,
                    const MbPlacement3D &        neutralPlane,
                    const MbSNameMaker &         names,
                          double                 angle,
                          MbeFacePropagation     faceProp,
                          bool                   reverse );
```

```
MbDraftSolidParams( const c3d::FacesSPtrVector & faces,
                    const MbPlacement3D &        neutralPlane,
                    const MbSNameMaker &         names,
                          double                 angle,
                          MbeFacePropagation     faceProp,
                          bool                   reverse,
                          bool                   rebuildFillets );
```

```
MbDraftSolidParams( const RPArray<MbFace> &   faces,
                    const MbPlacement3D &     neutralPlane,
                    const MbSNameMaker &      names,
                          double              angle,
                          MbeFacePropagation  faceProp,
                          bool                reverse );
```

где

- **faces** – уклоняемые грани;
- **neutralPlane** – нейтральная плоскость;
- **names** – именователь;
- **angle** – угол уклона;
- **faceProp** – признак захвата граней, гладко стыкующихся с уклоняемыми гранями;
- **reverse** – флаг обратного направления уклона.

Конструкторы по набору рёбер и именователю:

```
MbDraftSolidParams( const c3d::FacesSPtrVector & faces,
                    const c3d::EdgesSPtrVector & edges,
                    const MbVector3D &           dir,
                    const MbSNameMaker &         names,
                          double                 angle,
                          MbeFacePropagation     faceProp,
                          bool                   reverse );
```

```
MbDraftSolidParams( const c3d::FacesSPtrVector & faces,
                    const c3d::EdgesSPtrVector & edges,
                    const MbVector3D &           dir,
                    const MbSNameMaker &         names,
                          double                 angle,
                          MbeFacePropagation     faceProp,
                          bool                   reverse,
                          bool                   rebuildFillets );
```

где

- **faces** – уклоняемые грани;
- **edges** – нейтральные рёбра;
- **dir** – базовое направление;
- **names** – именователь;
- **angle** – угол уклона;
- **faceProp** – признак захвата граней, гладко стыкующихся с уклоняемыми гранями;
- **reverse** – флаг обратного направления уклона;
- **rebuildFillets** – флаг перестройки прилежащих скруглений.

Параметр **angle** задаёт угол уклона указанных граней тела из множества **faces** относительно нейтральной плоскости **neutralPlane** или относительно нейтральных рёбер **edges** и направления **dir** (в зависимости от используемого конструктора).

Параметр **faceProp** управляет добавлением к множеству уклоняемых граней **faces** других граней тела, гладко стыкующихся с уклоняемыми гранями.

Параметр **reverse** управляет направлением уклона.

Параметр **rebuildFillets** управляет перестройкой прилежащих скруглений.

Параметр **names** обеспечивает именование граней, рёбер и вершин построенного тела.

На рис. 1 показаны исходное тело **solid**, нейтральная плоскость **neutralPlane**, относительно плоскости XY которой выполняется уклон граней, и уклоняемые грани **faces**:

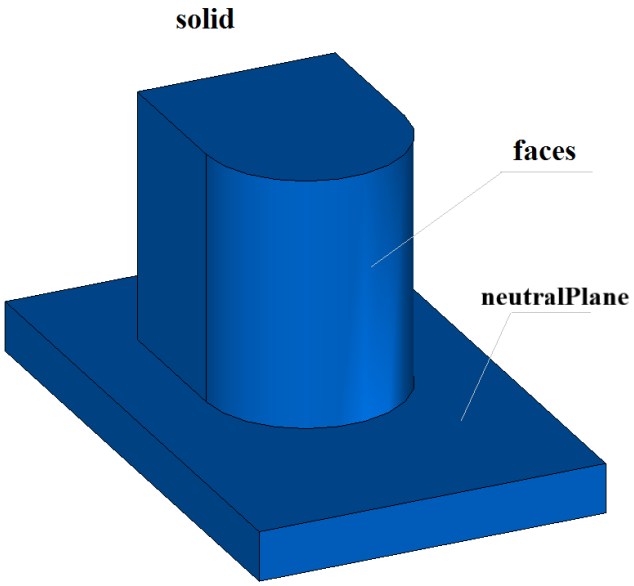


Рис. 1

На рис. 2 приведён результат построения тела с уклоном указанных граней:

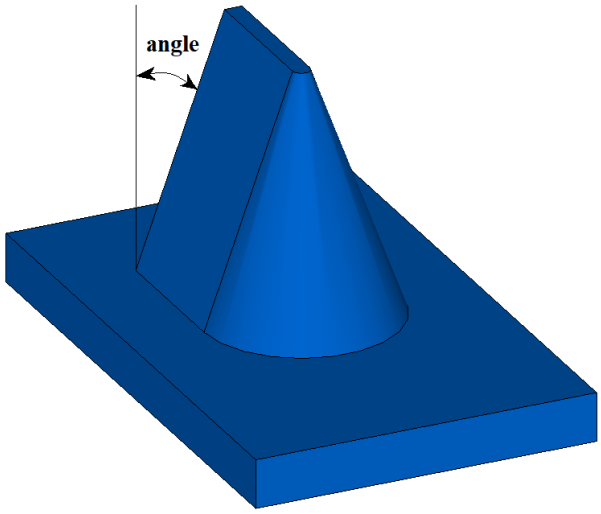


Рис. 2

Функция DraftSolid() добавляет в журнал построенного тела строитель MbDraftSolid, который содержит все необходимые данные для выполнения операции. Строитель MbDraftSolid объявлен в файле *cr_draft_solid.h*.

В тестовом приложении *test.exe* уклонение указанных граней тела выполняется командой меню «Создать→Тело→Обработкой граней→Уклонением граней».

3.2.19 Размножение тела

Функция

```
MbResultType DuplicationSolid( const MbSolid &                    solid,
                               const MbDuplicationSolidParams &   params,
                                     c3d::SolidSPtr &             result );
```

выполняет построение копий исходного тела, трансформацию их по указанному закону и объединение в одно тело.

Входными параметрами функции являются:

- **solid** - исходное тело;
- **params** - параметры построения.

В классе MbDuplicationSolidParams для инициализации объекта **params** с параметрами построения используются следующие конструкторы (см. *op_shell_parameter.h*):

```
MbDuplicationSolidParams( const DPtr<DuplicationValues> &  values,
                                IProgressIndicator *       prog = nullptr )

_duplicationValues ( values )
, _selIndices      (        )
, _adaptMode       ( false  )
, _names           (        )
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
, _progress        ( prog   )
{
   if ( _duplicationValues == nullptr )
   {
      C3D_ASSERT( _duplicationValues != nullptr );
      _duplicationValues = new DuplicationMatrixValues();
   }
}
```

```
MbDuplicationSolidParams( const DPtr<DuplicationValues> &  values,
                          const MbSNameMaker &             names,
                                IProgressIndicator *       prog = nullptr )

_duplicationValues ( values )
, _selIndices        (        )
, _adaptMode         ( false  )
, _names             (        )
, _progress          ( prog   )
{
   if ( _duplicationValues == nullptr )
   {
      C3D_ASSERT( _duplicationValues != nullptr );
      _duplicationValues = new DuplicationMatrixValues();
   }
}
```

Конструктор на основе класса DuplicationMeshValues для копирования по сетке:

```
MbDuplicationSolidParams(       bool                  isPolar,
                          const MbVector3D &          dir1,
                          const double                step1,
                          const uint                  num1,
                          const MbVector3D &          dir2,
                          const double                step2,
                          const uint                  num2,
                          const MbCartPoint3D *       center,
                                bool                  isAlongAxis,
                                IProgressIndicator *  prog = nullptr )

_duplicationValues( new DuplicationMeshValues( isPolar, dir1, step1, num1, dir2,␣
↪step2, num2, center, isAlongAxis ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

Конструкторы на основе класса DuplicationMatrixValues для копирования по матрицам:

```
MbDuplicationSolidParams( const MbMatrix3D &          matr,
                                IProgressIndicator *  prog = nullptr )
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
_duplicationValues( new DuplicationMatrixValues( matr ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

```
MbDuplicationSolidParams( const SArray<MbMatrix3D> &   matr,
                          IProgressIndicator *   prog = nullptr )

_duplicationValues( new DuplicationMatrixValues( matr ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

```
MbDuplicationSolidParams( const std::vector<MbMatrix3D> &  matr,
                          IProgressIndicator *       prog = nullptr )

: _duplicationValues( new DuplicationMatrixValues( matr ) )
, _selIndices()
, _adaptMode( false )
, _names()
, _progress( prog )
{}
```

где

- **_duplicationValues** - параметры копирования;
- **_selIndices** - индексы выбранных граней оболочки;
- **_adaptMode** - режим адаптации открытых оболочек;
- **_names** - именователь тел и граней, создаваемых операцией;
- **IProgressIndicator * _progress** - индикатор прогресса выполнения операции.

Выходным параметром функции является построенное тело **result**.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_solid.h*.

Параметр **_names** обеспечивает именование граней, рёбер и вершин построенного тела.

Параметр **values** задаёт параметры построения. На рис. 1 приведены данные, используемые при построении, и схема наследования параметров построения от абстрактного класса DuplicationValues:

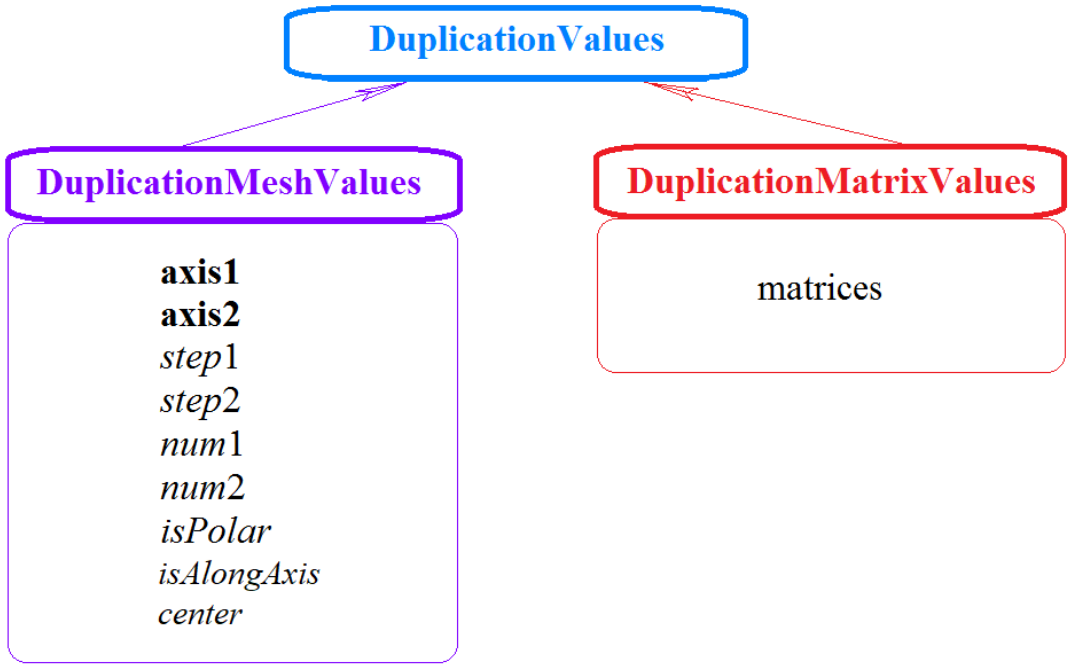


Рис. 1

Для размножения тела, копии которого располагаются по двумерной сетке, следует использовать параметры DuplicationMeshValues.

Поддерживаются два типа размножения: по двум направлениям и по полярной сетке.

Параметр **values**.*isPolar* определяет тип сетки.

axis1 и **axis2** - векторы в глобальной системе координат, задающие направления сетки (линейной или полярной).

Если **values**.*isPolar* = *false*, то исходное тело и построенные копии располагаются в узлах двумерной сетки с осями **values**.*axis1* и **values**.*axis2*. Началом отсчёта служит исходное тело. Вдоль оси **values**.*axis1* располагаются **values**.*num1* копий тела с шагом **values**.*step1*, вдоль оси **values**.*axis2* располагаются **values**.*num2* копий тела с шагом **values**.*step2*, включая исходное тело.

Если **values**.*isPolar* = *true*, построенные копии располагаются в узлах полярной сетки. Началом отсчёта служит исходное тело. Радиальное направление сетки определяется вектором **values**.*axis1*, а ось вращения определяется векторным произведением **values**.*axis1* и **values**.*axis2*. Вдоль радиальных направлений располагаются **values**.*num1* копий тела с шагом **values**.*step1*, на каждой окружности располагаются **values**.*num2* копий тела с угловым шагом **values**.*step2*.

Если **values**.*isAlongAxis* = *true*, то тела располагаются вдоль радиальной оси. Если **values**.*isAlongAxis* = *false*, то тела располагаются параллельно исходному телу.

Параметр **values**.*center* определяет центр локальной системы координат (а также точку приложения оси вращения в случае полярной системы).

Для размножения тела, копии которого трансформированы по набору матриц, следует использовать параметры DuplicationMatrixValues. Множество матриц трансформации определено параметром **values**.*matrices*.

Если после построения копии и исходное тело пересекаются друг с другом, то выполняется булева операция объединения пересекшихся тел. На рис. 2 приведён результат построения размноженного по полярной сетке тела:

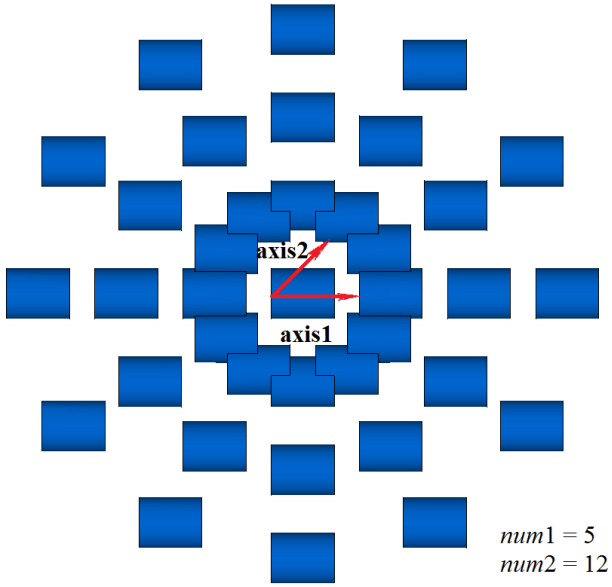


Рис. 2

Функция DuplicationSolid() добавляет в журнал построенного тела строитель MbDuplicationSolid, который содержит все необходимые данные для выполнения операции. Строитель MbDuplicationSolid объявлен в файле *cr_duplication_solid.h*.

В тестовом приложении *test.exe* размножение тела выполняется командами меню «Создать→Тело→На базе тела→Размножением по сетке» и «Создать→Тело→На базе тела→Размножением матрицами».

3.2.20 Разделение оболочки на части по заданному набору ребер

Функция

```
MbResultType DivideShell(       MbSolid &               solid,
                                MbeCopyMode             sameShell,
                          const MbDivideShellParams &   params,
                                c3d::SolidSPtr &        resSolid );
```

выполняет разделение оболочки, которая находится в **solid**, по заданному в параметрах набору рёбер. Разделение тела/оболочки выполняется путём удаления в рёбрах тополо-

гических связей между смежными гранями. После выполнения операции весь результат будет находиться в результирующей оболочке.

Входными параметрами являются:

- **solid** - исходное тело (оболочка), которое требуется разделить;
- **sameShell** - вариант копирования исходного тела (см. описание MbeCopyMode);
- **params** - параметры операции разделения оболочки (набор рёбер и дополнительные настройки).

Выходным параметром является возвращаемое тело **resSolid**, содержащее результирующую оболочку.

Функция возвращает код результата операции из перечисления *MbResultType*.

Функция объявлена в файле *action_shell.h*.

Класс MbDivideShellParams содержит входные данные для выполнения операции разделения оболочки. При создании объекта класса MbDivideShellParams его можно сразу инициализировать вектором рёбер, по которым будет производиться разделение.

Конструкторы класса MbDivideShellParams:

```
MbDivideShellParams( const RPArray<MbCurveEdge> &   edges,
                     const MbSNameMaker &           operNames );
```

```
MbDivideShellParams( const c3d::EdgesSPtrVector &   edges,
                     const MbSNameMaker &           operNames );
```

```
MbDivideShellParams( const MbSNameMaker &           operNames );
```

где

- **edges** - вектор рёбер, по которым будет производиться разделение;
- **operNames** - именователь.

В результате успешной работы функции исходное тело **solid** будет разделено, и все полученные незамкнутые тела будут помещены в оболочку возвращаемого тела **resSolid**.

Тестовое приложение *test.exe* выполняет разбиение указанных граней тела командами меню «Создать→Оболочку→На основе оболочки→Разделением по набору ребер».

3.3 МЕТОДЫ ПОСТРОЕНИЯ ДВУМЕРНЫХ КРИВЫХ

Двумерные кривые используются для описания области определения поверхностей, для работы с кривыми на поверхностях, для построения кривых пересечения поверхностей, для построения поверхностей сопряжения. При построении твердых тел двумерные кривые применяются в качестве входных параметров в виде элементов эскизов. Кроме того, двумерные кривые служат элементами плоских проекций геометрических моделей.

Все двумерные кривые являются наследниками класса MbCurve и представлены в разделе *КРИВЫЕ ДВУМЕРНОГО ПРОСТРАНСТВА*. Кривые могут быть построены непосредственным вызовом соответствующих конструкторов или функциями, приведёнными в данном разделе.

3.3.1 Построение двумерной прямой линии и отрезка

Функция

```
MbResultType Line( const MbCartPoint &   point1,
                   const MbCartPoint &   point2,
                         MbCurve *&      result );
```

выполняет построение двумерной прямой по двум несовпадающим точкам.

Входными параметрами функции являются:

- **point1** - первая точка, по которой проходит прямая;
- **point2** - вторая точка, по которой проходит прямая.

Выходным параметром функции является построенная прямая линия **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Параметр **point1** определяет начальную точку прямой, соответствующую нулевому параметру прямой. Касательный вектор (нормализованная производная) прямой линии имеет единичную длину и направлен от **point1** к **point2**, рис. 1.

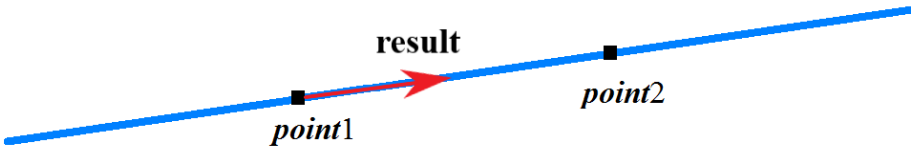


Рис. 1

Прямая линия описана в подразделе *Двумерная прямая MbLine*.

Функция

```
MbResultType Segment( const MbCartPoint &   point1,
                      const MbCartPoint &   point2,
                            MbCurve *&      result );
```

выполняет построение двумерного отрезка прямой по двум несовпадающим точкам.

Входными параметрами функции являются:

- **point1** - начальная точка отрезка;
- **point2** - конечная точка отрезка.

Выходным параметром функции является построенный отрезок **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Параметр **point1** определяет начальную точку отрезка прямой, соответствующую нулевому параметру. Параметр **point2** определяет конечную точку отрезка прямой, соответствующую параметру, равному единице, рис. 2:

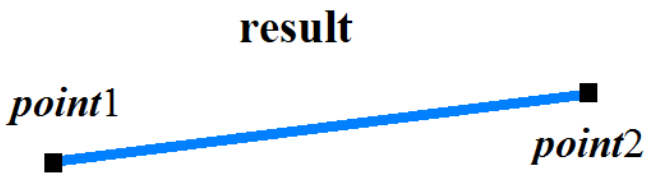


Рис. 2

Отрезок прямой описан в подразделе *Двумерный отрезок прямой MbLineSegment*.

3.3.2 Построение двумерной окружности, эллипса и их дуг

Функция

```
MbResultType  Arc(       MbeArcCreateWay          createWay,
                  const  MbCartPoint &            center,
                  const  c3d::ParamPointsVector & points,
                         double &                 a,
                         double &                 b,
                         double &                 c,
                         bool                     option,
                         MbCurve *&               result );
```

выполняет построение двумерной дуги эллипса, в частном случае функция выполняет построение двумерной дуги окружности.

Входными параметрами функции являются:

- **createWay** – способ создания эллипса (окружности) (см. перечисление MbeArcCreateWay):
  - *acw_CircleByCenterAndRadius* – окружность по центру и радиусу,
  - *acw_ArcByCenterAnd2Points* – дуга окружности по центру и двум точкам,
  - *acw_ArcByCenterAnd2Angles* – дуга окружности по центру и двум углам,
  - *acw_ArcBy3Points* – дуга окружности по трём точкам,
  - *acw_EllipseByCenterAndSemiaxis* – эллипс с заданными полуосями и углом наклона,
  - *acw_EllipseByCenterAnd3Points* – эллипс по центру и трём точкам,
  - *acw_EArcByCenterAnd2Points* – дуга эллипса, обрезанная двумя лучами из центра к заданным точкам;
- **center** – центр эллипса (окружности);
- **points** – конечные точки или точки, через которые проходит кривая;
- **a** – интерпретация параметра зависит от способа создания дуги (см. перечисление MbeArcCreateWay);
- **b** – интерпретация параметра зависит от способа создания дуги (см. перечисление MbeArcCreateWay);

- **c** – интерпретация параметра зависит от способа создания дуги (см. перечисление MbeArcCreateWay);
- **option** – направление кривой (*true* – по часовой стрелке, *false* – против часовой стрелки).

Выходным параметром функции является построенный эллипс (окружность) или его дуга **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Кривая может быть построена как по заданным точкам, так и по скалярным параметрам (значения полуосей, радиуса, углов наклона).

Параметр **center** определяет центральную точку эллипса.

Множество точек **points** может быть пустым, в этом случае должны быть ненулевыми длины полуосей эллипса **a** и **b**.

Если множество точек **points** содержит два элемента, то точки **center**, **points**[0], **points**[1] определяют плоскость расположения осей **axisX** и **axisY** локальной системы координат эллипса: ось **axisX** направлена из центра в точку **points**[0], ось **axisY** ортогональна оси **axisX** и направлена из центра в сторону точки **points**[1]. Расстояние между точками **center** и **points**[0] определяет длину первой полуоси эллипса **a**, а расстояние между точкой **center** и проекцией точки **points**[1] на ось **axisY** определяет длину второй полуоси эллипса **b**, рис. 1:

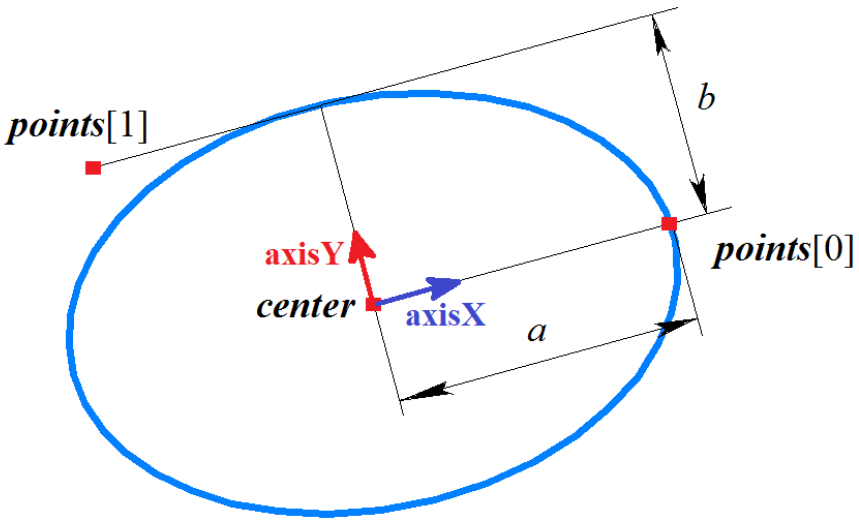


Рис. 1

Если множество точек **points** пусто и заданы **center** и радиус **c**, то будет построена окружность, а оси **axisX** и **axisY** локальной системы координат будут совпадать с глобальными осями координат (см. перечисление MbeArcCreateWay).

Если множество точек **points** содержит три элемента, то будет построен эллипс с центром в точке **center**, проходящий через точки **points**[0], **points**[1] и **points**[2]. Положение осей

локальной системы координат эллипса и длины полуосей эллипса будут вычислены по точкам **center**, **points**[0], **points**[1] и **points**[2], рис. 2.

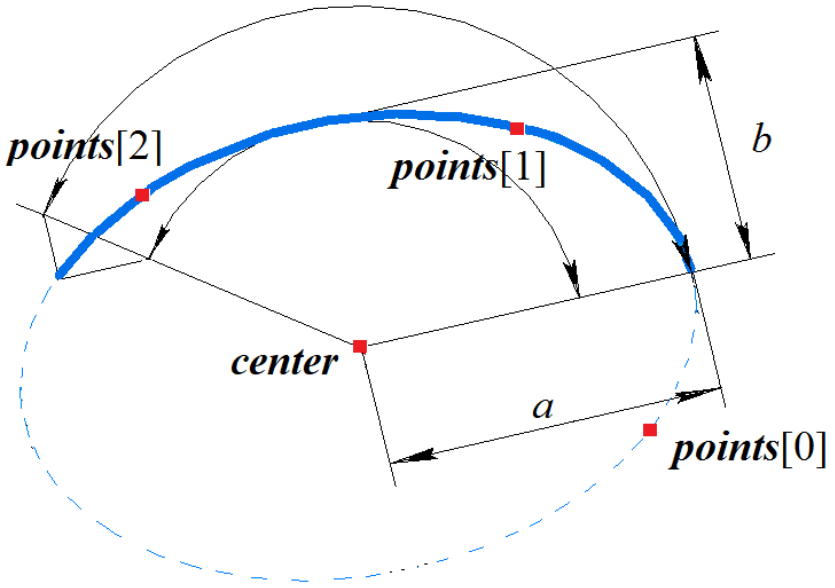


Рис. 2

Дуга эллипса описана в подразделе Двумерная дуга эллипса MbArc.

Если множество точек **points** пусто, то длины полуосей эллипса **a** и **b** и угол дуги **c** являются входными параметрами, а оси **axisX** и **axisY** локальной системы координат эллипса совпадают с глобальными осями координат.

В частном случае, когда длины полуосей эллипса равны, рассматриваемая функция выполняет построение окружности или дуги окружности.

3.3.3 Построение двумерной NURBS-кривой

Функция

```
MbResultType NurbsCurve( const SArray<MbCartPoint> & pointList,
                         const SArray<double> &      weightList,
                               size_t                degree,
                         const SArray<double> &      knotList,
                               bool                  curveClosed,
                               MbCurve *&            result );
```

выполняет построение двумерной NURBS-кривой на заданном множестве контрольных точек.

Входными параметрами функции являются:

- **pointList** - множество контрольных точек;
- **weightList** - множество весов контрольных точек;
- **degree** - порядок B-сплайна;

- **knotList** - множество параметрических узлов (узловой вектор);
- **curveClosed** - флаг циклической замкнутости кривой.

Выходным параметром функции является построенная кривая **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Рассматриваемая функция строит NURBS-кривую (кривую на основе *B*-сплайнов), которая описана в подразделе Двумерная NURBS-кривая MbNurbs.

Множество весов **weightList** должно быть согласовано с множеством контрольных точек **pointList**.

Порядок **degree** кривой не должен превосходить количество контрольных точек.

Параметр **curveClosed** определяет циклическую замкнутость построенной кривой.

Узловой вектор **knotList** представляет собой неубывающую последовательность действительных чисел и определяет область определения параметра кривой и форму кривой.

Для **curveClosed** = *false* узловой вектор должен содержать число элементов, равное числу контрольных точек плюс порядок кривой.

Для **curveClosed** = *true* узловой вектор должен содержать число элементов, равное числу контрольных точек плюс удвоенный порядок кривой и минус единица.

Чтобы незамкнутая NURBS-кривая проходила через крайние контрольные точки, необходимо, чтобы первые **degree** элементов узлового вектора **knotList** были равны между собой и последние **degree** элементов узлового вектора **knotList** были равны между собой.

На рис. 1 приведена замкнутая NURBS-кривая четвертого порядка:

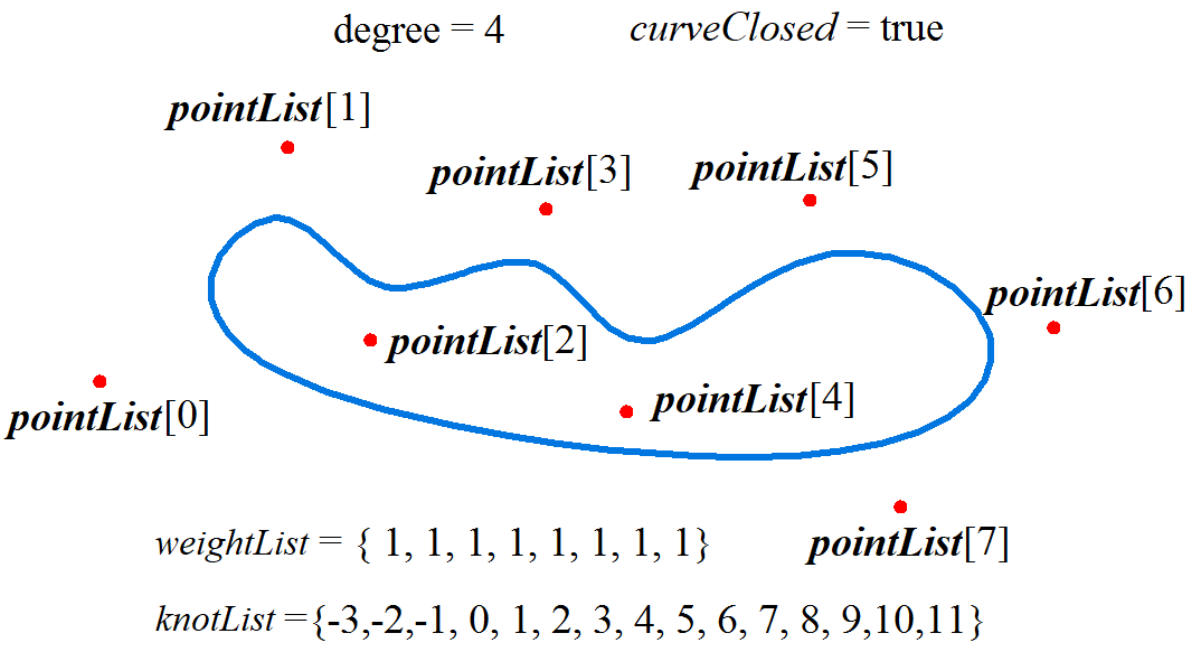


Рис. 1

На рис. 2 приведена незамкнутая NURBS-кривая четвертого порядка, проходящая через крайние контрольные точки:

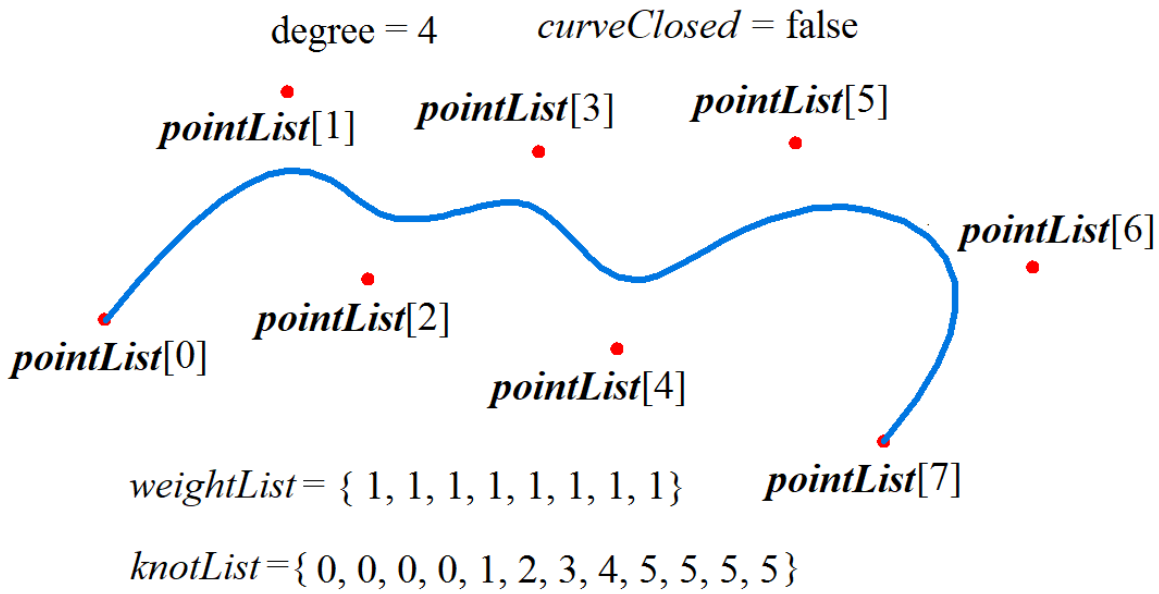


Рис. 2

На рис. 3 приведены три замкнутые NURBS-кривые различного порядка, построенные по одним и тем же контрольным точкам:

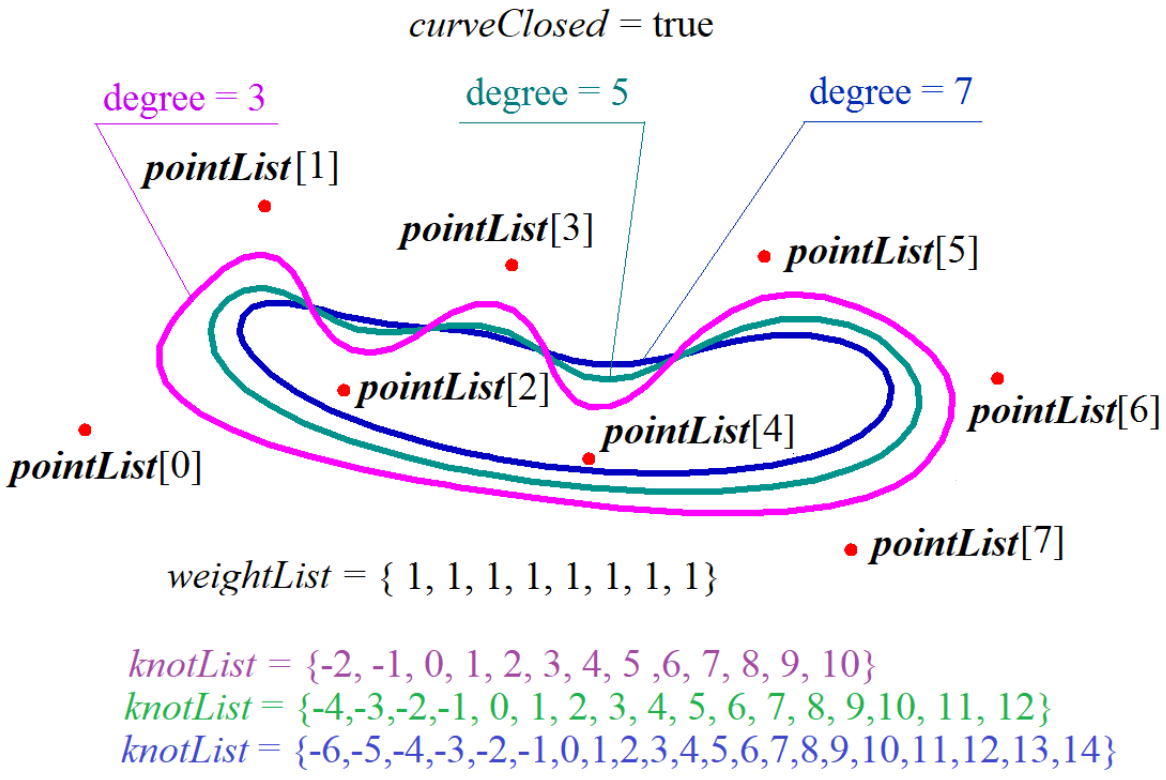


Рис. 3

На рис. 4 приведены три незамкнутые NURBS-кривые различного порядка, построенные по одним и тем же контрольным точкам:

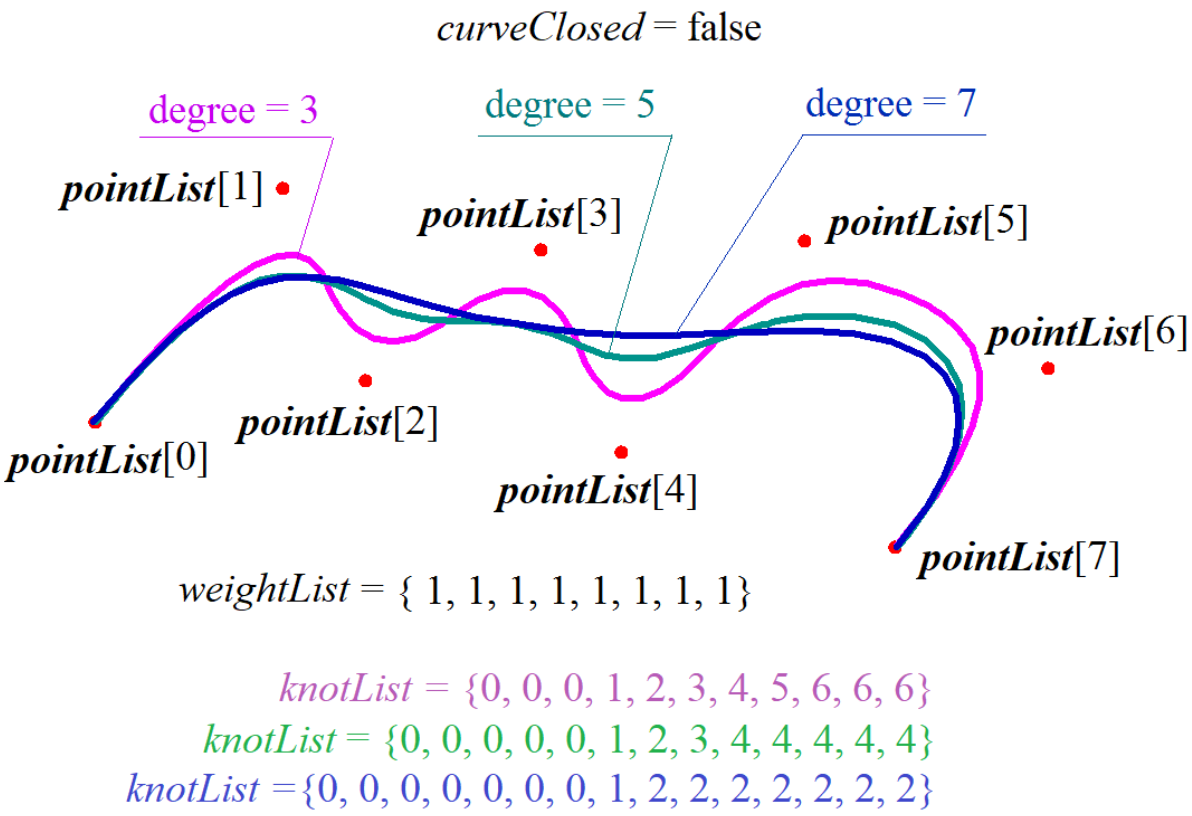


Рис. 4

На рис. 5 приведены три незамкнутые NURBS-кривые четвертого порядка, имеющие различные значения весов контрольных точек:

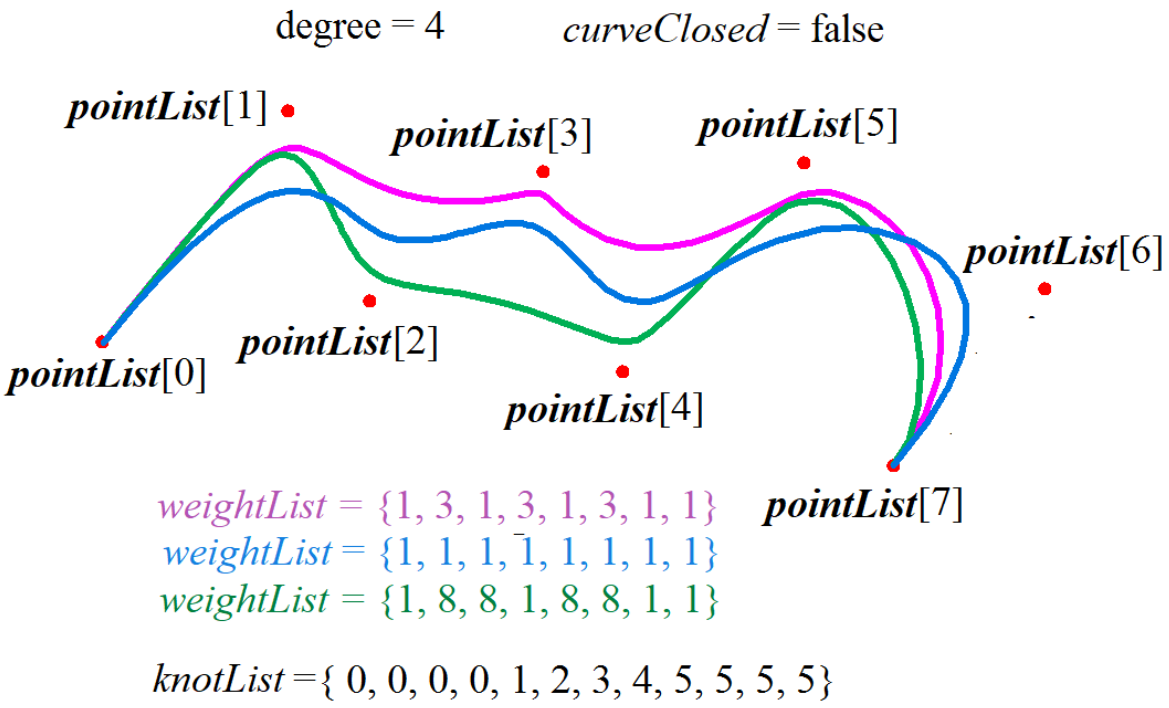


Рис. 5

На рис. 6 приведена незамкнутая NURBS-кривая третьего порядка, по форме совпадающая с дугой окружности. Расстояние между контрольными точками **pointList**[0] и **pointList**[1] равно расстоянию между контрольными точками **pointList**[1] и **pointList**[2], а вес *weightList*[1] средней контрольной точки кривой равен весу крайних контрольных точек, умноженному на косинус половины угла дуги.

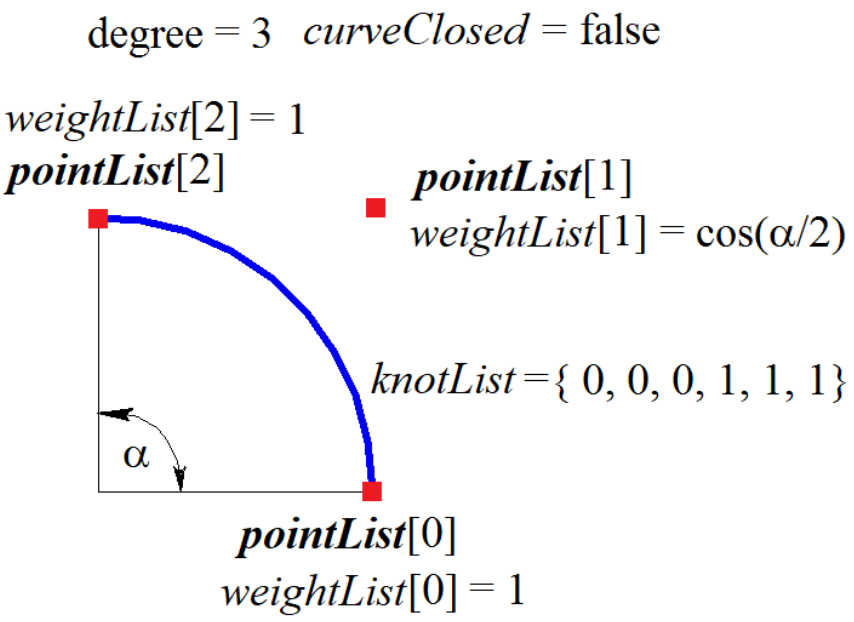


Рис. 6

Функция

```
MbResultType NurbsCopy( const MbCurve &   curve,
                              MbCurve *&  result );
```

выполняет построение NURBS-копии заданной двумерной кривой.

Входным параметром функции является исходная кривая **curve**.

Выходным параметром функции является построенная кривая **result**.

Рассматриваемая функция строит NURBS-кривую, форма которой копирует форму исходной кривой. Для большинства типов кривых NURBS-копия по форме полностью совпадает с исходной кривой. В случае невозможности точно воспроизвести форму исходной кривой NURBS-копия аппроксимирует исходную кривую.

В тестовом приложении *test.exe* построение двумерной NURBS-кривой выполняется командой меню «Создать→Кривую в пространстве→По точкам→NURBS».

3.3.4 Построение правильного многоугольника

Функция

```
MbResultType RegularPolygon( const MbCartPoint &   centre,
                             const MbCartPoint &   point,
                                   size_t          vertexCount,
                                   bool            describe,
                                   MbCurve *&      result )
```

выполняет построение замкнутой двумерной ломаной линии, представляющей собой правильный многоугольник, вписанный в обозначенную окружность или описанный около окружности.

Входными параметрами функции являются:

- **centre** - центр окружности, описанной или вписанной в равносторонний многоугольник;
- **point** - точка на окружности;
- **vertexCount** - количество вершин многоугольника;
- **describe** - флаг, определяющий положение многоугольника относительно окружности.

Выходным параметром функции является построенная кривая **result**.

Функция возвращает код результата операции из перечисления :ref:`MbResultType <MbResultType>` (см. *mb_operation_result.h*).

Функция объявлена в файле *action_curve.h*.

Параметры **centre** и **point** определяют окружность, в которую будет вписан (**describe** = *false*) многоугольник с числом вершин *vertexCount* или вокруг которой будет описан (**describe** = *true*) многоугольник с числом вершин **vertexCount**. Параметр **vertexCount** определяет число вершин правильного многоугольника:

- при **vertexCount** == 0 строится окружность с центром **centre**, проходящая через **point**;
- при **vertexCount** == 1 строится отрезок с крайними точками **centre** и **point**;
- при **vertexCount** == 2 строится прямоугольник со сторонами, параллельными глобальным осям, и противоположными вершинами в **centre** и **point**, рис. 1:



Рис. 1

- при **vertexCount** >= 3 строится правильный многоугольник с заданным числом сторон.

Построенная кривая будет замкнутой ломаной линией, описанной в подразделе Двумерная ломаная линия MbPolyline. На рис. 2 приведён вписанный в окружность правильный многоугольник, а на рис. 3 приведён описанный вокруг окружности правильный многоугольник:

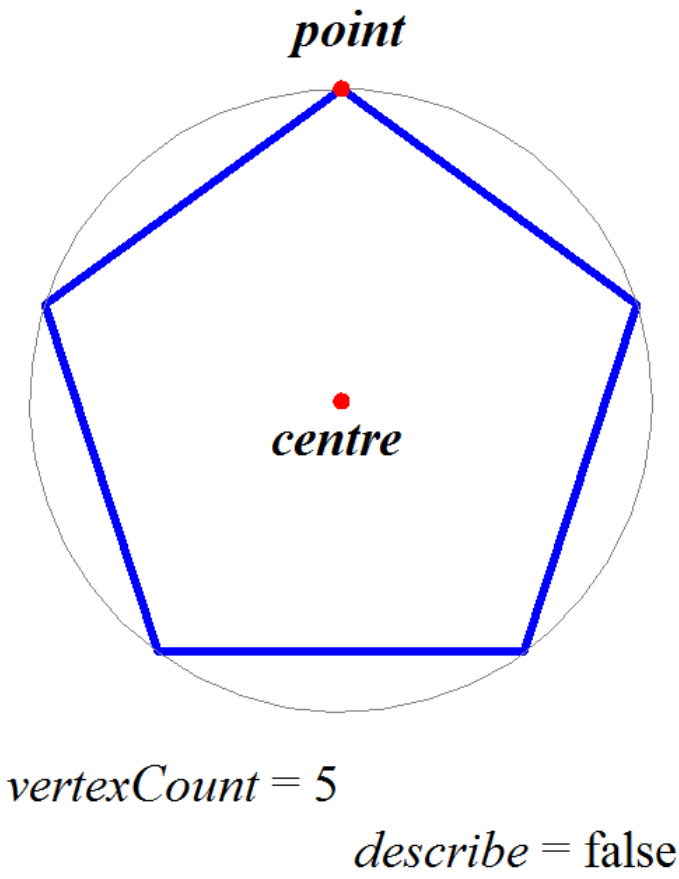


Рис. 2

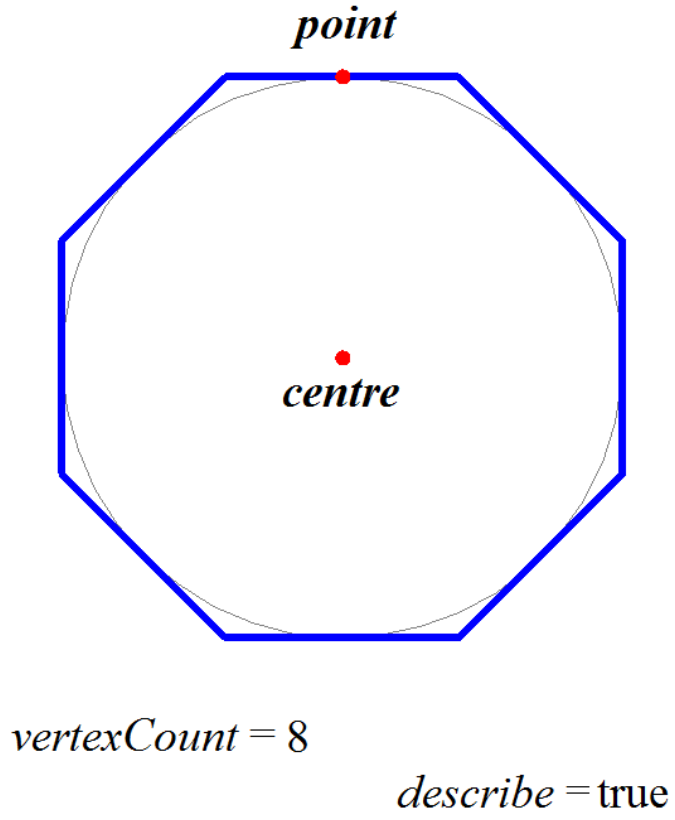


Рис. 3

В тестовом приложении *test.exe* построение выпуклой равносторонней двумерной ломаной выполняется командой меню «Создать→Кривую на плоскости→По точкам→Многоугольник».

3.3.5 Построение двумерной косинусоиды

Функция

```
MbResultType Cosinusoid( const MbCartPoint & point0,
                         const MbCartPoint & point1,
                               MbCartPoint & point2,
                               double        phase,
                               double        waveLength,
                               MbCurve *&    result );
```

выполняет построение косинусоиды (двумерной гармонической кривой).

Входными параметрами функции являются:

- **point0** — начало локальной системы координат;
- **point1** — точка на оси X локальной системы координат;
- **point2** — точка, определяющая ось Y локальной системы координат;
- **phase** — сдвиг фазы гармонической кривой (в радианах);

- **waveLength** — длина волны косинусоиды в единицах локальной системы координат.

Выходным параметром функции является указатель **result** на построенную двумерную кривую типа MbCurve, представляющую косинусоиду.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

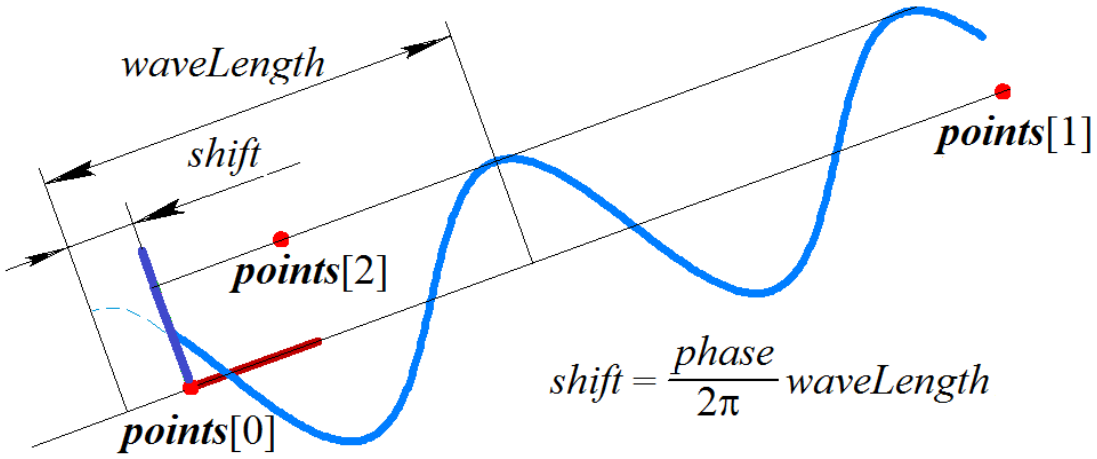
Точки **point0**, **point1** и **point2** определяют локальную систему координат косинусоиды. Начало локальной системы координат косинусоиды будет расположено в точке **point0**, ось **axisX** локальной системы координат будет направлена из точки **point0** в точку **point1**, ось **axisY** локальной системы координат будет ортогональна оси **axisX** и направлена из точки **point0** в сторону точки **point2**.

Точки **point0**, **point1** и **point2** не должны совпадать или лежать на одной прямой.

Двумерная косинусоида описана в параграфе Двумерная косинусоида MbCosinusoid.

Амплитуда косинусоиды равна длине проекции вектора (**point2** – **point0**) на ось **axisY**. Длина волны косинусоиды равна **waveLength**. Число волн косинусоиды определяется расстоянием между точками **point0** и **point1**.

Параметр **phase** определяет сдвиг фазы косинусоиды:



3.3.6 Построение двумерной составной кривой

Функция

```
MbResultType CreateContour( MbCurve  &   curve,
                            MbContour *& result )
```

выполняет построение составной кривой (контура) на базе одной исходной кривой.

Входным параметром функции является исходная кривая **curve**.

Выходным параметром функции является построенная составная кривая (контур) **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Функция создает составную кривую **result** на базе исходной кривой **curve**. Если исходная кривая **curve** является составной кривой, то в кривую **result** будут положены составляющие элементы исходной кривой. Двумерная составная кривая описана в параграфе *Двумерный контур MbContour*.

Функция

```
MbResultType AddCurveToContour( MbCurve   &  curve,
                                MbContour &  contour,
                                bool         toEnd,
                                double       epsilon = METRIC_REGION )
```

модифицирует составную кривую путём добавления кривой.

Входными параметрами функции являются:

- **curve** — добавляемая кривая;
- **contour** — модифицируемая составная кривая;
- **toEnd** — флаг места добавления кривой;
- **epsilon** — близость точек стыка при добавлении кривой.

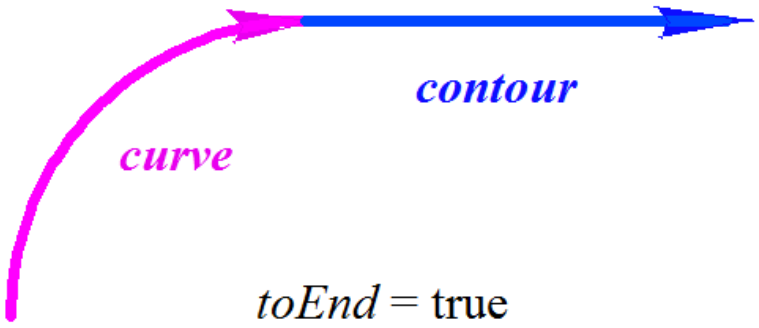
Выходным параметром функции является модифицируемая кривая **contour**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Функция модифицирует составную кривую **contour** путём добавления кривой **curve** в начало или конец составной кривой.

Если **toEnd** = *true*, то кривая **curve** будет добавлена в конец составной кривой **contour**, если **toEnd** = *false*, то кривая **curve** будет добавлена в начало составной кривой **contour**. Точки стыковки модифицируемой кривой и добавляемой кривой должны совпадать с точностью **epsilon**:



Если добавляемая кривая **curve** является составной кривой, то в кривую **contour** будут добавлены составляющие элементы добавляемой кривой.

3.3.7 Построение кривых пересечения поверхности и плоскости

Функция

```
void SurfaceSection( const MbSurface &          surface,
                     const MbPlacement3D &      place,
                           RPArray<MbCurve> &   result,
                           VERSION              version = Math::DefaultMathVersion() );
```

выполняет построение кривых пересечения поверхности с плоскостью.

Входными параметрами функции являются:

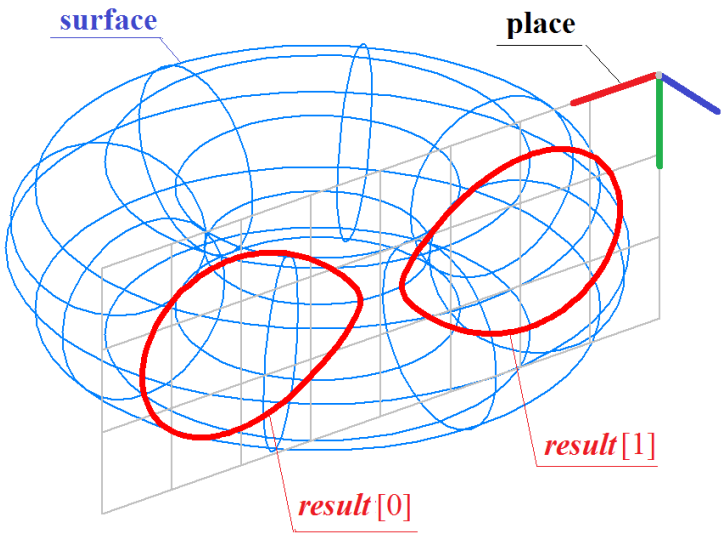
- **surface** — поверхность;
- **place** — локальная система координат плоскости;
- **version** — версия исполнения (опционально).

Выходным параметром функции является множество построенных двумерных кривых **result** на поверхности и двумерных кривых на плоскости.

Функция не имеет возвращаемого значения.

Функция объявлена в файле *action_curve.h*.

Функция строит пересечение поверхности **surface** с плоскостью XY локальной системы координат **place**. Кривые **result** строятся в плоскости XY локальной системы координат **place**. На рисунке приведен пример построения двумерных кривых пересечения поверхности тора и плоскости XY локальной системы координат:



В тестовом приложении *test.exe* построение кривых пересечения поверхности и плоскости выполняется командой меню «Создать→Кривую в пространстве→Пересечения→Двух поверхностей».

3.3.8 Построение двумерной кривой ребра грани

Функция

```
MbResultType FaceBoundSegment( const MbFace &     face,
                                     size_t       loopInd,
                                     size_t       edgeInd,
                               const MbSurface & surface,
                                     VERSION      version,
                                     MbCurve *&   result );
```

выполняет построение проекции ребра грани на поверхность.

Входными параметрами функции являются:

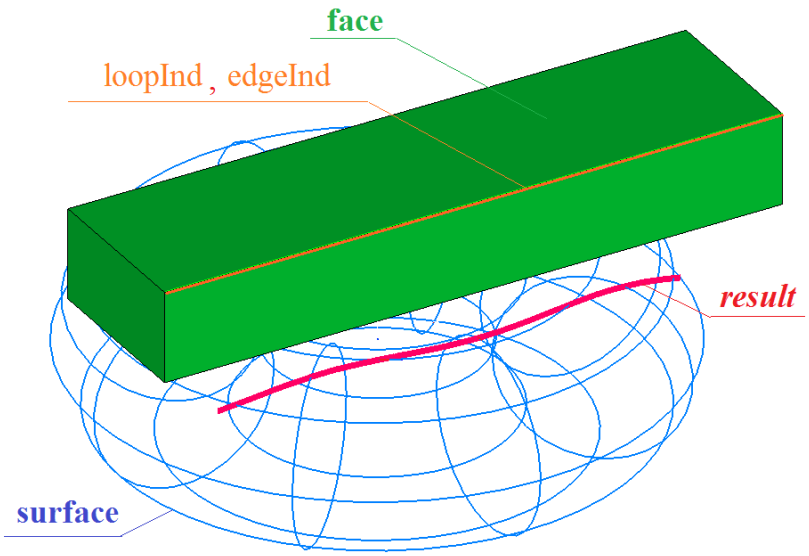
- **face** — грань;
- **loopInd** — индекс цикла в грани, в котором расположено проецируемое ребро;
- **edgeInd** — индекс проецируемого ребра в цикле;
- **surface** — поверхность проецирования;
- **version** — версия построения.

Выходным параметром функции является построенная кривая **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Функция строит проекцию ребра грани **face** на поверхность **surface**. Ребро в грани определено индексом цикла грани **loopInd** и индексом ребра в цикле **edgeInd**. Кривая **result** строится в пространстве параметров поверхности **surface**. На рисунке приведен пример построения проекции ребра грани на заданную поверхность.



В тестовом приложении *test.exe* построение кривых пересечения поверхности и плоскости выполняется командой меню «Создать→Кривую в пространстве→На базе тела→Проекции ребер грани».

3.3.9 Построение проекции кривой на поверхность

Функция

```
MbResultType SurfaceBoundContour( const MbSurface &   surface,
                                  const MbCurve3D &   spaceCurve,
                                        VERSION       version,
                                        MbContour *&  result );
```

выполняет построение двумерной кривой в пространстве параметров поверхности для пространственной кривой, лежащей на этой поверхности.

Входными параметрами функции являются:

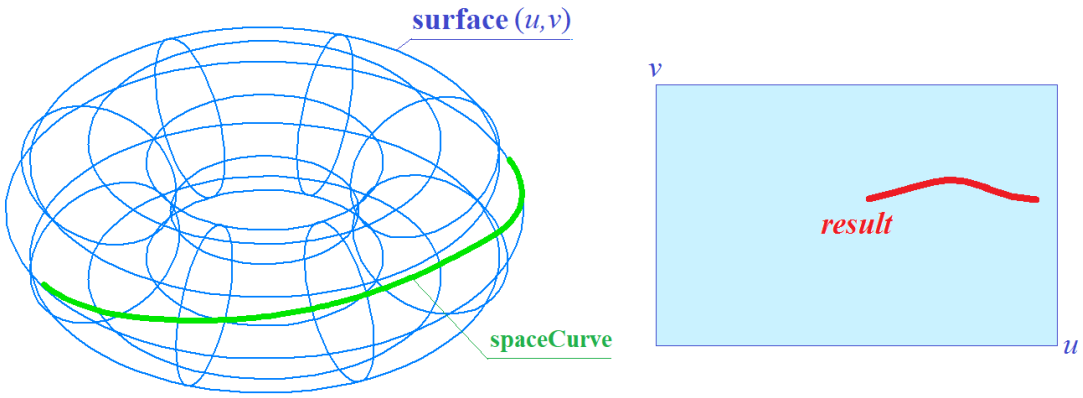
- **surface** — поверхность;
- **spaceCurve** — пространственная кривая;
- **version** — версия построения.

Выходным параметром функции является построенная составная кривая **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve.h*.

Пространственная кривая должна располагаться на поверхности **surface**. Кривая **result** строится в пространстве параметров поверхности **surface**. На рисунке приведен пример построения проекции пространственной кривой **spaceCurve** на заданную поверхность:



3.4 МЕТОДЫ ПОСТРОЕНИЯ КРИВЫХ

Кривые служат строительным материалом для построения поверхностей. По кривым выполняется стыковка поверхностей друг с другом. На базе кривых строятся рёбра, которые используются в твёрдотельной и каркасной моделях. В определённых случаях кривые выступают в роли опорных объектов для позиционирования других элементов геометрической модели.

Все кривые являются наследниками класса MbCurve3D и представлены в разделе *КРИВЫЕ*. Кривые могут быть построены непосредственным вызовом соответствующих конструкторов или методами, приведёнными в данном параграфе.

3.4.1 Построение прямой линии и отрезка

Функция

```
MbResultType Line( const MbCartPoint3D & point1,
                   const MbCartPoint3D & point2,
                         MbCurve3D *&    result );
```

выполняет построение прямой по двум несовпадающим точкам в пространстве.

Входными параметрами функции являются:

- **point1** — первая точка, по которой проходит прямая;
- **point2** — вторая точка, по которой проходит прямая.

Выходным параметром функции является построенная прямая **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

Параметр **point1** определяет начальную точку прямой, соответствующую нулевому параметру прямой. Вектор, начинающийся в точке **point1** и оканчивающийся в точке **point2**, определяет направление прямой, рис. 1. Производная прямой линии представляет собой вектор единичной длины.

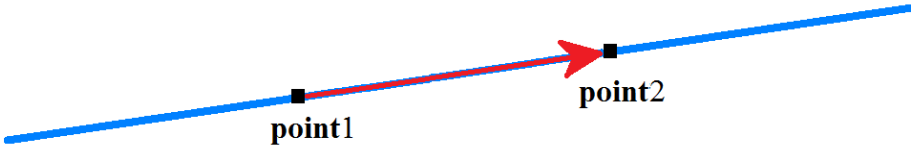


Рис. 1

Прямая линия описана в параграфе *Прямая линия MbLine3D*.

Функция

```
MbResultType Segment( const MbCartPoint3D & point1,
                      const MbCartPoint3D & point2,
                            MbCurve3D *&    result );
```

выполняет построение отрезка прямой по двум несовпадающим точкам в пространстве.

Входными параметрами функции являются:

- **point1** — начальная точка отрезка;
- **point2** — конечная точка отрезка.

Выходным параметром функции является построенный отрезок **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

Параметр **point1** определяет начальную точку отрезка, соответствующую нулевому параметру прямой. Параметр **point2** определяет конечную точку отрезка, соответствующую параметру прямой, равному единице, рис. 2:



Рис. 2

Отрезок прямой описан в подразделе *Отрезок прямой MbLineSegment3D*.

3.4.2 Построение окружности, эллипса и их дуг

Функция

```
MbResultType Arc( const MbCartPoint3D &          centre,
                  const SArray<MbCartPoint3D> & points,
                        bool                    curveClosed,
                        double                  angle,
                        double &                a,
                        double &                b,
                        MbCurve3D *&            result );
```

выполняет построение дуги эллипса, в частном случае функция выполняет построение дуги окружности.

Входными параметрами функции являются:

- **centre** — центр эллипса (окружности);
- **points** — множество точек дуги (контейнер **points** может содержать 0 или 2 элементов);
- **curveClosed** — флаг циклической замкнутости кривой;
- **angle** — угол раствора дуги эллипса (в радианах);
- **a** — длина большой полуоси эллипса (при непустом множестве **points** вычисляется);
- **b** — длина малой полуоси эллипса (при непустом множестве **points** вычисляется).

Выходным параметром функции является построенный эллипс (окружность) или его дуга **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

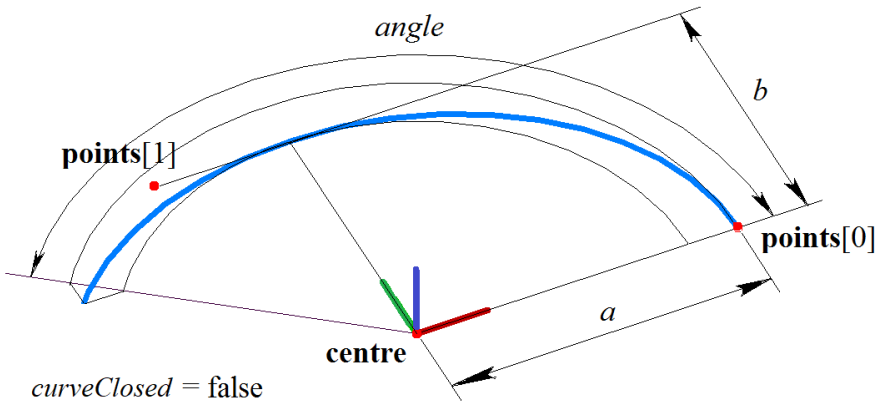
Кривая может быть построена как по заданным точкам, так и по скалярным параметрам.

Параметр **centre** определяет центральную точку эллипса.

Множество точек **points** может быть пустым, в этом случае должны быть ненулевыми длины полуосей эллипса **a** и **b**. Если множество точек **points** содержит два элемента, то точки **centre**, **points**[0], **points**[1] определяют плоскость расположения осей **axisX** и **axisY** локальной системы координат эллипса:

- ось **axisX** направлена из центра в точку **points**[0];
- ось **axisY** ортогональна оси **axisX** и направлена из центра в сторону точки **points**[1].

Расстояние между точками **centre** и **points**[0] определяет длину первой полуоси эллипса **a**, а расстояние между точкой **centre** и проекцией точки **points**[1] на ось **axisY** определяет длину второй полуоси эллипса **b**:



Дуга эллипса описана в параграфе *Дуга эллипса MbArc3D*.

Если множество точек **points** пусто, то длины полуосей эллипса **a** и **b** являются входными параметрами, а оси **axisX**, **axisY** и **axisZ** локальной системы координат эллипса совпадают с глобальными осями координат.

Параметр **curveClosed** определяет циклическую замкнутость кривой. Если **curveClosed** = *false*, то должен быть ненулевым параметр **angle**, который определяет угол раствора дуги эллипса в параметрических единицах.

В частном случае, когда **a** = **b**, рассматриваемая функция выполняет построение окружности (**curveClosed** = *true* или **angle** = 0) или дуги окружности (**curveClosed** = *false* и **angle** > 0 и **angle** < 2π).

3.4.3 Построение правильного многоугольника

Функция

```
MbResultType RegularPolygon( const MbCartPoint3D &  centre,
                             const MbCartPoint3D &  point,
                             const MbVector3D &     axisZ,
                                   size_t           vertexCount,
                                   bool             describe,
                                   MbCurve3D *&     result )
```

выполняет построение правильного многоугольника, вписанного в обозначенную окружность или описанного около окружности.

Входными параметрами функции являются:

- **centre** — центр окружности, описанной около равностороннего многоугольника или вписанной в равносторонний многоугольник;
- **point** — точка на окружности;
- **axisZ** — вектор, перпендикулярный плоскости окружности;
- **vertexCount** — количество вершин многоугольника;
- **describe** — флаг описанной или вписанной окружности (*true* — описанный многоугольник, *false* — вписанный многоугольник).

Выходным параметром функции является построенная ломаная кривая **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

Параметры **centre**, **point** и **axisZ** определяют окружность, в которую будет вписан (**describe** = *false*) многоугольник с числом вершин **vertexCount** или вокруг которой будет описан (**describe** = *true*) многоугольник с числом вершин **vertexCount**. Центр окружности будет расположен в точке **centre**, окружность пройдёт через точку **point**, ось окружности будет параллельна вектору **axisZ**.

Параметр **vertexCount** определяет число вершин правильного многоугольника:

- при **vertexCount** == 0 строится окружность с центром **centre**, проходящая через **point**;
- при **vertexCount** == 1 строится отрезок с крайними точками **centre** и **point**;
- при **vertexCount** == 2 строится прямоугольник со сторонами, параллельными глобальным осям и с противоположными вершинами в **centre** и **point**;
- при **vertexCount** >= 3 строится правильный многоугольник с заданным числом сторон, вписанный в окружность (**describe** == *false*) или описанный вокруг окружности (**describe** == *true*) с центром **centre**, проходящей через **point**.

Построенная кривая будет замкнутой ломаной линией, описанной в подразделе *Ломаная линия MbPolyline3D* (для **vertexCount** >= 3). На рис. 1 приведён вписанный в окружность правильный многоугольник, а на рис. 2 приведён описанный вокруг окружности правильный многоугольник:

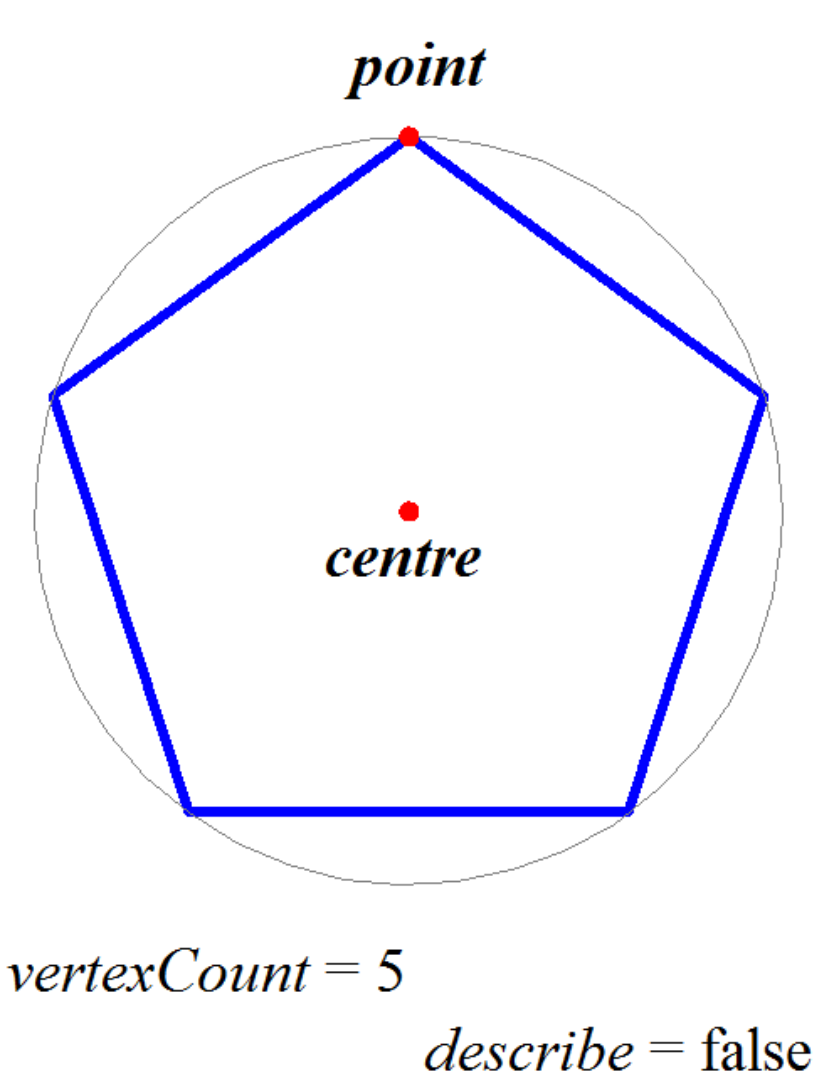


Рис. 1

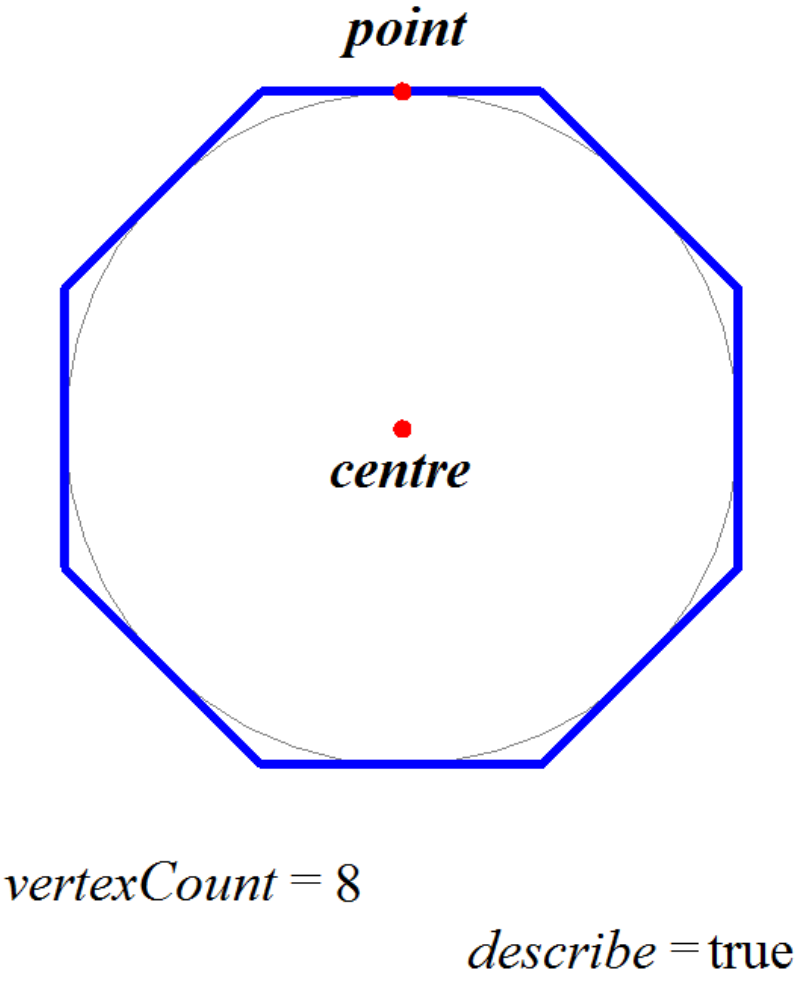


Рис. 2

3.4.4 Построение составной кривой

Функция

```
MbResultType CreateContour( MbCurve3D   &  curve,
                            MbContour3D *& result )
```

выполняет построение составной кривой на базе одной исходной кривой.

Входным параметром функции является исходная кривая **curve**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Выходным параметром функции является построенная составная кривая (контур) **result**.

При успешном выполнении в **result** возвращается указатель на новый объект MbContour3D; удаление объекта возлагается на вызывающий код.

Функция объявлена в файле *action_curve3d.h*.

Функция создает составную кривую **result** на базе исходной кривой **curve**. Если исходная кривая **curve** является составной кривой, то в кривую **result** будут положены составляющие элементы исходной кривой. Составная кривая описана в параграфе *Контур MbContour3D*.

Функция

```
MbResultType AddCurveToContour( MbCurve3D & curve,
                                MbCurve3D & contour,
                                bool        toEnd )
```

модифицирует составную кривую путём добавления кривой.

Входными параметрами функции являются:

- **curve** — добавляемая кривая;
- **contour** — модифицируемая составная кривая;
- **toEnd** — флаг места добавления кривой.

Выходным параметром функции является модифицируемая кривая **contour**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

Функция модифицирует составную кривую **contour** путём добавления кривой **curve** в начало или конец составной кривой. Если **toEnd** = *true*, то кривая **curve** будет добавлена в конец составной кривой **contour**, если **toEnd** = *false*, то кривая **curve** будет добавлена в начало составной кривой **contour**. Точки стыковки модифицируемой кривой и добавляемой кривой должны совпадать, рис. 1:

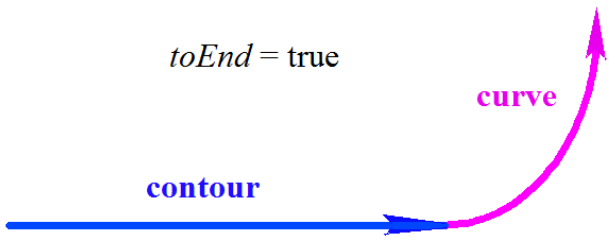


Рис. 1

Если добавляемая кривая **curve** является составной кривой, то в кривую **contour** будут добавлены составляющие элементы добавляемой кривой.

Функция

```
MbResultType CreateContours( RPArray<MbCurve3D>   & curves,
                             double                 metricEps,
                             RPArray<MbContour3D> & result,
                             bool                   onlySmoothConnected = false,
                             VERSION                version =␣
↪Math::DefaultMathVersion() )
```

выполняет построение составных кривых из множества исходных кривых (оригиналов кривых).

Входными параметрами функции являются:

- **curves** — множество исходных кривых;
- **metricEps** — радиус захвата для стыковки исходных кривых;
- **onlySmoothConnected** — флаг использования гладкой стыковки исходных кривых;
- **version** — версия исполнения.

Выходным параметром функции является множество построенных составных кривых **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

Функция на базе исходных кривых **curves** создает составные кривые и складывает их в множество **result**.

Если флаг **onlySmoothConnected** = *true*, то все построенные составные кривые будут гладкими, то есть в точках стыковки кривые будут иметь одинаково направленные касательные, но длина производной в точках стыковки может меняться скачком. Если какая-либо исходная кривая множества **curves** является составной кривой, то в составную кривую будут добавлены составляющие элементы такой исходной кривой.

Вырожденные кривые, кривые с нулевым указателем и дубликаты кривых удаляются из входного массива, если у них нет владельца.

Функция

```
MbResultType CreateContours( RPArray<MbCurve3D> & curves,
                             double               metricAcc,
                             bool                 onlySmoothConnected = false,
                             VERSION              version =␣
↪Math::DefaultMathVersion() )
```

выполняет построение контуров по набору кривых с удалением вырожденных кривых (на оригиналах кривых).

Входными параметрами функции являются:

- **curves** — множество исходных кривых (модифицируется);
- **metricAcc** — радиус захвата для стыковки исходных кривых;
- **onlySmoothConnected** — флаг использования в контуре только гладко стыкующихся сегментов;
- **version** — версия исполнения.

Выходным параметром функции является множество **curves**, которое после выполнения содержит построенные составные кривые.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_curve3d.h*.

Функция автоматически удаляет из массива **curves** вырожденные кривые (с нулевой длиной или с совпадающими начальной и конечной точками (для незамкнутых кривых)), кривые с нулевым указателем и дубликаты кривых, не имеющих владельца. Результирующие составные кривые складываются обратно в массив **curves**, заменяя исходные элементы.

3.4.5 Продление кривой

Функция

```
MbResultType ExtendCurve( const MbCurve3D &                    sourceCurve,
                          const MbCurveExtensionParameters3D & parameters,
                                SPtr<MbWireFrame> &            resFrame );
```

осуществляет продление заданной кривой в соответствии с заданными параметрами.

Параметрами функции являются:

- **sourceCurve** - исходная (продлеваемая) кривая;
- **parameters** - параметры продления (определение класса MbCurveExtensionParameters3D и соответствующих конструкторов см. в *op_curve_parameter.h*);
- **resFrame** - результат операции. Каркас, полностью покрывающий исходную кривую и дополнительно содержащий «продлевающий участок». Внутри содержит строитель MbExtendCurveCreator, объявленный в файле *cr_extending_curve.h*.

Функция объявлена в файле *action_curve3d.h*.

В тестовом приложении test.exe продление кривых выполняется командой меню «Модифицировать→Кривую в пространстве→Удлинить».

Рекомендуется оборачивать исходную кривую в каркас и вызывать функцию по продлению каркаса ExtendWireFrame(), детально описанную подразделе *Продление каркаса*.

3.4.6 Продление каркаса

Функция

```
MbResultType ExtendWireFrame(       c3d::WireFrameSPtr &           wireframe,
                                    MbeCopyMode                    sameEdges,
                              const MbCurveExtensionParameters3D & params,
                                    c3d::WireFrameSPtr &           result );
```

продлевает заданный каркас и тем самым обобщает операцию продления кривой.

Параметрами функции являются:

- **wireframe** - исходный (продлеваемый) каркас (рекомендуется передавать нормализованный каркас);
- **sameEdges** - режим копирования входного каркаса;
- **params** - параметры продления;
- **result** - результирующий каркас.

Функция объявлена в файле *action_wireframe.h*.

Класс MbCurveExtensionParameters3D наследует свойства класса MbCurveExtensionParameters. Объект **params** задаёт следующую группу параметров продления каркаса (кривой):

- параметры продления в начале - объект класса MbCurveExtensionEnds или одного из его наследников (MbCurveExtensionLinear, MbCurveExtensionCircular, MbCurveExtensionNatural);
- параметры продления в конце - объект класса MbCurveExtensionEnds или его наследников, задаётся независимо от параметров начала;
- флаг разрешения замкнутости продлённой кривой (**allowClosure**), определяющий, допускается ли замыкание кривой в процессе продления. В процессе продления (если длина продления будет достаточно большой) кривая может замкнуться. Это не всегда желательно, так как иногда между концами должен оставаться зазор. Для этого необходимо установить **allowClosure** = *false*. Величина зазора определяется внутренними алгоритмами и недоступна для изменения пользователем. В случае, когда продлённая кривая достигла своей предельной длины (кривая замкнулась или образовался максимально допустимый зазор между концами), выдаётся одно из следующих предупреждений: *rt_CurveClosedAtStart*, *rt_CurveClosedAtEnd* или *rt_CurveClosedBothSides* (см. таблицу).
- флаг продления вдоль поверхности (**alongSurface**), определяющий, должна ли поверхностная кривая (*MbSurfaceCurve*) оставаться на своей поверхности. Роль этого флага наглядно показана на рисунке 1, на котором приведена поверхность вращения и произвольная поверхностная кривая на ней (голубым цветом) с двумя вариантами продления «по дуге»:
  - если продлевать кривую как пространственную (**alongSurface** = *false*), то удлинённый участок (фиолетовый) будет представлять собой дугу окружности, но при этом может покинуть поверхность;
  - при продлении вдоль поверхности (**alongSurface** = *true*) кривая всегда остаётся на поверхности, но форма продлевающего сегмента (оранжевый) зависит от геометрии поверхности. Для неповерхностных кривых при **alongSurface** = *true* возвращается код ошибки *rt_IncorrectData*.

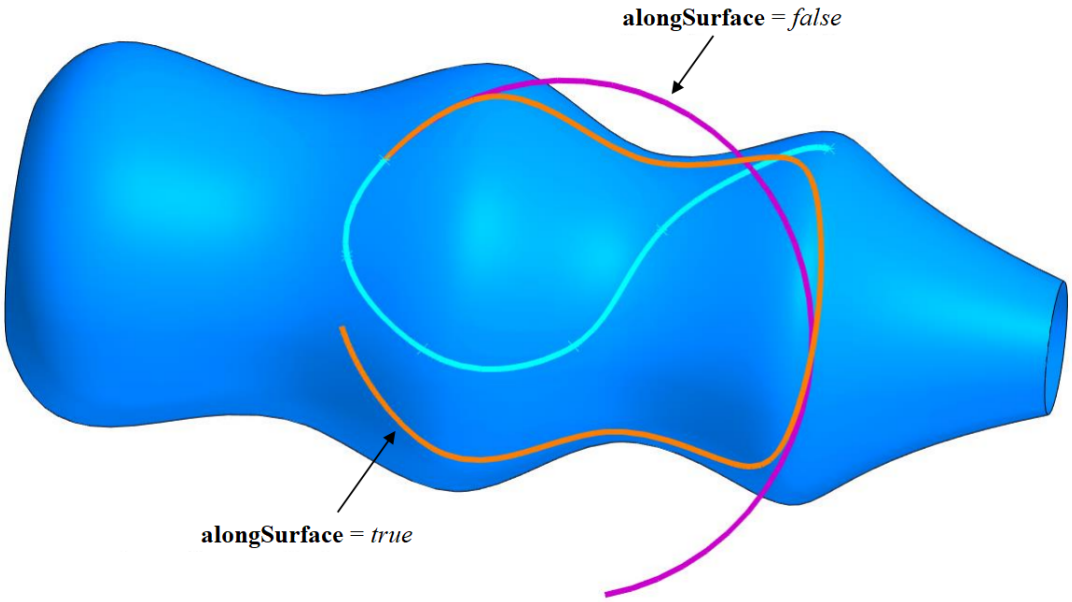


Рис. 1

- именователь операции (объект класса MbSNameMaker), который используется для именования созданного каркаса.

В большинстве случаев при продлении вдоль поверхности запрещён выбег продлённой кривой за границы поверхности. Если это произошло, операция возвращает ошибку *rt_BeyondLimitsExtension* (см. таблицу). Выбег за границы допускается для MbElementarySurface и периодических поверхностей для границ, соответствующих периодическому направлению.

Класс MbCurveExtensionEnds описывает параметры продления отдельного конца каркаса. В документации используются следующие понятия:

- **длина продления** — метрическая длина, на которую каркас продлевается от выбранного конца;
- **способ продления** — значение перечисления MbeCurveExtensionWays, определяющее метод продления (по касательной, по дуге или естественным продолжением исходного каркаса).

В геометрическом ядре реализованы три различных способа продления:

1. Продление «**по касательной**» или «**линейным**» способом (MbeCurveExtensionWays::cew_Linear) осуществляется восстановлением касательной в заданной точке кривой и откладыванием вдоль неё отрезка заданной длины. При таком способе продления обеспечивается гладкость не хуже G^1 в месте стыка с продлевающим сегментом. Параметры, отвечающие за данный способ продления, задаются через объекты класса MbCurveExtensionLinear.
2. Продление «**по дуге**» или «**круговым**» способом (MbeCurveExtensionWays::cew_Circular) выполняется построением соприкасающейся окружности на продлеваемом конце и взятием её дуги заданной длины. При таком способе продления обеспечивается гладкость не хуже G^2. Параметры, отвечающие за данный способ продления, задаются через объекты класса MbCurveExtensionCircular.
3. Продление естественным продолжением оригинальной кривой (формы, которую она имела непосредственно перед операцией продления) (MbeCurveExtensionWays::cew_CommonNatural). На выходе будет та же кривая, только имеющая большую длину. Но иногда продление осуществляется путём создания контура с пристыковкой к исходной кривой такого NURBS-сегмента, который обеспечивает гладкость не хуже G^3 в месте стыка. Параметры, отвечающие за данный способ продления, задаются через объекты класса MbCurveExtensionNatural.

Классы MbCurveExtensionLinear, MbCurveExtensionCircular и MbCurveExtensionNatural наследуются от MbCurveExtensionEnds и позволяют задать параметры, специфичные только для конкретного способа продления. Различие между перечисленными способами продления проиллюстрировано на рис. 2:

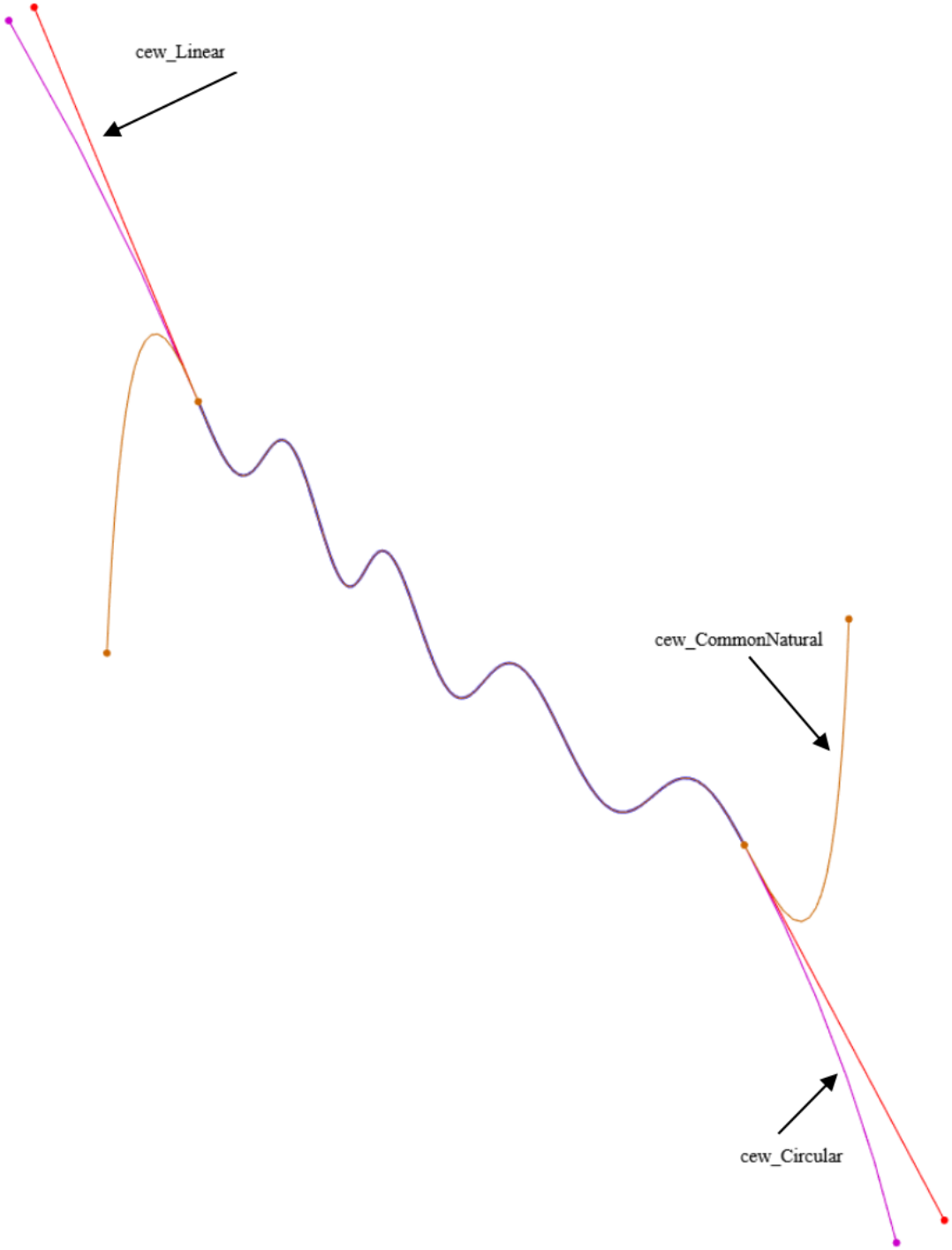


Рис. 2

Не существует никаких ограничений на комбинации опций продления отдельных концов. Продление каждого конца осуществляется независимо от другого. Например, на рис. 3 изображена дуга эллипса, продлённая в начале на 30 единиц «по касательной» и в конце «по дуге» на 60 единиц.

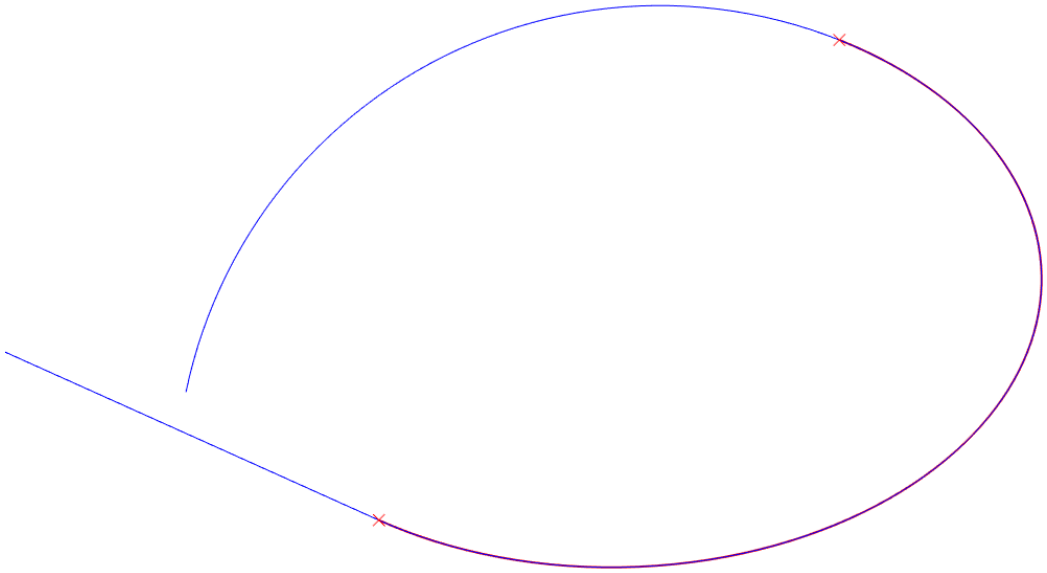


Рис. 3

Рассматриваемая функция ExtendWireFrame() возвращает код результата операции в соответствии с таблицей:

Коды результата операции

При критических ошибках указатель на результирующий каркас всегда нулевой; при некритических — каркас создаётся, но его геометрия может не удовлетворять требованиям.

Ниже приведена примерная последовательность действий для вызова алгоритма продления каркаса.

1. Подключить необходимые заголовочные файлы:

```
#include <op_wireframe_parameter.h>
#include <action_wireframe.h>
```

2. В качестве примера продлим каркас «по дуге» на 100 единиц в конце:

```
double extendLength = 100.0;   // Длина продления
```

3. Инициализируем параметры продления каждого отдельного конца: объекты класса MbCurveExtensionEnds. Если продление конкретного конца не требуется, необходимо параметры этого конца инициализировать конструктором по умолчанию.

```
// В начале не продлеваем. Поэтому используем конструктор по умолчанию.
MbCurveExtensionEnds     paramStart;

// В конце продлеваем по дуге
MbCurveExtensionCircular paramEnd( extendLength );
```

4. Далее инициализируем другие параметры продления и включаем в них созданные в п. 3 объекты с опциями продления концов.

```
bool allowClosure = false;  // Флаг разрешения замкнутости
bool alongSurface = false;  // Флаг продления вдоль поверхности
MbSNameMaker  name;                  // Именователь операции
MbCurveExtensionParameters3D opParameters( paramStart,
                                           paramEnd,
                                           allowClosure,
                                           alongSurface,
                                           name );
```

5. Вызвать API-функцию, осуществляющую продление:

```
SPtr<MbWireFrame> scrWireFrame;      // Заранее подготовленный ненулевой каркас

SPtr<MbWireFrame> result( nullptr ); // Результат продления
MbResultType  res = ::ExtendWireFrame( scrWireFrame,
                                       cm_Copy,
                                       opParameters,
                                       result );
```

Результирующий каркас будет содержать строитель MbExtendCurveCreator (наследник от MbCreator).

В тестовом приложении test.exe продление каркаса выполняется командой меню «Создать→Проволочный каркас→На базе проволочного каркаса→Продление каркаса».

3.5 МЕТОДЫ ПОСТРОЕНИЯ ПОВЕРХНОСТЕЙ

Поверхности описывают форму моделируемых объектов. На базе поверхностей строятся грани, из которых состоят твёрдые тела. В определённых случаях поверхности выступают в роли вспомогательных объектов для построения элементов геометрической модели.

Все поверхности являются наследниками класса MbSurface и представлены в главе *ПОВЕРХНОСТИ*. Поверхности могут быть построены непосредственным вызовом соответствующих конструкторов или функциями, приведёнными в данном разделе.

3.5.1 Построение элементарной поверхности

Функция

```
MbResultType ElementarySurface( const MbCartPoint3D & point0,
                                const MbCartPoint3D & point1,
                                const MbCartPoint3D & point2,
                                      MbeSpaceType    surfaceType,
                                      MbSurface *&    result )
```

выполняет построение элементарной поверхности.

Входными параметрами функции являются:

- **point0** – точка, определяющая начало локальной системы координат поверхности;
- **point1** – точка, определяющая направление осей **axisX**, **axisZ** (в зависимости от типа поверхности) локальной системы координат и высоту поверхности;

- **point2** – точка, определяющая направление оси **axisY** локальной системы;
- **surfaceType** – тип поверхности.

Выходным параметром функции является построенная поверхность **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_surface.h*.

Параметр **surfaceType** определяет тип создаваемой поверхности. Контрольные точки **point0**, **point1** и **point2** определяют локальную систему координат и размеры элементарной поверхности. Точка **point0** определяет начало локальной системы координат. В таблице приведены тип поверхности, ось локальной системы координат и величина, которую определяет точка **point1** для создания поверхности типа **surfaceType**:

surfaceType	**Тип поверхности**	Точка **point1** определяет
st_Plane	плоскость	**axisX**
st_CylinderSurface	цилиндрическая поверхность	**axisZ**, высоту
st_ConeSurface	коническая поверхность	**axisZ**, высоту
st_SphereSurface	сферическая поверхность	**axisZ**
st_TorusSurface	поверхность тора	**axisX**, больший радиус

Точка **point1** определяет высоту цилиндрической поверхности, высоту конической поверхности, больший радиус поверхности тора. Точка **point2** определяет радиус цилиндрической поверхности, радиус конической поверхности и угол конусности, радиус сферы, малый радиус поверхности тора.

При построении плоскости точки **point0**, **point1** и **point2** определяют локальную систему координат плоскости с началом в точке **point0**. Ось **axisX** локальной системы координат плоскости направлена из точки **point0** в точку **point1**, ось **axisY** локальной системы координат плоскости ортогональна оси **axisX** и направлена в сторону точки **point2**. Область определения первого параметра равна удвоенному расстоянию между точками **point0** и **point1**. Область определения второго параметра равна удвоенному расстоянию между точками **point0** и проекцией на ось **axisY** точки **point2**, рис. 1:

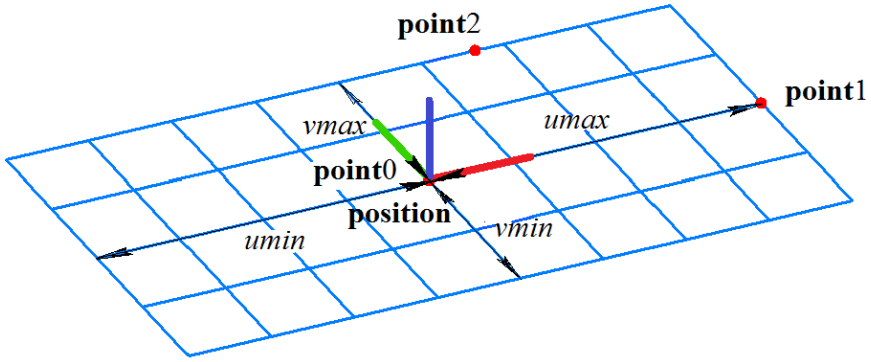


Рис. 1

При построении цилиндрической поверхности точка **point0** определяет центр нижнего основания цилиндра, в котором располагается начало локальной системы координат.

Точка **point1** определяет центр верхнего основания цилиндра. Ось **axisZ** локальной системы координат поверхности направлена из точки **point0** в точку **point1**. Точка **point2** вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат поверхности. Расстояние между точками **point0** и **point1** определяет высоту цилиндрической поверхности, расстояние от оси **axisZ** до точки **point2** определяет радиус цилиндрической поверхности, рис. 2:

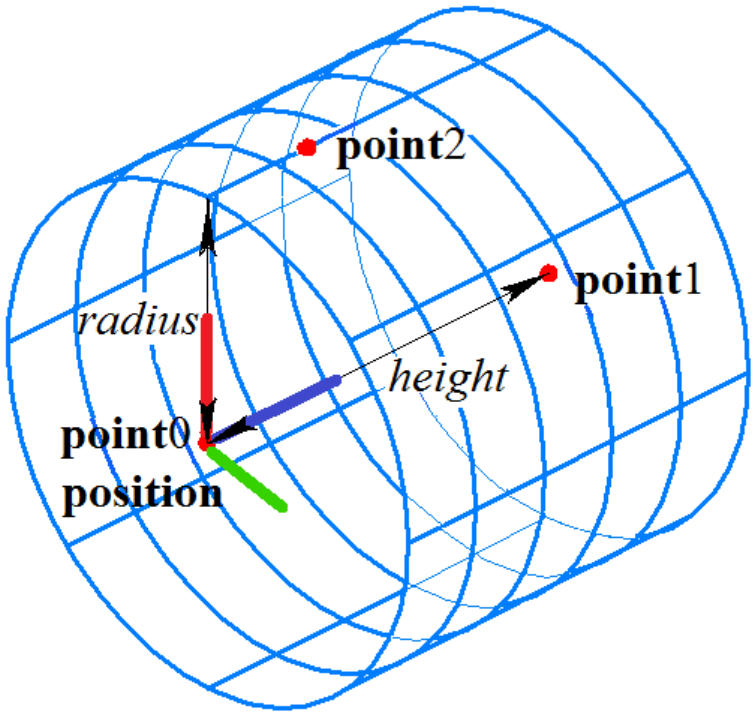


Рис. 2

Цилиндрическая поверхность описана в параграфе *Цилиндрическая поверхность MbCylinderSurface*.

При построении конической поверхности точка **point0** определяет вершину конуса, в которой располагается начало локальной системы координат. Точка **point1** определяет центр основания конуса и направление оси **axisZ** локальной системы координат конуса. Точка **point2** вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат поверхности. Расстояние между точками **point0** и **point1** определяет высоту конической поверхности. Точка **point2** определяет угол конусности, из условия, что точка **point2** лежит на конической поверхности, рис. 3:

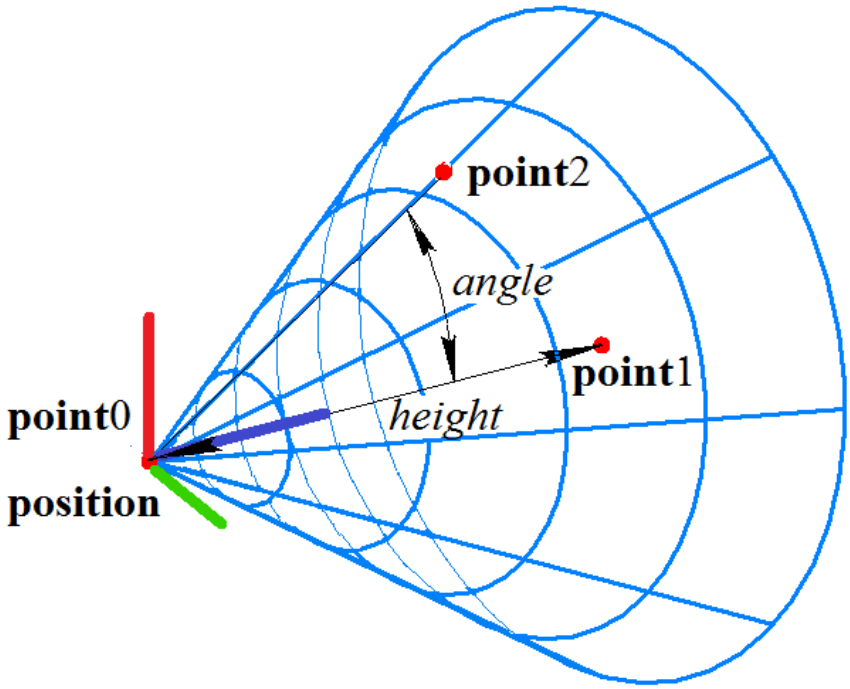


Рис. 3

Коническая поверхность описана в параграфе *Коническая поверхность MbConeSurface*.

При построении сферической поверхности точка **point0** определяет центр сферы, в котором располагается начало локальной системы координат. Ось **axisZ** локальной системы координат поверхности направлена из точки **point0** в точку **point1**. Точка **point2** вместе с предыдущими точками определяет плоскость расположения осей **axisX** и **axisZ** локальной системы координат поверхности. Расстояние между точками **point0** и **point2** определяет радиус сферической поверхности, рис. 4:

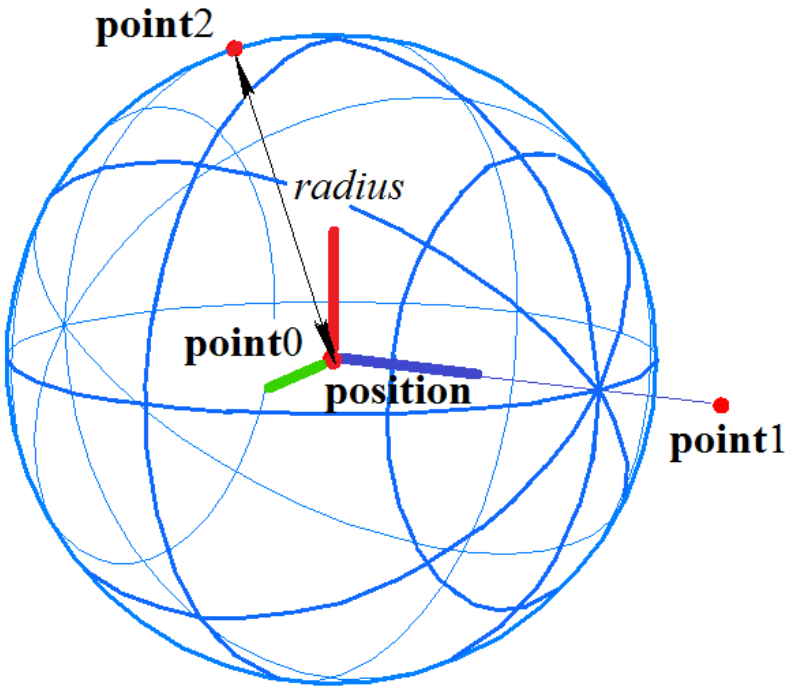


Рис. 4

Сферическая поверхность описана в параграфе *Сферическая поверхность MbSphereSurface*.

При построении поверхности тора точка **point0** определяет центр тора, в котором располагается начало локальной системы координат. Ось **axisX** локальной системы координат поверхности направлена из точки **point0** в точку **point1**. Точка **point2** вместе с предыдущими точками определяет плоскость, в которой располагаются оси **axisX** и **axisZ** локальной системы координат поверхности. Расстояние между точками **point0** и **point1** определяет больший радиус тора, расстояние между точками **point1** и **point2** определяет меньший радиус тора, рис. 5:

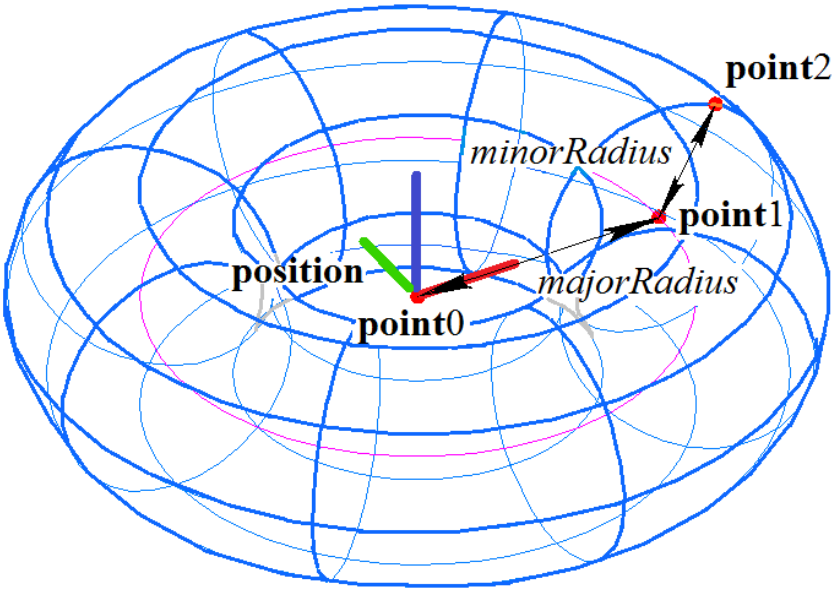


Рис. 5

Поверхность тора описана в параграфе *Поверхность тора MbTorusSurface*.

В тестовом приложении *test.exe* построение элементарной поверхности по заданным точкам выполняется командой меню «Создать → Поверхность → Элементарную».

3.5.2 Построение NURBS поверхности

Функция

```
MbResultType SplineSurface( const MbCartPoint3D & pUMinVMin,
                            const MbCartPoint3D & pUMaxVMin,
                            const MbCartPoint3D & pUMaxVMax,
                            const MbCartPoint3D & pUMinVMax,
                                  size_t          uCount,
                                  size_t          vCount,
                                  size_t          uDegree,
                                  size_t          vDegree,
                                  MbSurface *&    result );
```

выполняет построение NURBS-поверхности на основе четырёх угловых точек и равномерной сетки контрольных точек.

Входными параметрами функции являются:

- **pUMinVMin** – левая нижняя угловая точка поверхности;
- **pUMaxVMin** – правая нижняя угловая точка поверхности;
- **pUMaxVMax** – правая верхняя угловая точка поверхности;
- **pUMinVMax** – левая верхняя угловая точка поверхности;
- **uCount** – количество контрольных точек вдоль первого параметра (по горизонтали);
- **vCount** – количество контрольных точек вдоль второго параметра (по вертикали);

- **uDegree** – порядок B-сплайнов вдоль первого параметра;
- **vDegree** – порядок B-сплайнов вдоль второго параметра.

Выходным параметром функции является построенная сплайновая поверхность **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_surface.h*.

Рассматриваемая функция строит NURBS-поверхность, контрольные точки которой расположены в узлах четырехугольной таблицы с **uCount** колонками и **vCount** строками. Порядок *B*-сплайнов поверхности по первому параметру равен **uDegree**, порядок *B*-сплайнов поверхности по второму параметру равен **vDegree**. Контрольные точки поверхности рассчитываются из условия равномерного расположения и совпадения угловых точек с точками **pUMinVMin**, **pUMaxVMin**, **pUMaxVMax**, **pUMinVMax**, рис. 1:

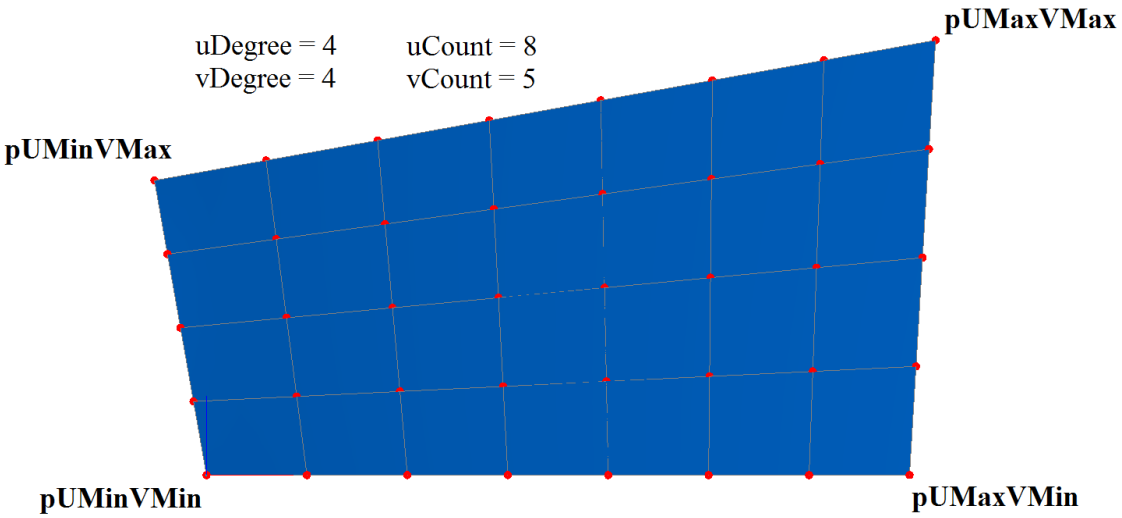


Рис. 1

Веса всех контрольных точек равны единице. Построенная поверхность предназначена для дальнейшей модификации. Для параметров функции должны выполняться равенства: **uCount** ≥ **uDegree** и **vCount** ≥ **vDegree**. NURBS-поверхность описана в параграфе *NURBS-поверхность MbSplineSurface*.

Функция

```
MbResultType SplineSurface( const SArray<MbCartPoint3D> & pointList,
                            const SArray<double> &        weightList,
                                  size_t                  uCount,
                                  size_t                  vCount,
                                  size_t                  uDegree,
                            const SArray<double> &        uKnotList,
                                  bool                    uClosed,
                                  size_t                  vDegree,
                            const SArray<double> &        vKnotList,
                                  bool                    vClosed,
                                  MbSurface *&            result );
```

выполняет построение NURBS поверхности по контрольным точкам и их весам.

Входными параметрами функции являются:

- **pointList** – множество контрольных точек, условно расположенных в виде **vCount** строк по **uCount** точек в каждой строке;
- **weightList** – множество весов контрольных точек, согласованное с множеством контрольных точек;
- **uCount** – количество контрольных точек вдоль первого параметра (в каждой строке);
- **vCount** – количество контрольных точек вдоль второго параметра (количество строк);
- **uDegree** – порядок B-сплайнов вдоль первого параметра;
- **uKnotList** – узловой вектор первого параметра;
- **uClosed** – циклическая замкнутость поверхности по первому параметру;
- **vDegree** – порядок B-сплайнов вдоль второго параметра;
- **vKnotList** – узловой вектор второго параметра;
- **vClosed** – циклическая замкнутость поверхности по второму параметру.

Выходным параметром функции является построенная поверхность **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_surface.h*.

Рассматриваемая функция строит NURBS-поверхность с контрольными точками **pointList**. Контрольные точки множества **pointList** будут разбиты на условные строки и столбцы. В каждой строке будет содержаться **uCount** контрольных точек из множества **pointList**, количество строк будет равно **vCount**. Таким образом множество **pointList** должно содержать **uCount**·**vCount** контрольных точек. Порядок *B*-сплайнов поверхности по первому параметру равен **uDegree**, порядок *B*-сплайнов поверхности по второму параметру равен **vDegree**. Для параметров функции должны выполняться равенства: **uCount** ≥ **uDegree** и **vCount** ≥ **vDegree**. NURBS-поверхность описана в параграфе *NURBS-поверхность MbSplineSurface*.

Множества **weightList**, **uKnotList**, **vKnotList** могут быть пустыми.

Если множество **weightList** пусто, то веса контрольных точек – единичные.

Если множества **uKnotList** или **vKnotList** пусты, то узловые векторы формируются автоматически.

Если множество **weightList** не пусто, то оно должно быть согласовано с множеством контрольных точек **pointList**.

Если множества **uKnotList** и **vKnotList** не пусты, то они должны содержать определенное количество элементов: для **uClosed** = *false* узловой вектор **uKnotList** должен содержать число элементов, равное **uCount** + **uDegree**; для **uClosed** = *true* узловой вектор **uKnotList** должен содержать число элементов, равное **uCount** + 2 · **uDegree** - 1; для **vClosed** = *false* узловой вектор **vKnotList** должен содержать число элементов, равное **vCount** + **vDegree**; для **vClosed** = *true* узловой вектор **vKnotList** должен содержать число элементов, равное **vCount** + 2 · **vDegree** - 1. На рис. 2 приведено расположение контрольных точек для построения сплайновой поверхности в форме тора:

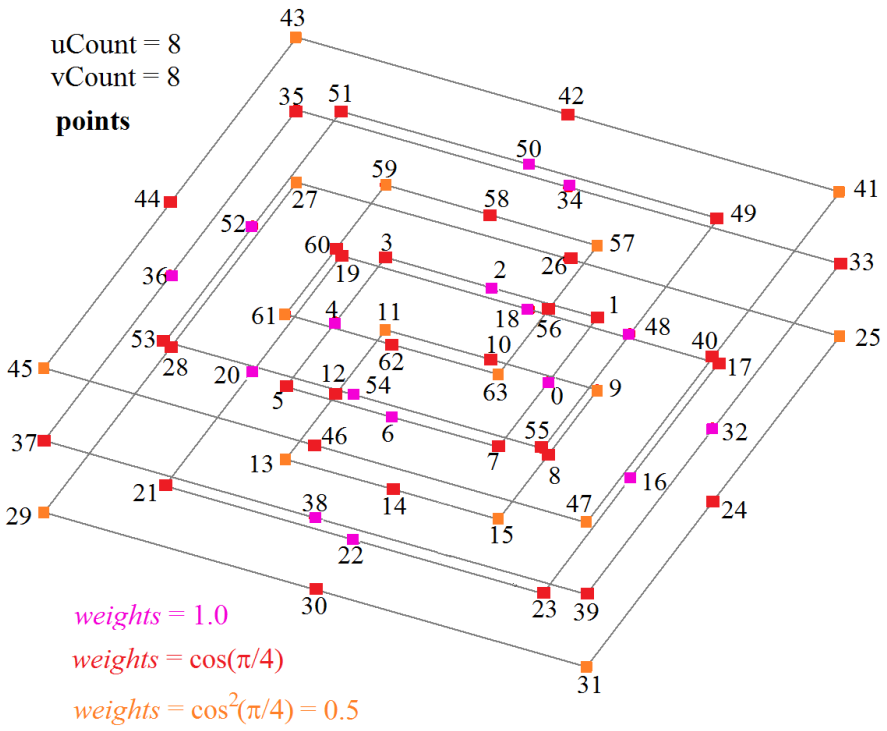


Рис. 2

На рис. 3 приведена сплайновая поверхность в форме тора, построенная на указанных выше контрольных точках:

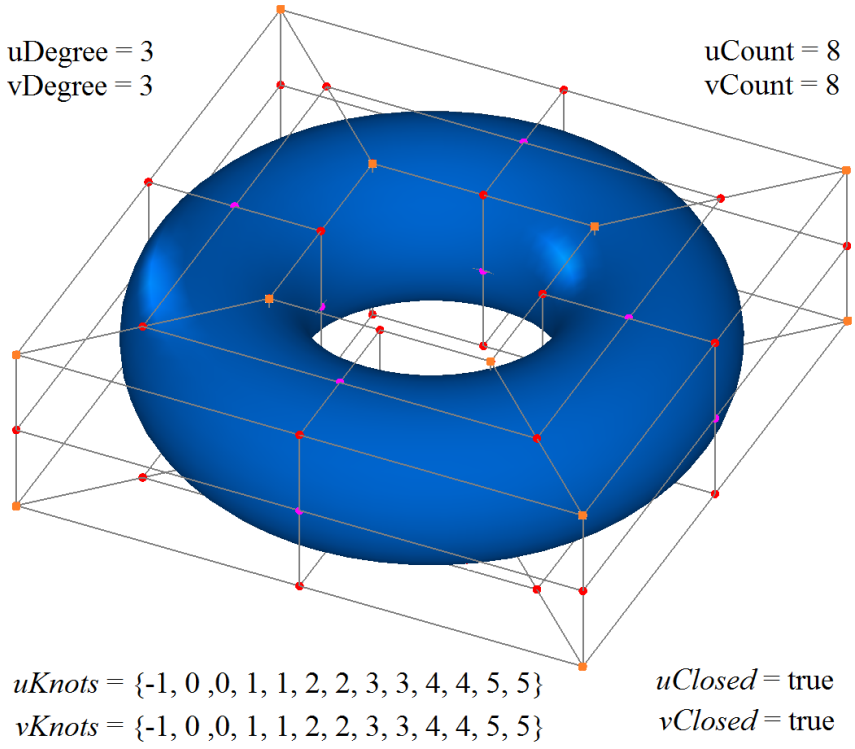


Рис. 3

Ниже приведен пример на языке программирования C++, реализующий построение сплайновой поверхности в форме тора рассматриваемой функцией (центр тора совпадает с началом глобальной системы координат, ось тора направлена по глобальной оси Z):

```
void AddTorusPoints( double r, double z,
SArray<MbCartPoint3D> & pointList, double w,
SArray<double> & weightList )
{
MbCartPoint3D p( r, 0.0, z );
MbVector3D toX( 1.0, 0.0, 0.0 ), toY( 0.0, 1.0, 0.0 );
double wI( w ), wA = w * ::cos( M_PI_4 );
pointList.Add( p ); weightList.Add( wI );
p.Add( toY, r ); pointList.Add( p ); weightList.Add( wA );
p.Add( toX,-r ); pointList.Add( p ); weightList.Add( wI );
p.Add( toX,-r ); pointList.Add( p ); weightList.Add( wA );
p.Add( toY,-r ); pointList.Add( p ); weightList.Add( wI );
p.Add( toY,-r ); pointList.Add( p ); weightList.Add( wA );
p.Add( toX, r ); pointList.Add( p ); weightList.Add( wI );
p.Add( toX, r ); pointList.Add( p ); weightList.Add( wA );
}

void GetTorusPoints( double majorR, double minorR,
SArray<MbCartPoint3D> & pointList,
SArray<double> & weightList )
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
{
double zLavel( 0.0 );
double w0( 1.0 ), wA = ::cos( M_PI_4 );
::AddTorusPoints( majorR - minorR, zLavel, pointList, w0, weightList );
::AddTorusPoints( majorR - minorR,-minorR, pointList, wA, weightList );
::AddTorusPoints( majorR ,-minorR, pointList, w0, weightList );
::AddTorusPoints( majorR + minorR,-minorR, pointList, wA, weightList );
::AddTorusPoints( majorR + minorR, zLavel, pointList, w0, weightList );
::AddTorusPoints( majorR + minorR, minorR, pointList, wA, weightList );
::AddTorusPoints( majorR , minorR, pointList, w0, weightList );
::AddTorusPoints( majorR - minorR, minorR, pointList, wA, weightList );
}

void GetTorusKnots( double tBeg, double tEnd,
SArray<double> & knots )
{
double dt = ( tEnd - tBeg ) / 4.0;
double t = tBeg - dt; knots.Add( t );
t = tBeg; knots.Add( t ); knots.Add( t );
t += dt; knots.Add( t ); knots.Add( t );
t += dt; knots.Add( t ); knots.Add( t );
t += dt; knots.Add( t ); knots.Add( t );
t = tEnd; knots.Add( t ); knots.Add( t );
t += dt; knots.Add( t ); knots.Add( t );
}

MbSurface * CreateTorusSurface( double majorR, double minorR )
{
MbSurface * result( NULL );
MbResultType res( rt_Error );
if ( majorR > METRIC_NEAR && majorR > minorR - EPSILON ) {
SArray<MbCartPoint3D> pointList( 64, 1 );
SArray<double> weightList( 64, 1 );
SArray<double> uKnotList( 13, 1 );
SArray<double> vKnotList( 13, 1 );
size_t uDegree( 3 ), vDegree( 3 );
size_t uCount( 8 ), vCount( 8 );
bool uClosed( true ), vClosed( true );
::GetTorusPoints( majorR, minorR, pointList, weightList );
::GetTorusKnots( 0.0, 4.0, uKnotList );
::GetTorusKnots( -2.0, 2.0, vKnotList );
res = ::SplineSurface( pointList, weightList, uCount, vCount, uDegree, uKnotList,
uClosed, vDegree, vKnotList, vClosed, result );
}
return ( res == rt_Success ) ? result : NULL;
}
```

Функция

```
MbResultType NurbsSurface( const MbSurface &   surface,
                           VERSION       version,
                           MbSurface *&  result );
```

выполняет построение NURBS-копии заданной поверхности.

Входными параметрами функции являются:

- **surface** – исходная поверхность,
- **version** – версия построения.

Выходным параметром функции является построенная поверхность **result**.

При успешном завершении работы функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType*.

Функция объявлена в файле *action_surface.h*.

Рассматриваемая функция строит NURBS-поверхность, форма которой копирует форму исходной поверхности. NURBS-поверхность описана в параграфе *NURBS-поверхность MbSplineSurface*. Для большинства типов поверхностей NURBS-копия по форме полностью совпадает с исходной поверхностью. В случае невозможности точно воспроизвести форму исходной поверхности NURBS-копия аппроксимирует исходную поверхность.

Если исходная поверхность является поверхностью с произвольными границами MbCurveBoundedSurface, описанной в параграфе *Поверхность с произвольными границами MbCurveBoundedSurface*, то рассматриваемая функция строит поверхность с произвольными границами, базовой поверхностью которой будет NURBS-копия базовой поверхности исходной поверхности.

В тестовом приложении test.exe построение NURBS поверхности выполняется командой меню «Создать → Поверхность → По точкам → NURBS».

3.5.3 Построение поверхности выдавливания

Функция

```
MbResultType ExtrusionSurface( const MbCurve3D &  curve,
                               const MbVector3D & direction,
                                     bool         simplify,
                                     MbSurface *& result )
```

выполняет построение поверхности выдавливания.

Входными параметрами функции являются:

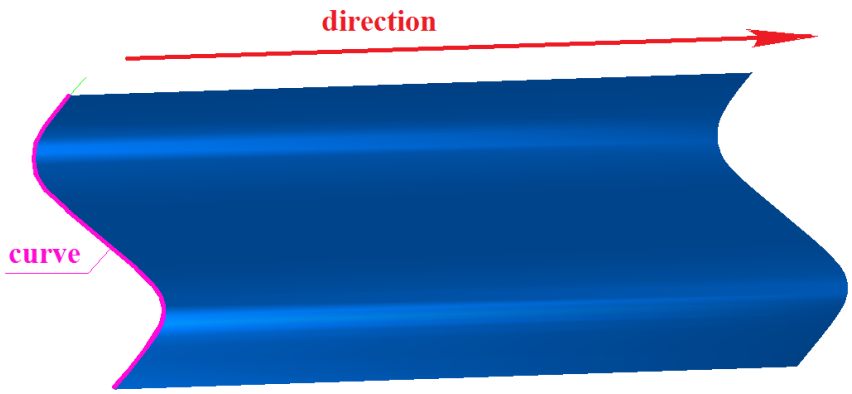
- **curve** - образующая кривая;
- **direction** - вектор выдавливания, определяющий направление и длину;
- **simplify** - флаг упрощения поверхности, если это возможно (например, сводить к более простому аналитическому типу, когда это допустимо).

Выходным параметром функции является построенная поверхность **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код результата из перечисления MbResultType (см. *mb_operation_result.h*).

Функция объявлена в файле *action_surface.h*.

Поверхность выдавливания относится к поверхностям движения, которые получают путём перемещения образующей кривой вдоль заданной траектории (направляющей). Для поверхности выдавливания траектория представляет собой прямолинейный отрезок, задаваемый вектором **direction**. Поверхность выдавливания представлена в подразделе *Поверхность выдавливания MbExtrusionSurface*. На рисунке приведена поверхность, построенная выдавливанием кривой вдоль заданного вектора:



В тестовом приложении *test.exe* построение поверхности выдавливания выполняется с помощью команды меню «Создать → Поверхность → На базе кривых → Выдавливания».

3.5.4 Построение поверхности вращения

Функция

```
MbResultType RevolutionSurface( const MbCurve3D &       curve,
                                const MbCartPoint3D &   origin,
                                const MbVector3D &      axis,
                                      double            angle,
                                      bool              simplify,
                                      MbSurface *&      result );
```

выполняет построение поверхности вращения.

Входными параметрами функции являются:

- **curve** – образующая кривая;
- **origin** – точка положения оси вращения;
- **axis** – направление оси вращения;
- **angle** – угол вращения;
- **simplify** - флаг упрощения результирующей поверхности (если возможно, создаётся элементарная поверхность, например, сфера, тор, цилиндр или конус вместо общей поверхности вращения).

Выходным параметром функции является построенная поверхность **result**.

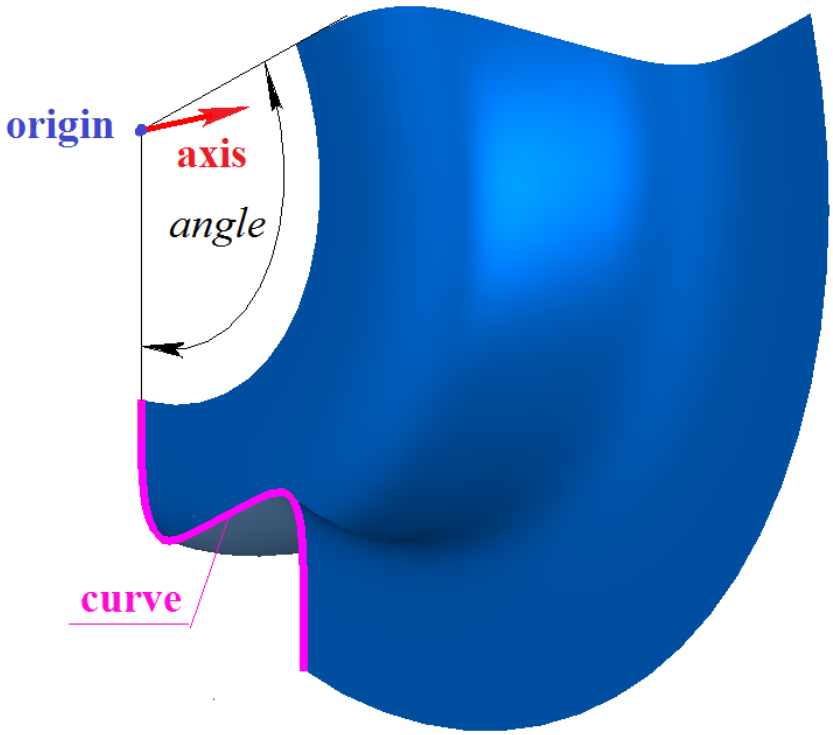
При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код результата из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_surface.h*.

Поверхность вращения получают путем движения образующей кривой вдоль дуги окружности, центр которой расположен на оси, проходящей через точку **origin** в направлении вектора **axis**, а угол раствора равен величине **angle**.

Если флаг **simplify** установлен в *true*, функция пытается создать упрощённую (элементарную) поверхность. Например, при вращении прямой линии вокруг оси может быть создана цилиндрическая, коническая или плоская поверхность; при вращении дуги окружности — сферическая или тороидальная поверхность.

Поверхность вращения представлена в параграфе *Поверхность вращения MbRevolutionSurface*. На рисунке приведена поверхность, построенная вращением кривой вокруг оси на заданный угол:



В тестовом приложении test.exe построение поверхности вращения выполняется с помощью команды меню

«Создать → Поверхность → На базе кривых → Вращения».

3.5.5 Построение поверхности по сети кривых

Функция

```
MbResultType MeshSurface( const RPArray<MbCurve3D> & uCurveList,
                          const RPArray<MbCurve3D> & vCurveList,
                                MbSurface *&         result );
```

выполняет построение поверхности по двум семействам кривых.

Входными параметрами функции являются:

- **uCurveList** – семейство кривых вдоль U-направления;
- **vCurveList** – семейство кривых вдоль V-направления.

Выходным параметром функции является построенная поверхность **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код результата из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_surface.h*.

Для построения поверхности каждая кривая множества **uCurveList** должна пересекаться со всеми кривыми множества **vCurveList** и, соответственно, каждая кривая множества **vCurveList** должна пересекаться со всеми кривыми множества **uCurveList**. Построенная поверхность будет проходить по кривым множества **uCurveList** и множества **vCurveList**.

Поверхность, построенная по сети кривых, представлена в параграфе *Поверхность на сети кривых MbMeshSurface*. На рис. 1 приведены два множества кривых, а на рис. 2 приведена построенная поверхность, построенная на этих кривых:

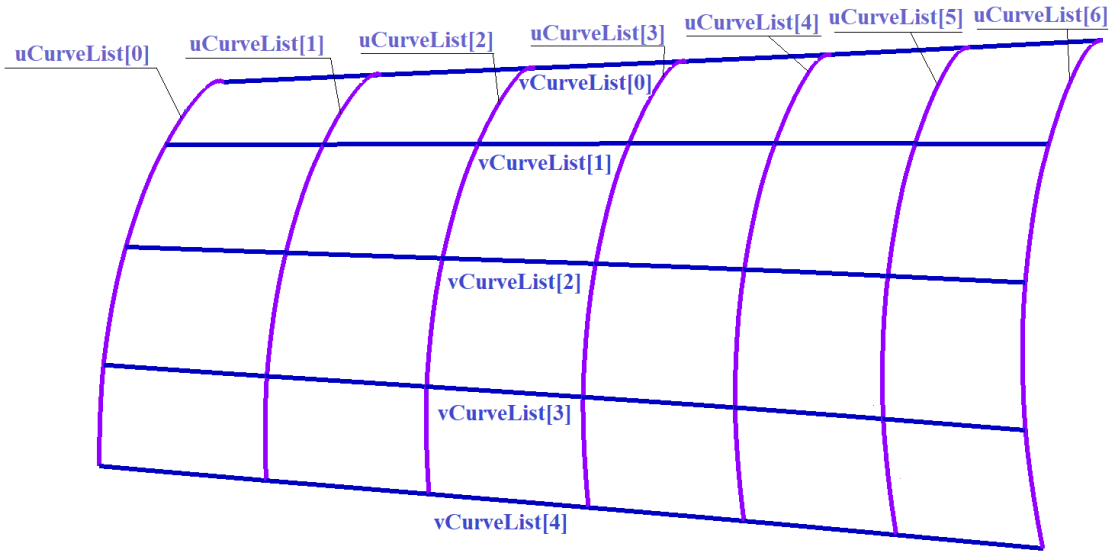


Рис. 1

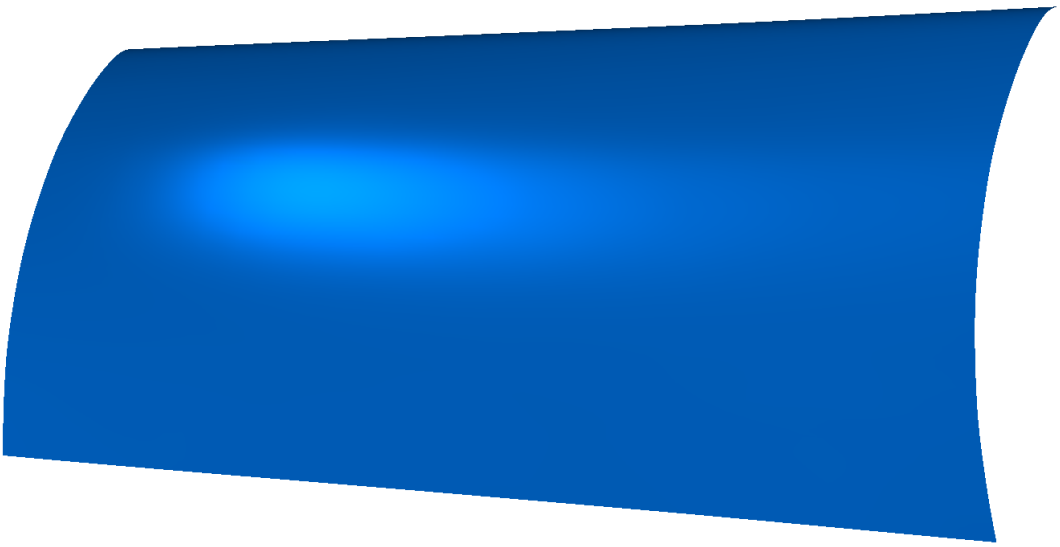


Рис. 2

В тестовом приложении test.exe построение поверхности по сети кривых выполняется с помощью команды меню «Создать → Поверхность → На базе кривых → На сетке кривых».

3.5.6 Построение поверхности с произвольными границами

Функция

```
MbResultType BoundedSurface(      MbSurface &          surface,
                            const RPArray<MbCurve> &  boundList,
                                  MbSurface *&        result );
```

выполняет построение поверхности с заданной границей.

Входными параметрами функции являются:

- **surface** – исходная поверхность;
- **boundList** – множество двумерных замкнутых кривых в пространстве параметров (плоскость (u,v)).

Выходным параметром функции является построенная поверхность **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код результата из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_surface.h*.

По умолчанию поверхности имеют прямоугольную область определения параметров. Исходная область определения параметров может быть изменена путем описания границы области двумерными замкнутыми кривыми **boundList**. Если кривых множества **boundList** больше одной, то первая кривая множества **boundList** должна описывать внешнюю границу, а остальные кривые должны лежать внутри первой кривой множества. Кривые множества **boundList** могут выходить за пределы исходной области определения параметров, при условии, что там отсутствуют особые точки (полюса, разрывы параметризации). Если множество границ пусто, то будет построена замкнутая составная кривая по исходной границе области определения параметров поверхности **surface**.

Поверхность с произвольными границами представлена в параграфе *Поверхность с произвольными границами MbCurveBoundedSurface*. На рис. 1 приведена поверхность и две замкнутые кривые на ней, а на рис. 2 приведен результат построения поверхности с границами в форме этих кривых:

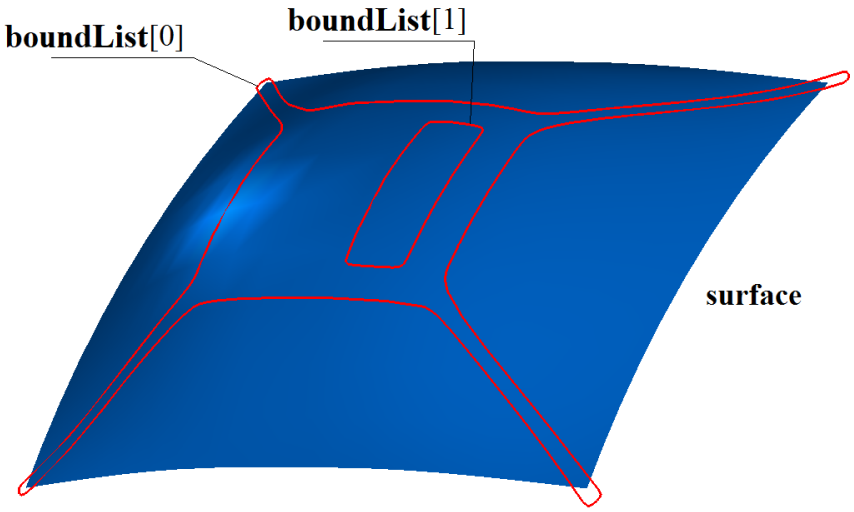


Рис. 1

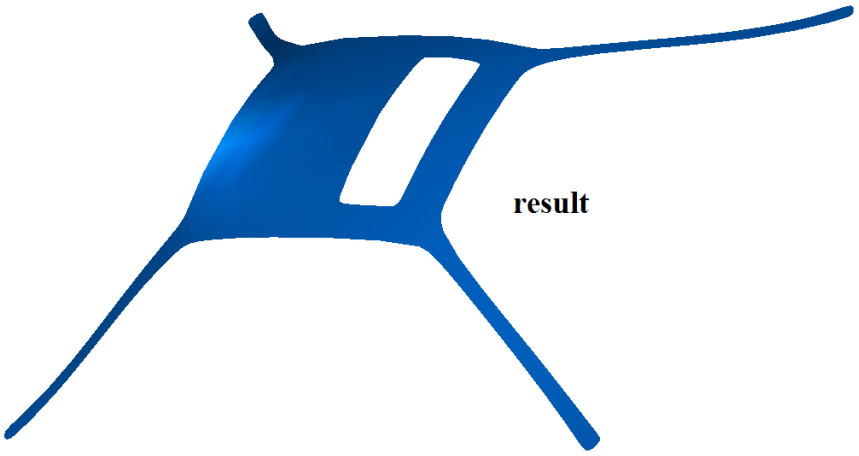


Рис. 2

3.6 МЕТОДЫ ПРЯМОГО МОДЕЛИРОВАНИЯ

Прямое моделирование позволяет модифицировать геометрическую модель путём изменения составляющих её элементов. Геометрическая модель может иметь любую степень готовности: она может представлять собой заготовку или быть готовой моделью. Методами прямого моделирования могут модифицироваться как все элементы геометрической модели, так и отдельная группа элементов. Например, группа граней тела может быть перемещена относительно остальных граней тела, заменена эквидистантными гранями или деформируемыми гранями. Из указанной группы граней может быть образовано новое тело. У геометрической модели могут быть удалены заданные скругления или характерные особенности, такие как отверстия или выступы.

3.6.1 Построение модифицированного тела

Функция

```
MbResultType ModifiedSolid(       MbSolid &                 solid,
                                  MbeCopyMode               sameShell,
                            const MbModifiedSolidParams &   params,
                                  MbModifiedSolidResults &  results );
```

В зависимости от параметров модификации MbModifiedSolidParams функция выполняет одно из следующих действий:

1. Удаление из тела выбранных граней без прилегающих скруглений (*dmm_RemoveFaces*).

   Удаление грани вместе с прилегающими скруглениями (*dmm_RemoveFacesWithAdjacentFillets*).

   Удаление замкнутых цилиндрических, конических, тороидальных, сферических граней, граней вращения и выдавливания радиуса меньше заданного без прилегающих скруглений (*dmm_RemoveBelowRadius*).

   Удаление замкнутых цилиндрических, сферических граней, граней вращения и выдавливания радиуса меньше заданного вместе с прилегающими скруглениями (*dmm_RemoveBelowRadiusWithAdjacentFillets*).

2. Создание тела из выбранных граней с окружением (*dmm_Create*).
3. Перемещение выбранных граней с окружением относительно оставшихся граней тела (*dmm_MoveFacesByVector*) или без адаптации прилегающих скруглений (*dmm_MoveFacesByVectorNoAdaptation*). Направление и величину перемещения определяет вектор **direction**.
4. Замена выбранных граней тела эквидистантными гранями (перемещение по нормали, изменение радиуса) (*dmm_OffsetFaces*). Расстояние смещения определяется параметром **value**.
5. Изменение радиуса выбранных граней скругления на заданную величину (*dmm_FilletsRadiusChange*). Величина изменения радиуса определяется параметром **value**.
6. Замена выбранных граней тела деформируемыми гранями (превращение в NURBS) для редактирования (*dmm_ConvertFacesToNurbs*).
7. Удаление выбранных граней скругления тела (*dmm_PurifyFillets*).

   Удаление также всех гладко прилегающих скруглений (*dmm_PurifyFilletsChains*).

   Удаление всех скруглений тела, меньших определённого радиуса (*dmm_PurifyFilletsBelowRadius*).
8. Слияние вершин рёбер и удаление рёбер (*dmm_DeleteEdges*).
9. Замена гладко стыкующихся граней одной гранью (*dmm_UniteFaces*).
10. Поворот выбранных граней с адаптацией прилегающих скруглений (*dmm_RotateFaces*) или без (*dmm_RotateFacesNoFilletAdaptation*). Угол поворота определяется параметром **angle**, ось поворота определяется осью **axis**.
11. Замена выбранных граней тела (*dmm_ReplaceFaces*).

12. Установка радиусов выбранных граней скругления для цепочки гладко стыкующихся скруглений (*dmm_FilletsRadiusSet*) и установка радиусов выбранных граней скругления (*dmm_FilletsRadiusSetSingle*). Новый радиус определяется параметром **value**.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – вариант копирования исходного тела;
- **params** – параметры модификации.

Выходным параметром функции является структура результатов модификации **results**, содержащая модифицированное тело и дополнительную информацию об операции.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Возможные коды ошибок:

rt_Success – ошибок нет;

rt_Error – неизвестная ошибка;

rt_IncorrectData – некорректные данные;

rt_ParameterError – ошибочный параметр.

Функция объявлена в файле *action_direct.h*.

При создании объекта **params** с параметрами модификации используется один из следующих конструкторов (см. *op_direct_mod_parameter.h*).

Конструкторы без набора граней

- **type** – тип модификации. Для данного конструктора доступны: *dmm_OffsetFaces*, *dmm_FilletsRadiusChange*;
- **value** – величина сдвига для *dmm_OffsetFaces* или величина изменения радиуса для *dmm_FilletsRadiusChange*;
- **names** – именователь операции.

```
MbModifiedSolidParams(       MbeModifyingMode type,
                       const MbSNameMaker &   names,
                             double           value );
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_RotateFaces*, *dmm_RotateFacesNoFilletAdaptation*;
- **axis** – ось вращения;
- **angle** – угол вращения;
- **names** – именователь операции.

```
MbModifiedSolidParams(       MbeModifyingMode  type,
                       const MbSNameMaker &    names,
                       const MbAxis3D &        axis,
                             double            angle );
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_MoveFacesByVector*, *dmm_MoveFacesByVectorNoAdaptation*;
- **direction** – направление перемещения;
- **names** – именователь операции.

```
MbModifiedSolidParams(       MbeModifyingMode  type,
                       const MbSNameMaker &    names,
                       const MbVector3D &      direction );
```

Конструкторы по набору граней

- **modType** – тип модификации. Для данного конструктора доступны: *dmm_UniteFaces*, *dmm_Create*, *dmm_ConvertFacesToNurbs*, *dmm_PurifyFillets*;
- **names** – именователь операции;
- **faces** – список граней для модификации.

```
MbModifiedSolidParams(       MbeModifyingMode   modType,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces );
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_OffsetFaces*, *dmm_FilletsRadiusChange*, *dmm_RemoveBelowRadius*, *dmm_RemoveBelowRadiusWithAdjacentFillets*, *dmm_PurifyFilletsBelowRadius*;
- **names** – именователь операции;
- **faces** – список граней для модификации;
- **value** – величина сдвига для *dmm_OffsetFaces*, величина изменения радиуса для *dmm_FilletsRadiusChange*, величина максимального радиуса для *dmm_Remove* и *dmm_Purify*.

```
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                             double             value )
: MbModifiedSolidParams( type, names, value )
{
  AddFaces( faces );
}
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_RotateFaces*, *dmm_RotateFacesNoFilletAdaptation*;
- **names** – именователь операции;
- **faces** – список граней для модификации;
- **axis** – ось вращения;
- **angle** – угол вращения.

```
MbModifiedSolidParams(       MbeModifyingMode    type,
                       const MbSNameMaker &      names,
                       const c3d::ItemIndices &  faces,
                       const MbAxis3D &          axis,
                             double              angle )
: MbModifiedSolidParams( type, names, axis, angle )
{
  AddFaces( faces );
}
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_MoveFacesByVector*;
- **names** – именователь операции;
- **faces** – список граней для модификации;
- **direction** – направление сдвига.

```
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                       const MbVector3D &       direction )
: MbModifiedSolidParams( type, names, direction )
{
  AddFaces( faces );
}
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_ConvertFacesToNurbs*;
- **names** – именователь операции;
- **faces** – список граней для модификации;
- **surfaces** – поверхности замены (может быть пустым массивом).

```
template<class SurfArray>
MbModifiedSolidParams(       MbeModifyingMode   type,
                       const MbSNameMaker &     names,
                       const c3d::ItemIndices & faces,
                       const SurfArray &        surfaces )
: MbModifiedSolidParams( type, names )
{
  AddFaces( faces );
  AddSurfaces( surfaces );
}
```

- **type** – тип модификации. Для данного конструктора доступны: *dmm_ReplaceFaces*;
- **names** – именователь операции;
- **replacedFaces** – список заменяемых граней;

- **replacementSurfaces** – список заменяющих поверхностей;
- **surfaceParts** – список номеров частей поверхностей, которые будут использоваться при нескольких возможных вариантах замены;
- **sameSurfaces** – использовать исходные поверхности (true) или их копии (false);
- **offsets** – список сдвигов от заменяющих поверхностей.

```
template<class SurfArray> MATH_FUNC_EX
MbModifiedSolidParams(       MbeModifyingMode     type,
                       const MbSNameMaker &       names,
                       const c3d::ItemIndices &   replacedFaces,
                       const SurfArray &          replacementSurfaces,
                             bool                 sameSurfaces,
                       const c3d::NumbersVector & surfaceParts,
                       const c3d::DoubleVector &  offsets );
```

- **modType** – тип модификации. Для данного конструктора доступны: *dmm_DeleteEdges*;
- **names** – именователь операции;
- **edges** – список рёбер для модификации;
- **beginOrEnd** – флаг удаления начала или конца соответствующего ребра.

```
MbModifiedSolidParams(       MbeModifyingMode         modType,
                       const MbSNameMaker &           names,
                       const c3d::EdgeFacesIndices & edges,
                       const c3d::BoolVector &       beginOrEnd )
: MbModifiedSolidParams( modType, names )
{
  AddEdges( edges, beginOrEnd );
}
```

Параметр **solid** содержит исходное тело, которое подлежит обработке.

Параметр **sameShell** управляет передачей данных от исходного тела **solid** к построенному телу **results** и может принимать одно из четырёх значений:

- *cm_Same* = 0 (тело не копируется);
- *cm_KeepHistory* (исходное тело и его копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходное тело и его копия имеют общие базовые поверхности);
- *cm_Copy* (исходное тело и его копия не имеют общих данных).

Перечисление MbeCopyMode подробно описано в подразделе *Копирование множества граней MbFaceShell*.

Параметр **params** описывает параметры модификации, которые определяются типом модификации и дополнительными параметрами в зависимости от используемого конструктора.

Тип модификации задаётся параметром **_modificationType**, который может принимать одно из значений перечисления MbeModifyingMode (см. *op_direct_mod_parameter.h*).

При типах модификации *dmm_RemoveFaces* и *dmm_RemoveFacesWithAdjacentFillets* рассматриваемая функция удаляет грани **faces** из копии тела **solid** и, используя окружение, обрабатывает место удалённых граней, сохраняя замкнутость тела. Если грани **faces** связаны с остальными гранями скруглениями, то в режиме *dmm_RemoveFacesWithAdjacentFillets* прилегающие грани скругления автоматически добавляются к множеству удаляемых граней.

При типах модификации *dmm_RemoveBelowRadius* и *dmm_RemoveBelowRadiusWithAdjacentFillets* функция удаляет замкнутые цилиндрические, конические, тороидальные, сферические грани, а также грани вращения и выдавливания радиуса меньше или равного заданному значению радиуса. В режиме *dmm_RemoveBelowRadiusWithAdjacentFillets* эти грани удаляются вместе с прилегающими скруглениями. Такая настройка позволяет автоматически убирать мелкие сквозные или глухие отверстия без явного перечисления всех соответствующих граней.

На рис. 1 приведено исходное тело, а на рис. 2 — построенное тело, полученное путём удаления указанных граней исходного тела:

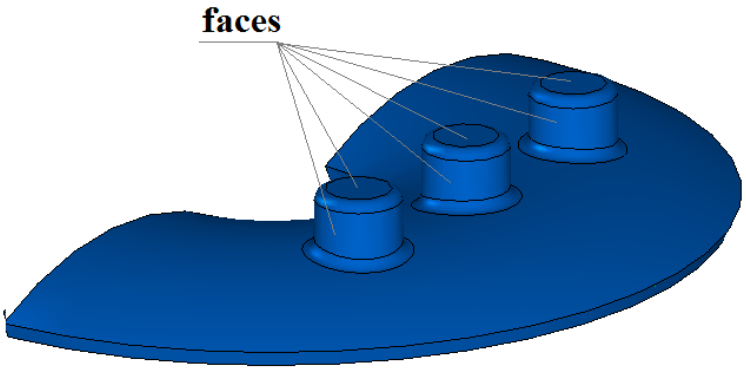


Рис. 1

_modificationType = *dmm_RemoveFacesWithAdjacentFillets*

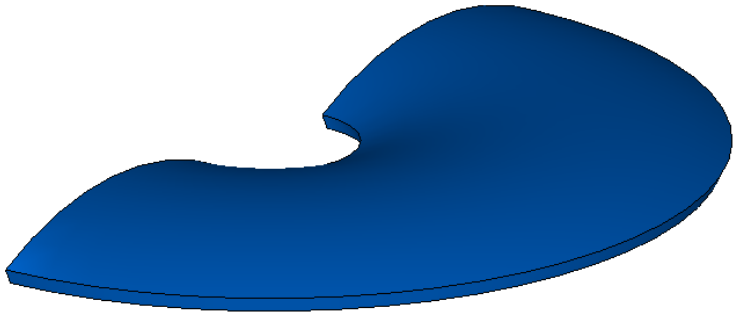


Рис. 2

Аналогичный результат можно получить, если выбрать все указанные грани и прилегающие к ним скругления и использовать тип *dmm_RemoveFaces*.

При типе модификации *dmm_Create* рассматриваемая функция создаёт новое тело из копий граней **faces**. Если грани **faces** связаны с остальными гранями скруглениями, то сначала удаляются прилегающие скругления, а затем из граней **faces** создаётся новое тело. Новое тело будет замкнутым, так как в него входят как исходные грани **faces**, так и грани, построенные на базовых поверхностях, смежных с гранями **faces**. Дополнительные грани необходимы для ликвидации краевых рёбер.

На рис. 3 приведено исходное тело, а на рис. 4 — вновь построенное тело, полученное из указанных граней исходного тела. Тело на рис. 3 состоит из пяти граней: трёх граней множества **faces** и двух граней на базе цилиндрической поверхности, на которую опирались два удалённых скругления:

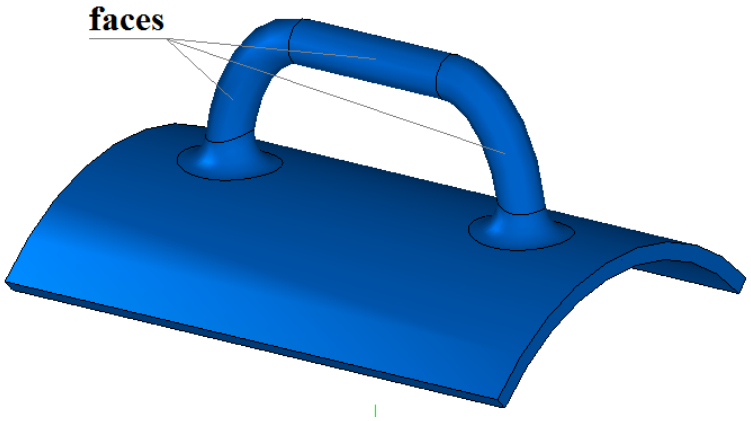


Рис. 3

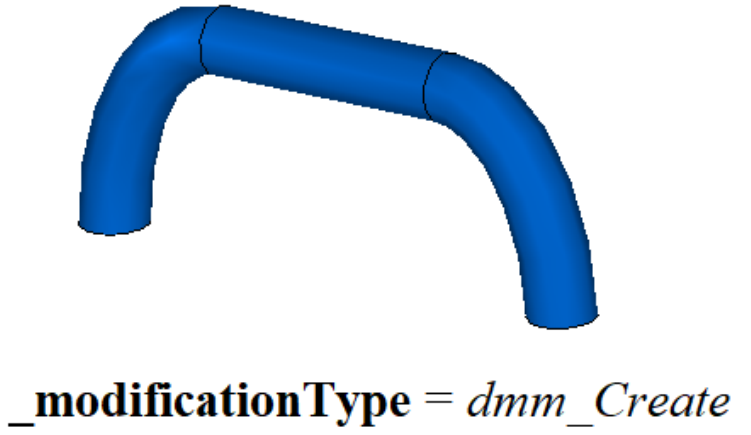
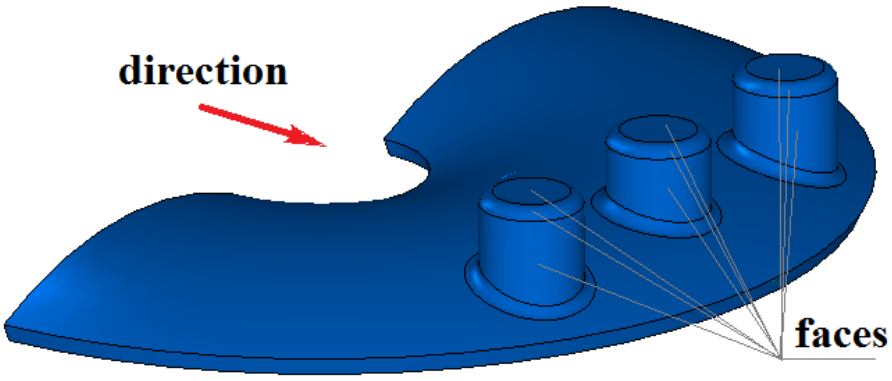


Рис. 4

При типах модификации *dmm_MoveFacesByVector* и *dmm_MoveFacesByVectorNoAdaptation* рассматриваемая функция перемещает грани **faces** относительно остальных граней в копии тела **solid**. Перемещение выполняется в направлении вектора **direction** на расстояние его длины. В режиме *dmm_MoveFacesByVector* адаптируются прилегающие скругления, в режиме *dmm_MoveFacesByVectorNoAdaptation* скругления не перестраиваются.

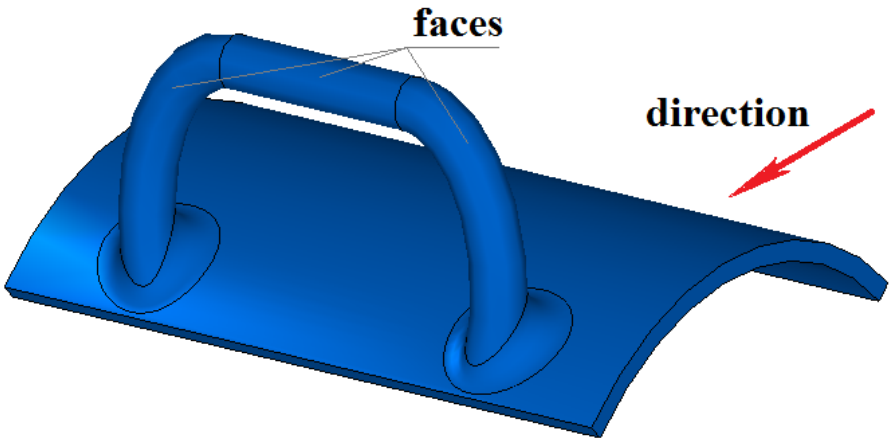
Если грани **faces** связаны с остальными гранями скруглениями, то в режиме с адаптацией сначала удаляются скругления, далее перемещаются грани **faces** и затем скругления восстанавливаются на новых местах.

На рис. 5 и рис. 6 приведены тела, построенные путём перемещения указанных граней исходных тел, приведённых на рис. 1 и рис. 3, соответственно:



_modificationType = *dmm_MoveFacesByVector*

Рис. 5



_modificationType = *dmm_MoveFacesByVector*

Рис. 6

При типе модификации *dmm_OffsetFaces* рассматриваемая функция заменяет грани **faces** в копии тела **solid** эквидистантными гранями. Эквидистантные грани смещены по нормали на расстояние, заданное параметром **value**. Если **value** положительно, смещение выполняется в сторону внешних нормалей, при отрицательном значении – в противоположную сторону.

Если грани **faces** связаны с остальными гранями скруглениями, то сначала удаляются скругления, далее грани **faces** заменяются эквидистантными, и затем скругления восстанавливаются на новых местах.

На рис. 7 и рис. 8 приведены тела, построенные путём замены указанных граней эквидистантными гранями. Исходные тела приведены на рис. 1 и рис. 3, соответственно:

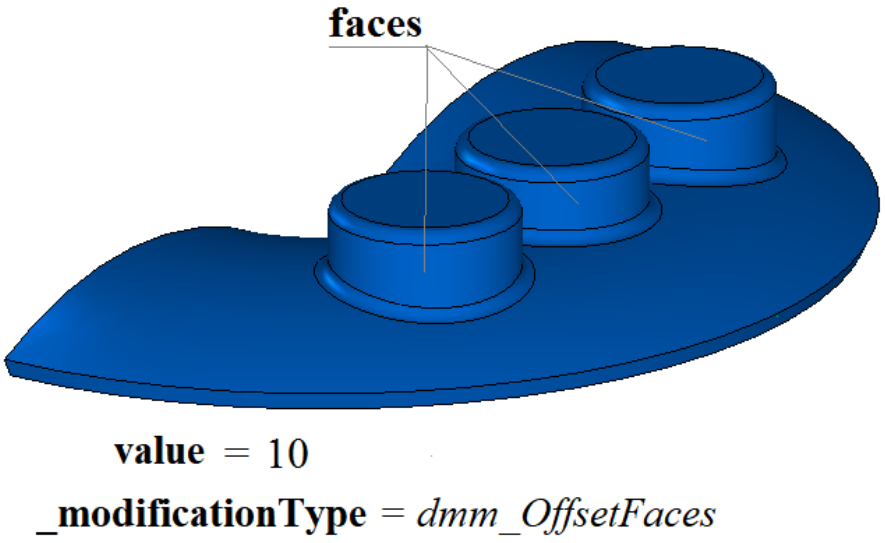


Рис. 7

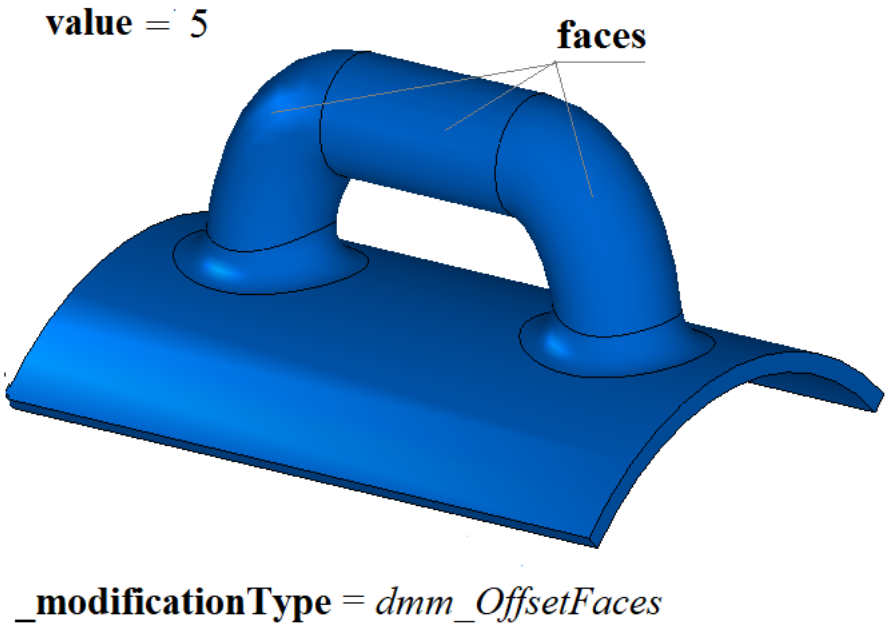
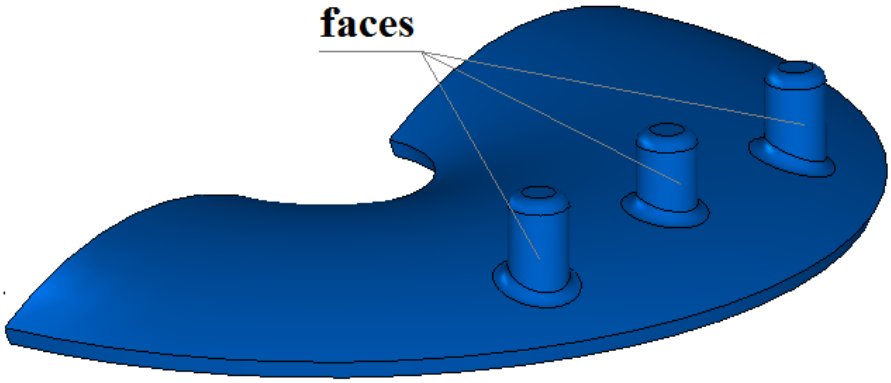


Рис. 8

Если параметр **value** отрицателен, то эквидистантные грани смещаются на отрицательное

расстояние. На рис. 9 и рис. 10 приведены тела, построенные путём замены указанных граней эквидистантными гранями с отрицательным смещением. Исходные тела приведены на рис. 5 и рис. 6, соответственно:

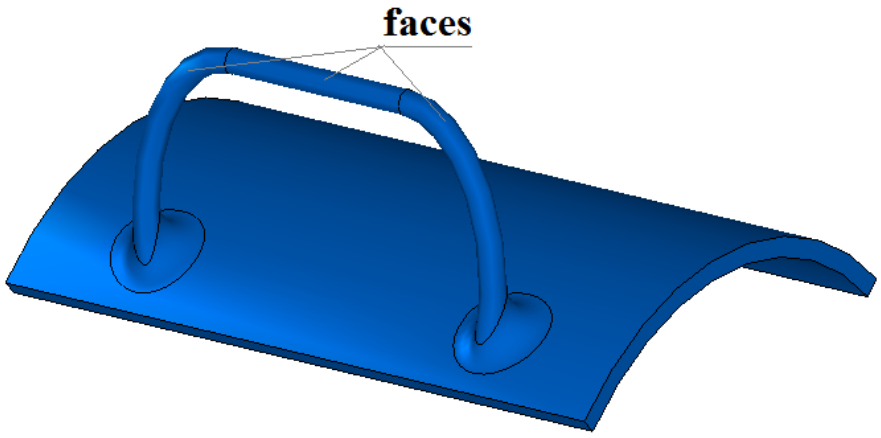


value = -5

_modificationType = *dmm_OffsetFaces*

Рис. 9

value = -2



_modificationType = *dmm_OffsetFaces*

Рис. 10

При типе модификации *dmm_FilletsRadiusChange* рассматриваемая функция изменяет радиусы граней скругления **faces** в копии тела **solid**. Радиус каждой грани скругления увеличивается или уменьшается на величину, заданную параметром **value**.

Положительное значение **value** приводит к увеличению радиуса скруглений, отрицатель-

ное – к уменьшению. При этом сохраняется замкнутость тела и непрерывность сопряжения с соседними гранями, насколько это возможно с учётом заданной геометрии.

На рис. 11 и рис. 12 приведены тела, построенные путём изменения радиуса указанных граней скругления. Исходные тела приведены на рис. 1 и рис. 3, соответственно:

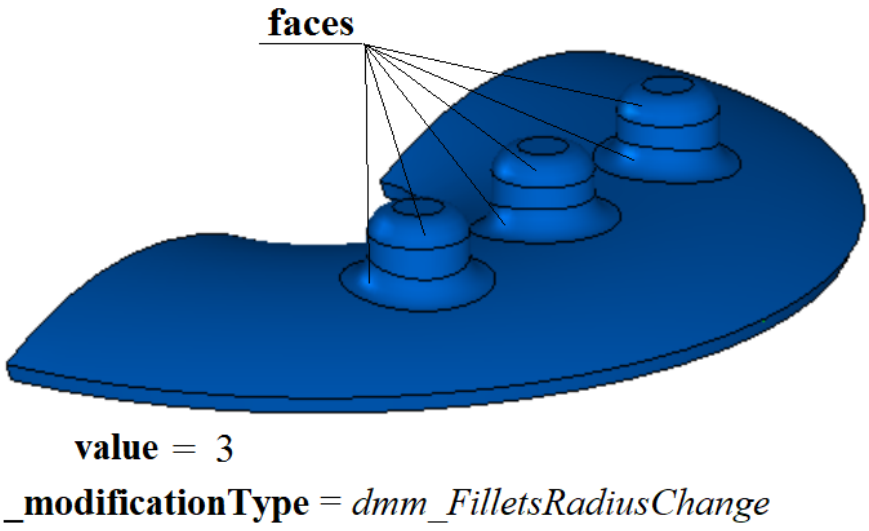


Рис. 11

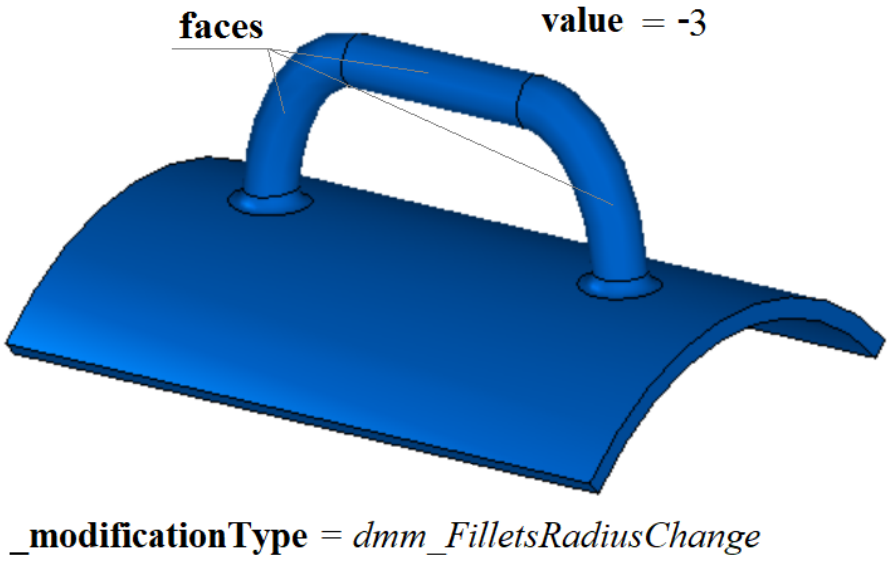


Рис. 12

При типе модификации *dmm_ConvertFacesToNurbs* рассматриваемая функция заменяет грани **faces** в копии тела **solid** деформируемыми гранями (NURBS-поверхностями), обеспечивая возможность дальнейшего редактирования этих граней по контрольным точкам.

Если грани **faces** связаны с остальными гранями скруглениями, то сначала удаляются

прилегающие скругления, далее грани **faces** заменяются деформируемыми гранями на базе соответствующих NURBS-поверхностей, после чего скругления восстанавливаются на новых местах.

На рис. 13 приведено исходное тело, а на рис. 14 – построенное тело, полученное путём замены указанной грани исходного тела деформируемой гранью, и показаны контрольные точки деформируемой грани. Деформируемая грань построена на базе NURBS-поверхности:

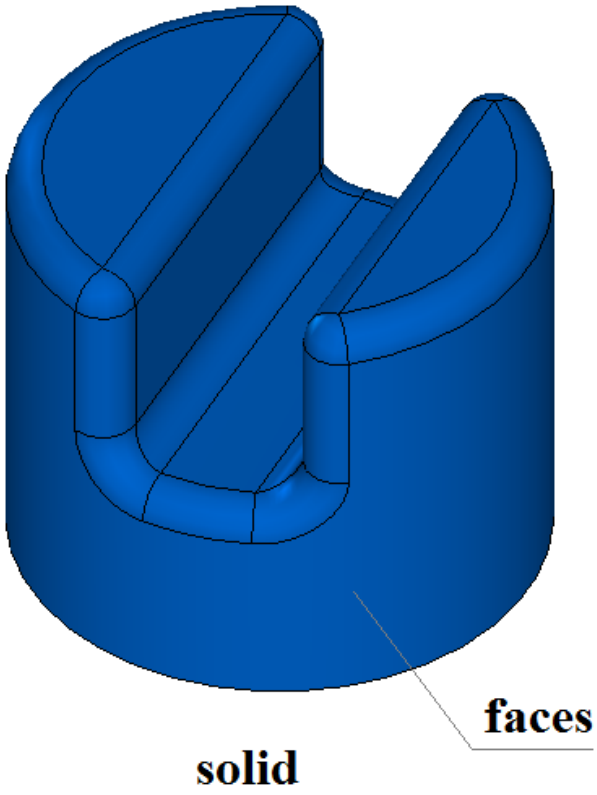
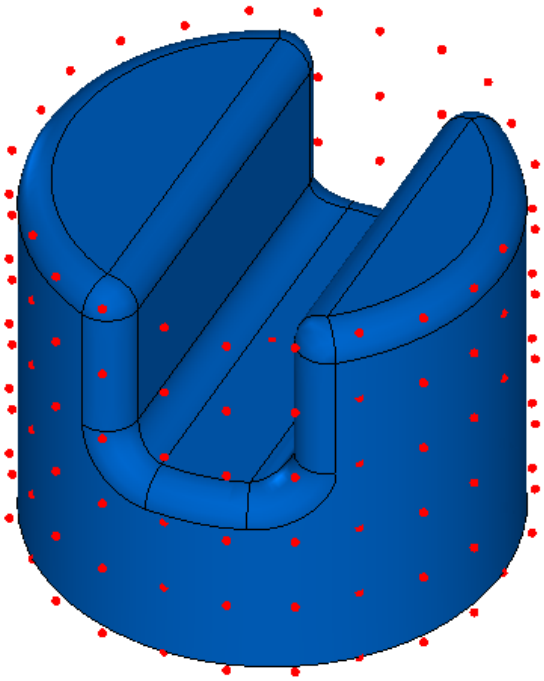


Рис. 13

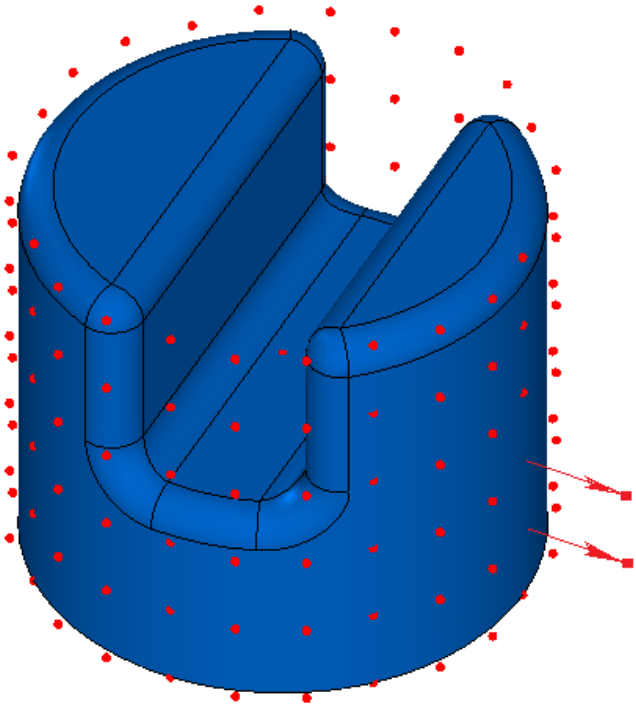


_modificationType =

dmm_ConvertFacesToNurbs

Рис. 14

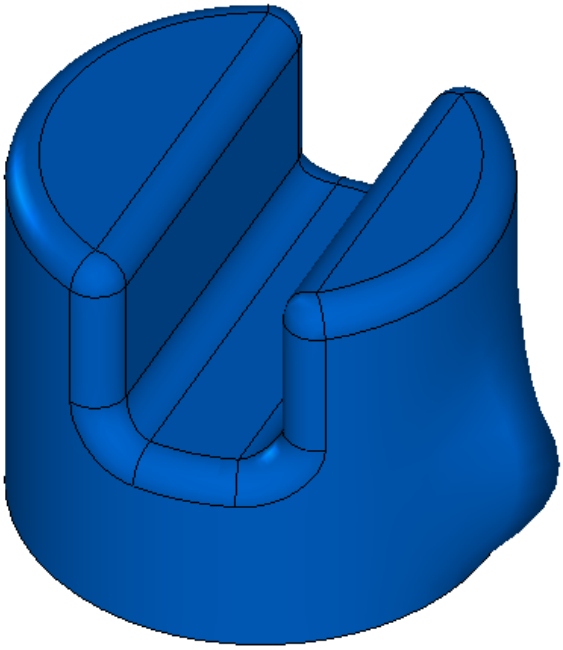
На рис. 15 приведено деформируемое тело с перемещёнными контрольными точками деформируемой грани. На рис. 16 приведён результат деформирования тела:



_modificationType =

dmm_ConvertFacesToNurbs

Рис. 15

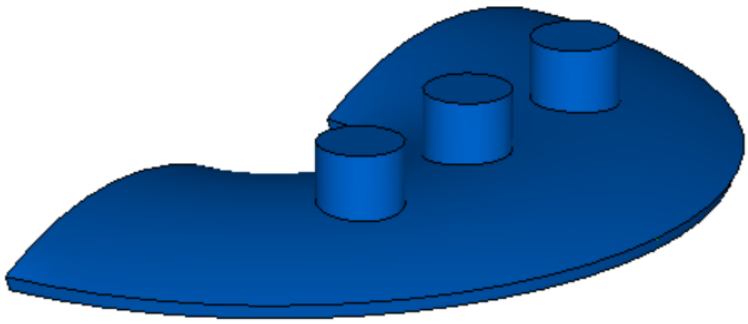


result

Рис. 16

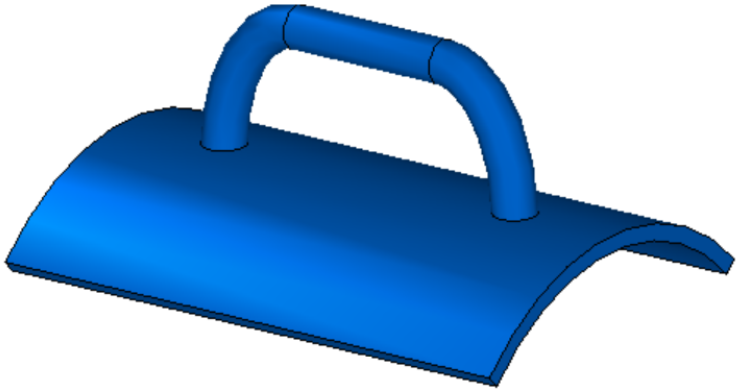
При типах модификации *dmm_PurifyFillets* и *dmm_PurifyFilletsChains* рассматриваемая функция удаляет грани скруглений **faces** в копии тела **solid** и, используя окружение, обрабатывает место удалённых граней, сохраняя замкнутость тела. В режиме *dmm_PurifyFilletsChains* дополнительно удаляются все скругления, гладко стыкующиеся с выбранными гранями.

На рис. 17 и рис. 18 приведены тела, построенные путём удаления указанных граней скругления. Исходные тела приведены на рис. 1 и рис. 3, соответственно:



_modificationType = *dmm_PurifyFillets*

Рис. 17



_modificationType = *dmm_PurifyFillets*

Рис. 18

Функция ModifiedSolid() работает при условии сохранения неизменной топологии редактируемого тела.

Функция ModifiedSolid() добавляет в журнал построенного тела строитель MbFaceModifiedSolid, который содержит все необходимые данные для выполнения операции. Строитель MbFaceModifiedSolid объявлен в файле *cr_modified_solid.h*.

Тестовое приложение *test.exe* выполняет построение модифицированного тела командами меню «Создать → Тело → Прямым редактированием».

3.6.2 Построение деформируемого тела

Построение деформируемого тела выполняется в четыре последовательных этапа, каждый из которых реализуется отдельной функцией. Цель – получить тело **result**, в котором выбранные грани исходного тела **solid** заменены на деформируемые грани. Вызов функции последующего этапа без успешного завершения предыдущего приводит к неопределённому поведению.

Этап 1. Подготовка граней к деформации

На первом этапе задают объект параметров типа **NurbsValues** и вызывают одну из функций ModifiedNurbsItem() – либо вариант с массивом граней **faces**, либо вариант с одной гранью **face**. В результате получают тело **result**, в котором указанные грани исходного тела **solid** заменены на NURBS-поверхности, пригодные для дальнейшей модификации. После успешного выполнения данного этапа следует перейти к **этапу 2** – вызову функции GetControlSurface().

Функция

```
MbResultType ModifiedNurbsItem(       MbSolid &          solid,
                                      MbeCopyMode        sameShell,
                                const NurbsValues &      p,
                                const RPArray<MbFace> &  faces,
                                const MbSNameMaker &     names,
                                      MbSolid *&         result )
```

выполняет замену выбранных граней тела деформируемыми гранями для последующего редактирования.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – вариант копирования исходного тела;
- **p** – объект с параметрами преобразования;
- **faces** – заменяемые грани тела;
- **names** – именователь операции.

Выходным параметром функции является построенное тело **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_direct.h*.

При создании объекта **p** с параметрами модификации используется следующий конструктор параметров (см. *op_shell_parameter.h*):

- **ud, vd** – порядок NURBS-копии по u и по v;
- **uc, vc** – количество контрольных точек по u и по v;
- **umin, umax, vmin, vmax** – диапазоны параметров по u и v для деформирования грани;
- **uapprox, vapprox** – флаги возможного построения приближенной поверхности, а не точной.

```
NurbsValues( size_t ud, size_t uc, double umin, double umax, bool uapprox,
             size_t vd, size_t vc, double vmin, double vmax, bool vapprox )
: uParameters( ud, uc, umin, umax, uapprox )
, vParameters( vd, vc, vmin, vmax, vapprox )
{}
```

Объект **p** хранит два набора параметров типа MbNurbsParameters: **uParameters** и **vParameters** необходимые для построения NURBS-поверхностей. Параметры **ud**, **uc**, **umin**, **umax**, **uapprox** задают поля **uParameters**, а параметры **vd**, **vc**, **vmin**, **vmax**, **vapprox** – поля **vParameters**.

Рассматриваемая функция выполняет копирование тела **solid** (в зависимости от параметра **sameShell**) и подготовку копии тела для дальнейшего деформирования указанных граней исходного тела.

Параметр **solid** содержит исходное тело, в котором указанные грани будут заменены на деформируемые грани на базе NURBS-поверхностей.

Параметр **sameShell** управляет передачей граней, рёбер и вершин от исходного тела **solid** построенному телу **result** и может принимать одно из четырех значений:

- *cm_Same* = 0 (тело не копируется);
- *cm_KeepHistory* (исходная оболочка и её копия имеют общие базовые поверхности, вершины и неизменённые операцией грани);
- *cm_KeepSurface* (исходное тело и его копия имеют общие базовые поверхности);
- *cm_Copy* (исходное тело и его копия не имеют общих данных).

Перечисление MbeCopyMode объявлено в файле *mb_enum.h* и подробно описано в параграфе *Копирование множества граней MbFaceShell*.

Параметр **faces** содержит множество граней тела **solid**, которые будут заменены на деформируемые грани на базе NURBS-поверхностей в построенном теле **result**.

Параметр **names** используется для именования новых граней, рёбер, вершин и версионирования операции.

Функция

```
MbResultType ModifiedNurbsItem(       MbSolid &       solid,
                                      MbeCopyMode     sameShell,
                                const NurbsValues &   params,
                                const MbFace &        face,
                                const MbSNameMaker &  names,
                                      MbSolid *&      result )
```

выполняет замену одной грани тела деформируемой гранью для последующего редактирования. Данная функция отличается от вышеописанной только четвёртым параметром, который содержит не множество граней, а одну заменяемую грань **face**.

Функция объявлена в файле *action_direct.h*.

Построенное тело **result** внешне может быть похожим на исходное тело **solid**, но в отличие от исходного тела имеет NURBS-поверхности, доступные для модификации. На рис. 1 слева приведено исходное цилиндрическое тело **solid**, боковая грань которого является цилиндром MbCylinderSurface, справа приведено тело **result** с боковой гранью, базирующейся на NURBS-поверхности. Контрольные точки NURBS-поверхности также приведены на рисунке справа:

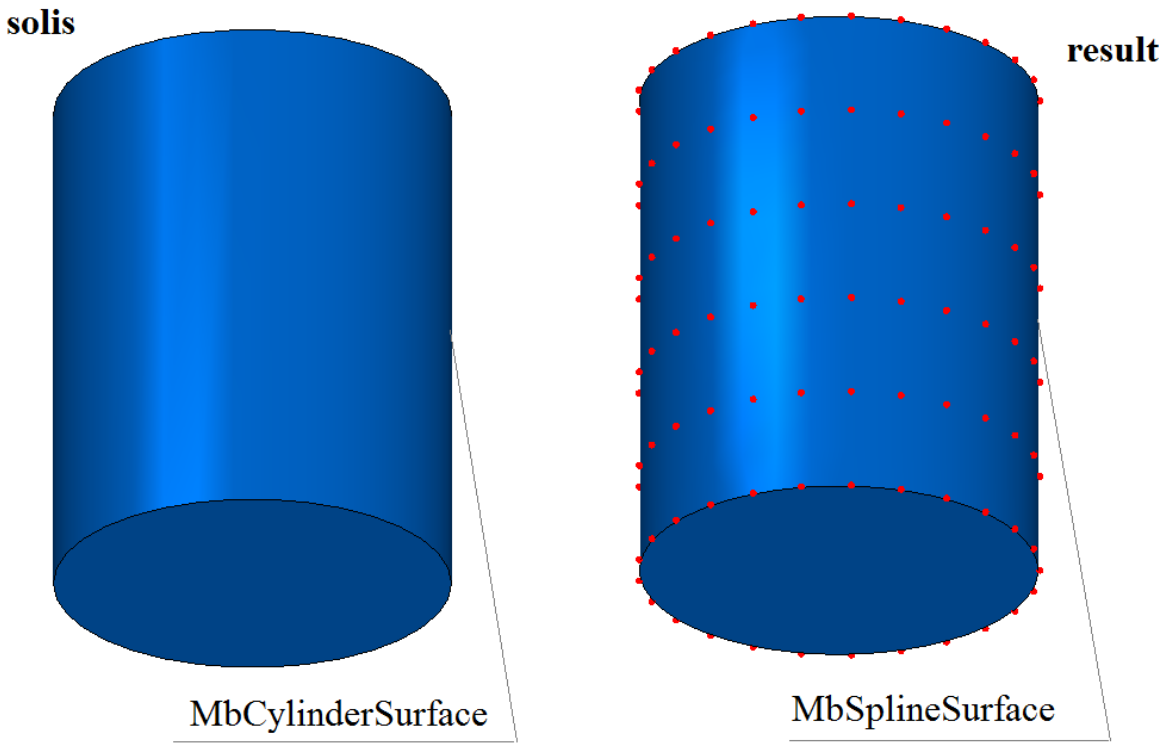


Рис. 1

Функции ModifiedNurbsItem() добавляют в журнал построенного тела строитель MbModifiedNurbsItem, который содержит все необходимые данные для выполнения операции. Строитель MbModifiedNurbsItem объявлен в файле *cr_modified_nurbs_.h*.

Этап 2. Получение базовой NURBS-поверхности грани

На втором этапе, при необходимости, для выбранной грани **face** тела **result** вызывают функцию GetControlSurface() и получают копию базовой NURBS-поверхности грани, которую можно изменять независимо от тела. После успешного выполнения данного этапа следует перейти к **этапу 3** – вызову функции FaceControlPoints().

Функция

```
MbSurface * GetControlSurface( const MbFace & face )
```

выполняет построение деформируемой NURBS-поверхности на базе исходной грани **face** для дальнейшего редактирования.

Входным параметром функции является исходная грань **face**.

Функция объявлена в файле *action_direct.h*.

Функция возвращает ненулевой указатель на поверхность, если грань **face** базируется на NURBS-поверхности. Это требование будет выполняться после успешной работы функций ModifiedNurbsItem(). Функция возвращает указатель на копию базовой поверхности грани **face**, а не на саму базовую поверхность грани. Далее у возвращаемой поверхности можно модифицировать множество контрольных точек и их веса.

Этап 3. Получение контрольных точек грани

На третьем этапе, если требуется управлять формой грани через её контрольные точки, для грани **face** вызывают функцию FaceControlPoints() и получают матрицу контрольных точек и их весов. Пользователь изменяет координаты контрольных точек и/или их веса в соответствии с требуемой деформацией. После успешного выполнения данного этапа следует перейти к **этапу 4** – вызову функции NurbsModification().

Функция

```
MbResultType FaceControlPoints( const MbFace &                face,
                                Array2<MbCartPoint3D> & controlPoints,
                                Array2<double> &        result )
```

выдаёт множество контрольных точек поверхности грани и множество их весов.

Входным параметром функции является исходная грань **face**.

Выходными параметрами функции являются:

- **controlPoints** – контрольные точки грани;
- **result** – веса контрольных точек.

Функция объявлена в файле *action_direct.h*.

Функция возвращает rt_Success, если грань **face** базируется на NURBS-поверхности, в противном случае возвращается код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*). Изменяя положение контрольных точек **controlPoints** и изменяя веса **result** контрольных точек, можно менять форму поверхности. Рассматриваемую функцию следует использовать после успешной работы функций ModifiedNurbsItem(), описанных выше.

Контрольные точки **controlPoints** передаются в виде прямоугольной матрицы, строки которой соответствуют первому параметру NURBS-поверхности, а столбцы соответствуют второму параметру NURBS-поверхности. То есть точки, лежащие в одной строке прямоугольной матрицы **controlPoints**, расположены вдоль координатной линии поверхности с фиксированным вторым параметром NURBS-поверхности, а точки, лежащие в одном столбце прямоугольной матрицы **controlPoints**, расположены вдоль координатной линии поверхности с фиксированным первым параметром NURBS-поверхности. На рис. 2 приведены контрольные точки на фоне координатных линий поверхности:

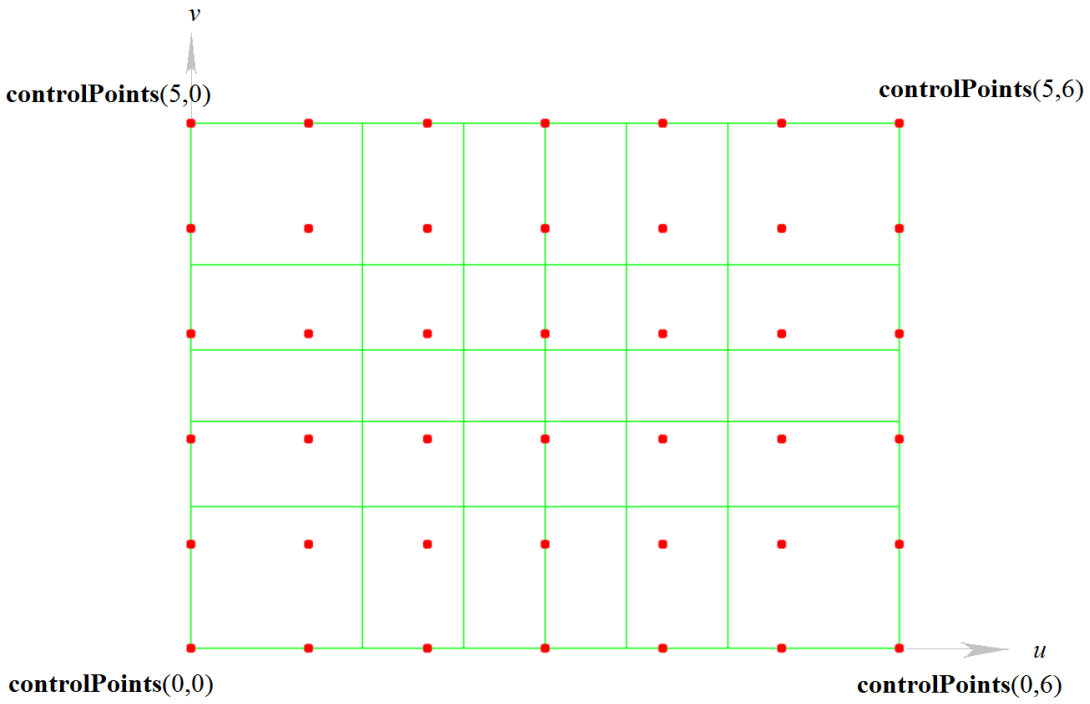


Рис. 2

Прямоугольная матрица **result** весов соответствует прямоугольной матрице контрольных точек **controlPoints**.

На рис. 3 слева приведена грань **face** и контрольные точки **controlPoints** ее поверхности, справа приведены отредактированные контрольные точки **controlPoints** для последующей модификации грани **face**:

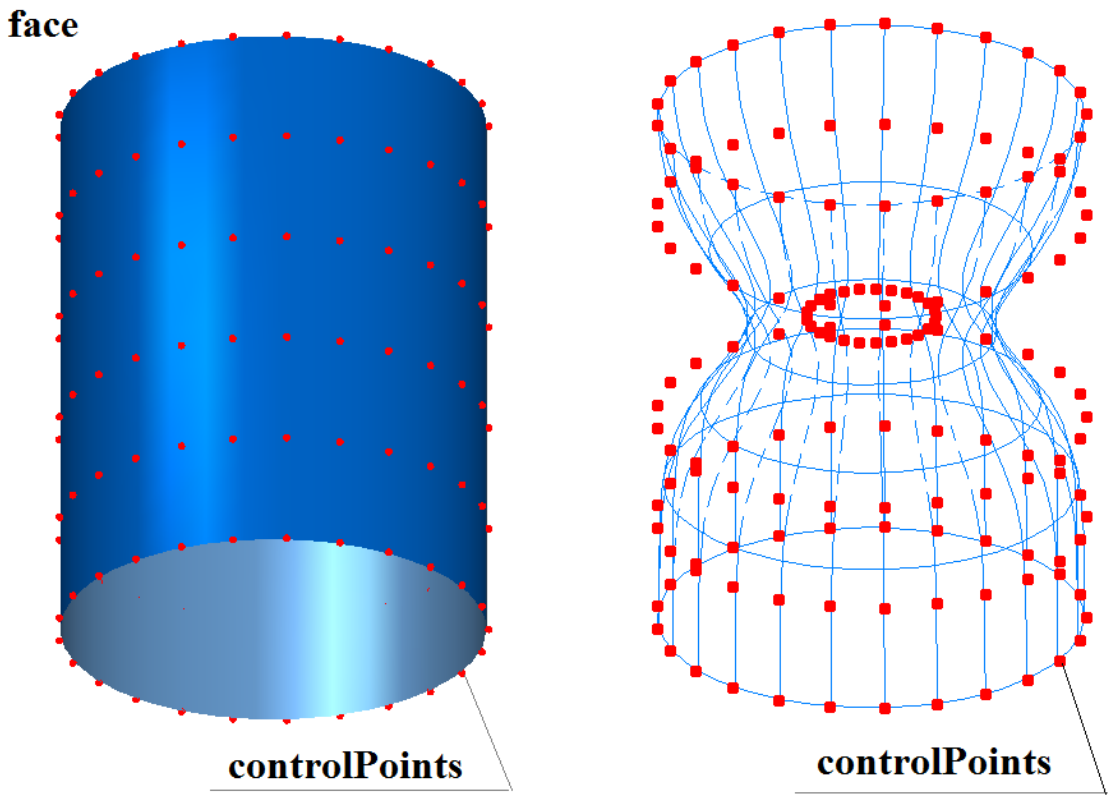


Рис. 3

Этап 4. Применение деформации к телу

На четвёртом этапе деформацию применяют к телу с помощью одной из функций NurbsModification(). В варианте с поверхностью передают модифицированную поверхность **faceSurface** и неподвижные узлы. В варианте с контрольными точками передают изменённые контрольные точки, веса и узлы. В обоих случаях на выходе получают модифицированное тело **result** с изменённой формой выбранной грани.

Функция

```
MbResultType NurbsModification(      MbSolid &        solid,
                                     MbeCopyMode      sameShell,
                                     MbFace *         face,
                                     MbSurface &      faceSurface,
                                     Array2<bool> &   fixedPoints,
                               const MbSNameMaker &   names,
                                     MbSolid *&       result )
```

выполняет деформирование заданной грани исходного тела путём подстановки присланной поверхности в изменяемую грань.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – вариант копирования исходного тела;

- **face** – изменяемая грань тела;
- **faceSurface** – новая NURBS-поверхность для грани;
- **fixedPoints** – неподвижные узлы (если указатель **fixedPoints** равен *nullptr*, все контрольные точки считаются подвижными);
- **names** – именователь операции.

Выходным параметром функции является построенное модифицированное тело **result**.

При удачной работе функция возвращает rt_Success, в противном случае возвращается код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_direct.h*.

Рассматриваемая функция копирует тело **solid** и подставляет присланную поверхность **faceSurface** в копию грани **face** тела **solid**.

Поверхность **faceSurface** должна быть NURBS-поверхностью. Параметр **fixedPoints** представляет собой прямоугольную матрицу, строки и столбцы которой соответствуют строкам и столбцам матрицы контрольных точек NURBS-поверхности **faceSurface**. Элементы матрицы со значением *true* определяют неподвижные контрольные точки NURBS-поверхности.

Функция

```
MbResultType NurbsModification(       MbSolid &                solid,
                                      MbeCopyMode              sameShell,
                                      MbFace *                 face,
                                const Array2<MbCartPoint3D> &  controlPoints,
                                const Array2<double> &         weights,
                                      Array2<bool> *           fixedPoints,
                                const MbSNameMaker &           names,
                                      MbSolid *&               result )
```

выполняет деформирование указанной грани исходного тела по заданным контрольным точкам и их весам.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – вариант копирования исходного тела;
- **face** – изменяемая грань тела;
- **controlPoints** – заданные контрольные точки изменяемой грани;
- **weights** – веса контрольных точек;
- **fixedPoints** – неподвижные узлы (если указатель **fixedPoints** равен *nullptr*, все контрольные точки считаются подвижными);
- **names** – именователь операции.

Выходным параметром функции является построенное тело **result**.

При удачной работе функция возвращает rt_Success, в противном случае возвращается код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_direct.h*.

Рассматриваемую функцию следует использовать после успешной работы функций ModifiedNurbsItem(), GetControlSurface() и FaceControlPoints(), описанных выше. Функции ModifiedNurbsItem() подготавливают грани для модификации.

На рис. 4 приведено исходное цилиндрическое тело **solid**, боковая грань которого базируется на NURBS-поверхности, и у которой подлежат модификации контрольные точки. Также на рисунке приведено тело **result** с модифицированной боковой гранью, контрольные точки **controlPoints** которой приведены на рис. 3:

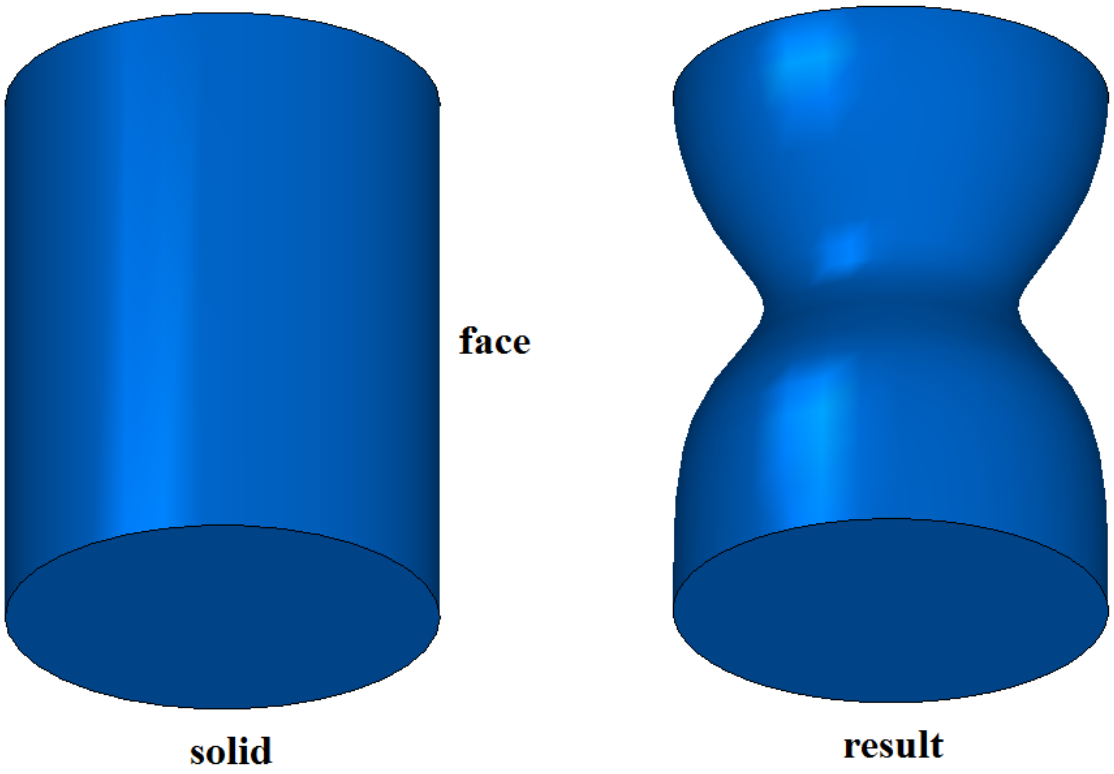


Рис. 4

Функции NurbsModification() добавляют в журнал построенного тела строитель MbNurbsModification, который содержит все необходимые данные для выполнения операции. Строитель MbNurbsModification объявлен в файле *cr_modified_nurbs_.h*.

3.6.3 Построение деформируемого параллелепипеда

Функция

```
MbResultType NurbsBlockSolid( const MbPlacement3D &     place,
                                    double              ax,
                                    double              ay,
                                    double              az,
                                    bool                outDir,
                              const MbSNameMaker &      names,
                                    SimpleName          name,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                                    NurbsBlockValues &  param,
                                    MbSolid *&          result )
```

выполняет построение тела в форме прямого параллелепипеда с деформируемыми гранями.

Входными параметрами функции являются:

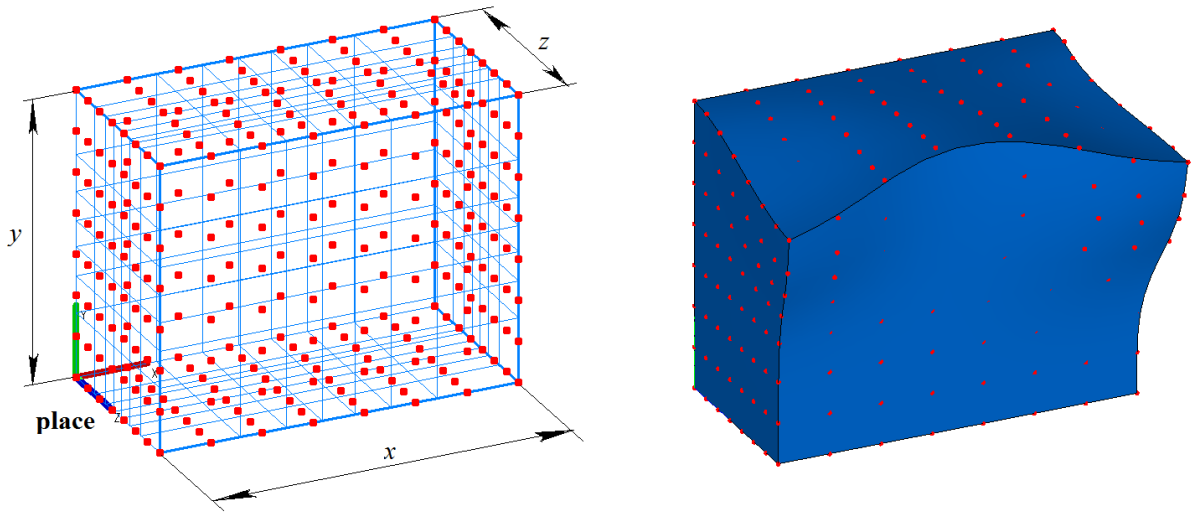
- **place** – локальная система координат;
- **ax** – размер по оси X;
- **ay** – размер по оси Y;
- **az** – размер по оси Z;
- **outDir** – ориентация нормалей граней наружу тела (*true* – нормали направлены наружу);
- **names** – именователь операции;
- **name** – главное имя;
- **param** – параметры построения граней (см. структуру NurbsBlockValues в *op_shell_parameter.h*).

Выходным параметром функции является тело **result**, построенное на NURBS-поверхностях.

Функция объявлена в файле *action_direct.h*.

Рассматриваемая функция позволяет получить деформируемое тело в форме прямоугольного параллелепипеда, размеры которого в локальной системе координат **place** определяют параметры **ax**, **ay**, **az**. Все грани параллелепипеда построены на базе плоских NURBS-поверхностей, порядок которых и количество контрольных точек определяются параметром **param**.

На рисунке приведен параллелепипед **result**, построенный с помощью функции NurbsBlockSolid(), и его контрольные точки. На рисунке справа приведен тот же параллелепипед после деформирования путем перемещения контрольных точек граней:



Функция NurbsBlockSolid() добавляет в журнал построенного тела строитель MbNurbsBlockSolid, который содержит все необходимые данные для выполнения операции. Строитель MbNurbsBlockSolid объявлен в файле *cr_nurbs_block_solid.h*.

3.6.4 Построение сглаженной поверхности

Функция

```
MbResultType SplineSurfaceSmoothing( const MbSplineSurface &  surface,
                                           size_t             udegree,
                                           size_t             vdegree,
                                           MbSplineSurface *& result );
```

выполняет сглаживание копии исходной поверхности.

Входными параметрами функции являются:

- **surface** - исходная сплайновая поверхность;
- **udegree** - новый порядок сплайновой поверхности по первому параметру поверхности;
- **vdegree** - новый порядок сплайновой поверхности по второму параметру.

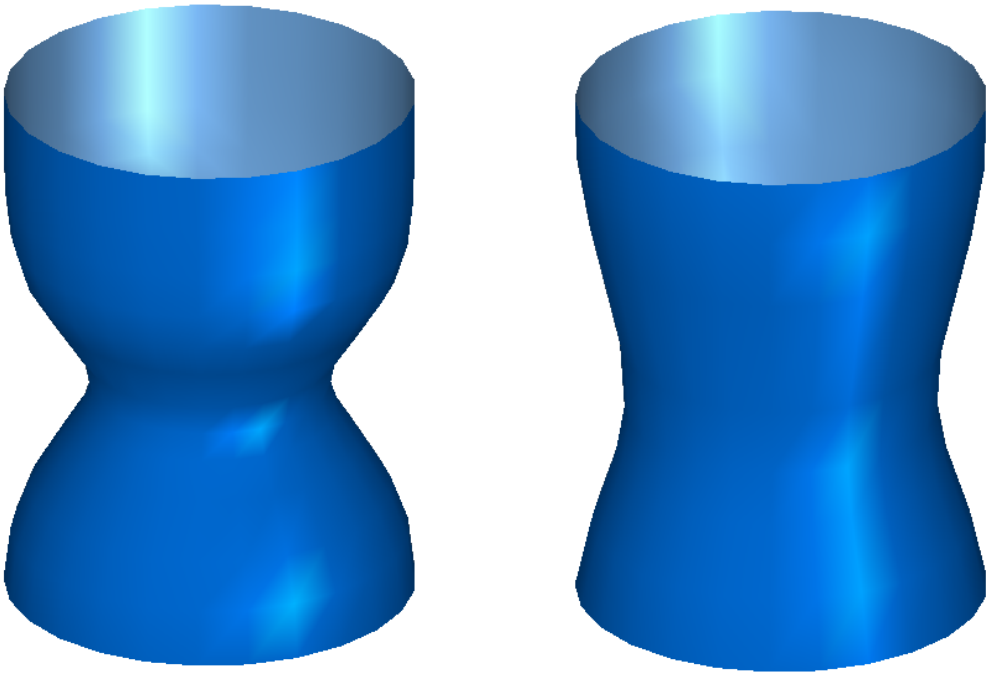
Выходным параметром функции является сглаженная копия исходной поверхности **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_direct.h*.

Рассматриваемая функция позволяет сгладить исходную поверхность **surface**, не меняя ее порядок сплайнов и количество контрольных точек. Сглаживанием управляют параметры **udegree** и **vdegree**, которые должны быть больше соответствующих порядков сплайнов исходной поверхности.

На рисунке приведены исходная NURBS-поверхность **surface** и сглаженная поверхность **result**:



3.6.5 Построение трансформированного тела

Функция

```
MbResultType TransformedSolid(      MbSolid &          solid,
                                    MbeCopyMode        sameShell,
                              const TransformValues &  p,
                              const MbSNameMaker &     names,
                                    MbSolid *&         result )
```

выполняет трансформацию исходного тела по заданной матрице, рассчитанной по габаритному кубу.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **sameShell** – вариант копирования исходного тела;
- **p** – параметры трансформации;
- **names** – именователь операции.

Выходным параметром функции является модифицированное тело **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_direct.h*.

При создании объекта **p** с параметрами трансформации используется один из следующих конструкторов (см. *op_shell_parameter.h*):

Конструктор по умолчанию:

```
TransformValues()
: matrix()
, fixedPoint()
, useFixed( false )
, isotropy( false )
{}
```

Конструктор по матрице:

```
TransformValues( const MbMatrix3D & m )
: matrix( m )
, fixedPoint()
, useFixed( false )
, isotropy( false )
{}
```

Конструктор по матрице и неподвижной точке преобразования:

```
TransformValues( const MbMatrix3D & m, const MbCartPoint3D & f, bool fix = false,␣
↪bool iso = false )
: matrix( m )
, fixedPoint( f )
, useFixed( fix )
, isotropy( iso )
{}
```

Конструктор по неподвижной точке преобразования и масштабам по осям:

```
TransformValues( double sX, double sY, double sZ, const MbCartPoint3D & fP );
```

Конструктор копирования:

```
TransformValues( const TransformValues & other )
: matrix    ( other.matrix )
, fixedPoint( other.fixedPoint )
, useFixed  ( other.useFixed )
, isotropy  ( other.isotropy )
{}
```

Рассматриваемая функция выполняет масштабирование тела с помощью матрицы, имеющей одинаковые или разные масштабы преобразования по осям глобальной системы координат.

Параметр **solid** содержит исходное тело, которое подлежит обработке.

Параметр **sameShell** управляет копированием данных от исходного тела **solid** построенному телу **result** (см. *Копирование множества граней MbFaceShell*).

Параметр **p** содержит матрицу трансформации **p**.*matrix* и величины **p**.*fixedPoint*, **p**.*useFixed*, **p**.*isotropy*, которые используются для расчёта матрицы трансформации. Параметр трансформации TransformValues объявлен в файле *op_shell_parameter.h*.

Параметр **names** используется для именования элементов тела.

Матрица трансформации рассчитывается по габаритному кубу исходного тела **solid** и смещению одной из точек габаритного куба методом:

```
bool MbCube::CalculateMatrix(      size_t              pIndex,
                             const MbCartPoint3D &     point,
                             const MbCartPoint3D &     fixedPoint,
                                   bool                useFixed,
                                   bool                isotropy,
                                   MbMatrix3D &        matrix ) const;
```

где

- **pIndex** – номер смещаемой точки габаритного куба;
- **point** – точка пространства, с которой нужно совместить точку габаритного куба с номером **pIndex**;
- **fixedPoint** – неподвижная точка преобразования (используется, если **useFixed** = *true*);
- **useFixed** – флаг использования неподвижной точки;
- **isotropy** – флаг равенства масштабов преобразования;
- **matrix** – рассчитанная матрица преобразования.

На рис. 1 приведена нумерация точек габаритного куба тела **solid**. Вершины габаритного куба имеют номера 0–7, середины рёбер габаритного куба имеют номера 8–19, центры граней габаритного куба имеют номера 20–25.

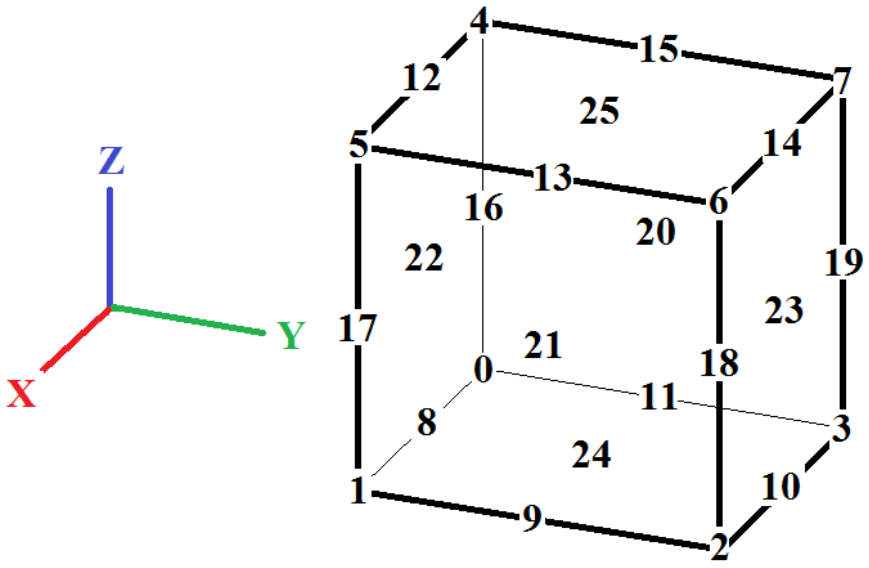


Рис. 1

Матрица трансформации **matrix** рассчитывается по деформированному кубу, полученному путём совмещения точки габаритного куба с номером **pIndex** с точкой пространства **point**.

Если **useFixed** = *true*, то неподвижной точкой преобразования будет точка **fixedPoint**. Если **useFixed** = *false*, то неподвижной точкой преобразования будет точка габаритного куба, противолежащая точке с номером **pIndex**. Противолежащими точками габаритного куба считаются точки: 0–6, 1–7, 2–4, 3–5, 8–14, 9–15, 10–12, 11–13, 16–18, 17–19, 20–21, 22–23, 24–25.

Если **isotropy** = *true*, то масштабы преобразования по всем осям будут одинаковы и пропорциональны вектору смещения точки габаритного куба с номером **pIndex**. Если **isotropy** = *false*, то масштабы преобразования будут пропорциональны проекциям на стороны габаритного куба вектора смещения точки габаритного куба с номером **pIndex**. Величины **p**.**fixedPoint**, **p**.*useFixed*, **p**.*isotropy* и **p**.**matrix** используются в качестве параметров **fixedPoint**, **useFixed**, **isotropy** и **matrix** метода CalculateMatrix().

На рис. 2 приведено исходное тело **solid**, точки габаритного куба и точка пространства **point**, с которой нужно совместить указанную точку габаритного куба, а на рис. 3 приведено трансформированное тело **result**, построенное рассматриваемой функцией с параметрами **useFixed** = *false* и **isotropy** = *false*.

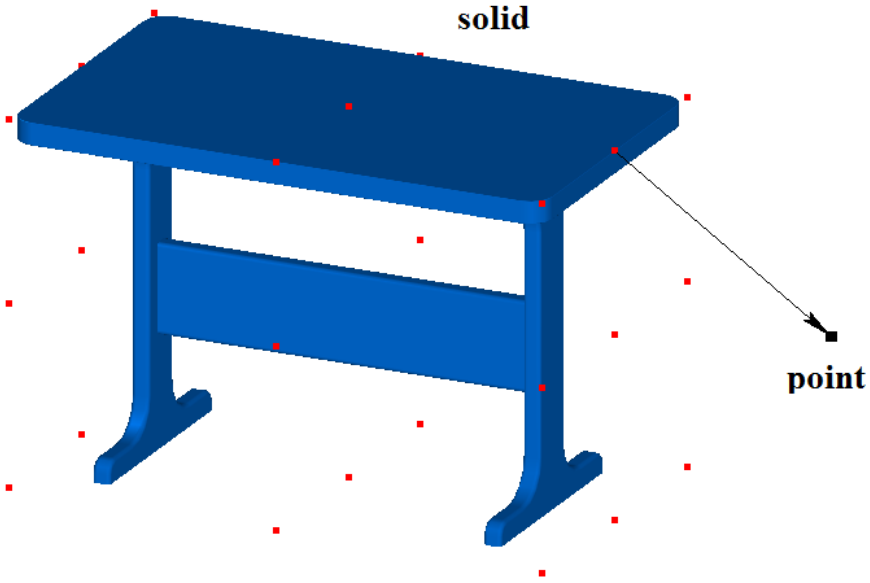


Рис. 2

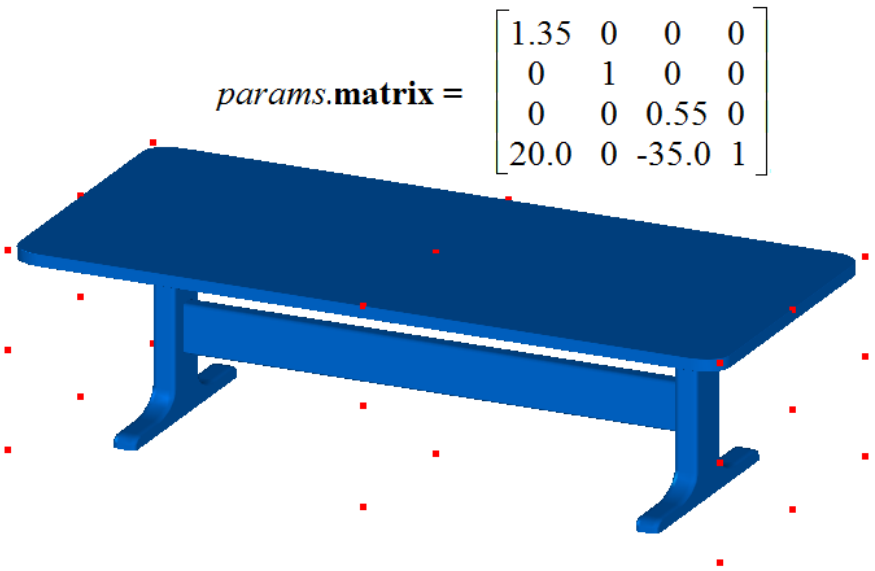


Рис. 3

Трансформировать тело по заданной матрице можно также методом Transform(), который перегружен у всех геометрических объектов, включая MbSolid, но при этом тело не будет содержать информацию о выполненных над ним действиях.

Функция TransformedSolid() добавляет в журнал построенного тела строитель MbTransformedSolid, который содержит все необходимые данные для выполнения операции. Строитель MbTransformedSolid объявлен в файле *cr_transformed_solid.h*.

Тестовое приложение *test.exe* выполняет построение трансформированного тела командой меню «Создать → Тело → Прямым редактированием → Трансформированием габарита».

3.7 МЕТОДЫ ПОСТРОЕНИЯ ТЕЛ ИЗ ЛИСТОВОГО МЕТАЛЛА

Геометрическое ядро C3D поддерживает функции, которые позволяют строить модели конструкций из листового металла. Это функции создания листовых тел (пластины и обечайки), а также операции над листовыми телами: сгиб по рёбрам, сгиб вдоль линии, подсечка, сгиб по эскизу, развёртка, замыкание углов, вырез, добавление пластины, штамповка, жалюзи и буртик. Также представлен ряд сервисных функций для работы с листовыми телами. Будем называть широкие грани пластины листовыми, а остальные — боковыми; при этом листовую грань сгиба с меньшим радиусом будем называть внутренней гранью сгиба, а с большим радиусом — внешней гранью сгиба.

3.7.1 Построение листового тела

Функция

```
MbResultType CreateSheetSolid( const MbPlacement3D &           placement,
                                     RPArray<MbContour> &      contours,
                                     bool                      unbended,
                               const MbSheetMetalValues &      params,
                                     RPArray<MbSNameMaker> *   nameMakers,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                                    RPArray<MbSMBendNames> &  resultBends,
                                    MbSolid *&                result );
```

выполняет построение листового тела выдавливанием плоских контуров.

Входными параметрами функции являются:

- **placement** – локальная система координат;
- **contours** – контуры листового тела, заданные в плоскости XY локальной системы координат;
- **unbended** – флаг построения тела в разогнутом состоянии;
- **params** – параметры построения;
- **nameMakers** – именователи построения.

Выходными параметрами функции являются имена листовых граней построенных сгибов, записываемые в **resultBends** и само тело **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция объявлена в файле *action_sheet.h*.

Если контуры замкнутые, то листовое тело строится на заданную толщину с заданным контуром. Среди замкнутых контуров один контур должен быть внешним, содержащим внутри себя остальные внутренние контуры. По внутренним контурам будут сформированы вырезы.

Если контуры незамкнутые, то листовое тело строится выдавливанием на заданные расстояния замкнутых контуров, полученных из незамкнутых контуров путём скругления углов между отрезками с радиусами, заданными в **params** или **resultBends**. Получившимся контурам придаётся толщина листового тела в ту или иную сторону. Сторона придания толщины указывается в параметрах операции **params**. Если контур уже содержит дугу, то она должна гладко стыковаться с соседними сегментами.

Только в случае незамкнутых контуров возможно построение листового тела со сгибами, и только для этого случая имеет смысл флаг **unbended**. В случае, если массив параметров сгибов **resultBends** пустой, сгибы строятся по единым параметрам, заданным в **params**. В этом случае имена листовых граней построенных сгибов никуда не записываются. Если же массив параметров **resultBends** заполнен, то количество элементов в этом массиве должно совпадать с количеством формируемых сгибов.

Параметры для сгиба ищутся следующим образом. Каждому контуру из массива **contours** соответствует именователь из массива **nameMakers**, а в каждом именователе содержится массив имён сегментов контура. Если сгиб формируется в месте соединения двух отрезков, то параметры этого сгиба находятся в элементе массива **resultBends**, содержащем имена этих отрезков в полях **segName1** и **segName2**. Если сгиб формируется на месте дуги, то параметры для него находятся в элементе массива **resultBends**, у которого в **segName1** лежит имя дуги, а **segName2** равен *SIMPLENAME_MAX*.

На рис. 1 приведено листовое тело, построенное по нескольким замкнутым эскизам:

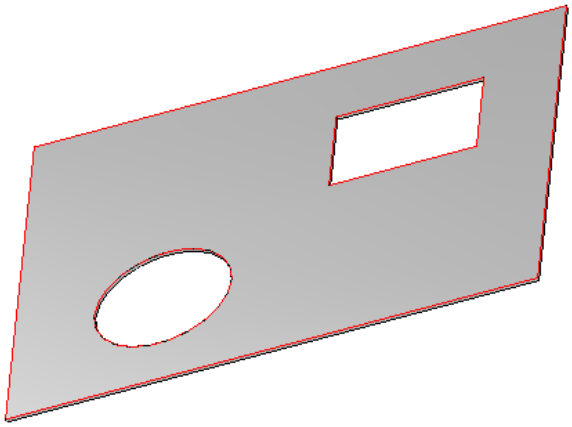


Рис. 1

На рис. 2 приведено листовое тело, построенное по незамкнутому эскизу, содержащему дугу и пару стыкующихся под углом отрезков:

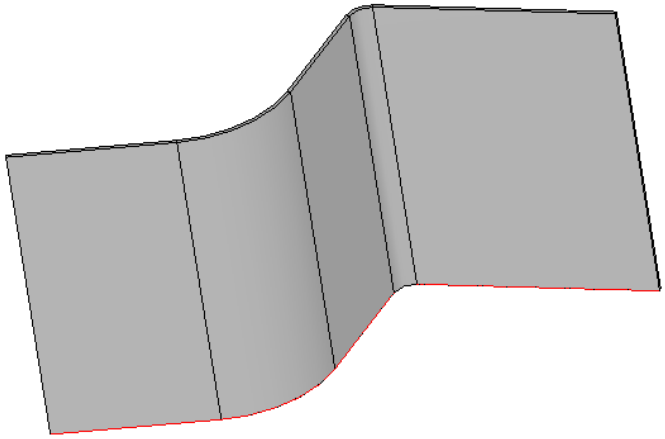


Рис. 2

Метод CreateSheetSolid() добавляет в журнал построенного тела строитель MbSheetMetalSolid, который содержит все необходимые данные для выполнения операции. Строитель MbSheetMetalSolid объявлен в файле *cr_sheet_metal_solid.h*.

3.7.2 Разделение сгибов листового тела на пары

Функция

```
bool CollectBends( const MbFaceShell &                faceShell,
                   const RPArray<MbFace> &            innerFaces,
                   const RPArray<MbFace> &            outerFaces,
                         RPArray<MbSheetMetalBend> & result );
```

разделяет грани сгибов листового тела на пары.

Входными параметрами функции являются:

- **faceShell** - оболочка листового тела, содержащая грани сгибов;
- **innerFaces** - внутренние грани сгибов;
- **outerFaces** - внешние грани сгибов.

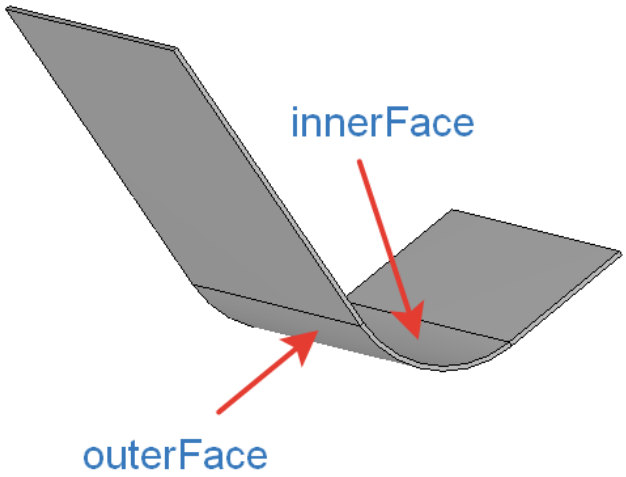
Выходным параметром функции является множество найденных пар граней **result**, составляющих сгибы.

При удачной работе функция возвращает *true*, в противном случае - *false*.

Функция объявлена в файле *action_sheet.h*.

Функция ищет для каждого сгиба пару граней (внутреннюю и внешнюю) среди неупорядоченного набора **innerFaces** и **outerFaces** и по найденным парам формирует объекты сгибов, которые добавляются в массив **result**.

На рисунке изображены внутренняя и внешняя грани сгиба:



3.7.3 Проверка грани на возможность быть фиксированной

Функция

```
bool IsSuitableForFixed( const MbFace & face );
```

выполняет проверку грани на возможность использования в качестве фиксированной при сгибе или разгибе сгиба листового тела.

Функция возвращает *true*, если грань может быть выбрана в качестве фиксированной для сгиба и разгиба листового тела, в противном случае возвращается *false*.

Функция объявлена в файле *action_sheet.h*.

3.7.4 Поиск граней для кривой

Функция

```
void FindCurveFaces( const RPArray<MbFace> &  faces,
                     const MbCurve3D &        curve,
                           RPArray<MbFace> &  result );
```

выполняет поиск граней, на которых лежит указанная прямолинейная кривая.

Входными параметрами функции являются:

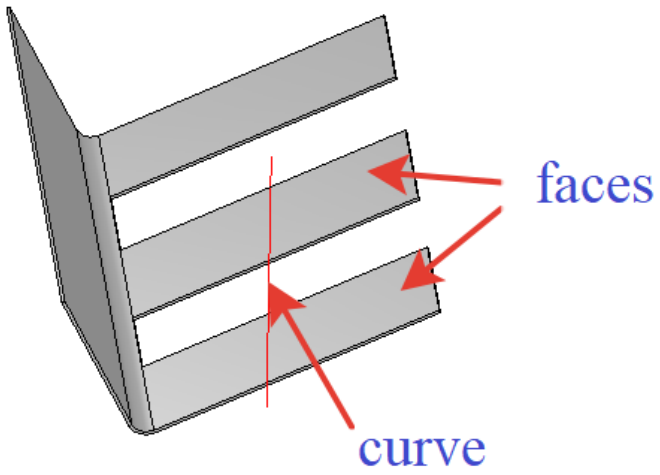
- **faces** - множество граней для поиска
- **curve** - прямолинейная кривая, которая может лежать на некоторых гранях множества **faces**

Выходным параметром функции является множество граней **result**, на которых лежит кривая **curve**.

Функция не возвращает значений.

Функция объявлена в файле *action_sheet.h*.

На рисунке изображены две листовые грани, лежащие под отрезком:



3.7.5 Поиск листовой грани тела

Функция

```
MbFace * FindSheetFace( const MbCurveEdge & edge );
```

выполняет поиск верхней или нижней грани листового тела, содержащей заданное ребро.

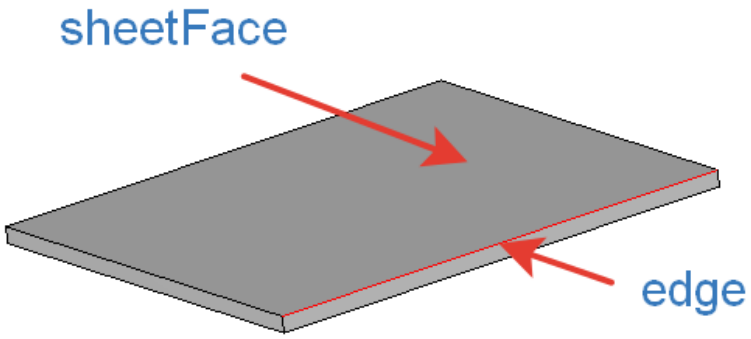
Входным параметром функции является неориентированное ребро листовой грани **edge**.

При удачной работе функция возвращает указатель на найденную листовую грань, в противном случае возвращается *NULL*.

Функция объявлена в файле *action_sheet.h*.

Поиск верхней или нижней грани листового тела выполняется среди двух стыкующихся в ребре **edge** граней.

На рисунке изображена листовая грань, смежная с ребром:



3.7.6 Поиск парной грани сгиба листового тела

Функция

```
MbFace * FindPairBendFace( const MbFace & face )
```

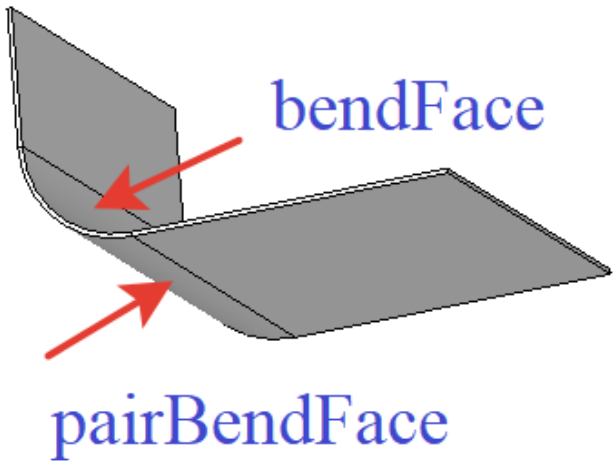
выполняет поиск грани сгиба листового тела, парной к заданной.

Входным параметром функции является грань сгиба листового тела **face**.

При удачной работе функция возвращает указатель на найденную грань, в противном случае возвращается *NULL*.

Функция объявлена в файле *action_sheet.h*.

На рисунке изображены грань сгиба **bendFace** и парная ей грань **pairBendFace**:



3.7.7 Поиск плоской грани листового тела

Функция

```
MbFace * GetPairPlanarFaceByEdge( const MbCurveEdge & edge,
                                  const double        begDistance,
                                  const double        endDistance );
```

выполняет поиск парной плоской грани листового тела по заданному ребру и расстояниям от его концов.

Входными параметрами функции являются:

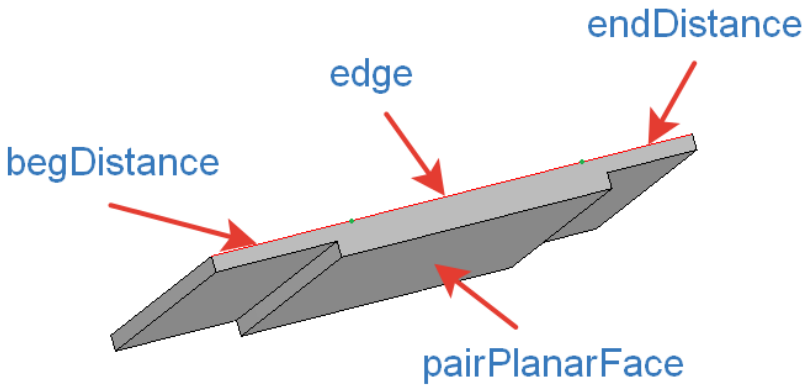
- **edge** – ребро, по которому искать.
- **begDistance** – расстояние от начала ребра.
- **endDistance** – расстояние от конца ребра.

При удачной работе функция возвращает указатель на найденную плоскую листовую грань, в противном случае возвращается *NULL*.

Функция объявлен в файле *action_sheet.h*.

Функция ищет парную грань листового тела для содержащей ребро **edge** листовой грани. Используется в операции сгиб на ребре для определения толщины листа в месте подклейки сгиба. Метод применяется для многотолщинных листовых тел в условиях, когда выбранной грани листового тела соответствует несколько парных ей граней, находящихся на разном расстоянии от неё. Положительные расстояния **begDistance** и **endDistance** означают отступ наружу вдоль ребра, а отрицательные расстояния **begDistance** и **endDistance** означают отступ внутрь вдоль ребра.

На рисунке изображена плоская листовая грань **pairPlanarFace**, найденная по ребру и отступам от его концов.



3.7.8 Поиск парной грани листового тела

Функция

```
MbFace * GetPairPlanarFaceByCurve( const MbFace &    sheetFace,
                                   const MbCurve3D & curve );
```

выполняет поиск плоской парной грани листового тела по лежащей на грани **sheetFace** прямолинейной кривой **curve**.

Входными параметрами функции являются:

- **sheetFace** – плоская грань листового тела.
- **curve** – лежащая на листовой грани прямолинейная кривая.

При удачной работе функция возвращает указатель на найденную грань, в противном случае функция возвращает *NULL*.

Функция объявлена в файле *action_sheet.h*.

Функция поиска парной грани листового тела является вспомогательной для функции построения сгиба по линии. Функция применяется для многотолщинных листовых тел в условиях, когда выбранной листовой грани соответствует несколько парных ей граней, находящихся на разном расстоянии от неё.

На рис. 1 изображена плоская листовая грань, найденная по противоположной ей грани и лежащей на ней кривой.

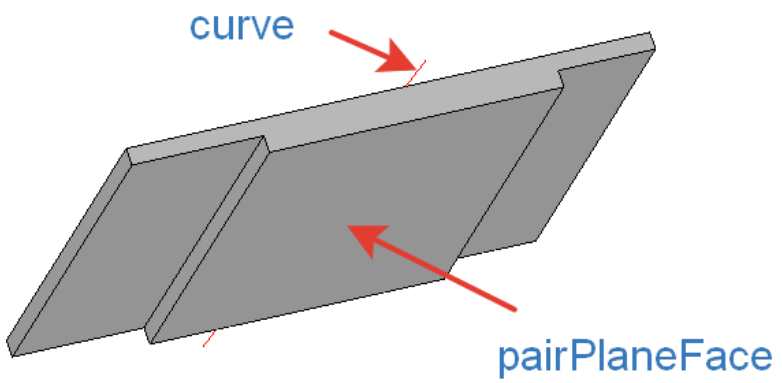


Рис. 1

Функция

```
MbFace * GetPairPlanarFaceByContour( const MbFaceShell &            shell,
                                     const MbFace &                 sheetFace,
                                     const MbPlacement3D &          place,
                                     const RPArray<const MbCurve> & segments );
```

выполняет поиск плоской парной грани листового тела по множеству сегментов контура.

Входными параметрами функции являются:

- **shell** – оболочка листового тела.
- **sheetFace** – грань листового тела.
- **place** – локальная система координат, лежащая на грани **sheetFace**.
- **segments** – кривые, лежащие в плоскости XY локальной системы координат **place**.

При удачной работе функция возвращает указатель на найденную грань, в противном случае функция возвращает *NULL*.

Функция объявлена в файле *action_sheet.h*.

Функция поиска парной грани листового тела является вспомогательной для функций построения листового тела, базирующихся на контурах. Функция применяется для многотолщинных листовых тел в условиях, когда выбранной листовой грани соответствует несколько парных ей граней, находящихся на разном расстоянии от неё.

На рис. 2 изображена плоская листовая грань, найденная по противоположной ей грани и лежащему на ней контуру.

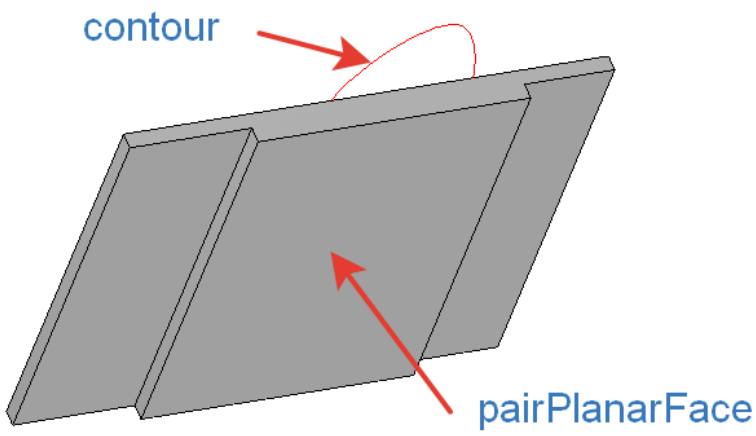


Рис. 2

Функция

```
MbFace * GetPairPlanarFace( const MbFaceShell * faceShell,
                            const MbFace &       sheetFace );
```

выполняет поиск парной плоской грани по заданной грани листового тела.

Входными параметрами функции являются:

- **faceShell** – оболочка листового тела.
- **sheetFace** – грань листового тела.

При удачной работе функция возвращает указатель на найденную грань, в противном случае – *NULL*.

Функция объявлена в файле *action_sheet.h*.

Поиск осуществляется сначала через рёбра внешнего цикла грани **sheetFace**, в случае неудачи – через вершины этого цикла, и если грань не найдена, то перебором по всем связным граням или граням множества **faceShell**. В последнем случае предпочтение отдаётся более близко расположенным граням.

На рис. 3 изображена плоская листовая грань **pairsheetFace**, найденная по противоположной ей плоской листовой грани **sheetFace**.

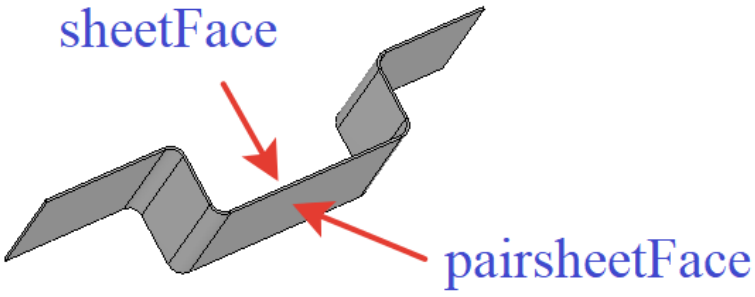


Рис. 3

3.7.9 Определение расстояния между подобными гранями

Функция

```
double GetDistanceIfSameAndOpposite( const MbFace & face1,
                                     const MbFace * face2 );
```

определяет расстояние между парой подобных граней листового тела.

Входными параметрами функции являются:

- **face1** – первая грань.
- **face2** – вторая грань.

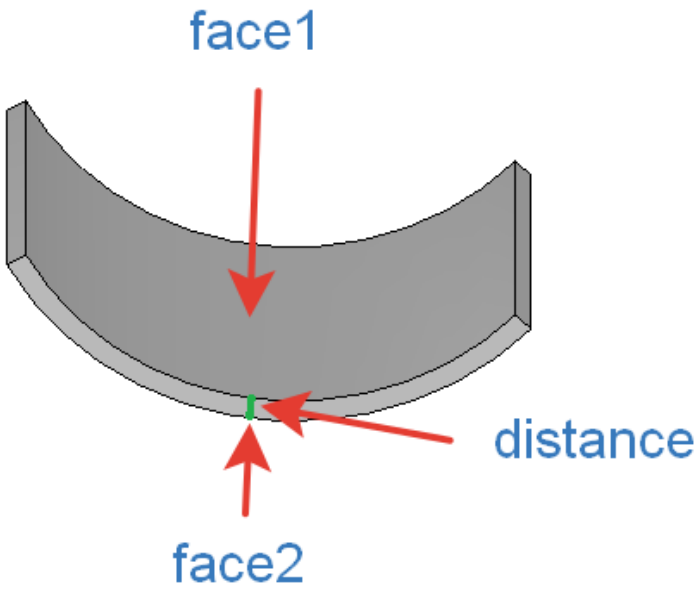
При удачной работе функция возвращает расстояние между гранями, в противном случае возвращает ноль.

Функция объявлена в файле *action_sheet.h*.

Подобными считаются пары плоских, цилиндрических и конических граней, а в случае линейчатого сгиба – линейчатые в паре с офсетными гранями, у которых нормали коллинеарны и противоположно направлены.

Расстояние считается положительным, если грань **face2** располагается со стороны, противоположной направлению нормали грани **face1**, и отрицательным – в противном случае.

На рисунке изображена пара подобных граней с расстоянием между ними:



3.7.10 Поиск подобных сгибов

Функция

```
void GetSimilarCylindricBends( const MbFaceShell &                shell,
                               RPArray<MbSheetMetalBend> & bends );
```

выполняет поиск подобных сгибов.

Входными параметрами функции являются оболочка листового тела **shell** и массив исходных сгибов **bends**, для которых надо найти подобные сгибы.

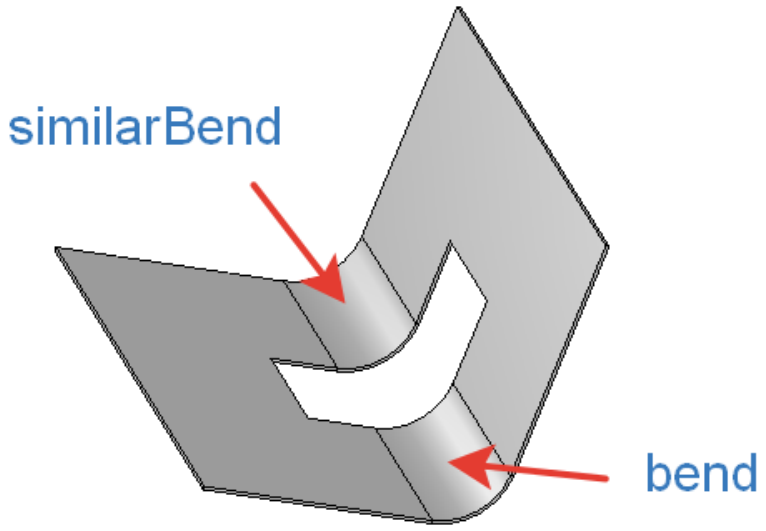
Выходным параметром функции является массив **bends**, дополненный найденными подобными сгибами.

Функция не возвращает значений.

Функция объявлена в файле *action_sheet.h*.

В множестве граней листового тела ищутся согнутые цилиндрические, конические или линейчатые сгибы, которые надо добавить к сгибам из **bends**, чтобы образовать группы сгибов, разгибаемых только совместно.

На рисунке изображена пара подобных сгибов, которые могут сгибаться или разгибаться только вместе:



3.7.11 Поиск касательной точки сгиба листового тела

Функция

```
bool CalculateTangentPoint( const MbFace &      face,
                            const MbPlane &     plane,
                                  MbCartPoint & tangentPoint );
```

выполняет расчёт касательной точки для сгиба/разгиба листового тела.

Входными параметрами функции являются:

- **face** – грань листового тела, содержащая точку касания.
- **plane** – касательная плоскость.
- **tangentPoint** – точка касания.

Выходным параметром функции является точка касания **tangentPoint**.

При удачной работе функция возвращает *true*, в противном случае возвращает *false*.

Функция объявлена в файле *action_sheet.h*.

Точка **tangentPoint** может находиться внутри грани (0.0 ≤ x ≤ 1.0 и 0.0 ≤ y ≤ 1.0) или за её пределами. В первом случае точка касания пересчитывается в координаты лежащей под гранью поверхности, во втором – находится одна из точек касания поверхности, лежащей под гранью **face**, и плоскости **plane**.

3.7.12 Поиск осевой линии сгиба

Функция

```
bool CalculateConicAxisLine( const MbFace &        face,
                                   MbLineSegment & axisLineSegment );
```

выполняет расчёт осевой линии разогнутого конического сгиба.

Входным параметром функции является грань **face** разогнутого конического сгиба листового тела.

Выходной параметр функции – осевая линия **axisLineSegment**.

При удачной работе функция возвращает *true*, в противном случае возвращает *false*.

Осевая линия **axisLineSegment** задаётся в координатах параметрической области плоской грани **face**.

Функция объявлена в файле *action_sheet.h*.

3.8 ВСПОМОГАТЕЛЬНЫЕ МЕТОДЫ

Для построения тел требуются параметры, которые в некоторых случаях нужно вычислить, а также вспомогательные элементы. В данной главе перечислены функции, выполняющие вспомогательные вычисления и построения.

3.8.1 Вычисление глубины тела выдавливания или угла тела вращения

Функция

```
bool GetSweptValue( const MbSweptData &   sweptData,
                    const MbAxis3D &      axis,
                    const MbVector3D &    direction,
                    const bool            rotation,
                    const bool            operationDirection,
                    const MbCartPoint3D & point,
                          double &        value );
```

вычисляет глубину выдавливания или угол вращения для последующего построения тела путем выдавливания или вращения образующей кривой.

Входными параметрами функции являются:

- **sweptData** – данные об образующей кривой;
- **axis** – ось вращения (для расчёта вращения при **rotation** == *true*);
- **direction** – направление выдавливания (для расчёта выдавливания при **rotation** == *false*);
- **rotation** – флаг выполнения расчёта: *true* для вращения, *false* для выдавливания;
- **operationDirection** – направление выполнения последующей операции: вперед (*true*), назад (*false*);
- **point** – точка, до которой требуется вращать или выдавливать образующую кривую.

Выходным параметром функции является **value** – глубина выдавливания или угол вращения.

При удачном выполнении расчёта функция возвращает *true*, в противном случае возвращает *false*.

Функция объявлена в файле *action_solid.h*.

Функция является вспомогательной для функций *ExtrusionSolid*, *RevolutionSolid*, *ExtrusionResult* и *RevolutionResult*.

3.8.2 Определение образа кривой для выдавливания или вращения

Функция

```
GetSweptImagePosition( const MbCurve3D &    generatrix,
                       const MbSurface &    surface,
                       const MbVector3D &   direction,
                       const MbAxis3D &     axis,
                       const bool           rotation,
                             MbCartPoint &  imagePosition,
                             MbResultType & resType );
```

вычисляет положение образа точки образующей кривой на поверхности для последующего построения тела путем выдавливания или вращения образующей кривой до заданной поверхности.

Входные параметры функции:

- **generatrix** – образующая кривая;
- **surface** – поверхность, до которой будет выполняться выдавливание или вращение кривой;
- **direction** – направление выдавливания (для расчёта выдавливания);
- **axis** – ось вращения (для расчёта вращения);
- **rotation** – флаг выполнения расчёта: для вращения (*true*), для выдавливания (*false*).

Выходные параметры функции:

- **imagePosition** – точка на поверхности, в которой лежит образ образующей кривой;
- **resType** – код результата операции: при удачной работе функция возвращает rt_Success, в противном случае возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция является вспомогательной для функций ExtrusionSolid(), RevolutionSolid(), ExtrusionResult() и RevolutionResult().

3.8.3 Определение параметров для выдавливания или вращения

Функция

```
void GetRangeToSurface( const MbSurface &    surface,
                        const MbCurve3D &    curve,
                        const MbVector3D &   direction,
                        const MbAxis3D &     axis,
                        const bool           rotation,
                        const bool           operationDirection,
                        const MbCartPoint &  imagePosition,
                              double         range,
                              MbRect &       rectOnSurface,
                              MbResultType & resType );
```

вычисляет глубины выдавливания в прямом и обратном направлениях или углы вращения в прямом и обратном направлениях для последующего построения тела путем выдавливания или вращения образующей кривой до заданной поверхности, а также габарит образа кривой.

Входными параметрами функции являются:

- **surface** – поверхность, до которой будет выполняться выдавливание или вращение кривой.
- **curve** – образующая кривая.
- **direction** – направление выдавливания (для расчёта выдавливания).
- **axis** – ось вращения (для расчёта вращения).
- **rotation** – флаг выполнения расчёта: *true* для вращения, *false* для выдавливания.
- **operationDirection** – направление выполнения последующей операции: вперед (*true*), назад (*false*).
- **imagePosition** – точка на поверхности, в которой лежит образ образующей кривой.

Выходными параметрами функции являются:

- **range** – расстояния до поверхности: range[0] – в обратном направлении, range[1] – в прямом направлении.
- **rectOnSurface** – габарит образа кривой на поверхности.
- **resType** – код результата операции: при удачной работе функция возвращает rt_Success, в противном случае функция возвращает код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция является вспомогательной для функций ExtrusionSolid(), RevolutionSolid(), ExtrusionResult() и RevolutionResult().

3.8.4 Определение ориентации образующей кривой

Функция

```
double AreaSign( const MbCurve3D &  curve,
                 const MbAxis3D &   axis,
                 const MbVector3D & direction,
                       bool         rotation )
```

вычисляет площадь проекции кривой на виртуальную координатную плоскость для определения ориентации образующей кривой при последующем построении тела путём выдавливания или вращения образующей кривой. Незамкнутая образующая кривая замыкается отрезком.

Входными параметрами функции являются:

- **curve** – образующая кривая;
- **axis** – ось вращения (для расчёта вращения);
- **direction** – направление выдавливания (для расчёта выдавливания);
- **rotation** – флаг выполнения расчёта: для вращения (*true*), для выдавливания (*false*).

Выходным параметром функции является площадь проекции кривой.

Функция является вспомогательной для функций ExtrusionSolid(), RevolutionSolid(), ExtrusionResult() и RevolutionResult().

3.8.5 Определение ориентации секущей поверхности

Функция

```
void AnalyzeSurfaceRelationToSweptOperation(const MbSurface &     cuttingSurface,
                                            const MbCartPoint &   imagePosition,
                                            const MbCurve3D &     curve,
                                            const MbVector3D &    direction,
                                            const MbAxis3D &      axis,
                                            const bool            rotation,
                                                  bool            operationDirection,
                                                  bool &          relativeSense,
                                                  MbResultType &  resType )
```

определяет ориентацию секущей поверхности относительно тела, которое будет строиться путём выдавливания или вращения образующей кривой до заданной поверхности.

Входными параметрами функции являются:

- **cuttingSurface** – секущая поверхность, относительно которой анализируется операция выдавливания/вращения;
- **imagePosition** – точка на поверхности, в которой лежит образ образующей кривой;
- **curve** – образующая кривая;
- **direction** – направление выдавливания (для расчёта выдавливания);
- **axis** – ось вращения (для расчёта вращения);
- **rotation** – флаг выполнения расчёта: для вращения (*true*), для выдавливания (*false*);
- **operationDirection** – направление выполнения последующей операции: вперёд (*true*), назад (*false*).

Выходными параметрами функции являются:

- **relativeSense** – ориентация поверхности для операции выдавливания или вращения;
- **resType** – код результата операции: при удачной работе функция возвращает rt_Success, в противном случае возвращается код ошибки из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция является вспомогательной для функций ExtrusionSolid(), RevolutionSolid(), ExtrusionResult() и RevolutionResult().

3.8.6 Ориентация кривых тела заметания

Функция

```
MbResultType EvolutionNormalize( const MbSurface &           surface,
                                 const RPArray<MbContour> &  contours,
                                 const MbCurve3D &           guide,
                                 const EvolutionValues &     parameters,
                                       MbAxis3D &            axis,
                                       double &              angle,
                                       VERSION               version,
                                       MbMatrix3D *          matrix = nullptr );
```

выполняет ориентацию образующего контура и направляющей кривой для построения кинематического тела. Также функция вычисляет матрицу преобразования сечения к ближайшему концу направляющей кривой.

Входными параметрами функции являются:

- **surface** – поверхность для кинематической операции;
- **contours** – образующие контуры;
- **guide** – направляющая кривая;
- **parameters** – параметры операции;
- **version** – версия операции.

Выходными параметрами функции являются:

- **axis** – ось доворота образующей (положительный — по правому правилу относительно оси **axis**);
- **angle** – угол доворота образующей;
- **matrix** – указатель на матрицу преобразования сечения к ближайшему концу направляющей кривой (если указатель равен *nullptr* – матрица не вычисляется).

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код результата из перечисления *MbResultType* (см. *mb_operation_result.h*).

Функция является вспомогательной для функций EvolutionSolid() и EvolutionResult().

3.8.7 Построение копии направляющей кривой тела заметания

Функция

```
MbCurve3D *TrimClosedSpine( const MbCurve3D & curve,
                                  double      t )
```

выполняет построение копии замкнутой кривой с началом в точке, определяемой параметром *t*.

Входными параметрами функции являются:

- **curve** — направляющая кривая;
- **t** — параметр кривой.

При удачной работе функция возвращает построенную копию кривой с началом в заданной точке, в противном случае возвращается ноль.

Функция является вспомогательной для функций EvolutionSolid() и EvolutionResult().

3.8.8 Построение ребра жёсткости

Функция

```
MbResultType RibElement( const MbSolid &                solid,
                         const MbRibSolidParameters &   pars,
                               MbSolid &                result )
```

выполняет построение ребра жёсткости для исходного тела.

Входными параметрами функции являются:

- **solid** – исходное тело;
- **params** – параметры ребра жёсткости.

Выходным параметром функции является построенное тело **result**.

При удачной работе функция возвращает rt_Success, в противном случае функция возвращает код результата из перечисления MbResultType (см. *mb_operation_result.h*).

Функция выполняет построение отдельного ребра жёсткости без объединения с исходным телом.

3.8.9 Проверка кривой для построения линейчатого тела

Функция

```
void CheckRuledCurve( const MbCurve3D &  curve0,
                      const MbCurve3D &  curve1,
                            bool &       isInverted,
                            bool &       isShifted,
                            VERSION      version )
```

проверяет вторую кривую на согласованность с первой кривой для построения линейчатого незамкнутого тела и определяет необходимость инверсии и/или сдвига начала второй кривой для корректного построения линейчатого тела.

Входными параметрами функции являются:

- **curve0** — первая кривая;
- **curve1** — вторая кривая;
- **version** — версия операции.

Выходными параметрами функции являются:

- **isInverted** — флаг, который на выходе становится равным *true*, если вторую кривую при построении следует использовать в обратном направлении относительно исходного;
- **isShifted** — флаг, который на выходе становится равным *true*, если начало параметризации второй кривой при построении следует сдвинуть (перепараметризовать) для согласования с первой кривой.

Возвращаемое значение у функции отсутствует.

Функция объявлена в файле *action_shell.h*.

Функция является вспомогательной для функции построения линейчатого незамкнутого тела по двум кривым RuledShell().

3.8.10 Проверка параметров кривой для построения линейчатого тела

Функция

```
bool CheckRuledParams( const MbCurve3D &        curve,
                       SArray<double> &  params,
                       bool              isAscending )
```

выполняет проверку параметров кривой и нормализацию параметров замкнутой кривой.

Входными параметрами функции являются:

- **curve** — кривая;
- **params** — множество параметров кривой;
- **isAscending** — флаг возрастания параметров множества.

Выходным параметром функции является множество параметров кривой **params**.

Если удалось нормализовать массив параметров, функция возвращает *true*, в противном случае возвращает *false*.

Функция объявлена в файле *action_shell.h*.

Функция является вспомогательной для функции построения линейчатого незамкнутого тела RuledShell().

3.8.11 Проверка кривой для построения тела соединения

Функция

```
void CheckJoinedCurve( const MbCurve3D &  curve1,
                       const MbCurve3D &  curve2,
                             bool &       isInverted1,
                             bool &       isShifted1,
                             VERSION      version )
```

проверяет вторую кривую на согласованность с первой кривой для построения незамкнутого тела соединения и определяет необходимость инверсии и/или сдвига начала второй кривой для корректного построения тела соединения.

Входными параметрами функции являются:

- **curve1** — первая кривая;
- **curve2** — вторая кривая;
- **version** — версия операции.

Выходными параметрами функции являются:

- **isInverted1** — флаг, который на выходе становится равным *true*, если вторую кривую при построении следует использовать в обратном направлении относительно исходного;
- **isShifted1** — флаг, который на выходе становится равным *true*, если начало параметризации второй кривой при построении следует сдвинуть (перепараметризовать) для согласования с первой кривой.

Возвращаемое значение у функции отсутствует.

Функция объявлена в файле *action_shell.h*.

Функция является вспомогательной для функции построения незамкнутого тела соединения по двум кривым JoinShell().

3.8.12 Проверка параметров кривой для построения тела соединения

Функция

```
bool CheckJoinedParams( const MbCurve3D &          curve,
                              SArray<double> &     params,
                              bool                 isAscending )
```

выполняет проверку корректности параметров кривой и при необходимости выполняет нормализацию параметров замкнутой (периодической) кривой к базовому диапазону параметров.

Входными параметрами функции являются:

- **curve** — кривая;
- **params** — множество параметров кривой;
- **isAscending** — флаг, задающий ожидаемый порядок параметров в массиве; при значении *true* параметры должны (и после нормализации остаются) упорядоченными по возрастанию.

Выходным параметром функции является множество параметров кривой **params**, которое при необходимости модифицируется (нормализуется и, при задании флага возрастания, переупорядочивается).

При успешной проверке и (при необходимости) нормализации массива параметров функция возвращает *true*; если параметры некорректны и не могут быть нормализованы, функция возвращает *false*.

Функция объявлена в файле *action_shell.h*.

Функция является вспомогательной для функции построения незамкнутого тела JoinShell().

3.8.13 Построение кривой по множеству рёбер

Функция

```
template <typename ConstEdgeVector, typename OrientsVector>
MbCurve3D * CreateJoinedCurve( const ConstEdgeVector &  edges,
                               const OrientsVector &    orients,
                               const MbMatrix3D &       matr,
                                     MbResultType &     res );
```

выполняет построение кривой по набору рёбер, лежащих на поверхности соединения, для последующего использования в операции JoinShell().

Входными параметрами функции являются:

- **edges** — множество рёбер (контейнер константных указателей на рёбра);
- **orients** — контейнер булевых флагов, задающих направления использования соответствующих рёбер из **edges** (размер контейнера должен совпадать с количеством рёбер);
- **matrix** — матрица преобразования рёбер (в которую предварительно трансформируются рёбра перед построением кривой).

Выходным параметром функции является значение **res** из перечисления MbResultType.

При удачной работе функция возвращает указатель на построенную кривую; в противном случае возвращает нулевой указатель.

Функция объявлена в файле *action_shell.h*.

3.8.14 Определение точек скрещения кривых

Функция

```
ptrdiff_t CurveCurveCrossing( const MbCurve3D &       curve1,
                              const MbCurve3D &       curve2,
                                    SArray<double> &  result1,
                                    SArray<double> &  result2,
                                    double            epsilon = Math::metricRegion );
```

находит точки скрещения двух пространственных кривых и возвращает параметры этих точек на каждой кривой.

Точками скрещения называются такие точки кривых, для которых отрезок, соединяющий эти точки, перпендикулярен касательным к обеим кривым.

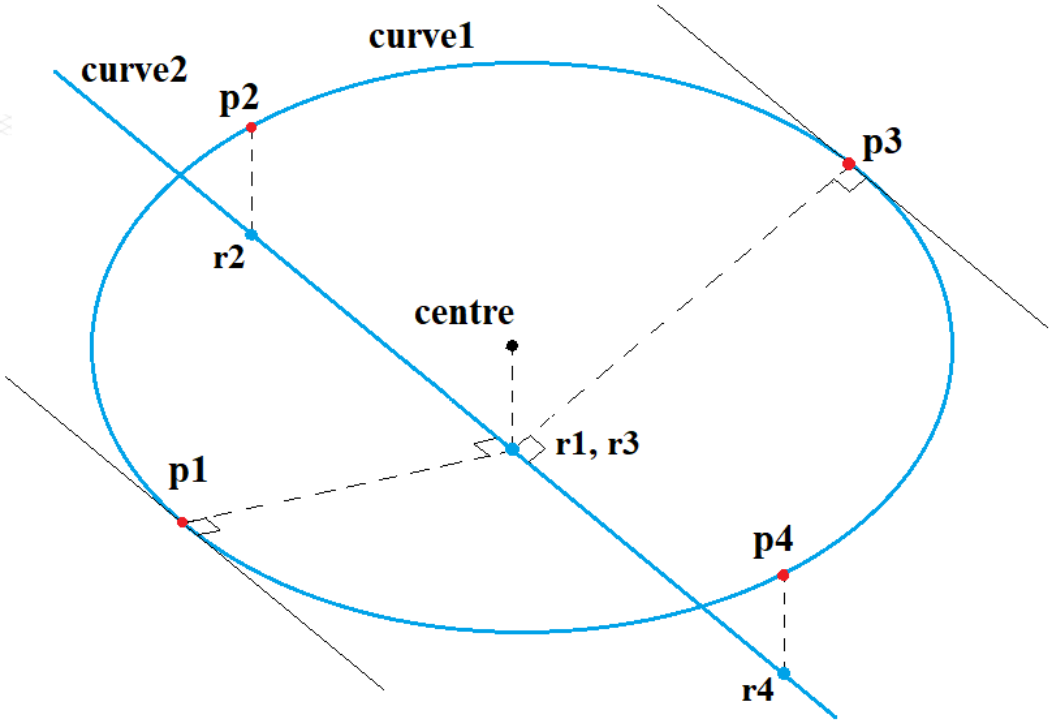
Параметры функции:

- **curve1** – первая кривая;
- **curve2** – вторая кривая;
- **result1** – параметры точек скрещения на первой кривой;
- **result2** – параметры точек скрещения на второй кривой;
- **epsilon** – точность поиска решения.

Возвращаемое значение – количество найденных скрещений.

Параметры найденных точек скрещения для первой кривой записываются в массив **result1**, а для второй кривой – в массив **result2**. В частном случае геометрического пересечения кривых отрезок между точками скрещения вырождается в нулевую длину, и точки скрещения становятся точками пересечения. Помимо собственных скрещений, функция CurveCurveCrossing() возвращает такие точки скрещения кривых, для которых отрезок, соединяющий эти точки, перпендикулярен касательным к кривым в этих точках.

Например, на рисунке функция CurveCurveCrossing() возвращает четыре пары параметров скрещения: на окружности **curve1** им соответствуют точки p1, p2, p3, p4, а на отрезке **curve2** точки r1, r2, r3, r4:



3.9 ЭЛЕМЕНТАРНЫЕ ОБЪЕКТЫ

К элементарным объектам относятся объекты геометрического ядра C3D, которые описывают такие математические сущности, как вектор, точка, ось, локальная система координат, матрица преобразования, габаритный параллелепипед и габаритный прямоугольник. Элементарные объекты имеют простые структуры данных, служат инструментом и строительным материалом для более сложных геометрических объектов и используются всеми модулями геометрического ядра.

3.9.1 Вектор в трёхмерном пространстве MbVector3D

Класс MbVector3D объявлен в файле *mb_vector3d.h*.

Класс поддерживает базовые векторные операции (сложение, вычитание, умножение на скаляр, скалярное и векторное произведения, нормализацию) и служит базовым строительным блоком для описания направлений, нормалей и смещений в трёхмерном пространстве.

Вектор MbVector3D описывает перемещение или направление в трёхмерном пространстве и определяется тремя компонентами *x*, *y*, *z* в декартовой системе координат.

Для обозначения векторов в трёхмерном пространстве будем использовать одну или несколько строчных букв латинского алфавита, выделенных полужирным шрифтом, а для объединения компонент вектора будем заключать их в квадратные скобки, например,

$$\mathbf{v} = [x, y, z]. \tag{3.1}$$

Вектор MbVector3D не привязан к точкам пространства и поэтому не имеет метода, перемещающего его в пространстве.

3.9.2 Радиус-вектор точки в трёхмерном пространстве MbCartPoint3D

Класс MbCartPoint3D объявлен в файле *mb_cart_point3d.h*.

Радиус-вектор MbCartPoint3D (картезианская точка) описывает положение точки в трёхмерном пространстве и определяется тремя координатами *x*, *y*, *z* в декартовой системе координат. Радиус-вектор описывает преобразование, переводящее начальную точку декартовой системы координат в точку пространства с заданными координатами в этой декартовой системе координат.

Для обозначения точек в трёхмерном пространстве будем использовать одну или несколько строчных букв латинского алфавита, выделенных полужирным шрифтом, а для объединения координат точки будем заключать их в квадратные скобки, например,

$$\mathbf{p} = [x, y, z]. \tag{3.2}$$

Радиус-вектор MbCartPoint3D в отличие от вектора MbVector3D привязан к началу выбранной системы координат. Координаты радиуса-вектора MbCartPoint3D и компоненты вектора *MbVector3D* по-разному изменяются при переходе от одной системы координат к другой и при изменении положения в пространстве с помощью методов:

MbCartPoint3D & Transform(const MbMatrix3D &),

MbCartPoint3D & Rotate(const MbAxis3D &, double angle),

MbCartPoint3D & Move(const MbVector3D &).

Перечисленные методы возвращают ссылку на себя после преобразования.

3.9.3 Расширенный вектор в трёхмерном пространстве MbHomogeneous3D

Класс MbHomogeneous3D объявлен в файле *mb_homogenius3d.h*.

Расширенный радиус-вектор MbHomogeneous3D описывает положение точки в трёхмерном пространстве и определяется четырьмя координатами *x*, *y*, *z*, *w*. Четвёртая координата называется весом. Расширенный радиус-вектор используется при вычислении радиуса-вектора *B*-кривых и *B*-поверхностей, построенных на основе *B*-сплайнов. Координаты x_w, y_w, z_w, *w* расширенного радиуса-вектора MbHomogeneous3D связаны с координатами *x*, *y*, *z* радиуса-вектора соотношениями:

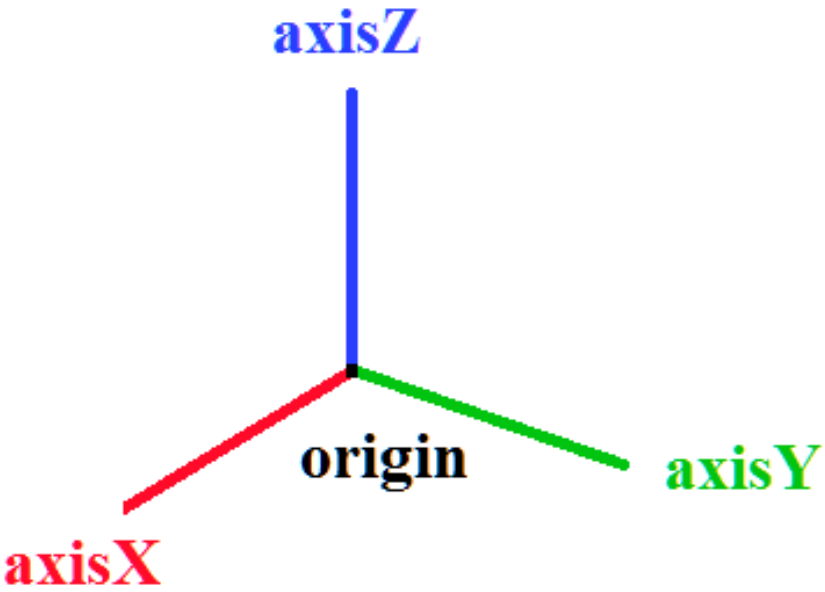
$$x = \frac{x_w}{w}; \quad y = \frac{y_w}{w}; \quad z = \frac{z_w}{w}.$$

В операциях умножения на расширенную матрицу *MbMatrix3D* можно считать, что векторы и точки также имеют четвёртую координату, для вектора *MbVector3D* она равна нулю, а для радиуса-вектора *MbCartPoint3D* она равна единице.

3.9.4 Локальная система координат MbPlacement3D

Класс MbPlacement3D объявлен в файле *mb_placement3d.h*.

Локальная система координат в трёхмерном пространстве MbPlacement3D описывается начальной точкой **origin** и тремя некомпланарными векторами **axisX**, **axisY**, **axisZ**:



В большинстве случаев система координат является правой, а векторы системы ортонормированны. С помощью преобразований система координат может стать левой и не ортонормированной. Состояние локальной системы координат можно запросить с помощью методов:

- bool IsLeft() — является ли система координат левой;
- bool IsRight() — является ли система координат правой;
- bool IsTranslation() — присутствует ли сдвиг начала **origin** системы координат;
- bool IsRotation() — присутствует ли поворот системы координат относительно глобальной;
- bool IsOrt() — является ли система координат ортогональной, но ненормированной;
- bool IsSingle() — совпадает ли система координат с мировой СК;
- bool IsNormal() — является ли система координат ортонормированной;
- bool IsOrthogonal() — является ли система координат ортогональной;
- bool IsCircular() — является ли система координат ортогональной с равными по длине осями **axisX**, **axisY**, в которой окружность остается окружностью;
- bool IsIsotropic() — является ли система координат ортогональной с равными по длине осями **axisX**, **axisY**, **axisZ**, в которой объекты не искажаются, а только масштабируются;
- bool IsAffine() — является ли система координат афинной (допускает сдвиг, масштабирование и наклон осей).

Локальная система координат является декартовой. Точка в декартовой системе координат определяется тремя координатами *x*, *y*, *z*. Локальная система может выступать в роли цилиндрической или сферической системы координат.

Локальная система координат MbPlacement3D может использоваться для задания цилиндрической системы координат. В этом случае начало цилиндрической системы совпадает

с началом декартовой, ось Oz совпадает с осью Z декартовой системы, а основная плоскость совпадает с плоскостью XY.

Точка в цилиндрической системе координат определяется тремя координатами *r*, *f*, *z*:

- *r*, *f* являются полярными координатами проекции точки на основную плоскость;
- *r* задаёт длину проекции радиуса-вектора;
- *f* задаёт полярный угол проекции радиуса-вектора, *z* задаёт расстояние от точки до основной плоскости.

При использовании локальной системы координат MbPlacement3D в роли сферической системы координат точка описывается тремя координатами *r*, *f*, *w*:

- *r* задаёт расстояние от начала координат до точки (длину радиуса-вектора);
- *f* задаёт угол между проекцией радиуса-вектора на плоскость и лучом в этой плоскости (долготу);
- *w* задаёт угол между радиус-вектором и нормалью к плоскости сферической системы координат (полярное расстояние).

Плоскость сферической системы координат совпадает с плоскостью XY декартовой системы, а долгота *f* отсчитывается от оси X в сторону положительного направления оси Y.

3.9.5 Расширенная матрица в трёхмерном пространстве MbMatrix3D

Класс MbMatrix3D объявлен в файле *mb_matrix3d.h*.

В трёхмерном пространстве матрица MbMatrix3D описывает преобразование из одной системы координат в другую. Матрица имеет размерность четыре на четыре. Пусть в выбранной системе координат построена локальная аффинная система координат с началом в точке **r** с координатами r_1, r_2, r_3 и базисными векторами **a** = [a_1 a_2 a_3], **b** = [b_1 b_2 b_3], **c** = [c_1 c_2 c_3]. Векторы **a**, **b**, **c** должны быть линейно независимыми, но могут быть не ортогональными друг другу и иметь произвольную длину. Матрица MbMatrix3D преобразования координат из локальной системы в выбранную систему координат имеет вид:

$$\mathbf{M} = \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

Для обозначения расширенных матриц в трёхмерном пространстве будем использовать прописные буквы латинского алфавита, выделенные полужирным шрифтом, например **M**. Заметим, что каждому базисному вектору **a**, **b**, **c** и начальной точке **r** локальной системы координат соответствует своя строка в матрице преобразования из локальной системы в выбранную систему координат.

Матрица MbMatrix3D является расширенной и работает с однородными радиусами-векторами и однородными векторами MbHomogeneous3D в трёхмерном пространстве.

При преобразовании радиуса-вектора *MbCartPoint3D* по матрице MbMatrix3D точке следует добавить четвёртую координату, равную единице. Пусть точка с координатами x_1, x_2, x_3 в локальной системе координат имеет координаты p_1, p_2, p_3 в выбранной системе координат, тогда при участии расширенной матрицы MbMatrix3D координаты будут связаны соотношением

$$\begin{bmatrix} p_1 & p_2 & p_3 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

Заметим, что трёхмерный радиус-вектор умножается на расширенную матрицу MbMatrix3D справа.

При преобразовании вектора *MbVector3D* по матрице MbMatrix3D вектору следует добавить четвёртую компоненту, равную нулю. Пусть вектор с компонентами y_1, y_2, y_3 в локальной системе координат имеет компоненты v_1, v_2, v_3 в выбранной системе координат, тогда при участии расширенной матрицы MbMatrix3D компоненты будут связаны соотношением

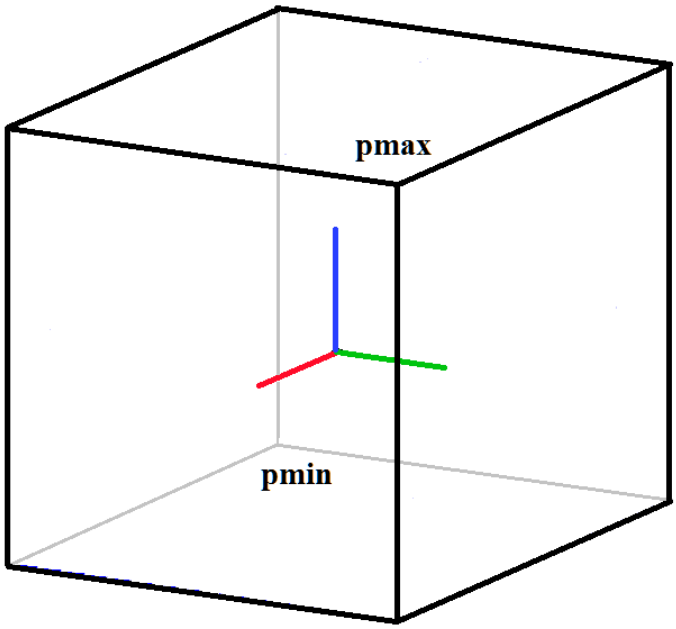
$$\begin{bmatrix} v_1 & v_2 & v_3 & 0 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & y_3 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ r_1 & r_2 & r_3 & 1 \end{bmatrix}.$$

Заметим, что трёхмерный вектор умножается на расширенную матрицу MbMatrix3D справа.

3.9.6 Габаритный параллелепипед в трёхмерном пространстве MbCube

Класс MbCube объявлен в файле *mb_cube.h*.

Габаритный параллелепипед MbCube описывает габариты протяжённого объекта (кривой, поверхности, тела, нескольких тел) в трёхмерном пространстве и определяется двумя точками **pmin** и **pmax**. Грани габаритного параллелепипеда параллельны плоскостям системы координат, в которой описан куб. Точки **pmin** и **pmax** описывают две противоположные вершины габаритного куба, имеющие наименьшие и наибольшие координаты, соответственно:



Если габариты протяжённого объекта не определены, то габаритный параллелепипед считается пустым и **pmin** = [10^{-300} 10^{-300} 10^{-300}], **pmax** =

= [-10^{-300} -10^{-300} -10^{-300}]. Для пустого габаритного куба **pmin** > **pmax** и метод IsEmpty() возвращает *true*.

3.9.7 Одномерный габарит MbRect1D

Класс MbRect1D объявлен в файле *mb_rect1d.h*.

Одномерный габарит MbRect1D описывает одномерную область, например, область определения параметра кривой и определяется двумя числами: *zmin* и *zmax*:



Числа *zmin* и *zmax* описывают начальный и конечный край области. Если одномерная область не определена, то одномерный габарит считается пустым и *zmin* > *zmax*. Для пустого одномерного габарита метод IsEmpty() возвращает *true*.

3.9.8 Вектор в двумерном пространстве MbVector

Класс MbVector объявлен в файле *mb_vector.h*.

Вектор MbVector описывает перемещение или направление в двумерном пространстве и определяется двумя компонентами *x* и *y* в декартовой системе координат.

Для обозначения векторов в двумерном пространстве будем использовать одну или несколько строчных букв латинского алфавита, выделенных полужирным наклонным шрифтом, а для объединения компонент вектора будем заключать их в квадратные скобки, например,

$$\mathbf{v} = [x, y]. \tag{3.3}$$

Вектор MbVector не привязан к точкам пространства и поэтому не имеет метода, перемещающего его в пространстве.

3.9.9 Нормализованный вектор в двумерном пространстве MbDirection

Класс MbDirection объявлен в файле *mb_vector.h*.

Нормализованный вектор MbDirection описывает направление или угол поворота в двумерном пространстве и определяется двумя компонентами *ax* и *ay* в декартовой системе координат. Длина нормализованного вектора равна единице, а его компоненты равны соответственно синусу и косинусу угла между осью OX и нормализованным вектором. Таким образом, *ax* = cos(□), *ay* = sin(□), где □ – угол между нормализованным вектором и осью абсцисс системы координат.

3.9.10 Радиус-вектор точки в двумерном пространстве MbCartPoint

Класс MbCartPoint объявлен в файле *mb_cart_point.h*.

Радиус-вектор MbCartPoint (картезианская точка) описывает положение точки в двумерном пространстве и определяется двумя координатами *x* и *y* в декартовой системе координат. Радиус-вектор описывает преобразование, переводящее начальную точку декар-

товой системы координат в точку пространства с заданными координатами в этой декартовой системе координат.

Для обозначения точек в двумерном пространстве будем использовать одну или несколько строчных букв латинского алфавита, выделенных полужирным наклонным шрифтом, а для объединения координат точки будем заключать их в квадратные скобки, например,

$$\mathbf{point} = [x\ y]. \tag{3.4}$$

Радиус-вектор в отличие от вектора связан с началом координат. Координаты радиуса-вектора MbCartPoint и компоненты вектора *MbVector* по-разному изменяются при переходе от одной системы координат к другой системе координат и при изменениях положения в пространстве, реализованных в методах:

void Transform(const *MbMatrix* &);

void Rotate(const MbCartPoint &, double angle);

void Move(const *MbVector* &).

3.9.11 Расширенный вектор в двумерном пространстве MbHomogeneous

Класс MbHomogeneous объявлен в файле *mb_homogeneous.h*.

Расширенный радиус-вектор MbHomogeneous описывает положение точки в двумерном пространстве и определяется тремя координатами *x*, *y*, *w*. Третья координата называется весом.

Расширенный радиус-вектор используется при вычислении радиуса-вектора двумерных *B*-кривых, построенных на основе *B*-сплайнов. Координаты x_w, y_w, w расширенного радиуса-вектора MbHomogeneous связаны с координатами *x*, *y* радиуса-вектора *MbCartPoint* соотношениями:

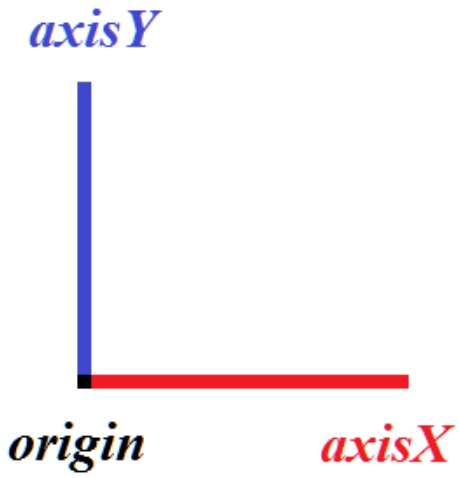
$$x = \frac{x_w}{w}; \quad y = \frac{y_w}{w}.$$

В операциях умножения на расширенную матрицу *MbMatrix* можно считать, что двумерные векторы и точки также имеют третью координату, для вектора *MbVector* она равна нулю, а для радиуса-вектора MbCartPoint она равна единице.

3.9.12 Локальная система координат MbPlacement

Класс MbPlacement объявлен в файле *mb_placement.h*.

Локальная система координат MbPlacement используется для задания положения и ориентации объектов в двумерном пространстве и для построения матриц преобразований MbMatrix. Локальная система координат в двумерном пространстве MbPlacement описывается начальной точкой **origin** и двумя непараллельными векторами **axisX** и **axisY**:



В большинстве случаев система координат является правой, а векторы системы ортонормированны. С помощью преобразований система координат может стать левой и не ортонормированной. Состояние локальной системы координат можно запросить с помощью методов:

bool IsLeft() – является ли система координат левой;

bool IsSingle() – совпадает ли система координат с той, в которой она задана;

bool IsNormal() – является ли система координат ортонормированной;

bool IsCircular() – является ли система координат ортогональной с равными по длине осями **axisX**, **axisY**, в которой окружность остается окружностью;

bool IsIsotropic() – является ли система координат ортогональной с равными по длине осями **axisX**, **axisY**, в которой объекты не искажаются, а только масштабируются;

bool IsAffine() – является ли система координат афинной (в противном случае система является ортонормированной).

Локальная система координат является декартовой.

3.9.13 Расширенная матрица в двумерном пространстве MbMatrix

Класс MbMatrix объявлен в файле *mb_matrix.h*.

В двумерном пространстве матрица MbMatrix3D описывает преобразование из одной системы координат в другую. Матрица имеет размерность три на три. Пусть в выбранной системе координат построена локальная аффинная система координат с началом в точке r с координатами r_1, r_2 и базисными векторами a = [a_1 a_2], b = [b_1 b_2]. Векторы a и b не должны быть коллинеарными, но могут быть не ортогональными друг другу и иметь произвольную длину. Матрица MbMatrix преобразования координат из локальной системы в выбранную систему координат имеет вид:

$$\mathrm{M} = \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}.$$

Для обозначения расширенных матриц в двумерном пространстве будем использовать прописные буквы латинского алфавита, выделенные полужирным наклонным шрифтом, например M. Заметим, что каждому базисному вектору a, b и начальной точке r локальной

системы координат соответствует своя строка в матрице преобразования из локальной системы в выбранную систему координат.

Матрица MbMatrix является расширенной и работает с однородными радиусами-векторами и однородными векторами *MbHomogeneous* в двумерном пространстве.

При преобразовании радиуса-вектора *MbCartPoint* по матрице MbMatrix точке следует добавить третью координату, равную единице. Пусть точка с координатами x_1, x_2 в локальной системе координат имеет координаты p_1, p_2 в выбранной системе координат, тогда при участии расширенной матрицы MbMatrix координаты будут связаны соотношением

$$\begin{bmatrix} p_1 & p_2 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}$$

Заметим, что двумерный радиус-вектор умножается на расширенную матрицу MbMatrix справа.

При преобразовании вектора MbVector по матрице MbMatrix вектору следует добавить третью компоненту, равную нулю. Пусть вектор с компонентами y_1, y_2 в локальной системе координат имеет компоненты r_1, r_2 в выбранной системе координат, тогда при участии расширенной матрицы MbMatrix компоненты будут связаны соотношением

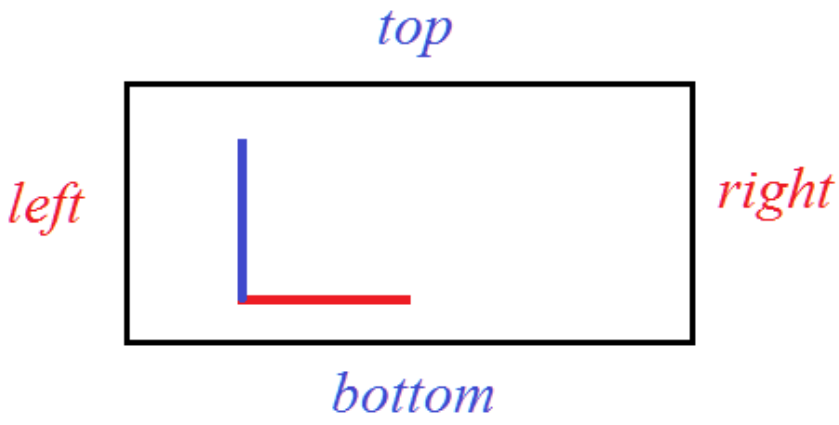
$$\begin{bmatrix} r_1 & r_2 & 0 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \\ r_1 & r_2 & 1 \end{bmatrix}.$$

Заметим, что двумерный вектор умножается на расширенную матрицу MbMatrix справа.

3.9.14 Габаритный прямоугольник в двумерном пространстве MbRect

Класс MbRect объявлен в файле *mb_rect.h*.

Габаритный прямоугольник MbRect описывает габариты протяжённого объекта (кривой, нескольких кривых) в двумерном пространстве и определяется четырьмя числами: *left*, *right*, *bottom*, *top*. Стороны габаритного прямоугольника параллельны осям системы координат, в которой описан прямоугольник. Числа *left* и *right* описывают минимальную и максимальную абсциссу габаритного прямоугольника, а числа *bottom* и *top* описывают минимальную и максимальную ординату габаритного прямоугольника:



Если габариты протяжённого объекта не определены, то габаритный прямоугольник считается пустым, если одновременно выполняются неравенства *left* > *right* и *bottom* > *top*. Для пустого габаритного прямоугольника метод IsEmpty() возвращает *true*.

3.10 ГЕОМЕТРИЧЕСКИЕ ОБЪЕКТЫ

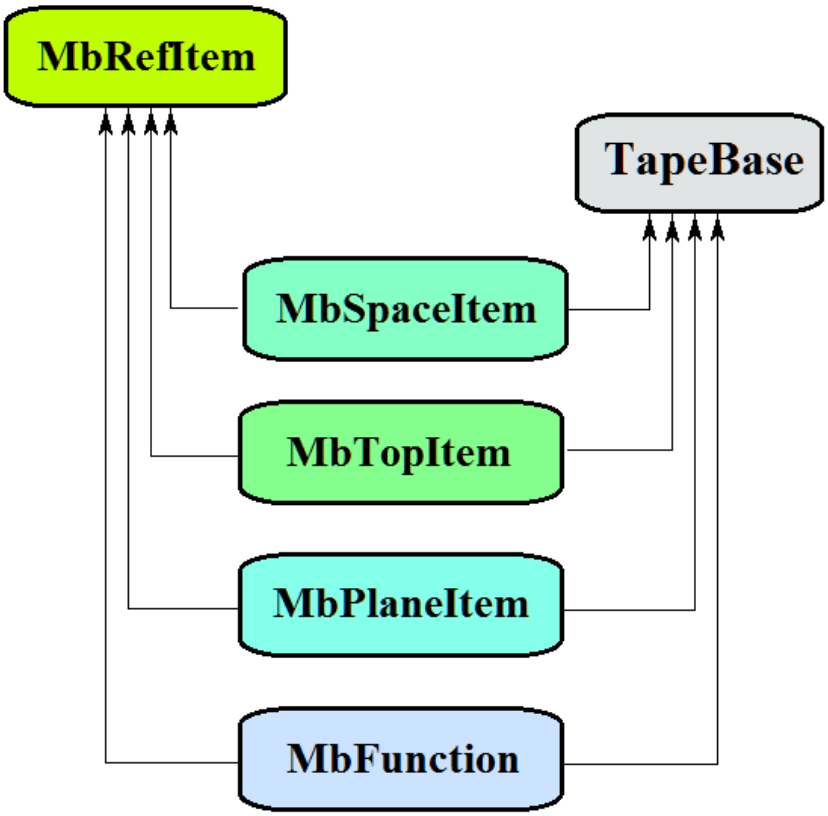
Геометрические объекты описывают форму моделируемых объектов. К геометрическим объектам относятся кривые, поверхности, тела, а также топологические объекты, описывающие геометрические свойства, не зависящие от количественных характеристик и характеризующие непрерывную связь точек трёхмерного пространства друг с другом. Геометрические объекты могут быть трёхмерные и двумерные. Двумерные объекты используются для работы в области определения параметров поверхностей и для работы в плоскостях трёхмерных локальных систем координат. В данном разделе описаны родительские классы геометрических объектов.

3.10.1 Счётчик ссылок MbRefItem

Класс MbRefItem объявлен в файле *reference_item.h*.

Класс MbRefItem описывается количеством своих владельцев *useCount* и представляет собой счётчик объектов, владеющих данным объектом.

Все геометрические объекты ядра C3D делятся на три группы: двумерные геометрические объекты, трёхмерные геометрические объекты и топологические объекты. Все геометрические объекты являются наследниками классов MbRefItem и TapeBase:



Класс TapeBase обеспечивает для своих наследников запись в поток и чтение из потока.

Наследниками классов MbRefItem и TapeBase являются следующие объекты:

MbSpaceItem – базовый абстрактный класс трёхмерных геометрических объектов;

MbTopItem – базовый абстрактный класс топологических объектов;

MbPlaneItem – базовый абстрактный класс двумерных геометрических объектов;

MbFunction – базовый абстрактный класс скалярных функций.

Счётчик ссылок обеспечивает корректную работу классов и методов, содержащих указатели на геометрические объекты. Если некоторый класс в своих данных содержит указатель на геометрический объект, то он обязан в конструкторе увеличить счётчик ссылок у геометрического объекта на единицу методом AddRef(), а в деструкторе должен вызвать у геометрического объекта метод Release(), который уменьшает счётчик ссылок геометрического объекта на единицу и, при достижении счётчиком ссылок нуля, удаляет геометрический объект.

Метод DecRef() уменьшает счётчик ссылок геометрического объекта на единицу.

Класс MbRefItem обрабатывается регистратором дублирования MbRegDuplicate и регистратором преобразования MbRegTransform.

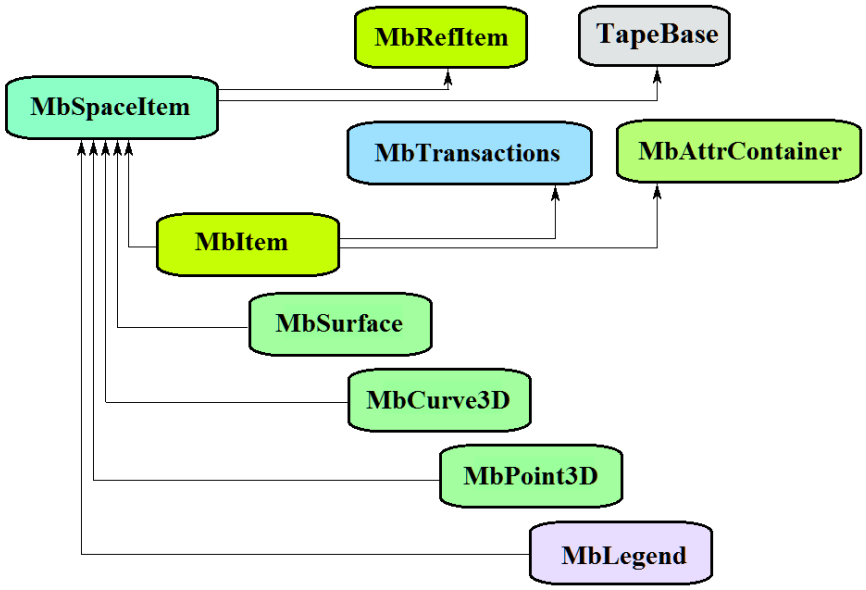
Метод RefType() возвращает регистрационный тип объекта, использующего счётчик ссылок.

3.10.2 Трёхмерный геометрический объект MbSpaceItem

Класс MbSpaceItem объявлен в файле *space_item.h*.

Класс MbSpaceItem является наследником классов *MbRefItem*, TapeBase и родительским классом для трёхмерных геометрических объектов.

К трёхмерным геометрическим объектам ядра C3D относятся: точка, кривые, поверхности, вспомогательные объекты и объекты геометрической модели:



Наследниками класса MbSpaceItem являются следующие семейства трёхмерных геометрических объектов:

MbPoint3D – точка;

MbCurve3D – кривая;

MbSurface – поверхность;

MbLegend – вспомогательный геометрический объект;

MbItem – объект геометрической модели.

Основными методами трёхмерных геометрических объектов являются:

```
virtual void Move( const MbVector3D &     to,
                         MbRegTransform * iReg = nullptr );

virtual void Rotate( const MbAxis3D &       axis,
                           double           angle,
                           MbRegTransform * iReg = nullptr );

virtual void Transform( const MbMatrix3D &     matr,
                              MbRegTransform * iReg = nullptr );
```

обслуживающие преобразование геометрического объекта. При преобразовании используется регистратор MbRegTransform для предотвращения многократного преобра-

зования вложенных объектов. Если объект содержит указатели или ссылки на другие объекты, то вложенные объекты так же трансформируются. Регистратор необходимо использовать, если надо последовательно преобразовать несколько взаимосвязанных объектов, связь которых обусловлена наличием в них указателей или ссылок на общие объекты. При преобразовании без регистратора можно получить многократное преобразование общих вложенных объектов.

Кроме того, все геометрические объекты имеют методы, обеспечивающие копирование, проверку на совпадение, проверку на возможность сделать совпадающими и делающие объекты совпадающими:

```
// Создать копию объекта с использованием регистратора
MbSpaceItem & Duplicate( MbRegDuplicate * iReg = nullptr );

// Определить, являются ли объекты равными
virtual bool IsSame( const MbSpaceItem & other,
                           double        accuracy = LENGTH_EPSILON );

// Определить, являются ли объекты подобными
virtual bool IsSimilar( const MbSpaceItem & item );

// Сделать объекты равными, если они подобны
virtual bool SetEqual( const MbSpaceItem & item );
```

При создании копии используется регистратор MbRegDuplicate для предотвращения многократного копирования вложенных объектов. Если объект содержит указатели или ссылки на другие объекты, то вложенные объекты так же копируются. Регистратор необходимо использовать, если надо последовательно копировать несколько взаимосвязанных объектов, связь которых обусловлена наличием в них указателей или ссылок на общие объекты. При копировании без регистратора можно получить набор копий одного и того же вложенного объекта вместо одной его копии.

Для идентификации типа геометрические объекты снабжены методами

```
MbeSpaceType IsA();
MbeSpaceType Type();
MbeSpaceType Family();
```

возвращающими тип из перечисления трёхмерных геометрических объектов.

Методы

```
MbProperty & CreateProperty( MbePrompt name );

void GetProperties( MbProperties & properties );

void SetProperties( MbProperties & properties );
```

обеспечивают выдачу и редактирование внутренних данных геометрических объектов.

Метод GetProperties() добавляет к множеству **properties** данные объекта в виде наследников класса MbProperty.

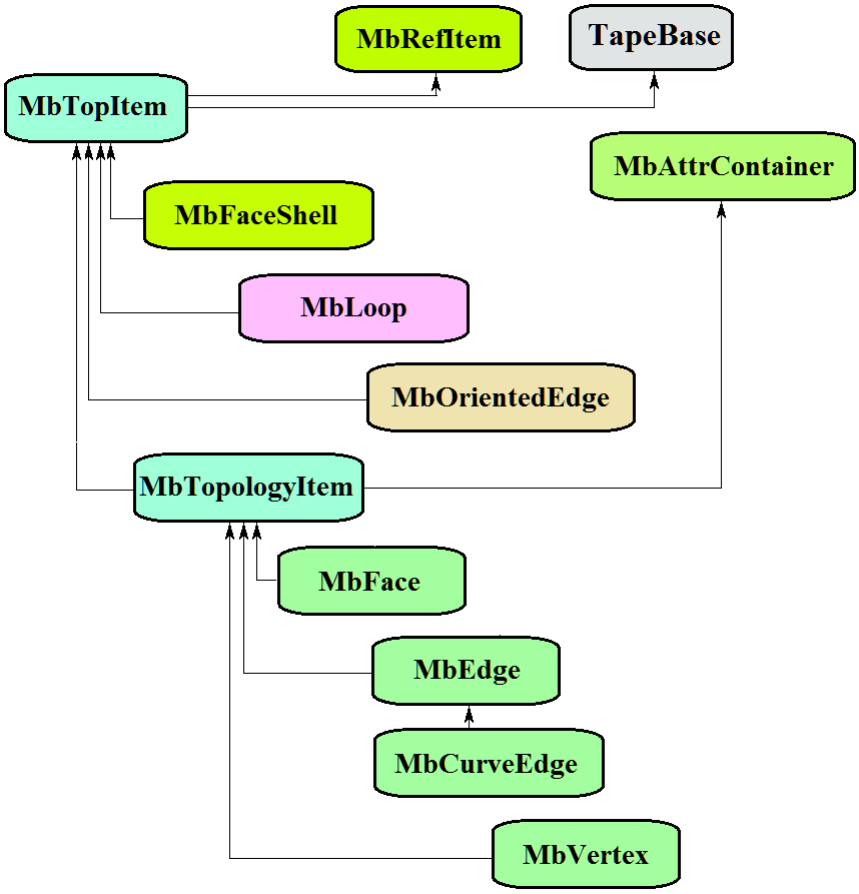
Метод CalculateWire(double sag, *MbMesh* & **mesh**) строит полигональную копию геометрического объекта, которая используется для визуализации.

3.10.3 Топологический объект MbTopItem

Класс MbTopItem объявлен в файле *topology_item.h*.

Класс MbTopItem является наследником классов *MbRefItem*, TapeBase и родительским классом для топологических объектов. Среди топологических объектов выделен класс именованных топологических объектов *MbTopologyItem*, который является наследником классов MbTopItem и MbAttributeContainer. Заменитель топологического объекта MbTopologyProxy является наследником MbTopologyItem и используется в алгоритмах именования. Класс *MbTopologyItem* объявлен также в файле *topology_item.h*.

Геометрическое ядро C3D оперирует топологическими объектами, которые приведены на рисунке:



Наследниками класса MbTopItem являются следующие топологические объекты:

MbFaceShell – множество граней;

MbLoop – цикл рёбер границы грани;

MbOrientedEdge – ориентированное ребро цикла;

MbTopologyItem – именованный топологический объект.

Наследниками именованного топологического объекта MbTopologyItem являются следующие объекты:

MbVertex – вершина;

MbEdge – ребро;

MbFace – грань.

Ребро имеет наследника *MbCurveEdge*, описывающее гладкий участок стыковки двух граней или край грани.

Работу именованных топологических объектов с атрибутами обеспечивает контейнер атрибутов MbAttributeContainer.

Основными методами именованных топологических объектов являются:

```
virtual void Transform( const MbMatrix3D &     matr,
                              MbRegTransform * iReg = nullptr );
virtual void Move( const MbVector3D &     to,
                         MbRegTransform * iReg = nullptr );
virtual void Rotate( const MbAxis3D &       axis,
                           double           angle,
                           MbRegTransform * iReg = nullptr );
```

обслуживающие преобразование топологического объекта и методы работы с именем и атрибутами. При преобразовании используется регистратор MbRegTransform для предотвращения многократного преобразования вложенных объектов. Если объект содержит указатели или ссылки на другие объекты, то вложенные объекты так же трансформируются. Регистратор необходимо использовать, если надо последовательно преобразовать несколько взаимосвязанных объектов, связь которых обусловлена наличием в них указателей или ссылок на общие объекты. При преобразовании без регистратора можно получить многократное преобразование общих вложенных объектов.

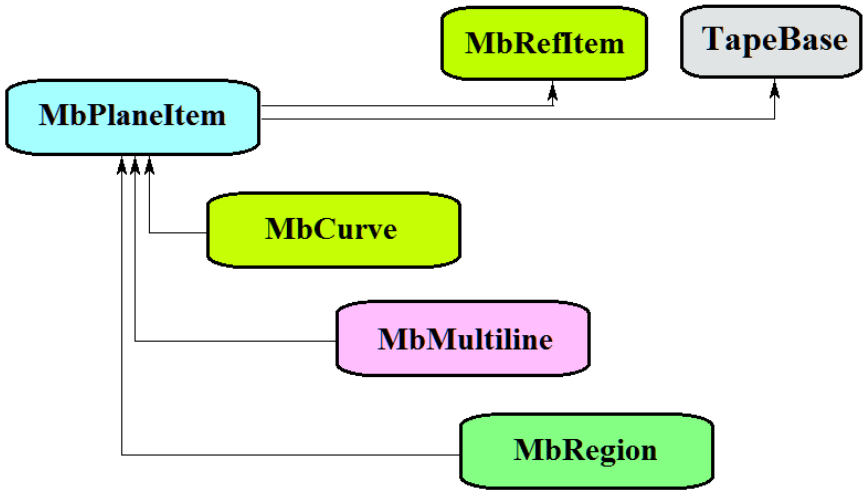
Для идентификации типа топологические объекты снабжены методом **IsA**(), возвращающим тип из перечисления топологических объектов MbeTopologyType.

3.10.4 Двумерный геометрический объект MbPlaneItem

Класс MbPlaneItem объявлен в файле *plane_item.h*.

Класс MbPlaneItem является наследником классов *MbRefItem*, TapeBase и родительским классом для всех двумерных геометрических объектов.

К двумерным геометрическим объектам ядра C3D относятся: кривые, мультилиния и регион:



Наследниками класса MbPlaneItem являются следующие семейства двумерных геометрических объектов:

MbCurve – двумерная кривая;

MbMultiline – мультилиния;

MbRegion – регион.

Основными методами двумерных геометрических объектов являются:

```
virtual void Move( const MbVector &       to,
                         MbRegTransform * iReg       = nullptr,
                   const MbSurface *      newSurface = nullptr );

virtual void Rotate( const MbCartPoint &     pnt,
                     const MbDirection &     angle,
                           MbRegTransform *  iReg       = nullptr,
                     const MbSurface *       newSurface = nullptr ) = 0;

virtual void Transform( const MbMatrix &       matr,
                              MbRegTransform * iReg       = nullptr,
                        const MbSurface *      newSurface = nullptr );
```

обслуживающие преобразование двумерного геометрического объекта. При преобразовании используется регистратор MbRegTransform для предотвращения многократного преобразования вложенных объектов. Если объект содержит указатели или ссылки на другие объекты, то вложенные объекты так же трансформируются. Регистратор необходимо использовать, если надо последовательно преобразовать несколько взаимосвязанных объектов, связь которых обусловлена наличием в них указателей или ссылок на общие объекты. При преобразовании без регистратора можно получить многократное преобразование общих вложенных объектов.

Кроме того все геометрические объекты имеют методы, обеспечивающие дублирование, проверку на совпадение, проверку на возможность сделать совпадающими и делающие объекты совпадающими:

```
MbPlaneItem & Duplicate( MbRegDuplicate * iReg = nullptr );

bool IsSame( const MbPlaneItem & item,
                   double        accuracy = LENGTH_EPSILON );

bool IsSimilar( const MbPlaneItem & item );

bool SetEqual( const MbPlaneItem & item );
```

При создании копии используется регистратор MbRegDuplicate для предотвращения многократного копирования вложенных объектов. Если объект содержит указатели или ссылки на другие объекты, то вложенные объекты так же копируются. Регистратор необходимо использовать, если надо последовательно копировать несколько взаимосвязанных объектов, связь которых обусловлена наличием в них указателей или ссылок на общие объекты. При копировании без регистратора можно получить набор копий одного и того же вложенного объекта вместо одной его копии.

Для идентификации типа геометрические объекты снабжены методами

```
MbePlaneType IsA();

MbePlaneType Type();

MbePlaneType Family();
```

возвращающими тип из перечисления двумерных геометрических объектов.

Методы

```
MbProperty & CreateProperty( MbePrompt name );

void GetProperties( MbProperties & properties );

void SetProperties( MbProperties & properties );
```

обеспечивают выдачу и редактирование внутренних данных геометрических объектов. Метод GetProperties() добавляет к множеству **properties** данные объекта в виде наследников класса MbProperty.

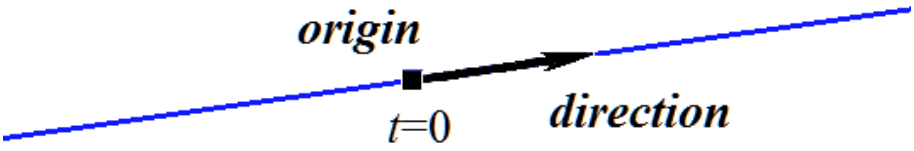
3.11 КРИВЫЕ ДВУМЕРНОГО ПРОСТРАНСТВА

Двумерные кривые используются для описания области определения параметров поверхностей, построения плоских эскизов, построения трёхмерных кривых на поверхностях, кривых пересечения поверхностей, проекций трёхмерных кривых на поверхности и плоскости локальных систем координат. Многие двумерные кривые устроены аналогично трёхмерным кривым с той разницей, что вместо трёхмерных точек и векторов в двумерных кривых используются двумерные точки и векторы. Векторы, радиусы-векторы точек, матрицы в двумерном пространстве обозначаются буквами латинского алфавита, выделенными полужирным наклонным шрифтом.

3.11.1 Двумерная прямая MbLine

Класс MbLine объявлен в файле *cur_line.h*.

Двумерная прямая линия MbLine описывается начальной точкой MbCartPoint **origin** и вектором направления MbDirection **direction**:



В методе PointOn(double & *t*, MbCartPoint & r) радиус-вектор прямой r описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{origin} + t \cdot \mathbf{direction}.$$

Прямая линия ведёт себя как бесконечный объект, хотя в своих данных имеет граничные значения параметра t_{min} и t_{max}. Заметим, что в отличие от других кривых, в методах вычисления радиуса-вектора и его производных прямая не корректирует параметр *t* при его выходе за предельные значения t_{min} и t_{max}.

3.11.2 Двумерный отрезок прямой MbLineSegment

Класс MbLineSegment объявлен в файле *cur_line_segment.h*.

Двумерный отрезок прямой MbLineSegment описывается начальной точкой *MbCartPoint* **point1** и конечной точкой *MbCartPoint* **point2**:



В методе PointOn(double & *t*, *MbCartPoint* & p) радиус-вектор отрезка r описывается векторной функцией

$$\mathbf{r}(t) = (1 - t) \cdot \mathbf{point1} + t \cdot \mathbf{point2}$$

Область определения параметра отрезка располагается в пределах от нуля до единицы. Начальной точке отрезка **point1** соответствует параметр t_{min} = 0, конечной точке отрезка **point2** соответствует параметр t_{max} = 1.

3.11.3 Двумерная кривая-точка MbPointCurve

Класс MbPointCurve объявлен в файле *cur_point_curve.h*.

Двумерная кривая-точка описывается точкой *MbCartPoint* **point**, минимальным параметром кривой **tmin**, максимальным параметром кривой **tmax** и признаком замкнутости кривой **closed**.

В методе

```
PointOn( double & t, MbCartPoint & r)
```

радиус-вектор кривой **r** описывается векторной функцией **r**(*t*) = **point**.

Параметр кривой **t** принимает значения на отрезке: **tmin** ≤ **t** ≤ **tmax**. Кривая может быть периодической. Для параметров должно соблюдаться неравенство: **tmin** < **tmax**.

Двумерная кривая-точка используется в паре с другой двумерной кривой для описания пересечения поверхностей, одна из которых имеет особую точку, например, полюс. Параметры **tmin**, **tmax**, **closed** кривой-точки соответствуют параметрам двумерной кривой, используемой в паре с кривой-точкой.

3.11.4 Двумерная проекционная кривая MbProjCurve

Класс MbProjCurve объявлен в файле *cur_projection_curve.h*.

Двумерная проекционная кривая описывается пространственной кривой *MbCurve3D** **spaceCurve**, поверхностью *MbSurface** **surface** и двумерной кривой MbCurve* **curve**. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Двумерная проекционная кривая представляет собой проекцию пространственной кривой **spaceCurve** на поверхность **surface**, которая приближённо описывается двумерной кривой **curve** в области определения параметров поверхности. Области определения параметров кривых **spaceCurve** и **curve** совпадают. Двумерная кривая **curve**, как правило, является сплайном, контрольные точки которого получены проецированием точек пространственной кривой **spaceCurve** на поверхность **surface**. Параметризация кривой **curve** согласована с параметризацией пространственной кривой в контрольных точках. Двумерная кривая **curve** может располагаться за пределами области определения параметров поверхности.

В методе

```
PointOn( double & t, MbCartPoint & r)
```

радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = [u \quad v],$$

где *u*, *v* — параметры проекции точки **spaceCurve**(*t*) на поверхность **surface**.

Метод PointOn() корректирует параметр *t*, если он выходит за пределы области определения кривой. Для вычисления точки без коррекции параметра используется метод _PointOn().

Начальное приближение параметров *u* и *v* вычисляется методом

```
curve->PointOn(t, point),
```

u = **point**.x, *v* = **point**.y.

Далее параметры *u* и *v* уточняются итерационным методом, использующим уравнения

$$\textbf{deriveU} \cdot (\textbf{spaceCurve}(t) - \textbf{surface}(u, v)) = 0,$$
$$\textbf{deriveV} \cdot (\textbf{spaceCurve}(t) - \textbf{surface}(u, v)) = 0,$$

где **deriveU** и **deriveV** — частные производные радиуса-вектора поверхности, которые вычисляются методами

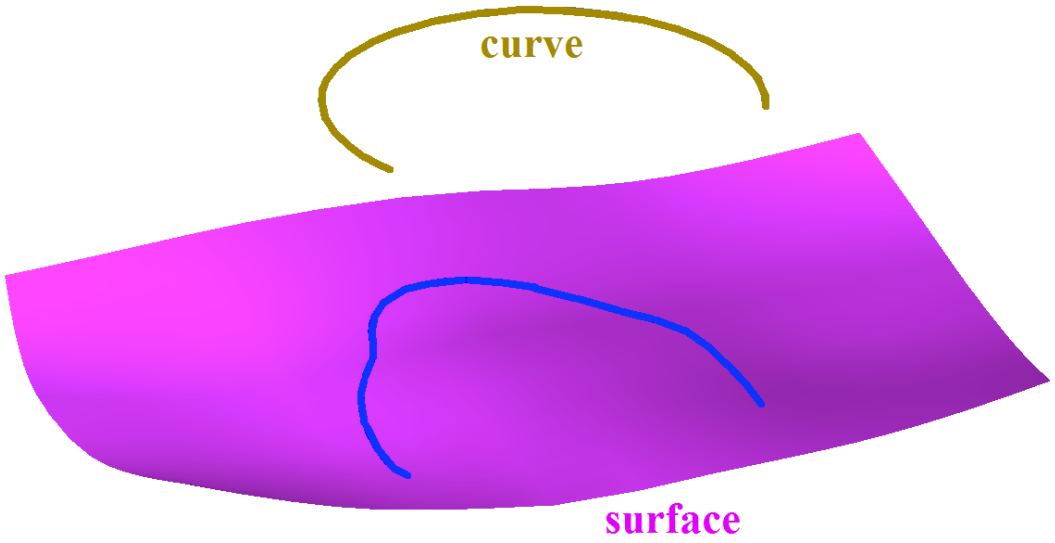
```
surface->_DeriveU(u, v, deriveU)
```

и

```
surface->_DeriveV(u, v, deriveV),
```

соответственно.

Проекционная кривая приведена на рисунке:



Проекционная кривая используется для точного описания проекции пространственной кривой на поверхность.

3.11.5 Двумерный контур MbContour

Класс MbContour объявлен в файле *cur_contour.h*.

Двумерный контур MbContour описывается множеством RPArray<MbCurve> **segments** стыкующихся друг с другом кривых и признаком замкнутости кривой **closed**.

Двумерный контур представляет собой составную кривую. В отличие от других кривых контур может иметь изломы. Кривые, образующие контур, будем называть **сегментами**. Для сегментов контура выполняются следующие условия: начало каждого последующего сегмента совпадает с концом предыдущего сегмента. Для периодического контура начало первого сегмента совпадает с концом последнего сегмента. В случае, если контур содержит только один сегмент одного из следующих типов, он будет считаться непериодическим: MbLine, MbLineSegment, MbCosinusoid.

Для проверки замкнутости и непрерывности точек контура с заданной точностью следует использовать метод IsClosedContinuousC0(). Метод проверяет совпадение первой и последней точек контура, а также совпадение конца каждого сегмента с началом следующего. Проверка не требует гладкого соединения сегментов. По умолчанию используется точность 5.0 * PARAM_NEAR.

Начальное значение параметра контура равно нулю: $t_{min} = 0$. Параметрическая длина контура равна сумме параметрических длин составляющих его сегментов:

$$t_{max} = \sum (w_{imax} - w_{imin}),$$

где $w_{i\,min}$ и $w_{i\,max}$ — минимальное и максимальное значение параметра i-го сегмента. При вычислении радиуса-вектора точки контура сначала по значению параметра определяется рабочий сегмент и значение его локального параметра, далее вычисляется радиус-вектор рабочего сегмента, который служит радиусом-вектором контура.

В методе

```
PointOn( double &      t,
         MbCartPoint & r )
```

радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{segments}[k](w_k),$$

где **segments**[k](w_k) — рабочий сегмент контура с индексом k, w_k — параметр рабочего сегмента, равный

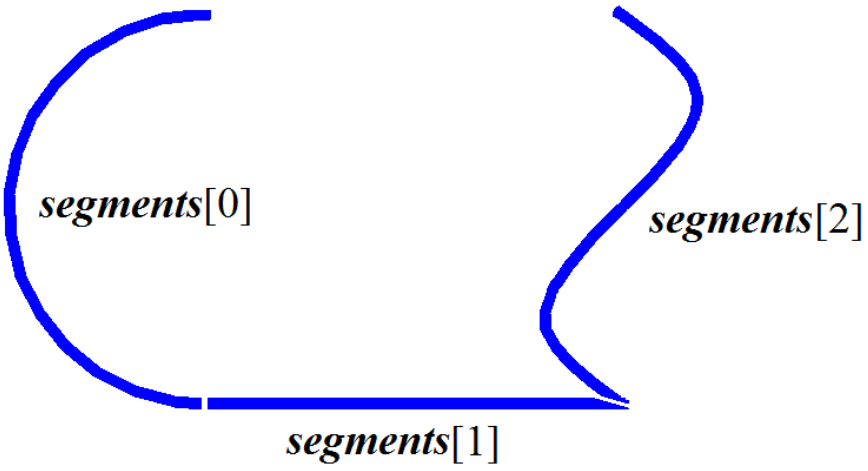
$$w_k = w_{kmin} + t - \sum_{i=0}^{k-1} (w_{imax} - w_{imin}).$$

Сегмент с индексом k определяется по значению параметра контура t из условия

$$\sum_{i=0}^{k-1} (w_{imax} - w_{imin}) \le t < \sum_{i=0}^{k} (w_{imax} - w_{imin}),$$

где $w_{i\,min}$ и $w_{i\,max}$ — минимальное и максимальное значение параметра i-го сегмента.

Контур приведён на рисунке:



В качестве сегментов двумерного контура не должны использоваться другие двумерные контуры. Если контур нужно построить на основе других контуров, то последние должны рассматриваться как совокупность составляющих их кривых, а не как единые кривые.

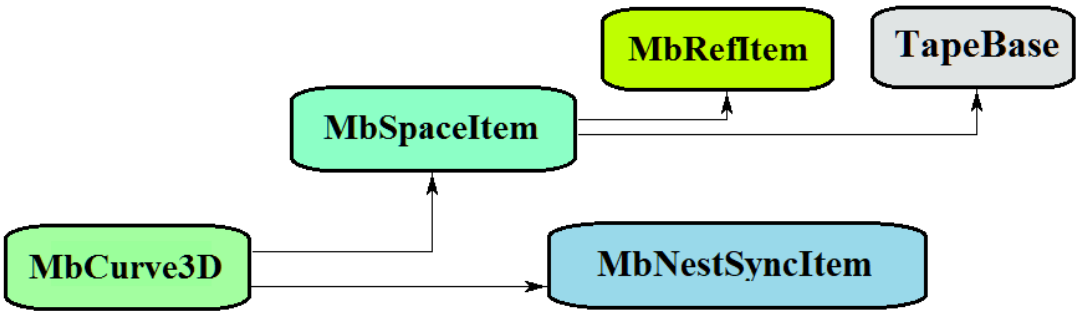
3.12 КРИВЫЕ

Кривые являются представителями семейства трёхмерных геометрических объектов *MbSpaceItem*. Все кривые имеют общий родительский класс *MbCurve3D*. В геометрическом ядре C3D используются кривые, которые построены с помощью аналитических функций, по набору точек, на базе кривых и на базе поверхностей. Кривые используются для построения поверхностей, а также вспомогательных элементов геометрической модели. Векторы, радиусы-векторы точек, матрицы в трёхмерном пространстве обозначаются буквами латинского алфавита, выделенными **полужирным** шрифтом.

3.12.1 Кривая MbCurve3D

Класс MbCurve3D объявлен в файле *curve3d.h*.

Кривая MbCurve3D является наследником классов *MbSpaceItem* и MbNestSyncItem:



Кривая является абстрактным классом. В геометрическом ядре C3D реализованы следующие кривые, которые являются наследниками класса MbCurve3D:

MbLine3D – прямая линия;

MbLineSegment3D – отрезок прямой;

MbArc3D – дуга эллипса;

MbPolyline3D – ломаная линия;

MbNurbs3D – B-кривая (NonUniform Rational B-Spline);

MbBezier3D – составная кривая Безье;

MbHermit3D – кривая Эрмита;

MbCubicSpline3D – кубический сплайн;

MbOffsetCurve3D – эквидистантная кривая;

MbTrimmedCurve3D – усеченная кривая;

MbReparamCurve3D – репараметризованная кривая;

MbConeSpiral – коническая спираль;

MbCurveSpiral – спираль с прямолинейной осью и переменным радиусом;

MbCrookedSpiral – спираль с осью в виде плоской кривой;

MbBridgeCurve3D – сплайн Эрмита, соединяющий две кривые;

MbContour3D – контур (составная кривая);

MbPlaneCurve – плоская кривая в пространстве;

MbSurfaceCurve – кривая на поверхности;

MbSilhouetteCurve – силуэтная кривая поверхности;

MbContourOnSurface – контур на поверхности;

MbContourOnPlane – контур на плоскости;

MbSurfaceIntersectionCurve – кривая пересечения поверхностей.

Кривая MbCurve3D представляет собой векторную функцию

$$\text{curve}(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix}$$

скалярного параметра *t*, принимающего значения на отрезке [t_{min}, t_{max}]. Кривая представляет собой непрерывное отображение некоторого участка числовой оси в трёхмерное пространство. Область изменения параметра кривой есть отрезок [t_{min}, t_{max}] в одномерном пространстве. Координаты $x(t), y(t), z(t)$ точки кривой **curve**(*t*) являются однозначными непрерывными функциями параметра *t*.

Граничные значения t_{min} и t_{max} области определения параметра выдают методы кривой double GetTMin() и double GetTMax(), соответственно.

Кривую будем называть периодической, если существует $p > 0$, такое, что **curve**($t \pm kp$) = **curve**(*t*), где *k* — целое число.

Методы

```
bool IsPeriodic()
```

и

```
bool IsClosed()
```

периодической кривой возвращают *true*.

Метод

```
double GetPeriod()
```

периодической кривой или кривой, которая может быть расширена до периодической, выдает период *p*. Область определения параметра периодической кривой всегда лежит в пределах одного периода.

Кривая считается периодической, если:

- Начальная и конечная точки кривой совпадают. 3D кривые следующих типов не считаются периодическими при совпадении начальной и конечной точки, например в случае вырожденной кривой; для них методы IsClosed(), IsPeriodic() всегда возвращают *false*:
  - *MbLine3D*;
  - *MbLineSegment3D*;

  - MbSpiral и производные классы;
  - *MbBridgeCurve3D*;
  - *MbContour3D*, в случае если он содержит только один сегмент одного из указанных выше типов.
- Производные в начальной и конечной точках совпадают. Если по своей природе кривая допускает изломы (контур, ломаная), то допускается несовпадение производных. У кривой Безье производные должны совпадать по направлению, но могут не совпадать по модулю.

Для проверки совпадения начальной и конечной точек кривой с заданной точностью необходимо использовать метод IsTouch(), который определяет совпадение независимо от гладкости замыкания.

Основным методом кривой является метод

```
void PointOn( double &        t,
              MbCartPoint3D & r );
```

Он выдаёт радиус-вектор **r** точки кривой для заданного параметра *t*.

Методы

```
void FirstDer( double &       t,
               MbVector3D &   rt );

void SecondDer( double &      t,
                MbVector3D &  rtt );

void ThirdDer( double &       t,
               MbVector3D &   rttt );
```

выдают соответственно первую $\mathbf{r}_{t}$, вторую $\mathbf{r}_{tt}$ и третью $\mathbf{r}_{ttt}$ производные радиуса-вектора кривой для заданного параметра *t*. Перечисленные методы корректируют параметр кривой при его выходе за пределы области определения (исключение составляет прямая MbLine3D). При выходе параметра *t* кривой за пределы отрезка $[t_{min}, t_{max}]$ непериодические кривые смещают параметр *t* к ближайшей границе t_{min} или t_{max}, а периодические кривые добавляют или вычитают необходимое количество периодов.

Метод

```
void _PointOn( double          t,
               MbCartPoint3D & r );
```

выдаёт радиус-вектор **r** точки кривой для заданного параметра *t* как в области определения параметра кривой, так и за её пределами. В общем случае непериодическая кривая за пределами области определения параметра продолжается по касательной в крайней точке. Исключение составляют периодические кривые, дуга MbArc3D, спирали MbSpiral, символьная кривая MbCharacterCurve3D и усечённая кривая MbTrimmedCurve3D в пределах базовой кривой. Периодические кривые за пределами области определения параметра продолжаются циклически. Методы

```
void _FirstDer( double          t,
                MbVector3D &    rt );
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
void _SecondDer( double         t,
                 MbVector3D &   rtt );

void _ThirdDer( double          t,
                MbVector3D &    rttt );
```

выдают соответственно первую $\mathbf{r}_{t}$, вторую $\mathbf{r}_{tt}$ и третью $\mathbf{r}_{ttt}$ производные радиуса-вектора кривой для заданного параметра t как в области определения кривой, так и за её пределами.

Кривые перегружают такие методы трёхмерного геометрического объекта, как:

методы, обслуживающие преобразование геометрического объекта,

```
void Move( const MbVector3D &      to,
                 MbRegTransform *  iReg = nullptr );

void Rotate( const MbAxis3D &       axis,
                   double           angle,
                   MbRegTransform * iReg = nullptr );

void Transform( const MbMatrix3D &      matr,
                      MbRegTransform *  iReg = nullptr );
```

методы, обеспечивающие копирование, проверку на совпадение, проверку на возможность сделать совпадающими, делающие объекты совпадающими:

```
MbSpaceItem & Duplicate( MbRegDuplicate * iReg = nullptr );

bool IsSame( const MbSpaceItem & other,
                   double        accuracy = LENGTH_EPSILON );

bool IsSimilar( const MbSpaceItem & item );

bool SetEqual( const MbSpaceItem & item );
```

методы, возвращающие тип из перечисления геометрических объектов:

```
MbeSpaceType IsA();

MbeSpaceType Type();

MbeSpaceType Family();
```

методы, обеспечивающие выдачу и редактирование внутренних данных объекта:

```
MbProperty & CreateProperty( MbePrompt   name );

void GetProperties( MbProperties &      properties );

void SetProperties( MbProperties &      properties );
```

метод, выполняющий построение полигональной копии геометрического объекта:

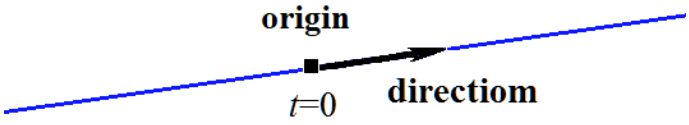
```
void CalculateMesh( const MbStepData & stepData,
                    const MbFormNote & note,
                          MbMesh &     mesh ) const;
```

Все кривые, кроме *MbContour3D*, *MbContourOnSurface*, *MbContourOnPlane*, как правило, не имеют изломов. *MbContour3D*, *MbContourOnSurface*, *MbContourOnPlane* представляют собой составные кривые и могут иметь изломы в точках сочленения составляющих их сегментов.

3.12.2 Прямая линия MbLine3D

Класс MbLine3D объявлен в файле *cur_line_3d.h*.

Прямая линия MbLine3D описывается начальной точкой *MbCartPoint3D* **origin** и вектором направления *MbVector3D* **direction**:



В методе PointOn(double & *t*, MbCartPoint3D & **r**) радиус-вектор прямой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{origin} + t \cdot \mathbf{direction}.$$

Прямая линия ведёт себя как бесконечный объект, хотя в своих данных имеет граничные значения параметра *tmin* и *tmax*. Заметим, что в отличие от других кривых в методах вычисления радиуса-вектора и его производных прямая не корректирует параметр *t* при его выходе за предельные значения *tmin* и *tmax*.

3.12.3 Отрезок прямой MbLineSegment3D

Класс MbLineSegment3D объявлен в файле *cur_line_segment_3d.h*.

Отрезок прямой MbLineSegment3D описывается начальной точкой *MbCartPoint3D* **point1** и конечной точкой *MbCartPoint3D* **point2**:



В методе

```
PointOn( double & t, MbCartPoint3D & r )
```

радиус-вектор отрезка **r** описывается векторной функцией

$$\mathbf{r}(t) = (1 - t) \cdot \mathbf{point1} + t \cdot \mathbf{point2}.$$

Область определения параметра отрезка располагается в пределах от нуля до единицы. Начальной точке отрезка **point1** соответствует параметр t_{min} = 0, конечной точке отрезка **point2** соответствует параметр t_{max} = 1.

3.12.4 Дуга эллипса MbArc3D

Класс MbArc3D объявлен в файле *cur_arc_3d.h*.

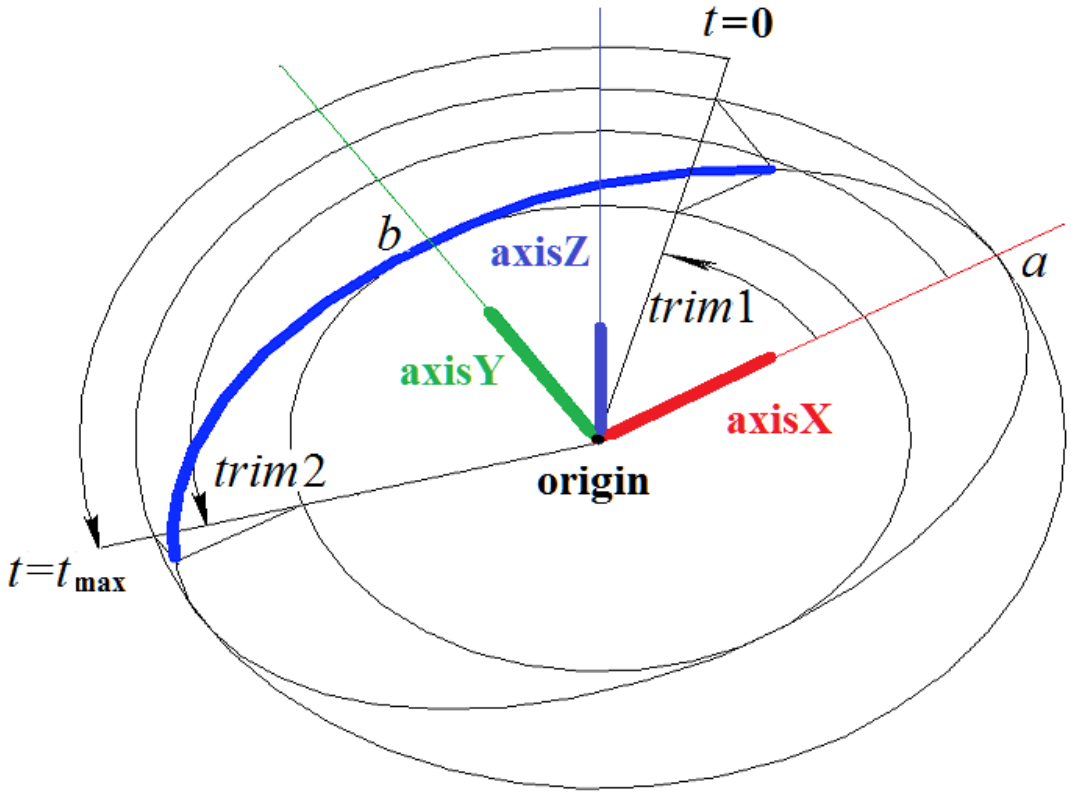
Дуга эллипса является наследником кривой MbCurve3D. Дуга эллипса MbArc3D описывается двумя радиусами *a* и *b* и двумя углами *trim1* и *trim2*, заданными в локальной системе координат *MbPlacement3D* **position**.

Углы *trim1* и *trim2* отсчитываются по дуге в направлении движения от вектора **position.axisX** к вектору **position.axisY**. Углы *trim1* и *trim2* будем называть параметрами усечения. Значения параметров усечения, равные нулю и 2π, соответствуют точке на координатной оси **position.axisX**. Параметр кривой *t* принимает значения на отрезке: 0 ≤ *t* ≤ *trim2* – *trim1*. Кривая может быть периодической. У периодической кривой *trim2* – *trim1* = 2π.

В методе PointOn(double & *t*, MbCartPoint3D & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{position}.\mathbf{origin} + a\cos(trim1 + t)\,\mathbf{position}.\mathbf{axisX} + b\sin(trim1 + t)\,\mathbf{position}.\mathbf{axisY}.$$

Дуга эллипса приведена на рисунке:



Радиусы кривой должны быть больше нуля: *a* > 0, *b* > 0. Для параметров усечения должны соблюдаться неравенства: *trim1* < *trim2*.

Локальная система координат **position** может быть как правой, так и левой.

3.12.5 Ломаная линия MbPolyline3D

Класс MbPolyline3D объявлен в файле *cur_polyline3d.h*.

Ломаная является наследником кривой MbPolyCurve3D. Ломаная MbPolyline3D описывается количеством сегментов *segmentsCount*, множеством контрольных точек SArray<*MbCartPoint3D*> **pointList** и признаком периодичности кривой *closed*.

Кривая проходит через множество точек **pointList**[*i*], *i* = 0,..., *segmentsCount*, при значениях параметра *t* = 0,..., *segmentsCount*. Если *closed* = true, то кривая содержит сегмент, соединяющий последнюю точку множества **pointList**[*segmentsCount* - 1] с начальной точкой **pointList**[0]. Параметр кривой *t* принимает значения на отрезке: 0 ≤ *t* ≤ *segmentsCount*.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

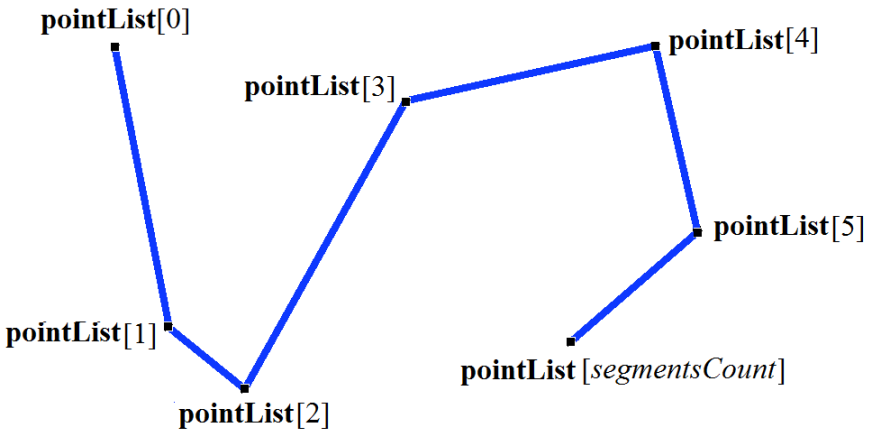
$$\mathbf{r}(t) = \mathbf{pointList}[i] \cdot (1 - w) + \mathbf{pointList}[i + 1] \cdot w,$$

где

$$w = \frac{t - t_i}{t_{i+1} - t_i},$$

а $t_i \le t \le t_{i+1}$. Ломаная является простейшей кривой, построенной по набору точек. Она состоит из отрезков, последовательно соединяющих контрольные точки.

Ломаная приведена на рисунке:



Кривая может быть периодической. Период периодической кривой равен *segmentsCount*. Производные кривой в контрольных точках (при целочисленных значениях параметра) теряют непрерывность по длине и направлению. Производные кривой в контрольных точках имеют особое направление и длину. Ломаная обладает рядом полезных свойств: работа с ней требует минимум вычислений, проекция ломаной линии также будет ломаной линией.

3.12.6 NURBS-кривая MbNurbs3D

Класс MbNurbs3D объявлен в файле *cur_nurbs3d.h*.

NURBS-кривая получила название от первых букв сочетания слов NonUniform Rational B-Spline. Кривая является наследником кривой MbPolyCurve3D. Кривая описывается множеством контрольных точек SArray<MbCartPoint3D> **pointList**, множеством весов контрольных точек *weights*, узловым вектором *knots*, порядком сплайна *degree*, параметром формы кривой *form* и признаком периодичности кривой *closed*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Кривая построена на основе B-сплайнов. Узловой вектор *knots* представляет собой неубывающую последовательность действительных чисел и определяет область определения параметра кривой и форму кривой. Параметр формы кривой *form* в общем случае принимает значение *ncf_Unspecified*, в частных случаях он хранит информацию об исходной кривой, с которой была получена NURBS-копия. Порядок *degree* NURBS-кривой равен порядку разделённых разностей, по которым вычисляются B-сплайны. Пусть узловой вектор содержит *knotsCount* элементов, а множество контрольных точек содержит *pointsCount* элементов. Для непериодической NURBS-кривой количества элементов в множествах связаны равенством *knotsCount* = *pointsCount* + *degree*. Для периодической NURBS-кривой количество элементов в множествах связаны равенством *knotsCount* = *pointsCount* + 2 *degree* – 1.

В методе PointOn(double & *t*, MbCartPoint3D & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$r(t) = \frac{\sum_{j=0}^{pointsCount-1} N_{j^{degree}}(t)\, weight[j]\, \mathbf{pointList}[j]}{\sum_{j=0}^{pointsCount-1} N_{j^{degree}}(t)\, weight[j]},$$

где $N_j^{degree}(t)$ — B-сплайны порядка *degree* для *j*-й контрольной точки **pointList**[*j*].

NURBS-кривая приведена на рис. 1:

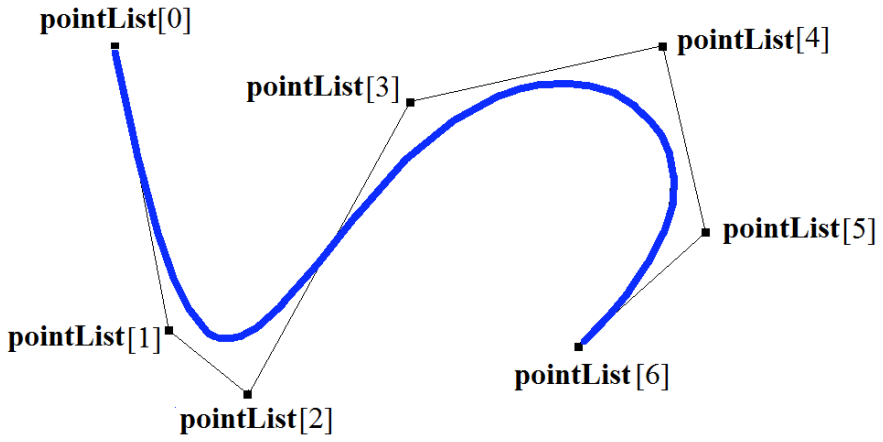


Рис. 1

Параметр кривой *t* принимает значения на отрезке *tmin* ≤ *t* ≤ *tmax*, где *tmin* = *knots* [*degree* – 1], *tmax* = *knots* [*knotsCount* – *degree*]. Кривая может быть периодической. Периодическая NURBS-кривая приведена на рис. 2:

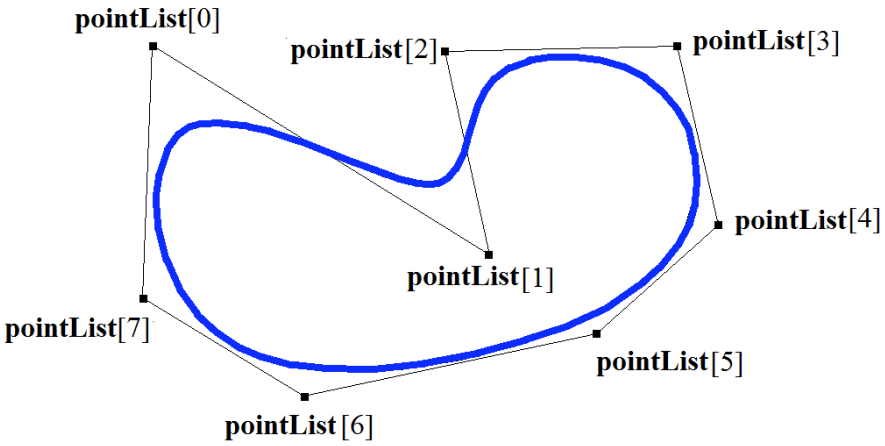


Рис. 2

Форма NURBS-кривой зависит от расположения контрольных точек, от веса контрольных точек и от значений узлового вектора. NURBS-кривая в общем случае не проходит через множество точек **pointList**[i], *i* = 0,..., **pointsCount** – 1. Чтобы незамкнутая NURBS-кривая проходила через крайние контрольные точки, требуется чтобы первые *degree* элементов и последние *degree* элементов узлового вектора *knots* совпадали. Чем больше вес контрольной точки, тем ближе к этой точке проходит кривая при прочих равных условиях.

Каждая кривая может построить свою NURBS-копию виртуальным методом NurbsCurve(const MbNurbsParameters & *tParameters*).

3.12.7 Кривая Эрмита MbHermit3D

Класс MbHermit3D объявлен в файле *cur_hermit3d.h*.

Кривая Эрмита является наследником кривой MbPolyCurve3D. Кривая описывается множеством контрольных точек SArray<MbCartPoint3D> **pointList**, множеством производных кривой в контрольных точках SArray <MbVector3D> **vectorList**, множеством значений параметра кривой в контрольных точках *tList*, количеством *splinesCount* кубических сплайнов Эрмита и признаком периодичности кривой *closed*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Кривая Эрмита при значении параметра *tList[i]*, *i* = 0, 1, ..., *splinesCount*, проходит через контрольную точку **pointList**[*i*] и имеет в ней производную **vectorList**[*i*]. Кривая построена на основе *splinesCount* сплайнов Эрмита третьей степени, которые гладко стыкуются между собой. Каждый кубический сплайн Эрмита описывает участок кривой между двумя соседними контрольными точками. Каждый кубический сплайн Эрмита определяется двумя крайними точками и двумя производными кривой в этих точках.

При вычислении радиуса-вектора точки кривой Эрмита сначала по значению параметра *t* кривой определяется номер *i* рабочего участка (номер кубического сплайна Эрмита) из условия *tList[i]* ≤ *t* ≤ *tList[i+1]*. Радиус-вектор кривой вычисляется как радиус-вектор найденного участка для его локального параметра *w*, который определяется по *tList[i]* и *tList[i+1]*.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией найденного участка для его локального параметра *w*:

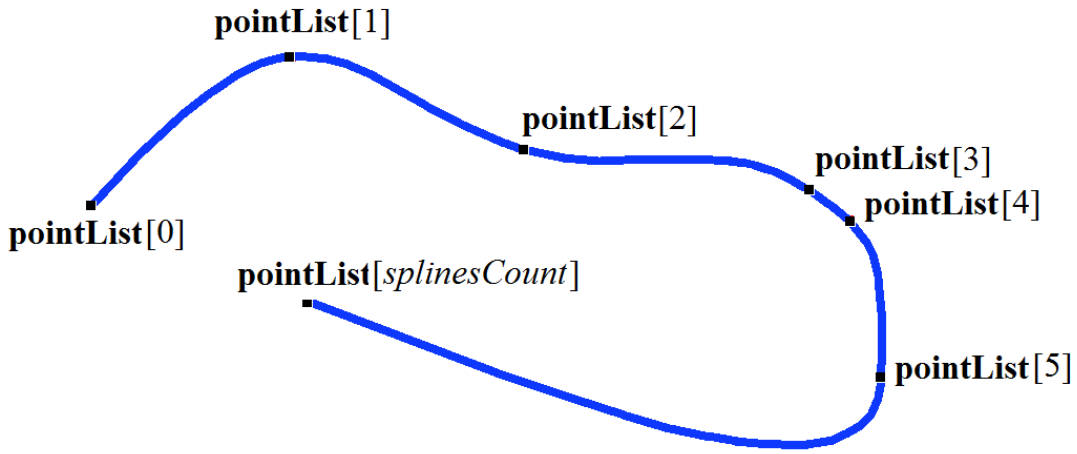
$$r(t) = (1-3w^2+2w^3)\mathbf{pointList}[i]+(3w^2-2w^3)\mathbf{pointList}[i+1]+\big((w-2w^2+w^3)\mathbf{vectorList}[i]+(-w^2+w^3)\mathbf{vectorList}[i \tag{3.5}$$

где

$$w = \frac{t - tList[i]}{tList[i+1] - tList[i]},$$

а *tList[i]* ≤ *t* ≤ *tList[i+1]*.

Кривая Эрмита приведена на рисунке:



Параметр кривой *t* принимает значения на отрезке *tmin* ≤ *t* ≤ *tmax*, где *tmin* = *tList[0]*, *tmax* = *tList[splinesCount]*. Кривая может быть периодической.

Форма кривой зависит от расположения контрольных точек, от производных кривой в контрольных точках и от множества *tList* значений параметра в контрольных точках. При

построении кривой только по контрольным точкам значения параметра кривой в контрольных точках *tList[i]* (*i* = 0, 1, ..., *splinesCount*) изменяются пропорционально расстоянию между точками, а производные **vectorList**[*i*] (*i* = 1, 2, ..., *splinesCount-1*) вычисляются путём построения параболы, проходящей по трём соседним точкам **pointList**[*i-1*], **pointList**[*i*], **pointList**[*i+1*] при соответствующих значениях параметра *tList[i−1]*, *tList[i]*, *tList[i+1]* и вычисления производной параболы в средней точке.

3.12.8 Составная кривая Безье MbBezier3D

Класс MbBezier3D объявлен в файле *cur_bezier3d.h*.

Составная кривая Безье является наследником кривой MbPolyCurve3D. Кривая описывается множеством контрольных точек SArray<MbCartPoint3D> **pointList**, количеством **splinesCount** кривых Безье и признаком периодичности кривой **closed**. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Кривая построена на основе **splinesCount** кривых Безье третьей степени, которые гладко стыкуются между собой. Каждая кривая Безье определяется четырьмя контрольными точками и проходит только через две крайние точки. Составная кривая используется для построения сплайна, проходящего через заданные точки. Заданные точки служат точками сочленения кривых Безье третьей степени. Пара внутренних контрольных точек каждой кривой Безье третьей степени определяется из условия гладкой стыковки кривой с соседними кривыми. Для составной кривой количество контрольных точек равно 3(*splinesCount*+1). У непериодической составной кривой первая **pointList**[0] и последняя контрольные точки не используются.

На каждой кривой Безье третьей степени параметр составной кривой увеличивается на единицу. При вычислении радиуса-вектора точки составной кривой сначала по значению параметра *t* кривой определяется номер рабочего участка (номер кривой Безье третьей степени), который равен максимальному целому числу, не превосходящему *t*. Пусть номер кривой Безье третьей степени равен *n*. Далее определяется дробная часть параметра *w* = *t*–*n*. Радиус-вектор составной кривой вычисляется как радиус-вектор найденного участка для его локального параметра *w*.

В методе PointOn(double & *t*, MbCartPoint3D & **r**) радиус-вектор кривой **r** описывается векторной функцией найденного участка для его локального параметра *w*:

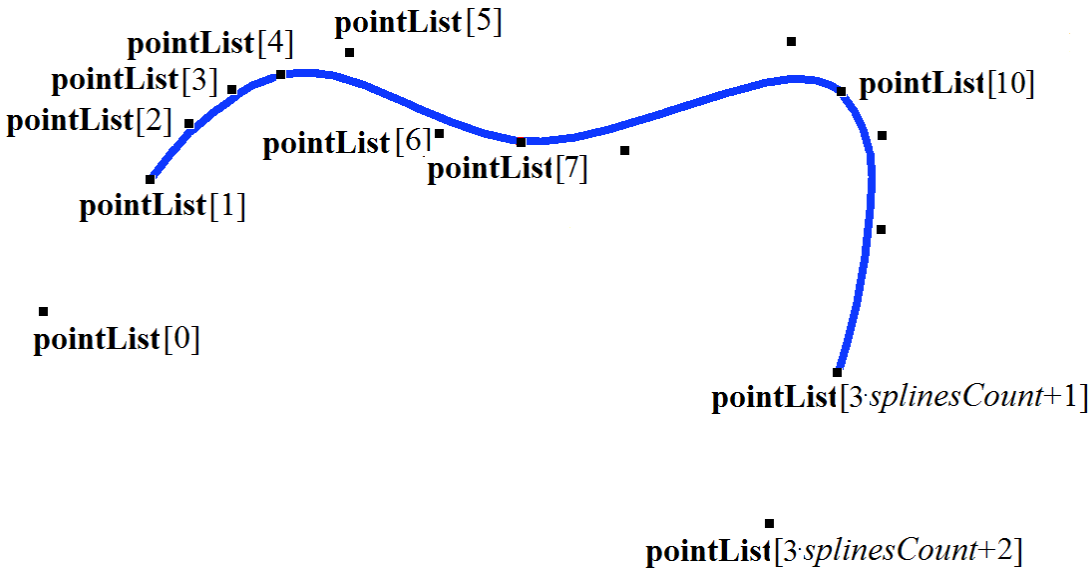
$$r(t) = \sum_{j=0}^{3} \frac{3!}{j!(3-j)!} \cdot w^j (1-w)^{3-j} \cdot \mathbf{pointList}[3n+j],$$

где w = t - n, n ≤ t ≤ n + 1, 0 ≤ w ≤ 1,

$$B_j^3(w) = \frac{3!}{j!(3-j)!} \cdot w^j (1-w)^{3-j}$$

– функции Бернштейна третьей степени для j-ой, j = 0, 1, 2, 3, контрольной точки **pointList**[*3n + j*] найденного участка с номером *n*.

Составная кривая Безье приведена на рисунке:



Параметр кривой *t* принимает значения на отрезке: 0 ≤ *t* ≤ *splinesCount*. Кривая может быть периодической. Период периодической кривой равен **splinesCount**.

При целочисленных значениях параметра кривая проходит через контрольные точки, например, при параметре *t* = *n* кривая пройдёт через контрольную точку **pointList**[3*n*], *n* = 0,1,..., **splinesCount**.Производные кривой в точках сочленения кривых Безье третьей степени (при целочисленных значениях параметра) теряют непрерывность по длине.

3.12.9 Кубический сплайн MbCubicSpline3D

Класс MbCubicSpline3D объявлен в файле **cur_cubic_spline3d.h**.

Кубический сплайн является наследником кривой MbPolyCurve3D. Кривая описывается множеством контрольных точек SArray<*MbCartPoint3D*> **pointList**, множеством вторых производных кривой в контрольных точках SArray<*MbVector3D*> **vectorList**, множеством значений параметра кривой в контрольных точках *tList*, максимальным значением индекса множества параметров *splinesCount* и признаком периодичности кривой *closed*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Кубический сплайн при значении параметра *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, проходит через контрольную точку **pointList**[*i*] и имеет в ней вторую производную **vectorList**[*i*]. Кривая построена так, что при переходе из точки **pointList**[*i*]в точку **pointList**[*i* + 1] вторая производная радиуса вектора кривой изменяется линейно от **vectorList**[*i*] до **vectorList**[*i* + 1].

При вычислении радиуса-вектора точки составной кривой сначала по значению параметра *t* кривой определяется номер *i* рабочего участка, из условия *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1]. Радиус-вектор кривой вычисляется по значениям **pointList**[*i*], **pointList**[*i* + 1], **vectorList**[*i*], **vectorList**[*i* + 1] найденного участка для локального параметра *w*, который определяется по *tList*[*i*] и *tList*[*i* + 1].

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

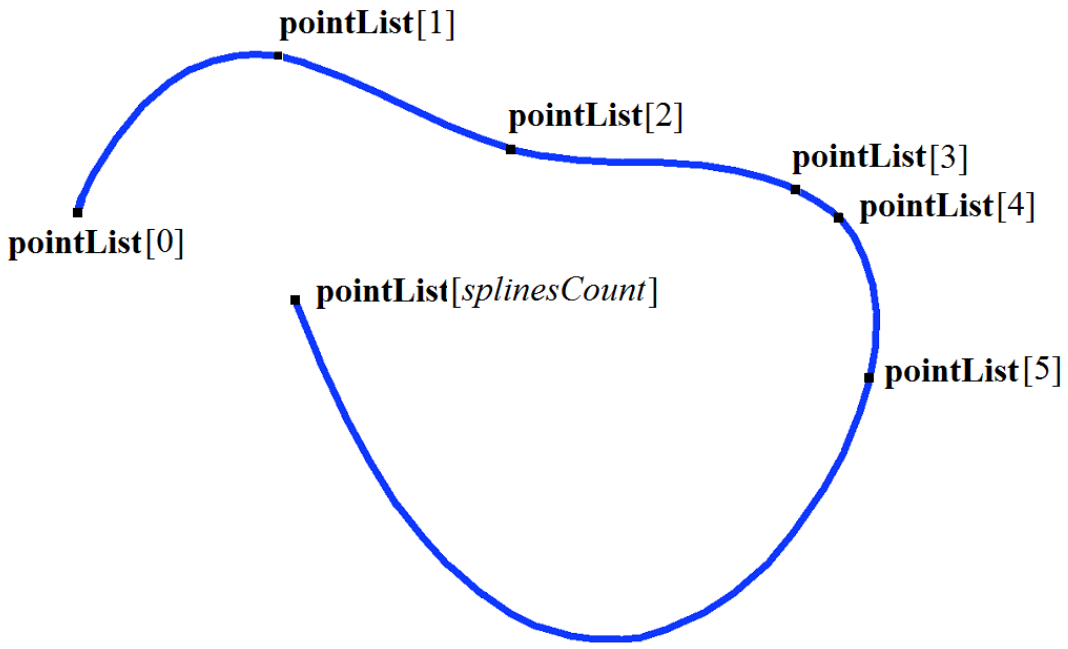
$$r(t) = (1-w)\mathbf{pointList}[i] + w\mathbf{pointList}[i+1] + \left((-2w+3w^2-w^3)\mathbf{vectorList}[i] + (-w+w^3)\mathbf{vectorList}[i+1]\right)\frac{(tList}{}$$

где

$$w = \frac{t - tList[i]}{tList[i+1] - tList[i]},$$

а *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1].

Кубический сплайн, построенный по тем же контрольным точкам, что и составная кривая Эрмита, приведён на рисунке:



Параметр кривой *t* принимает значения на отрезке: *tmin* ≤ *t* ≤ *tmax*, где *tmin* = *tList*[0], *tmax* = *tList*[*splinesCount*]. Кривая может быть периодической.

Форма кривой зависит от расположения контрольных точек и от множества *tList* значений параметра в контрольных точках. При построении кривой только по контрольным точкам значения параметра кривой в контрольных точках *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, изменяются пропорционально расстоянию между точками, а вторые производные **vectorList**[*i*], *i* = 1, 2, ..., *splinesCount* – 1 вычисляются путём решения системы уравнений.

3.12.10 Усечённая кривая MbTrimmedCurve3D

Класс MbTrimmedCurve3D объявлен в файле *cur_trimmed_curve3d.h*.

Усечённая кривая описывается базовой кривой *MbCurve3D** **basisCurve**, начальным параметром усечения базовой кривой **trim1**, конечным параметром усечения базовой кривой **trim2** и признаком совпадения направлений базовой кривой и усечённой кривой **sense**.

Усечённая кривая совпадает с базовой кривой на участке, определённом параметрами **trim1** и **trim2**, но может иметь противоположное с ним направление. Если **sense** = 1, то **trim1** < **trim2** и усечённая кривая совпадает по направлению с базовой кривой. Если **sense** = -1, то **trim2** < **trim1** и усечённая кривая направлена против базовой кривой.

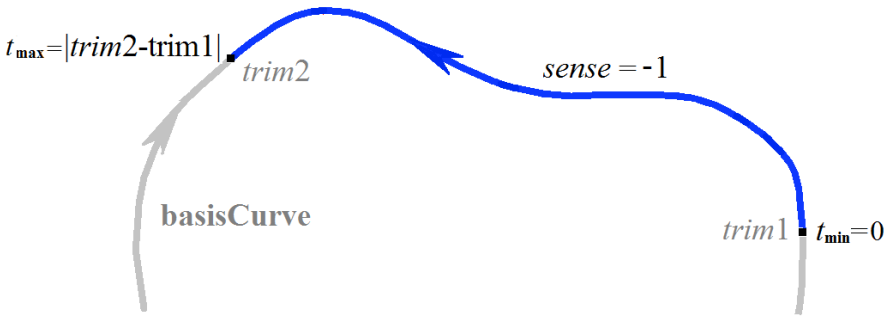
В методе

```
PointOn( double &          t,
         MbCartPoint3D  & r)
```

радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{basisCurve}(trim1 + sense \cdot t).$$

Усеченная кривая приведена на рисунке:



Параметр кривой *t* принимает значения на отрезке:

$$0 \le t \le sense \cdot (trim2 - trim1).$$

Усечённая кривая теоретически может применяться для изменения направления кривой, но лучше пользоваться методом Inverse().

Усечённая кривая может применяться для изменения положения начальной точки периодической кривой. Для этого базовая кривая должна быть периодической и **trim2** = **trim1** + **period**. В этом случае усечённая кривая так же будет периодической.

В качестве базовой кривой не следует использовать другую усечённую кривую. Следует использовать её базовую кривую, предварительно пересчитав параметры усечения.

Каждая кривая может построить свою усеченную копию виртуальным методом

```
Trimmed( double t1, double t2, int sense).
```

3.12.11 Репараметризованная кривая MbReparamCurve3D

Класс MbReparamCurve3D объявлен в файле *cur_reparam_curve3d.h*.

Репараметризованная кривая описывается базовой кривой MbCurve3D **basisCurve**, начальным параметром *tmin*, конечным параметром *tmax* и производной *dt* параметра базовой кривой по параметру репараметризованной кривой.

Репараметризованная кривая полностью совпадает с базовой кривой, но имеет другую область изменения параметра.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{basisCurve}(v(t)),$$

где

$$v(t) = b_{\min}\frac{trim2 - t}{trim2 - trim1} + b_{\max}\frac{t - trim1}{trim2 - trim1},$$

bmin, *bmax* — граничные значения области определения параметра базовой кривой.

Параметр кривой *t* принимает значения на отрезке: *tmin* ≤ *t* ≤ *tmax*.

Репараметризованная кривая совпадает с базовой кривой, но имеет другую область определения параметра. Кривая с изменённой длиной параметра применяется для согласования областей изменения параметра двух кривых. Например, если требуется, чтобы отрезок и дуга имели одинаковые области изменения параметра, то на базе одной из указанных кривых создаётся репараметризованная кривая с областью изменения параметра, взятой у другой кривой.

В качестве базовой кривой для репараметризованной кривой не должна использоваться другая репараметризованная кривая, а должна использоваться базовая кривая последней.

3.12.12 Эквидистантная кривая MbOffsetCurve3D

Класс MbOffsetCurve3D объявлен в файле *cur_offset_curve3d.h*.

Эквидистантная кривая описывается базовой кривой *MbCurve3D* **basisCurve** и вектором смещения *MbVector3D* **offset**. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Базовая кривая **basisCurve** представляет собой объект MbSpine, который для кривой строит перемещающуюся по ней локальную систему координат, первая координатная ось которой направлена по касательной к кривой. Вектор смещения **offset** определяет смещение начальной точки базовой кривой в начальную точку эквидистантной кривой. Вектор смещения **offset** ортогонален касательному вектору базовой кривой в начальной точке. В движущейся локальной системе координат смещение всех точек базовой кривой в соответствующую точку эквидистантной кривой равно вектору смещения и ортогонально касательному вектору базовой кривой в текущей точке.

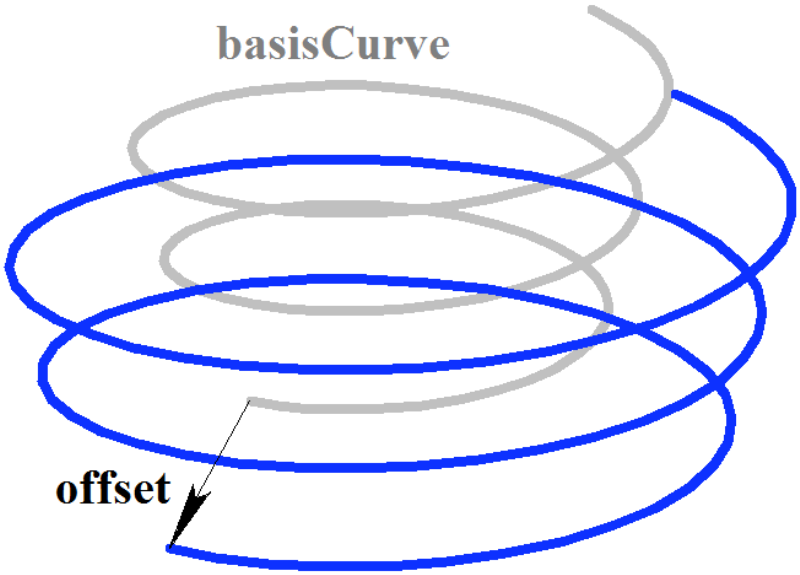
Вычисление радиуса-вектора точки эквидистантной кривой выполняется следующим образом. Для заданного параметра базовой кривой вычисляется точка на направляющей кривой и локальная система координат с началом в этой точке и первой координатной осью, направленной по касательной к кривой в этой точке. Далее вычисляется матрица поворота локальной системы координат при её перемещении из начальной точки базовой кривой в заданную точку. По матрице поворота трансформируется копия вектора смещения и на вычисленный вектор смещается вычисленная точка базовой кривой.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{basisCurve}(t) + \mathbf{offset} \cdot \mathbf{A}(t),$$

где **A**(t) — матрица поворота локальной системы координат при её перемещении из начальной точки базовой кривой в заданную точку.

Эквидистантная кривая к конической спирали приведена на рисунке:



Область изменения параметра эквидистантной кривой совпадает с областью изменения параметра базовой кривой.

В качестве базовой кривой для эквидистантной кривой не должна использоваться другая эквидистантная кривая, а должна использоваться направляющая кривая последней с соответствующим пересчётом вектора смещения.

3.12.13 Символьная кривая MbCharacterCurve3D

Класс MbCharacterCurve3D объявлен в файле *cur_character_curve3d.h*.

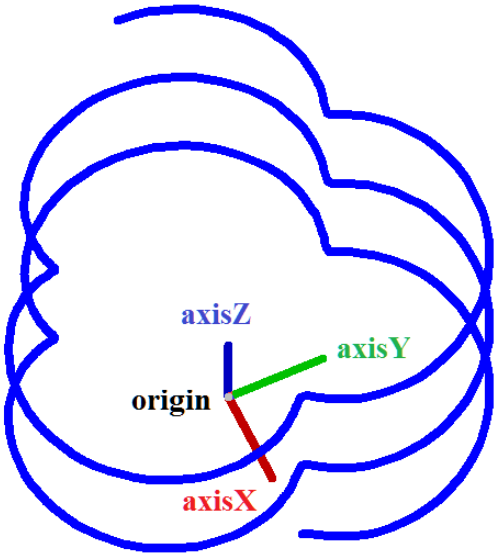
Символьная кривая описывается координатными функциями *xFunction*, *yFunction*, *zFunction*, локальной системой координат *MbPlacement3D* **position** функций, матрицей трансформации **transform**, граничными значениями области определения параметра кривой *tmin* и *tmax*, признаком периодичности кривой *closed* и типом системы координат (декартова, цилиндрическая, сферическая) *coordinateType*, в которой заданы координатные функции. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Координатные функции *xFunction*(t), *yFunction*(t), *zFunction*(t) символьной кривой представляют собой скалярные функции общего параметра *t* и заданы в виде символьных выражений. Для каждого символьного выражения выполнен лексический анализ и построено дерево, которое вычисляет значение символьного выражения для заданного параметра и производные символьного выражения по параметру. Параметр кривой *t* принимает значения на отрезке: *tmin* ≤ *t* ≤ *tmax*.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = [xFunction(t)\ yFunction(t)\ zFunction(t)].$$

Символьная кривая приведена на рис. 1:



$$xFunction(t)=100*\cos(t)-20*\cos(4*t)$$

$$yFunction(t)=100*\sin(t)-20*\sin(4*t)$$

$$zFunction(t)=t*10$$

Рис. 1

Кривая может быть периодической. Символьные выражения на области определения кривой должны описывать непрерывные конечные и однозначные функции.

Синтаксический анализатор входных выражений распознает базовые математические операции с учётом приоритетов и скобок: сложение (+), вычитание (-), умножение (*), деление (/), возведение в степень (^), а также унарный минус (-). Операции осуществляются над переменными, константами и функциями.

Синтаксический анализатор распознаёт имена основных математических функций, которые приведены в таблице 1:

Имя функции	Описание
sin	синус
cos	косинус
tan	тангенс
sind	синус, входной аргумент задаётся в градусах
cosd	косинус, входной аргумент задаётся в градусах
tand	тангенс, входной аргумент задаётся в градусах
asin	арксинус
acos	арккосинус
atan	арктангенс
asind	арксинус, значение выходного параметра в градусах
acosd	арккосинус, значение выходного параметра в градусах
atand	арктангенс, значение выходного параметра в градусах
sqrt	квадратный корень
exp	экспонента
ln	натуральный логарифм
lg	десятичный логарифм
deg	функция конвертации радиан в градусы
rad	функция конвертации градусов в радианы
abs	модуль

Неименованные константы (числа) могут быть заданы как числа с плавающей точкой (строка из цифр, разделённых точкой), либо в формате целого числа.

В символьных выражениях возможно использование именованных констант, приведённых в таблице 2:

Имя константы	Описание
M_PI	π — отношение длины окружности к её диаметру
M_PI_2	$\pi / 2$
M_PI_4	$\pi / 4$
M_SQRT2	$\sqrt{2}$
M_E	e — экспонента
M_FI	Φ — число золотого сечения
M_RADDEG	$180 / \pi$ — коэффициент перевода радиан в градусы
M_DEGRAD	$\pi / 180$ — коэффициент перевода градусов в радианы

Именем переменной аналитического выражения может являться последовательность строковых и числовых литералов (первый символ — литерал строкового типа), которая не совпадает с именами функций и именами констант.

Ограничения при создании символьной кривой:

- максимальная длина переменной — 512 символов;
- максимальная длина выражения — 2048 символов;
- отсутствие операций интегрирования, дифференцирования, вычисления пределов функций;
- отсутствие возможности упрощения математических выражений (например, сокращения дробей).

Ввиду имеющихся ограничений верное вычисление значения аналитического выражения может быть получено для относительно простых математических выражений и функций. Например, значение отношения $sin(x)/x$ в нуле может быть вычислено неверно.

Ниже приведён пример кода, демонстрирующий возможность создания символьной трёхмерной кривой.

```
MbCharacterCurve3D * CreateAnalyticalCurve3D()
{
    MbCharacterCurve3D * curve( nullptr );

    // Строковые представления заданных координатных функций.
    string_t xFunction = _T( "100*cos(t)-20*cos(4*t)" );
    string_t yFunction = _T( "100*sin(t)-20*sin(4*t)" );
    string_t zFunction = _T( "t*10" );
    string_t argument  = _T( "t" );

    double t1 = 0.0, t2 = 5.0 * M_PI;

    // Создание аналитических координатных функций с помощью фабрики функций.
    MbFunctionFactory factory;

    SPtr<MbFunction> x( factory.CreateAnalyticalFunction( xFunction, argument, t1,␣
↪t2, true ) );
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
    SPtr<MbFunction> y( factory.CreateAnalyticalFunction( yFunction, argument, t1, t2, true ) );
    SPtr<MbFunction> z( factory.CreateAnalyticalFunction( zFunction, argument, t1, t2, true ) );

    if ( x != nullptr && y != nullptr && z != nullptr ) {
        MbCartPoint3D point( 0.0, 0.0, 0.0 );           // Начало локальной системы координат.
        MbVector3D    axisX( 1.0, 0.0, 0.0 );
        MbVector3D    axisY( 0.0, 1.0, 0.0 );

        MbPlacement3D place( point, axisX, axisY );     // Локальная система координат.

        MbeLocalSystemType3D cs = ls_CartesianSystem;   // Тип локальной системы координат.

        // Создание символьной трёхмерной кривой.
        curve = new MbCharacterCurve3D( *x, *y, *z, cs, place, t1, t2 );
    }

    return curve;
}
```

3.12.14 Коническая спираль MbConeSpiral

Класс MbConeSpiral объявлен в файле *cur_cone_spiral.h*.

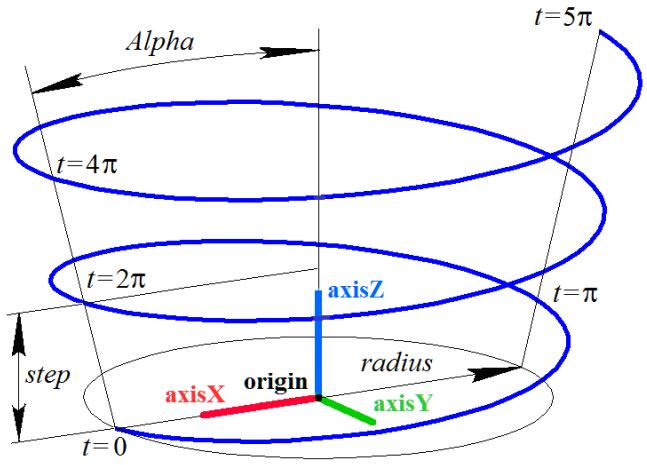
Коническая спираль является наследником кривой MbSpiral. Спираль описывается локальной системой координат *MbPlacement3D* **position**, радиусом *radius*, тангенсом угла конусности *tgAlpha*, шагом спирали, делённым на 2π, *step_2pi* и двумя граничными параметрами спирали *tmin* и *tmax*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Ось спирали совпадает с координатной осью **position.axisZ** локальной системы координат. Параметр *tgAlpha* равен тангенсу угла между осью спирали и образующей конуса спирали. Параметры *tmin* и *tmax* являются углами и отсчитываются от вектора **position.axisX** к вектору **position.axisY**. Значения углов, кратные 2π, соответствуют точке кривой в плоскости XZ локальной системы координат. Параметр кривой *t* принимает значения на отрезке: *tmin* ≤ *t* ≤ *tmax*.

В методе PointOn(double & *t*, MbCartPoint3D & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{position.origin} + (radius + t \cdot tgAlpha \cdot step\_2pi)\,(\cos(t) \cdot \mathbf{position.axisX} + \sin(t) \cdot \mathbf{position.axisY}) + (t \cdot step$$

Коническая спираль приведена на рисунке:



Радиус кривой должен быть больше нуля: *radius* > 0. Для граничных параметров должно выполняться неравенство *tmin* < *tmax*. Кривая не может быть периодической.

Локальная система координат **position** может быть как правой, так и левой. Для цилиндрической спирали *tgAlpha* = 0.

3.12.15 Спираль переменного радиуса MbCurveSpiral

Класс MbCurveSpiral объявлен в файле *cur_curve_spiral.h*.

Спираль переменного радиуса является наследником кривой MbSpiral. Спираль описывается локальной системой координат *MbPlacement3D* **position**, двумерной кривой **curve**, задающей закон изменения радиуса, шагом спирали *step*, двумя граничными параметрами спирали *tmin* и *tmax*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

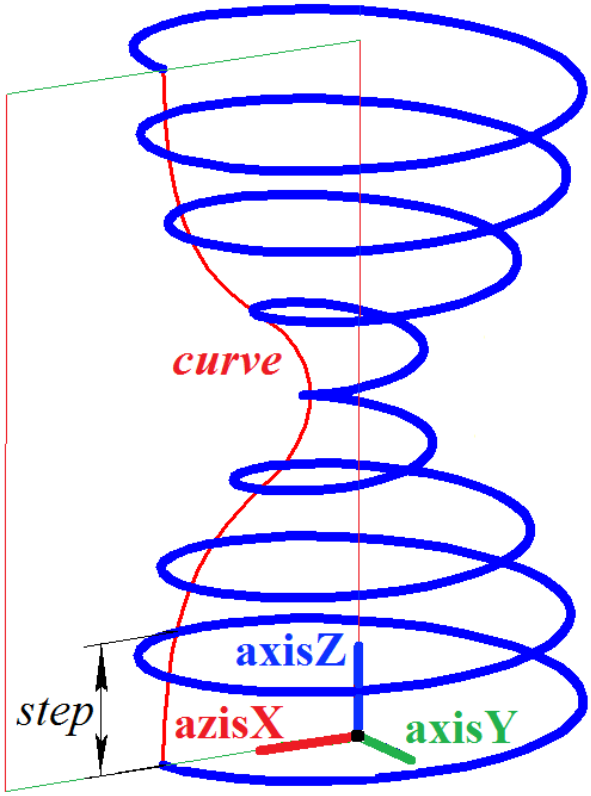
Ось спирали совпадает с координатной осью **position.axisZ** локальной системы координат. Двумерная кривая **curve** располагается в плоскости XZ локальной системы координат и определяет закон изменения радиуса спирали. Ось **position.axisZ** служит осью абсцисс, а ось **position.axisX** служит осью ординат двумерного пространства кривой **curve**. Начало двумерной системы координат кривой **curve** совпадает с началом локальной системы координат **position**. Радиус спирали равен ординате точек двумерной кривой **curve**. Параметры *tmin* и *tmax* являются углами и отсчитываются от вектора **position.axisX** к вектору **position.axisY**. Значения углов, кратные 2π, соответствуют точке кривой в плоскости XZ локальной системы координат.

В методе PointOn(double & *t*, *MbCartPoint* & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{position.origin} + radius(t)\,(\cos(t)\cdot\mathbf{position.axisX} + \sin(t)\cdot\mathbf{position.axisY}) + \left(t\cdot\frac{step}{2\pi}\right)\mathbf{position.axisZ},$$

где *radius*(*t*) — локальный радиус. Локальный радиус *radius*(*t*) вычисляется следующим образом. По заданному параметру *t* определяем абсциссу *t*·*step* / 2π искомой двумерной точки кривой. Далее определяем точку пересечения кривой **curve** и вертикальной прямой, пересекающей ось абсцисс в точке *t*·*step* / 2π. Ордината двумерной точки пересечения **curve** и вертикальной прямой равна искомому радиусу спирали *radius*(*t*). Локальный радиус равен расстоянию от локальной оси абсцисс до точки пересечения вертикальной прямой с кривой **curve** в двумерном пространстве плоскости XZ локальной системы координат **position**.

Спираль переменного радиуса приведена на рисунке:



Кривая **curve** должна располагаться выше оси абсцисс и не должна пересекать ось абсцисс своей двумерной системы координат. Кривая **curve** не должна иметь вертикальных касательных. Для граничных параметров должно выполняться неравенство *tmin* < *tmax*. Кривая не может быть периодической.

Локальная система координат **position** может быть как правой, так и левой.

3.12.16 Спираль с криволинейной плоской осью MbCrookedSpiral

Класс MbCrookedSpiral объявлен в файле *cur_crooked_spiral.h*.

Спираль с криволинейной плоской осью является наследником кривой MbSpiral. Спираль описывается локальной системой координат *MbPlacement3D* **position**, двумерной кривой MbCurve **curve**, определяющей ось спирали, радиусом спирали *radius*, шагом спирали *step,* двумя граничными параметрами спирали *tmin* и *tmax*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

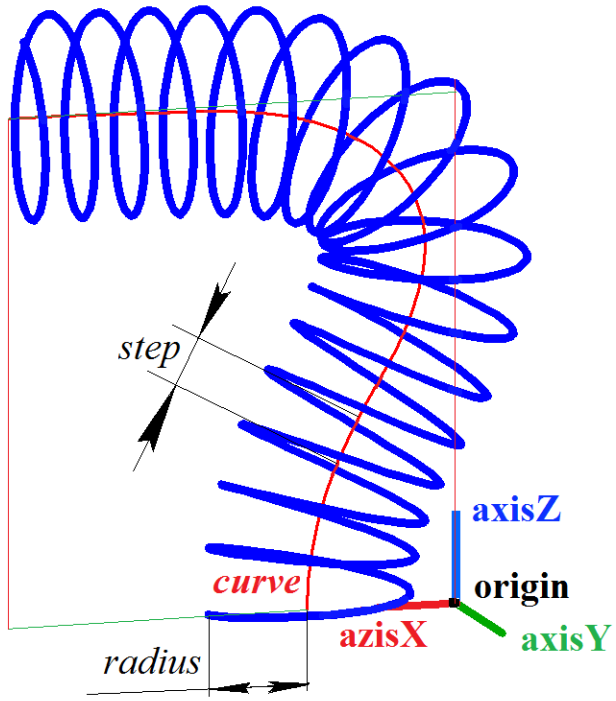
Двумерная кривая **curve** располагается в плоскости XZ локальной системы координат **position** и определяет ось спирали. Ось **position.axisZ** служит осью абсцисс, а ось **position.axisX** служит осью ординат двумерного пространства кривой **curve**. Начало двумерной системы координат кривой **curve** совпадает с началом локальной системы координат **position**. Радиус спирали постоянен. Параметры *tmin* и *tmax* являются углами и отсчитываются от вектора **position.axisX** к вектору **position.axisY**. Значения углов, кратные 2π, соответствуют точке кривой в плоскости XZ локальной системы координат.

В методе PointOn(double & *t*, MbCartPoint3D & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{position}.\mathbf{origin} + \Big((\mathbf{point}.y + \mathbf{radius}\cdot\cos(t)\cdot\mathbf{normal}.y)\cdot\mathbf{position}.\mathbf{axisX}\Big) + \Big(\mathbf{radius}\cdot\sin(t)\cdot\mathbf{position}.\mathbf{ax}$$
$$\Big((\mathbf{point}.x + \mathbf{radius}\cdot\cos(t)\cdot\mathbf{normal}.x)\cdot\mathbf{position}.\mathbf{a}$$

где **point** — точка двумерной кривой, которая вычисляется методом **curve**->PointOn(*t*, **point**), **normal** — нормаль двумерной кривой, которая вычисляется методом **curve**→Normal(*t*, **normal**).

Спираль переменного радиуса приведена на рисунке:



Минимальный радиус кривизны кривой **curve** не должен быть меньше радиуса спирали. Для граничных параметров должны соблюдаться неравенства: *tmin* < *tmax*. Кривая не может быть периодической.

Локальная система координат **position** может быть как правой, так и левой.

3.12.17 Соединительная кривая MbBridgeCurve3D

Класс MbBridgeCurve3D объявлен в файле *cur_bridge3d.h*.

Соединительная кривая является наследником кривой *MbCurve3D* и описывается двумя кривыми MbCurve3D **curve1** и MbCurve3D **curve2**, параметрами точек этих кривых **param1** и **param2**, признаками **sense1** и **sense2** совпадения направлений производных соединительной кривой и соединяемых кривых, двумя граничными параметрами соединительной кривой **tmin** и **tmax**.

Соединительная кривая строится конструктором

```
MbBridgeCurve3D( MbCurve3D & c1, double t1, bool s1,
                 MbCurve3D & c2, double t2, bool s2,
                 double _tmin = 0.0, double _tmax = 1.0 ).
```

Соединительная кривая служит для плавного соединения двух указанных точек кривых **curve1** и **curve2** и представляет собой кубический сплайн Эрмита, построенный по двум крайним точкам и производным кривой в этих точках. Параметр кривой **t** принимает значения на отрезке **tmin** ≤ **t** ≤ **tmax**.

В методе

```
PointOn(double & t, MbCartPoint3D & r)
```

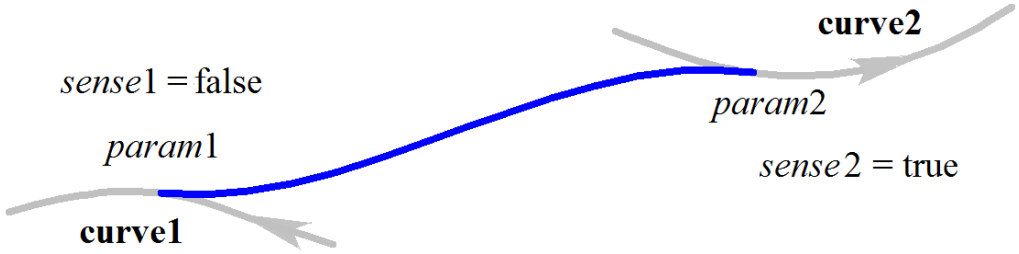
радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = (1 - 3w^2 + 2w^3) \cdot point1 + (3w^2 + 2w^3) \cdot point2 + \Big(\big(w - 2w^2 + w^3\big) \cdot derive1 + \big(-w^2 + w^3\big) \cdot derive2 \Big) \cdot (tmax - tmin$$

где

$$w = \frac{t - tmin}{tmax - tmin}$$

– относительное значение параметра, **point1** – точка кривой **curve1**, которая вычисляется методом PointOn(**param1**, **point1**), **point2** – точка кривой **curve2**, которая вычисляется методом PointOn(**param2**, **point2**), **derive1** и **derive2** – производные соединительной кривой в крайних точках. Векторы **derive1** и **derive2** направлены параллельно производным соединяемых кривых. Длина векторов **derive1** и **derive2** равна расстоянию между крайними точками, делённому на **tmax** – **tmin**. Соединительная кривая приведена на рисунке:



Форма кривой зависит от расположения крайних точек и направлений соединяемых кривых в этих точках. Для граничных параметров должны соблюдаться неравенства: **tmin** < **tmax**. Кривая не может быть периодической.

3.12.18 Контур MbContour3D

Класс MbContour3D объявлен в файле *cur_contour3d.h*.

Контур MbContour3D описывается множеством RPArray<MbCurve3D> **segments** стыкующихся друг за другом кривых и признаком периодичности кривой *closed*.

Контур представляет собой составную кривую. В отличие от других кривых контур может иметь изломы. Кривые, образующие контур, будем называть сегментами. Для сегментов контура выполняются следующие условия: начало каждого последующего сегмента

совпадает с концом предыдущего сегмента. Для периодического контура начало первого сегмента совпадает с концом последнего сегмента. В случае, если контур содержит только один сегмент одного из следующих типов, он будет считаться непериодическим: MbLine3D, MbLineSegment3D, MbSpiral и производные от него классы, MbBridgeCurve3D. Для проверки совпадения начальной и конечной точек контура с заданной точностью необходимо использовать метод IsTouch(), который определяет совпадение независимо от гладкости замыкания. В общем случае в местах стыковки сегментов производные контура терпят разрыв по длине и направлению.

Начальное значение параметра контура равно нулю: $t_{\text{min}} = 0$. Параметрическая длина контура равна сумме параметрических длин составляющих его сегментов:

$$t_{\text{max}} = \sum \left(w_{i\text{max}} - w_{i\text{min}} \right),$$

где w_{imin} и w_{imax} — минимальное и максимальное значение параметра *i*-го сегмента. При вычислении радиуса-вектора точки контура сначала по значению параметра определяется рабочий сегмент и значение его локального параметра, далее вычисляется радиус-вектор рабочего сегмента, который служит радиусом-вектором контура.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{segments}[k]\big(w_k\big),$$

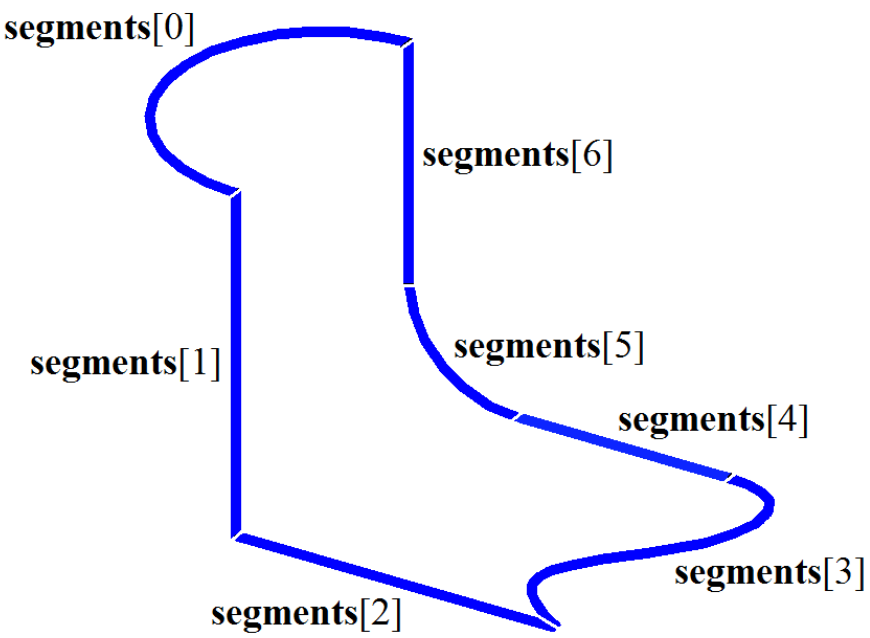
где **segments**[k](w_{k}) — рабочий сегмент контура с индексом *k*, w_{k} — параметр рабочего сегмента, равный

$$w_k = w_{k\text{min}} + t - \sum_{i=0}^{k-1} \big(w_{i\text{max}} - w_{i\text{min}}\big).$$

Сегмент с индексом *k* определяется по значению параметра контура *t* из условия

$$\sum_{i=0}^{k-1} \big(w_{i\text{max}} - w_{i\text{min}}\big) \le t < \sum_{i=0}^{k} \big(w_{i\text{max}} - w_{i\text{min}}\big),$$

где w_{imin} и w_{imax} — минимальное и максимальное значение параметра *i*-го сегмента. Контур приведён на рисунке:



В качестве сегментов контура не должны использоваться другие контуры. Если контур нужно построить на основе других контуров, то последние должны рассматриваться как совокупность составляющих их кривых, а не как единые кривые.

Контур MbContour3D представляет собой наиболее общий вид кривой.

3.12.19 Плоская кривая MbPlaneCurve

Класс MbPlaneCurve объявлен в файле *cur_plane_curve.h*.

Плоская кривая MbPlaneCurve описывается локальной системой координат *MbPlacement3D* **position** и двумерной кривой MbCurve **curve** в плоскости XY локальной системы координат.

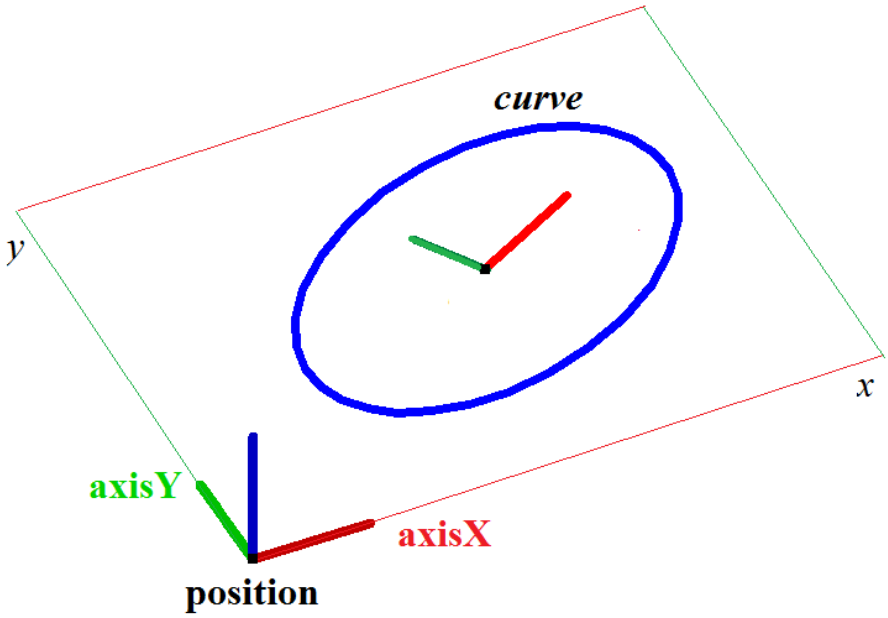
Плоская кривая представляет собой отображение кривой двумерного пространства плоскости XY локальной системы координат в трёхмерное пространство.

В методе PointOn(double & *t*, *MbCartPoint3D* & **r**) радиус-вектор кривой **r** описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{position}.\mathbf{origin} + (\mathbf{point}.x \cdot \mathbf{position}.\mathbf{axisX}) + (\mathbf{point}.y \cdot \mathbf{position}.\mathbf{axisY}),$$

где **point** — точка двумерной кривой, которая вычисляется методом **curve**-> PointOn(*t*, **point**).

Плоская кривая в составе двумерного эллипса и локальной системы координат приведена на рисунке:



Область определения параметров и периодичность плоской кривой совпадают с областью определения параметров и периодичностью двумерной кривой **curve**.

3.12.20 Кривая на поверхности MbSurfaceCurve

Класс MbSurfaceCurve объявлен в файле *cur_surface_curve.h*.

Кривая на поверхности MbSurfaceCurve описывается поверхностью *MbSurface surface*, двумерной кривой MbCurve *curve* в пространстве параметров поверхности и признаком периодичности кривой *closed*. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Кривая на поверхности представляет собой отображение кривой двумерного пространства параметров поверхности в трёхмерное пространство. Двумерная кривая *curve* может располагаться за пределами области определения параметров поверхности. Область определения параметров кривой на поверхности совпадает с областью определения параметров двумерной кривой *curve*.

В методе PointOn(double & *t*, MbCartPoint3D & *r*) радиус-вектор кривой *r* описывается векторной функцией

r(*t*) = *surface*(*point.x*, *point.y*),

где *point* – точка двумерной кривой, которая вычисляется методом *curve* → PointOn(*t*, *point*). Координаты *x* и *y* двумерной точки *point* служат параметрами *u* и *v* поверхности *surface*(*u*, *v*). Кривая на поверхности построена путём введения зависимости параметров поверхности *u* и *v* от некоторого общего для них параметра *t*: *u* = *u*(*t*), *v* = *v*(*t*). Эту зависимость описывает двумерная кривая *curve*.

Кривая на поверхности приведена на рис. 1:

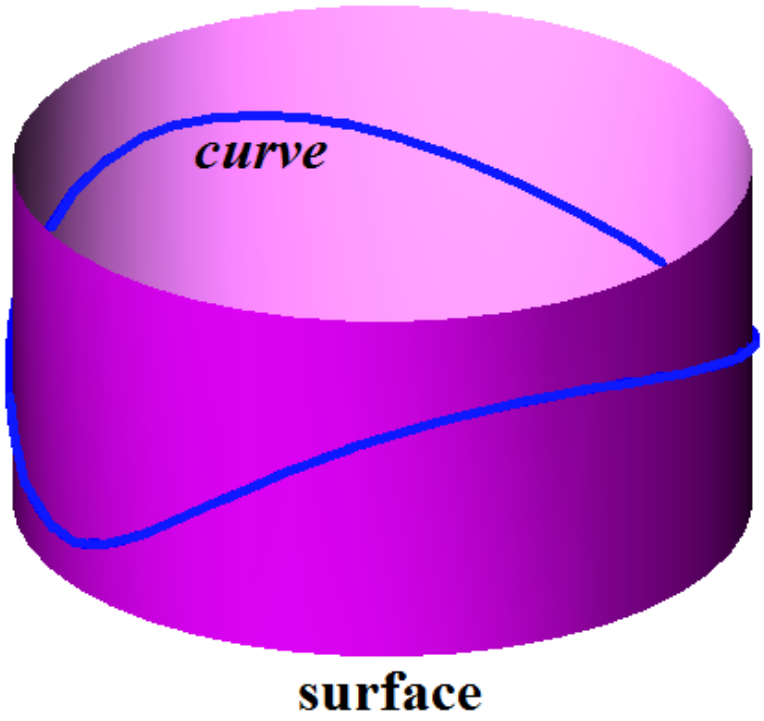


Рис. 1

Двумерная кривая в области определения параметров поверхности приведена на рис. 2:

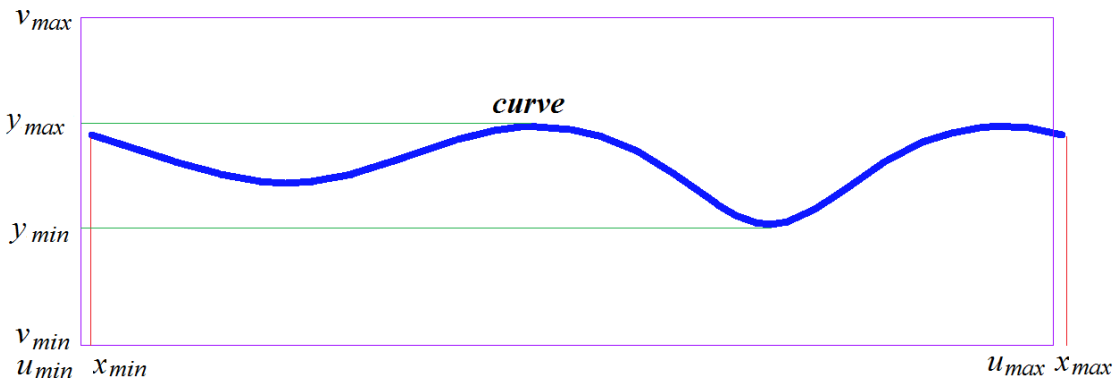


Рис. 2

Производная кривой на поверхности вычисляется как сложная функция

$$\frac{dr(t)}{\text{dt}} = \frac{\partial \text{surface}(u,v)}{\partial u}\text{derive}.x + \frac{\partial \text{surface}(u,v)}{\partial v}\text{derive}.y,$$

где *derive* – производная двумерной кривой, которая вычисляется методом *curve* →

FirstDer(*t*, *derive*). Производная кривой на поверхности лежит в касательной плоскости поверхности, построенной в рассматриваемой точке.

Кривая на поверхности может быть периодической, если периодической является двумерная кривая *curve* или если периодической является поверхность *surface*, а кривая *curve* имеет совпадающие производные на краях и крайние точки кривой смещены на соответствующий период периодической по первому или второму параметру поверхности *surface*.

Поверхностью *surface* кривой на поверхности может служить любая поверхность, кроме поверхности, ограниченной кривыми *MbCurveBoundedSurface*. Если требуется построить кривую на поверхности, ограниченной кривыми, то поверхностью будет служить базовая поверхность *MbCurveBoundedSurface*.

3.12.21 Силуэтная кривая MbSilhouetteCurve

Класс MbSilhouetteCurve объявлен в файле *cur_silhouette_curve.h*.

Силуэтная кривая поверхности MbSilhouetteCurve является наследником кривой на поверхности *MbSurfaceCurve*. Силуэтная кривая описывается поверхностью *MbSurface* **surface**, двумерной кривой MbCurve **curve** в пространстве параметров поверхности, признаком периодичности кривой **closed**, признаком перспективы *perspective*, вектором взгляда **eye** и типом кривой **species**. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Силуэтная кривая представляет собой кривую на поверхности *surface*, разделяющую видимую из точки наблюдения часть этой поверхности и невидимую из точки наблюдения часть этой поверхности. Если *perspective = true*, то точку наблюдения описывает вектор взгляда **eye**. Если *perspective = false*, то точка наблюдения находится на бесконечном расстоянии, а вектор взгляда *eye* описывает направление из точки наблюдения к поверхности. Нормаль поверхности *surface* на силуэтной кривой ортогональна прямой линии, соединяющей эту точку поверхности и точку наблюдения.

В частном случае, когда можно построить точную силуэтную кривую поверхности, тип кривой *species* принимает значение *cbt_Ordinary*. Например, для сферы силуэтной кривой служит окружность. В частном случае строится точная пространственная кривая *exactCurve*, которая точно описывает силуэт поверхности и используется для вычисления радиуса-вектора силуэтной кривой и его производных.

В общем случае тип кривой *species* принимает значение *cbt_Specific*, двумерная кривая *curve* является сплайном и аппроксимирует силуэт поверхности. В общем случае точка силуэтной кривой вычисляется итерационным методом, использующим двумерную кривую *curve* в качестве начального приближения.

Область определения параметров силуэтной кривой поверхности совпадает с областью определения параметров двумерной кривой *curve*.

В методе PointOn(double & *t*, MbCartPoint3D & *r*) радиус-вектор кривой *r* описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{surface}(u, v),$$

где *u*, *v* – координаты двумерной точки, начальное приближение которой вычисляется методом *curve* → PointOn(*t*, *point*), *u = point.x*, *v = point.y*. Далее параметры *u* и *v* уточняются итерационным методом, использующим уравнение

$$\mathbf{vector} \cdot \mathbf{n}(u, v) = 0,$$

где *n(u, v)* – нормаль поверхности, которая вычисляется методом *surface* → Normal(*u*, *v*, *n*), *vector* – вектор взгляда (для бесконечно удалённой точки наблюдения *vector = eye*).

Силуэтная кривая поверхности тора приведена на рис. 1:

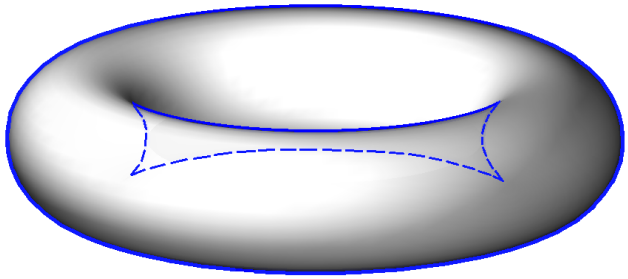


Рис. 1

Силуэтная кривая поверхности тора с другого направления приведена на рис. 2:

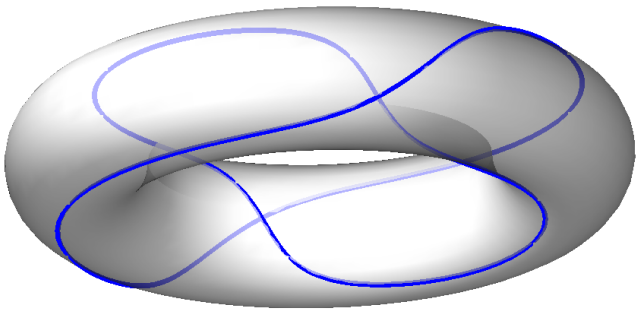


Рис. 2

При переходе через силуэтную кривую скалярное произведение вектора нормали поверхности и вектора взгляда меняет знак. Силуэтная кривая всегда или замкнута, или начинается и оканчивается на краях поверхности. Силуэтная кривая используется для построения проекций силуэта криволинейной поверхности на плоскость.

3.12.22 Контур на поверхности MbContourOnSurface

Класс MbContourOnSurface объявлен в файле *cur_contour_on_surface.h*.

Контур на поверхности MbContourOnSurface описывается поверхностью *MbSurface* **surface** и двумерным контуром *MbContour* **contour** в пространстве параметров поверхности. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Контур на поверхности представляет собой составную кривую, поэтому может иметь изломы в точках стыковки сегментов двумерного контура. Контур на поверхности представляет собой отображение контура двумерного пространства параметров поверхности в трёхмерное пространство. Двумерный контур **contour** может располагаться за пределами области определения параметров поверхности. Область определения параметров

контура на поверхности совпадает с областью определения параметров двумерного контура **contour**.

В методе PointOn(double & *t*, MbCartPoint3D & *r*) радиус-вектор кривой *r* описывается векторной функцией

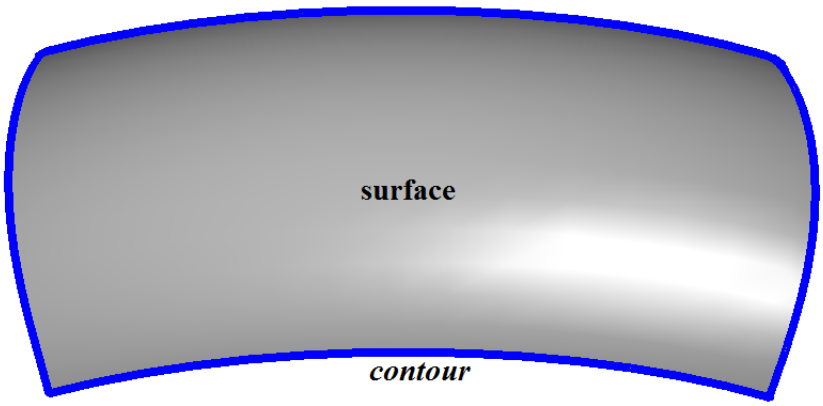
$$\mathbf{r}(t) = \mathbf{surface}(\mathbf{point}.x, \mathbf{point}.y),$$

где *point* – точка двумерного контура, которая вычисляется методом *contour* → PointOn(*t*, *point*). Координаты *x* и *y* двумерной точки *point* служат параметрами *u* и *v* поверхности *surface*(*u*,*v*). Контур на поверхности построен путём введения зависимости параметров поверхности *u* и *v* от некоторого общего для них параметра *t*: *u* = *u*(*t*), *v* = *v*(*t*). Эту зависимость описывает двумерный контур *contour*. Производная контура на поверхности вычисляется как сложная функция

$$\frac{dr(t)}{\mathrm{dt}} = \frac{\partial \mathrm{surface}(u,v)}{\partial u}\mathrm{derive}.x + \frac{\partial \mathrm{surface}(u,v)}{\partial v}\mathrm{derive}.y,$$

где **derive** – производная двумерного контура, которая вычисляется методом *contour* → FirstDer(*t*, *derive*). Производная контура на поверхности лежит в касательной плоскости поверхности, построенной в рассматриваемой точке.

Контур на поверхности приведен на рисунке:



Контур на поверхности может быть периодическим, если периодическим является двумерный контур *contour* или если периодической является поверхность *surface*, а крайние точки контура *contour* смещены на соответствующий период периодической по первому или второму параметру поверхности *surface*.

Периодический контур на поверхности обычно используется для описания границы этой поверхности.

Поверхностью *surface* контура на поверхности может служить любая поверхность, кроме поверхности, ограниченной кривыми *MbCurveBoundedSurface*. Если требуется построить контур на поверхности, ограниченной кривыми, то поверхностью будет служить базовая поверхность *MbCurveBoundedSurface*.

3.12.23 Контур на плоскости MbContourOnPlane

Класс MbContourOnPlane объявлен в файле *cur_contour_on_plane.h*.

Контур на плоскости MbContourOnPlane является наследником класса *MbContourOnSurface*. Контур на плоскости описывается плоскостью *MbSurface surface* и двумерным контуром *MbContour contour* в пространстве параметров плоскости. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

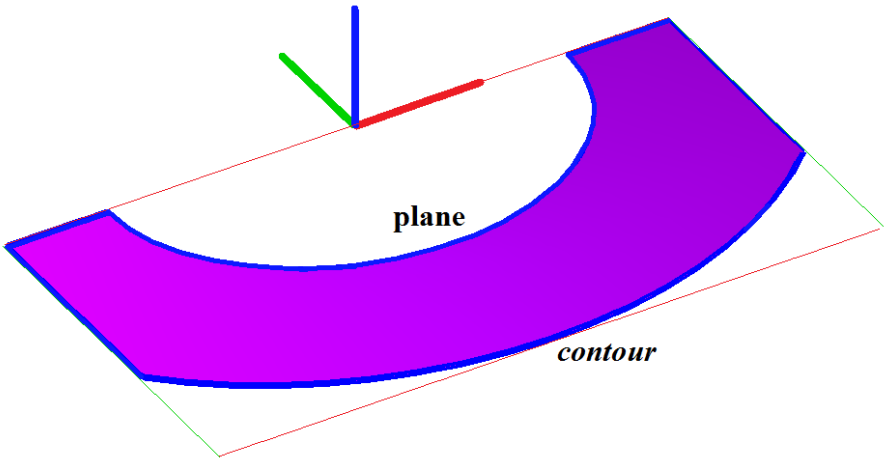
Контур на плоскости представляет собой составную кривую, поэтому может иметь изломы в точках стыковки сегментов двумерного контура. Контур на плоскости представляет собой отображение контура двумерного пространства параметров плоскости в трёхмерное пространство. Двумерный контур *contour* может располагаться за пределами области определения параметров плоскости. Область определения параметров контура на плоскости совпадает с областью определения параметров двумерного контура *contour*.

В методе PointOn(double & *t*, MbCartPoint3D & *r*) радиус-вектор кривой *r* описывается векторной функцией

$$\mathbf{r}(t) = \mathbf{position.origin} + (\mathbf{point}.x\,\mathbf{position.axisX}) + (\mathbf{point}.y\,\mathbf{position.axisY}),$$

где **position** – локальная система координат плоскости **surface**, **point** – точка двумерного контура, которая вычисляется методом *contour* → PointOn(*t*, *point*).

Контур на плоскости приведен на рисунке:



Контур на плоскости может быть периодическим, если периодическим является двумерный контур *contour* или если периодической является плоскость *surface*, а крайние точки контура *contour* смещены на соответствующий период периодической по первому или второму параметру плоскости *surface*.

Периодический контур на плоскости обычно используется для описания границы этой плоскости.

Контур на плоскости аналогичен контуру на поверхности, но обладает более высокой скоростью вычислений.

3.12.24 Кривая пересечения поверхностей MbSurfaceIntersectionCurve

Класс MbSurfaceIntersectionCurve объявлен в файле *cur_surface_intersection.h*.

Кривая пересечения поверхностей MbSurfaceIntersectionCurve описывается двумя кривыми MbSurfaceCurve **curveOne** и *MbSurfaceCurve* **curveTwo** на пересекающихся поверхностях, параметром построения **buildType** и точностью **tolerance**. У кривой есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов кривой.

Кривые **curveOne**(*t*) и **curveTwo**(*t*) имеют одинаковые области определения параметра *t* и совпадают в пространстве с некоторой известной точностью. Параметр **buildType** кривой пересечения описывает тип кривой и несет информацию о способе вычисления радиуса-вектора точки кривой. Параметр **buildType** принимает значения: *cbt_Specific*, *cbt_Ordinary*, *cbt_Boundary*, *cbt_Tolerant*. На рис. 1 приведены две поверхности и кривая их пересечения:

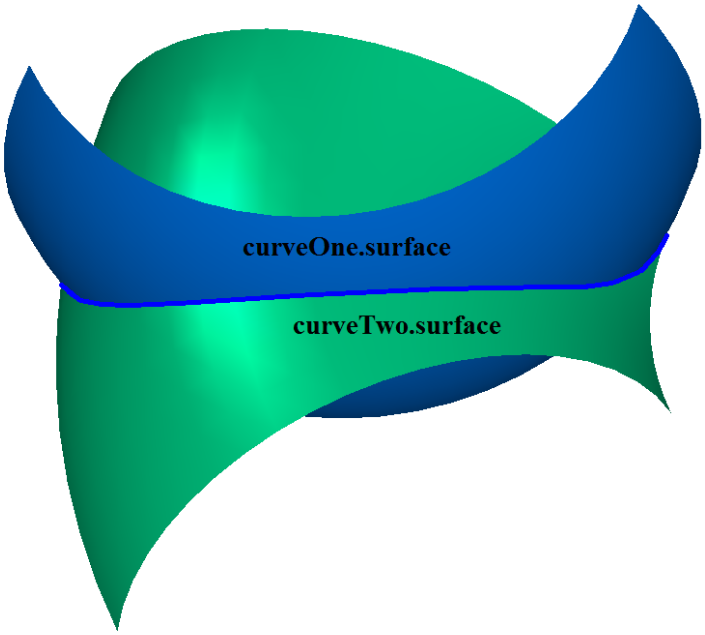


Рис. 1

На рис. 2 и 3 приведены кривые на поверхностях, из которых строится кривая пересечения.

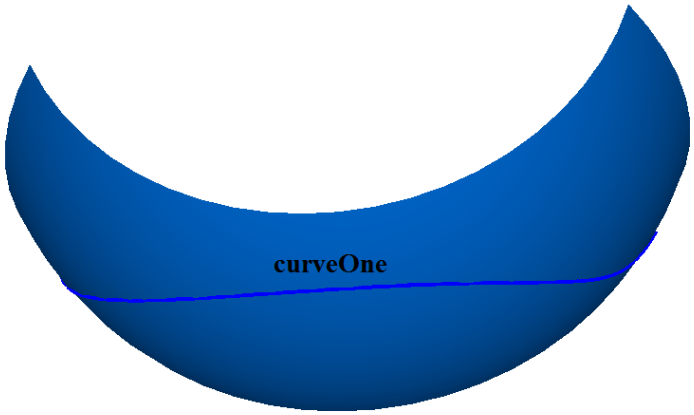


Рис. 2

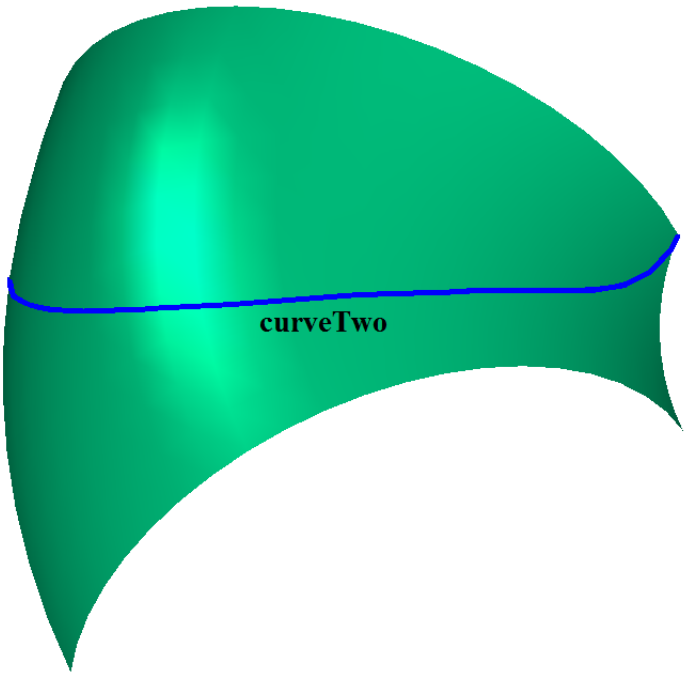


Рис. 3

В общем случае кривая пересечения имеет тип *cbt_Specific*, двумерные кривые **curveOne.curve** и **curveTwo.curve** являются сплайнами и аппроксимируют пересечение поверхностей *curveOne.surface* и *curveTwo.surface*. Сплайны на поверхностях имеют согласованные контрольные точки. В контрольных точках сплайны на поверхностях *curveOne* и *curveTwo* совпадают и имеют одинаковые значения параметров. На участках между контрольными точками сплайнов кривая MbSurfaceIntersectionCurve также выдает точное значение радиуса-вектора точки. В общем случае точка кривой пересечения поверхностей вычисляется итерационным методом, использующим двумерные кривые *curveOne*.*curve* и *curveTwo*.*curve* в качестве начального приближения.

В частных случаях кривая пересечения имеет типы *cbt_Ordinary*, *cbt_Boundary*, *cbt_Tolerant*, а точка кривой пересечения поверхностей вычисляется как среднее арифметическое радиусов-векторов кривых **curveOne**(*t*) и **curveTwo**(*t*).

Если *buildType = cbt_Ordinary*, то кривая MbSurfaceIntersectionCurve точно описывает пересечение поверхностей, а кривые **curveOne**(*t*) и **curveTwo**(*t*) совпадают в пространстве. Примером такой кривой может служить кривая пересечения плоскости и цилиндрической поверхности, ось которой ортогональна плоскости, рис. 4.

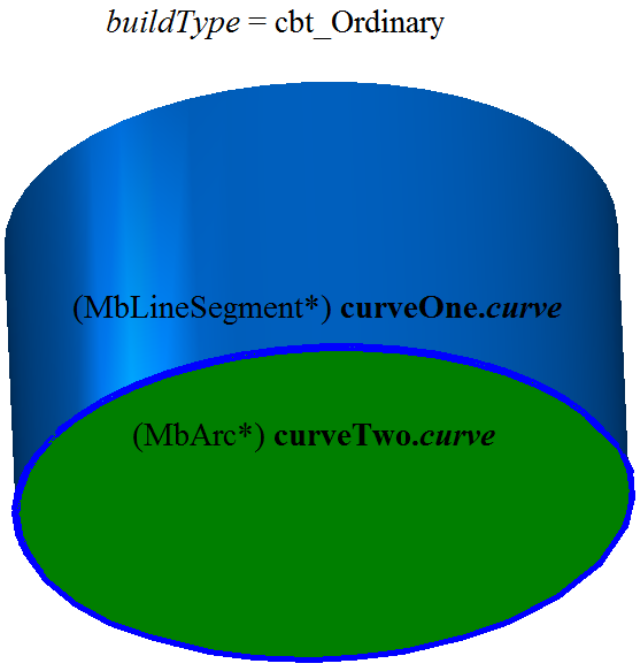


Рис. 4

На плоскости **curveOne**. *curve* является окружностью, а на цилиндрической поверхности **curveTwo**. *curve* является отрезком с параметрической длиной, равной параметрической длине двумерной кривой на плоскости. Равенство параметрических длин достигается построением на базе отрезка репараметризованной кривой MbReparamCurve.

Если *buildType = cbt_Boundary*, то кривая MbSurfaceIntersectionCurve описывает край поверхности, рис. 5. Для такой кривой выполняются равенства *curveOne*.*curve* = *curveTwo*.*curve* и *curveOne.surface* = *curveTwo.surface*.

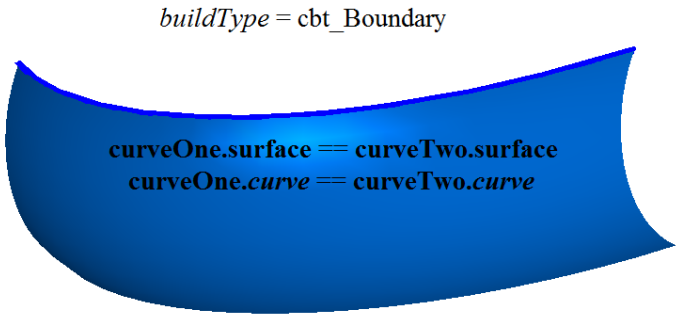


Рис. 5

Если *buildType = cbt_Tolerant*, то кривая MbSurfaceIntersectionCurve описывает пересечение поверхностей приближённо. Кривые *curveOne*(*t*) и **curveTwo**(*t*) совпадают в пространстве с точностью **tolerance**. Такие кривые строятся в тех случаях, когда другое построение невозможно, например, при необходимости пересечь две поверхности, касающиеся друг друга не точно, а с некоторым «шумом».

В методе PointOn(double & *t*, MbMatrix3D & *r*) радиус-вектор кривой *r* описывается векторной функцией

$$\mathbf{r}(t) = 0.5\,(\mathbf{curveOne.surface}(u_1, v_1) + \mathbf{curveTwo.surface}(u_2, v_2))\,,$$

где u_1, v_1 – координаты двумерной точки, начальное приближение которой вычисляется методом *curveOne*.*curve* –> PointOn(*t*, *point1*), u_1 = *point1.x*, v_1 = *point1.y*, u_2, v_2 – координаты двумерной точки, начальное приближение которой вычисляется методом *curveTwo*.*curve* –> PointOn(*t*, *point2*), u_2 = *point2.x*, v_2 = *point2.y*. Далее в общем случае (*buildType = cbt_Specific*) параметры u_1, v_1, u_2, v_2 уточняются итерационным методом, использующим уравнения

$$\mathbf{curveOne.surface}(u_1, v_1) = \mathbf{plane}(x, y),$$

$$\mathbf{curveTwo.surface}(u_2, v_2) = \mathbf{plane}(x, y),$$

где **plane** – плоскость, перпендикулярная отрезку, соединяющему две ближайшие контрольные точки кривой пересечения. Кривая пересечения в общем случае и контрольные точки, по которым построены кривые **curveOne** и **curveTwo**, приведены на рис. 6.

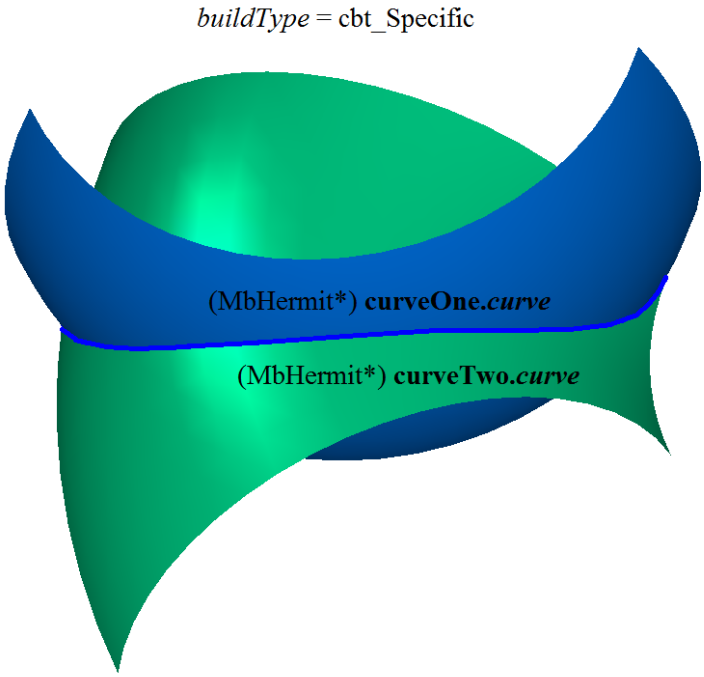


Рис. 6

Область определения параметра кривой пересечения совпадает с областью определения общего параметра кривых **curveOne*** и **curveTwo**. Кривая пересечения поверхностей может быть периодической.

Кривая пересечения поверхностей среди своих данных содержит трёхмерную кривую **spaceCurve**, которая с точностью **tolerance** совпадает с кривой пересечения. Кривая **spaceCurve** используется для построения плоских проекций рёбер. Кривая **spaceCurve** является вспомогательным объектом и рассчитывается только при необходимости.

3.13 ПОВЕРХНОСТИ

Поверхности являются представителями семейства трёхмерных геометрических объектов *MbSpaceItem*. Поверхности играют главную роль в построении геометрической модели. Поверхностями описывают гладкие участки геометрической формы моделируемых объектов. Поверхности строятся с помощью аналитических функций, по набору точек, на базе кривых и на базе поверхностей. Векторы, радиусы-векторы точек, матрицы в трёхмерном пространстве обозначаются буквами латинского алфавита, выделенными **полужирным** шрифтом.

3.13.1 Поверхность MbSurface

Класс MbSurface объявлен в файле *surface.h*.

Поверхность MbSurface является наследником класса *MbSpaceItem*, рис. 1.

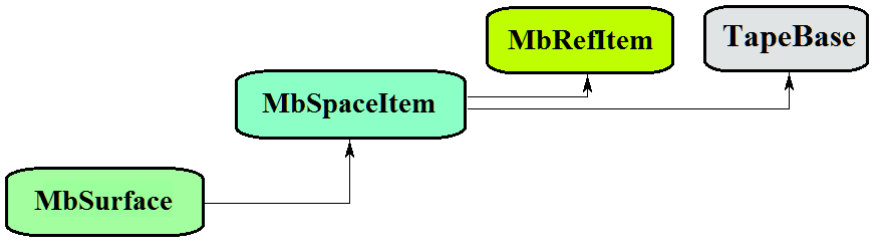


Рис. 1

Поверхность является абстрактным классом. В геометрическом ядре C3D реализованы следующие поверхности, которые являются наследниками класса MbSurface:

- *MbPlane* — плоскость,
- *MbCylinderSurface* — цилиндрическая поверхность,
- *MbConeSurface* — коническая поверхность,
- *MbSphereSurface* — сферическая поверхность,
- *MbTorusSurface* — поверхность тора,
- *MbExtrusionSurface* — поверхность выдавливания,
- *MbRevolutionSurface* — поверхность вращения,
- *MbExpansionSurface* — плоскопараллельная кинематическая поверхность,
- *MbSpiralSurface* — спиральная поверхность,
- *MbEvolutionSurface* — кинематическая поверхность,
- *MbExactionSurface* — кинематическая поверхность с адаптацией,
- *MbSectorSurface* — секториальная поверхность,
- *MbRuledSurface* — линейчатая поверхность,
- *MbLoftedSurface* — поверхность на семействе кривых,
- *MbElevationSurface* — поверхность на семействе кривых и направляющей,
- *MbCornerSurface* — поверхность на трёх кривых,
- *MbCoverSurface* — поверхность на четырёх кривых,
- *MbCoonsPatchSurface* — бикубическая поверхность Кунса,
- *MbMeshSurface* — поверхность на сети кривых,
- *MbJoinSurface* — поверхность соединения,
- *MbSplineSurface* — NURBS-поверхность (NonUniform Rational B-Spline поверхность),
- *MbOffsetSurface* — эквидистантная поверхность,
- *MbChamferSurface* — поверхность фаски,
- *MbFilletSurface* — поверхность скругления,
- *MbChannelSurface* — поверхность скругления с переменным радиусом,
- *MbCurveBoundedSurface* — поверхность с произвольными границами.

Поверхность MbSurface представляет собой векторную функцию

$$\mathbf{surface}(u, v) = \begin{bmatrix} x(u,v) & y(u,v) & z(u,v) \end{bmatrix}$$

двух скалярных параметров *u* и *v*, принимающих значения на двумерной связной области Ω. Поверхность представляет собой непрерывное отображение двумерной связной области Ω в трёхмерное пространство. Область Ω будем описывать в двумерной декартовой системе координат. В частном случае область Ω представляет собой прямоугольник, и параметры поверхности принимают значения в пределах $u_{min} \le u \le u_{max}$, $v_{min} \le v \le v_{max}$. В общем случае область Ω описывается двумерными кривыми. Координаты *x*(*u*, *v*), *y*(*u*, *v*), *z*(*u*, *v*) точки поверхности **surface**(*u*, *v*) являются однозначными непрерывными функциями параметров *u* и *v*.

Граничные значения u_{min}, u_{max}, v_{min}, v_{max} области определения параметров выдают методы поверхности double GetUMin(), double GetUMax(), double GetVMin(), double GetVMax() соответственно.

Поверхность будем называть периодической по первому параметру, если существует $p_u > 0$, такое, что **surface**(*u* ± kp_u, *v*) = **surface**(*u*, *v*), где *k* — целое число. Поверхность будем называть периодической по второму параметру, если существует $p_v > 0$, такое, что **surface**(*u*, *v* ± kp_v) = **surface**(*u*, *v*), где *k* — целое число. Область определения периодического параметра поверхности лежит в пределах одного периода для соответствующего параметра.

Метод bool IsUClosed() периодической по первому параметру поверхности возвращает *true*.

Метод bool IsVClosed() периодической по второму параметру поверхности возвращает *true*.

Метод double GetUPeriod() периодической по первому параметру поверхности или поверхности, которая может быть расширена до периодической, выдаёт период p_u. Метод double GetVPeriod() периодической по второму параметру поверхности или поверхности, которая может быть расширена до периодической, выдаёт период p_v. Область определения параметра периодической поверхности всегда лежит в пределах одного периода.

Введём обозначения

$$s_u = \frac{\partial \mathbf{surface}(u,v)}{\partial u};$$

$$s_v = \frac{\partial \mathbf{surface}(u,v)}{\partial v};$$

$$s_{uu} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial u^2};$$

$$s_{vv} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial v^2};$$

$$s_{uv} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial u \partial v} = \frac{\partial^2 \mathbf{surface}(u,v)}{\partial v \partial u};$$

$$s_{uuu} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial u^3};$$

$$s_{uuv} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial u \partial u \partial v};$$

$$s_{uvv} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial u \partial v \partial v};$$

$$s_{vvv} = \frac{\partial^3 \mathbf{surface}(u,v)}{\partial v^3}$$

для частных производных поверхности по её параметрам.

Основным методом поверхности является метод

void PointOn(double & *u*, double & *v*, *MbCartPoint3D* & **s**).

Он выдаёт радиус-вектор **s**(*u*, *v*) точки поверхности для заданных параметров *u* и *v*.

Методы

void DeriveU(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_u$),

void DeriveV(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_v$),

void DeriveUU(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{uu}$),

void DeriveUV(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{uv}$),

void DeriveVV(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{vv}$),

void DeriveUUU(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{uuu}$),

void DeriveUUV(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{uuv}$),

void DeriveUVV(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{uvv}$),

void DeriveVVV(double & *u*, double & *v*, *MbVector3D* & $\mathbf{s}_{vvv}$)

выдают соответственно производные $\mathbf{s}_u$, $\mathbf{s}_v$, $\mathbf{s}_{uu}$, $\mathbf{s}_{uv}$, $\mathbf{s}_{vv}$, $\mathbf{s}_{uuu}$, $\mathbf{s}_{uuv}$, $\mathbf{s}_{uvv}$, $\mathbf{s}_{vvv}$ радиуса-вектора поверхности для заданных параметров *u* и *v*. Перечисленные методы корректируют параметры поверхности при их выходе за пределы области определения (исключение составляет плоскость *MbPlane*). При выходе параметра *u* за пределы отрезка [u_{min}, u_{max}] непериодические по первому параметру поверхности смещают параметр *u* к ближайшей границе u_{min} или u_{max}, а периодические по первому параметру поверхности добавляют или вычитают необходимое количество периодов. При выходе параметра *v* за пределы отрезка [v_{min}, v_{max}] непериодические по второму параметру поверхности смещают параметр *v* к ближайшей границе v_{min} или v_{max}, а периодические по второму параметру поверхности добавляют или вычитают необходимое количество периодов.

Метод

void _PointOn(double *u*, double *v*, MbCartPoint3D & **s**)

выдаёт радиус-вектор **s**(*u*, *v*) точки поверхности для заданных параметров *u* и *v* как в области определения параметров поверхности, так и за её пределами. Каждая непериодическая поверхность за пределами области определения параметров продолжается по своему закону. При отсутствии такого закона (в общем случае) непериодическая поверхность за пределами области определения параметров продолжается по касательной к соответствующей крайней точке поверхности.

Методы

void _DeriveU(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_u$),

void _DeriveV(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_v$),

void _DeriveUU(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{uu}$),

void _DeriveUV(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{uv}$),

void _DeriveVV(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{vv}$),

void _DeriveUUU(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{uuu}$),

void _DeriveUUV(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{uuv}$),

void _DeriveUVV(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{uvv}$),

void _DeriveVVV(double *u*, double *v*, *MbVector3D* & $\mathbf{s}_{vvv}$)

выдают соответственно производные $\mathbf{s}_u$, $\mathbf{s}_v$, $\mathbf{s}_{uu}$, $\mathbf{s}_{uv}$, $\mathbf{s}_{vv}$, $\mathbf{s}_{uuu}$, $\mathbf{s}_{uuv}$, $\mathbf{s}_{uvv}$, $\mathbf{s}_{vvv}$ радиуса-вектора поверхности для заданных параметров *u* и *v* как в области определения поверхности, так и за её пределами.

Поверхности перегружают такие методы трёхмерного геометрического объекта как:

методы, обслуживающие преобразование геометрического объекта

void Move(const *MbVector3D* & **v**, MbRegTransform * iReg = NULL),

void Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),

void Transform(const *MbMatrix3D* & **m**, MbRegTransform * iReg = NULL),

методы, обеспечивающие копирование, проверку на совпадение, проверку на возможность сделать совпадающими, делающие объекты совпадающими,

MbSpaceItem & Duplicate(MbRegDuplicate * iReg = NULL),

bool IsSame(const *MbSpaceItem* & **item**),

bool IsSimilar(const *MbSpaceItem* & **item**),

bool SetEqual(const *MbSpaceItem* & **item**),

методы, возвращающие тип из перечисления геометрических объектов,

MbeSpaceType IsA(),

MbeSpaceType Type(),

MbeSpaceType Family(),

методы, обеспечивающие выдачу и редактирование внутренних данных объекта,

MbProperty & CreateProperty(MbePrompt name),

void GetProperties(MbProperties & *properties*),

void SetProperties(MbProperties & *properties*),

метод, наполняющий полигональную копию геометрического объекта,

void CalculateWire(double sag, *MbMesh* & **mesh**).

Большинство поверхностей имеют прямоугольную область определения параметров. Среди всех поверхностей выделим *MbCurveBoundedSurface*, которая является универсальной поверхностью. *MbCurveBoundedSurface* имеет криволинейные края и может иметь произвольные вырезы внутри. MbCurveBoundedSurface строится на основе любой поверхности с прямоугольной областью определения параметров.

3.13.2 Плоскость MbPlane

Класс MbPlane объявлен в файле *surf_plane.h*.

Плоскость MbPlane принадлежит к группе элементарных поверхностей MbElementarySurface. Плоскость описывается плоскостью XY локальной системы координат MbPlacement3D **position**. Первый параметр плоскости отсчитывается вдоль вектора **position.axisX**, второй параметр плоскости отсчитывается вдоль вектора **position.axisY**. Область определения плоскости описывают граничные значения параметров *umin*, *umax* и *vmin*, *vmax*, рис. 1:

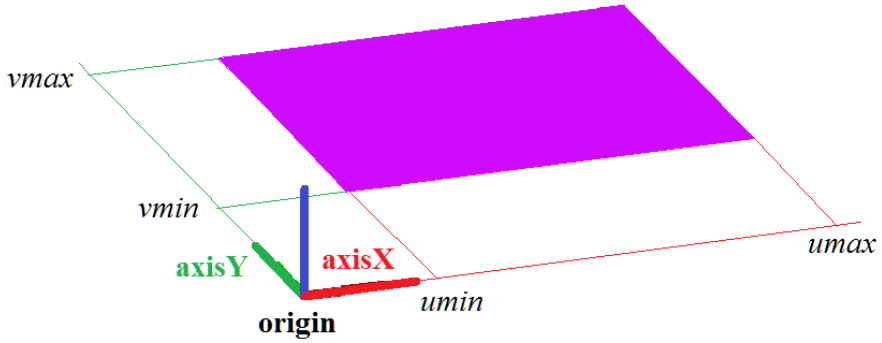


Рис. 1

В методе PointOn(double *u*, double *v*, MbCartPoint3D & **s**) радиус-вектор плоскости **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + u \cdot \mathbf{position.axisX} + v \cdot \mathbf{position.axisY}.$$

Плоскость ведёт себя как бесконечный объект, хотя в своих данных имеет граничные значения параметров *umin*, *umax* и *vmin*, *vmax*. Заметим, что в отличие от других поверхностей в методах вычисления радиуса-вектора и его производных плоскость не корректирует параметры *u* и *v* при их выходе за пределы области определения, заданной значениями *umin*, *umax* и *vmin*, *vmax*.

Локальная система координат **position** может быть как правой, так и левой. Если локальная система координат левая, то нормаль плоскости направлена в сторону, противоположную направлению вектора **position.axisZ**.

3.13.3 Цилиндрическая поверхность MbCylinderSurface

Класс MbCylinderSurface объявлен в файле *surf_cylinder_surface.h*.

Цилиндрическая поверхность MbCylinderSurface принадлежит к группе элементарных поверхностей MbElementarySurface. Цилиндрическая поверхность описывается радиусом *radius* и высотой *height*, заданными в локальной системе координат *MbPlacement3D* **position**.

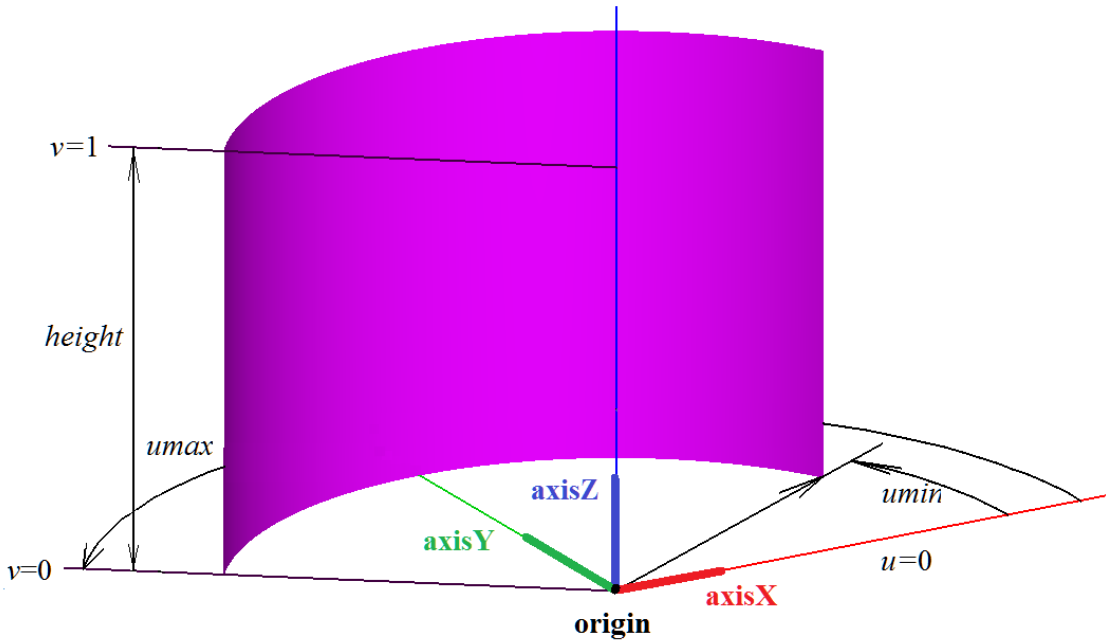
Первый параметр поверхности отсчитывается по дуге от вектора **position.axisX** в направлении вектора **position.axisY**. Первый параметр поверхности *u* принимает значения на отрезке: *umin* ≤ *u* ≤ *umax*. Значения *u* = 0 и *u* = 2π соответствуют точке на плоскости XZ. Поверхность может быть периодической по первому параметру. У периодической поверхности *umax* - *umin* = 2π, у не периодической поверхности *umax* - *umin* < 2π.

Второй параметр поверхности отсчитывается по прямой вдоль вектора **position.axisZ**. Второй параметр поверхности *v* принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Значение *v* = 0 соответствует началу локальной системы координат, а значение *v* = 1 соответствует точке на расстоянии *height* от плоскости XY локальной системы координат поверхности.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + height \cdot v\,\mathbf{position.axisZ} \tag{3.6}$$

Цилиндрическая поверхность приведена на рисунке:



Радиус и высота должны быть больше нуля: *radius* > 0, *height* > 0. Для граничных параметров поверхности должны соблюдаться неравенства: *umin* < *umax*, *vmin* < *vmax*.

Локальная система координат **position** может быть как правой, так и левой. Если локальная система координат правая, то нормаль направлена в сторону выпуклости поверхности (от оси поверхности), если локальная система координат левая, то нормаль направлена в сторону вогнутости поверхности (в сторону оси поверхности).

3.13.4 Коническая поверхность MbConeSurface

Класс MbConeSurface объявлен в файле *surf_cone_surface.h*.

Коническая поверхность MbConeSurface принадлежит к группе элементарных поверхностей MbElementarySurface. Коническая поверхность описывается радиусом *radius*, высотой *height* и углом конусности *angle*, заданными в локальной системе координат *MbPlacement3D* **position**.

Первый параметр поверхности отсчитывается по дуге от вектора **position.axisX** в направлении вектора **position.axisY**. Первый параметр поверхности *u* принимает значения на отрезке: *umin* ≤ *u* ≤ *umax*. Значения *u* = 0 и *u* = 2π соответствуют точке на плоскости XZ. Поверхность может быть периодической по первому параметру. У периодической поверхности *umax* - *umin* = 2π, у непериодической поверхности *umax* - *umin* < 2π.

Второй параметр поверхности отсчитывается по прямой вдоль вектора **position.axisZ**. Второй параметр поверхности *v* принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Значение *v* = 0 соответствует началу локальной системы координат, а значение *v* = 1 соответствует точке на расстоянии *height* от плоскости XY локальной системы координат поверхности.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (radius + height \cdot v \cdot \mathrm{tg}(angle))\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + heig \tag{3.7}$$

Коническая поверхность приведена на рис. 1:

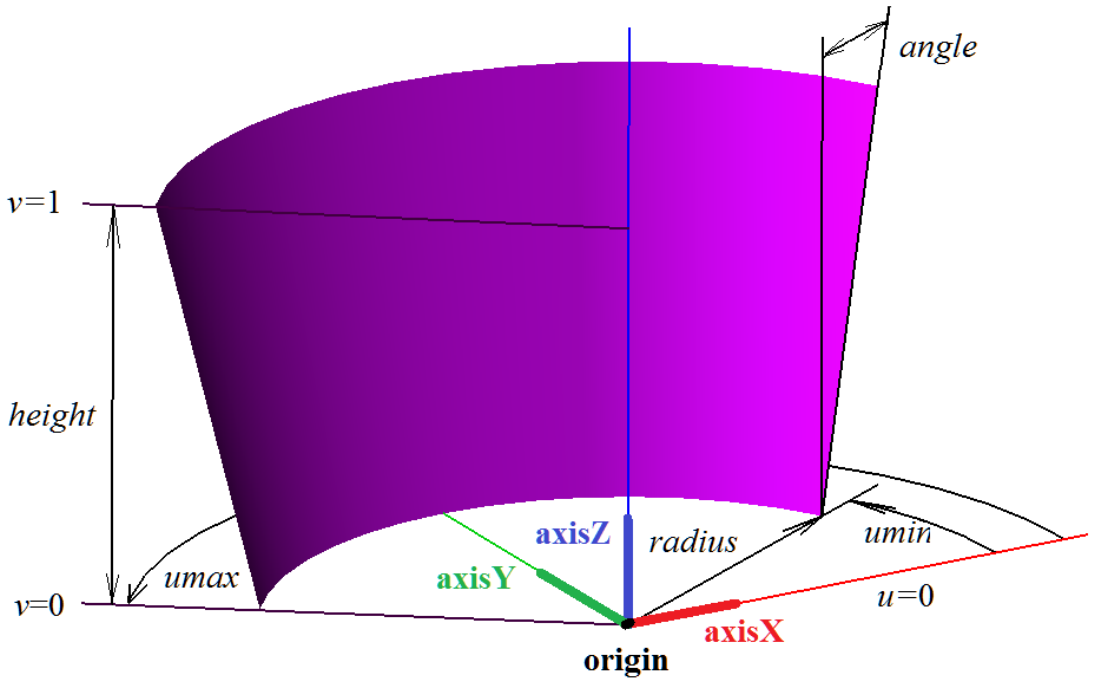


Рис. 1

Радиус и высота должны быть больше нуля, а угол по модулю не должен превышать π/2: *radius* > 0, *height* > 0, -π/2 < *angle* < π/2. При *angle* = 0 коническая поверхность эквивалентна цилиндрической поверхности. Для граничных параметров поверхности должны соблюдаться неравенства: *umin* < *umax*, *vmin* < *vmax*. Полюсу поверхности соответствует значение второго параметра *v* = -*radius* / (*height* · tg(*angle*)). Граничные параметры *vmax* и *vmin* принимают такие значения, при которых поверхность располагается с одной стороны от полюса.

Локальная система координат **position** может быть как правой, так и левой. Если локальная система координат правая, то нормаль направлена в сторону выпуклости поверхности (от оси поверхности), если локальная система координат левая, то нормаль направлена в сторону вогнутости поверхности (в сторону оси поверхности).

3.13.5 Сферическая поверхность MbSphereSurface

Класс MbSphereSurface объявлен в файле *surf_sphere_surface.h*.

Сфера MbSphereSurface принадлежит к группе элементарных поверхностей MbElementarySurface. Сфера описывается радиусом *radius*, заданным в локальной системе координат *MbPlacement3D* **position**.

Первый параметр поверхности отсчитывается по дуге от вектора **position.axisX** в направлении вектора **position.axisY**. Первый параметр поверхности *u* принимает значения на отрезке: *umin* ≤ *u* ≤ *umax*. Значения *u* = 0 и *u* = 2π соответствуют точке на плоскости XZ. Поверхность может быть периодической по первому параметру. У периодической поверхности *umax* - *umin* = 2π, у не периодической поверхности *umax* - *umin* < 2π.

Второй параметр поверхности отсчитывается по дуге от плоскости XY локальной системы координат поверхности в направлении вектора **position.axisZ**. Второй параметр поверх-

ности *v* принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Значение *v* = 0 соответствует точке на плоскости XY локальной системы координат поверхности. Поверхность непериодическая по второму параметру.

В методе PointOn(double *u*, double *v*, MbCartPoint3D & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY}) + radius \cdot \sin(v)\,\mathbf{position.axisZ} \tag{3.8}$$

Сфера приведена на рис. 1:

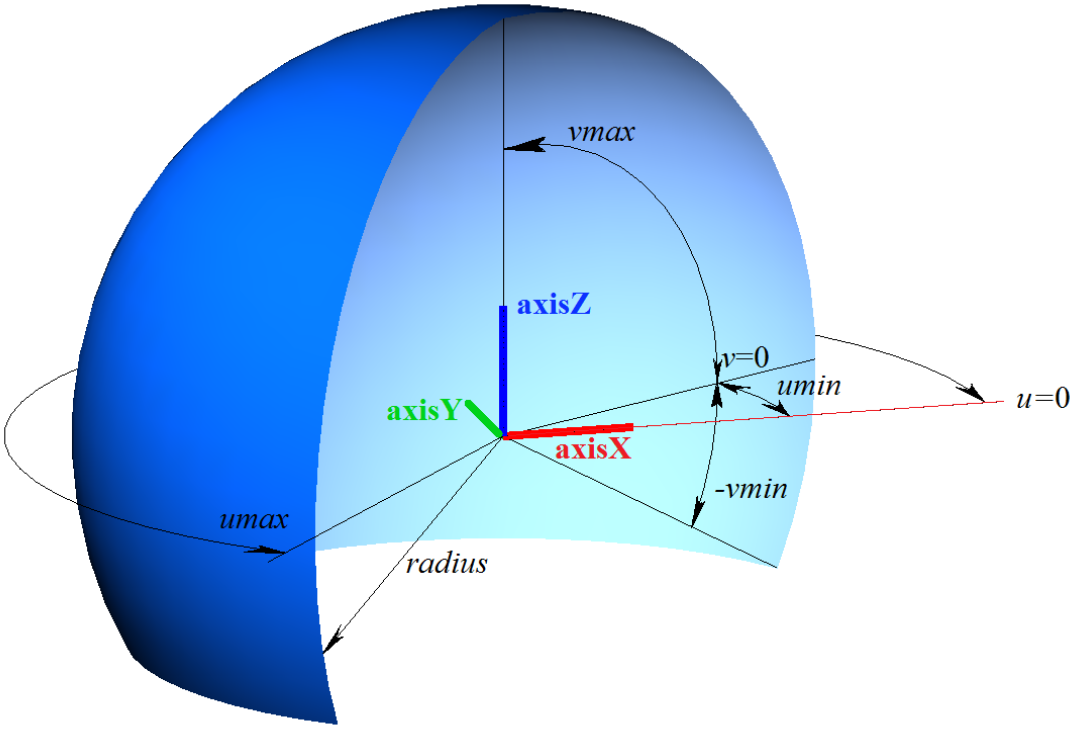


Рис. 1

Радиус сферы должен быть больше нуля: *radius* > 0. Сфера имеет полюсы для параметра *v* = π/2 и *v* = -π/2. Для граничных параметров поверхности должны соблюдаться неравенства: *umin* < *umax*, *vmin* < *vmax*, *vmax* ≤ π/2, *vmin* ≥ -π/2.

Локальная система координат **position** может быть как правой, так и левой. Если локальная система координат правая, то нормаль направлена наружу сферы, если локальная система координат левая, то нормаль направлена внутрь сферы.

3.13.6 Поверхность тора MbTorusSurface

Класс MbTorusSurface объявлен в файле *surf_torus_surface.h*.

Поверхность тора MbTorusSurface принадлежит к группе элементарных поверхностей MbElementarySurface. Поверхность тора описывается радиусом центров *majorRadius* и радиусом трубки *minorRadius*, заданными в локальной системе координат *MbPlacement3D* **position**.

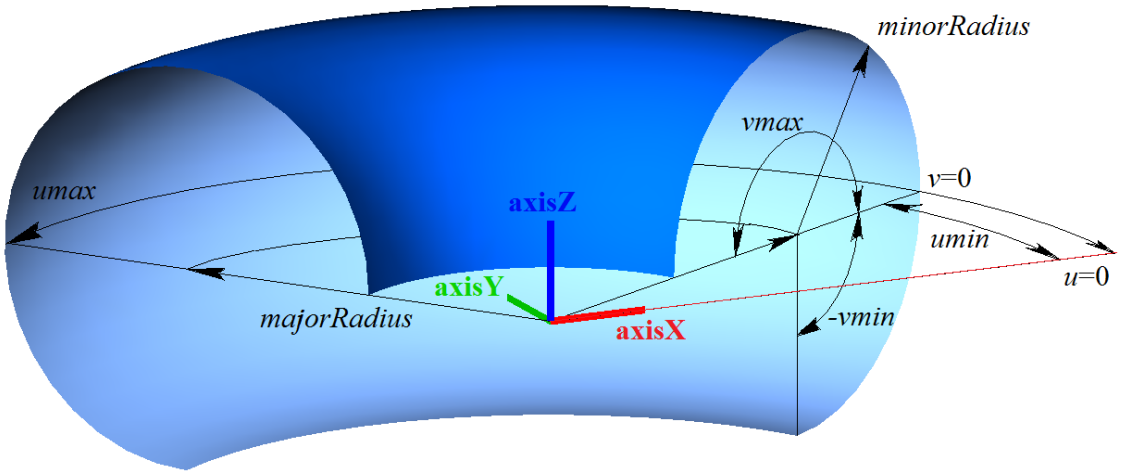
Первый параметр поверхности отсчитывается по дуге от вектора **position.axisX** в направлении вектора **position.axisY**. Первый параметр поверхности *u* принимает значения на отрезке: *umin* ≤ *u* ≤ *umax*. Значения *u* = 0 и *u* = 2π соответствуют точке на плоскости XZ. Поверхность может быть периодической по первому параметру. У периодической поверхности *umax* - *umin* = 2π, у не периодической поверхности *umax* - *umin* < 2π.

Второй параметр поверхности отсчитывается по дуге от плоскости XY локальной системы координат поверхности в направлении вектора **position.axisZ**. Второй параметр поверхности *v* принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Значения *v* = 0 и *v* = 2π соответствуют точке на плоскости XY локальной системы координат поверхности. Поверхность может быть периодической по второму параметру при *majorRadius* > *minorRadius*. У периодической поверхности *vmax* - *vmin* = 2π, у непериодической поверхности *vmax* - *vmin* < 2π.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (majorRadius + minorRadius \cdot \cos(v))\,(\cos(u)\,\mathbf{position.axisX} + \sin(u)\,\mathbf{position.axisY} \tag{3.9}$$

Поверхность тора приведена на рисунке:



Радиус трубки должен быть больше нуля: *minorRadius* > 0. Радиус центров должен быть не меньше радиуса трубки, взятого с обратным знаком: *majorRadius* > -*minorRadius*. Если *majorRadius* < *minorRadius*, то поверхность имеет полюс для параметра *v* = arccos(*majorRadius* / *minorRadius*) и *v* = 2π - arccos(*majorRadius* / *minorRadius*). Для граничных параметров поверхности должны соблюдаться неравенства: *umin* < *umax*, *vmin* < *vmax*.

Локальная система координат **position** может быть как правой, так и левой. Если локальная система координат правая, то нормаль направлена от трубки поверхности, если локальная система координат левая, то нормаль направлена внутрь трубки поверхности.

3.13.7 Поверхность выдавливания MbExtrusionSurface

Класс MbExtrusionSurface объявлен в файле *surf_extrusion_surface.h*.

Поверхность выдавливания MbExtrusionSurface принадлежит к группе поверхностей движения MbSweptSurface. Поверхность выдавливания является частным случаем поверхности движения с прямолинейной направляющей кривой. Поверхность выдавливания опи-

сывается образующей кривой *MbCurve3D* **curve**, вектором направления выдавливания MbVector3D **direction** и длиной выдавливания *distance*.

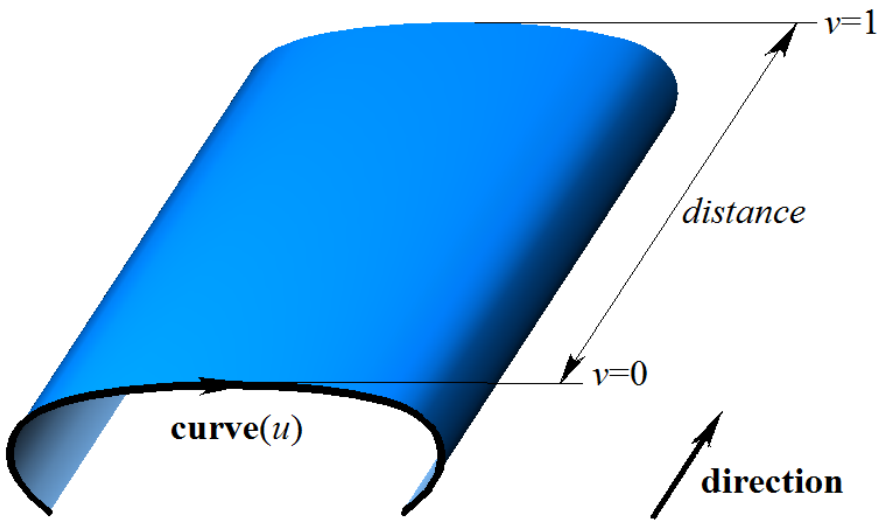
Первый параметр поверхности *u* совпадает с параметром образующей кривой. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения образующей кривой. Поверхность может быть периодической по первому параметру, если периодической является образующая кривая.

Второй параметр поверхности *v* принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Значение *v* = 0 соответствует точке на образующей кривой, значение *v* = 1 соответствует точке образующей кривой, смещённой на вектор **direction** * *distance*. Поверхность не может быть периодической по второму параметру.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{curve}(u) + \mathbf{direction} \cdot distance \cdot v \tag{3.10}$$

Поверхность выдавливания приведена на рисунке:



Для граничных значений второго параметра поверхности должно соблюдаться неравенство: *vmin* < *vmax*.

3.13.8 Поверхность вращения MbRevolutionSurface

Класс MbRevolutionSurface объявлен в файле *surf_revolution_surface.h*.

Поверхность вращения MbRevolutionSurface принадлежит к группе поверхностей движения MbSweptSurface. Поверхность вращения является частным случаем поверхности движения с направляющей кривой в форме окружности или её дуги. Поверхность вращения описывается образующей кривой *MbCurve3D* * **curve**, локальной системой координат *MbPlacement3D* **position**, вектор **position.axisZ** которой является осью вращения, признаком расположения кривой и оси вращения в одной плоскости *planeData*, признаком наличия полюса поверхности при начальном значении первого параметра *poleMin*, признаком наличия полюса поверхности при конечном значении первого параметра *poleMax*, значениями первого параметра поверхности в полюсах поверхности *uPoleMin*, *uPoleMax*,

если соответствующий полюс присутствует. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* совпадает с параметром образующей кривой. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения образующей кривой. Поверхность может быть периодической по первому параметру, если периодической является образующая кривая.

Второй параметр поверхности *v* принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Значения *v* = 0 и *v* = 2π соответствуют точке на образующей кривой. Поверхность может быть периодической по второму параметру. У периодической поверхности *vmax* - *vmin* = 2π, у не периодической поверхности *vmax* - *vmin* < 2π.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + (\mathbf{curve}(u) - \mathbf{position.origin})\,\mathbf{M}(v), \tag{3.11}$$

где **M**(*v*) — матрица вращения. Заметим, что умножение вектора (**curve**(*u*) - **position.origin**) на матрицу **M**(*v*) выполняется справа. Матрица вращения имеет вид

$$M(v) = A^{-1}\cdot\begin{bmatrix}\cos v & -\sin v & 0\\ \sin v & \cos v & 0\\ 0 & 0 & 1\end{bmatrix}\cdot A = \begin{bmatrix}\mathbf{position.axisX}\\ \mathbf{position.axisY}\\ \mathbf{position.axisZ}\end{bmatrix}^{-1}\cdot\begin{bmatrix}\cos v & -\sin v & 0\\ \sin v & \cos v & 0\\ 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}\mathbf{position.axisX}\\ \mathbf{position.axisY}\\ \mathbf{position.axisZ}\end{bmatrix}. \tag{3.12}$$

Матрица **A** является матрицей преобразования координат радиуса-вектора точки из локальной системы координат **position** в глобальную систему координат. Строки матрицы **A** составлены из компонент базисных векторов локальной системы координат. Матрица **M**(*v*) переводит вектор **curve**(*u*) - **position.origin** в локальную систему координат, поворачивает его в ней на угол *v* вокруг оси вращения и возвращает повёрнутый вектор обратно в глобальную систему координат. Поверхность вращения приведена на рис. 1:

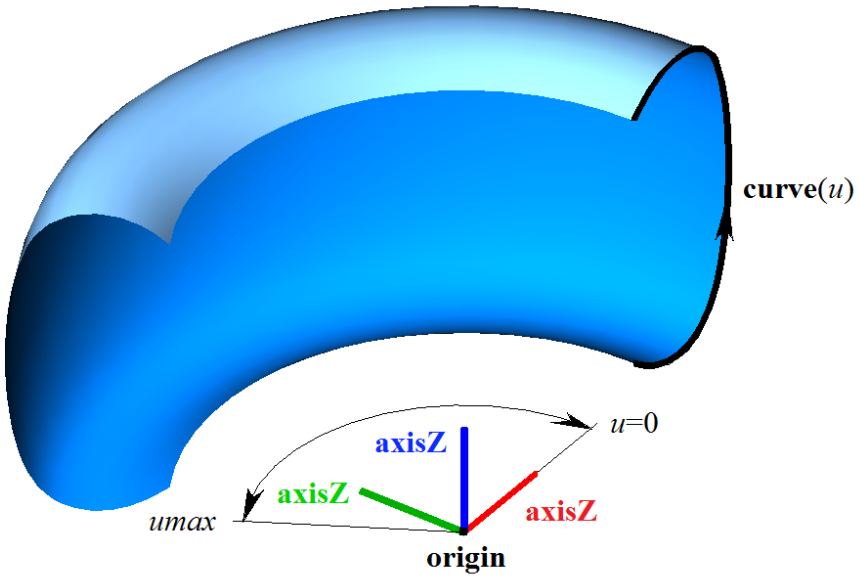


Рис. 1

Если начальный или конечный край образующей кривой проходит через ось вращения, то поверхность имеет полюс для параметра *umin* или *umax* соответственно. Для гранич-

ных значений второго параметра поверхности должно соблюдаться неравенство: *vmin* < *vmax*.

Вращение точек образующей кривой выполняется по дуге вокруг вектора **position.axisZ** от вектора **position.axisX** в направлении вектора **position.axisY**. Локальная система координат **position** может быть как правой, так и левой.

3.13.9 Поверхность перемещения MbExpansionSurface

Класс MbExpansionSurface объявлен в файле *surf_expansion_surface.h*.

Поверхность перемещения MbExpansionSurface принадлежит к группе поверхностей движения MbSweptSurface. Поверхность перемещения является частным случаем поверхности движения с криволинейной направляющей кривой. Поверхность перемещения описывается образующей кривой *MbCurve3D* * **curve**, направляющей кривой *MbCurve3D* * **spine**, точкой в начале направляющей MbCartPoint3D **origin**. Поверхность образована перемещением образующей кривой вдоль направляющей кривой. В частном случае образующая кривая поверхности перемещения может менять свою форму. В последнем случае в данных кривой присутствует вторая образующая кривая **brink**, точка в конце направляющей **ending**, начальный параметр *tmin* кривой **brink** и производная *dt* параметра кривой **brink** по параметру образующей кривой **curve**. Производная *dt* определяется равенством

$$dt = \frac{tmax - tmin}{umax - umin}, \tag{3.13}$$

где *tmax* — конечный параметр кривой **brink**. В общем случае указатель на вторую образующую кривую **brink** может быть равен нулю, что означает, что вторая образующая кривая отсутствует.

Первый параметр *u* поверхности совпадает с параметром образующей кривой **curve**. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения образующей кривой. Поверхность может быть периодической по первому параметру, если периодической является образующая кривая.

Второй параметр поверхности *v* совпадает с параметром направляющей кривой и принимает значения на отрезке: *vmin* ≤ *v* ≤ *vmax*. Поверхность может быть периодической по второму параметру, если периодической является направляющая кривая и отсутствует вторая образующая кривая.

В методе PointOn(double *u*, double *v*, MbCartPoint3D & **s**) в общем случае радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u, v) = \mathbf{spine}(v) + \mathbf{curve}(u) - \mathbf{origin} \tag{3.14}$$

Поверхность перемещения в общем случае приведена на рис. 1.

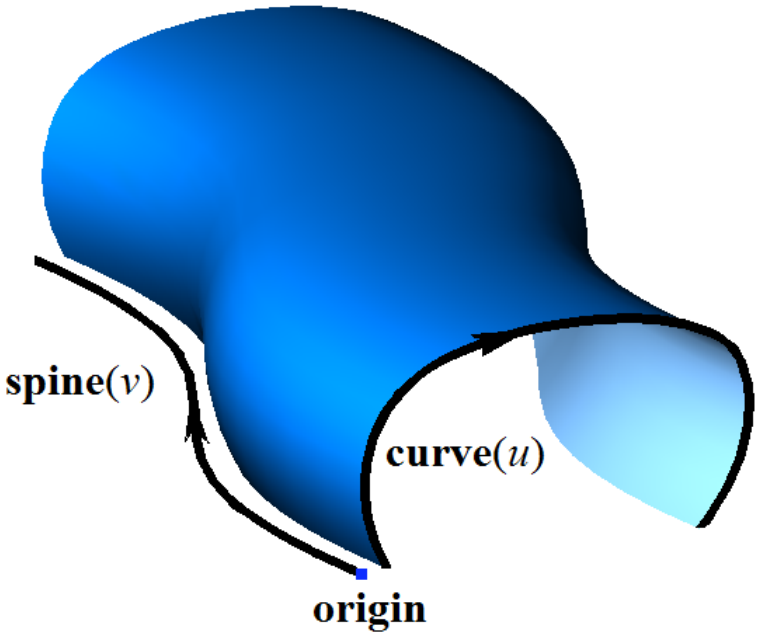


Рис. 1

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) в частном случае радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + (\mathbf{curve}(u) - \mathbf{origin})\,(1-w) + (\mathbf{brink}(t) - \mathbf{ending})\,w, \tag{3.15}$$

где

$$w = \frac{v - vmin}{vmax - vmin}, \quad t = tmin + (u - umin)\cdot dt. \tag{3.16}$$

Поверхность перемещения в частном случае приведена на рис. 2:

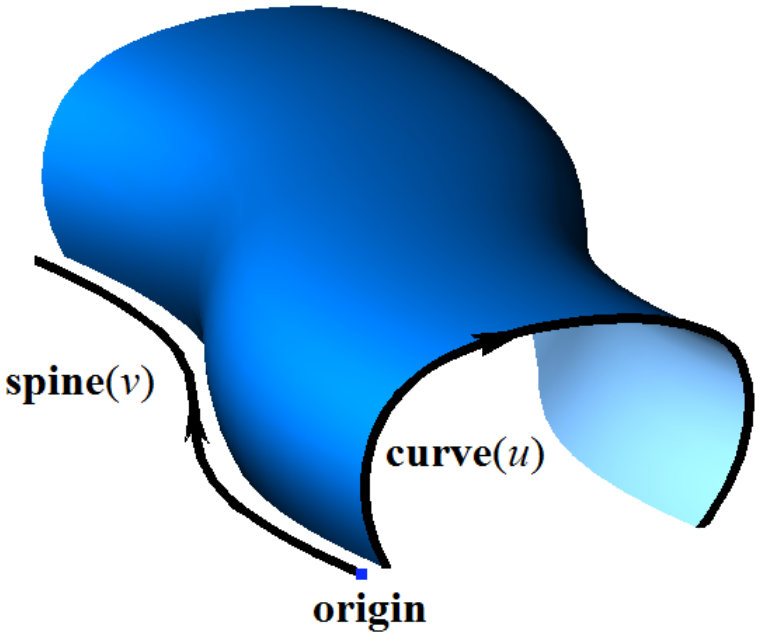


Рис. 2

Для отсутствия самопересечений поверхности образующая и направляющие кривые не должны иметь параллельных друг другу участков. В определённых случаях поверхность перемещения может иметь особые точки.

3.13.10 Спиральная поверхность MbSpiralSurface

Класс MbSpiralSurface объявлен в файле *surf_spiral_surface.h*.

Спиральная поверхность MbSpiralSurface принадлежит к группе поверхностей движения MbSweptSurface. Спиральная поверхность является частным случаем поверхности движения с направляющей кривой в форме цилиндрической спирали. Спиральная поверхность описывается образующей кривой *MbCurve3D* * **curve**, локальной системой координат *MbPlacement3D* **position**, вектор **position.axisZ** которой является осью спирали, радиусом спирали *radius*, шагом спирали *step*, положением начала спирали **origin** и граничными параметрами спирали *vmin* и *vmax*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Ось спирали совпадает с координатной осью **position.axisZ** локальной системы координат. Первый параметр поверхности *u* совпадает с параметром образующей кривой. Первый параметр поверхности принимает значения на отрезке $umin \le u \le umax$, который соответствует области определения образующей кривой. Поверхность может быть периодической по первому параметру, если периодической является образующая кривая.

Второй параметр поверхности *v* принимает значения на отрезке: $vmin \le v \le vmax$. Значение $v = 0$ соответствует точке на образующей кривой. Значение второго параметра *v*, равное 2π, соответствует точкам образующей кривой, смещённым на вектор **position.axisZ**, умноженный на *step*. Поверхность не может быть периодической по второму параметру.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s**

описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{position.origin} + radius\,(\cos(t)\,\mathbf{position.axisX} + \sin(t)\,\mathbf{position.axisY}) + \left(\frac{t \cdot step}{2\pi}\right)\mathbf{position.axisZ} + (\mathbf{cu} \tag{3.17}$$

где **M**(*v*) — матрица вращения. Заметим, что умножение вектора (**curve**(*u*) - **origin**) на матрицу **M**(*v*) выполняется справа. Матрица вращения имеет вид

$$M(v) = A^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot A = \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{position.axisX} \\ \mathbf{position.axisY} \\ \mathbf{position.axisZ} \end{bmatrix}. \tag{3.18}$$

Матрица **A** является матрицей преобразования координат радиуса-вектора точки из локальной системы координат **position** в глобальную систему координат. Строки матрицы **A** составлены из компонент базисных векторов локальной системы координат. Матрица **M**(*v*) переводит вектор **curve**(*u*) - **origin** в локальную систему координат, поворачивает его в ней на угол *v* вокруг оси вращения и возвращает повёрнутый вектор обратно в глобальную систему координат. Спиральная поверхность приведена на рис. 1:

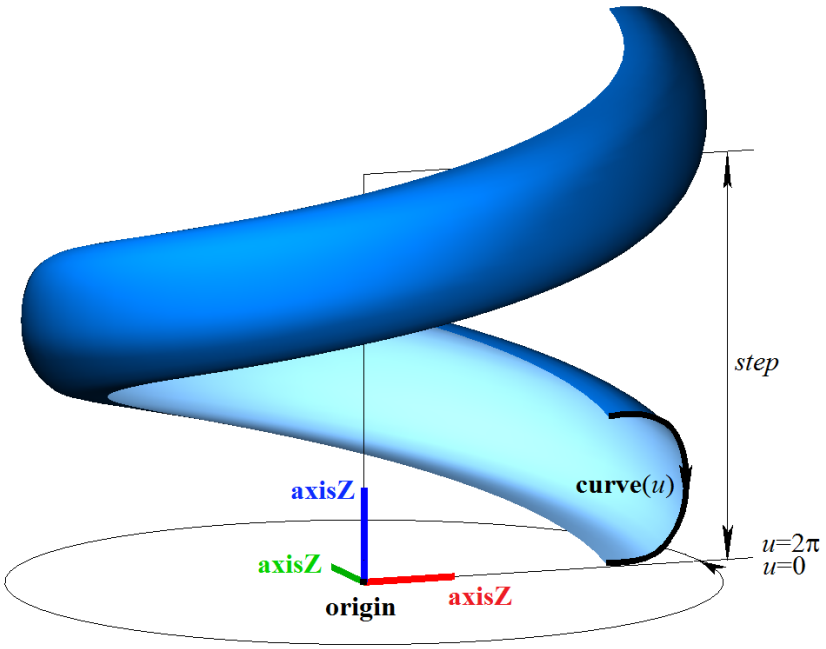


Рис. 1

Для граничных значений второго параметра поверхности должно соблюдаться неравенство: *vmin* < *vmax*.

3.13.11 Кинематическая поверхность MbEvolutionSurface

Класс MbEvolutionSurface объявлен в файле *surf_evolution_surface.h*.

Кинематическая поверхность MbEvolutionSurface принадлежит к группе поверхностей движения MbSweptSurface. Кинематическая поверхность является общим случаем поверхности движения с произвольной направляющей кривой. Поверхность вращения описывается образующей кривой *MbCurve3D* * **curve**, направляющим объектом

MbCurve3D * **spine**, положением начала направляющей *MbCartPoint3D* **origin**. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* совпадает с параметром образующей кривой **curve**. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения образующей кривой. Поверхность может быть периодической по первому параметру, если периодической является образующая кривая.

Направляющий объект **spine** заменяет собой направляющую кривую, построен на базе кривой и отличается от последней тем, что может генерировать локальную систему координат, связанную с кривой. Второй параметр поверхности *v* совпадает с параметром кривой направляющего объекта **spine**. Второй параметр поверхности принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*, который соответствует области определения направляющей кривой. Поверхность может быть периодической по второму параметру, если периодической является направляющая кривая.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{spine}(v) + (\mathbf{curve}(u) - \mathbf{origin})\,\mathbf{M}(v), \tag{3.19}$$

где **M**(*v*) — матрица, связанная с направляющей кривой. Заметим, что умножение вектора (**curve**(*u*) - **origin**) на матрицу **M**(*v*) выполняется справа. Матрица **M**(*v*) имеет вид

$$\mathbf{M}(v) = \mathbf{A}^{-1}(vmin)\cdot\mathbf{A}(v), \tag{3.20}$$

где матрица **A**(*v*) является матрицей преобразования координат радиуса-вектора точки из подвижной системы координат, связанной с направляющей кривой, в глобальную систему координат. Матрица **A**(*v*) зависит от второго параметра поверхности. Строки матрицы **A**(*v*) составлены из компонент базисных векторов подвижной системы координат:

$$A(v) = \begin{bmatrix} i_1(v) \\ i_2(v) \\ i_3(v) \end{bmatrix}, \tag{3.21}$$

где $\mathbf{i}_1(v)$ — касательный вектор направляющей кривой, $\mathbf{i}_2(v)$ — вектор, ортогональный $\mathbf{i}_1(v)$ и связанный с вектором **direction** направляющего объекта **spine**, $\mathbf{i}_3(v)$ — вектор, ортогональный $\mathbf{i}_1(v)$ и $\mathbf{i}_2(v)$. Поверхность вращения приведена на рис. 1:

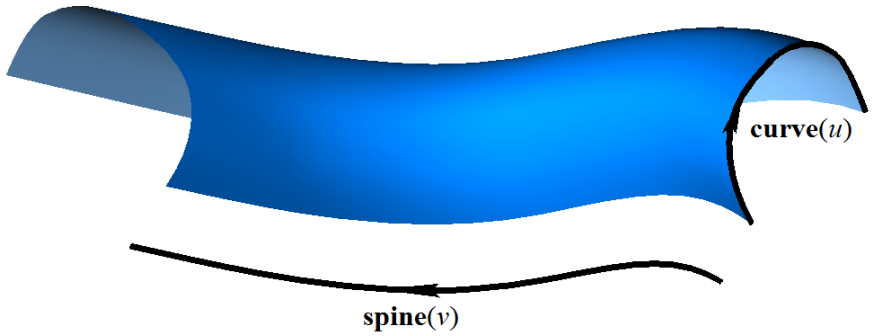


Рис. 1

Касательный вектор $\mathbf{i}_1(v)$ вычисляется по направляющей кривой. Вектор $\mathbf{i}_2(v)$ вычисляется из условия плавного изменения при переходе от точки к точке направляющей кривой и

ортогональности $\mathbf{i}_1(v)$. Вектор $\mathbf{i}_3(v)$ вычисляется как векторное произведение векторов $\mathbf{i}_1(v)$ и $\mathbf{i}_2(v)$.

3.13.12 Кинематическая поверхность с адаптацией MbExactionSurface

Класс MbExactionSurface объявлен в файле *surf_exaction_surface.h*.

Кинематическая поверхность с адаптацией MbExactionSurface является наследником кинематической поверхности MbEvolutionSurface. Кинематическая поверхность с адаптацией используется для построения тел кинематической операцией с направляющей в форме составной кривой, сегменты которой стыкуются с изломом, рис. 1:

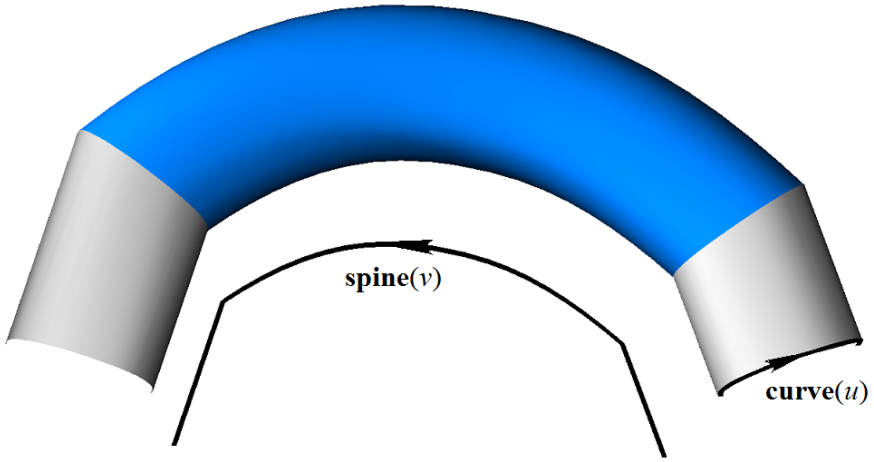


Рис. 1

Кинематическая поверхность MbExactionSurface подстраивает свои торцы для обеспечения стыковки с другой поверхностью.

3.13.13 Секториальная поверхность MbSectorSurface

Класс MbSectorSurface объявлен в файле *surf_sector_surface.h*.

Секториальная поверхность MbSectorSurface принадлежит к группе поверхностей движения MbSweptSurface. Секториальная поверхность описывается кривой *MbCurve3D* **curve** и точкой *MbCartPoint3D* **origin**.

Первый параметр поверхности *u* совпадает с параметром кривой **curve**. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривой **curve**. Поверхность может быть периодической по первому параметру, если периодической является кривая **curve**.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*. Значение *v* = *vmin* соответствует точке на кривой **curve**, значение *v* = *vmax* соответствует точке **origin**. Поверхность не может быть периодической по второму параметру.

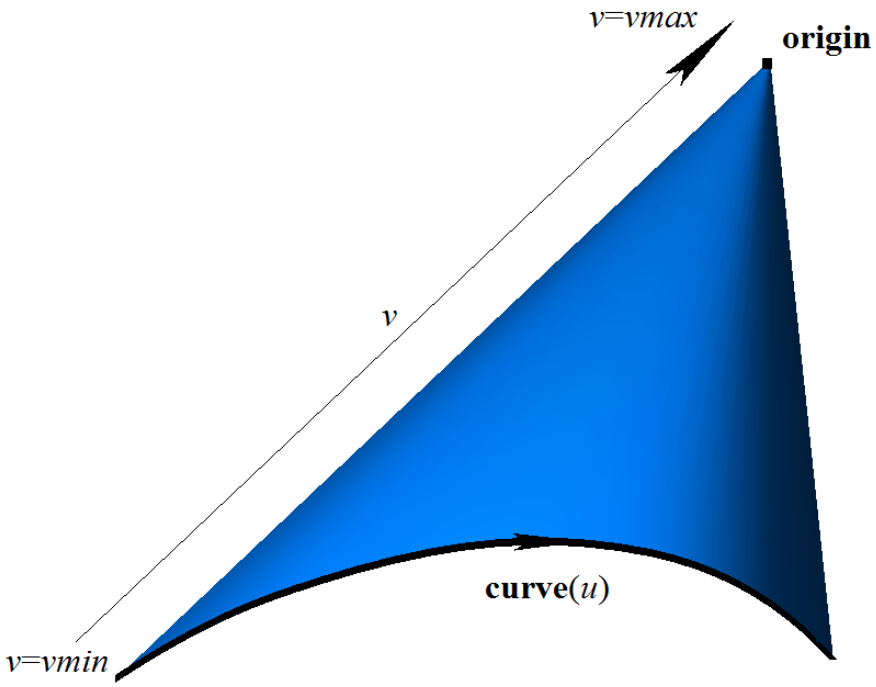
В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{curve}(u)(1-w) + \mathbf{origin}\,w, \tag{3.22}$$

где

$$w = \frac{v - vmin}{vmax - vmin}. \tag{3.23}$$

Секториальная поверхность приведена на рисунке:



Кривые поверхности **s**(*const*, *v*) являются отрезками прямой. Поверхность имеет полюс в точке **origin** при *v* = *vmax*. Секториальная поверхность является частным случаем линейчатой поверхности.

3.13.14 Линейчатая поверхность MbRuledSurface

Класс MbRuledSurface объявлен в файле *surf_ruled_surface.h*.

Линейчатая поверхность MbRuledSurface принадлежит к группе поверхностей движения MbSweptSurface. Линейчатая поверхность описывается кривой *MbCurve3D* * **curve**, кривой *MbCurve3D* * **sline**, признаком наличия полюса поверхности при начальном значении первого параметра *poleMin*, признаком наличия полюса поверхности при конечном значении первого параметра *poleMax*, начальным параметром *tmin* кривой **sline**, производной *dt* параметра кривой **sline** по параметру кривой **curve** и типом формы поверхности *type*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* совпадает с параметром кривой **curve**. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривой **curve**. Производная *dt* определяется равенством

$$dt = \frac{tmax - tmin}{umax - umin}, \tag{3.24}$$

где *tmax* — конечный параметр кривой **sline**. Поверхность может быть периодической по первому параметру, если периодическими являются кривые **curve** и **sline**.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*. Значение *v* = *vmin* соответствует точке на кривой **curve**, значение *v* = *vmax* соответствует точке на кривой **sline**. Поверхность не может быть периодической по второму параметру.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u, v) = \mathbf{curve}(u)(1 - w) + \mathbf{sline}(t)\, w, \tag{3.25}$$

где

$$w = \frac{v - vmin}{vmax - vmin}, \quad t = tmin + (u - umin) \cdot dt. \tag{3.26}$$

Линейчатая поверхность приведена на рис. 1:

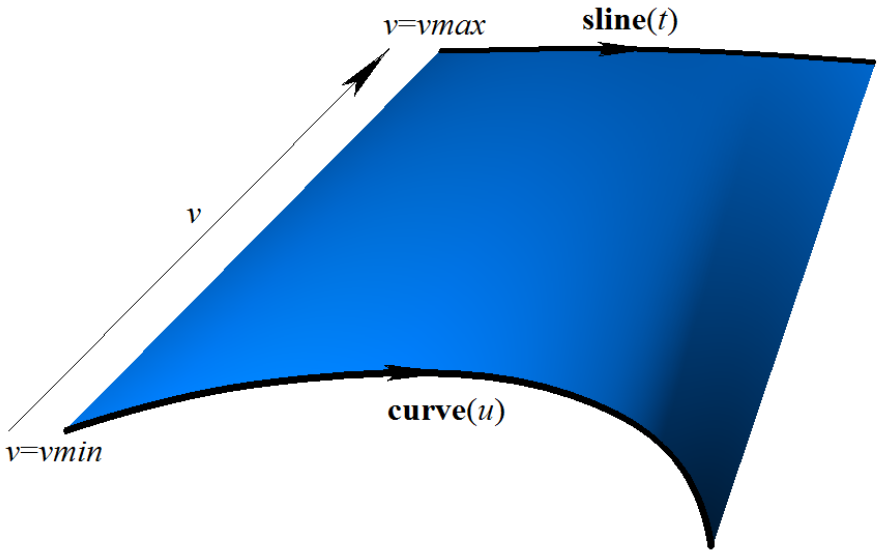


Рис. 1

Кривые поверхности **s**(*const*, *v*) с параметрами *u* = const являются отрезками прямой. Поверхность может иметь полюс, если одна из кривых **curve** или **sline** стянута в точку, или если кривые **curve** и **sline** совпадают на одном из краёв.

3.13.15 Поверхность на семействе кривых MbLoftedSurface

Класс MbLoftedSurface объявлен в файле *surf_lofted_surface.h*.

Поверхность MbLoftedSurface описывается множеством кривых RPArray< *MbCurve3D* > **uCurves**, множеством значений второго параметра поверхности для кривых *vParams*, множеством признаков одинаковых кривых *vLabels*, граничными значениями параметров *umin*, *umax*, *vmin*, *vmax*, признаками замкнутости поверхности по первому и второму параметрам *uClosed* и *vClosed*, вектором направления непериодической поверхности при *vmin* *MbVector3D* **derive1**, вектором направления непериодической поверхности при *vmax* *MbVector3D* **derive2**, признаками наличия полюсов поверхности на границе области определения *poleUMin*, *poleUMax*, *poleVMin*, *poleVMax*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* совпадает с параметрами кривых **curves**. Все кривые **curves** должны иметь одинаковую область определения параметра, для этого могут

использоваться кривые MbReperamCurve3D. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривых **curves**. Поверхность может быть периодической по первому параметру, если периодическими являются все кривые множества **curves**.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*. Значение *v* = *vmin* соответствует начальному значению множества *vParams*[0], значение *v* = *vmax* соответствует конечному значению множества *vParams*[*vParams*.MaxIndex()]. Поверхность может быть периодической по второму параметру.

Если все кривые множества **curves** разные, то значения множества *vLabels* равны индексу кривых множества **curves**. Если среди рядом расположенных кривых множества **curves** есть одинаковые кривые (но смещённые друг относительно друга), то соответствующие значения множества *vLabels* равны минимальному индексу размноженной кривой множества **curves**.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = (1-3w^2+2w^3)\,\mathbf{curves}[i](u)+(3w^2+2w^3)\,\mathbf{curves}[i+1](u)+(w-2w^2+w^3)\,\mathbf{derive}[i](u)+(-w^2+w^3)\,\mathbf{derive}[i+1](u) \tag{3.27}$$

где

$$w = \frac{v - vParams[i]}{vParams[i+1] - vParams[i]}, \tag{3.28}$$

derive[i] и **derive**[i+1] — производные кривых **curves**[i] и **curves**[i+1], соответственно. Индекс *i* рабочего участка для вычисления радиуса-вектора точки и его производных вычисляется из условия *vParams*[i] ≤ *v* ≤ *vParams*[i+1]. Если среди соседних элементов множества *vLabels* есть одинаковые значения, то между соответствующими кривыми поверхность эквивалентна поверхности выдавливания. Поверхность на семействе кривых приведена на рис. 1:

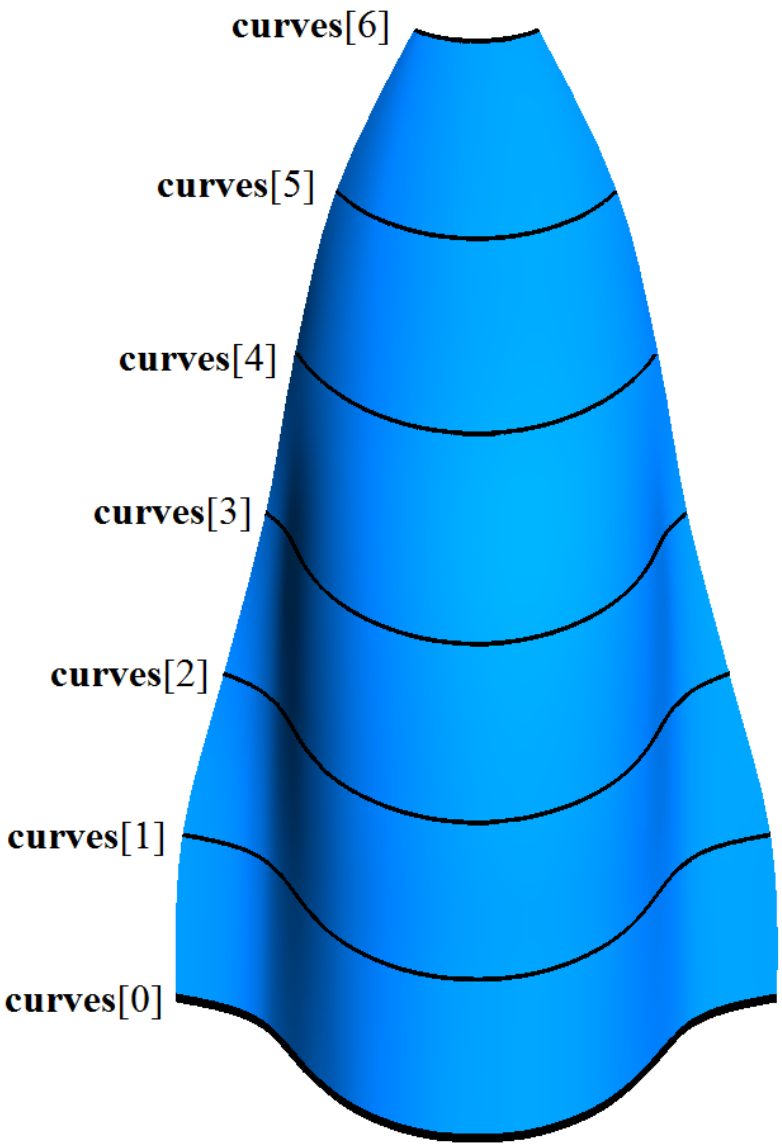


Рис. 1

Форма поверхности зависит от расположения кривых и от множества *vParams* значений параметров, при которых поверхность проходит по кривым. Для предотвращения самопересечений поверхности необходимо, чтобы значения множества *vParams* изменялись пропорционально среднему расстоянию между кривыми.

Кривые поверхности **s**(*const*, *v*) с параметрами *u* = const являются кривыми Эрмита *MbHermit3D*. Поверхность может иметь полюсы, если первая и/или последняя из кривых **curves** стянута в точку, или если все кривые **curves** совпадают на одном из краёв.

3.13.16 Поверхность на семействе кривых и направляющей MbElevationSurface

Класс MbElevationSurface объявлен в файле *surf_elevation_surface.h*.

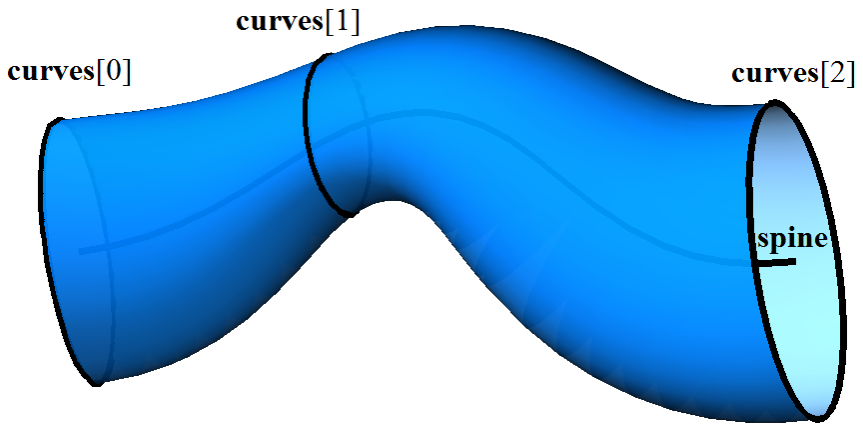
Поверхность на семействе кривых и направляющей является наследником клас-

са *MbLoftedSurface*. Так же как и поверхность *MbLoftedSurface* поверхность MbElevationSurface описывается множеством образующих кривых RPArray<*MbCurve3D*> **uCurves**, множеством значений второго параметра поверхности для кривых *vParams*, граничными значениями параметров *umin*, *umax*, *vmin*, *vmax*, признаками замкнутости поверхности по первому и второму параметрам *uClosed* и *vClosed*, признаками наличия полюсов поверхности на границе области определения *poleUMin*, *poleUMax*, *poleVMin*, *poleVMax*. Кроме перечисленных данных поверхность MbElevationSurface описывается направляющей кривой MbCurve3D **spine**. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* совпадает с параметрами кривых **curves**. Все кривые **curves** должны иметь одинаковую область определения параметра, для этого могут использоваться кривые MbReperamCurve3D. Первый параметр поверхности принимает значения на отрезке $umin \le u \le umax$, который соответствует области определения кривых **curves**. Поверхность может быть периодической по первому параметру, если периодическими являются все кривые множества **curves**.

Второй параметр поверхности *v* совпадает с параметром направляющей кривой **spine**. Второй параметр поверхности принимает значения на отрезке $vmin \le v \le vmax$, который соответствует области определения направляющей кривой. Поверхность может быть периодической по второму параметру, если периодической является направляющая кривая.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** вычисляется аналогично вычислению радиуса-вектора поверхности MbLoftedSurface с коррекцией смещения направляющей кривой. Поверхность на семействе кривых и направляющей приведена на рисунке:



Форма поверхности зависит от расположения образующих кривых, направляющей кривой и от множества *vParams* значений параметров, при которых поверхность проходит по кривым. Значения множества *vParams* определяются путём проецирования центров масс образующих кривых на направляющую кривую.

3.13.17 Поверхность на трёх кривых MbCornerSurface

Класс MbCornerSurface объявлен в файле *surf_corner_surface.h*.

Поверхность на трёх кривых MbCornerSurface описывается кривыми *MbCurve3D* * **curve0**, **curve1**, **curve2**, тремя точками *MbCartPoint3D* **vertex**[3] и тремя парами граничных значе-

ний параметров соответствующих кривых: *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* принимает значения на отрезке 0 ≤ *u* ≤ 1. Поверхность не может быть периодической по первому параметру. Поверхность имеет особую точку при минимальном значении первого параметра *u* = 0. В особой точке производная радиуса-вектора поверхности по второму параметру равна нулю.

Второй параметр поверхности *v* принимает значения на отрезке 0 ≤ *v* ≤ 1. Поверхность не может быть периодической по второму параметру.

Кривые **curve0**, **curve1**, **curve2** должны иметь точки пересечения или точки скрещения друг с другом. Параметры кривых *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max* вычисляются по точкам пересечения или скрещения кривых и определяют рабочие участки кривых и точки **vertex**[3]. Направления кривых для поверхности не имеют значения.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = w_0\,(\mathbf{curve2}(t_2) + \mathbf{curve1}(s_1) - \mathbf{vertex}[0]) + w_1\,(\mathbf{curve0}(t_0) + \mathbf{curve2}(s_2) - \mathbf{vertex}[1]) + w_2\,(\mathbf{curve1}(t_1) + \mathbf{cur} \tag{3.29}$$

где w_0 = 1 - *u*, w_1 = 0.5(*u* - *uv*), w_2 = 0.5(*u* + *uv*) — барицентрические координаты поверхности,

$$t_0 = w_2 \cdot t0min + (1 - w_2) \cdot t0max, \quad s_0 = (1 - w_1) \cdot t0min + w_1 \cdot t0max \tag{3.30}$$

— параметры кривой **curve0**,

$$t_1 = w_0 \cdot t1min + (1 - w_0) \cdot t1max, \quad s_1 = (1 - w_2) \cdot t1min + w_2 \cdot t1max \tag{3.31}$$

— параметры кривой **curve1**,

$$t_2 = w_1 \cdot t2min + (1 - w_1) \cdot t2max, \quad s_2 = (1 - w_0) \cdot t2min + w_0 \cdot t2max \tag{3.32}$$

— параметры кривой **curve2**.

Поверхность на трёх скрещивающихся кривых приведена на рис. 1:

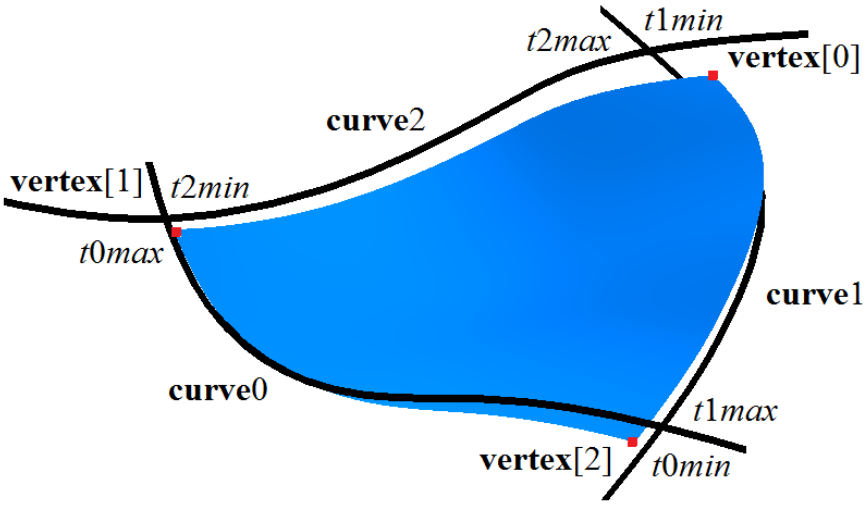


Рис. 1

На рис. 2 приведена совпадающая с частью поверхности сферы поверхность, построенная на трёх одинаковых дугах окружностей: плоскости дуг окружностей ортогональны друг другу, дуги пересекаются в крайних точках, каждая дуга содержит четверть окружности.

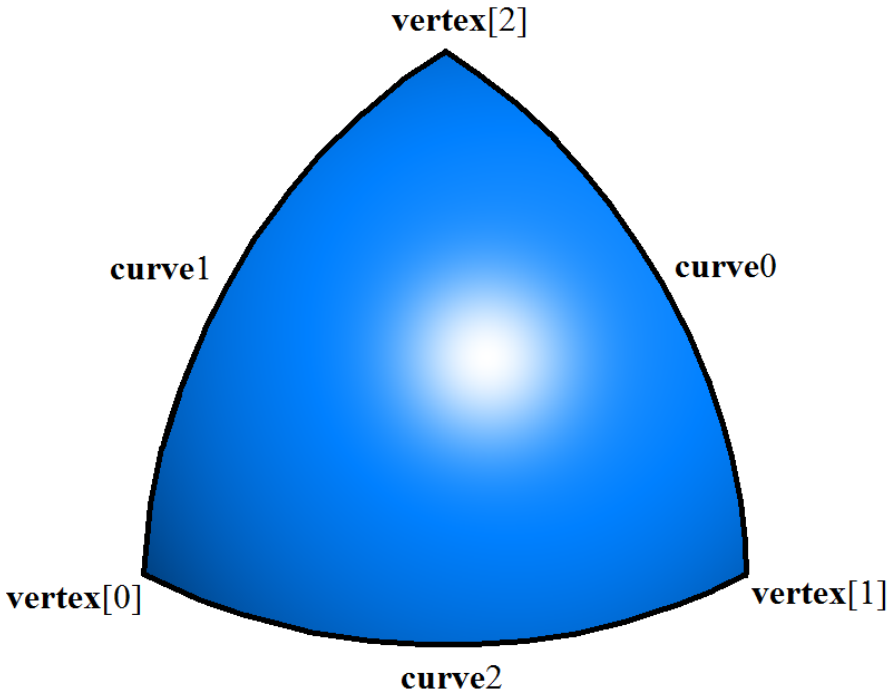


Рис. 2

Форма поверхности зависит от формы кривых. Если кривые не пересекаются, то поверхность не проходит по кривым. Если кривые пересекаются, то точки **vertex**[3] расположены в точках пересечения, и поверхность проходит по участкам кривых: при $w_0 = 0$ поверхность проходит по участку кривой **curve0**, при $w_1 = 0$ поверхность проходит по участку кривой **curve1**, при $w_2 = 0$ поверхность проходит по участку кривой **curve2**.

3.13.18 Поверхность Кунса MbCoverSurface

Класс MbCoverSurface объявлен в файле *surf_cover_surface.h*.

Поверхность Кунса MbCoverSurface описывается кривыми *MbCurve3D* **curve0**, **curve1**, **curve2**, **curve3**, четырьмя точками *MbCartPoint3D* **vertex**[4], четырьмя парами граничных значений параметров соответствующих кривых: *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max*, признаком периодичности поверхности по первому параметру *uclosed*, признаком периодичности поверхности по второму параметру *vclosed*, признаком наличия полюса поверхности при начальном значении первого параметра *poleUMin*, признаком наличия полюса поверхности при конечном значении первого параметра *poleUMax*, признаком наличия полюса поверхности при начальном значении второго параметра *poleVMin*, признаком наличия полюса поверхности при конечном значении второго параметра *poleVMax*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности u принимает значения на отрезке $0 \le u \le 1$. Поверхность

может быть периодической по первому параметру, если периодическими являются кривые **curve0** и **curve2**.

Второй параметр поверхности *v* принимает значения на отрезке 0 ≤ *v* ≤ 1. Поверхность может быть периодической по второму параметру, если периодическими являются кривые **curve1** и **curve3**.

Соседние кривые **curve0**, **curve1**, **curve2**, **curve3** должны иметь точки пересечения или точки скрещения друг с другом. Параметры кривых *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max* вычисляются по точкам пересечения или скрещения кривых и определяют рабочие участки кривых и точки **vertex**[4]. Направления кривых для поверхности не имеют значения.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = (1-v)\,(\mathbf{curve0}(t_0) - (1-u)\mathbf{vertex}[0]) + u\,(\mathbf{curve1}(t_1) - (1-v)\mathbf{vertex}[1]) + v\,(\mathbf{curve2}(t_2) - u\mathbf{vertex}[2]) + (1-u \tag{3.33}$$

где w_0 = 1 - *u*, w_1 = 0.5(*u* - *uv*), w_2 = 0.5(*u* + *uv*) — барицентрические координаты поверхности,

$$t_0 = (1-u)\cdot t0min + u\cdot t0max \tag{3.34}$$

— параметр кривой **curve0**,

$$t_1 = (1-v)\cdot t1min + v\cdot t1max \tag{3.35}$$

— параметр кривой **curve1**,

$$t_2 = (1-u)\cdot t2min + u\cdot t2max \tag{3.36}$$

— параметр кривой **curve2**,

$$t_3 = (1-v)\cdot t3min + v\cdot t3max \tag{3.37}$$

— параметр кривой **curve3**.

Поверхность Кунса на четырёх скрещивающихся кривых приведена на рис. 1.

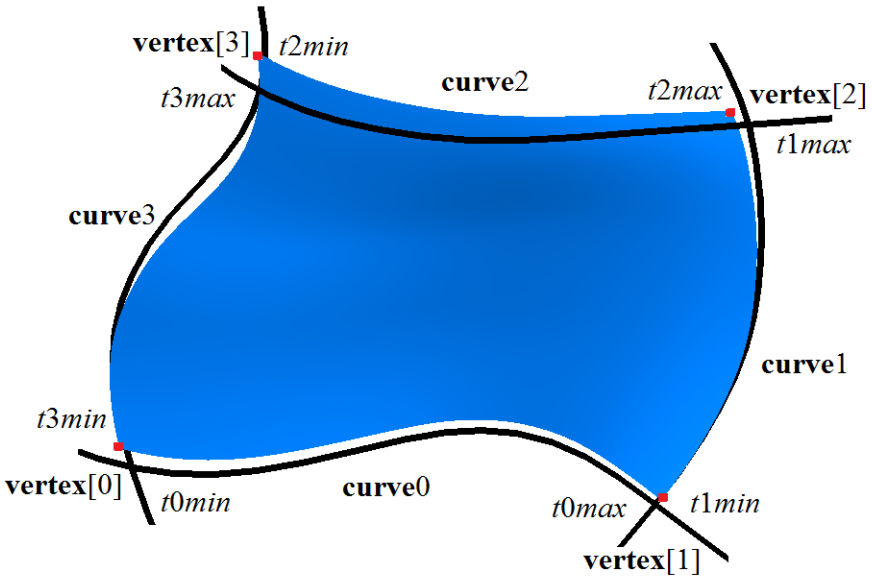


Рис. 1

Форма поверхности зависит от формы кривых. Если соседние кривые пересекаются в точках **vertex**[4], то поверхность проходит по участкам кривых: при *u* = 0 поверхность проходит по участку кривой **curve3**, при *v* = 0 поверхность проходит по участку кривой **curve0**, при *u* = 1 поверхность проходит по участку кривой **curve1**, при *v* = 1 поверхность проходит по участку кривой **curve2**.

3.13.19 Поверхность Кунса MbCoonsPatchSurface

Класс MbCoonsPatchSurface объявлен в файле *surf_coons_surface.h*.

Бикубическая поверхность Кунса MbCoonsPatchSurface построена аналогично поверхности Кунса *MbCoverSurface* и имеет дополнительные условия для радиуса-вектора на краях. Бикубическая поверхность Кунса MbCoonsPatchSurface описывается четырьмя кривыми MbCurve3D * **curve0**, **curve1**, **curve2**, **curve3**, производной **curve0V** радиуса-вектора поверхности вдоль кривой **curve0** по второму параметру поверхности, производной **curve1U** радиуса-вектора поверхности вдоль кривой **curve1** по первому параметру поверхности, производной **curve2V** радиуса-вектора поверхности вдоль кривой **curve2** по второму параметру поверхности, производной **curve3U** радиуса-вектора поверхности вдоль кривой **curve3** по первому параметру поверхности, четырьмя угловыми точками **vertex**[4], четырьмя производными по первому параметру поверхности в угловых точках **vertexU**[4], четырьмя производными по второму параметру поверхности в угловых точках **vertexV**[4], четырьмя смешанными производными по первому и второму параметрам поверхности в угловых точках **vertexUV**[4], четырьмя парами граничных значений параметров соответствующих кривых: *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max*, признаком периодичности поверхности по первому параметру *uclosed*, признаком периодичности поверхности по второму параметру *vclosed*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Первый параметр поверхности *u* принимает значения на отрезке $0 \le u \le 1$. Поверхность может быть периодической по первому параметру, если периодическими являются кривые **curve0**, **curve2**, **curveV0**, **curveV2**, а кривые **curveU1** и **curveU3** совпадают.

Второй параметр поверхности *v* принимает значения на отрезке $0 \le v \le 1$. Поверхность может быть периодической по второму параметру, если периодическими являются кривые **curve1** и **curve3**, **curveU1**, **curveU3**, а кривые **curveV0** и **curveV2** совпадают.

Соседние кривые **curve0**, **curve1**, **curve2**, **curve3** должны иметь точки пересечения или точки скрещения друг с другом. Параметры кривых *t0min*, *t0max*, *t1min*, *t1max*, *t2min*, *t2max*, *t3min*, *t3max* вычисляются по точкам пересечения или скрещения кривых и определяют рабочие участки кривых, точки **vertex**[4], **vertexU**[4], **vertexV**[4], **vertexUV**[4]. Направления кривых для поверхности не имеют значения, но параметризация пар кривых **curve0** и **curveV0**, **curve2** и **curveV2**, **curve1** и **curveU1**, **curve3** и **curveU3** должна совпадать.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией, которая приведена в книге «Геометрическое моделирование» Голованова Н.Н. Бикубическая поверхность Кунса строится по заранее рассчитанным данным и используется для построения сопряжений и заплаток с условиями сопряжения на краях. Бикубическая поверхность Кунса приведена на рис. 1.

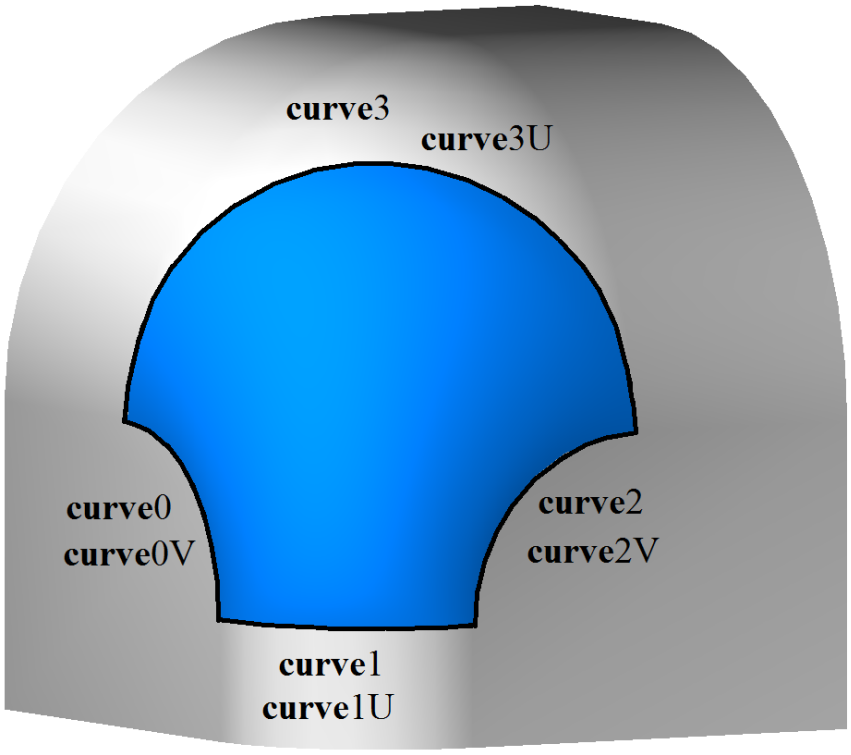


Рис. 1

Форма поверхности зависит от формы кривых и производных. Если соседние кривые пересекаются в точках **vertex**[4], то поверхность проходит по участкам кривых: при *u* = 0 поверхность проходит по участку кривой **curve3**, при *v* = 0 поверхность проходит по участку кривой **curve0**, при *u* = 1 поверхность проходит по участку кривой **curve1**, при *v* = 1 поверхность проходит по участку кривой **curve2**.

3.13.20 Поверхность на сети кривых MbMeshSurface

Класс MbMeshSurface объявлен в файле *surf_mesh_surface.h*.

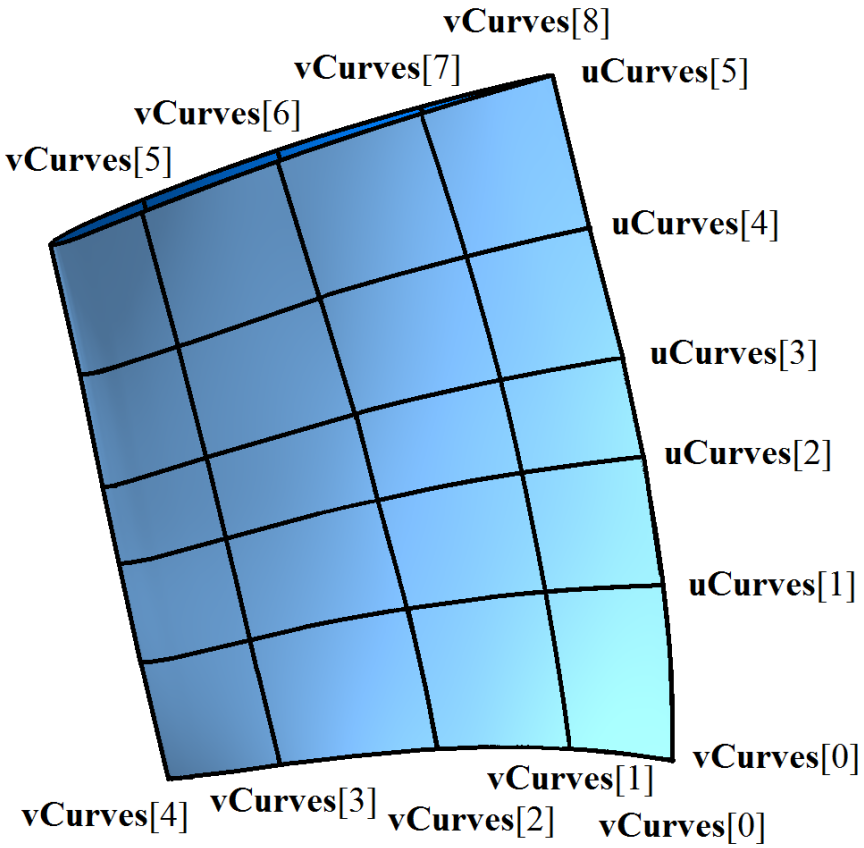
Поверхность на сети кривых MbMeshSurface описывается множеством кривых RPArray< *MbCurve3D* > **uCurves**, множеством кривых RPArray< *MbCurve3D* > **vCurves**, множеством значений первого параметра поверхности *uParams*, множеством значений второго параметра поверхности *vParams*, граничными значениями первого параметра поверхности *umin* и *umax*, граничными значениями второго параметра поверхности *vmin* и *vmax*, признаками наличия полюса поверхности при граничных значениях первого параметра *poleUMin*, *poleUMax*, признаками наличия полюса поверхности при граничных значениях второго параметра *poleVMin*, *poleVMax*, признаком периодичности поверхности по первому параметру *uclosed*, признаком периодичности поверхности по второму параметру *vclosed*, типом *type0* сопряжения поверхности на краю, соответствующем второму параметру *vmin*, типом *type1* сопряжения поверхности на краю, соответствующем первому параметру *umin*, типом *type2* сопряжения поверхности на краю, соответствующем второму параметру *vmax*, типом *type3* сопряжения поверхности на краю, соответствующем первому параметру *umax*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Количество элементов множества кривых **vCurves** и множества значений первого параметра поверхности *uParams* согласованы так, что точкам кривой **vCurves**[*j*] соответствует параметр *uParams*[*j*]. Первый параметр поверхности *u* принимает значения на отрезке *uParams*[0] ≤ *u* ≤ *uParams*[*uParams*.MaxIndex()]. Поверхность может быть периодической по первому параметру, если периодическими являются все кривые **uCurves**.

Количество элементов множества кривых **uCurves** и множества значений второго параметра поверхности *vParams* согласованы так, что точкам кривой **uCurves**[*i*] соответствует параметр *vParams*[*i*]. Второй параметр поверхности *v* принимает значения на отрезке *vParams*[0] ≤ *v* ≤ *vParams*[*vParams*.MaxIndex()]. Поверхность может быть периодической по второму параметру, если периодическими являются все кривые **vCurves**.

Каждая кривая **uCurves**[*i*] должна иметь точки пересечения или точки скрещения с каждой кривой **vCurves**[*j*]. Соседние кривые множества **uCurves** не должны иметь противоположные направления. Соседние кривые множества **vCurves** также не должны иметь противоположные направления.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией, которая приведена в книге «Геометрическое моделирование», автор Голованов Н.Н. Поверхность на сети кривых приведена на рисунке:



Форма поверхности зависит от формы кривых, их взаимного расположения и значений параметров множеств *uParams* и *vParams*. Если каждая кривая **uCurves**[*i*] пересекается с каждой кривой **vCurves**[*j*], то поверхность проходит по кривым **vCurves**[*j*] при параметрах *u* = *uParams*[*j*] и по кривым **uCurves**[*i*] при параметрах *v* = *vParams*[*i*].

3.13.21 Поверхность соединения MbJoinSurface

Класс MbJoinSurface объявлен в файле *surf_joint_surface.h*.

Поверхность соединения MbJoinSurface описывается множеством кривых RPArray< *MbCurve3D* > **curves**, узловым вектором *knots*, порядком сплайна *degree*, граничными значениями первого параметра поверхности *umin* и *umax*, признаком периодичности первого параметра поверхности *closedU*, признаком периодичности второго параметра поверхности *closedV*, признаком наличия полюса поверхности при граничных значениях первого параметра *isPoleUmin*, *isPoleUmax*, признаком наличия полюса поверхности при граничных значениях второго параметра *isPoleUmin*, *isPoleUmax*.

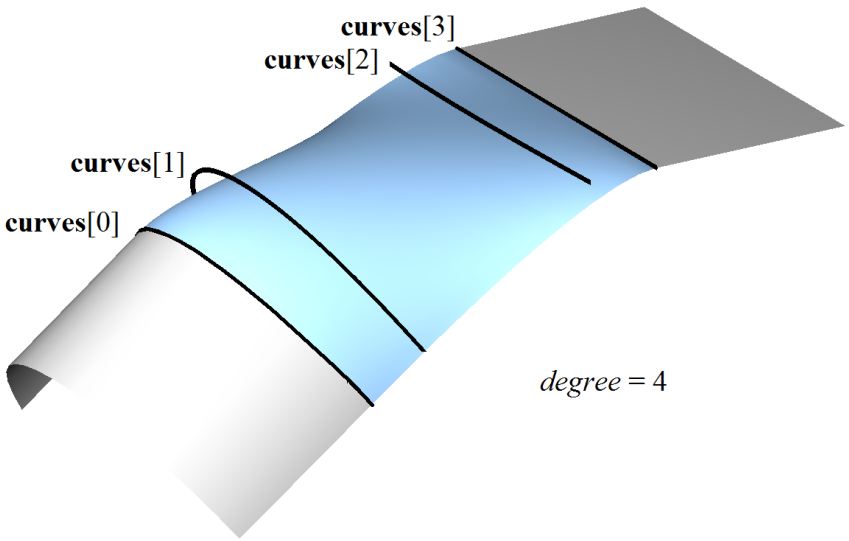
Кривые **curves** согласованы друг с другом: имеют одинаковое направление и одинаковую область определения параметра. Первый параметр поверхности *u* совпадает с параметром кривых **curves**, который является общим для них. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривых **curves**. Поверхность может быть периодической по первому параметру, если периодическими являются все кривые **curves**. Пусть узловой вектор *knots* содержит *knotsCount* элементов, а количество кривых в множестве **curves** равно *curvesCount*. Количества элементов в множествах связаны равенством *curvesCount* + *degree* = *knotsCount*.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*, где *vmin* = *knots*[*degree* − 1], *vmax* = *knots*[*knotsCount* − *degree*]. Поверхность не может быть периодической по второму параметру.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$s(u,v) = \frac{\sum_{j=0}^{\text{curvesCount}-1} N_j^{degree}(v)\,\text{curves}[j](u)}{\sum_{j=0}^{\text{curvesCount}-1} N_j^{degree}(v)},$$

где $N_j^{degree}(v)$ — B-сплайны порядка *degree* для *j*-ой кривой **curves**[*j*]. Поверхность соединения приведена на рисунке:



Каждая кривая **s**(*const*, *v*) с фиксированным первым параметром поверхности *u* = const является NURBS-кривой порядка *degree*, построенной по точкам **curves**[*i*](*const*).

3.13.22 NURBS-поверхность MbSplineSurface

Класс MbSplineSurface объявлен в файле *surf_spline_surface.h*.

NURBS-поверхность (NonUniform Rational B-Spline поверхность) MbSplineSurface описывается контрольными точками SArray<MbCartPoint3D> **points**[*i*][*j*], *i* = 0, 1, ..., *vcount* − 1, *j* = 0, 1, ..., *ucount* − 1, условно расположенными в узлах прямоугольной таблицы с *ucount* колонками и *vcount* строками, весами контрольных точек, заданных в таблице *weight*[*i*][*j*], порядком B-сплайнов вдоль первого параметра поверхности *udegree*, порядком B-сплайнов вдоль второго параметра поверхности *vdegree*, узловым вектором вдоль первого параметра *uknots*, узловым вектором вдоль второго параметра *vknots*, признаками периодичности поверхности по первому и второму параметрам *uclosed* и *vclosed*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Порядок B-сплайнов вдоль параметров поверхности совпадает с порядком разделённой разности, по которой вычисляются соответствующие B-сплайны. Узловые векторы *uknots* и *vknots* представляют собой неубывающие последовательности действительных чисел и определяют область определения параметров поверхности и её форму. Пусть узловой вектор *uknots* содержит *uknotsCount* элементов, а количество точек в каждой строке прямоугольной таблицы равно *ucount*. Для непериодической по первому параметру NURBS-поверхности количества элементов в множествах связаны равенством *ucount* + *degree* = *uknotsCount*. Для периодической NURBS-поверхности количество элементов в множествах связаны равенством *knotsCount* + 2 *degree* − 1 = *knotsCount*.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$r(u,v) = \frac{\sum_{i=0}^{\text{vcount}-1} \sum_{j=0}^{\text{ucount}-1} N_i^{vdegree}(v) N_j^{udegree}(u) \text{weight}[i][j] \text{points}[i][j]}{\sum_{i=0}^{\text{vcount}-1} \sum_{j=0}^{\text{ucount}-1} N_i^{vdegree}(v) N_j^{udegree}(u) \text{weight}[i][j]},$$

где $N_i^{vdegree}(v)$ и $N_j^{udegree}(u)$ — B-сплайны.

На рис. 1 приведены отрезки, связывающие соседние контрольные точки **points**[*i*][*j*].

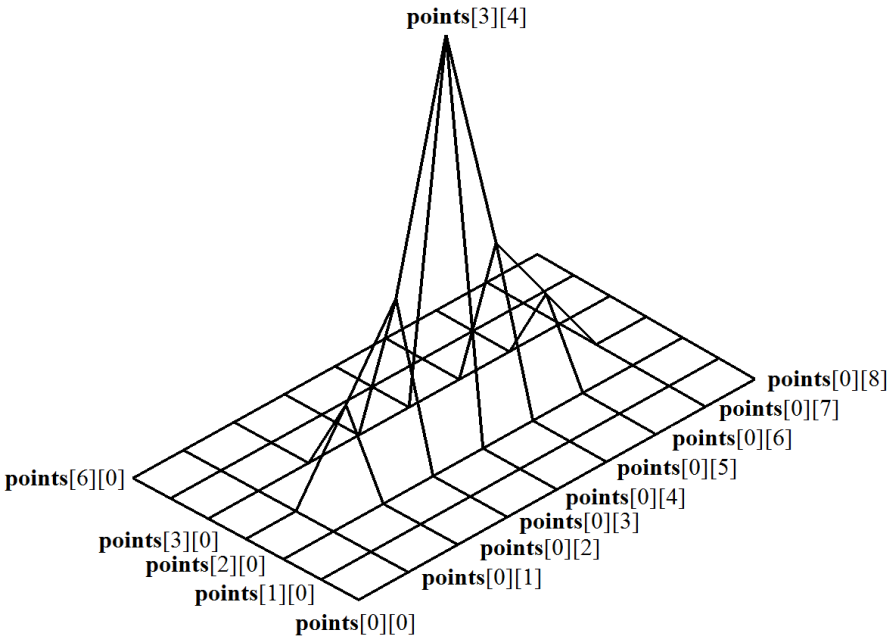


Рис. 1

Рис. 1 демонстрирует условное расположение контрольных точек в узлах прямоугольной таблицы. NURBS-поверхность, построенная на тех же контрольных точках, приведена на рис. 2:

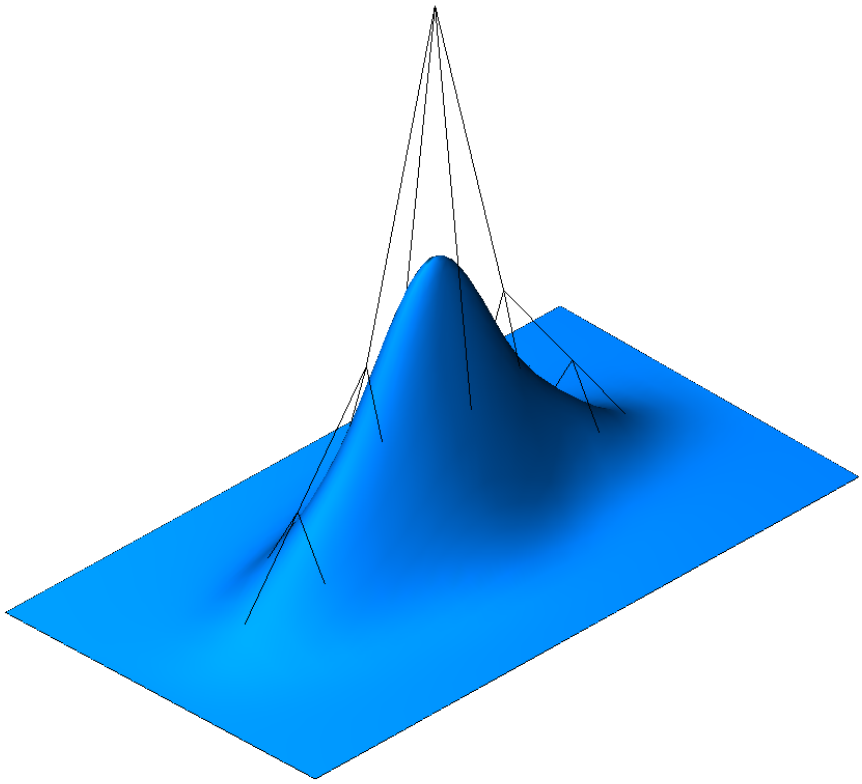


Рис. 2

Кривые на поверхности **s**(*const*, *v*) и **s**(*u*, *const*) с параметрами *u* = const или *v* = const представляют собой B-кривые степени *udegree* и *vdegree* соответственно. По первому параметру поверхность имеет порядок *udegree*, а по второму параметру поверхность имеет порядок *vdegree*. Область изменения параметров непериодической NURBS-поверхности представляет собой прямоугольник: *uknots*[*udegree* − 1] ≤ *u* ≤ *uknots*[*ucount*], *vknots*[*vdegree* − 1] ≤ *v* ≤ *vknots*[*vcount*].

Поверхность может быть периодической как по первому параметру, так и по второму параметру. Для периодически замкнутой поверхности узловые векторы *uknots* и *vknots* имеют на *udegree* − 1 и *vdegree* − 1 элементов соответственно больше. Область изменения параметров периодической по обоим параметрам NURBS-поверхности представляет собой прямоугольник: *uknots*[*udegree* − 1] ≤ *u* ≤ *uknots*[*ucount* + *udegree* − 1], *vknots*[*vdegree* − 1] ≤ *v* ≤ *vknots*[*vcount* + *vdegree* − 1].

Каждая поверхность может построить свою NURBS-копию виртуальным методом NurbsSurface(const MbNurbsParameters & *uParam*, const MbNurbsParameters & *vParam*).

3.13.23 Эквидистантная поверхность MbOffsetSurface

Класс MbOffsetSurface объявлен в файле *surf_offset_surface.h*.

Эквидистантная поверхность MbOffsetSurface описывается базовой поверхностью *MbSurface* *basisSurface*, смещением вдоль нормали базовой поверхности *distance*, граничными значениями первого параметра базовой поверхности *u0min*, *u0max*, граничными значениями второго параметра базовой поверхности *v0min*, *v0max*, признаками периодичности базовой поверхности *u0closed*, *v0closed*, увеличениями граничных значений первого параметра базовой поверхности *dumin*, *dumax*, увеличениями граничных

значений второго параметра базовой поверхности *dvmin*, *dvmax*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Вычисление радиус-вектора точки эквидистантной поверхности выполняется следующим образом. Для заданного параметра вычисляется точка базовой поверхности и нормаль в этой точке.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{r}(u,v) = \mathbf{basisSurface}(u,v) + \mathbf{normal}(u,v) \cdot distance,$$

где **normal**(*u*, *v*) — нормаль поверхности в заданной точке. Эквидистантная поверхность и её базовая поверхность приведены на рис. 1.

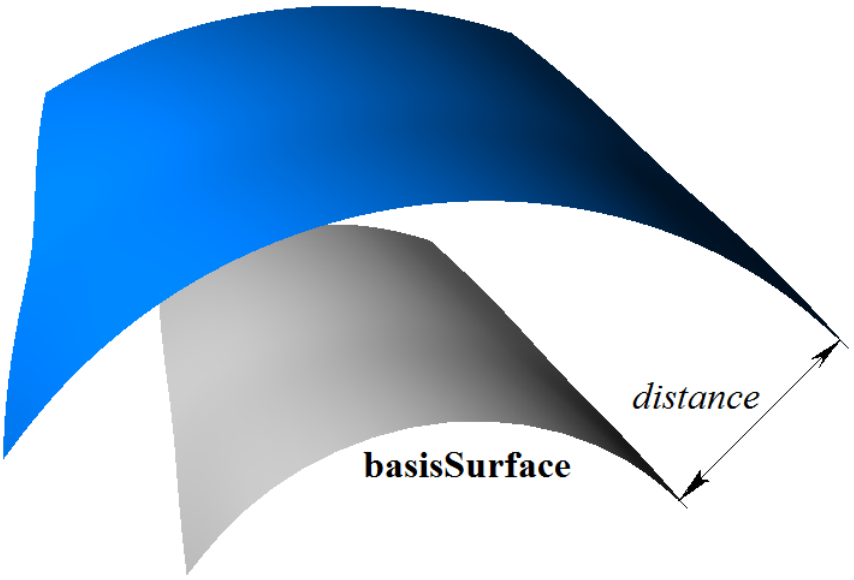


Рис. 1

Область изменения параметров эквидистантной поверхности может отличаться от области изменения параметров базовой поверхности. Область изменения первого параметра эквидистантной поверхности определяется неравенствами: *u0min* + *dumin* ≤ *u* ≤ *u0max* + *dumax*. Область изменения второго параметра эквидистантной поверхности определяется неравенствами: *v0min* + *dvmin* ≤ *v* ≤ *v0max* + *dvmax*. Эквидистантная поверхность с отрицательным смещением и увеличенной областью определения параметров и её базовая поверхность приведены на рис. 2.

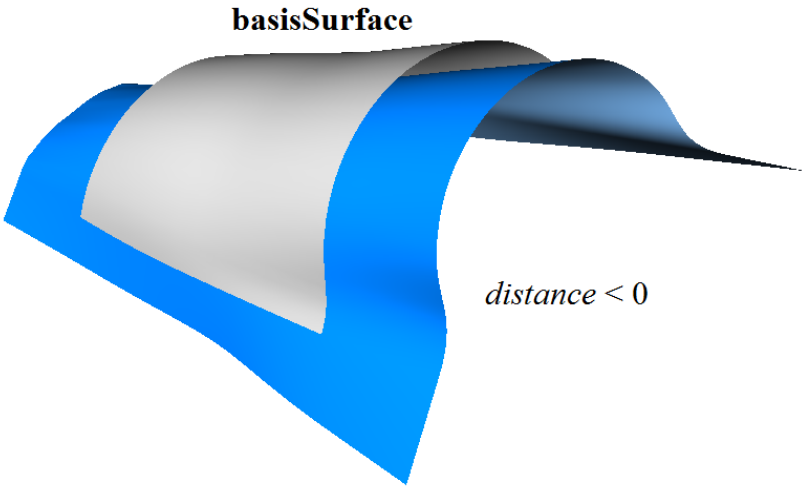


Рис. 2

В качестве базовой поверхности для эквидистантной поверхности не должна использоваться другая эквидистантная поверхность, а должна использоваться базовая поверхность последней с соответствующим пересчётом величины смещения.

Каждая поверхность может построить эквидистантную поверхность виртуальным методом Offset(double *distance*, bool *sense*).

3.13.24 Поверхность фаски MbChamferSurface

Класс MbChamferSurface объявлен в файле *surf_chamfer_surface.h*.

Поверхность фаски MbChamferSurface принадлежит к группе поверхностей сопряжения MbSmoothSurface. Поверхность фаски описывается кривой на первой сопрягаемой поверхности *MbSurfaceCurve* *curve1*, кривой на второй сопрягаемой поверхности *MbSurfaceCurve* *curve2*, способом построения фаски *form*, катетами фаски *distance1* и *distance2*, граничными значениями *umin* и *umax* параметра кривых *curve1* и *curve2*, граничными значениями второго параметра поверхности *vmin* и *vmax*, признаком периодичности первого параметра поверхности *uclosed*, признаком наличия полюса поверхности при начальном значении первого параметра *poleMin*, признаком наличия полюса поверхности при конечном значении первого параметра *poleMax*.

Кривые *curve1* и *curve2* согласованы друг с другом и имеют одинаковую область определения параметра. Первый параметр поверхности *u* совпадает с параметром кривых *curve1* и *curve2*, который является общим для них. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривых *curve1* и *curve2*. Поверхность может быть периодической по первому параметру, если периодическими являются кривые *curve1* и *curve2*.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*. Значение *v* = *vmin* соответствует точке на кривой *curve1*, значение *v* = *vmax* соответствует точке на кривой *curve2*. Поверхность не может быть периодической по второму параметру.

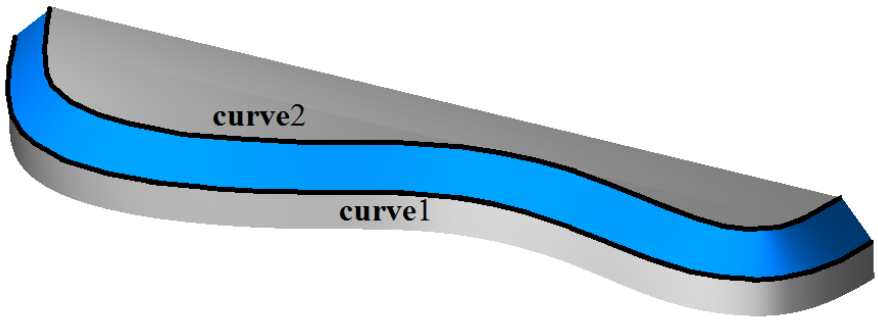
В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{curve1}(u)\,(1-w) + \mathbf{curve2}(u)\,w,$$

где

$$w = \frac{v - \text{vmin}}{\text{vmax} - \text{vmin}}.$$

Поверхность фаски приведена на рисунке:



Кривые поверхности **s**(*const*, *v*) с параметрами *u* = const являются отрезками прямой. Поверхность может иметь полюс при *u* = *umin* и *u* = *umax*, если соответствующие края кривых *curve1* и *curve2* совпадают.

3.13.25 Поверхность скругления MbFilletSurface

Класс MbFilletSurface объявлен в файле *surf_fillet_surface.h*.

Поверхность скругления MbFilletSurface принадлежит к группе поверхностей сопряжения MbSmoothSurface. Поверхность скругления описывается кривой на первой сопрягаемой поверхности *MbSurfaceCurve curve1*, кривой на второй сопрягаемой поверхности *MbSurfaceCurve curve2*, кривой MbCurve3D *curve0*, функцией *MbFunction weights0* веса кривой *curve0*, способом построения скругления *form*, радиусами скругления *distance1* и *distance2*, коэффициентом формы *conic*, граничными значениями *umin* и *umax* параметра кривых *curve1*, *curve2*, *curve0* и функции *weights0*, граничными значениями второго параметра поверхности *vmin* и *vmax*, признаком периодичности первого параметра поверхности *uclosed*, признаком наличия полюса поверхности при начальном значении первого параметра *poleMin*, признаком наличия полюса поверхности при конечном значении первого параметра *poleMax*, признаком равномерной параметризации поверхности по второму параметру *even*, признаком гладкого сопряжения поверхности с сопрягаемыми поверхностями *equable*, признаком *byCurve1* наличия кромки вдоль кривой *curve2* или *curve1* (*equable* = false). У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Кривые *curve1*, *curve2*, *curve0* и функция *weights0* согласованы друг с другом и имеют одинаковую область определения параметра. Первый параметр поверхности *u* совпадает с параметром кривых *curve1*, *curve2*, *curve0* и функции *weights0*, который является общим для них. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривых *curve1*, *curve2*, *curve0* и функции *weights0*. Поверхность может быть периодической по первому параметру, если периодическими являются кривые *curve1*, *curve2*, *curve0* и функция *weights0*.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*. Значение *v* = *vmin* соответствует точке на кривой *curve1*, значение *v* = *vmax* соответствует точке на кривой *curve2*. Поверхность не может быть периодической по второму параметру.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$s(u,v) = \frac{(1-v)^2\,\mathbf{curve1}(u) + 2t(1-v)\,\mathrm{weight0}(u)\,\mathbf{curve0}(u) + v^2\,\mathbf{curve2}(u)}{(1-v)^2 + 2v(1-v)\,\mathrm{weight0}(u) + v^2}.$$

$$.. + 2v(1-v)?$$

Каждая кривая **s**(*const*, *v*) с фиксированным первым параметром поверхности *u* = const является NURBS-кривой третьего порядка, построенной по точкам *curve1*(*u*), *curve0*(*u*), *curve2*(*u*), у которой крайние точки имеют единичный вес, а средняя точка *curve0*(*u*) имеет вес *weights0*(*u*). При *conic* = _ARC_ функция *weights0*(*u*) вычисляется из условия, что все кривые **s**(*const*, *v*) являются дугами окружности. При *conic* ≠ _ARC_ функция *weights0* является константой и равна коэффициенту формы *conic*. Поверхность скругления приведена на рис. 1.

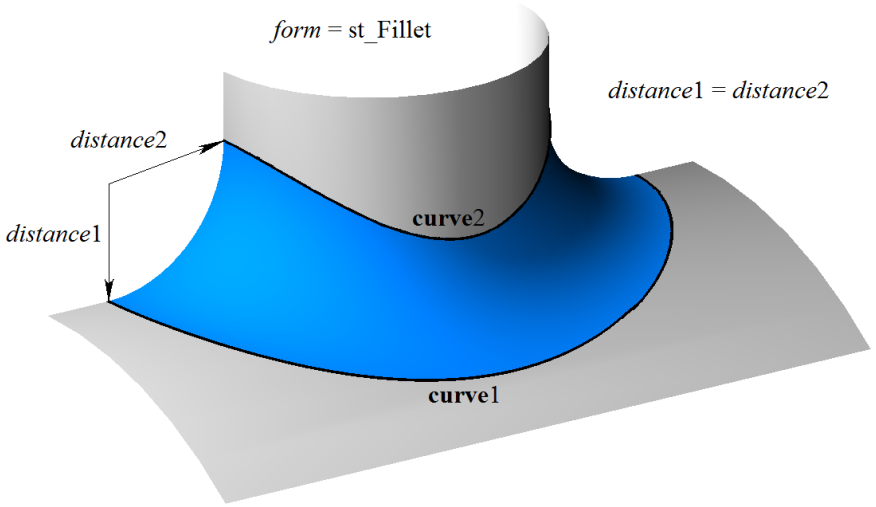


Рис. 1

В общем случае поверхность гладко сопрягается с поверхностями, на которых расположены кривые *curve1*(*u*) и *curve2*(*u*). В этом случае параметр *equable* = true. Если *equable* = false, то по одной из кривых *curve1* или *curve2* сопряжение является гладким, а другая кривая является кромкой. При *byCurve1* = true сопряжение является гладким по кривой *curve1*, в противном случае сопряжение является гладким по кривой *curve2*. Поверхность скругления с сохранением кромки приведена на рис. 2.

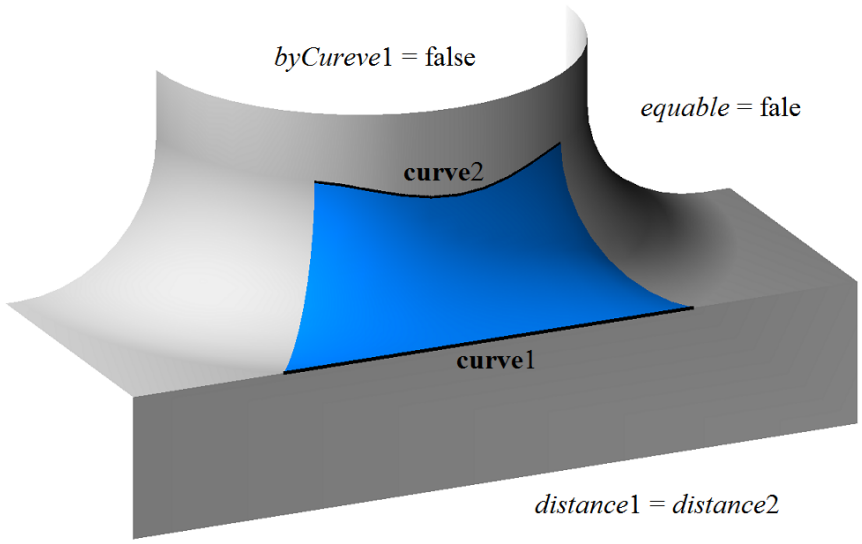


Рис. 2

Если радиусы скругления *distance1* и *distance2* не равны, то поверхность скругления в сечении **s**(*const*, *v*) с параметрами *u* = const описывает эллипс. Эллиптическая поверхность скругления приведена на рис. 3.

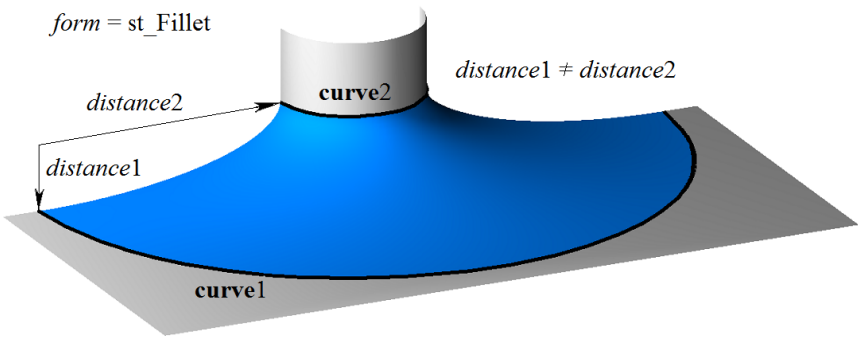


Рис. 3

В общем случае способ скругления *form* = st_Fillet. Если способ скругления *form* = st_Span, то *distance1* = *distance2* и равны расстоянию между кривыми *curve1* и *curve2*, а радиус поверхности скругления является переменным. Поверхность скругления с сохранением расстояния между опорными кривыми приведена на рис. 4.

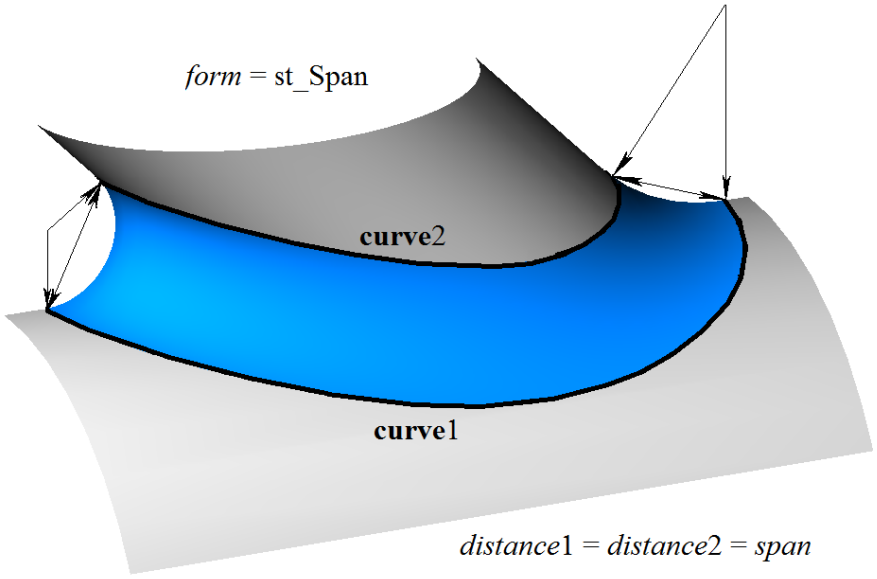


Рис. 4

Кривые поверхности **s**(*const*, *v*) с параметрами *u* = const являются коническими сечениями, форма которых зависит от параметра *conic*. Если *conic* = _ARC_ = 0, то кривые поверхности **s**(*const*, *v*) являются дугами окружности. Если *conic* = 0.5, то кривые поверхности **s**(*const*, *v*) являются дугами параболы. Если 0.05 < *conic* < 0.5, то кривые поверхности **s**(*const*, *v*) являются дугами эллипса. Если 0.5 < *conic* < 0.95, то кривые поверхности **s**(*const*, *v*) являются дугами гиперболы. Поверхность может иметь полюс при *u* = *umin* и *u* = *umax*, если соответствующие края кривых *curve0*, *curve1* и *curve2* совпадают.

3.13.26 Поверхность скругления MbChannelSurface

Класс MbChannelSurface объявлен в файле *surf_channel_surface.h*.

Поверхность скругления с переменным радиусом MbChannelSurface является наследником поверхности скругления MbFilletSurface. Поверхность скругления с переменным радиусом описывается кривой на первой сопрягаемой поверхности *MbFilletSurface* *curve1*, кривой на второй сопрягаемой поверхности *MbSurfaceCurve* *curve2*, кривой MbCurve3D *curve0*, функцией MbFunction *weights0* веса кривой *curve0*, функцией изменения радиуса MbFunction *function*, способом построения скругления *form*, радиусами скругления *distance1* и *distance2*, коэффициентом формы *conic*, граничными значениями *umin* и *umax* параметра кривых *curve1*, *curve2*, *curve0* и функций *weights0* и *function*, граничными значениями второго параметра поверхности *vmin* и *vmax*, признаком периодичности первого параметра поверхности *uclosed*, признаком наличия полюса поверхности при начальном значении первого параметра *poleMin*, признаком наличия полюса поверхности при конечном значении первого параметра *poleMax*. У поверхности есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов поверхности.

Кривые *curve1*, *curve2*, *curve0* и функции *weights0*, *function* согласованы друг с другом и имеют одинаковую область определения параметра. Первый параметр поверхности *u* совпадает с параметром кривых *curve1*, *curve2*, *curve0* и функций *weights0* и *function*, который является общим для них. Первый параметр поверхности принимает значения на отрезке *umin* ≤ *u* ≤ *umax*, который соответствует области определения кривых *curve1*, *curve2*, *curve0* и функций *weights0* и *function*. Поверхность может быть периодической по первому параметру, если периодическими являются кривые *curve1*, *curve2*, *curve0* и функции

weights0 и *function*.

Второй параметр поверхности *v* принимает значения на отрезке *vmin* ≤ *v* ≤ *vmax*. Значение *v* = *vmin* соответствует точке на кривой *curve1*, значение *v* = *vmax* соответствует точке на кривой *curve2*. Поверхность не может быть периодической по второму параметру.

В методе PointOn(double *u*, double *v*, MbCartPoint3D & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$s(u,v) = \frac{(1-v)^2\,\mathbf{curve1}(u) + 2v(1-v)\,\mathrm{weight0}(u)\,\mathbf{curve0}(u) + v^2\,\mathbf{curve2}(u)}{(1-v)^2 + 2v(1-v)\,\mathrm{weight0}(u) + v^2}.$$

Каждая кривая **s**(*const*, *v*) с фиксированным первым параметром поверхности *u* = const является NURBS-кривой третьего порядка, построенной по точкам *curve1*(*u*), *curve0*(*u*), *curve2*(*u*), у которой крайние точки имеют единичный вес, а средняя точка *curve0*(*u*) имеет вес *weights0*(*u*). При изменении первого параметра поверхности радиусы скругления меняются по законам R1(*u*) = *distance1* · *function*(*u*) и R2(*u*) = *distance2* · *function*(*u*). При *conic* = _ARC_ функция *weights0*(*u*) вычисляется из условия, что все кривые **s**(*const*, *v*) являются дугами окружности. При *conic* ≠ _ARC_ функция *weights0* является константой и равна коэффициенту формы *conic*. Поверхность скругления с переменным радиусом приведена на рис. 1.

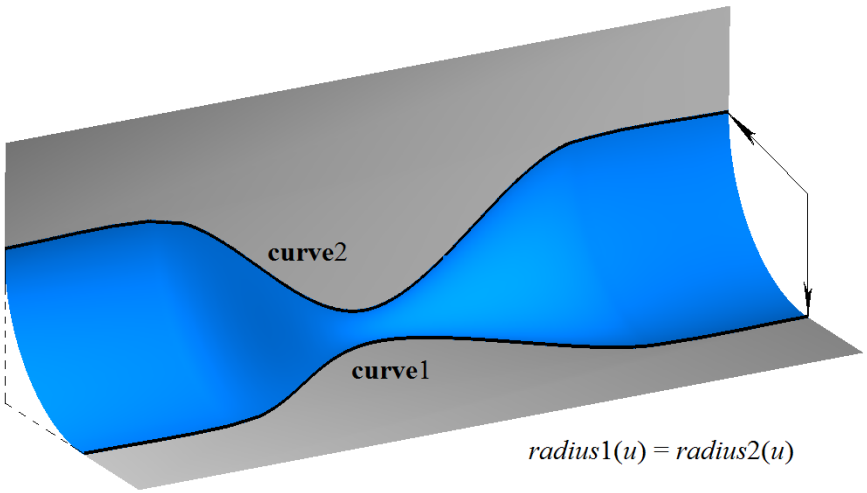


Рис. 1

Поверхность скругления с переменным радиусом гладко сопрягается с поверхностями, на которых расположены кривые *curve1*(*u*) и *curve2*(*u*), при этом параметр *equable* = true и *form* = st_Fillet.

Кривые поверхности **s**(*const*, *v*) с параметрами *u* = const являются коническими сечениями, форма которых зависит от параметра *conic*. Если *conic* = _ARC_ = 0, то кривые поверхности **s**(*const*, *v*) являются дугами окружности. Если *conic* = 0.5, то кривые поверхности **s**(*const*, *v*) являются дугами параболы. Если 0.05 < *conic* < 0.5, то кривые поверхности **s**(*const*, *v*) являются дугами эллипса. Если 0.5 < *conic* < 0.95, то кривые поверхности **s**(*const*, *v*) являются дугами гиперболы. Поверхность может иметь полюс при *u* = *umin* и *u* = *umax*, если соответствующие края кривых *curve0*, *curve1* и *curve2* совпадают.

3.13.27 Поверхность с произвольными границами MbCurveBoundedSurface

Класс MbCurveBoundedSurface объявлен в файле *surf_curve_bounded_surface.h*.

Поверхность MbCurveBoundedSurface описывается базовой поверхностью MbSurface **basisSurface**, множеством граничных кривых RPArray<*MbContourOnSurface*> **curves** (контуров на поверхности) и граничными значениями параметров *umin*, *umax*, *vmin*, *vmax*, определяющими габаритный прямоугольник области определения параметров.

Поверхность MbCurveBoundedSurface имеет криволинейные края и может иметь произвольные вырезы внутри. Границы поверхности описывают контуры на поверхности контейнера **curves**.

В методе PointOn(double *u*, double *v*, *MbCartPoint3D* & **s**) радиус-вектор поверхности **s** описывается векторной функцией

$$\mathbf{s}(u,v) = \mathbf{basisSurface}(u,v), \quad u,v \in \Omega,$$

где Ω — область определения параметров, которая представляет собой двумерную связную область. Радиус-вектор ограниченной контурами поверхности описывается тем же законом, что и поверхность **basisSurface**, но имеет другую область определения параметров. Базовая поверхность и контуры на ней приведены на рис. 1:

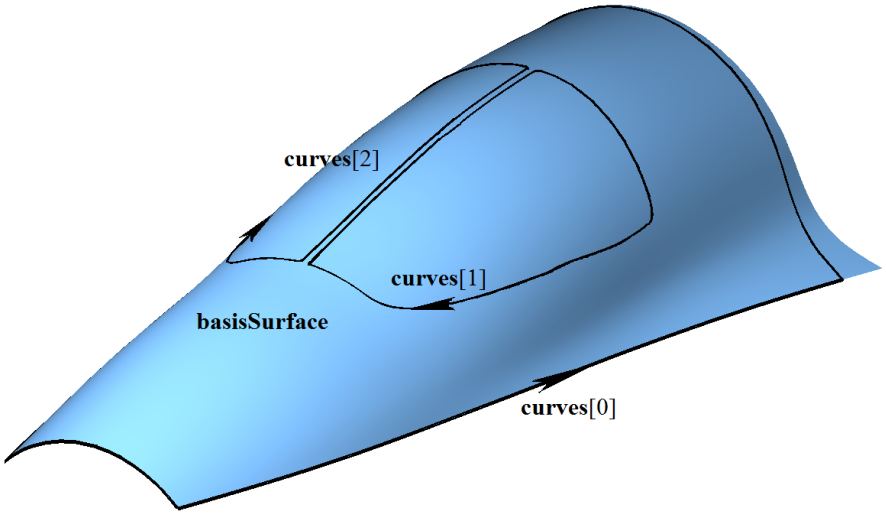


Рис. 1

В общем случае область определения параметров Ω ограниченной контурами поверхности может выходить за область определения параметров поверхности **basisSurface**. Вне области определения параметров поверхности **basisSurface** радиус-вектор **r**(*u*, *v*) вычисляется методом _PointOn(*u*, *v*, **s**) в соответствии с правилами продолжения поверхности **basisSurface**. Поверхность с произвольной границей приведена на рис. 2:

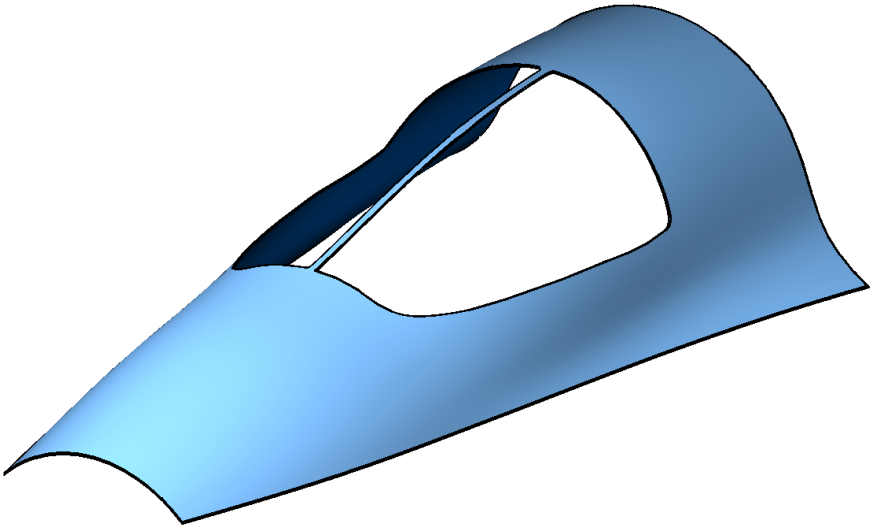


Рис. 2

Каждая кривая контейнера **curves** описывает одну замкнутую границу поверхности. Каждая кривая контейнера **curves** представляет собой контур на поверхности *MbContourOnSurface*. Каждый контур на поверхности описывается поверхностью **surface**, совпадающей с **basisSurface**, и двумерным контуром **contour**. Контур **contour** представляет собой составную замкнутую двумерную кривую. Сегментами контура **contour** могут быть любые двумерные кривые MbCurve за исключением составных кривых *MbContour*. В общем случае в местах стыковки сегментов производные контура терпят разрыв по длине и направлению. Двумерные контуры описывают границы области определения Ω поверхности MbCurveBoundedSurface. Граничные контуры контейнера **curves** удовлетворяют следующим условиям: они не пересекают сами себя и друг друга, первый контур **curves**[0] контейнера описывает внешнюю границу и содержит внутри все остальные контуры, которые описывают внутренние вырезы в поверхности. Внутренние контуры не могут быть вложены друг в друга. Для быстрого определения положения некоторой двумерной точки относительно области определения параметров поверхности граничные контуры ориентированы так, что при движении вдоль границы поверхность всегда находится слева, если смотреть навстречу нормали поверхности. Таким образом, внешний контур ориентирован так, что обход по нему осуществляется против движения часовой стрелки, если смотреть на поверхность навстречу её нормали, а внутренние контуры ориентированы в противоположном направлении.

Базовой поверхностью **basisSurface** не может быть поверхность с произвольными границами MbCurveBoundedSurface. Если нужно построить поверхность с произвольными границами на основе другой поверхности с произвольными границами, то в качестве базовой поверхности следует использовать базовую поверхность последней.

3.14 СПЕЦИАЛЬНЫЕ ОБЪЕКТЫ

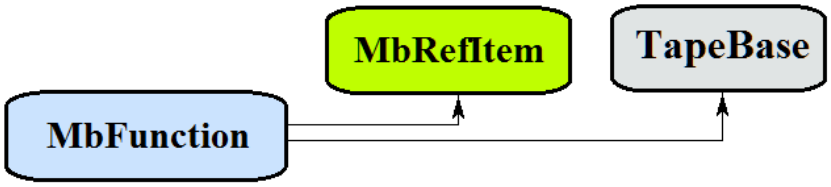
К специальным объектам относятся скалярные функции, представляющие собой аналоги кривых в одномерном пространстве. Специальные объекты используются для конкретно определенных целей, например, для описания изменения радиуса поверхности скругления в виде функции одного из параметров поверхности. В двумерном пространстве к специальным объектам относятся мультилиния и область. Для обслуживания мультилинии на базе двумерного контура создан контур с разрывами. В трёхмерном пространстве

специальные объекты описывают базовые точки других объектов, резьбу, выносные линии, шероховатости и прочие условные обозначения.

3.14.1 Функция MbFunction

Класс MbFunction объявлен в файле *function.h*.

Функция MbFunction является абстрактным классом, наследником классов *MbRefItem* и TapeBase:



В геометрическом ядре C3D реализованы следующие скалярные функции, которые являются наследниками класса MbFunction:

MbConctFunction – константная функция,

MbLineFunction – линейная функция,

MbCubicFunction – кубическая функция Эрмита,

MbCubicSplineFunction – кубическая сплайн-функция,

MdCharacterFunction – аналитическая функция.

MbFunction представляет собой скалярную функцию

$$function(t) = f(t)$$

скалярного параметра t, принимающего значения на отрезке [t_{min}, t_{max}]. Область изменения параметра функции есть отрезок [t_{min}, t_{max}] в одномерном пространстве. Функция $f(t)$ должна быть однозначной и непрерывной.

Граничные значения t_{min} и t_{max} области определения параметра выдают методы функции double GetTMin() и double GetTMax(), соответственно.

Функцию будем называть периодической, если существует $p > 0$, такое, что $f(t \pm kp) = f(t)$, где k – целое число. Метод IsClosed() периодической функции возвращает true. Метод double GetPeriod() периодической функции или функции, которая может быть расширена до периодической, выдает период p. Область определения параметра периодической функции всегда лежит в пределах одного периода.

Основным методом функции является метод

double Value(double & *t*).

Он выдаёт значение функции для заданного параметра *t*.

Методы

double FirstDer(double & *t*),

double SecondDer(double & *t*),

double ThirdDer(double & *t*)

выдают соответственно первую, вторую и третью производные функции для заданного параметра *t*. Перечисленные методы корректируют параметр функции при его выходе за пределы области определения. При выходе параметра *t* за пределы отрезка [t_{min}, t_{max}] непериодические функции смещают параметр *t* к ближайшей границе t_{min} или t_{max}, а периодические функции добавляют или вычитают необходимое количество периодов.

Метод

double _Value(double *t*)

выдаёт значение функции для заданного параметра *t* как в области определения параметра функции, так и за её пределами. В общем случае непериодическая функция за пределами области определения параметра продолжается по касательной в крайней точке. Исключение составляют аналитические функции. Периодические функции за пределами области определения параметра продолжаются циклически.

Методы

double _FirstDer(double *t*),

double _SecondDer(double *t*),

double _ThirdDer(double *t*)

выдают соответственно первую, вторую и третью производные функции для заданного параметра *t* как в области определения функции, так и за её пределами.

Функции перегружают такие методы, как

методы, обеспечивающие копирование, проверку на совпадение, проверку на возможность сделать совпадающими, делающие объекты совпадающими,

MbFunction & Duplicate(),

Boolean IsSame(const MbFunction & **item**),

Boolean IsSimilar(const MbFunction & **item**),

Boolean SetEqual(const MbFunction & **item**),

метод, возвращающий тип из перечисления функция,

MbeFunctionType IsA(),

методы, обеспечивающие выдачу и редактирование внутренних данных объекта,

MbProperty & CreateProperty(MbePrompt name),

void GetProperties(MbProperties & *properties*),

void SetProperties(MbProperties & *properties*).

Все функции, как правило, не имеют изломов. Функция MbAnaliticalFunction, задаваемая пользователем, также должна быть непрерывной и не иметь особых точек.

3.14.2 Константная функция MbConstFunction

Класс MbConctFunction объявлен в файле *func_const_function.h*.

Константная функция MbConctFunction описывается одним значением функции *value*.

В методе double Value(double & *t*) работает функция

$$f(t) = value.$$

Область определения параметра функции располагается в пределах 0 ≤ *t* ≤ 1. Функция не может быть периодической.

3.14.3 Линейная функция MbLineFunction

Класс MbLineFunction объявлен в файле *func_line_function.h*.

Линейная функция MbLineFunction описывается двумя граничными значениями функции *value1*, *value2* и граничными значениями параметра *tmin*, *tmax*.

В методе double Value(double & *t*) работает функция

$$f(t) = value1(1 - t) + value2t.$$

Область определения параметра функции располагается в пределах *tmin* ≤ *t* ≤ *tmax*. Функция не может быть периодической.

3.14.4 Кубическая функция Эрмита MbCubicFunction

Класс MbCubicFunction объявлен в файле *cur_cubic_function.h*.

Кубическая функция Эрмита MbCubicFunction описывается множеством контрольных точек *valueList*, множеством производных функции в контрольных точках *firctList*, множеством значений параметра функции в контрольных точках *tList* и признаком периодичности функции *closed*. У функции есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов функции.

Кубическая функция Эрмита при значении параметра *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, где *splinesCount* = *tList*.MaxIndex(), проходит через контрольную точку *valueList*[*i*] и имеет в ней производную *firctList*[*i*]. Функция построена на основе *splinesCount* сплайнов Эрмита третьей степени, которые гладко стыкуются между собой. Каждый кубический сплайн Эрмита описывает участок функции между двумя соседними контрольными точками. Каждый кубический сплайн Эрмита определяется двумя крайними точками и двумя производными кривой в этих точках.

При вычислении функции сначала по значению параметра *t* кривой определяется номер *i* рабочего участка (номер кубического сплайна Эрмита) из условия *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1]. Функция вычисляется для найденного рабочего участка по его локальному параметру *w*, который определяется по *tList*[*i*] и *tList*[*i* + 1].

В методе double Value(double & *t*) работает функция

$$f(t) = (1 - 3w^2 + 2w^3)\, valueList[i] + (3w^2 + 2w^3)\, valueList[i+1] + \\ + \big((w - 2w^2 + w^3)\, valueList[i] + (-w^2 + w^3)\, valueList[i+1]\big)\, (tList[i+1] - tList[i]),$$

где

$$w = \frac{t - tList[i]}{tList[i+1] - tList[i]}$$

— локальный параметр рабочего участка tList[i] ≤ t ≤ tList[i + 1].

Кубическая функция Эрмита приведена на рис. 1:

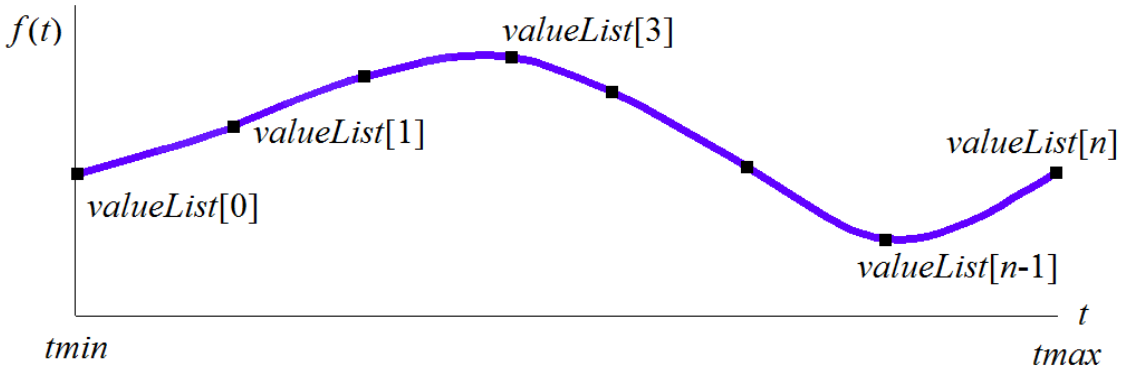


Рис. 1

Область определения параметра функции располагается в пределах *tmin* ≤ *t* ≤ *tmax*, где *tmin* = *tList*[0], *tmax* = *tList*[*splinesCount*]. Функция может быть периодической.

Форма функции зависит от расположения контрольных точек, от производных функции в контрольных точках и от множества *tList* значений параметра в контрольных точках. При построении функции только по контрольным точкам производные *firctList*[*i*] вычисляются путём построения параболы, проходящей по трём соседним точкам *valueList*[*i* – 1], *valueList*[*i*], *valueList*[*i* + 1] при соответствующих значениях параметра *tList*[*i* – 1], *tList*[*i*], *tList*[*i* + 1], и вычисления производной параболы в средней точке.

3.14.5 Кубическая сплайн-функция MbCubicSplineFunction

Класс MbCubicSplineFunction объявлен в файле *cur_cubic_spline_function.h*.

Кубическая сплайн-функция MbCubicSplineFunction описывается множеством контрольных точек *valueList*, множеством вторых производных функции в контрольных точках *secondList*, множеством значений параметра функции в контрольных точках *tList* и признаком периодичности функции *closed*. У функции есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов функции.

Кубическая сплайн-функция при значении параметра *tList*[*i*], *i* = 0, 1, ..., *splinesCount*, где *splinesCount* = *tList*.MaxIndex(), проходит через контрольную точку *valueList*[*i*] и имеет в ней вторую производную *secondList*[*i*].

При вычислении функции сначала по значению параметра *t* кривой определяется номер *i* рабочего участка из условия *tList*[*i*] ≤ *t* ≤ *tList*[*i* + 1]. Функция вычисляется для найденного рабочего участка по его локальному параметру *w*, который определяется по *tList*[*i*] и *tList*[*i* + 1].

В методе double Value(double & *t*) работает функция

$$r(t) = (1-w)\,\text{valueList}[i] + w\,\text{valueList}[i+1] + \left((-2w+3w^2-w^3)\,\text{valueList}[i] + (-w+w^3)\,\text{valueList}[i+1]\right)\frac{(\text{tList}[i+}$$

где

$$w = \frac{t - \text{tList}[i]}{\text{tList}[i+1] - \text{tList}[i]}$$

— локальный параметр рабочего участка tList[i] ≤ t ≤ tList[i + 1].

Кубическая сплайн-функция приведена на рис. 1:

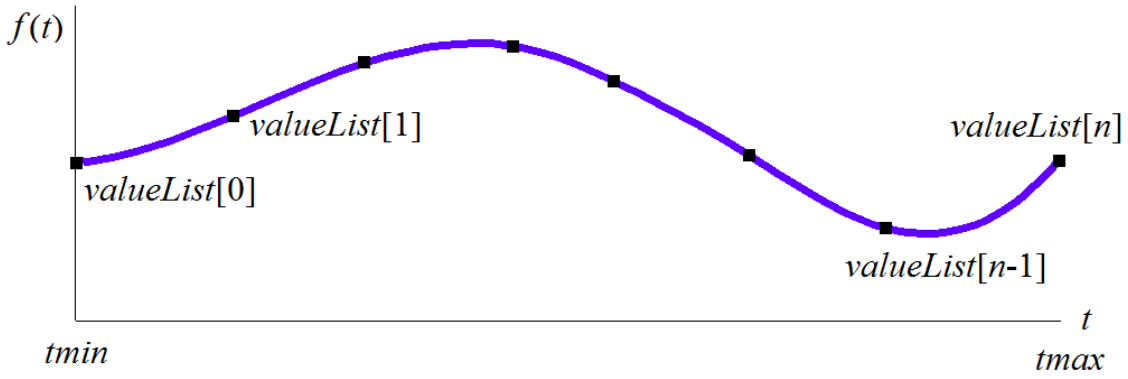


Рис. 1

Область определения параметра функции располагается в пределах *tmin* ≤ *t* ≤ *tmax*, где *tmin* = *tList*[0], *tmax* = *tList*[*splinesCount*]. Функция может быть периодической.

Форма функции зависит от расположения контрольных точек и от множества *tList* значений параметра в контрольных точках. При построении функции по контрольным точкам производные *secondList*[*i*] вычисляются из условия линейного изменения вторых производных между контрольными точками.

3.14.6 Символьная функция MdCharacterFunction

Класс MdCharacterFunction объявлен в файле *func_analytical_function.h*.

Символьная функция MdCharacterFunction описывается строковым выражением функции, деревом действий для вычисления выражения, граничными значениями параметра *tmin*, *tmax* и направлением *sense*. У функции есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов функции.

С помощью символьной функции можно описать любую функцию параметра *t* в виде строкового выражения, содержащего аналитические функции и арифметические операции.

В методе double Value(double & *t*) работает дерево действий для вычисления значения символьного выражения.

Область определения параметра функции располагается в пределах *tmin* ≤ *t* ≤ *tmax*. Функция может быть периодической.

3.14.7 Мультилиния MbMultiline

Класс MbMultiline объявлен в файле *multiline.h*.

Мультилиния MbMultiline является наследником класса *MbPlaneItem*, рис. 1:

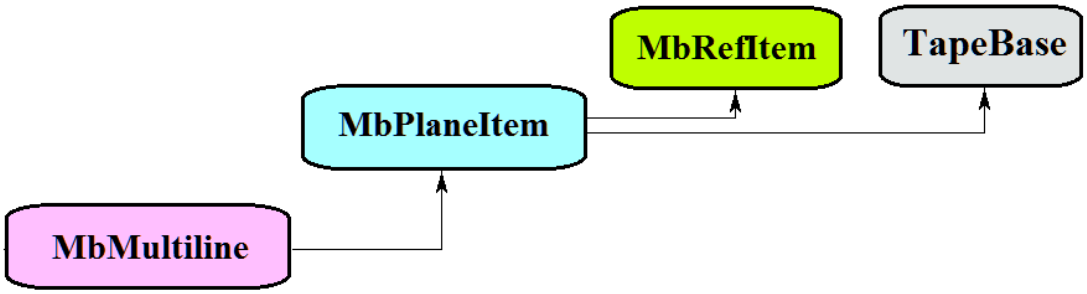


Рис. 1

Мультилиния описывается базовым контуром **basisCurve**, множеством вершин **vertices**, множеством радиусов *equidRadii*, параметрами начального края мультилинии *begTipParams*, параметрами конечного края мультилинии *endTipParams*, признаком обработки периодичности *processClosed*, признаком прозрачности *isTransparent*, множеством кривых **curves**, множеством законцовок в вершинах мультилинии **tipCurves**, законцовкой в начальной вершине мультилинии **begTipCurves**, законцовкой в конечной вершине мультилинии **endTipCurves**.

Мультилиния представляет собой контур, имеющий толщину. Толщина контура переменная. Края контура и места соединения сегментов контура могут иметь различную форму.

Мультилиния приведена на рис. 2:

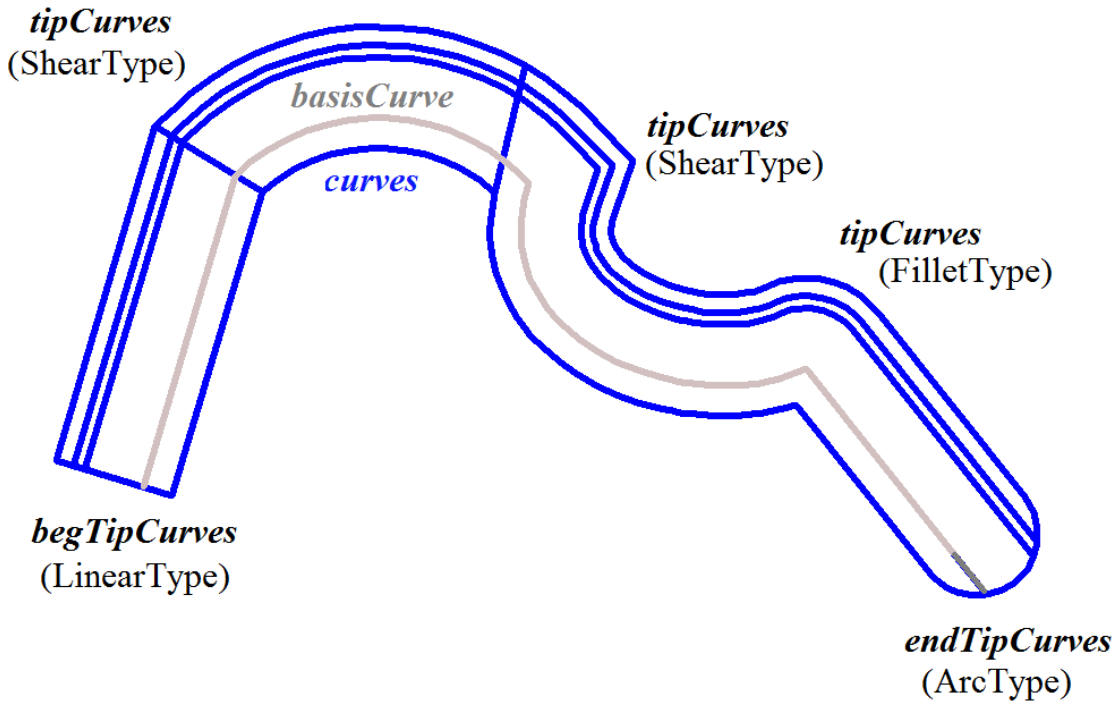


Рис. 2

Мультилиния используется для обмена данными с другими системами.

3.14.8 Двумерный контур с разрывами MbContourWithBreaks

Класс MbContourWithBreaks объявлен в файле *cur_contour_with_breaks.h*.

Двумерный контур с разрывами является наследником двумерного контура *MbContour*, рис. 1:

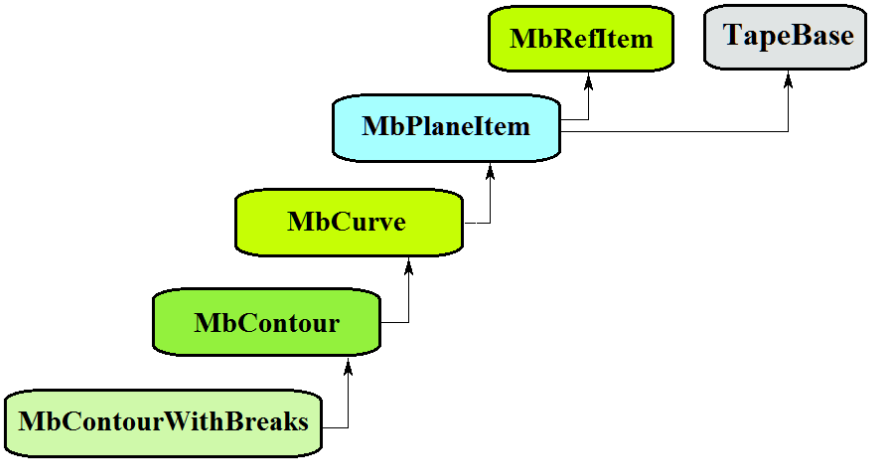


Рис. 1

Двумерный контур с разрывами описывается множеством **segments** стыкующихся друг с другом кривых, признаком периодичности кривой *closed*, разрывами **breaks**, множеством видимых участков контура **visibleContours** и множеством номеров сегментов контура *baseSegNumbers* для задания неподвижных точек разрыва.

Контур с разрывами приведён на рис. 2:

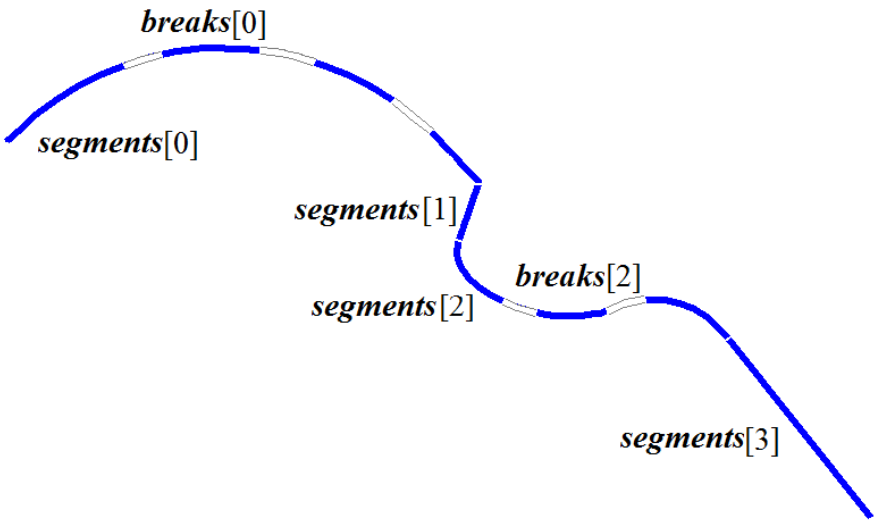


Рис. 2

В качестве сегментов двумерного контура с разрывами не должны использоваться другие двумерные контуры.

Двумерный контур с разрывами используется только для построения мультилиний.

3.14.9 Регион MbRegion

Класс MbRegion объявлен в файле *region.h*.

Регион MbRegion является наследником класса *MbPlaneItem*, рис. 1.

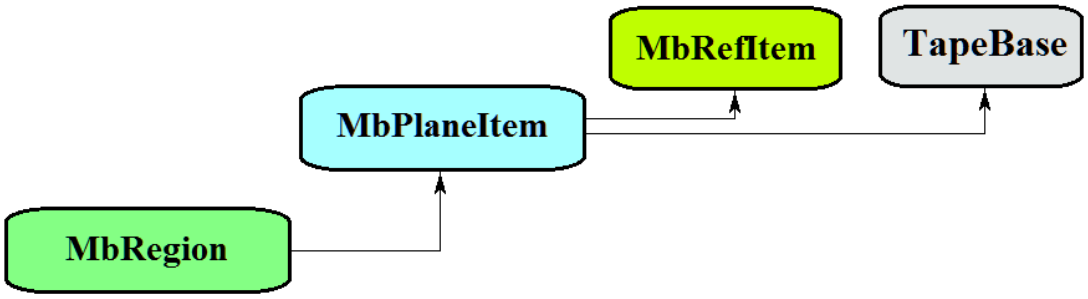


Рис. 1

Регион описывается множеством контуров **contours**. Контуры региона периодические и не пересекают сами себя и друг друга. Среди контуров множества один контур является внешним, а остальные являются внутренними и располагаются внутри внешнего контура. В множестве **contours** первым всегда лежит внешний контур.

Регион представляет собой связное множество точек двумерного пространства, границы которого описывают двумерные периодические контуры. Контуры региона ориентированы так, что при движении вдоль любого контура описываемое множество точек располагается слева от контура. То есть, внешний контур региона ориентирован против движения часовой стрелки, а внутренние контуры ориентированы по движению часовой стрелки.

Регион приведён на рис. 2:

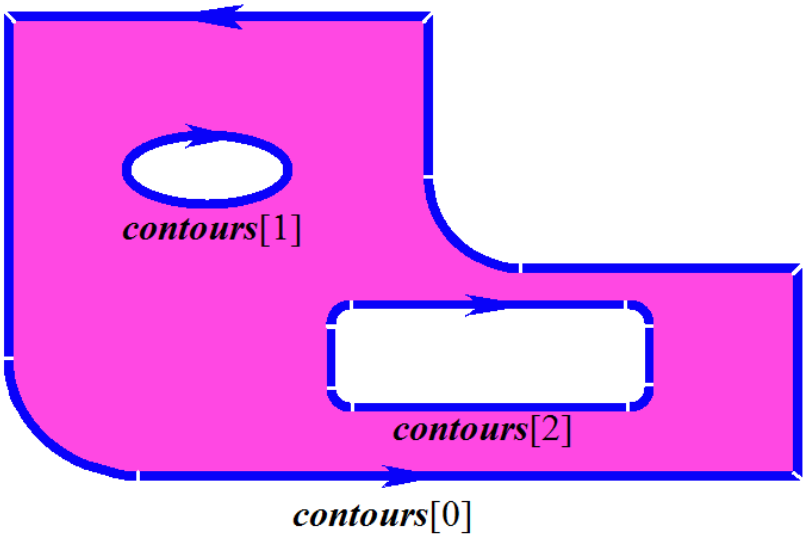


Рис. 2

Регион используется для описания двумерных связных областей. Над регионами можно выполнять булевы операции.

3.14.10 Вспомогательный геометрический объект MbLegend

Класс MbLegend объявлен в файле *legend.h*.

Вспомогательный объект MbLegend является наследником класса *MbSpaceItem*, рис. 1.

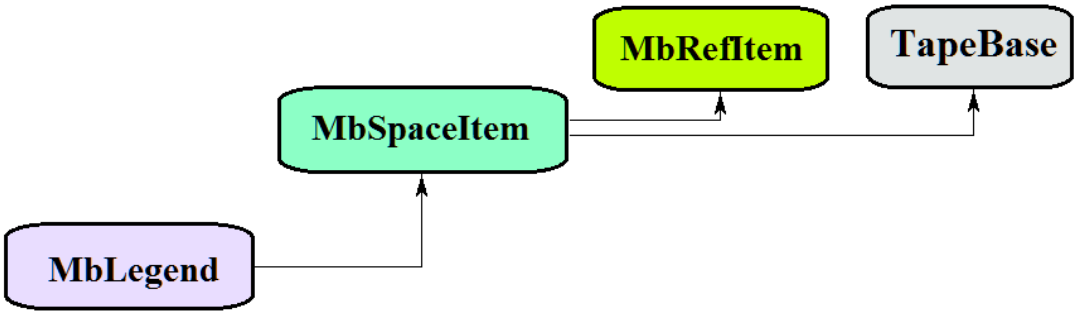


Рис. 1

Вспомогательный объект является абстрактным классом. В геометрическом ядре C3D реализованы следующие вспомогательные объекты, которые являются наследниками класса MbLegend:

MbMarker – точка и два ортонормированных вектора,

MbThread – резьба,

MbPointsSymbol – условное обозначение на базовых точках,

MbRough – обозначение шероховатости,

MbLeader – обозначение линии выноски.

Вспомогательные объекты используются для различных целей, но объединяет их взаимодействие с кривыми, поверхностями и объектами геометрической модели.

Вспомогательные объекты перегружают такие методы трёхмерного геометрического объекта, как:

методы, обслуживающие преобразование геометрического объекта,

Move(const MbVector3D & **v**, MbRegTransform * iReg = NULL),

Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),

Transform(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),

методы, обеспечивающие копирование, проверку на совпадение, проверку на возможность сделать совпадающими, делающие объекты совпадающими,

MbSpaceItem & Duplicate(MbRegDuplicate * iReg = NULL),

bool IsSame(const MbSpaceItem & **item**),

bool IsSimilar(const MbSpaceItem & **item**),

bool SetEqual(const MbSpaceItem & **item**),

методы, возвращающие тип из перечисления геометрических объектов,

MbeSpaceType IsA(),

MbeSpaceType Type(),

MbeSpaceType Family(),

методы, обеспечивающие выдачу и редактирование внутренних данных объекта,

MbProperty & CreateProperty(MbePrompt name),

GetProperties(MbProperties & *properties*),

SetProperties(MbProperties & *properties*),

метод, наполняющий полигональную копию геометрического объекта,

CalculateWire(double sag, MbMesh & **mesh**).

3.14.11 Маркер MbMarker

Класс MbMarker объявлен в файле *marker.h*.

Вспомогательный объект маркер MbMarker описывается точкой **origin** и двумя ортогональными векторами **axisZ**, **axisX**.

Маркер используется для установки геометрических ограничений на объекты трёхмерного пространства. Маркер является представителем геометрического объекта и может заменять трёхмерную точку, прямую, плоскость, локальную систему координат и другие объекты при работе геометрических ограничений.

3.14.12 Условное обозначение резьбы MbThread

Класс MbThread объявлен в файле *mb_thread.h*.

Условное обозначение резьбы MbThread описывается локальной системой координат резьбы place, начальным радиусом резьбы на поверхности *radObj*, начальным радиусом резьбы в теле *radThr*, длиной резьбы *length*, углом конусности резьбы *angle*, именем резьбы **name**, множеством тел **bodies**. У объекта есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов объекта.

Ось резьбового соединения направлена вдоль оси **place**.*axisZ*. Имя служит для идентификации условного обозначения резьбы среди плоских проекций граней тела из множества **bodies**. Условное обозначение резьбы приведено на рис. 1.

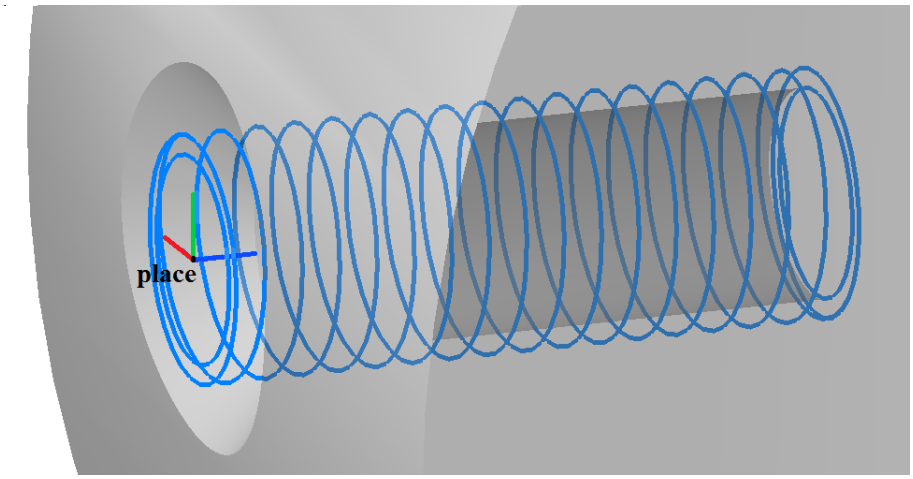


Рис. 1

Условное обозначение резьбы описывает элемент резьбового соединения геометрической модели и используется для построения плоских проекций резьбовых соединений.

3.14.13 Условное обозначение MbPointsSymbol

Класс MbPointsSymbol объявлен в файле *mb_symbol.h*.

Условное обозначение MbPointsSymbol является наследником класса MbSymbol. Условное обозначение MbPointsSymbol описывается набором идентификаторов, множеством трехмерных точек **points** и данными об участках сложных разрезов *steps*.

Объект несет информацию о базовых точках условных обозначений, связанных с элементами геометрической модели.

Объект используется при построении плоских проекций условных обозначений, для которых достаточно знать положение базовых точек. Он определяет базовые точки условных обозначений элементов геометрической модели.

3.14.14 Обозначение шероховатости MbRough

Класс MbRough объявлен в файле *mb_rough.h*.

Условное обозначение шероховатости MbRough является наследником класса MbPointsSymbol. Условное обозначение шероховатости описывается множеством трехмерных точек **points**, данными об участках сложных разрезов *steps* и топологическим объектом привязки **item**.

Объект несет информацию о базовых точках условного обозначения шероховатости, связанных с топологическим объектом **item**.

Объект используется при построении плоских проекций условного обозначения шероховатости элемента геометрической модели **item**.

3.14.15 Условное обозначение линии выноски MbLeader

Класс MbLeader объявлен в файле **mb_rough.h**.

Условное обозначение линии выноски MbLeader является наследником класса MbSymbol. Условное обозначение линии выноски описывается набором идентификаторов и множеством обозначений шероховатости **branches**.

Объект несет информацию о линии выноски обозначения шероховатости для топологических объектов.

Объект используется при построении плоских проекций условного обозначения линии выноски шероховатости для элементов геометрической модели.

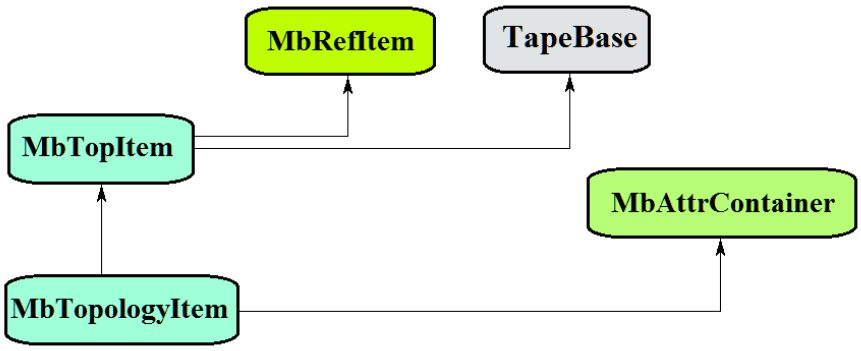
3.15 ТОПОЛОГИЧЕСКИЕ ОБЪЕКТЫ

Топологическими называют геометрические свойства, которые не зависят от количественных характеристик (длин и углов), а отражают непрерывную связь элементов объекта и его окружения. Топологические объекты геометрического ядра C3D описывают и геометрические свойства объекта, зависящие от количественных характеристик, и геометрические свойства, отражающие непрерывную связь объекта с соседними элементами. Топологические объекты строятся на основе поверхностей, кривых и точек путём добавления к их данным, свойствам и методам новых данных, свойств и методов, отражающих связь объекта с его окружением.

3.15.1 Топологический объект MbTopologyItem

Класс MbTopologyItem объявлен в файле *topology_item.h*.

Топологический объект MbTopologyItem отличается от других топологических объектов наличием имени **name**, признака изменения *changed* и метки *label*. Топологический объект MbTopologyItem является наследником топологического объекта *MbTopItem*:



Контейнер атрибутов MbAttrContainer наделяет именованные топологические объекты атрибутами.

В геометрическом ядре C3D реализованы следующие топологические объекты, которые являются наследниками класса MbTopologyItem:

MbFace – грань,

MbEdge – ребро,

MbVertex – вершина.

Ребро MbEdge имеет наследника *MbCurveEdge* – ребро стыковки граней.

Именованные топологические объекты имеют такие методы как:

методы, обслуживающие преобразование топологического объекта:

void Move(const MbVector3D & **v**, MbRegTransform * iReg = NULL),

void Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),

void Transform(const MbMatrix3D & **m**, MbRegTransform * iReg = NULL),

методы, обслуживающие работу с именем топологического объекта:

MbName & GetName(),

SimpleName & GetMainName(),

SimpleName & GetFirstName(),

метод, возвращающий тип из перечисления топологических объектов:

MbeTopologyType IsA().

Именованные топологические объекты используются в качестве элементов для построения объектов геометрической модели.

3.15.2 Грань MbFace

Класс MbFace объявлен в файле *topology.h*.

Грань MbFace является наследником топологического объекта *MbTopologyItem* и представляет собой конечный кусок поверхности, которому приписано направление нормали и определены границы. Сторону поверхности и грани, при взгляде на которую мы смотрим навстречу нормали, будем называть внешней, другую сторону будем называть внутренней. Стороны поверхности *MbSurface* не обладают равноправием относительно нормали, так как у поверхности одна сторона всегда внешняя, а другая сторона всегда внутренняя. В отличие от поверхности для грани мы имеем возможность назначить направление нормали и тем самым назначить внешнюю и внутреннюю стороны.

Структура данных грани содержит указатель на поверхность *MbSurface** **surface**, признак совпадения направления нормали грани с направлением нормали поверхности **sameSense** и совокупность циклов грани RPArray<*MbLoop*> **loops**. У грани есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов грани.

Указатель на поверхность не может быть нулём. Признак совпадения направления нормали грани с направлением нормали поверхности принимает значение true, если нормали грани и поверхности совпадают, в противном случае признак принимает значение false. Сторону грани, при взгляде на которую мы смотрим навстречу нормали, называют внешней стороной, а другую сторону грани называют внутренней стороной.

Циклы грани описывают границы грани. Каждая граница грани замкнута. Каждый цикл описывается последовательностью рёбер *MbOrientedEdge* в порядке их следования вдоль границы. Количество циклов грани равно количеству границ поверхности грани. Одна из границ грани является внешней и содержит границы внутренних вырезов. Цикл,

располагающийся первым по счёту в контейнере циклов, описывает внешнюю границу грани и содержит внутри себя внутренние циклы, которые описывают внутренние границы грани. Внешний цикл грани ориентирован против часовой стрелки, а внутренние циклы ориентированы по часовой стрелке, если смотреть навстречу нормали грани. Таким образом, при движении вдоль цикла грани по её внешней стороне грань всегда располагается слева. На рис. 1 стрелки указывают направления циклов грани и её нормали.

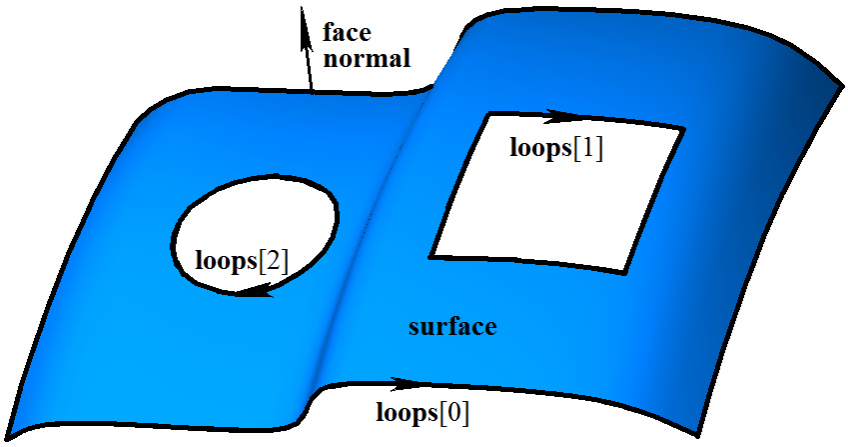


Рис. 1

Циклы одной грани не должны пересекать друг друга и сами себя.

Грань может быть именована и может иметь атрибуты. Имя может использоваться для идентификации грани, атрибуты могут нести дополнительную информацию о грани, например, цвет, прозрачность, происхождение и так далее.

Грань используется для твердотельного и гибридного моделирования.

3.15.3 Ребро MbEdge

Класс MbEdge объявлен в файле *topology.h*.

Ребро MbEdge является наследником топологического объекта MbTopologyItem и представляет собой кривую, которой приписано направление. Направление кривой MbCurve3D жёстко связано с направлением возрастания её параметра. В отличие от кривой ребро может быть направлено как в сторону возрастания параметра, так и в сторону уменьшения параметра кривой. Ребро всегда начинается и оканчивается в некоторой вершине *MbVertex*.

Структура данных ребра содержит указатель на кривую MbCurve3D **curve**, признак совпадения направления ребра с направлением кривой *sameSense*, указатель на начальную вершину MbVertex **begVertex** и указатель на конечную вершину MbVertex **endVertex**. На рис. 1 приведено ребро.

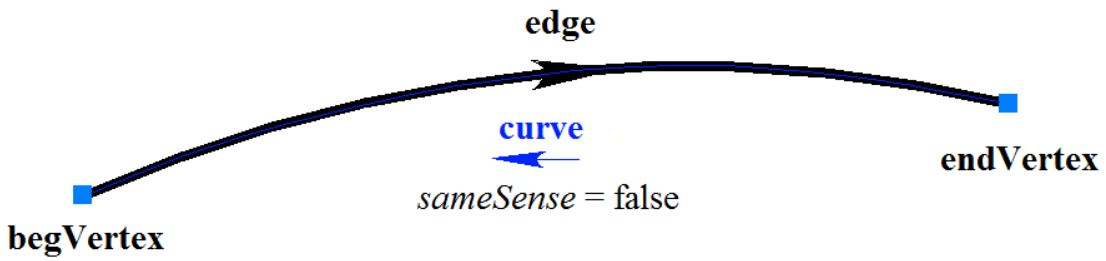


Рис. 1

Указатели на кривую и вершины не могут быть нулём. Признак совпадения направления ребра с направлением кривой принимает значение true, если ребро совпадает с кривой, и принимает значение false, если ребро направлено против кривой. Если ребро начинается и заканчивается в одной и той же вершине, то такое ребро является замкнутым. Указатели на начальную и конечную вершину замкнутого ребра равны.

Ребро может быть именовано и может иметь атрибуты. Имя может использоваться для идентификации ребра, атрибуты могут нести дополнительную информацию о ребре, например, цвет, стиль отображения, происхождение и так далее.

Ребро используется для каркасного моделирования.

3.15.4 Вершина MbVertex

Класс MbVertex объявлен в файле *topology.h*.

Вершина MbVertex является наследником топологического объекта MbTopologyItem и представляет собой точку с известной погрешностью расположения. Структура данных вершины содержит точку *MbCartPoint* **point** и погрешность *tolerance* расположения этой точки.

Вершина может описывать отдельную точку точечного каркаса или стык рёбер. В вершине может стыковаться любое конечное число рёбер. Стыкующиеся рёбра указывают на одну и ту же общую для них вершину. На рис. 1 приведена вершина, в которой стыкуются три ребра:

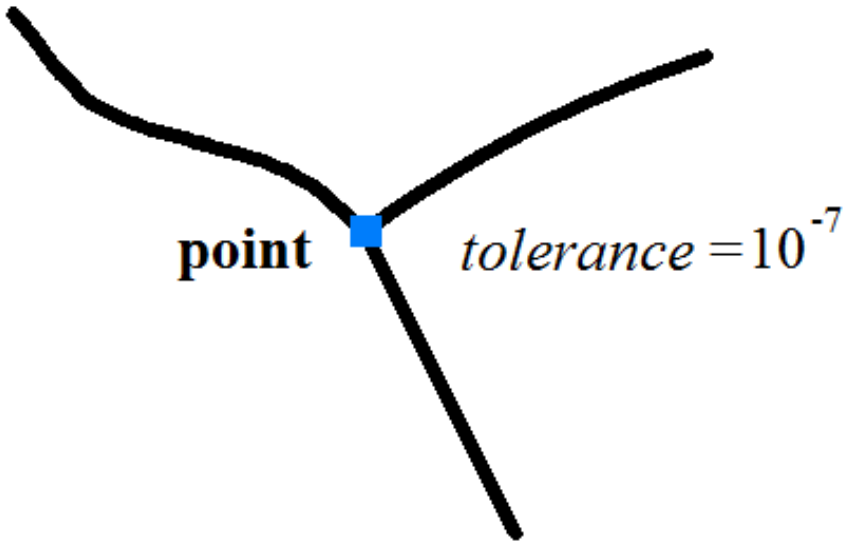


Рис. 1

При неточной стыковке рёбер погрешность расположения вершины равна расстоянию от точки вершины до края наиболее удалённого ребра. На рис. 2 приведена толерантная вершина, в которой стыкуются четыре ребра.

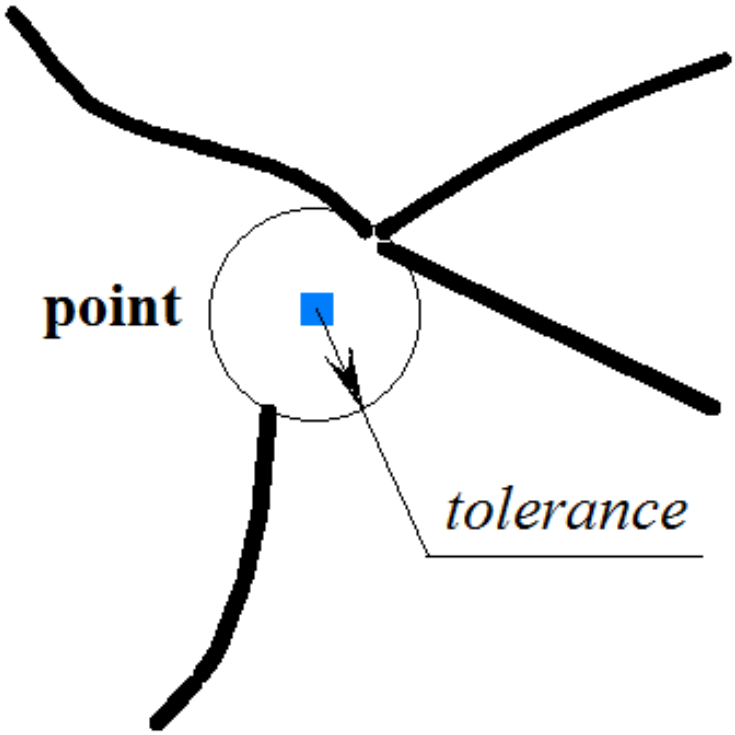


Рис. 2

Вершина может быть именована и может иметь атрибуты. Имя может использоваться для идентификации вершины, атрибуты могут нести дополнительную информацию о вершине, например, цвет, стиль отображения, происхождение и так далее. Вершина используется для всех методов моделирования.

3.15.5 Ребро грани MbCurveEdge

Класс MbCurveEdge объявлен в файле *topology.h*.

Ребро MbCurveEdge является наследником ребра MbEdge и представляет собой ребро, построенное на кривой пересечения поверхностей *MbSurfaceIntersectionCurve*. Ребро MbCurveEdge предназначено для описания участка границы грани. В отличие от ребра MbEdge ребро MbCurveEdge описывает не просто кривую, а участок стыковки двух граней или участок края грани.

Структура данных ребра грани содержит указатель на кривую пересечения поверхностей *MbSurfaceIntersectionCurve* **curve**, признак совпадения направления ребра с направлением кривой *sameSense*, указатель на начальную вершину MbVertex **begVertex**, указатель на конечную вершину MbVertex **endVertex**, указатель на грань слева MbFace **facePlus** и указатель на грань справа MbFace **faceMinus** от ребра. На рис. 1 приведено ребро, соединяющее две грани.

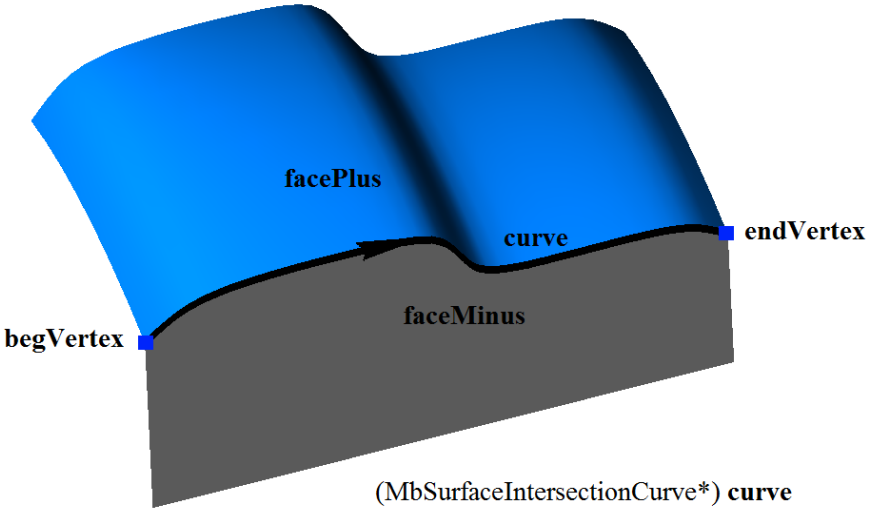


Рис. 1

Ребро грани может описывать участок стыковки двух различных граней. В этом случае указатели на грань слева и грань справа от ребра не равны нулю и не равны друг другу. Поверхности, лежащие в структурах данных соединяемых ребром граней, совпадают с поверхностями, лежащими в структуре данных кривой ребра, а именно:

facePlus→**surface**→GetSurface() == **curve**→curveOne.suface и

faceMinus→**surface**→GetSurface() == **curve**→curveTwo.suface

или

facePlus→**surface**→GetSurface() == **curve**→curveTwo.suface и

faceMinus→**surface**→GetSurface() == **curve**→curveOne.suface.

У циклически замкнутой по одному или обоим параметрам поверхности грани присутствуют участки границы, по которым грань стыкуется сама с собой. Такое ребро является швом. В этом случае указатели на грань слева и грань справа от ребра равны друг другу, рис. 2:

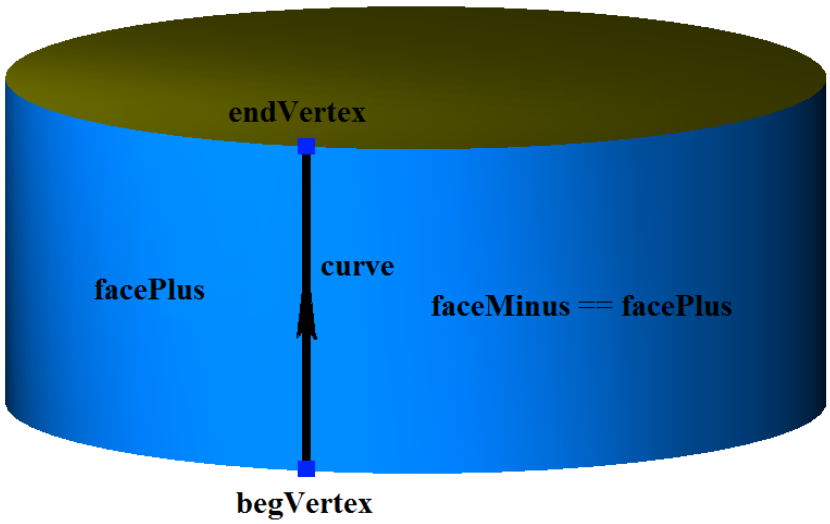


Рис. 2

Ребро грани может описывать участок края грани. Такое ребро является краевым. В этом случае указатель на грань слева или грань справа от ребра равен нулю, рис. 3.

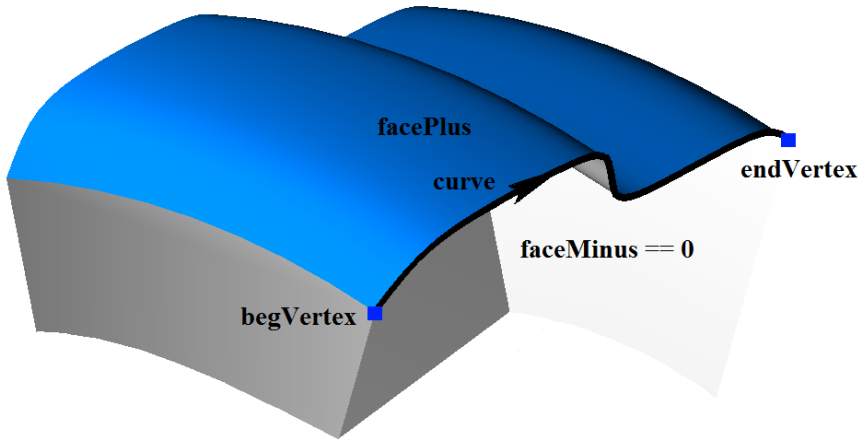


Рис. 3

Ребро, имеющее нулевую длину, является полюсным и описывает полюс грани. Полюсное ребро не является краевым, так как грань в полюсном ребре не имеет края. Как правило, полюсное ребро располагается в особых точках поверхности грани. В области параметров поверхности грани полюсному участку соответствует некоторая кривая. Указатели на начальную и конечную вершину полюсного ребра равны. Полюсное ребро приведено на рис. 4:

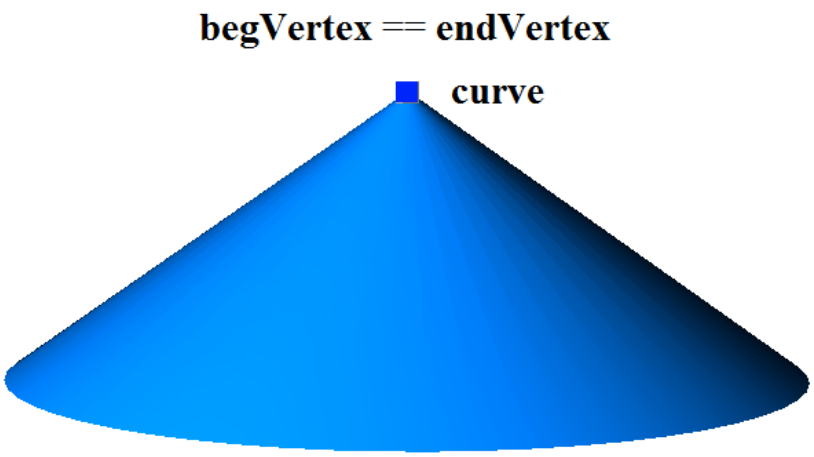


Рис. 4

У полюсного ребра указатель на грань слева или грань справа от ребра равен нулю. Кривая пересечения полюсного ребра имеет следующие данные:

curve→curveOne.suface == **curve**→curveTwo.suface, а отрезок

curve→curveOne.curve является дублем отрезка **curve**→curveTwo.curve.

Ребро грани может быть именовано и может иметь атрибуты. Имя может использоваться для идентификации ребра, атрибуты могут нести дополнительную информацию о ребре грани, например, цвет, стиль, происхождение и так далее.

Ребро грани используется для твердотельного и гибридного моделирования.

3.15.6 Цикл грани MbLoop

Класс MbLoop объявлен в файле *topology.h*.

Цикл грани MbLoop является наследником топологического объекта *MbTopItem* и описывает последовательность рёбер, исчерпывающую некоторую границу грани.

Структура данных цикла содержит множество ориентированных рёбер RPArray<*MbOrientedEdge*> **edgeList** в порядке их следования вдоль границы грани. У цикла есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов цикла.

Направление ориентированных рёбер совпадает с направлением цикла. Конец каждого ориентированного ребра цикла стыкуется с началом следующего за ним ориентированного ребра цикла, рис. 1:

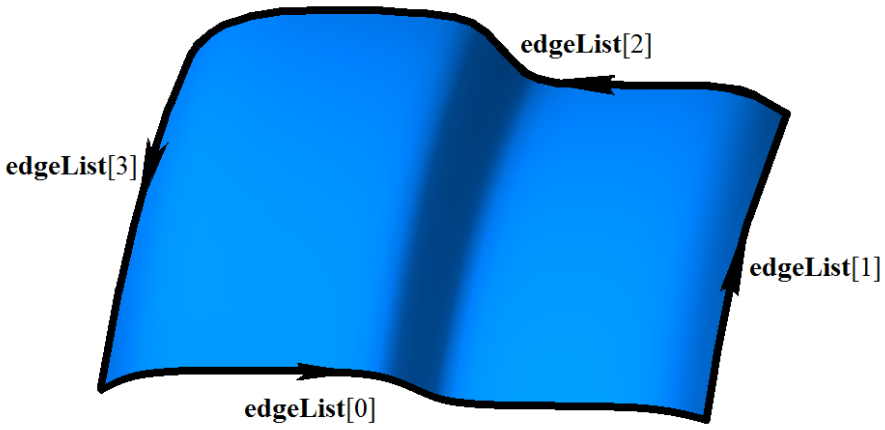


Рис. 1

Конец последнего ориентированного ребра цикла стыкуется с началом первого ориентированного ребра цикла.

На рис. 2 приведён цикл сферической грани, состоящий из четырёх ориентированных рёбер, два из которых построены на ребре шва и два из которых являются полюсными рёбрами:

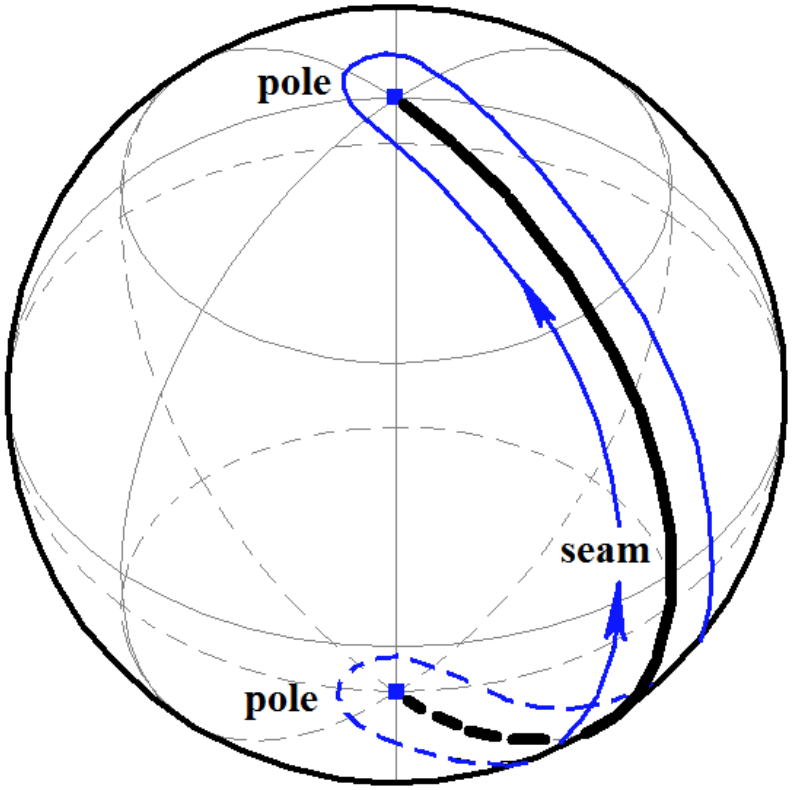


Рис. 2

Если цикл сферической грани нарисовать в пространстве параметров сферы, то он будет представлять собой прямоугольник. Цикл, так же как и граница грани, всегда замкнут. Цикл направлен так, чтобы грань всегда для нас находилась слева, если мы движемся вдоль цикла с внешней стороны грани.

3.15.7 Ориентированное ребро грани MbOrientedEdge

Класс MbOrientedEdge объявлен в файле *topology.h*.

Ориентированное ребро MbOrientedEdge является наследником топологического объекта *MbTopItem* и описывает участок границы грани. Структура данных ориентированного ребра содержит ребро грани MbCurveEdge *curveEdge* и признак совпадения направления ребра грани с направлением ориентированного ребра и, соответственно, с направлением цикла грани *orientation*.

Если направление ребра грани *curveEdge* совпадает с направлением цикла, то в цикле этой грани соответствующее ориентированное ребро имеет признак *orientation* = true. Если направление ребра грани *curveEdge* не совпадает с направлением цикла, то в цикле этой грани соответствующее ориентированное ребро имеет признак *orientation* = false. На рис. 1 приведены два ориентированных ребра, построенных на одном и том же ребре грани *curveEdge*.

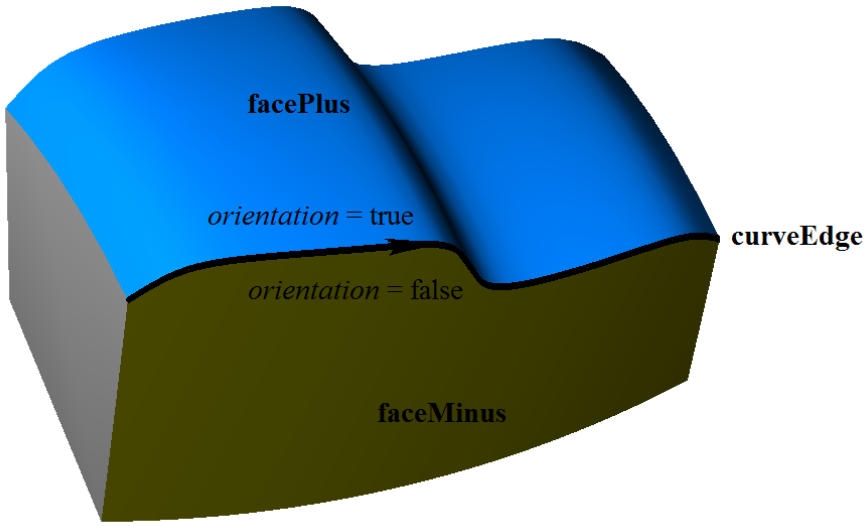


Рис. 1

В общем случае ребро грани MbCurveEdge входит в два цикла, эти циклы принадлежат соединяемым ребром граням. В один цикл ребро грани входит с признаком ориентации *orientation* = *true*, эта грань находится слева от ребра и поле данных **facePlus** указывает на неё. В другой цикл ребро грани входит с признаком ориентации *orientation* = *false*, эта грань находится справа от ребра и поле данных **faceMinus** указывает на неё. Таким образом смежные грани и соединяющие их рёбра связаны друг с другом.

3.15.8 Множество граней MbFaceShell

Класс MbFaceShell объявлен в файле *topology_faceset.h*.

Множество граней MbFaceShell является наследником топологического объекта *MbTopItem*. Структура данных MbFaceShell содержит множество граней RPArray<*MbFace*> **faceSet** и признак замкнутости *closed* множества граней. У множества граней есть ещё некоторые данные, которые не обязательны и служат для ускорения работы методов множества граней.

Множество граней, как правило, описывает связный кусок поверхности моделируемого объекта. Связанные между собой грани множества удовлетворяют определённым условиям, а именно: грани стыкуются между собой по общим рёбрам, в каждом ребре стыкуются только две грани, причём грани стыкуются так, что внешняя сторона одной грани переходит во внешнюю сторону другой грани, и образуемая стыкующимися гранями общая поверхность не должна иметь самопересечений.

На рис. 1 приведено множество связанных между собой граней. Все рёбра граней входят в два цикла, у всех рёбер указатели **facePlus** и **faceMinus** не равны нулю, а кривые *MbSurfaceIntersectionCurve*, на которых построены рёбра граней, имеют **curveOne.surface** != **curveTwo.surface**.

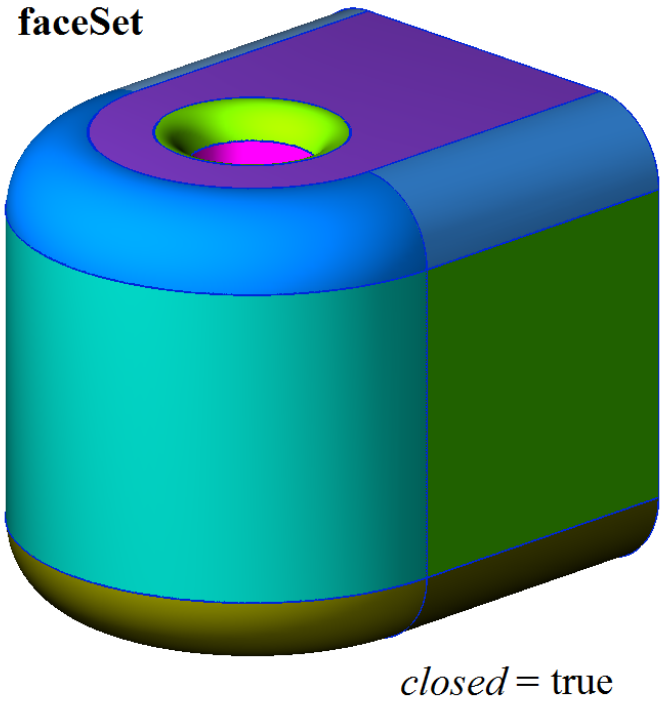


Рис. 1

Грани, приведённые на рисунке, образуют общую замкнутую поверхность, внешняя сторона каждой грани переходит во внешнюю сторону соседней грани. У множества граней, не имеющего ни одного краевого ребра, признак *closed* принимает значение true.

Если хотя бы одна грань множества имеет хотя бы одно краевое ребро, то признак *closed* множества граней принимает значение false. На рис. 2 приведено множество граней, некоторые из которых имеют краевые рёбра. Краевые рёбра входят только в один

цикл и у краевых рёбер грани только один из указателей **facePlus** или **faceMinus** не равен нулю, а кривые *MbSurfaceIntersectionCurve*, на которых построены рёбра, имеют **curveOne.surface** == **curveTwo.surface** и параметр *buildType = cbt_Boundary*. Полюсное ребро не является краевым, так как не образует край.

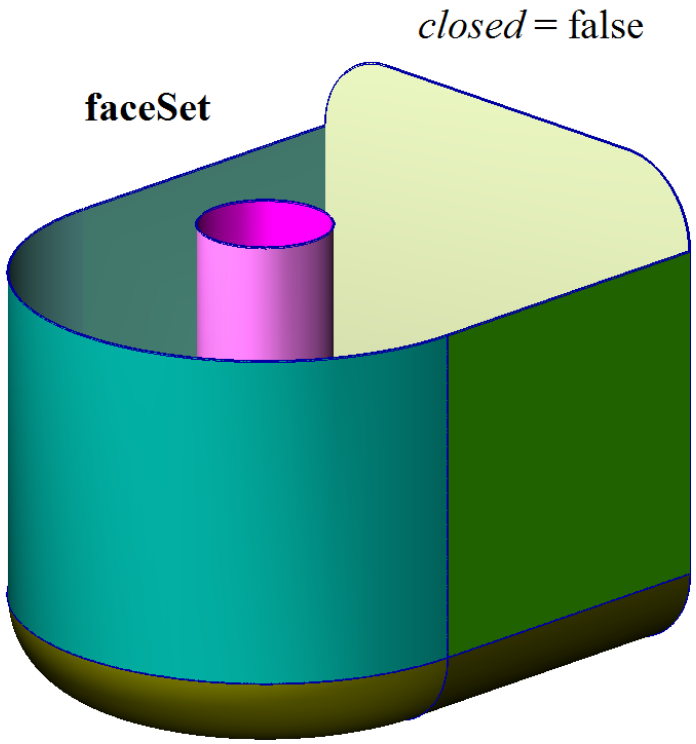


Рис. 2

Связное множество граней называют *замкнутой оболочкой*, если грани множества не имеют края. Если стыкующиеся друг с другом грани имеют хотя бы одно краевое ребро, то такое связное множество граней называют *незамкнутой оболочкой*. Подчеркнём, что замкнутая оболочка и незамкнутая оболочка представляют собой связное множество стыкующихся друг с другом граней.

Множество граней может состоять из нескольких не связанных между собой частей. На рис. 3 приведено множество граней, описывающее две незамкнутые оболочки.

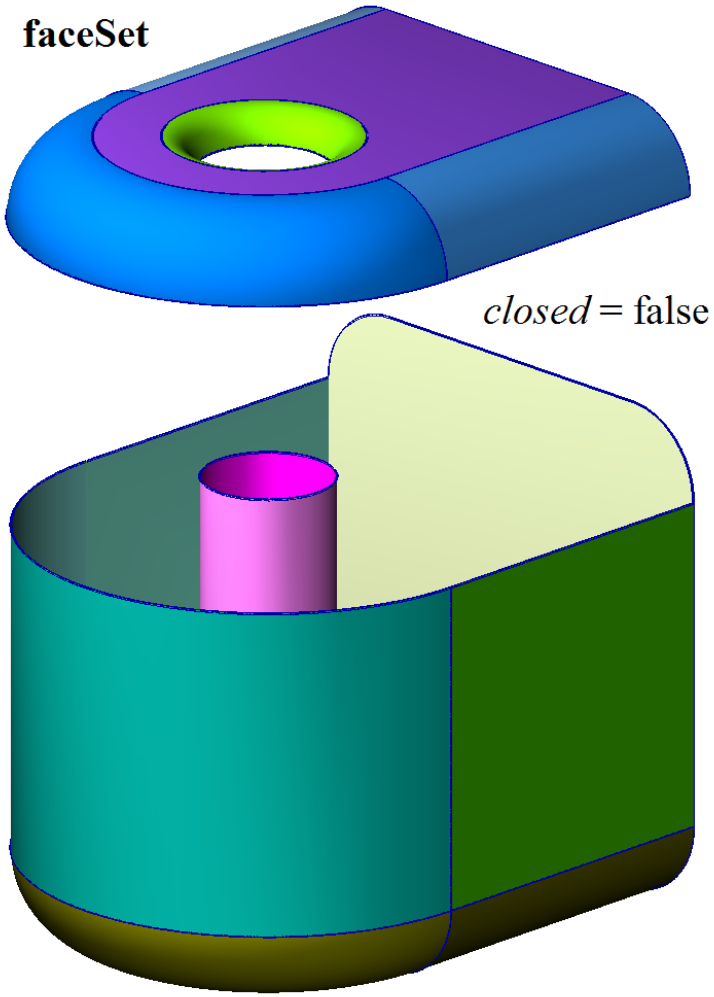


Рис. 3

Формально на грани множества не наложено никаких ограничений. Множество граней может состоять из отдельных граней. На рис. 4 приведено множество не связанных между собой граней. Каждая грань, приведённая на рис., образует отдельную оболочку, все рёбра граней являются краевыми.

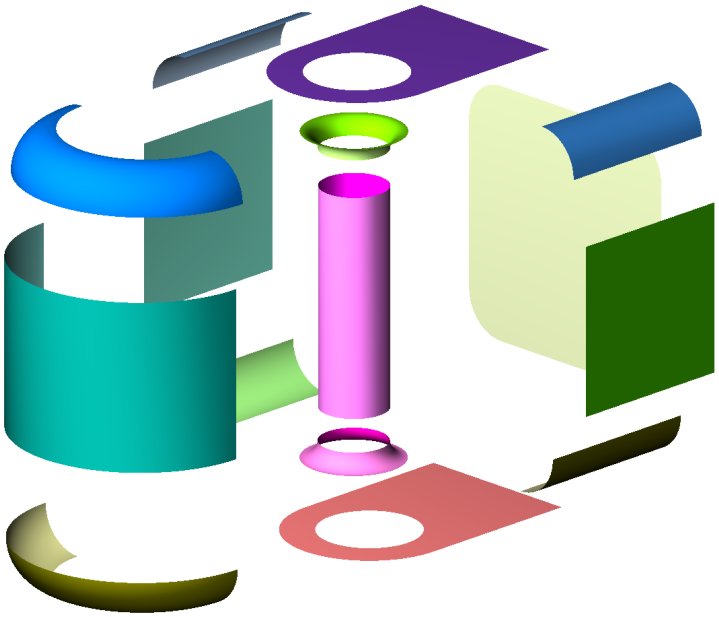


Рис. 4

Основными методами оболочки являются методы преобразования её в пространстве:

void Move(const *MbVector3D* & **v**, MbRegTransform * iReg = NULL),

void Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),

void Transform(const *MbMatrix3D* & **m**, MbRegTransform * iReg = NULL),

методы поиска граней, рёбер и вершин, а также методы определения положения точки относительно замкнутой оболочки:

bool DistanceToBound(const *MbCartPoint3D* &, ...),

bool PointClassification(const *MbCartPoint3D* &, ...).

Незамкнутая оболочка описывает только часть граничной поверхности моделируемого объекта. Незамкнутые оболочки используются для поверхностного моделирования. Если к замкнутой оболочке добавить множество точек, лежащих внутри неё, то мы получим твердое тело. Замкнутые оболочки используются для твердотельного моделирования. Обрезка и стыковка граней выполняется одновременно с построением оболочки. Это обеспечивают методы геометрического ядра C3D. На базе замкнутых и незамкнутых оболочек строятся объекты геометрической модели.

3.15.9 Копирование множества граней MbFaceShell

Каждый метод построения, который в составе входящих параметров использует множество граней, представленных в виде *MbFaceShell* или *MbSolid*, модифицирует некоторые вершины, рёбра и грани исходных объектов. Для ускорения построений и сохранения неизменным исходного множества граней применяется полное или частичное копирование данных объекта MbFaceShell. В геометрическом ядре C3D используются четыре способа копирования множества граней MbFaceShell, определяемые перечислением MbeCopyMode. Как правило, в методы построения вместе с модифицируемым множеством граней передаётся параметр типа MbeCopyMode, который управляет передачей граней, рёбер и вершин от исходного объекта построенному объекту.

Перечисление MbeCopyMode объявлено в файле *mb_enum.h*. Параметр типа MbeCopyMode может принимать одно из четырёх значений: *cm_Copy*, *cm_KeepSurface*, *cm_KeepHistory*, *cm_Same*.

При значении *cm_Copy* исходное множество граней модифицируемого объекта полностью копируется, поэтому исходный объект и построенный объект не будут иметь общих поверхностей, кривых, граней, рёбер, вершин и других объектов. Этот вариант используется в случаях, когда требуется, чтобы построенный объект не был связан с исходным объектом.

При значении *cm_KeepSurface* исходный объект и построенный объект будут иметь одни и те же базовые поверхности граней. Этот вариант используется в случаях, когда требуется высокая скорость построения.

При значении *cm_KeepHistory* исходный объект и построенный объект будут иметь одни и те же вершины, базовые поверхности граней и грани, не модифицированные выполненным построением или другим действием. Этот вариант используется в случаях, когда требуется максимально экономить память.

При значении *cm_Same* в построенный объект будут перенесены все необходимые данные исходного объекта, поэтому после построения исходный объект должен быть удалён. Этот вариант используется в случаях, когда исходный объект в дальнейшем не нужен и был построен специально для выполненного построения.

Перечисление MbeCopyMode присутствует в методе копирования множества граней MbFaceShell MbFaceShell::Copy(MbeCopyMode, MbShellHistory). Этот метод используется в операциях построения тел, в составе входящих параметров которых присутствуют другие тела.

При значении перечисления *cm_Copy* исходное множество граней и его копия не имеют общих данных. Варианту, при котором исходное множество граней и его копия имеют общие базовые поверхности, соответствует значение перечисления *cm_KeepSurface*. Варианту, при котором исходное множество граней и его копия имеют общие базовые поверхности, вершины и не изменённые операцией грани, соответствует значение перечисления *cm_KeepHistory*. В этом случае метод Copy(...) использует указатель на объект MbShellHistory, который запоминает соответствие между исходным множеством граней и его копией. После выполнения операции копия множества граней передаётся параметром в метод MbShellHistory::SetOrigins(MbFaceShell &) для замены в копии не изменённых граней их оригиналами исходного множества граней. Варианту, при котором множество граней в методе Copy(...) не копируется, соответствует значение перечисления *cm_Same*. Следует заметить, что при неудачном выполнении операции исходное множество граней будет модифицировано.

Для копирования множества граней MbFaceShell можно использовать метод Duplicate(MbRegDuplicate * iReg). Объект MbRegDuplicate используется для сохранения в копии структуры взаимных ссылок, которая присутствует в исходном множестве граней. Копируемое множество граней и его копия не будут связаны между собой.

3.15.10 Именование граней, рёбер и вершин

Грани, рёбра и вершины в структуре данных имеют имя MbName. Класс MbName объявлен в файле *name_item.h*. Именование граней, рёбер и вершин выполняется геометрическим ядром C3D при построении множества граней *MbFaceShell* во всех формообразующих операциях. В параметрах каждой формообразующей операции присутствует объект MbSNameMaker. Объект MbSNameMaker содержит главное имя операции и контейнер простых имён типа SimpleName, которые используются для именования граней. Объект MbSNameMaker выполняет именование построенных граней с помощью элементов кон-

тейнера простых имён. Контейнер простых имён может иметь или набор уникальных чисел, или одно целое число, или ни одного целого числа.

Если контейнер простых имён содержит одно число, то остальные требуемые простые имена MbSNameMaker создаст сам путём прибавления к исходному числу чисел натуральной последовательности. Если контейнер простых имён пуст, то стартовым числом будет нуль. Каждое число из контейнера простых имён будет соответствовать одному из геометрических входных параметров (сегменту формообразующего контура операции, обрабатываемому ребру операции, модифицируемой грани операции и тому подобное) и будет использоваться для именования новой грани, рождённой операцией.

Имена граней будут уникальными при уникальности элементов контейнера простых имён. Рёбра получают имена хешированием имён соединяемых ими граней. Вершины получают имена хешированием имён стыкующихся в них рёбер. В качестве идентификатора грани, ребра или вершины можно использовать число, которое выдаст метод MbName::Hash() соответствующего топологического объекта. Грани, рёбра и вершины в теле можно искать по известному имени MbName методами FindFaceByName(...), FindEdgeByName(...), FindVertexByName(...).

Кроме того, объект MbSNameMaker содержит версию способа построения и обеспечивает сохранение старых способов построений при их модификации в процессе развития геометрического ядра.

3.15.11 Принцип работы флагов изменения топологических объектов

Флаги изменения имеются у всех топологических объектов (MbTopologyItem). Перед выполнением каждой операции они выставляются в состояние *tct_Unchanged*, что означает отсутствие изменений.

Для топологических объектов существуют следующие флаги изменений:

tct_Unchanged — объект в операции не изменялся;

tct_Modified — при любой модификации объекта устанавливается этот флаг;

tct_Created — объект был создан. В теле не было этого объекта, а затем он был добавлен (например, при создании элементарного тела функцией ElementarySolid);

tct_Transformed — объект был перемещён (например, при применении перемещения к геометрии тела (MbSolid::Move) все объекты тоже переместятся);

tct_Reoriented — грань создана как копия другой грани, у которой сменилось направление нормального вектора на противоположное, то есть грань «вывернулась». Пример: при булевой операции новая грань в исходном теле была создана из копии грани режущего тела, у которой было изменено направление нормали;

tct_Deleted — при удалении дочерних элементов объекта или пометке самого объекта на удаление (например, при удалении цикла рёбер (при его замене в операции) или поверхности грани);

tct_Truncated — объект (грань, ребро) был подрезан в операции;

tct_Merged — флаг актуален для рёбер (MbEdge) — ребро было объединено или сшито;

tct_Replaced — у грани в процессе операции была заменена поверхность или цикл рёбер грани;

tct_Added — добавлен дочерний элемент;

tct_Renamed — объект был переименован в процессе операции.

Рассмотрим применение флагов на примере выполнения операции SolidCutting (см. *Рис. 1*). Большой куб разрезается маленьким параллелепипедом:

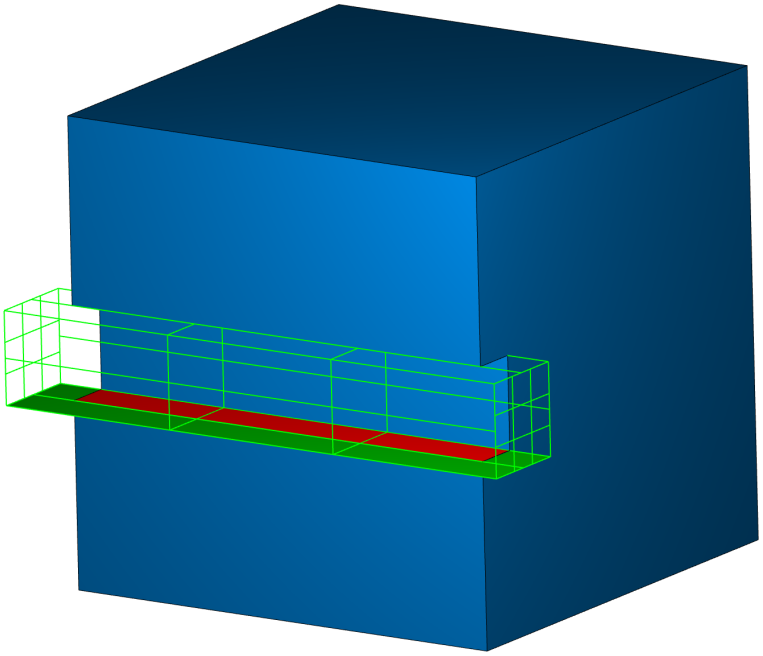


Рис. 1

Рассмотрим нижнюю грань паза в полученном теле (выделена красным цветом).

После операции она имеет следующие флаги:

tct_Modified, *tct_Created*, *tct_Reoriented*, *tct_Deleted*, *tct_Truncated*, *tct_Added*

Это означает следующее:

1. Грань была модифицирована каким-либо образом в выполненной операции (*tct_Modified*).
2. Этой грани ранее не было в этом теле и она была создана (*tct_Created*).
3. На самом деле грань была создана из грани режущего тела (выделена зелёным цветом), но вектор нормали был развёрнут (*tct_Reoriented*). Полное имя у неё полностью совпадает с именем грани режущего тела.
4. Грань была обрезана (*tct_Truncated*), для чего у неё был удалён (*tct_Deleted*) и добавлен (*tct_Added*) цикл рёбер грани.

Далее выполним ещё одно разрезание тем же режущим телом. Перед операцией изменим положение режущего параллелепипеда (*Рис. 2*).

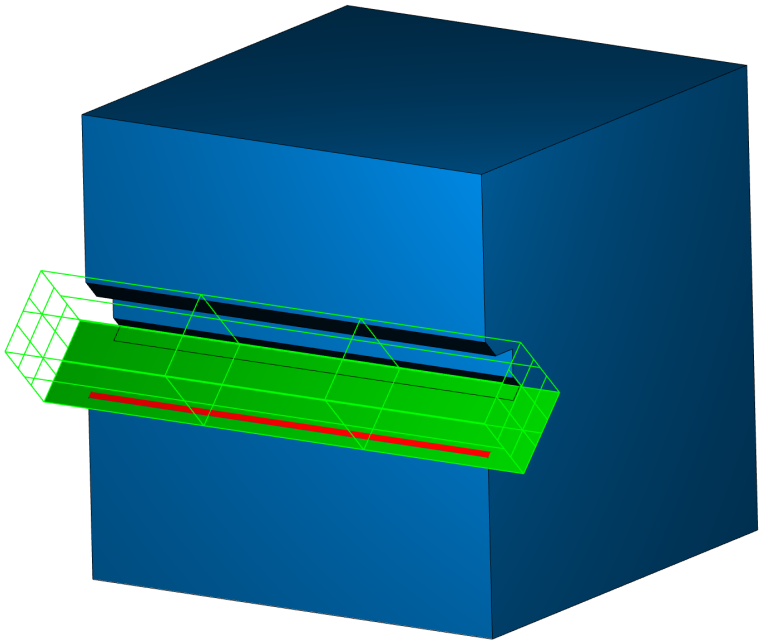


Рис. 2

Рассмотрим следующую грань в результате данного разрезания (выделена зелёным цветом на *Рис. 3*):

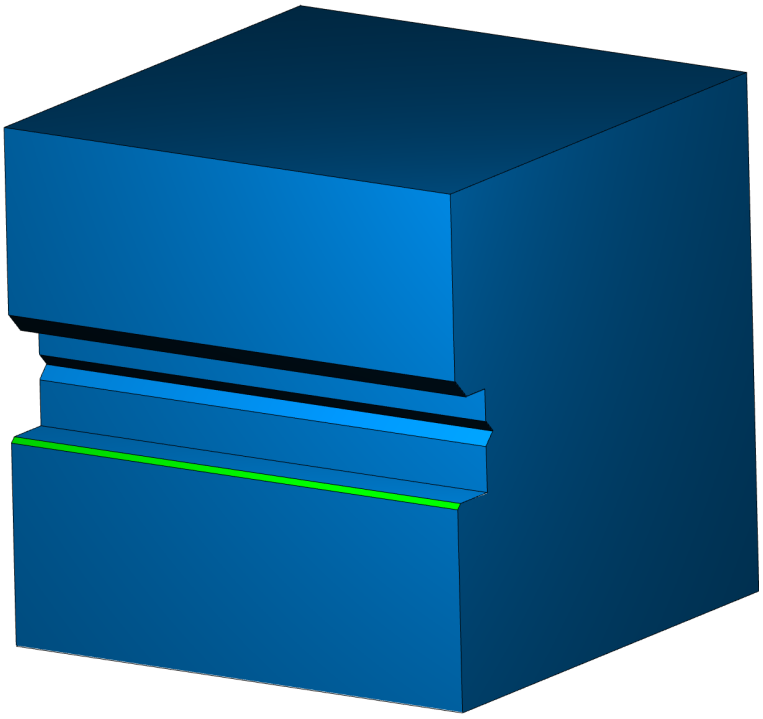


Рис. 3

После операции новая грань имеет следующие флаги:

tct_Modified, *tct_Created*, *tct_Transformed*, *tct_Reoriented*, *tct_Deleted*, *tct_Truncated*, *tct_Added*, *tct_Replaced*

Это означает следующее:

1. Грань была модифицирована каким-либо образом в выполненной операции (*tct_Modified*).
2. Этой грани ранее не было в этом теле и она была создана (*tct_Created*).
3. На самом деле грань была создана из грани режущего тела (*Рис. 2*), но вектор нормали был развёрнут (*tct_Reoriented*). Режущее тело было перемещено перед операцией (*tct_Transformed*).
4. Грань была обрезана (*tct_Truncated*), для чего у неё был удалён (*tct_Deleted*) и добавлен (*tct_Added*) цикл рёбер грани.
5. У грани в процессе операции была заменена поверхность или цикл рёбер грани (*tct_Replaced*).

В классе MbTopologyItem определены (в файле *topology_item.h*) следующие методы для чтения/записи флагов:

Метод

```
bool GetOwnChanged() const
```

получает флаг, свидетельствующий о том, что объект был (не был) изменён.

Метод

```
bool GetOwnChanged( MbeChangedType n ) const
```

получает флаг, свидетельствующий о том, что объект был (не был) изменён.

Метод

```
void SetOwnChanged( MbeChangedType )
```

устанавливает флаг, свидетельствующий о том, что объект был (не был) изменён.

Метод

```
void CopyOwnChanged( const MbTopologyItem & ti )
```

копирует флаг, свидетельствующий о том, что объект был (не был) изменён.

Метод

```
uint16 GetOwnChangedFlag() const
```

получает флаг, свидетельствующий о том, что объект был (не был) изменён.

Метод

```
template<class Uint> void SetOwnChangedFlag( Uint n )
```

устанавливает флаг, свидетельствующий о том, что объект был (не был) изменён.

Метод

```
bool IsOwnRenamedOnly() const
```

получает флаг, свидетельствующий о том, что объект был только переименован.

Метод

```
bool IsOwnTransformedOnly() const
```

получает флаг, свидетельствующий о том, что объект был только трансформирован.

Метод

```
bool IsOwnReorientedOnly() const
```

получает флаг, свидетельствующий о том, что объект был только переориентирован.

Метод

```
bool IsOwnChangedWeakly() const
```

получает флаг, свидетельствующий о том, что объект был только создан и/или к нему были применены только не изменяющие структуру геометрии объекта «слабые» изменения (переименование, трансформация, переориентация). При наличии любых других изменений метод возвращает false.

3.16 ОБЪЕКТЫ ГЕОМЕТРИЧЕСКОЙ МОДЕЛИ

Среди трёхмерных геометрических объектов выделен класс объектов геометрической модели. Для твердотельного, поверхностного и прямого моделирования используется такой объект геометрической модели, как тело. Тела строятся и при моделировании объектов из листового металла. Кроме тела к объектам геометрической модели относятся проволочный каркас, точечный каркас и полигональный объект. Из объектов геометрической модели можно создавать сборочные единицы. Для вспомогательных построений могут использоваться такие объекты геометрической модели как локальная система координат. Среди объектов геометрической модели есть объект для построения эскизов.

3.16.1 Объект геометрической модели MbItem

Класс MbItem объявлен в файле *model_item.h*.

Объект геометрической модели MbItem является наследником классов *MbSpaceItem*, MbTransactions и MbAttributeContainer. Геометрическое ядро C3D оперирует объектами геометрической модели, которые приведены на рис. 1:

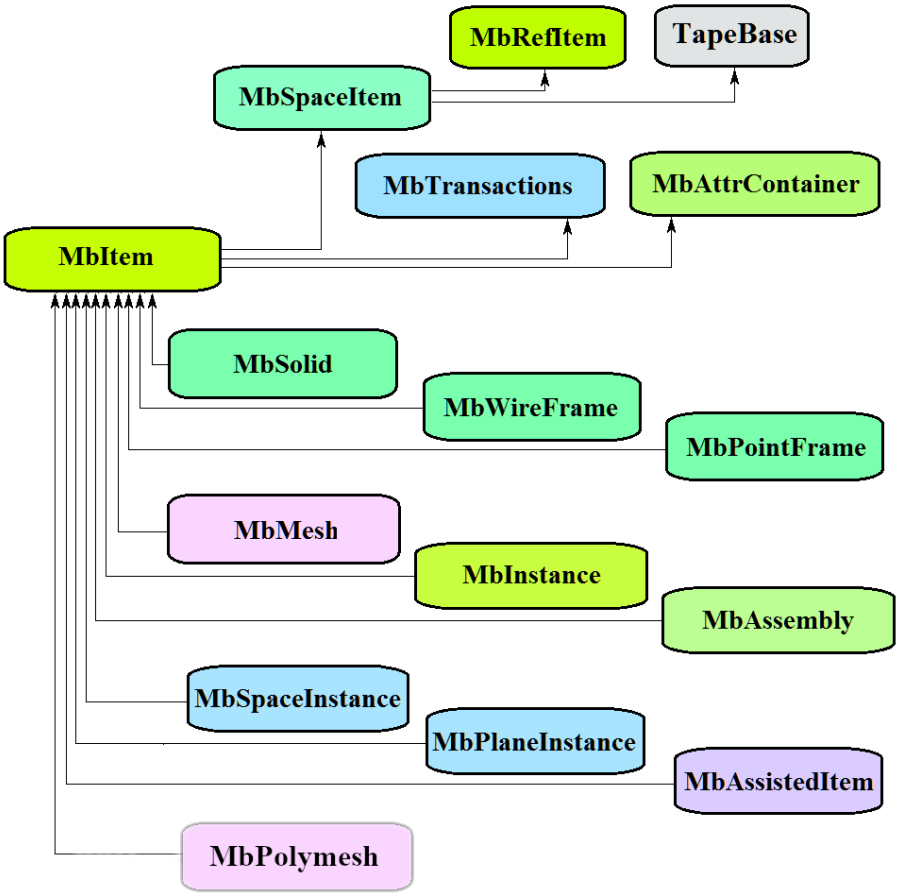


Рис. 1

Журнал построения MbTransactions содержит данные, необходимые для построения объекта и позволяет повторить построение объекта с отредактированными параметрами. Контейнер атрибутов MbAttrContainer наделяет объекты геометрической модели атрибутами. Таким образом, все объекты геометрической модели кроме своих специфических данных содержат следующие данные:

size_t *m_countRegistrable* – количество регистрирований объекта при записи и чтении,

ptrdiff_t *useCount* – количество использований объекта другими объектами,

std::vector< MbCreator * > **transactions** – упорядоченное множество строителей объекта,

std::multimap< int, MbAttribute * > **attributes** – множество атрибутов объекта.

Наследниками класса MbItem являются следующие объекты геометрической модели:

MbSolid – твёрдое тело,

MbWireFrame – проволочный каркас,

MbPointFrame – точечный каркас,

MbMesh – полигональный объект,

MbInstance – вставка объекта геометрической модели,

MbAssembly – сборочная единица объектов геометрической модели,

MbAssistingItem – вспомогательный объект,

MbSpaceInstance – вставка трёхмерного объекта,

MbPlaneInstance – вставка множества двумерных объектов.

MbPolymesh – полигональный объект с топологией.

Основными методами объектов геометрической модели являются методы, обеспечивающие редактирование и визуализацию объектов.

Метод

bool RebuildItem(MbeCopyMode sameShell, RPArray< *MbSpaceItem* > * **items**)

выполняет построение объекта заново по журналу построения. Этот метод вызывается после редактирования внутренних данных объекта.

Метод

MbItem * CreateMesh(MbeStepData data, bool wire, bool grid, MbRegDuplicate * iReg)

создаёт полигональную копию объекта. Если объект представляет собой сборочную единицу или вставку, то копия объекта будет также сборочной единицей или вставкой с полигональными объектами.

Метод

bool AddYourMesh(MbeStepData data, bool wire, bool grid, *MbItem* & **mesh**)

добавляет полигональную копию объекта в объект **mesh**.

Метод

bool NearestMesh(MbeSpaceType sType, MbeTopologyType tType, MbePlaneType pType, const MbAxis3D & **axis**, double maxDistance, double & t, double & dMin, MbItem *& **find*, SimpleName & findName, *MbRefItem* *& **element*, SimpleName & elementName, MbPath & path, *MbMatrix3D* & **from**)

выполняет поиск ближайшего полигонального объекта **find**, его элемента **element**, их имён findName и elementName, пути в структуре сборочной единицы path и матрицы преобразования в глобальную систему координат **from**.

Объекты геометрической модели перегружают такие методы трёхмерного объекта как:

методы, обслуживающие преобразование геометрического объекта,

void Move(const *MbVector3D* & **v**, MbRegTransform * iReg = NULL),

void Rotate(const MbAxis3D & **axis**, double **angle**, MbRegTransform * iReg = NULL),

void Transform(const *MbMatrix3D* & **m**, MbRegTransform * iReg = NULL),

методы, обеспечивающие копирование, проверку на совпадение, проверку на возможность сделать совпадающими, делающие объекты совпадающими,

MbSpaceItem & Duplicate(MbRegDuplicate * iReg = NULL),

bool IsSame(const *MbSpaceItem* & **item**),

bool IsSimilar(const *MbSpaceItem* & **item**),

bool SetEqual(const *MbSpaceItem* & **item**),

методы, возвращающие тип из перечисления геометрических объектов:

MbeSpaceType IsA(),

MbeSpaceType Type(),

MbeSpaceType Family(),

методы, обеспечивающие выдачу и редактирование внутренних данных объекта:

MbProperty & CreateProperty(MbePrompt name),

GetProperties(MbProperties & *properties*),

SetProperties(MbProperties & *properties*).

3.16.2 Твёрдое тело MbSolid

Класс MbSolid объявлен в файле *solid.h*.

Твёрдое тело или просто тело MbSolid является наследником класса *MbItem* и описывается множеством граней *MbFaceShell* *outer* и типом связности *multiState*.

Тело представляет собой набор граней, стыкующихся друг с другом по рёбрам и описывающих поверхность моделируемого объекта. Тело может описывать одно или несколько связных множеств точек. Тип связности *multiState* сообщает о том, что тело описывает одно связное множество точек или что тело описывает несколько связных множеств точек и может быть разбито на несколько тел.

Множество граней тела *outer* в зависимости от наличия краевых рёбер может описывать два принципиально разных множества точек. Если множество граней не имеет края, то тело описывает множество точек, располагающихся на поверхности граней и с внутренней стороны от этих граней. Такое тело называется *замкнутым* и описывается замкнутой оболочкой. Замкнутое тело приведено на рис. 1:

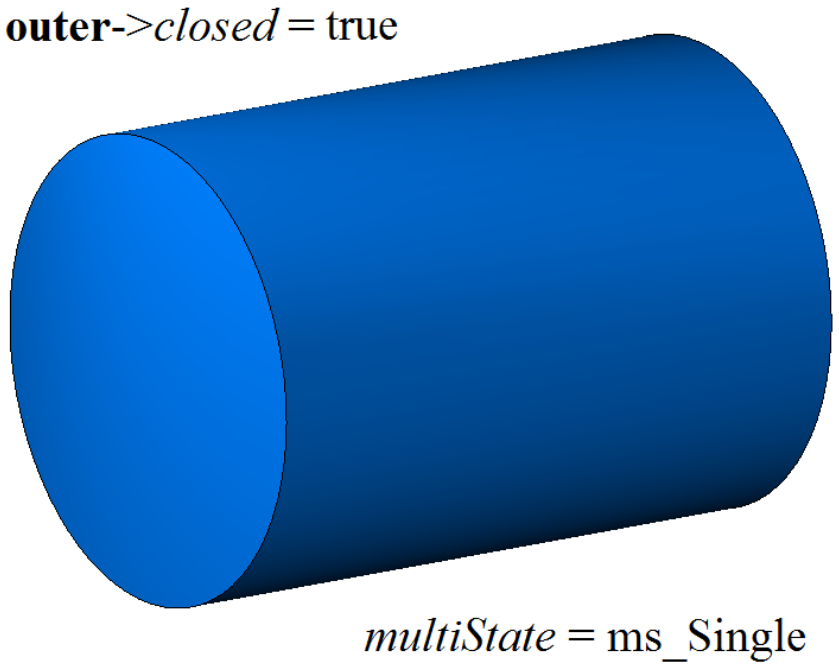


Рис. 1

Если множество граней тела имеет край или края, то тело описывает множество точек, располагающихся на поверхности граней и только. Такое тело называется *незамкнутым* и описывается незамкнутой оболочкой. Незамкнутое тело приведено на рис. 2:

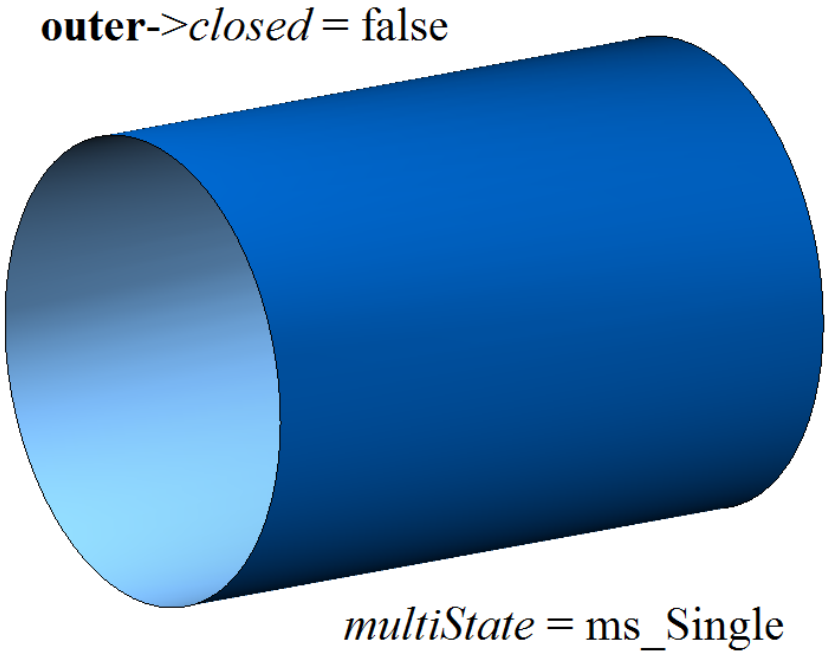


Рис. 2

В большинстве случаев замкнутое тело описывается одним связным множеством граней – одной оболочкой. Если у замкнутого тела есть внутренние полости, тело описывается несколькими связными наборами граней. На рис. 3 приведено замкнутое тело, которое описывается двумя замкнутыми оболочками, одной внешней, а другой внутренней, располагающейся внутри внешней оболочки.

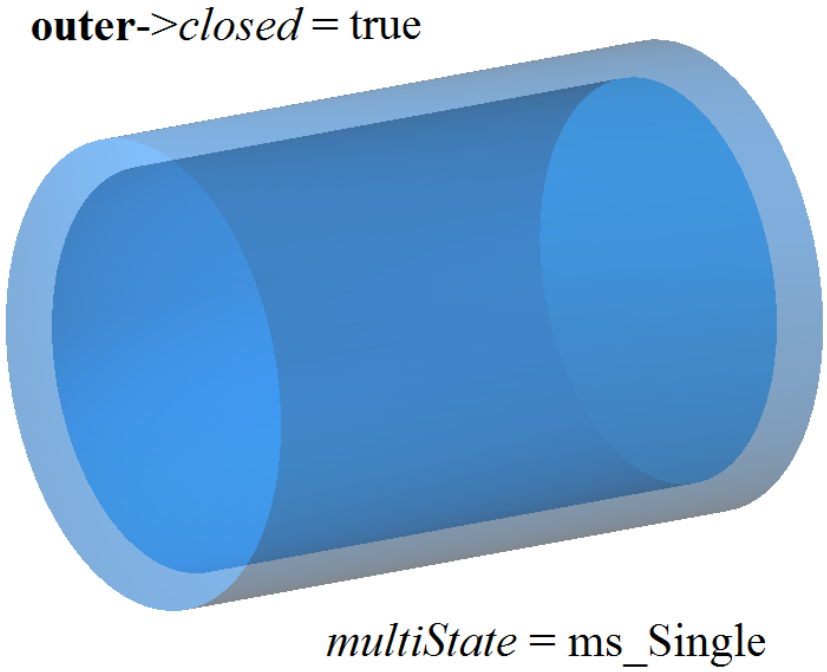


Рис. 3

Чтобы увидеть полость тела, тело выполнено полупрозрачным.

Над телами можно выполнять различные операции – совокупность действий, которая приводит к образованию тела иной формы, например булевы операции. Результат вычитания двух замкнутых тел приведён на рис. 4:

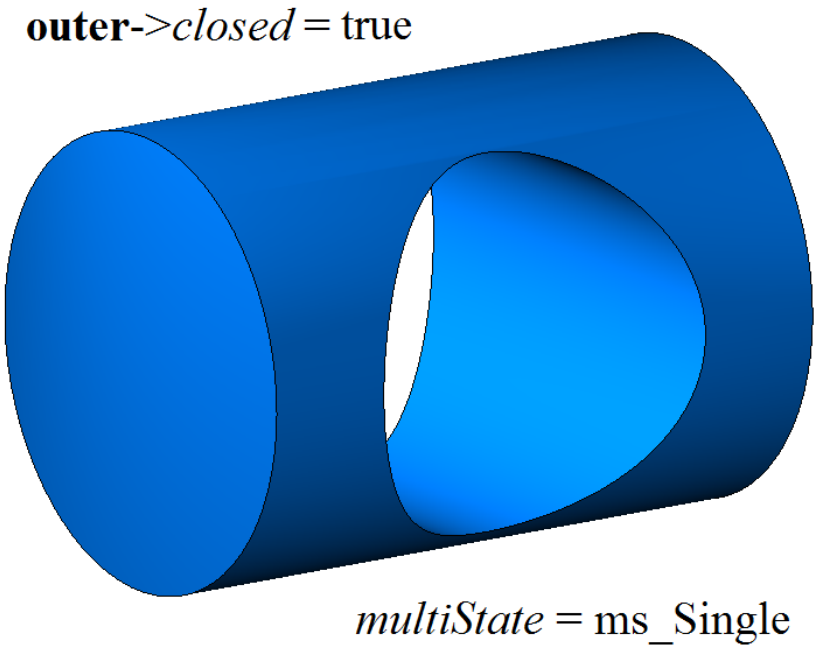


Рис. 4

Результат операции над замкнутым и незамкнутым телом принципиально отличается, так как действия выполняются над разными множествами точек. Результат вычитания тела из незамкнутого тела приведён на рис. 5:

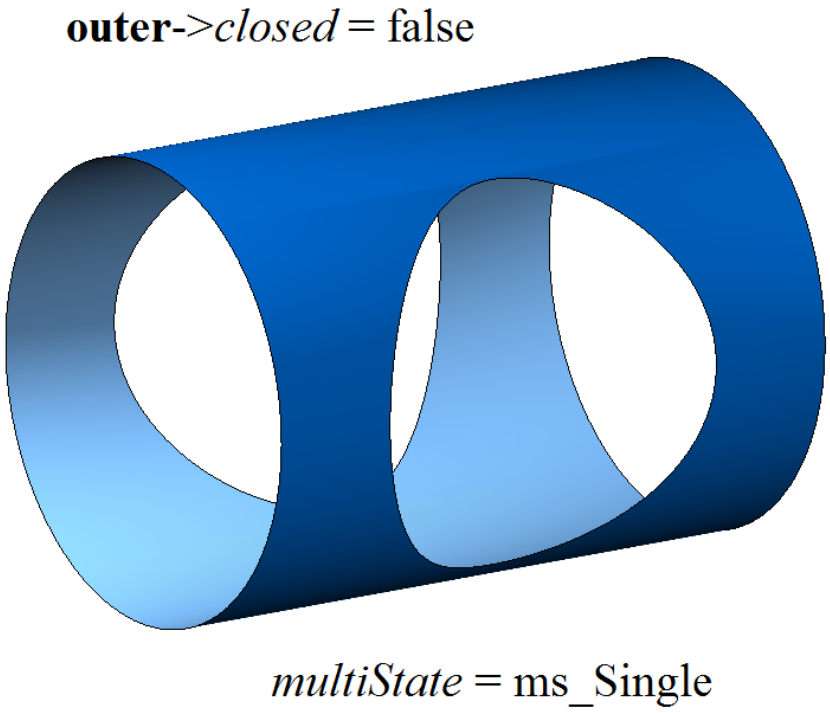


Рис. 5

Тело может быть многосвязным, то есть может состоять из нескольких отдельных частей. В этом случае *multiState* принимает значение ms_Multiple. Двусвязное тело, описываемое двумя замкнутыми оболочками, приведено на рис. 6.

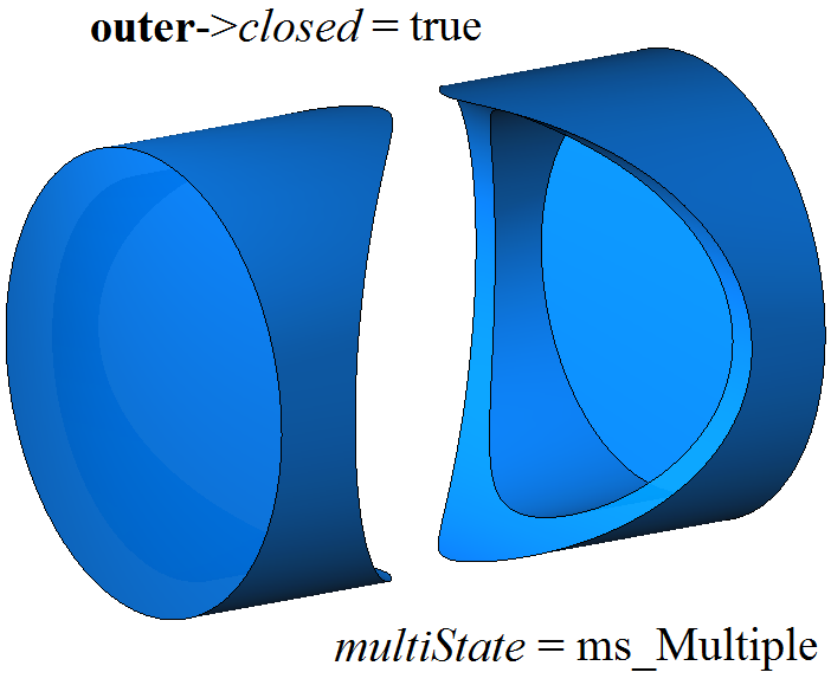


Рис. 6

Такое тело может быть разделено на два односвязных замкнутых тела методом DetachParts(). Тело на рис. 6 выполнено полупрозрачным. Двусвязное тело, описываемое двумя незамкнутыми оболочками, приведено на рис. 7:

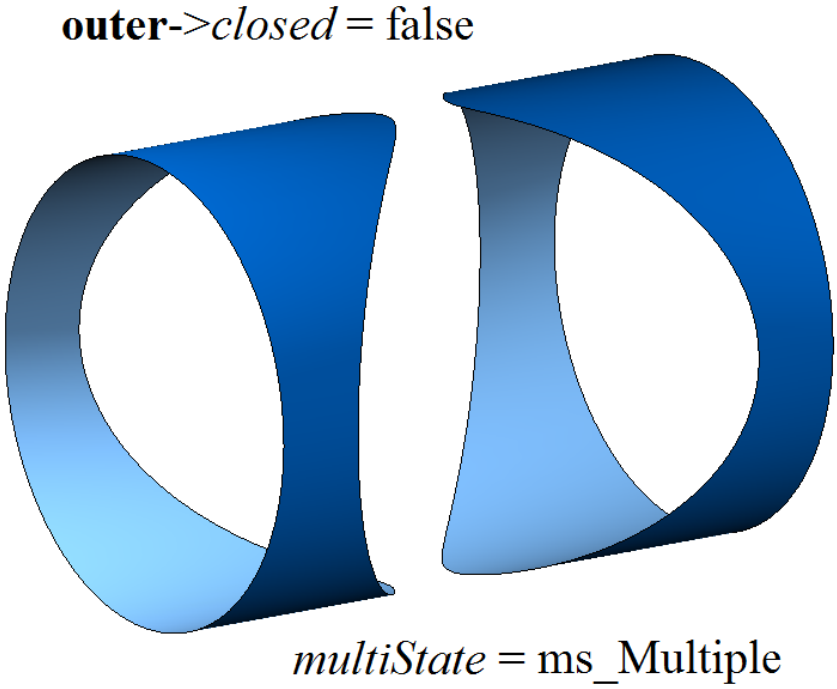


Рис. 7

Такое тело может быть разделено на два односвязных незамкнутых тела.

Тела создаются методами геометрического ядра C3D. Тела простой формы создаются по точкам, кривым и поверхностям. С помощью операций из тел простой формы можно получить более сложные тела. Редактировать исходные тела и создавать подобные тела можно путём изменения параметров журнала построения MbTransactions или путём непосредственной модификации элементов уже построенных тел. Незамкнутые тела используются в поверхностном моделировании. Незамкнутое тело позволяет сосредоточить усилия на сложных формах моделируемого объекта.

3.16.3 Проволочный каркас MbWireFrame

Класс MbWireFrame объявлен в файле *wire_frame.h*.

Проволочный каркас MbWireFrame является наследником класса *MbItem* и описывается множеством рёбер std::vector< *MbEdge* * > **edges**, количеством связных частей *parts* и признаком отсутствия краевых вершин *closed*.

Проволочный каркас представляет собой множество рёбер, стыкующихся друг с другом в вершинах и описывающих каркасную конструкцию моделируемого объекта. Проволочный каркас может описывать одно или несколько связных множеств точек. Количество связных частей *parts* сообщает о том, что каркас описывает одно связное множество точек или что каркас описывает несколько связных множеств точек и может быть разбит на несколько проволочных каркасов.

В зависимости от отсутствия или наличия краевых вершин проволочный каркас будем называть *замкнутым* или *незамкнутым*. На рис. 1 приведён замкнутый проволочный каркас:

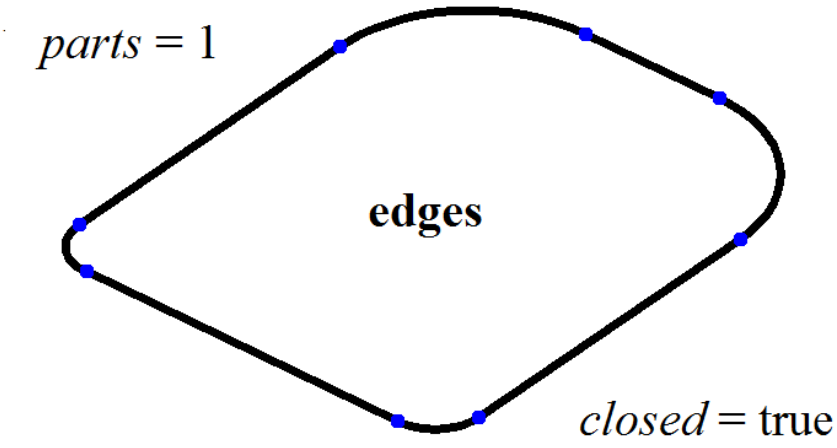


Рис. 1

Незамкнутый проволочный каркас приведён на рис. 2:

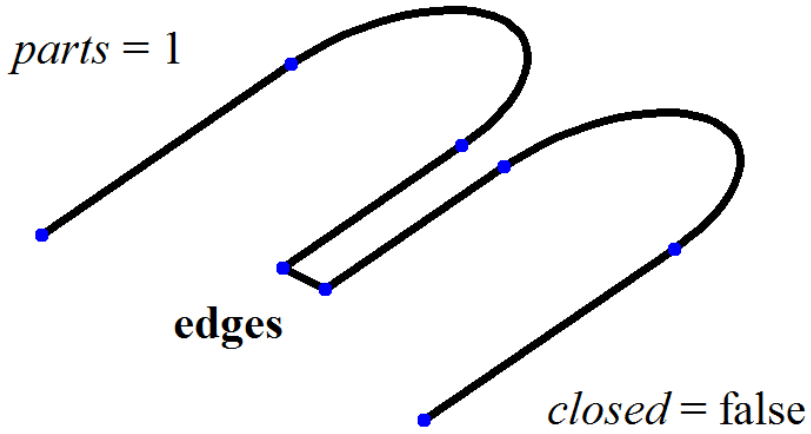


Рис. 2

На рис. 3 приведён незамкнутый проволочный каркас, состоящий из двух связных множеств рёбер:

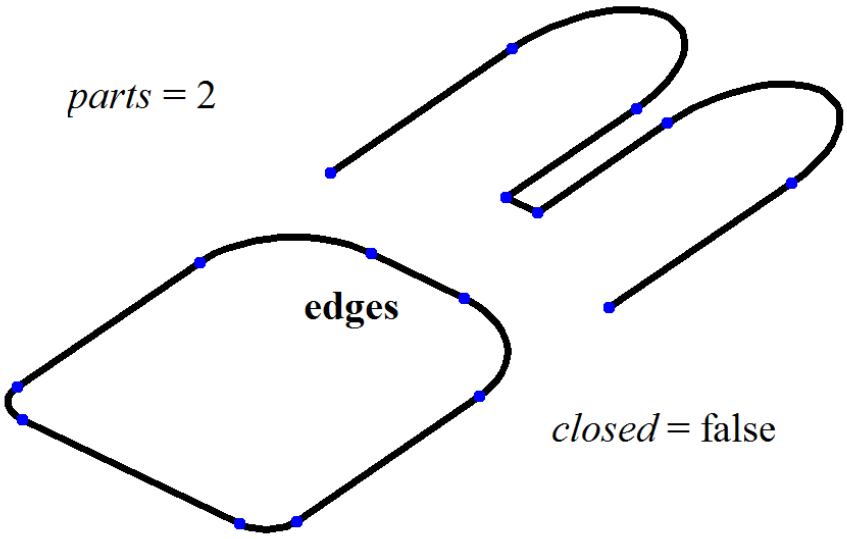


Рис. 3

На рис. 4 приведён замкнутый проволочный каркас, состоящий из двух связных множеств рёбер:

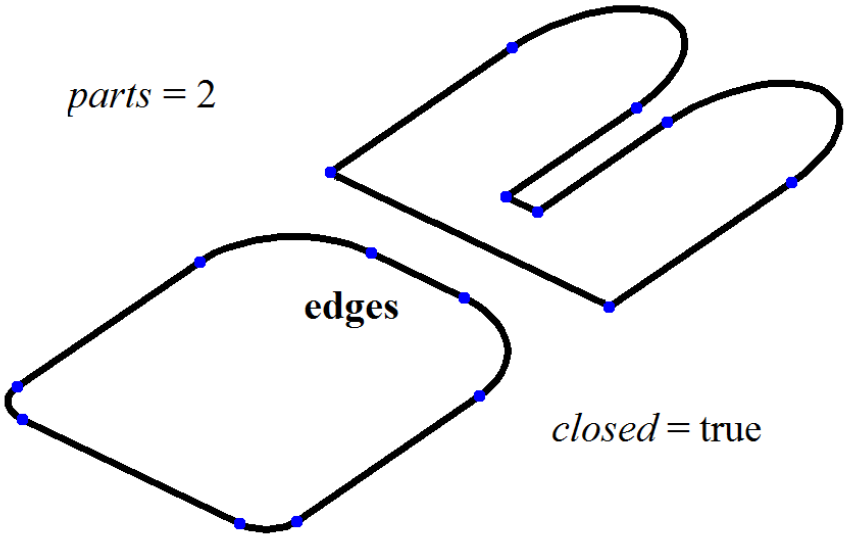


Рис. 4

Проволочный каркас может использоваться для построения траекторий, пространственных эскизов и для вспомогательных построений.

3.16.4 Точечный каркас MbPointFrame

Класс MbPointFrame объявлен в файле *point_frame.h*.

Точечный каркас MbPointFrame является наследником класса *MbItem* и описывается множеством точек, представленных в виде вершин, std::vector< *MbVertex* * > **vertices**.

На рис. 1 приведён точечный каркас:

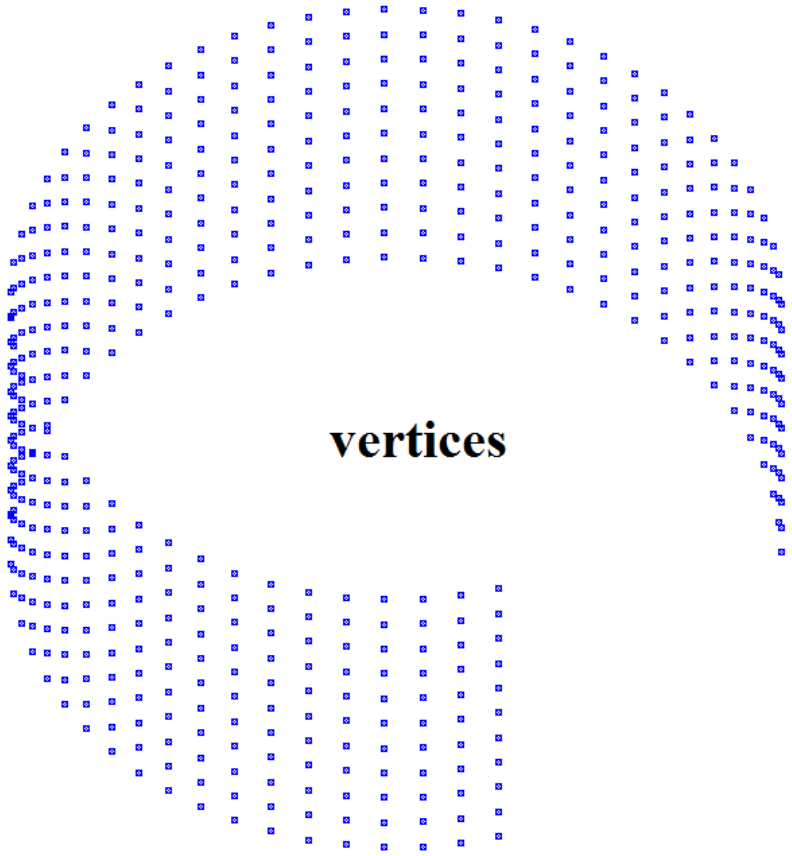


Рис. 1

Точечный каркас может использоваться для позиционирования других объектов и для вспомогательных построений.

3.16.5 Полигональный объект MbMesh

Класс MbMesh объявлен в файле *mesh.h*.

Полигональный объект MbMesh является наследником класса MbItem и описывается множеством триангуляций RPArray< MbGrid > **grids**, множеством полигонов RPArray< MbPolygon3D > **wires**, множеством апексов RPArray< MbApex3D > **peaks**, указателем на оригинальный объект *MbRefItem item*, типом *type*, габаритным кубом **cube** и признаком замкнутости *closed*.

Полигональный объект представляет собой набор треугольных и четырёхугольных пластин, ломаных линий и отдельных точек. Одним из способов построения полигонального объекта является аппроксимация других объектов геометрической модели, например тела. Каждая *i*-ая грань тела аппроксимируется триангуляцией **grids**[*i*], каждое *j*-ое ребро тела аппроксимируется полигоном **wires**[*j*], каждой *k*-ой вершине тела соответствует апекс **peaks**[*k*], признак замкнутости *closed* соответствует замкнутости тела. Ещё одним способом получения полигонального объекта является импорт данных с помощью конвертера полигонального представления.

Триангуляция MbGrid представляет собой множество точек SArray< MbFloatPoint3D > **points**, согласованное с ним множество нормалей SArray< MbFloatVector3D > **normals** (количество точек равно количеству нормалей), множество двумерных точек параметрической области поверхности SArray< MbFloatPoint > **params** (количество двумерных точек равно количеству трёхмерных точек или равно нулю – множество может быть пустым), множество треугольных пластин SArray< MbTriangle > *triangles* в виде трёх индексов множества точек **points**, множество четырёхугольных пластин SArray< MbQuadrangle > *quadrangles* в виде четырёх индексов множества точек **points**. Пластины триангуляции аппроксимируют некоторую поверхность.

Полигон MbPolygon3D представляет собой упорядоченное множество точек, последовательное соединение которых даёт ломаную линию, аппроксимирующую некоторую кривую.

Апекс MbApex3D представляет собой точку, наделённую дополнительными данными.

Указатель на оригинальный объект **item** может быть равен нулю, а тип *type* может быть неопределённым.

Векторное изображение полигонального объекта приведено на рис. 1:

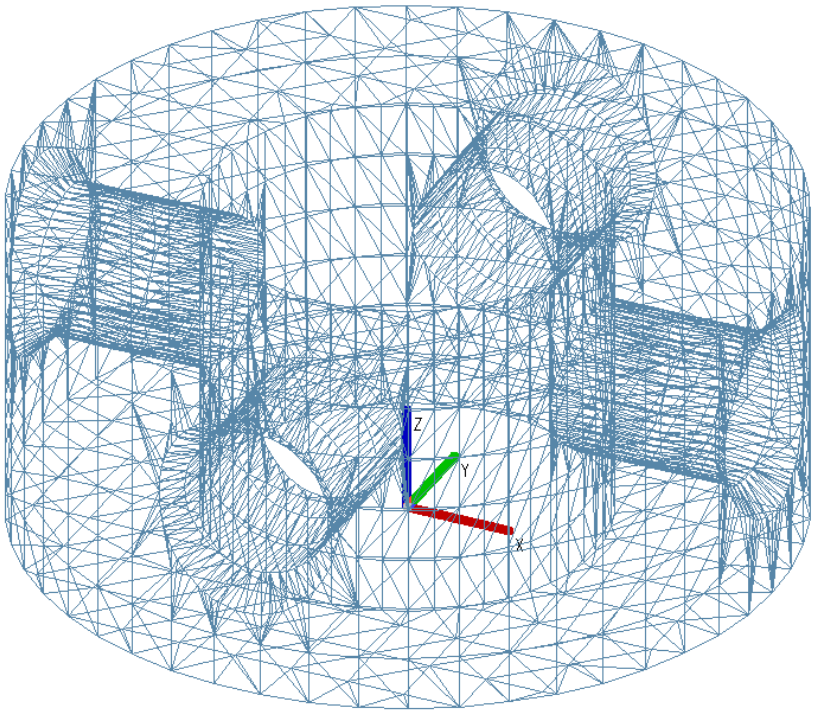


Рис. 1

Тонированное изображение полигонального объекта приведено на рис. 2:

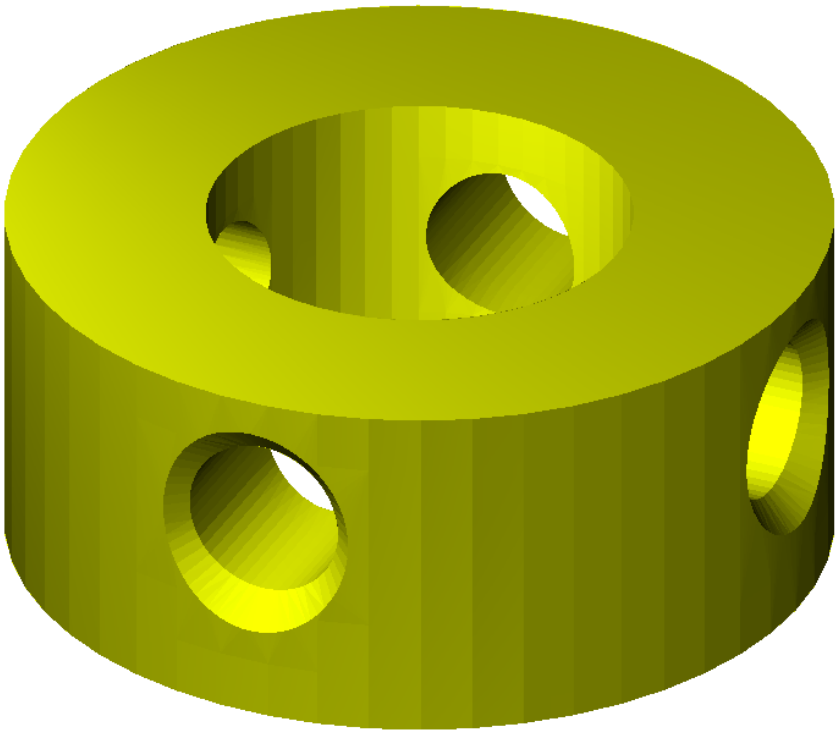


Рис. 2

На рисунке видны отдельные треугольники объекта из-за того, что направление нормалей в каждом треугольнике постоянно. Если направление нормалей в треугольниках триангуляции непрерывно меняется, то отдельные треугольники объекта становятся не видны, см. рис. 3:

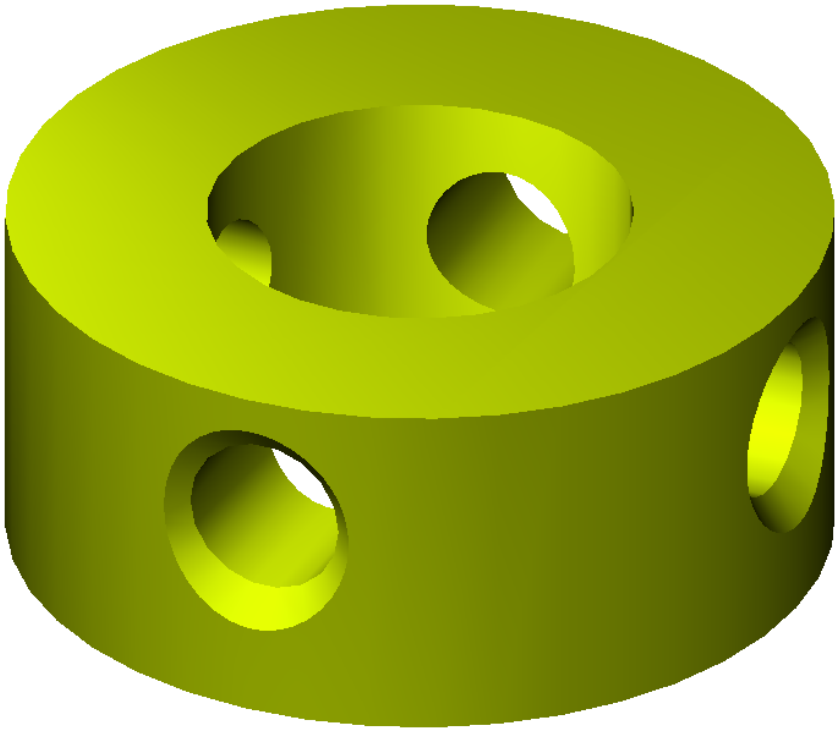


Рис. 3

Полигональный объект может быть создан методом MbItem:: CreateMesh() или методом MbItem:: AddYourMesh(). Полигональный объект используется для визуализации, расчётов и производства моделируемых объектов. Основным преимуществом полигонального объекта является простота и высокая скорость вычислений, например пересечения с прямой линией.

3.16.6 Вставка MbInstance

Класс MbInstance объявлен в файле *instance_item.h*.

Вставка объекта **MbInstance** является наследником класса *MbItem* и описывается объектом геометрической модели MbItem **item** и локальной системой координат *MbPlacement3D* **place**. Вставка представляет собой объект **item**, перенесённый в локальную систему координат **place**.

Вставка обладает всеми свойствами объекта **item**. Отличие заключается в том, что методы Move(), Rotate(), Transform() изменяют систему координат **place**, не модифицируя геометрический объект **item**, который может совместно использоваться несколькими вставками.

Вставка объекта может содержать тело, проволочный каркас, точечный каркас, полигональный объект, но не может содержать другую вставку (**MbInstance** нельзя использовать в качестве **item** для другой **MbInstance**).

3.16.7 Сборочная единица MbAssembly

Класс MbAssembly объявлен в файле *assembly.h*.

Сборочная единица MbAssembly или сборка является наследником класса MbItem и описывается множеством объектов геометрической модели std::vector< MbItem * > **assemblyItems**.

Сборка представляет собой множество объектов геометрической модели, которыми можно оперировать как единым объектом.

Сборочная единица приведена на рис. 1:

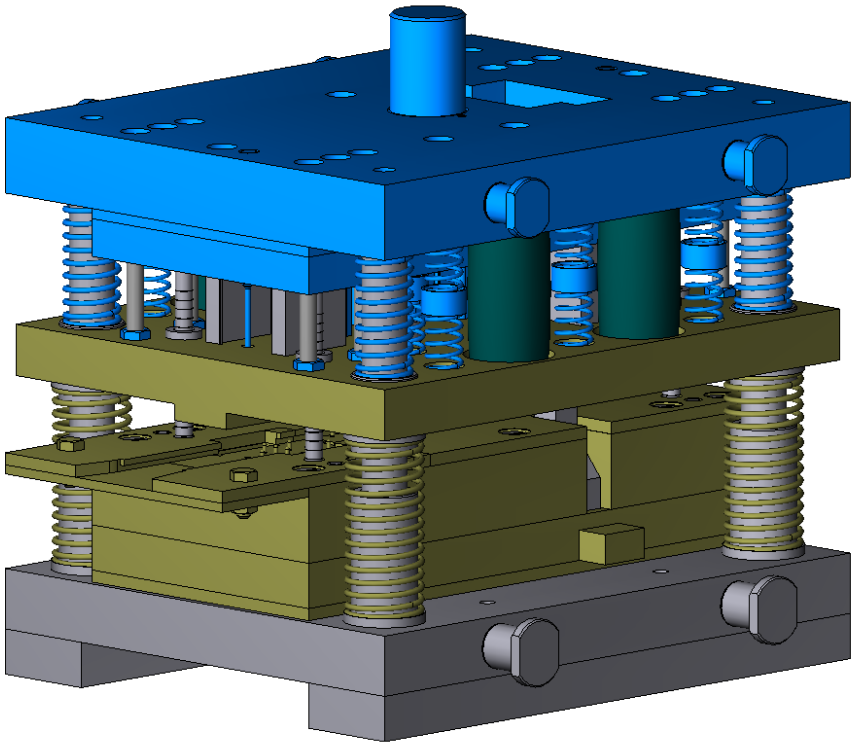


Рис. 1

Сборочная единица может содержать другие сборочные единицы в качестве своих объектов, то есть может иметь древовидную структуру.

3.16.8 Вставка трёхмерного объекта MbSpaceInstance

Класс MbSpaceInstance объявлен в файле *space_instanse.h*.

Вставка трёхмерного объекта MbSpaceInstance является наследником класса *MbItem* и описывается геометрическим объектом *MbSpaceItem* *spaceItem*.

Вставка играет роль обёртки геометрического объекта, позволяющей работать с ним как с объектом геометрической модели. Вставка придаёт обычному геометрическому объекту журнал построения, наделяет объект атрибутами и методами объекта геометрической модели *MbItem*. Вставка трёхмерного объекта предназначена для вспомогательных построений. Вставка поверхности приведена на рис. 1:

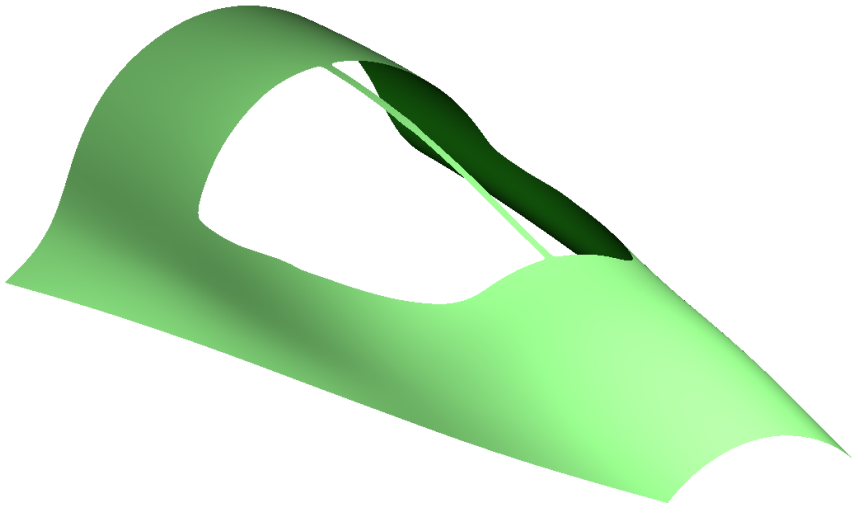


Рис. 1

Вставка объекта может содержать поверхность *MbSurface*, кривую *MbCurve3D*, точку MbPoint3D или вспомогательный геометрический объект *MbLegend*. Для объектов, наследников объекта геометрической модели MbItem (тело, проволочный каркас, точечный каркас, полигональный объект, сборка, вставка), вставка геометрического объекта не применяется.

3.16.9 Вставка двумерных объектов MbPlaneInstance

Класс MbPlaneInstance объявлен в файле *plane_instanse.h*.

Вставка двумерных объектов MbPlaneInstance является наследником класса *MbItem* и описывается множеством двумерных геометрических объектов std::vector< *MbPlaneItem* > **planeItems** и локальной системой координат *MbPlacement3D* **place**. Двумерные объекты располагаются в плоскости XY локальной системы координат.

Вставка играет роль обёртки двумерных геометрических объектов, позволяющей работать с ними как с объектом геометрической модели. Вставка придаёт двумерным геометрическим объектам журнал построения, наделяет их атрибутами и методами объекта геометрической модели *MbItem*. Вставка двумерных объектов предназначена для вспомогательных построений. Вставка двумерных кривых приведена на рис. 1:

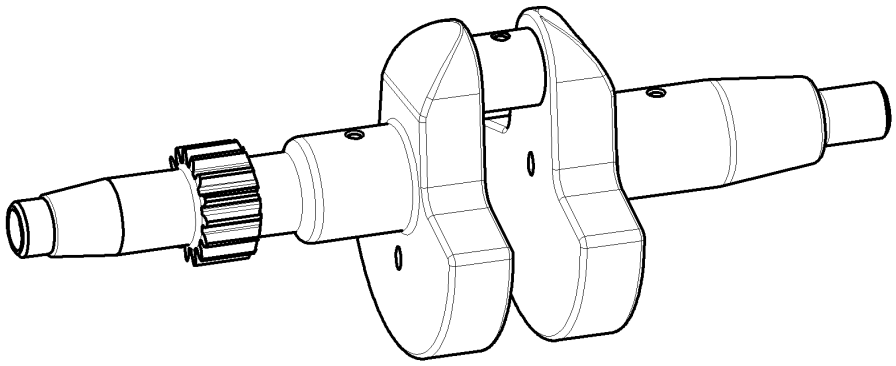


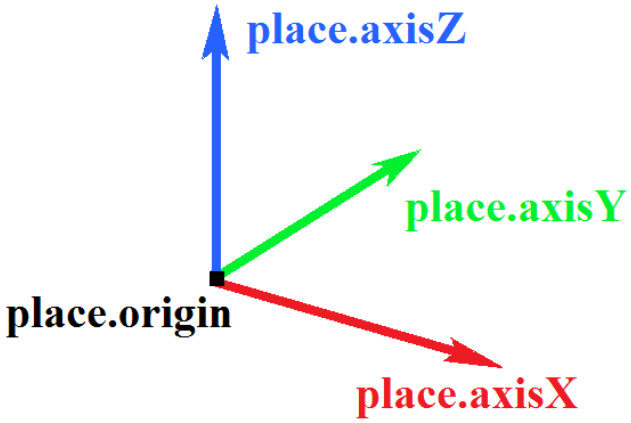
Рис. 1

Вставка двумерных объектов может содержать двумерную кривую MbCurve, мультилинию *MbMultiline* или регион *MbRegion*.

3.16.10 Вспомогательный объект MbAssistingItem

Класс MbAssistingItem объявлен в файле *assisting_item.h*.

Вспомогательный объект геометрической модели MbAssistingItem является наследником класса MbItem и описывается локальной системой координат MbPlacement3D **place**. Вспомогательный объект предназначен для позиционирования других объектов. Вспомогательный объект обладает журналом построения, атрибутами и методами объекта геометрической модели MbItem вспомогательных построений. Вспомогательный объект приведён на рисунке:



3.16.11 Полигональный объект с топологией MbPolymesh

Класс MbPolymesh объявлен в файле *polymesh.h*.

Полигональный объект MbPolymesh является наследником класса *MbItem* и описывается топологией *_pTopo*, представляющей собой описание полигонального объекта в виде структуры Halfedge.

Представление Halfedge – это представление полигональной модели в виде графа, где ключевым элементом представления является направленное ребро, ассоциированное с фасетом, которому оно принадлежит, и вершиной, из которой ребро исходит. Таким образом, каждое неграничное ребро поверхности представлено в виде пары противоположно направленных рёбер (реберной пары). Рёбра организуются в замкнутые циклы – фасеты.

Как дополнительный набор информации объект, помимо топологии, хранит информацию о сегментации объекта. Объект может быть разделён на регионы. Каждый регион содержит набор связных компонент, именуемых подрегионами. По аналогии с организацией Halfedge каждый подрегион может иметь один или несколько сегментационных рёберных циклов.

Для обеспечения гибкости представления каждому элементу топологии может быть назначен атрибут, хранящий дополнительную информацию о данном элементе. Так, например, у вершины может быть атрибут, хранящий информацию о её 3D-координатах. Через атрибуты возможно назначение пользователем производных данных типов bool, float, double, unsigned int, int и size_t.

Изображение полигонального объекта с визуализацией рёбер приведено на рис. 1. Регионы сегментации показаны разным цветом.

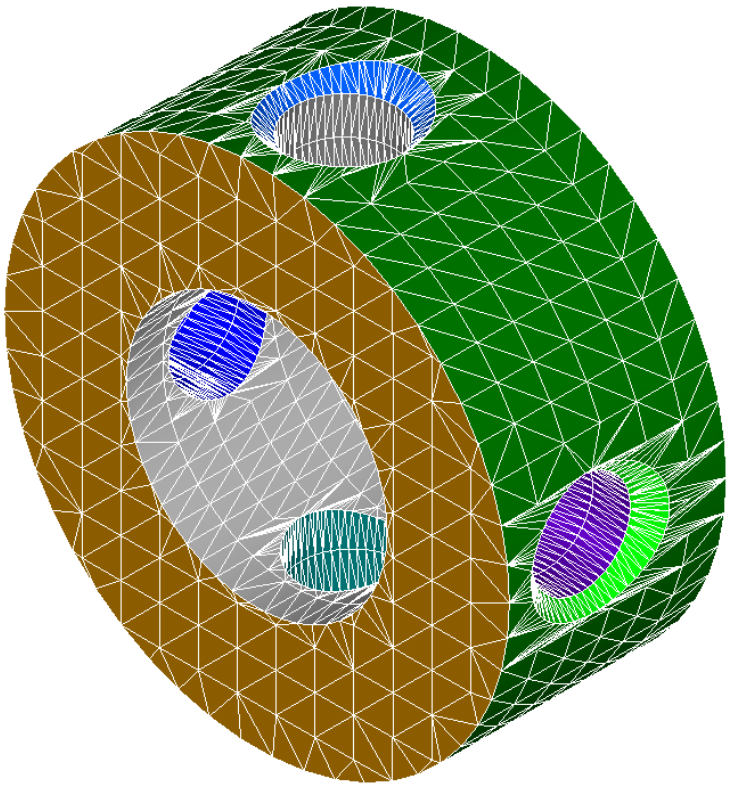


Рис. 1

Тонированное изображение полигонального объекта приведено на рис. 2. Циклы подрегионов показаны белым цветом.

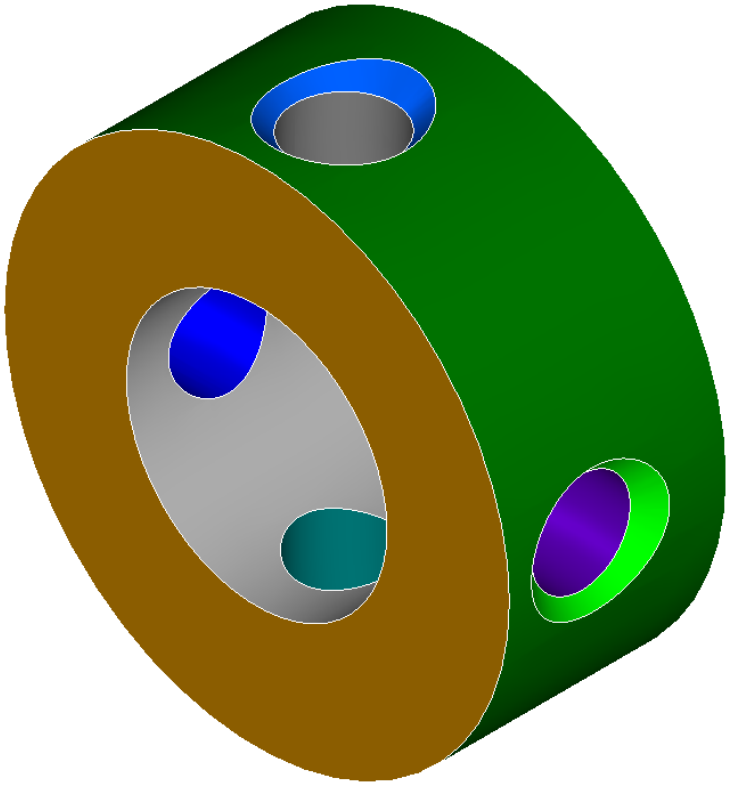


Рис. 2

Фасеты могут иметь произвольное количество рёбер. Например, на рис. 3 видны пятиугольные фасеты, составляющие грани додекаэдра:

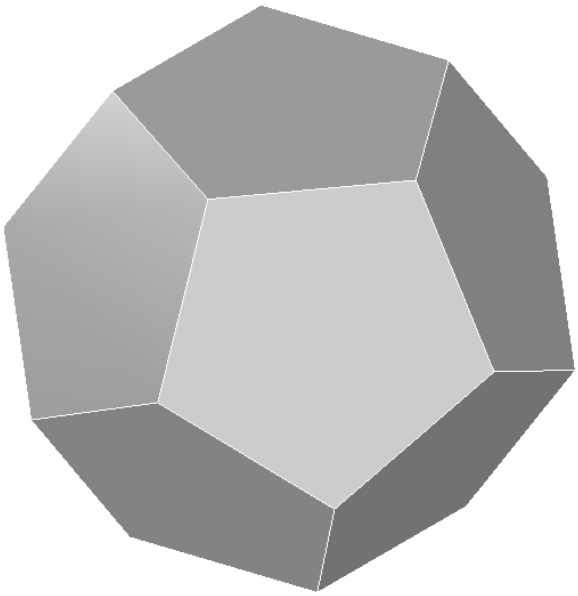


Рис. 3

Полигональный объект с топологией может быть создан из объекта MbMesh, а также сконструирован пользователем самостоятельно. Пример создания такого объекта можно найти в тестовом приложении в файле *test_sample_polymesh.cpp*. Топология необходима для анализа, лечения и редактирования полигональных объектов.

3.17 МНОГОПОТОЧНОСТЬ

Поддержка многопоточности в математическом ядре подразумевает:

- Использование многопоточных вычислений в ядре.
- Поддержку многопоточности в пользовательских приложениях.

Математическое ядро реализует механизмы, обеспечивающие потокобезопасное использование интерфейсов ядра в параллельных вычислениях.

Для организации внутренней многопоточности ядро использует технологию **OpenMP**.

Основные многопоточные операции ядра включают в себя следующее (но не ограничиваются данным списком):

- Построение плоских проекций
- Расчёт полигональных сеток
- Расчёт массо-центровочных характеристик

- Операции конвертеров

Если реализация интерфейса использует параллельные вычисления, то соответствующая информация содержится в комментариях к интерфейсу.

Для обеспечения потокобезопасности использования интерфейсов ядра в многопоточных приложениях используются объекты синхронизации на базе системных механизмов.

3.17.1 Потокобезопасность объектов ядра

Потокобезопасность объектов ядра реализуется специальным механизмом – многопоточным кэшированием, которое обеспечивает потокобезопасный доступ к данным объекта ядра и делает возможным эффективное распараллеливание вычислений в случаях, когда объект обрабатывается одновременно в разных потоках.

Каждый поток работает со своей копией кэшированных данных, что предотвращает конкуренцию за данные между потоками. Управляет многопоточными кэшами менеджер кэшей, который отвечает за создание, хранение и выдачу кэшированных данных объекта для текущего потока.

Важно, что многопоточное кэширование эффективно работает как при многопоточных вычислениях, так и при последовательном выполнении вычислений. Несомненным плюсом является и то, что переход к использованию многопоточных кэшей требует минимальной переработки кода. При этом надо учитывать накладные расходы на поддержку многопоточных кэшей:

- Несколько увеличивается использование памяти.
- При отсутствии параллельности могут присутствовать незначительные дополнительные затраты времени на получение кэшированных данных.

Механизм многопоточных кэшей может быть включен или отключен при переключении режима многопоточности ядра.

3.17.2 Реализация многопоточных кэшей

Базовый класс для кэшированных данных и класс менеджера кэшей определены в файле *tool_multithreading.h*.

Менеджер кэшей CacheManager

Менеджер кэшей определен как шаблонный класс **CacheManager** для класса кэшированных данных и содержит члены:

- **longTerm** – данные главного потока; используются при последовательном выполнении и для инициализации данных потоков при параллельном выполнении.
- **tcache** – список кэшей с данными, которые используются при параллельном выполнении. Каждый поток по идентификатору threadKey использует только свою копию данных. Для многопоточной обработки зависимых (имеющих общие данные) объектов должен использоваться режим многопоточных вычислений mtm_Items.
- **lock** – блокировка менеджера кэшей, которая используется при изменении списка кэшей и изменении кэша главного потока.

Шаблонный класс **CacheManager<class T>** реализует методы:

- **T * operator()** – возвращает указатель на кэш (данные) текущего потока. Всегда возвращает ненулевое значение.

- **void Reset(bool resetLongTerm)** – аннулирует данные кэшей. Если resetLongTerm == true, также удаляет кэш главного потока.
- **void CleanAll(bool doPostproc, bool force)** – удаляет все кэши и отписывается от сборки мусора. Должна вызываться в последовательном участке кода.
- **T * LongTerm()** – возвращает указатель на кэш (данные) главного потока. Всегда возвращает ненулевое значение. Все операции с кэшем главного потока должны быть защищены блокировкой кэша.
- CommonMutex *** GetLock()** – возвращает указатель на блокировку для операций с кэшем главного потока с учетом того, исполняется ли код параллельно. Может возвращать нулевое значение (удобно для использования с ScopedLock).
- CommonMutex *** GetLockHard()** – возвращает указатель на блокировку для операций с кэшем главного потока. Всегда возвращает ненулевое значение.
- **bool ResetCacheData()** – функция очистки, используемая сборщиком мусора.

Алгоритм работы менеджера кэшей зависит от режима многопоточности ядра MbeMultithreadedMode.

Если определен режим многопоточных вычислений не ниже mtm_SafeItems, менеджер кэшей инициирует использование выделенных кэшей для каждого потока. При этом менеджер кэшей по идентификатору потока выбирает и выдает потоку его копию данных.

При переходе к последовательной работе менеджер вызывает функцию **Postprocess()** для пост-обработки кэшей. Указанная функция итерируется по кэшам, использованным при параллельных вычислениях, и вызывает функцию longTerm.MergeWith(AuxiliaryData *) с данными каждого кэша в качестве параметра. После завершения работы функции **Postprocess()** многопоточные кэши удаляются. Далее на запрос кэша менеджер кэшей выдает только кэш главного потока (**longTerm**).

Класс CacheManager является производным от класса CacheCleaner, что позволяет запускать чистку кэшей по требованию.

Базовый класс кэшированных данных AuxiliaryData

Класс кэшированных данных должен наследоваться от базового класса **AuxiliaryData** и содержать определения конструктора по умолчанию и конструктора копирования.

Кроме того, если требуется пост-обработка кэшей после выхода из параллельных вычислений, наследник класса AuxiliaryData должен переопределить метод **void MergeWith(AuxiliaryData *)**, который вызывается менеджером кэшей для данных основного потока с данными каждого из многопоточных кэшей в качестве параметра. Реализация **MergeWith()** по умолчанию не делает никакой пост-обработки.

Класс объекта ядра, использующего кэширование, должен определять кэшированные данные как класс, наследованный от AuxiliaryData, и включать в себя экземпляр CacheManager как член класса.

При многопоточной обработке зависимых (имеющих общие данные) объектов должен использоваться режим многопоточных вычислений не ниже mtm_SafeItems.

3.17.3 Сборка мусора в менеджере кэшей

Менеджер кэшей CacheManager удаляет ненужные кэши при первом вызове из последовательного кода.

Кроме того, при выходе из параллельного региона автоматически запускается сборка мусора в кэшах.

Математическое ядро предоставляет интерфейс для очистки кэшированных данных – класс MbGarbageCollection.

Класс MbGarbageCollection – это сборщик мусора в объектах типа CacheCleaner, хранящих кэшированные данные. По требованию он очищает кэши в зарегистрированных объектах.

Для того, чтобы в объекте типа CacheCleaner стала возможна сборка мусора, он должен реализовать метод ResetCacheData() для очистки кэшей и подписаться на сборку в классе MbGarbageCollection.

Класс CacheManager является производным от класса CacheCleaner, что позволяет запускать чистку кэшей в менеджере автоматически. Для этого при создании кэша для потока менеджер кэшей регистрируется в MbGarbageCollection.

Базовый класс объектов, требующих сборки мусора

Для того, чтобы подписаться на сборку мусора, класс должен наследоваться от класса CacheCleaner и реализовать метод **ResetCacheData** для удаления кэшированных данных.

CacheCleaner – это базовый класс для объектов, требующих сборки мусора. Он реализует методы:

- **void SubcribeOnCleaning()** – подписаться на сборку мусора.
- **void UnsubcribeOnCleaning()** – отписаться от сборки мусора.

Класс CacheCleaner также предоставляет метод:

- **bool ResetCacheData()** – очистить кэшированные данные. Должен возвращать true, если кэшированные данные были очищены и объект отписан от сборки мусора.

Класс для сборки мусора MbGarbageCollection

Класс MbGarbageCollection – это сборщик мусора в объектах типа CacheCleaner, хранящих кэшированные данные. По требованию он очищает кэши в зарегистрированных объектах, вызывая метод ResetCacheData для каждого объекта.

Класс MbGarbageCollection реализует методы:

- **void Subscribe(CacheCleaner * obj)** – подписать объект на сборку мусора.
- **void Unsubscribe(CacheCleaner * obj)** – отписать объект от сборки мусора.
- **static bool Run(bool force = false)** – выполнить сборку мусора. Должна вызываться в последовательном участке кода. При вызове в параллельном регионе ничего не делает. Если force == false, то инициируется сборка мусора в кэшах, созданных для потоков, которые уже завершены, если force == true, то инициируется принудительная сборка мусора во всех кэшах.
- **static void Enable(bool allow = true)** – активировать/деактивировать сбор данных для проведения сборки мусора. По умолчанию сбор данных активирован.

3.17.4 Режимы многопоточности ядра

Режим многопоточности ядра управляет распараллеливанием операций ядра и механизмами потокобезопасности ядра. Он включает или отключает параллельность вычислений в ядре и включает или отключает механизмы, обеспечивающие потокобезопасность объектов и операций ядра (что важно при организации многопоточных вычислений в пользовательском приложении).

Режим многопоточности ядра определен как перечисление **MbeMultithreadedMode** в файле *tool_multithreading.h*.

Ядро может работать в следующих режимах:

- **mtm_Off** – многопоточность ядра отключена. В этом режиме все операции выполняются последовательно. Механизм, обеспечивающий потокобезопасность объектов ядра, отключен.
- **mtm_Standard** – в стандартном режиме многопоточности работает ограниченное распараллеливание операций ядра, а именно распараллеливаются только операции, обрабатывающие независимые данные. Механизм потокобезопасности объектов ядра отключен.
- **mtm_SafeItems** – режим потокобезопасности объектов ядра. В этом режиме включается механизм многопоточного кэширования, но по-прежнему работает ограниченное распараллеливание операций ядра. Данный режим нужен для поддержки многопоточных вычислений в пользовательских приложениях.
- **mtm_Items** – режим многопоточности объектов. В этом режиме включено полное распараллеливание операций ядра, то есть распараллеливаются операции, обрабатывающие как независимые, так и зависимые данные. Данный режим также может использоваться для поддержки многопоточных вычислений в пользовательских приложениях.
- **mtm_Max** – включена максимальная многопоточность ядра. В настоящее время фактически совпадает с режимом **mtm_Items**.

По умолчанию в ядре установлен режим **mtm_Max**.

Ядро предоставляет интерфейсы для динамического изменения режима многопоточности ядра.

Интерфейсы для динамического изменения режима многопоточности ядра объявлены в файле *mb_variables.h*.

Для переключения между режимами многопоточности предназначены статические методы класса **Math**:

- **bool Multithreaded()** – проверяет, включен ли режим mtm_Standard (работает ли распараллеливание).
- **void SetMultithreaded(bool b)** – если b == false, устанавливает режим mtm_Off (отключает многопоточность). Если b == true, устанавливает режим mtm_Standard (включает ограниченную многопоточность).
- **MbeMultithreadedMode MultithreadedMode()** – возвращает текущий режим многопоточности.
- **bool CheckMultithreadedMode(MbeMultithreadedMode mode)** – проверяет, что текущий режим многопоточности не ниже **mode**.
- **void SetMultithreadedMode(MbeMultithreadedMode mode)** – устанавливает режим многопоточности **mode**.

3.17.5 Объекты синхронизации

Объекты синхронизации определены в файле *tool_mutex.h*.

Блокировки

По умолчанию ядро использует классы блокировок, реализованные на основе системных механизмов синхронизации, что позволяет безопасно использовать любые механизмы распараллеливания (в том числе отличные от OpenMP) в пользовательском приложении.

Но ядро может быть переключено на использование блокировок на базе блокировок OpenMP.

На платформе Windows системные блокировки по умолчанию реализуются на базе критической секции. На платформе Linux системные блокировки реализуются на базе pthread_mutex_t.

Реализация классов блокировок управляется переменной **C3D_NATIVE_LOCK**. Переопределять переменную **C3D_NATIVE_LOCK** в пользовательском приложении запрещено.

Классы **CommonMutex** и **CommonRecursiveMutex** определяют блокировку и рекурсивную блокировку. Они реализуют методы lock() и unlock().

Классы **ScopedLock** и **ScopedRecursiveLock** определяют блокировку и рекурсивную блокировку в области видимости. Они могут принимать нулевой указатель на мьютекс. Блокировка происходит, если указатель на мьютекс ненулевой и код выполняется параллельно.

Базовые объекты синхронизации

Класс **MbSyncItem** – базовый объект синхронизации с отложенной инициализацией, реализующий методы **Lock()** и **Unlock()**. Создает блокировку (мьютекс) при необходимости.

Класс **MbNestSyncItem** – базовый объект синхронизации с отложенной инициализацией, поддерживающий множественные блокировки. Создает блокировку (мьютекс) при необходимости. Реализует методы **Lock()** и **Unlock()**.

Класс **MbPersistentSyncItem** – базовый объект синхронизации, реализующий методы **Lock()** и **Unlock()**. Всегда создает блокировку.

Класс **MbPersistentNestSyncItem** – базовый объект синхронизации, поддерживающий множественные блокировки. Всегда создает блокировку. Реализует методы **Lock()** и **Unlock()**.

Объекты ядра, которым нужны средства синхронизации, являются наследниками одного из этих классов.

3.17.6 Поддержка многопоточности в пользовательском приложении

Чтобы использовать интерфейсы ядра в нескольких потоках, для ядра должен быть установлен режим многопоточных вычислений не ниже mtm_SafeItems. По умолчанию в ядре установлен максимальный режим многопоточности mtm_Max.

Если пользовательское приложение использует интерфейсы ядра в нескольких потоках, оно обязано сообщать ядру о входе в каждый параллельный регион и о выходе из него. Для этого могут быть использованы класс ParallelRegionGuard (защитник параллельного региона в области видимости) или парные функции EnterParallelRegion и ExitParallelRegion.

Защита параллельного кода в пользовательском приложении

Класс ParallelRegionGuard сообщает ядру, что код в области видимости выполняется параллельно. Он должен использоваться для защиты параллельного кода.

Вместо класса **ParallelRegionGuard** следующая пара функций также может быть использована для нотификации ядра о границах параллельного региона кода:

- Функция **EnterParallelRegion** сообщает ядру о входе в параллельный блок кода.
- Функция **ExitParallelRegion** сообщает ядру о выходе из параллельного блока кода.

Использование ParallelRegionGuard или функций **EnterParallelRegion** и **ExitParallelRegion** является обязательным, если интерфейсы ядра используются в нескольких потоках.

Прежде чем начать использовать параллельные вычисления, нужно убедиться, что в ядре установлен режим многопоточности не ниже mtm_SafeItems (по умолчанию в ядре установлен максимальный режим многопоточности mtm_Max).

Примеры уведомления ядра о его использовании в параллельных вычислениях

ParallelRegionGuard должен вызываться один раз перед началом работы в нескольких потоках. Например:

```
// ... Последовательный код ...

if( Math::CheckMultithreadedMode( mtm_Items ) ) {
  ParallelRegionGuard guard; // Работает в области видимости

  std::thread t1( func1 );
  std::thread t2( func2 );

  t1.join();
  t2.join();
}
```

```
// ... Последовательный код ...
```

То же самое можно сделать, используя пару функций **EnterParallelRegion()** и **ExitParallelRegion()** вместо **ParallelRegionGuard**:

```
// ... Последовательный код ...

if( Math::CheckMultithreadedMode( mtm_Items ) ) {
  EnterParallelRegion();

  std::thread t1( func1 );
  std::thread t2( func2 );

  t1.join();
  t2.join();

  ExitParallelRegion();
}

// ... Последовательный код ...
```

Интерфейсы ParallelRegionGuard, EnterParallelRegion и ExitParallelRegion должны использоваться, чтобы нотифицировать ядро о распараллеливании в пользовательском приложении. Эти нотификации нужны, чтобы обеспечить правильную работу механизмов потокобезопасности ядра.

Краткая справка по организации параллельных вычислений с использованием интерфейсов ядра C3D

1. Перед запуском параллельных вычислений приложение должно проверить, что установленный режим многопоточности ядра позволяет использовать параллельные вычисления. Чтобы использовать интерфейсы ядра в нескольких потоках, в ядре должен быть установлен режим многопоточных вычислений не ниже **mtm_SafeItems**.

   Пример проверки режима ядра:

   bool useParallel = Math::CheckMultithreadedMode(mtm_SafeItems);

2. При использовании параллельных вычислений пользовательское приложение обязано нотифицировать ядро о входе в параллельный регион и выходе из него.

   Если используется параллельный цикл, то перед его началом нужно вызвать EnterParallelRegion, а после окончания – ExitParallelRegion.

   Пример:

```
if( Math::CheckMultithreadedMode( mtm_Items ) ) {
  EnterParallelRegion();

  #pragma omp parallel for
  for( ptrdiff_t i = 0; i < count; ++i ) {
    // Тело цикла
  }

  ExitParallelRegion();
}
```

   Если интерфейсы ядра просто вызываются из разных потоков, то в каждом потоке перед вызовами интерфейсов ядра должен быть вызов EnterParallelRegion, а после окончания использования ядра – вызов ExitParallelRegion.

   Пример:

```
// Поток 1:
EnterParallelRegion();
// Вызовы интерфейсов ядра
ExitParallelRegion();

// Поток 2:
EnterParallelRegion();
// Вызовы интерфейсов ядра
ExitParallelRegion();
```

   После первого вызова EnterParallelRegion ядро начнет использовать многопоточные кэши, а после последнего вызова ExitParallelRegion вернется в базовый режим одного потока.

3. Не рекомендуется вызывать функции EnterParallelRegion и ExitParallelRegion без необходимости (в отсутствие многопоточности), так как использование многопоточных кэшей влечет за собой дополнительные затраты времени на получение кэшированных данных.

3.18 ФОРМАТ C3D

Геометрическая модель C3D может быть сохранена в компактном виде в буфер памяти или в файл. Существуют две разновидности формата хранения геометрической модели C3D – компактный и расширенный.

Данный раздел описывает форматы хранения геометрической модели и интерфейсы, предоставляемые геометрическим ядром для записи и чтения данных геометрической модели.

3.18.1 Формат хранения геометрической модели

Понятия и термины

Напомним, что **сериализация** – это процесс преобразования объекта (класса) в последовательность байтов с целью сохранения в буфере памяти (файле). **Десериализация** – обратный процесс восстановления объекта из последовательности байтов.

Потоковые операции – это операции записи и чтения модели и ее объектов (работа с потоками записи и чтения).

Объекты, которые могут сериализоваться (представлять себя в виде последовательности байтов), будем называть **потоковыми**.

Кластером будем называть непрерывный участок памяти (последовательность байтов), а **файловым пространством** – набор кластеров. Файловое пространство имеет одну точку записи (чтения) – позицию в текущем кластере.

Принципы записи геометрической модели

Геометрическая модель C3D может записываться в сериализованном виде (в виде последовательности байтов) в набор файловых пространств.

Все основные объекты геометрической модели являются потоковыми, то есть умеют писать и читать себя в файловое пространство. Каждый объект модели записывается, начиная с последнего незаполненного кластера текущего файлового пространства.

Запись геометрической модели начинается с объекта MbModel.

Каждый объект модели сначала записывает данные своего родительского объекта (если он существует), затем собственные данные и объекты, которые он содержит или на которые ссылается.

После завершения записи объекта MbModel геометрическая модель полностью сериализована в набор файловых пространств.

Далее вызовом специальной функции формируется единый последовательный буфер в памяти, куда записывается специальный заголовок и последовательно копируются данные всех кластеров файловых пространств.

Сформированный таким образом буфер памяти, содержащий данные геометрической модели в виде последовательности байтов, может быть передан куда-либо для последующего восстановления модели или записан в файл.

Компактный формат C3D

Компактный формат обеспечивает компактную и быструю запись данных геометрической модели в буфер памяти (файл) и обеспечивает чтение модели из буфера памяти (файла) целиком.

Особенностями компактного формата являются:

- Минимум накладных расходов при записи и чтении модели.
- Компактность хранения данных.
- Возможность чтения модели целиком.
- Информация о структуре модели и отдельных объектах доступна только после прочтения всей модели.
- Чтение объектов модели в произвольном порядке и импорт отдельных объектов невозможны.

Компактный формат помещает все данные модели в одно файловое пространство, то есть подразумевает наличие только одной точки записи или чтения данных – текущий кластер данного файлового пространства.

Рис. 1 показывает структуру сериализованных данных геометрической модели перед сохранением в последовательный буфер памяти при использовании компактного формата:

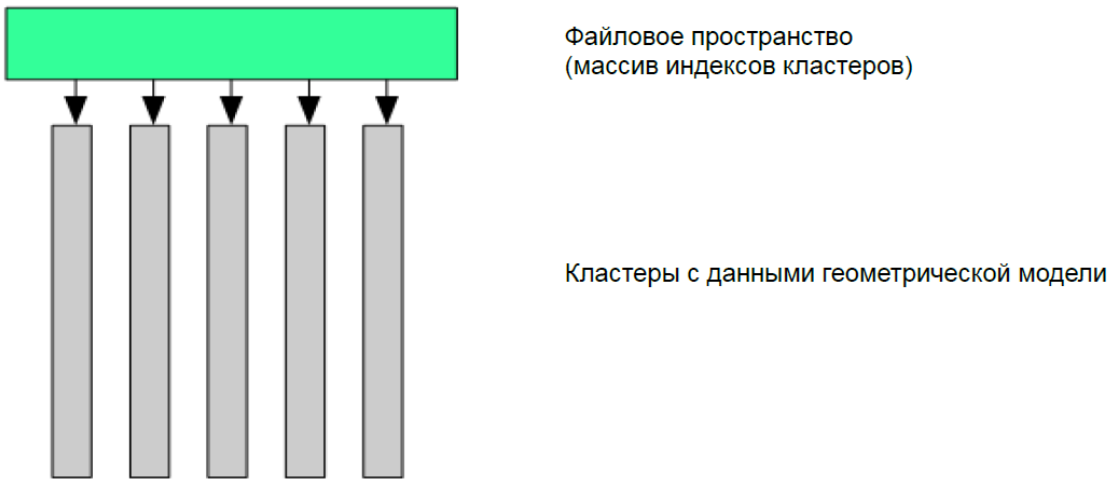


Рис. 1

Во время записи модели, если геометрический объект содержит внутри себя другой объект или ссылается на другой объект, то последний записывается внутри ссылающегося на него объекта.

Объект, на который ссылаются несколько объектов, объявляется *регистрируемым*. Такой объект во время первой своей записи регистрируется в специальной таблице регистрируемых объектов. Все последующие записываемые объекты, ссылающиеся на данный регистрируемый объект, содержат только ссылку на запись в таблице.

Рис. 2 показывает структуру хранения данных геометрической модели в последовательном буфере памяти:

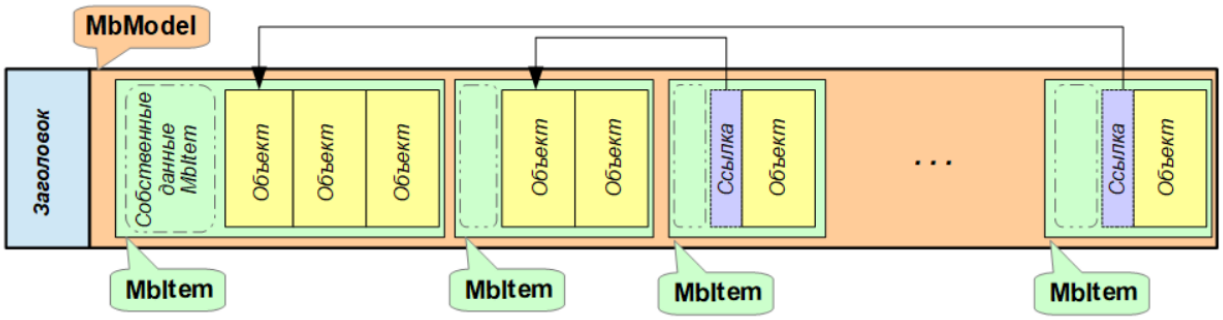


Рис. 2

Расширенный формат C3D

Расширенный формат обеспечивает независимое хранение объектов геометрической модели. Помимо данных объектов модели, он хранит информацию о позициях их хранения и краткое оглавление модели, что позволяет получить сведения о структуре модели без ее полного чтения, а также дает возможность выборочного чтения объектов модели.

Особенности расширенного формата:

- Независимое хранение объектов, которое обеспечивает возможность выборочного чтения объектов модели.
- Наличие оглавления модели с данными об объектах модели и ссылками на позиции их хранения.
- Некоторое увеличение размера хранимых данных.

Расширенный формат записывает объекты модели в несколько файловых пространств, то есть обеспечивает несколько точек записи (чтения) данных – текущий кластер в каждом файловом пространстве.

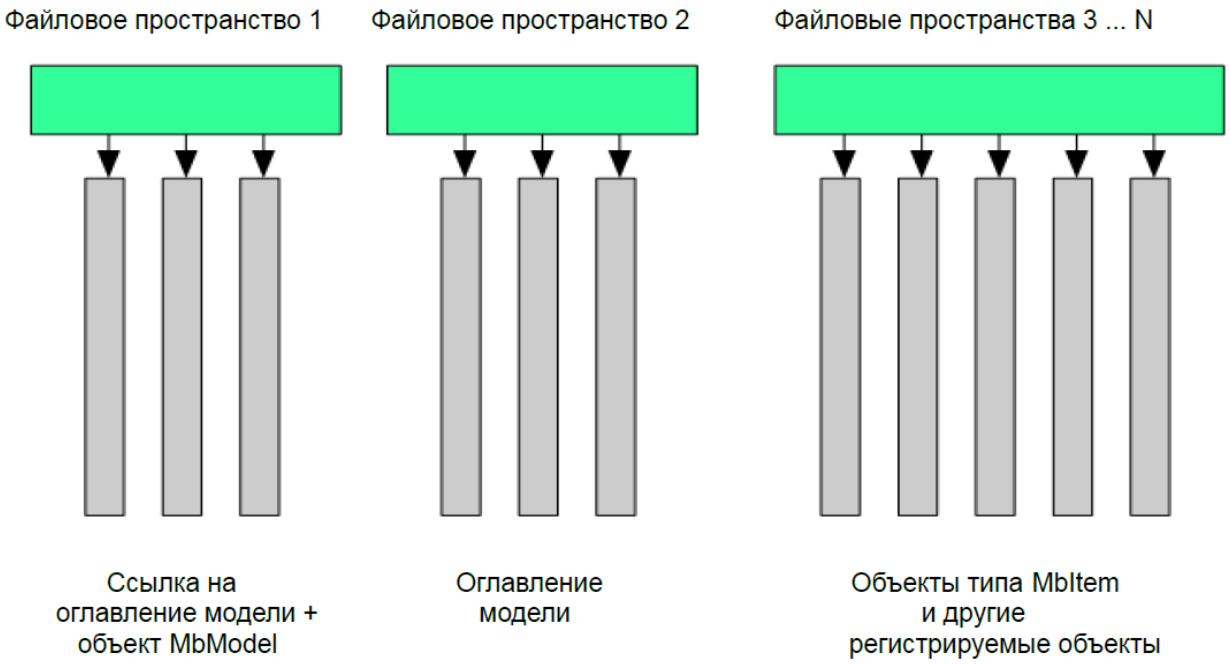


Рис. 3

Рис. 3 показывает структуру сериализованных данных геометрической модели перед сохранением в последовательный буфер памяти при использовании расширенного формата.

Объекты каждого уровня модели записываются в отдельное файловое пространство (количество используемых файловых пространств определяется глубиной вложенности объектов модели).

Рис. 4 показывает структуру хранения данных геометрической модели в последовательном буфере памяти при использовании расширенного формата:

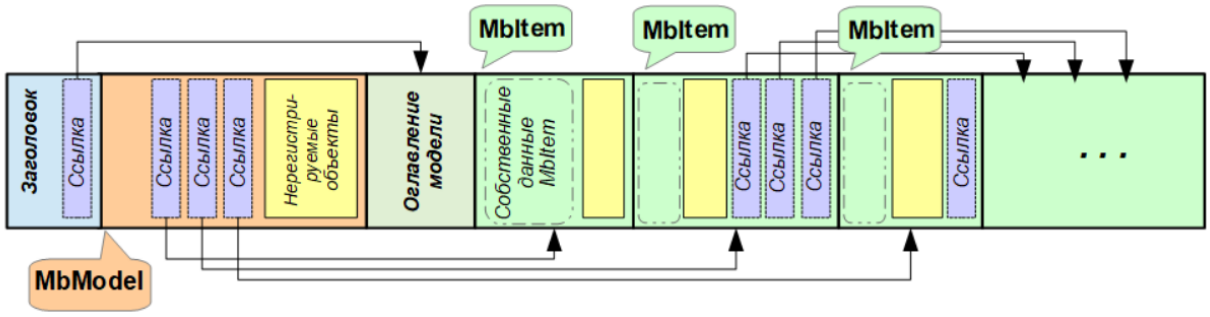


Рис. 4

Оглавление модели в расширенном формате

Расширенный формат реализует генерацию дерева геометрической модели и его запись/чтение в буфере памяти (файле).

Дерево создается для всех объектов геометрической модели типа MbItem и записывается

как *оглавление модели* в отдельное файловое пространство, ссылка на которое добавляется в заголовок файла.

Оглавление модели, записываемое в файл, содержит:

1. Данные узлов дерева модели.

Каждый узел дерева хранит информацию для одного объекта геометрической модели. В узле хранятся следующие данные:

- Уникальный ID узла в дереве модели.
- Тип объекта (MbeSpaceType).
- Имя объекта (SimpleName).
- Габарит объекта.
- Информация об атрибутах объекта.
- Список непосредственных потомков узла.
- Позиция записи/чтения объекта в файле.

2. Данные о корнях дерева модели.

Оглавление модели может быть прочитано отдельно от всей модели, что позволяет получить сведения о структуре модели и основных ее объектах без полной загрузки модели. Используя данные оглавления модели, можно выбрать и прочитать отдельные объекты модели.

Работа с исполнениями

Расширенный формат C3D предоставляет возможность работы с исполнениями (вариантами реализации модели):

- Модель с исполнениями может быть сохранена (сериализована) в файл в расширенном формате с помощью класса writer_ex и прочитана с помощью класса reader_ex.
- Прочитав дерево модели из файла с исполнениями, можно посмотреть иерархию исполнений, выбрать исполнение и прочитать его содержимое как отдельную модель.
- При чтении файла с исполнениями без использования дерева модели (например, с помощью прямого вызова функции ReadModelItems()) будет прочитано первое по порядку (умолчательное) исполнение.

Модель с исполнениями представляет собой объект MbModel, содержащий набор сборок (объектов MbAssembly), каждая из которых представляет собой одно исполнение (содержит объекты одного исполнения) и имеет атрибут типа at_Embodiment. Первая по порядку сборка является умолчательным исполнением.

Сборка считается исполнением и обрабатывается как исполнение, только если она находится непосредственно в корне модели и имеет атрибут типа at_Embodiment. Объекты MbAssembly, содержащиеся внутри какого-либо исполнения (глубже корня в дереве модели), обрабатываются как обычные сборки.

Атрибут типа at_Embodiment служит индикатором исполнения. Он должен содержать имя (SimpleName) текущего исполнения и имя (SimpleName) родительского исполнения (или **UNDEFINED_SNAME**, если родителя нет).

Чтобы обеспечить возможность просмотра иерархии исполнений и выборочного чтения, модель с исполнениями должна записываться с помощью класса writer_ex. После этого модель с исполнениями должна читаться с помощью класса reader_ex.

При чтении файла с исполнениями может быть прочитано только одно исполнение.

Если читать файл с исполнениями обычным способом, то есть без использования дерева модели (например, с помощью прямого вызова функции ReadModelItems()), то будет прочитано первое по порядку (умолчательное) исполнение.

Чтобы прочитать любое другое исполнение, надо сначала прочитать дерево модели. Затем, используя дерево модели, можно посмотреть иерархию исполнений, выбрать исполнение и прочитать его содержимое как отдельную модель.

3.18.2 Чтение и запись потоковых объектов

Классы для чтения и записи потоковых объектов модели (потоки для чтения и записи) определены в файле **tape.h**. Они наследуют от базового класса tape.

Базовый класс для потоков чтения и записи

Базовый класс потока для чтения и записи tape содержит ссылки на буфер данных, менеджер потоков чтения (записи) и структуру для регистрации прочитанных (записанных) объектов.

Он предоставляет методы управления буфером данных, регистрацией объектов и методы работы с индикатором прогресса.

Доступ к буферу данных:

- const iobuf_Seq & GetIOBuffer() – доступ к буферу данных.
- iobuf_Seq & GetIOBuffer() – доступ к буферу данных.
- bool IsOwnBuffer() – владеем ли буфером?
- void SetOwnBuffer(bool) – установить флаг владения буфером.

Управление режимом работы буфера данных:

- uint8 mode() – узнать режим работы буфера.
- void setMode(uint8) – установить режим работы буфера.
- void clearState(state) – убрать состояние буфера.
- void setState(state) – добавить состояние буфера.

Опрос состояния буфера данных:

- int fresh() – свежий ли буфер?
- bool good() – корректно ли состояние буфера?
- uint8 eof() – достигнут ли конец файла?
- uint32 state() – получить флаг состояния буфера.
- io::pos tell() – получить текущую позицию в потоке.

Управление версиями:

- void SetVersionsByStorage() – установить текущую версию равной версии хранилища.
- VERSION MathVersion() – вернуть главную версию (версию математического ядра).
- VERSION AppVersion(size_t ind) – вернуть дополнительную версию (версию конечного приложения).

- const VersionContainer & GetVersionsContainer() – получить доступ к контейнеру версий.
- void SetVersionsContainer(const VersionContainer &) – установить версию открытого файла.
- VERSION SetStorageVersion(VERSION) – установить версию хранилища.

Управление регистрацией объектов:

- void registrate(const TapeBase *) – зарегистрировать указатель.
- void unregistrate(const TapeBase *) – отменить регистрацию указателя.
- bool exist(const TapeBase *) – есть ли зарегистрированный объект?
- void flushRegister() – очистить массив регистрации.
- size_t RegisteredCount() – получить количество зарегистрированных объектов.
- size_t GetMaxRegisteredCount() – получить максимально возможное количество объектов для регистрации.
- void ReserveRegistered(size_t n) – зарезервировать память под n объектов.
- uint8 GetIndexType(size_t index) – получить тип индекса.

Работа с индикатором прогресса:

- void InitProgress(IProgressIndicator * pr) – инициализировать индикатор прогресса.
- void InitProgress(ProgressBarWrapper & pr) – инициализировать индикатор прогресса.
- void ResetProgress() – освободить текущий индикатор прогресса, установить родительский индикатор прогресса, если он есть.
- ProgressBarWrapper * GetProgress() – получить индикатор прогресса.

Запись модели

Класс writer

Класс writer обеспечивает запись модели в буфер данных (файловое пространство) в компактном формате. Он наследует от базового класса tape.

Класс writer реализует:

1. Статические функции для создания экземпляра класса:

- writer_ptr CreateWriter(std::unique_ptr< iobuf_Seq >, uint16) – создать экземпляр класса для последовательного буфера памяти типа iobuf_Seq.
- writer_ptr CreateMemWriter(membuf &, uint8) – создать экземпляр класса для буфера памяти типа membuf.

2. Методы для записи в буфер объектов различных типов:

- void writeObject(const TapeBase *) – записать объект.
- void writeObjectPointer(const TapeBase *) – записать указатель на объект.
- void writeByte(uint8) – записать байт в буфер.
- void writeBytes(const void *, size_t) – записать последовательность байтов в буфер.
- void writeUInt64(const uint64 &) – записать беззнаковое 64-разрядное целое число. Возвращает количество записанных байт.

- void writeInt64(const int64 &) – записать 64-разрядное целое число. Возвращает количество записанных байт.
- writer & _writeChar(const char *) – записать CHAR-строку в поток (кодировка ANSI, русская локаль).
- writer & _writeWchar(const TCHAR *) – записать WCHAR-строку в поток (в потоке хранится как UTF-16).
- writer & _writeWcharT(const wchar_t *) – записать WCHAR-строку в поток (в потоке хранится как UTF-16).
- size_t _lenWchar(const TCHAR *) – получить длину записи WCHAR-строки в поток (в потоке хранится как UTF-16).

3. Методы для записи дерева модели и доступа к нему (в данном классе определяются, но не поддерживаются, содержат пустую реализацию):

- void WriteModelCatalog() – записать дерево модели.
- const c3d:: IModelTree * GetModelTree() const – получить указатель на дерево модели.

Класс writer_ex

Класс writer_ex обеспечивает запись модели в буфер данных (в набор файловых пространств) в расширенном формате. Он наследует от базового класса writer.

Класс writer_ex реализует:

1. Статические функции для создания экземпляра класса:

- std::unique_ptr< writer_ex > CreateWriterEx(std::unique_ptr< iobuf_Seq >, uint16) – создать экземпляр класса для последовательного буфера памяти типа iobuf_Seq.
- std::unique_ptr< writer_ex > CreateMemWriterEx(membuf &, uint8) – создать экземпляр класса для буфера памяти типа membuf.

2. Методы для записи дерева модели и доступа к нему:

- void WriteModelCatalog() – записать дерево модели.
- const c3d:: IModelTree * GetModelTree() const – получить указатель на дерево модели.

Чтение модели

Класс reader

Класс reader обеспечивает чтение модели из буфера данных (файлового пространства), записанного в компактном формате. Он наследует от базового класса tape.

Класс **reader** реализует:

1. Статические функции для создания экземпляра класса:

- reader_ptr CreateReader(std::unique_ptr< iobuf_Seq >, uint16) – создать экземпляр класса для последовательного буфера.
- reader_ptr CreateMemReader(membuf &, uint8) – создать экземпляр класса для буфера в памяти.

2. Методы для чтения из буфера объектов различных типов:

- TapeBase * readObject(TapeBase *) – прочитать объект.
- TapeBase * readObjectPointer() – прочитать указатель на объект.
- bool readUInt64(uint64 &) – прочитать беззнаковое 64-разрядное целое число.

- bool readInt64(int64 &) – прочитать 64-разрядное целое число.
- int readByte() – прочитать байт из буфера.
- bool readBytes(void *, size_t) – прочитать последовательность байтов из буфера.

3. Методы управления чтением дерева модели и частичным чтением объектов модели (в данном классе определяются, но не поддерживаются, содержат пустую реализацию):

- void ReadObjectCatalog() – прочитать каталог объектов.
- membuf * ReadObjectByPosition(const ClusterReference &) – прочитать объект по позиции чтения.
- bool SetReadPosition(ClusterReference &) – установить позицию чтения.
- const c3d:: IModelTree * GetModelTree() const – получить указатель на дерево модели.
- bool IsFullRead() – получить признак полного чтения текущего объекта (читается ли текущий объект целиком).
- void SetFullRead(bool) – установить признак полного чтения текущего объекта (читать ли текущий объект целиком).

4. Метод получения ошибок чтения:

- uint32 GetLastError() – получить ошибки чтения.

5. Методы работы с индикатором прогресса:

- void InitProgress(IProgressIndicator *) – инициализировать индикатор прогресса.
- void InitProgress(ProgressBarWrapper &) – инициализировать индикатор прогресса.

Класс reader_ex

Класс reader_ex обеспечивает чтение модели из буфера данных (набора файловых пространств), записанного в компактном или расширенном формате. Может читать из нескольких файловых пространств по заданным позициям. Наследует от базового класса reader.

Класс reader_ex реализует:

1. Статические функции для создания экземпляра класса:

- std::unique_ptr< reader_ex > CreateReaderEx(std::unique_ptr< iobuf_Seq >, uint16) – создать экземпляр класса для последовательного буфера.
- std::unique_ptr< reader_ex > CreateMemReaderEx(membuf &, uint8) – создать экземпляр класса для буфера в памяти.

2. Методы управления чтением дерева модели и частичным чтением объектов модели:

- void ReadObjectCatalog() – прочитать каталог объектов.
- TapeBase * ReadObjectByPosition(const ClusterReference &) – прочитать объект по позиции чтения.
- bool SetReadPosition(ClusterReference &) – установить позицию чтения.
- const c3d:: IModelTree * GetModelTree() const – получить указатель на дерево модели.
- bool IsFullRead() – получить признак полного чтения текущего объекта (читается ли текущий объект целиком).

- void SetFullRead(bool) – установить признак полного чтения текущего объекта (читать ли текущий объект целиком).

3. Метод получения ошибок чтения:

- uint32 GetLastError() – получить ошибки чтения.

Индикатор прогресса в области видимости для классов чтения

Класс **ScopedReadProgress** реализует индикатор прогресса в области видимости для классов чтения модели. Он создает дочерний индикатор прогресса для индикатора прогресса в заданном экземпляре класса reader или reader_ex. При выходе из области видимости текущий индикатор прогресса освобождается и устанавливается родительский индикатор прогресса.

Класс определяет оператор доступа к текущему индикатору прогресса:

- ProgressBarWrapper * operator()().

Чтение и запись модели

Класс rw обеспечивает чтение и запись модели в буфер данных (в набор файловых пространств) в компактном формате. Он наследует от классов reader и writer.

Класс реализует статическую функцию для создания экземпляра класса:

- rw_ptr CreateMemWriter(membuf &, uint8).

Дерево модели

Классы дерева модели определены в файле *io_tree.h*.

Данные узла дерева объявлены как структура MbItemData.

Класс IModelTreeNode определяет интерфейс для узла дерева модели, который может иметь несколько потомков, умеет записываться в поток и читаться из потока.

Класс IModelTreeNode содержит упорядоченные массивы указателей на непосредственных потомков узла и непосредственных предков узла.

Класс IModelTreeNode объявляет методы:

- std::set< const IModelTreeNode * > & GetParents() – доступ к непосредственным предкам узла.
- const std::set< const IModelTreeNode * > & GetParents() const – доступ к непосредственным предкам узла.
- std::set< const IModelTreeNode * > & GetChildren() – доступ к непосредственным потомкам узла.
- const std::set< const IModelTreeNode * > & GetChildren() const – доступ к непосредственным потомкам узла.
- void AddParent(IModelTreeNode * parent) – добавить предка.
- void AddChild(IModelTreeNode * child) – добавить потомка.
- MbItemData & GetData() – доступ к данным узла.
- const MbItemData & GetData() const – доступ к данным узла.
- ClusterReference & GetPosition() – доступ к позиции чтения/записи узла.

- bool PartialRead() – узнать, читать ли только часть узла. При чтении объекта может возникнуть необходимость чтения некоторых данных его родителя. В этом случае объект родителя читается частично и имеет соответствующий флаг.
- void SetPartialRead(bool partial) – установить признак частичного или полного чтения узла.
- writer & operator >>(writer &) – записать узел в поток.
- reader & operator <<(reader &) – прочитать узел из потока.

Класс IModelTree определяет интерфейс обобщенного дерева модели, которое используется для формирования, записи и чтения оглавления модели в буфере памяти.

Класс определяет типы callback-функций:

- FilterNodesFunc – тип функции для выбора узлов дерева по фильтрам.
- NodeToAddFunc – тип функции для определения, нужно ли добавлять объект в дерево модели, и заполнения данных узла дерева.

Класс IModelTree определяет перечисление TreeType для типов дерева:

- mtt_Model – дерево содержит стандартную модель (объект MbModel).
- mtt_Embodiment – дерево содержит исполнения.

Класс объявляет методы:

- TreeType GetType() const – выдать тип дерева.
- void SetType(TreeType type) – установить тип дерева.
- void AddNode(const TapeBase *****, const ClusterReference &) – добавить узел в дерево.
- void CloseNode(const IModelTreeNode *) – нотификация об окончании чтения/записи текущего узла (функция должна вызываться по окончании чтения/записи заданного узла дерева).
- std::unique_ptr< const IModelTree > GetFilteredTree(const std::vector< MbItemData > & filters) – построить и выдать дерево из узлов текущего дерева, выбранных по фильтрам. В случае дерева с типом mtt_Embodiment функция работает с первым исполнением.
- std::unique_ptr< const IModelTree > GetFilteredTree(std::vector< const IModelTreeNode * > & nodes) – построить и выдать дерево из заданных узлов. Не применимо к дереву с типом mtt_Embodiment (в этом случае возвращает NULL).
- const IEmbodimentTree * GetEmbodimentsTree() const – выдать указатель на дерево исполнений. Выдает NULL, если не применимо (в случае дерева с типом mtt_Model).
- void SetNodeToAddFunction(NodeToAddFunc callback) – установить функцию для выбора геометрического объекта для добавления в дерево модели и заполнения данных узла дерева.
- void SetFilterFunction(FilterNodesFunc callback) – установить функцию для выбора узлов из дерева модели.
- writer & operator >>(writer &) – записать дерево в поток.
- reader & operator <<(reader &) – прочитать дерево из потока.
- Доступ к корням дерева. Узел дерева может быть рекурсивно вложен (например, объект вставка может содержать сборку, которая содержит другой объект вставка, ссылающийся на эту же сборку):

  - const std::vector< const IModelTreeNode * > & GetRoots() – получить корни дерева.
  - std::vector< const IModelTreeNode * > & GetRoots() – получить корни дерева.
- VERSION GetVersion() – получить версию дерева.
- void SetVersion(VERSION) – задать версию дерева.
- static IModelTree * CreateModelTree() – создать экземпляр дерева.

Класс IEmbodimentNode определяет интерфейс для узла дерева исполнений. Он содержит упорядоченный массив указателей на непосредственных потомков узла в дереве исполнений.

Класс объявляет методы:

- std::unique_ptr< const IModelTree > GetEmbodiment() const – построить дерево модели типа mtt_Model, которое содержится в данном исполнении.
- const MbItemData & GetEmbodimentData() const – доступ к информации об исполнении.
- std::set< const IEmbodimentNode * > & GetChildren() – доступ к непосредственным потомкам узла.
- const std::set< const IEmbodimentNode * > & GetChildren() const – доступ к непосредственным потомкам узла.
- void AddChild(IEmbodimentNode * child) – добавить потомка.

Класс IEmbodimentTree определяет интерфейс для дерева, которое отражает иерархию вариантов реализации модели (исполнений). Каждый узел дерева представляет одно исполнение.

Класс объявляет методы:

- const std::vector< const IEmbodimentNode * > & GetRoots() const – доступ к корням дерева.
- std::vector< const IEmbodimentNode * > & GetRoots() – доступ к корням дерева.

Потоковые объекты

Объекты модели, которые могут сериализоваться (представлять себя в виде последовательности байтов), называются потоковыми. Все классы потоковых объектов модели наследуются от базового класса TapeBase, который определен в файле *tape.h*.

Потоковые объекты могут быть регистрируемыми и нерегистрируемыми. От этого зависит, как они будут записаны в поток. Регистрируемый объект пишется отдельно, и объекты модели, которые ссылаются на него, содержат ссылку. Нерегистрируемый объект пишется внутри объекта, который на него ссылается.

Типы регистрации потоковых объектов

Типы регистрации потоковых объектов определены в перечислении RegistrableRec:

- noRegistrable – нерегистрируемый объект.
- registrable – регистрируемый объект.

Базовый класс потоковых объектов TapeBase

Класс ClassDescriptor определяет упакованное имя потокового класса. Он хранит хэш имени класса и идентификатор приложения.

Классы потоковых объектов модели наследуются от базового класса TapeBase, который определяет следующие методы:

- RegistrableRec GetRegistrable() const – получить тип регистрации потокового класса.
- void SetRegistrable(RegistrableRec regs) – установить состояние регистрации потокового класса.
- ClassDescriptor GetClassDescriptor(const VersionContainer &) – получить дескриптор класса.
- const char * GetPureName(const VersionContainer &) – получить имя класса.
- bool IsFamilyRegistrable() const – принадлежит ли объект к регистрируемому семейству.

Режимы потоковых операций

Режимы потоковых операций описываются перечислением io:: mode_flags в файле *io_buffer.h*:

- in – открыть поток для чтения.
- out – открыть поток для записи.
- trunc – открыть существующий файл и удалить его содержимое.
- speedOnClose – упорядочить поток при закрытии.
- delIfEmpty – удалить файл, если он оказался пустым.
- delOnClose – удалить файл при закрытии.
- recovery – режим восстановления.
- appSpecial – вспомогательный флаг приложения.
- createNew – создать новый файл. Выдается ошибка, если такой файл уже существует.
- createAlways – создать новый файл. Если такой файл уже существует, то он перезаписывается.
- openExisting – открыть существующий файл. Выдается ошибка, если файл не существует.
- openAlways – открыть существующий файл. Если файл не существует, то создается новый.
- truncExisting – открыть файл с удалением его содержимого.

Флаги состояния потока

Флаги состояния потока определены в перечислении state в файле *io_buffer.h*:

- good – все в порядке (никакие биты не выставлены).
- eof – конец файла.
- outOfRead – выход за пределы файла.
- outOfMemory – не получилось выделить необходимую память.
- fail – ошибка операции ввода-вывода.
- badData – неверная структура файла.
- notFound – файл не найден.

- accessViolation – доступ запрещен.
- cantOpenStore – не получилось открыть хранилище.
- cantCreateStore – не получилось создать хранилище.
- badSig – нет подписи или подпись чужая.
- cantReadCatalog – не получилось прочитать каталог хранилища.
- cantWriteCatalog – не получилось записать каталог хранилища.
- cantFind – не получилось найти файл в каталоге.
- cantRead – не получилось прочитать файл.
- cantWrite – не получилось записать файл.
- badClassId – прочитанный идентификатор класса не найден в базе данных.
- doubledClassId – попытка повторной регистрации идентификатора класса.
- verViolation – версия файла старше версии задачи.
- hardFail – ошибка файловой операции.
- closed – буфер закрыт.
- writeProtect – у файла установлен атрибут «Только чтение».
- cantWriteObject – не могу записать объект (версия потока младше версии появления нового класса объектов).
- underflow64to32 – не могу прочитать файл с 64-битными данными в 32-битной задаче (потеря старшего слова uint32 при чтении uint64 в 32-битной задаче).
- encrypted – файл защищен или закодирован.
- skippedUnknown – файл в расширенном формате прочитан частично (неизвестные объекты пропущены).
- readAborted – чтение файла прервано пользователем.
- allMask – все ошибки.

3.18.3 Работа с буфером потока

Кластер

Кластер - это непрерывный участок памяти (последовательность байтов), который описывается классом Cluster в файле *io_buffer.h*.

Класс содержит начало кластера (для дисковых кластеров - смещение, для памяти - адрес) и длину кластера в байтах.

Класс реализует методы доступа к данным кластера:

- uint16 _len() - Получить длину кластера.
- size_t _off() - Получить начало кластера.
- const uint8 * _ptr() - Выдать адрес.
- uint8 * _getMemPointer() - Выдать адрес.
- void clear() - Очистить кластер.
- void AllocFile(size_t beg, uint16 len) - Запомнить смещение в файле и кол-во байт.

- void SetClusterOffset(size_t off) - Установить начало кластера.
- void SetClusterLength(uint16 len) - Установить длину кластера.
- static size_t SizeOf(VERSION version) - Размер данных о кластере в потоке указанной версии.

Файловое пространство

Файловое пространство - это набор кластеров. Файловое пространство описывается классом FileSpace в файле *io_buffer.h*.

Класс содержит массив индексов кластеров и количество занятых байтов в последнем кластере.

Класс реализует методы доступа к массиву индексов кластеров:

- size_t Count() - Получить количество элементов в массиве.
- void Flush() - Обнулить количество элементов в массиве.
- void RemoveInd(size_t) - Удалить элемент из массива.
- size_t * Add() - Добавить элемент в конец массива.
- size_t * Add(const size_t &) - Добавить элемент в конец массива.
- void SetSize(size_t, bool) - Установить новый размер массива.
- void Reserve(size_t) - Зарезервировать место под столько элементов.
- bool IsExist(size_t &) - Выдать true, если элемент найден.
- size_t FindIt(size_t & el) - Найти индекс элемента (если не найден вернуть -1).
- size_t * InsertInd(size_t index, const size_t & el) - Вставить пустой элемент перед указанным.
- const size_t * GetAddr() - Выдать адрес начала массива.
- size_t * AddItems(size_t n) - Добавить n элементов в конец массива.
- size_t & operator [](size_t loc) - Оператор доступа к элементу массива по индексу.
- uint16 & rest() - Получить количество занятых байт в последнем кластере.

Позиция чтения/записи

Класс ClusterReference описывает позицию в кластере для чтения/записи. Он определен в файле *io_buffer.h*.

Класс содержит индекс кластера в массиве кластеров буфера iobuf_Seq и смещение в данном кластере.

Потоковый последовательный буфер

Класс iobuf_Seq описывает потоковый буфер, который обеспечивает только последовательную запись/чтение (базовый класс). Класс iobuf_Seq и его наследники служат для выполнения операций чтения и записи в интересах потока (класс tape). Объявлен в файле *io_buffer.h*.

Используются следующие термины:

- Хранилище - файл на диске или область памяти.
- Файл — файловое пространство внутри хранилища.

Основные данные класса:

- FileSpace sys - «системное» файловое пространство. Открывается в конструкторе класса tape.
- PArray<FileSpace> files - список файловых пространств, содержащихся в iobuf_Seq. Первым элементом в нем всегда лежит адрес sys.
- uint32 stateFlag - состояние буфера.
- VERSION storageVers - версия хранилища.
- VERSION curFileVers - версия текущего открытого файлового пространства (потока). В общем случае версия хранилища и версия любого файлового пространства в нем могут не совпадать.
- uint8 bufferMode - Режим, в котором может работать буфер.
- uint8 curFileMode - Режим открытия текущего файлового пространства. В общем случае режим хранилища и режим открытого файлового пространства в нем могут не совпадать. Ограничение - если режим буфера io::in, то попытка открыть файловое пространство на запись (io::out) не приводит к открытию.
- uint8 * base - Указатель на начало буфера в памяти.
- uint8 * ptr - Указатель на след символ в памяти.
- uint8 * end - Указатель на конец буфера в памяти.

Основные методы класса:

- Доступ к массиву кластеров:
  - void Reserve(size_t n, bool addAdditionalSpace) - Зарезервировать место под заданное количество элементов.
  - void Flush() - Обнулить количество элементов.
  - void HardFlush() - Освободить всю память.
  - void Adjust() - Удалить лишнюю память.
  - Cluster * Add() - Добавить элемент в конец массива.
  - Cluster * Add(const Cluster &) - Добавить данный элемент в конец массива.
  - size_t Count() - Выдать количество элементов массива.
  - Cluster & operator [](size_t) - Оператор доступа по индексу.
- Доступ к файловым пространствам:
  - void ReserveFiles(size_t n, bool addAdditionalSpace) - Зарезервировать место под заданное количество файловых пространств.
  - ClusterReference getCurrentClusterPos() - Получить текущую позицию в буфере.
  - FileSpace & sysFile() - Получить доступ к системному файловому пространству.
  - FileSpace * openedFile() - Получить доступ к открытому файловому пространству.
- Доступ к версиям:
  - void SetVersionsByStorage() - Установить текущую версию равной версии хранилища.
  - VERSION MathVersion() - вернуть главную версию (математического ядра).

- VERSION AppVersion(size_t) - Вернуть дополнительную версию (версию конечного приложения).
- const VersionContainer & GetVersionsContainer() - Получить версии буфера.
- VERSION GetStorageVersion() - Узнать версию хранилища.
- VERSION GetFormatVersion() - Узнать версию формата.
- void SetFormatVersion(VERSION) - Установить версию формата.

Потоковый буфер с произвольным доступом

Класс iobuf реализует потоковый буфер с произвольным доступом. Он наследует от класса iobuf_Seq и расширяет его функциональность возможностью удаления и перезаписи файловых пространств (базовый класс). Объявлен в файле *io_buffer.h*.

Класс содержит файловое пространство со списком освобожденных кластеров. При удалении файла все его кластеры переносятся сюда. При записи, когда необходимо распределить новый кластер, сначала проверяется этот массив на наличие свободных кластеров. Кластеры в файловом пространстве всегда лежат упорядоченными по смещению от начала файла.

Методы класса:

- bool open(FileSpace &, uint8, const VersionContainer &, bool fullCheck = true) - Открыть файловое пространство, если оно свое. Флаг fullCheck = false отключает избыточные проверки (ради производительности).
- bool del(FileSpace &) - Освободить место, занимаемое файловым пространством.
- bool truncate(FileSpace &, size_t) - Урезать файловое пространство.
- bool detach(FileSpace &) - Отсоединить файловое пространство от буфера.
- bool speedOnClose() - Нужно ли упорядочивать при закрытии.
- void speedOnClose(bool) - Установить флаг необходимости упорядочивания при закрытии.
- void free(size_t) - Освободить кластер.

Потоковый буфер в памяти

Класс membuf реализует потоковый буфер памяти. Наследует от iobuf. Объявлен в файле *io_memory_buffer.h*.

Потоковый буфер памяти предназначен для использования в потоках чтения и записи. Кроме необходимых для буфера свойств, имеет функции для упаковки в непрерывный блок памяти, что позволяет передавать данные буфера через память в другое приложение.

Класс содержит массив файловых пространств, который хранит все используемые файловые пространства, кроме sys, который определен в iobuf_Seq. При использовании расширенного формата объекты каждого уровня записываются в отдельное файловое пространство. При этом индекс файлового пространства в векторе соответствует уровню объекта в модели.

Основные методы класса:

- bool isEmpty() - Пуст ли буфер.

- size_t toMemory(const char *& memory, size_t addSize = 0) - Записать в непрерывную память. Если memory != 0 (память уже распределена), то addSize равен ее размеру, если memory = 0, то addSize определяет, сколько байтов добавить в начале, обнулив их.
- bool fromMemory(const char *) - Прочитать из непрерывной памяти.
- size_t getMemLen() - Вычислить необходимую длину непрерывного куска памяти для буфера.
- void clean() - Очистить буфер.
- void closeBuff() - Закрыть буфер.

Чтение и запись буфера памяти в файле на диске

Функции записи непрерывного буфера памяти в файл на диске и чтения в непрерывный буфер памяти из файла на диске определены в *io_memory_buffer.h*.

Чтение непрерывного буфера памяти из файла на диске:

- iobuf & createiobuf(const TCHAR * fileName).

Запись непрерывного буфера памяти в файл на диске:

- bool writeiobuftodisk(const TCHAR * fileName, membuf & buf).

3.18.4 Контейнер версий

Тип VERSION определяет версию. Определен в файле *system_types.h*.

Класс VersionContainer определяет контейнер версий объектов геометрического ядра. Он определяет следующие основные методы:

VERSION GetMathVersion() - Вернуть главную версию (версию математического ядра).

VERSION GetAppVersion(size_t ind) - Вернуть дополнительную версию (версию конечного приложения).

void Flush() - Очистить контейнер.

void SetVersion(size_t index, VERSION ver) - Установить версию по индексу.

size_t ToMemory(const char *&) - Записать контейнер в кусок памяти.

size_t FromMemory(const char *) - Прочитать контейнер из куска памяти.

Класс VersionContainer определен в файле *io_version_container.h*.

3.19 АТРИБУТЫ

Атрибуты предназначены для хранения информации, дополняющей геометрическую форму объекта. Различные типы атрибутов могут хранить разнообразные данные, которые могут быть использованы для идентификации объектов, определения данных предметной области приложения, а также для передачи данных между различными приложениями при импорте-экспорте.

Атрибуты могут быть сохранены в любом объекте, являющимся потомком класса MbAttributeContainer.

Атрибуты группируются по следующим семействам:

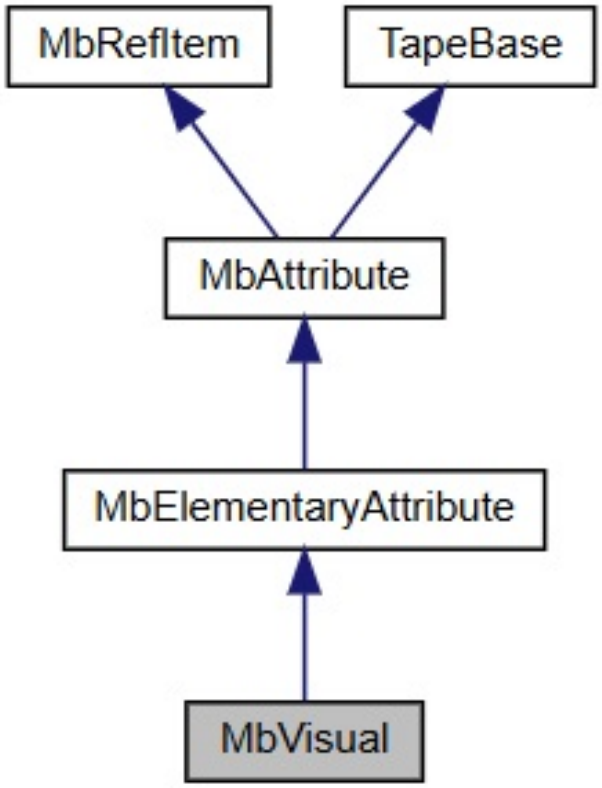
1. **Простые атрибуты**. К ним относятся типы атрибутов, содержащие данные об идентификаторе объекта, его визуальных свойствах, физических характеристиках.
2. **Обобщенные атрибуты**. Атрибуты этого семейства предназначены для хранения данных в одном из стандартных типов, например, целое число, вещественное число, строка, точка, вектор, указатель. С помощью таких атрибутов приложения могут обмениваться какой-либо специфичной информацией.
3. **Связующие атрибуты**. Эти атрибуты предназначены для связи объекта геометрического ядра с абстрактным контейнером данных, то есть набором данных, формат и смысловая нагрузка которых не может быть описана в рамках других типов атрибутов.
4. **Директивные атрибуты**. Эти атрибуты определяют предназначение объекта или действия, которые необходимо с ним произвести, например атрибут «вычитание» подразумевает, что некое тело предназначено для вычитания из другого тела.
5. **Продуктовые атрибуты**. Атрибуты этой группы предназначены для хранения общей информации об изделии.
6. **Пользовательские атрибуты**.
7. **Внешние атрибуты**.

Пример работы с пользовательским атрибутом есть в файле *test_sample_attributes.cpp*, поставляющемся с кодом тестового приложения в дистрибутиве ядра C3D.

3.19.1 Класс атрибута для хранения оптических свойств геометрических объектов для свойств отображения MbVisual

Класс MbVisual предназначен для задания оптических свойств геометрических объектов в C3D Modeler (рекомендуется к использованию взамен устаревшего класса MbColor). Объявления и описания членов класса находятся в файле *attr_color.h*.

Граф наследования MbVisual показан на рисунке:



В предыдущих версиях геометрического ядра C3D (до версии 117978) информация о цвете тела и дочерних топологических объектах (гранях, ребрах, вершинах) сохранялась в атрибуте MbColor. В данном атрибуте возможно хранение четырех компонентов цвета в модели RGBA. Однако для визуализации объекта в 3D-приложениях используется больше параметров и при их отсутствии используются значения по умолчанию, что в некоторых случаях приводит к различию получаемых цветов при открытии модели в разных программах.

Начиная с версии геометрического ядра 117978 для сохранения всех необходимых для визуализации данных используется только атрибут MbVisual, который содержит подробную информацию о каждом компоненте визуализации.

Атрибут MbVisual содержит в себе компоненты:

1. Компонент *_ambient* хранит цвета RED, GREEN, BLUE и коэффициент общего фона в диапазоне 0.0 - 1.0 (0 - 255 в UInt8).
2. Компонент *_diffuse* хранит цвета RED, GREEN, BLUE и коэффициент диффузного отражения в диапазоне 0.0 - 1.0 (0 - 255 в UInt8).
3. Компонент *_specular* хранит цвета RED, GREEN, BLUE и коэффициент зеркального отражения света в диапазоне 0.0 - 1.0 (0 - 255 в UInt8).
4. Компонент *_emission* хранит цвета RED, GREEN, BLUE и коэффициент излучения в диапазоне 0.0 - 1.0 (0 - 255 в UInt8).
5. Компонент *_shininess* хранит блеск (показатель степени в законе зеркального отражения) в диапазоне 0 - 128 UInt8.
6. Компонент *_opacity* хранит коэффициент непрозрачности (коэффициент суммарного отражения) в диапазоне 0.0 - 1.0 (0 - 255 в UInt8).

7. Компонент *_chrom* хранит коэффициент зеркального отражения объектов в диапазоне 0.0 - 1.0 (0 - 255 в UInt8).

Примечание - компонент *_shininess* согласно спецификации OpenGL может иметь значения от 0 до 128 и тип float.

Для хранения компонентов 1–4 используется тип MbRGBA, в котором в полях RGB хранится значение цвета, а в поле A — коэффициент использования цвета.

Ниже приведены методы, применяемые при использовании только диффузного цвета:

1. При задании цвета для тела (грани, ребра, вершины) методом MbAttributeContainer:: SetColor(int iR, int iG, int iB) получается атрибут MbVisual, если его нет, то он создается. В компонент *_diffuse* записывается цвет RGB, в альфа-канал (A) записывается 255 (1.0, 100%).
2. При получении цвета методом uint32 MbAttributeContainer:: GetColor() const получается атрибут MbVisual, если его нет — возвращается цвет по умолчанию (MB_C3DCOLOR). Если атрибут есть, то цвет получается из компонента *_diffuse* (RGB).

При чтении файлов старых версий, содержащих атрибут MbColor, данный атрибут будет преобразован в MbVisual следующим образом:

1. При чтении файла, в котором есть только MbVisual (новая версия), все атрибуты цвета и коэффициенты смешивания используются согласно спецификации MbVisual.
2. При чтении файлов предыдущих версий, в которых содержится только MbColor, создается новый атрибут MbVisual, в компонент *_diffuse* записывается цвет из MbColor, в альфа-канал записывается 255 (1.0, 100%). Атрибут MbColor удаляется.
3. При чтении файлов предыдущих версий, содержащих MbColor и MbVisual, где в MbVisual в каждом компоненте (RGB) содержится коэффициент смешивания (предыдущий вариант использования атрибута MbVisual), а в MbColor сохранен основной цвет, выполняются следующие действия:

   а) Создается новый атрибут MbVisual. Во все компоненты (*_ambient*, *_diffuse*, *_specularity* и *_emission*) записывается основной цвет из прочитанного MbColor.

   б) В альфа-канал каждого из компонентов записывается значение из прочитанного файла MbVisual.

   в) Прочитанные атрибуты MbColor и MbVisual удаляются, остается только новый атрибут MbVisual.
4. При сохранении файла C3D для старой версии создается атрибут MbColor, в который записывается цвет из *_diffuse* нового атрибута MbVisual, далее создается еще один атрибут MbVisual, в который записываются коэффициенты из альфа-каналов компонентов имеющегося атрибута MbVisual. Атрибут MbVisual, который был в модели (новый вариант), удаляется и записывается файл в старом формате.

Атрибут MbColor — устаревший и не рекомендуется к использованию. Вместо него необходимо использовать атрибут MbVisual.

Для создания атрибута рекомендуется использовать следующие статические методы:

1. static SPtr<MbVisual> Create(uint32 **c**);
2. static SPtr<MbVisual> Create(int iR, int iG, int iB);
3. static SPtr<MbVisual> Create(const MbRGBA & **c**);

Входными параметрами методов являются:

- **c** — цвет (32 бит). Для его создания из компонентов можно использовать inline-функции uint32 RGB2uint32() (находятся в файле *attr_color.h*).
- iR, iG, iB — компоненты цвета.

В результате работы этих функций возвращается созданный объект MbVisual.

Метод

```
bool Init( const MbAttribute & ) override;
```

позволяет инициализировать данные по присланному атрибуту.

Метод

```
void Init( const MbRGBA & ambient,
           const MbRGBA & diffuse,
           const MbRGBA & specular,
           const MbRGBA & emission,
                 uint8    shininess,
                 uint8    opacity,
                 uint8    chrom );
```

позволяет задать значение каждого из используемых визуальных компонентов.

Метод

```
void Get( MbRGBA & ambient,
          MbRGBA & diffuse,
          MbRGBA & specular,
          MbRGBA & emission,
          uint8 &  shininess,
          uint8 &  opacity,
          uint8 &  chrom ) const;
```

позволяет получить значение каждого из используемых визуальных компонентов.

Методы

```
float Shininess() const;         // Дать блеск (показатель степени в законе␣
↪зеркального отражения).
float Opacity() const;           // Дать коэффициент непрозрачности.
float Chrom() const;             // Дать коэффициент зеркального отражения объектов.
const MbRGBA & GetAmbient() const;     // Дать коэффициент общего фона.
const MbRGBA & GetDiffuse() const;     // Дать коэффициент диффузного отражения.
const MbRGBA & GetSpecularity() const; // Дать коэффициент зеркального отражения␣
↪света.
const MbRGBA & GetEmission() const;    // Дать коэффициент излучения.
const MbUuid & GetIdentifier() const;  // Дать идентификатор.
uint8 GetShininess() const;      // Дать блеск (показатель степени в законе␣
↪зеркального отражения).
uint8 GetOpacity() const;        // Дать коэффициент непрозрачности.
uint8 GetChrom() const;          // Дать коэффициент зеркального отражения объектов.
```

обеспечивают выдачу отдельных свойств для OpenGL.

Методы, позволяющие установить отдельные свойства для OpenGL:

```
void SetShininess( float v );          // Установить блеск (показатель степени в
↪законе зеркального отражения).
 void SetOpacity( float v );            // Установить коэффициент непрозрачности.
 void SetChrom( float v );              // Установить коэффициент зеркального
↪отражения объектов.
 void SetAmbient( const MbRGBA & v );   // Установить коэффициент общего фона.
 void SetDiffuse( const MbRGBA & v );   // Установить коэффициент диффузного
↪отражения.
 void SetSpecularity( const MbRGBA & v ); // Установить коэффициент зеркального
↪отражения света.
 void SetEmission( const MbRGBA & v );  // Установить коэффициент излучения.
 void SetIdentifier( const MbUuid & v );// Установить идентификатор.
 void SetShininessInt( uint8 v );       // Установить блеск (показатель степени в
↪законе зеркального отражения).
 void SetOpacityInt( uint8 v );         // Установить коэффициент непрозрачности.
 void SetChromInt( uint8 v );           // Установить коэффициент зеркального
↪отражения объектов.
```

Методы работы с составляющей цвета (использовать в случае, если используется только цвет):

```
uint32 GetColor() const { return MbRGBA( _diffuse[0], _diffuse[1], _diffuse[2], 0
↪).Get(); } // Выдача составляющей цвета.
void SetColor( uint32 c );                  // Установить составляющую цвета.
void SetColor( const MbRGBA & c );          // Установить составляющую цвета.
void SetColor( int iR, int iG, int iB );    // Установить составляющую цвета.
```

Устаревшие методы, не рекомендуемые к использованию (используются для совместимости с прошлыми версиями ядра C3D)

```
MbVisual( float a = MB_AMBIENT,
          float d = MB_DIFFUSE,
          float s = MB_SPECULARITY,
          float h = MB_SHININESS,
          float t = MB_OPACITY,
          float e = MB_EMISSION );

MbVisual( uint8 a, uint8 d, uint8 s, uint8 h, uint8 t, uint8 e );

void Init( float a = MB_AMBIENT,
           float d = MB_DIFFUSE,
           float s = MB_SPECULARITY,
           float h = MB_SHININESS,
           float t = MB_OPACITY,
           float e = MB_EMISSION,
           uint rgb = 0 );

 // Позволяет получить значение каждого из используемых визуальных компонентов
 void Get( float & a, float & d, float & s, float & h, float & t, float & e, uint rgb
↪= 0 ) const;

 float Ambient( uint rgb = 0 ) const;     // (устаревший) Дать коэффициент общего
↪фона.
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
float Diffuse( uint rgb = 0 ) const;      // (устаревший) Дать коэффициент диффузного␣
↪отражения.
float Specularity( uint rgb = 0 ) const; // (устаревший) Дать коэффициент␣
↪зеркального отражения света.
float Emission( uint rgb = 0 ) const;     // (устаревший) Дать коэффициент излучения.

void SetAmbient( float v, uint rgb = 0 );      // (устаревший) Установить коэффициент␣
↪общего фона.
void SetDiffuse( float v, uint rgb = 0 );      // (устаревший) Установить коэффициент␣
↪диффузного отражения.
void SetSpecularity( float v, uint rgb = 0 ); // (устаревший) Установить коэффициент␣
↪зеркального отражения света.
void SetEmission( float v, uint rgb = 0 );     // (устаревший) Установить коэффициент␣
↪излучения.
```

3.20 ПОСТРОЕНИЕ ТРИАНГУЛЯЦИИ

Геометрическое ядро C3D выполняет построение полигонального представления геометрической модели по её граничному представлению. Полигональное представление содержит набор триангуляций. Каждая триангуляция аппроксимирует отдельную грань моделируемого объекта плоскими пластинами треугольной и четырёхугольной формы. Полигональное представление используется для визуализации геометрической модели, вычисления инерционных характеристик, определения соударений элементов модели.

3.20.1 Управление вычислением триангуляции

На входе в методы построения полигональных представлений используется структура MbStepData, которая приведена на рис. 1:

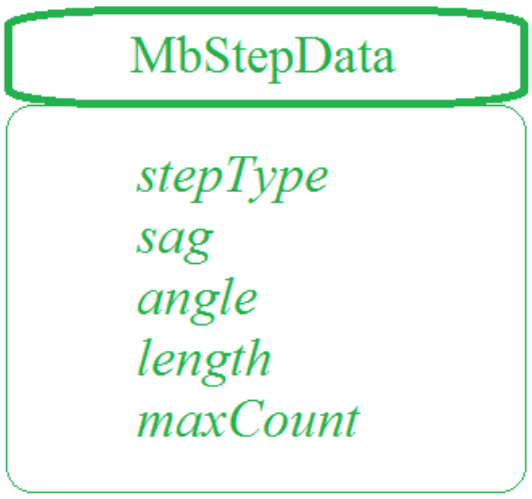


Рис. 1

Структура *MbStepData* объявлена в файле *mb_data.h*. Структура *MbStepData* содержит

следующие данные:

- *unsigned char stepType* – поле, определяющее способ вычисления приращения параметра;
- *double sag* – максимально допустимое отклонение по прогибу;
- *double angle* – максимально допустимое отклонение по углу касательных или нормалей;
- *double length* – максимально допустимое расстояние между соседними точками;
- *unsigned int maxCount* – максимальное количество ячеек в строке или столбце триангуляционной сетки.

Структура *MbStepData* управляет густотой сетки полигонального объекта и содержит данные, по которым вычисляется шаг по параметру при движении вдоль кривых и поверхностей модели. Поле *stepType* определяет способ вычисления приращения параметра при движении вдоль кривой или поверхности и может содержать маски перечисления MbeStepType, объявленного в файле *mb_enum.h*:

- *ist_SpaceStep* – для визуализации геометрической формы;
- *ist_DeviationStep* – для операций построения;
- *ist_MetricStep* – для 3D принтеров;
- *ist_ParamStep* – для визуализации геометрической формы объектов с привязкой текстуры к параметрам поверхности;
- *ist_CollisionStep* – для определения столкновений элементов модели;
- *ist_MipStep* – для вычисления инерционных характеристик.

Параметр *sag* ограничивает шаг по параметру кривой или поверхности исходя из максимально допустимого отклонения полигонального объекта от оригинала по прогибу. Параметр *angle* ограничивает шаг по параметру кривой или поверхности исходя из максимально допустимого отклонения полигонального объекта от оригинала по угловому отклонению касательных кривой или нормалей поверхности в соседних точках на расстоянии шага. Параметр *length* ограничивает шаг по параметру кривой или поверхности исходя из максимально допустимого размера элемента полигонального объекта – стороны треугольника или отрезка полигона. Параметр *maxCount* ограничивает шаг по параметру кривой или поверхности исходя из максимально допустимого количества разбиений в строке или столбце триангуляционной сетки.

На входе в методы построения полигональных представлений используется структура *MbFormNote*, которая приведена на рис. 2:

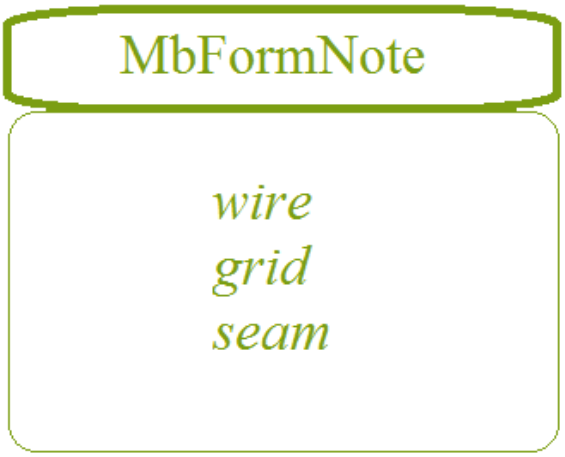


Рис. 2

Структура *MbFormNote* объявлена в файле *mb_data.h*. Структура *MbFormNote* содержит следующие данные:

- *bool wire* – флаг построения полигонного объекта;
- *bool grid* – флаг построения полигонного объекта;
- *bool seam* – флаг использования шовных рёбер.

Структура *MbFormNote* определяет способ построения полигонального объекта: при *wire* == true полигональный объект заполняется ломаными линиями, при *grid* == true полигональный объект заполняется триангуляциями. Параметр *seam* определяет способ представления швов в полигональном объекте: при *seam* == true триангуляция не замыкается на швах и шовные рёбра обрабатываются как обычные рёбра – в триангуляции их точки считаются краевыми и для них строятся полигоны; при *seam* == false триангуляция замыкается на швах и шовные рёбра игнорируются – триангуляции строятся замкнутыми, а полигоны для швов не строятся.

3.20.2 Построение полигонального объекта

Виртуальный метод объектов геометрической модели

```
MbItem * MbItem::CalculateMesh( const MbStepData &     stepData,
                                const MbFormNote &     note,
                                      MbRegDuplicate * iReg ) const;
```

строит полигональный объект, аппроксимирующий рассматриваемый объект геометрической модели.

Входными параметрами метода являются:

- *stepData* – данные для вычисления шага аппроксимации;
- *note* – способ построения полигонального объекта;
- *iReg* – регистратор копируемых объектов.

В случае успешной работы метод возвращает указатель на вновь построенный объект геометрической модели *MbItem*; в противном случае возвращает ноль.

Метод объявлен в файле *item.h* и заголовочных файлах наследников MbItem.

Рассматриваемый метод аппроксимирует объекты модели, создавая полигональные копии с аналогичной структурой. Для тела, проволочного каркаса и точечного каркаса рассматриваемый метод создаст полигональный объект *MbMesh*, аппроксимирующий исходный объект, и вернёт указатель на созданный объект. Каждая триангуляция *grids*[*i*] объекта *MbMesh* будет аппроксимировать *i*-ую грань, каждый полигон *wires*[*i*] объекта *MbMesh* будет аппроксимировать *i*-ое ребро, каждый апекс *peaks*[*i*] объекта *MbMesh* будет аппроксимировать *i*-ую вершину. Для полигонального объекта *MbMesh* рассматриваемый метод создаст полигональную копию. Для вставки рассматриваемый метод создаст вставку с объектом, который создаст этот же метод для содержимого вставки. Для сборочной единицы рассматриваемый метод создаст сборочную единицу с объектами, которые создаст этот же метод для объектов сборочной единицы.

Параметр *stepData* управляет густотой сетки полигонального объекта и содержит данные для вычисления шага при движении вдоль кривых и поверхностей объекта модели. Параметр *note* определяет способ построения полигонального объекта. Параметры *stepData* и *note* описаны в параграфе *Triangulation_Calculation_Control*. При *note.wire* == true рассматриваемый метод заполняет множество указателей на полигоны *mesh.wires*; при *note*.*grid* == true рассматриваемый метод заполняет множество указателей на триангуляции *mesh.grids* (для граней и поверхностей), множество указателей на полигоны *mesh.wires* (для рёбер), множество указателей на апексы *mesh.peakes* (для вершин).

Параметр *iReg* может быть равен нулю. Он служит для передачи вложенным методам информации о уже обработанных объектах.

На рис. 1 приведён полигональный объект тела, построенный с параметрами *note.wire* == false, *note.grid* == true, содержащий триангуляции, полигоны и апексы. На рис. 2 приведён полигональный объект тела, построенный с параметрами *note.wire* == true, *note.grid* == false, содержащий только полигоны.

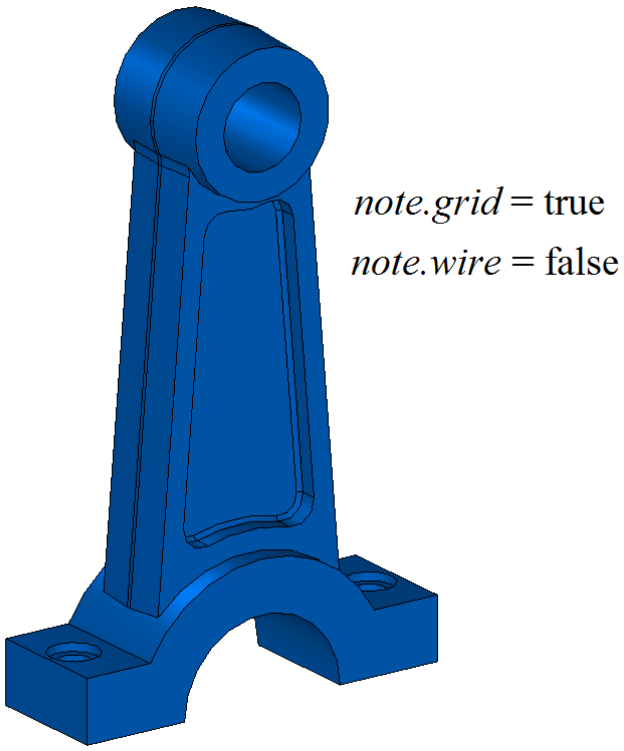


Рис. 1

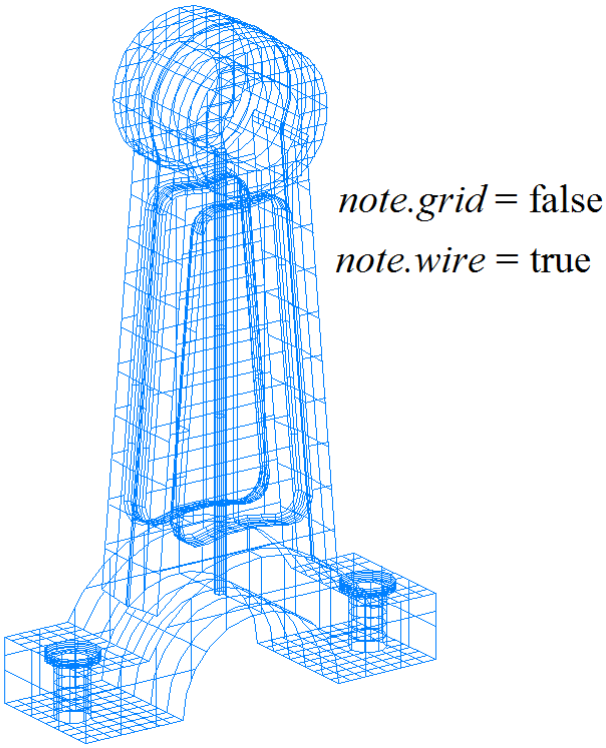


Рис. 2

На рис. 3 приведён полигональный объект сборочной единицы, построенный с параметрами *note.wire* == false, *note.grid* == true, состоящий из сборочной единицы полигональных объектов, аппроксимирующих детали:

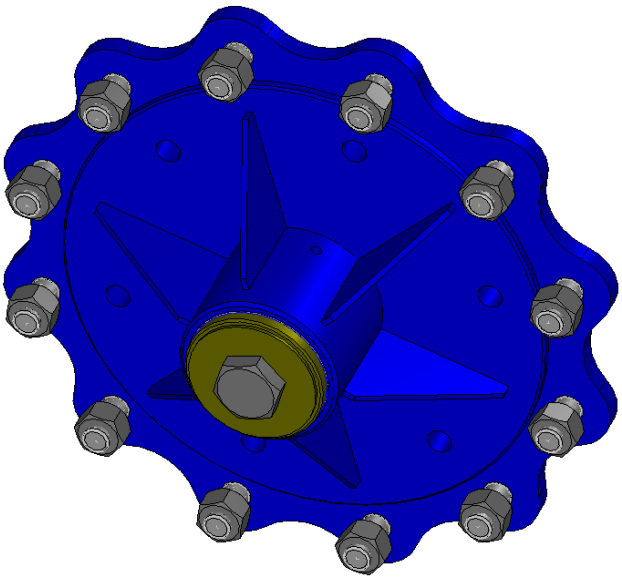


Рис. 3

Рассматриваемый метод используется для визуализации объектов геометрической модели. Полигональные объекты легко трансформируются, для них можно быстро найти пересечение с прямой линией. На экране рисуется именно полигональная копия объекта геометрической модели, а объект в исходном представлении остаётся за кадром.

3.20.3 Добавление полигонального объекта

Виртуальный метод объектов геометрической модели

```
bool MbItem::AddYourMesh( const MbStepData & stepData,
                          const MbFormNote & note,
                                MbMesh &     mesh ) const;
```

строит и добавляет свою полигональную копию в присланный полигональный объект *mesh*.

Входными параметрами метода являются:

- *stepData* – данные для вычисления шага аппроксимации;
- *note* – способ построения полигонального объекта.

Выходным параметром метода является полигональный объект *mesh*.

В случае успешной работы метод возвращает true.

Метод объявлен в файле *item.h* и заголовочных файлах наследников MbItem.

Рассматриваемый метод аппроксимирует объекты модели полигональными копиями и добавляет их в присланный объект *mesh*. Для вставки рассматриваемый метод создаст полигональную копию содержимого вставки, трансформирует её в глобальную систему координат и добавит в присланный объект *mesh*. Для сборочной единицы рассматриваемый метод создаст полигональную копию содержимого сборочной единицы, трансформирует её в глобальную систему координат и добавит в присланный объект *mesh*.

Так же, как в методе CalculateMesh(), параметр *stepData* управляет густотой сетки полигонального объекта, а параметр *note* определяет способ построения полигонального объекта. Параметры *stepData* и *note* описаны в параграфе *Triangulation_Calculation_Control*. В отличие от вышеупомянутого метода рассматриваемый метод для сложных объектов создаёт единый полигональный объект. На рис. 1 приведён полигональный объект показанной на *рис. 3* сборочной единицы, построенный рассматриваемым методом:

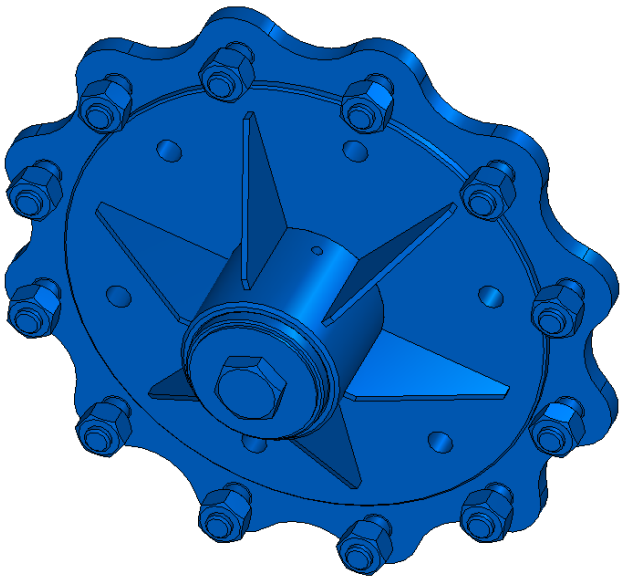


Рис. 1

3.20.4 Построение полигонов объекта

Виртуальный метод трёхмерных геометрических объектов

```
void MbSpaceItem::CalculateWire( double sag,
                                 MbMesh & mesh );
```

заполняет присланный полигональный объект *mesh* набором полигонов, аппроксимирующих геометрический объект.

Входным параметром метода является *sag* – максимально допустимое отклонение полигонального элемента от оригинала по прогибу.

Выходным параметром метода является полигональный объект *mesh*.

Метод объявлен в файле *space_item.h* и заголовочных файлах наследников MbSpaceItem.

Полигональный объект описан в подразделе *Полигональный объект MbMesh*. Параметр *sag* определяет максимально допустимое расстояние между объектом и ломаной линией, проходящей по точкам полигонов. Рассматриваемый метод заполняет только множество указателей на полигоны *mesh.wires* (ломаные линии).

Кривую рассматриваемый метод аппроксимирует одним полигоном. Контур рассматриваемый метод аппроксимирует несколькими полигонами, каждый из которых проходит по соответствующему сегменту контура. На рис. 1 приведена кривая и её полигональный объект, состоящий из одного полигона.

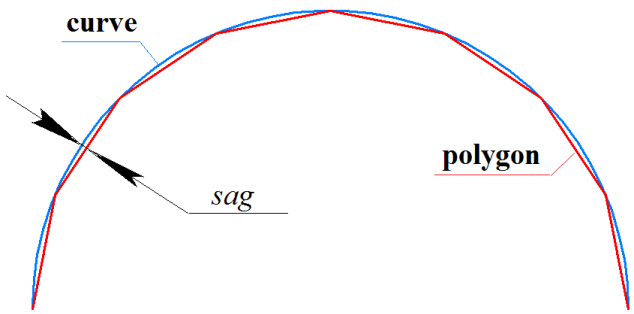


Рис. 1

Поверхность рассматриваемый метод аппроксимирует набором полигонов, проходящих по *u*-линиям, *v*-линиям и по границе поверхности. На рис. 2 приведён полигональный объект поверхности, состоящий из нескольких ломаных линий.

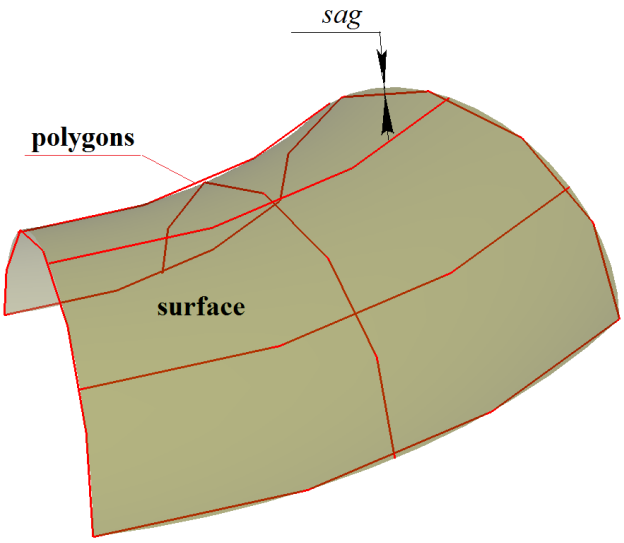


Рис. 2

Тело рассматриваемый метод аппроксимирует набором ломаных линий, проходящих внутри грани вдоль *u*-линий и *v*-линий поверхностей граней, и набором ломаных, аппроксимирующих кривые, на которых базируются рёбра тела. На рис. 3 справа приведён полигональный объект тела:

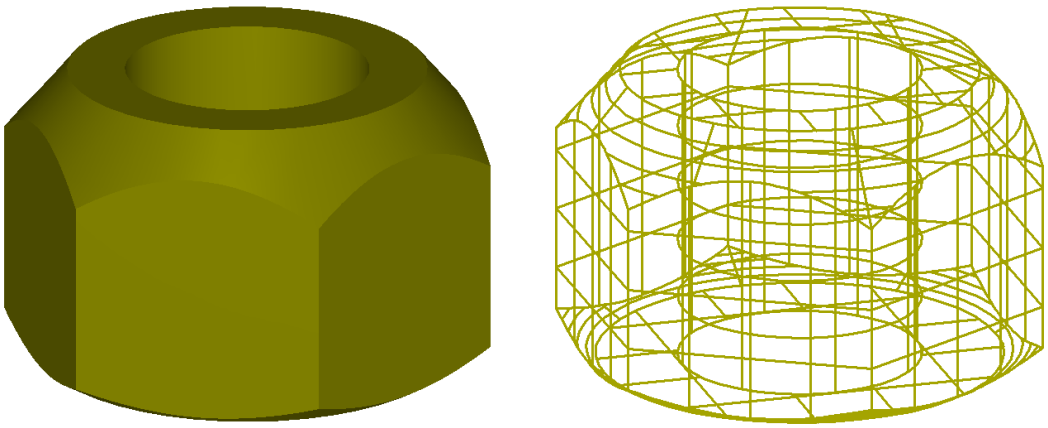


Рис. 3

Для объектов геометрической модели рассматриваемый метод работает аналогично методу CalculateMesh() при *stepData.stepType* == *ist_SpaceStep*, *stepData.sag* == *sag*, *note.wire* == true, *note.seam* == true.

3.20.5 Построение триангуляции грани

Метод

```
void CalculateGrid( const MbFace &     face,
                    const MbStepData & stepData,
                          bool         edgePoints,
                          MbGrid &     grid,
                          bool         dualSeams = true );
```

аппроксимирует грань треугольными и четырёхугольными пластинами.

Входными параметрами метода являются:

- *face* – грань;
- *stepData* – данные для вычисления шага аппроксимации;
- *edgePoints* – флаг использования пространственных точек;
- *dualSeams* – флаг обработки швов.

Выходным параметром метода является триангуляция *grid*.

Метод объявлен в файле *tri_face.h*.

Параметр *stepData* управляет густотой триангуляции и содержит данные для вычисления шага при движении вдоль поверхности грани. Параметр *stepData* описан в подразделе *Управление вычислением триангуляции*. Для различных значений *stepData.stepType* заполняются различные данные триангуляции *grid*. Если *stepData.stepType* содержит маску *ist_MipStep*, то заполняется множество *grid.params*. Если *stepData.stepType* содержит маску *ist_CollisionStep* или *ist_ParamStep*, то заполняются множества *grid.params*, *grid.points* и *grid.normals*. В остальных случаях заполняются множества *grid.params* и *grid.points*.

На рис. 1 приведена триангуляция плоской грани с маской *ist_SpaceStep*:

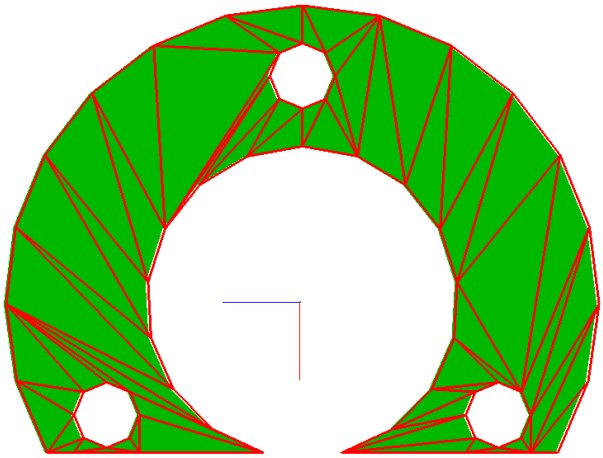


Рис. 1

На рис. 2 приведена триангуляция криволинейной грани с маской *ist_SpaceStep*.

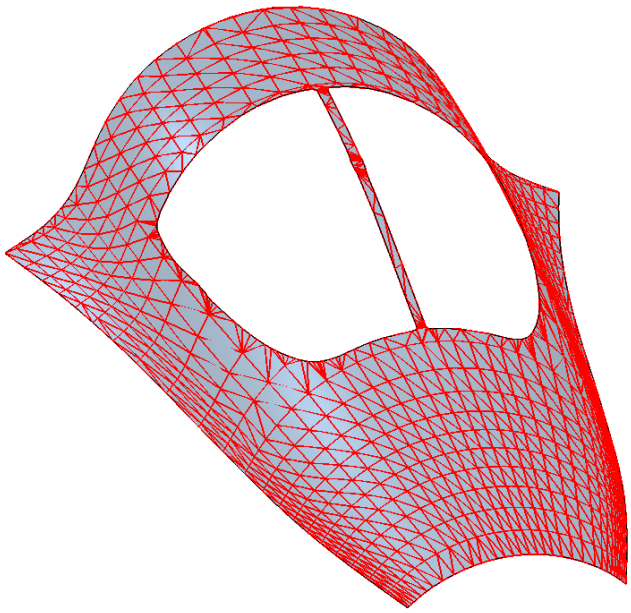


Рис. 2

Рассматриваемый метод вызывается при построении полигональных объектов методами CalculateMesh() и AddYourMesh(), если *note.grid* == true.

3.20.6 Построение триангуляции тела

Метод

```
void CalculateGrid( const MbSolid &          solid,
                    const MbStepData &       stepData,
                          RPArray<MbGrid> & grids );
```

строит множество триангуляций для граней тела.

Входными параметрами метода являются:

- *solid* – тело;
- *stepData* – данные для вычисления шага аппроксимации.

Выходным параметром метода является множество *grids*.

Метод не возвращает значений.

Метод объявлен в файле *mip_solid_area_volume.h*.

Рассматриваемый метод аппроксимирует грани тела *solid* триангуляциями *grids*. При вызове метода множество *grids* должно быть пустым. Каждой *i*-ой грани тела *solid* соответствует построенный объект *grids*[*i*].

Параметр *stepData* управляет густотой сетки полигональных объектов и содержит данные для вычисления шага при движении вдоль кривых и поверхностей тел. Параметр *stepData* описан в подразделе *Управление вычислением триангуляции*.

3.20.7 Построение полигональных объектов множества тел

Метод

```
void CalculateGrid( const RPArray<MbSolid> & solids,
                    const MbStepData &       stepData,
                          RPArray<MbMesh> &  meshs );
```

строит множество полигональных объектов для множества тел.

Входными параметрами метода являются:

- *solids* – множество тел;
- *stepData* – данные для вычисления шага аппроксимации.

Выходным параметром метода является множество *meshs*.

Метод не возвращает значений.

Метод объявлен в файле *mip_solid_area_volume.h*.

Рассматриваемый метод аппроксимирует тела *solids* полигональными объектами *meshs*. При вызове метода множество *meshs* должно быть пустым. Каждому телу *solids*[*i*] соответствует построенный объект *meshs*[*i*].

Параметр *stepData* управляет густотой сетки полигональных объектов и содержит данные для вычисления шага при движении вдоль кривых и поверхностей тел. Параметр *stepData* описан в подразделе *Управление вычислением триангуляции*.

3.21 ПОСТРОЕНИЕ ПЛОСКИХ ПРОЕКЦИЙ

Для построения плоской проекции моделируемого объекта геометрическое ядро C3D использует каркасную модель. Каркасную модель получим из граничного представления геометрической модели, взяв рёбра и добавив вместо граней их линии очерка. Линии

очерка проходят внутри граней и делят их на видимые и невидимые из точки наблюдения части. Плоские проекции более информативны, если в каркасной модели скрыты невидимые из точки наблюдения рёбра и линии очерка.

3.21.1 Данные построения плоских проекций

На входе в метод построения плоских проекций для передачи тел используется структура **MbLump**, которая приведена на рис. 1.

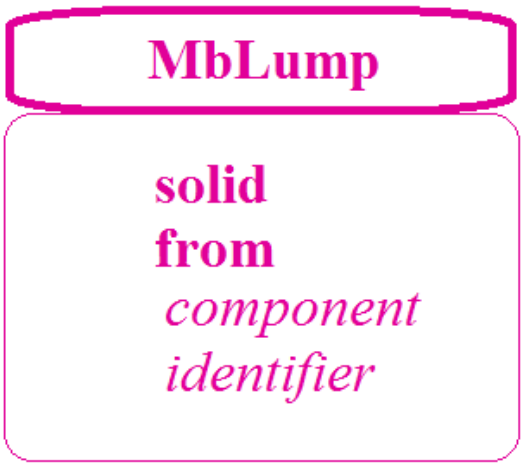


Рис. 1

Структура **MbLump** объявлена в файле *lump.h*. Структура **MbLump** содержит указатель на тело **solid**, матрицу преобразования тела из локальной системы координат **from**, идентификационные параметры тела *component* и *identifier*.

Для передачи вспомогательных объектов в метод построения плоских проекций используется класс **MbProjectionsObjects**, который приведен на рис. 2:

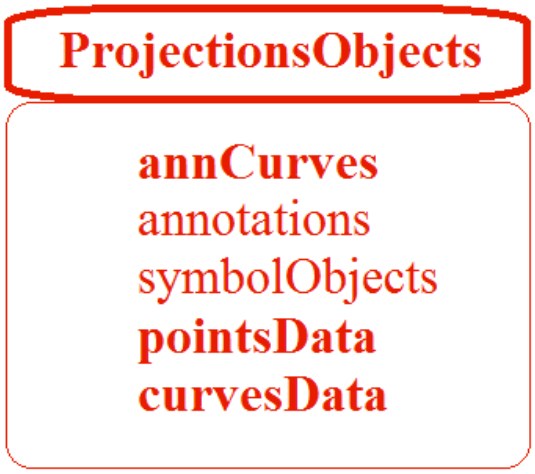


Рис. 2

Класс MbProjectionsObjects объявлен в файле *map_create.h*. Класс MbProjectionsObjects содержит следующие данные:

- TPointer< PArray< MbAnnCurves > > **annCurves** – аннотационные кривые,
- TPointer< RPArray< MbSimbolthThreadView > > annotations – вспомогательные объекты,
- TPointer< RPArray< MbSymbol > > symbolObjects – условные обозначения,
- TPointer< RPArray< MbSpacePoints > > **pointsData** – точки,
- TPointer< RPArray< MbSpaceCurves > > **curvesData** – кривые.

Для передачи результатов построения плоской проекции объекта используется структура **MbVEFVestiges**, которая приведена на рис. 3:

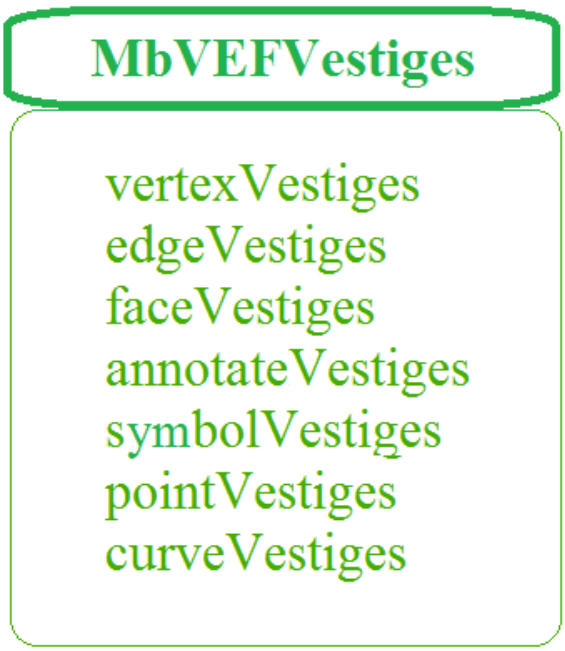


Рис. 3

Структура **MbVEFVestiges** объявлена в файле *map_vestiges.h*. Структура **MbVEFVestiges** содержит проекции элементов объекта на проекционную плоскость, объединенные в группы. Каждый элемент группы проекций состоит из построенной проекции, указателя на породивший проекцию объект, сведений о видимости этой проекции и другой информации о проекции. Структура **MbVEFVestiges** содержит следующие группы:

- PArray< MbVertexVestige > vertexVestiges – множество проекций вершин,
- PArray< MbEdgeVestige > edgeVestiges – множество проекций ребер,
- PArray< MbFaceVestige > faceVestiges – множество проекций граней,
- PArray< MbAnnotationVestige > annotateVestiges – множество проекций аннотационных объектов,
- PArray< MbSymbolVestige > symbolVestiges – множество проекций условных обозначений,
- PArray< MbVertexVestige > pointVestiges – множество проекций точек,
- PArray< MbEdgeVestige > curveVestiges – множество проекций кривых.

3.21.2 Построение плоской проекции модели

Метод

```
void GetVestiges( const MbPlacement3D &        place,
                  double                       znear,
                  const RPArray< MbLump > &    lumps,
                  const MbProjectionsObjects & objects,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                        MbVEFVestiges &         result,
                        bool                    invisible,
                        VERSION                 version );
```

выполняет построение плоской проекции множества тел и других объектов.

Входными параметрами метода являются:

- **place** – проекционная плоскость,
- *znear* – параметр точки наблюдения,
- **lumps** – проецируемые тела в локальных системах координат,
- **objects** – прочие проецируемые объекты,
- *invisible* – признак построения невидимых линий,
- version – версия построения, последняя версия Math::DefaultMathVersion().

Выходным параметром метода является **result** – структура, содержащая информацию о проекции.

Метод не возвращает значений.

Метод объявлен в файле *map_create.h*.

Проекционной плоскостью является плоскость XY локальной системы координат **place**.

Параметр *znear* определяет тип изображения. При *znear* = 0 строится параллельная проекция объектов. Параметр **lumps** содержит тела и матрицы преобразования их из локальной системы координат. Параметр **objects** содержит вспомогательные объекты, необходимые для оформления проекции: вспомогательные точки, кривые, условные обозначения, объекты аннотации. Параметр *invisible* сообщает о необходимости строить невидимые линии. В некоторых случаях при отказе от построения невидимых линий метод работает заметно быстрее.

На рис. 1 приведено тело, линии проекции которого на плоскость, параллельную плоскости экрана, приведены на рис. 2. На рис. 3 приведены только видимые линии проекции тела, показанного на рис. 1.

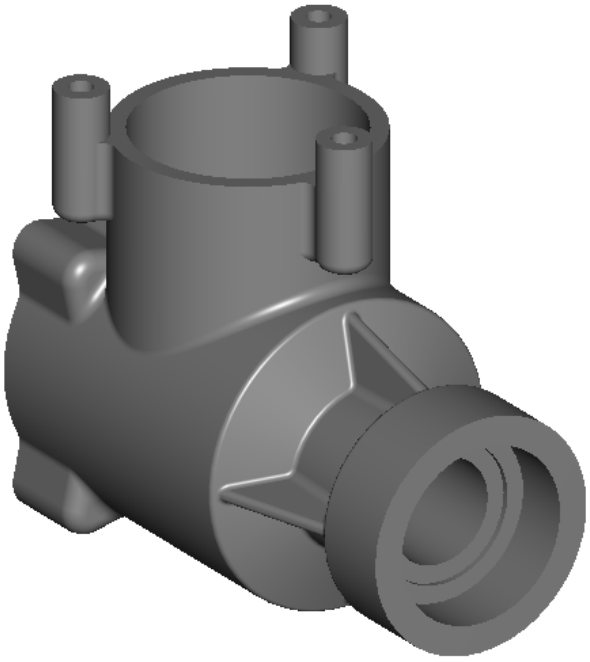


Рис. 1

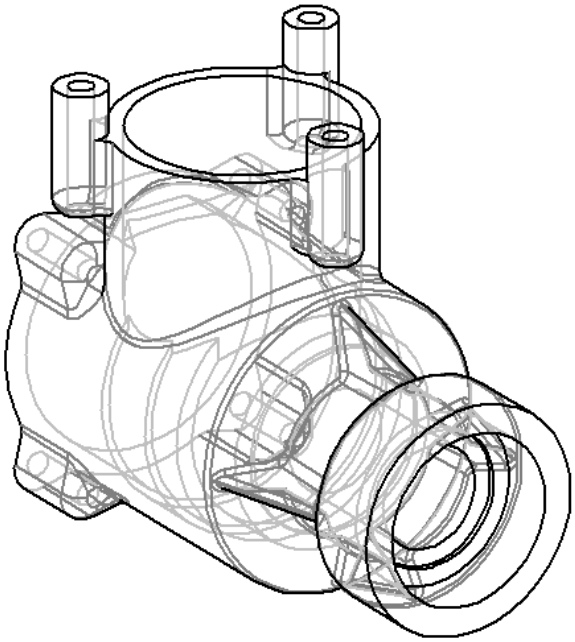


Рис. 2

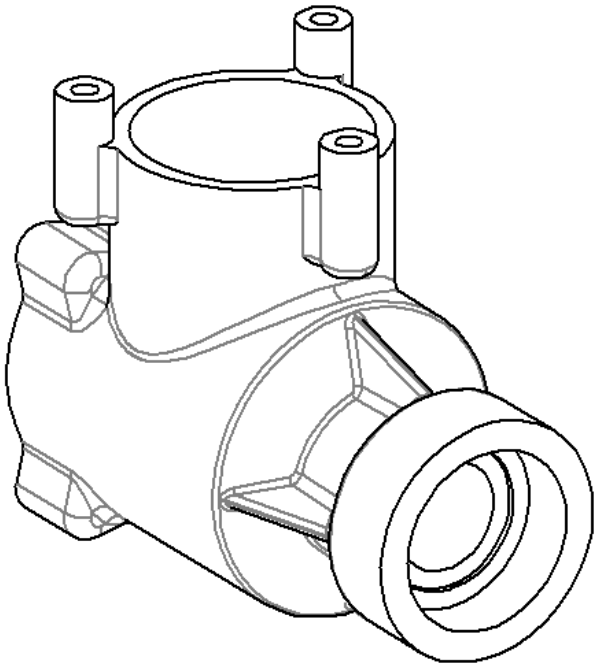


Рис. 3

Последний параметр рассматриваемого метода используется для поддержки предыдущих версий построения.

Метод

```
void VisualLinesMapping( const MbPlacement3D &     place,
                         double                    znear,
                         const RPArray< MbLump > & lumps,
                               MbVEFVestiges &     result,
                               bool                invisible = true );
```

выполняет построение плоской проекции множества тел и отличается от метода GetVestiges() отсутствием дополнительных объектов **objects**.

3.21.3 Построение полигональной проекции тел

Метод

```
void HiddenLinesMapping( const RPArray< MbLump > &           lumps,
                         const MbPlacement3D &               place,
                               double                        znear,
                               double                        sag,
                               RPArray< MbPolygon3DSolid > & visibleEdges,
                               RPArray< MbPolygon3DSolid > & hiddenEdges,
                               RPArray< MbPolygon3DSolid > & visibleTangs,
                               RPArray< MbPolygon3DSolid > & hiddenTangs );
```

выполняет построение полигональной проекции множества тел на заданную плоскость.

Входными параметрами метода являются:

- **lumps** – проецируемые тела в локальных системах координат,
- **place** – проекционная плоскость,
- **znear** – параметр точки наблюдения,
- **sag** – максимально допустимое отклонение по прогибу.

Выходными параметрами метода являются:

- visibleEdges – полигоны видимых негладких ребер,
- hiddenEdges – полигоны невидимых негладких ребер,
- visibleTangs – полигоны видимых гладких ребер,
- hiddenTangs – полигоны невидимых гладких ребер.

Метод не возвращает значений.

Метод объявлен в файле *map_create.h*.

Проекционной плоскостью является плоскость XY локальной системы координат **place**.

Параметр **lumps** содержит тела и матрицы преобразования их из локальной системы координат. Параметр *znear* определяет тип изображения. При *znear* = 0 строится параллельная проекция объектов. Класс MbPolygon3DSolid содержит номер компонента и указатель на полигон, состоящий из множества точек, последовательное соединение которых дает аппроксимацию проекции ребра на плоскость. Точность аппроксимации определяет параметр *sag*. Полигоны visibleEdges и hiddenEdges представляют собой соответственно видимые и невидимые полигональные проекции негладких ребер на плоскость XY локальной системы координат **place**. Полигоны **visibleTangs** и **hiddenTangs** представляют собой соответственно видимые и невидимые полигональные проекции гладких ребер на плоскость XY локальной системы координат **place**.

Метод

```
void VisualLinesMapping( const RPArray< MbLump > &           lumps,
                         const MbPlacement3D &               place,
                               double                        znear,
                               double                        sag,
                               RPArray< MbPolygon3DSolid > & visibleEdges,
                               RPArray< MbPolygon3DSolid > & visibleTangs );
```

выполняет построение только видимой полигональной проекции множества тел на заданную плоскость. Рассматриваемый метод выполняет те же построения, что и метод HiddenLinesMapping(), с той разницей, что строит только видимые проекции. В некоторых случаях метод VisualLinesMapping() работает заметно быстрее метода HiddenLinesMapping().

3.21.4 Построение линий очерка триангуляции

Метод

```
void CalculateBoundsSltFast( const MbGrid &                    grid,
                             const MbMatrix3D &                matrix,
                                   bool                        perspective,
                                   RPArray< MbFloatPoint3D > & points );
```

выполняет построение линий очерка триангуляции.

Входными параметрами метода являются:

- **grid** – триангуляция грани тела,
- **matrix** – матрица, определяющая вектор взгляда,
- **perspective** – признак перспективного отображения.

Выходным параметром метода является:

points – множество указателей на точки из триангуляции **grid.points**.

Метод не возвращает значений.

Метод объявлен в файле *map_create.h*.

Метод предназначен для построения полигональных линий очерка и используется для визуализации силуэта триангуляции.

3.22 ВЫЧИСЛЕНИЕ ИНЕРЦИОННЫХ ХАРАКТЕРИСТИК

Геометрическое ядро C3D вычисляет площадь поверхности, объём, положение центра масс и моменты инерции моделируемого объекта. В общем случае вычисление выполняется путем численного интегрирования. Интегрирование по объему с помощью теоремы Остроградского–Гаусса сводится к интегрированию по поверхности моделируемого объекта. Интегрирование по поверхности использует триангуляцию двумерной области определения параметров поверхности. При вычислении перечисленных характеристик модели предусмотрена возможность использования готовых данных для отдельных элементов модели.

3.22.1 Инерционные характеристики модели

Для передачи инерционных характеристик модели используется класс InertiaProperties, который приведен на рис. 1:



Рис. 1

Класс InertiaProperties объявлен в файле *mip_solid_mass_inertia.h*. Класс InertiaProperties содержит следующие данные о модели:

- double *area* – площадь поверхности,
- double *volume* – объем,
- double *mass* – масса,
- double **inertia**[3] – статические моменты в исходной системе координат,
- double **initial**[3][3] – моменты инерции в исходной системе координат,
- double **moments**[3][3] – моменты инерции в центральной системе координат,
- double **general**[3] – главные центральные моменты инерции,
- MbCartPoint3D **center** – центр масс,
- MbVector3D **direction**[3] – векторы направлений главных осей инерции.

При инициализации класса InertiaProperties все данные принимают нулевые значения, а координаты центра масс полагаются равными NOT_INITIAL_DBL. Моменты инерции **initial** вычисляются в системе координат, в которой описана модель. Моменты инерции **moments** вычисляются в системе координат, начало которой находится в точке **center**, а координатные оси совпадают с осями исходной системы координат, в которой описана модель. Моменты инерции **general** вычисляются в главной центральной системе координат, начало которой находится в точке **center**, а координатные оси получены вычислением и совпадают с главными осями инерции модели. Центробежные моменты инерции в главных системах координат отсутствуют.

Векторы **direction** дают направления главных осей инерции. Если все главные моменты инерции **general**[*i*], *i* = 1, 2, 3, разные, то все векторы **direction**[*i*], *i* = 1, 2, 3, не равны нулю. Если все главные моменты инерции **general**[*i*], *i* = 1, 2, 3, одинаковые, то все векторы **direction**[*i*], *i* = 1, 2, 3, равны нулю, и главными направлениями могут служить любые три взаимно ортогональных вектора. Если два из трех главных моментов инерции равны, например **general**[*j*] == **general**[*k*], то два из трех векторов равны нулю **direction**[*j*] = **direction**[*k*] = 0, а не равный нулю вектор **direction**[*i*] определяет направление главной оси инерции, момент относительно которой отличается от других; двумя другими главными направлениями могут служить любые два взаимно ортогональных и ортогональных не равному нулю вектору **direction**[*i*] вектора.

Возможность использования готовых данных для отдельных элементов модели позволяют реализовать классы SolidMIAttire и AssemblyMIAttire, которые приведены на рис. 2 и рис. 3. Классы объявлены в файле *mip_solid_mass_inertia.h*.



Рис. 2



Рис. 3

Класс SolidMIAttire содержит следующие данные:

- const MbSolid & **solid** – тело,
- double *density* – плотность или удельная масса на единицу площади,
- MbMatrix3D **matrix** – матрица преобразования тела в систему ближайшей сборки,
- InertiaProperties * **properties** – назначенные инерционные характеристики тела, могут быть ноль или заданы не полностью,
- bool *ready* – признак, показывающий, что характеристики не требуется считать.

Класс AssemblyMIAttire содержит следующие данные:

- RPArray< **AssemblyMIAttire** > **assemblies** – множество сборочных единиц следующего уровня,
- RPArray< **SolidMIAttire** > **solids** – тела сборочной единицы,
- MbMatrix3D **matrix** – матрица преобразования сборочной единицы в систему ближайшей сборки,
- InertiaProperties * **properties** – назначенные инерционные характеристики сборочной единицы, могут быть нулями или заданы не полностью,
- bool *ready* – признак, показывающий, что характеристики не требуется считать.

Если экземпляры классов **SolidMIAttire** и **AssemblyMIAttire** содержат не нулевые **properties** и *ready* == true, то инерционные характеристики соответствующего объекта не будут вычисляться, а результат вычисления будет содержать данные из **properties**.

Если экземпляры классов **SolidMIAttire** и **AssemblyMIAttire** содержат не нулевые **properties** и *ready* == false, то результат вычисления будет содержать данные из **properties**, которые не равны NULL_EPSILON или NOT_INITIAL_DBL. Данные, которые в **properties** равны NULL_EPSILON или NOT_INITIAL_DBL, будут вычислены. Рассматриваемые классы позволяют смешивать рассчитанные данные с назначенными.

3.22.2 Инерционные характеристики тела

Функция

```
void MassInertiaProperties( const MbSolid *             solid,
                            double                      density,
                            double                      deviateAngle,
                            InertiaProperties &         properties,
                            IfProgressIndicator * progress = 0 );
```

вычисляет площадь поверхности, объем, массу, центр масс и моменты инерции тела.

Входными параметрами функции являются:

- **solid** – тело,
- **density** – плотность или удельная масса на единицу площади,
- **deviateAngle** – параметр управления точностью расчёта.

Выходными параметрами функции являются:

- **properties** – рассчитанные инерционные характеристики,
- progress – индикатор выполнения расчета.

Функция не возвращает значения.

Функция объявлена в файле *mip_solid_mass_inertia.h*.

Для замкнутого тела **solid** параметр *density* определяет плотность тела. Для незамкнутого тела **solid** параметр *density* определяет удельную массу единицы площади тела. Вычисление в общем случае выполняется численно и использует триангуляцию области определения параметров поверхности граней. Триангуляция параметрической области граней выполняется методом CalculateGrid(), описанным в параграфе *Построение полигонов объекта*, при значениях *stepData.stepType* = ist_MipStep и *stepData.angle* = *deviateAngle*. Параметр *deviateAngle* определяет максимально допустимый угол между нормалями соседних треугольников и четырёхугольников триангуляции поверхности. Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчета. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан. Следует заметить, что при малых значениях *deviateAngle* для сложных моделей требуется большое время расчета.

Инерционные характеристики **properties** описаны в параграфе *Построение полигонов объекта*. Параметр progress выдает информацию о выполнении в процессе расчета и может использоваться для остановки вычислений.

3.22.3 Инерционные характеристики множества тел

Функция

```
void MassInertiaProperties( const RPArray<MbSolid> &             solids,
                            const SArray<double> &               densities,
                            const SArray<MbMatrix3D> &           matricies,
                            const RPArray<InertiaProperties> &   mpSolids,
                                  double                         deviateAngle,
                                  InertiaProperties &            properties,
                                  IfProgressIndicator *          progress = 0 )
```

вычисляет площадь поверхности, объем, массу, центр масс и моменты инерции множества тела.

Входными параметрами метода являются:

- **solids** – множество тел
- **densities** – множество плотностей или удельных масс на единицу площади
- **matricies** – множество матриц преобразования тел в общую систему координат
- **mpSolids** – множество имеющихся характеристик тел (может содержать нули)
- **deviateAngle** – параметр управления точностью расчёта

Выходными параметрами метода являются:

- **properties** – рассчитанные инерционные характеристики
- **progress** – индикатор выполнения расчёта

Метод не возвращает значения.

Метод объявлен в файле *mip_solid_mass_inertia.h*.

Количество элементов в множествах *densities*, **matricies**, **mpSolids** должно совпадать с количеством тел в множестве **solids**. Второй параметр определяет плотность тела для замкнутых тел или удельную массу единицы площади для незамкнутых тел. Параметр **matricies** содержит матрицы преобразования тел в систему координат, в которой требуется выполнить расчёт. Параметр **mpSolids** содержит назначенные характеристики соответствующих тел, которыми необходимо заменить соответствующие характеристики, полученные в результате расчёта. Рассматриваемый метод может использоваться для вычисления инерционных характеристик сборочной единицы, инерционные характеристики элементов которой были вычислены ранее в локальных системах координат.

Вычисление в общем случае выполняется численно и использует триангуляцию области определения параметров поверхности граней. Триангуляция параметрической области граней выполняется методом CalculateGrid(), описанным в подразделе *Построение полигонов объекта*, при значениях *stepData*.*stepType* = ist_MipStep и *stepData.angle* = *deviateAngle*. Параметр *deviateAngle* определяет максимально допустимый угол между нормалями соседних треугольников и четырёхугольников триангуляции поверхности. Параметр *deviateAngle* управляет точностью расчёта и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан. Следует заметить, что при малых значениях *deviateAngle* для сложных моделей требуется большое время расчёта.

Инерционные характеристики **properties** описаны в подразделе *Инерционные характеристики модели*. Параметр *progress* выдаёт информацию о выполнении в процессе расчёта и может использоваться для остановки вычислений.

3.22.4 Инерционные характеристики модели

Функция

```
void MassInertiaProperties( const AssemblyMIAttire &    assembly,
                            double                       deviateAngle,
                            InertiaProperties &          properties,
                            IfProgressIndicator * progress = 0 )
```

вычисляет площадь поверхности, объём, массу, центр масс и моменты инерции модели, представленной в виде сборочной единицы.

Входными параметрами метода являются:

- **assembly** – сборочная единица, которая может содержать рассчитанные ранее характеристики
- *deviateAngle* – параметр управления точностью расчёта

Выходными параметрами метода являются:

- **properties** – рассчитанные инерционные характеристики
- progress – индикатор выполнения расчёта

Метод не возвращает значения.

Метод объявлен в файле *mip_solid_mass_inertia.h*.

Параметр **assembly** представляет собой аналог сборочной единицы, элементы которой являются такими же сборочными единицами в локальных системах координат и телами с заданной плотностью в локальных системах координат. Элементы сборочной единицы могут иметь рассчитанные ранее инерционные характеристики и параметры управления ими. При наличии у элемента рассчитанных ранее характеристик последние будут использоваться в общей сумме, что сокращает общее время расчёта.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* определяет максимально допустимый угол между нормалями соседних треугольников и четырёхугольников триангуляции поверхности. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан. Следует заметить, что при малых значениях *deviateAngle* для сложных моделей требуется большое время расчёта.

Инерционные характеристики **properties** описаны в подразделе *Инерционные характеристики модели*. Параметр progress выдаёт информацию о выполнении в процессе расчёта и может использоваться для остановки вычислений.

3.22.5 Вычисление площади поверхности

Метод

```
double CalculateArea( const RPArray<MbFace> & faces,
                      double                   deviateAngle )
```

вычисляет площадь поверхности набора граней.

Входными параметрами метода являются:

- **faces** – набор граней
- *deviateAngle* – параметр управления точностью расчёта

Метод возвращает площадь набора граней.

Метод объявлен в файле *mip_solid_mass_inertial.h*.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан. Вычисление в общем случае выполняется численно и использует триангуляцию области определения параметров поверхности граней. Триангуляция параметрической области граней выполняется методом CalculateGrid(), описанным в параграфе *Построение полигонов объекта*, при значениях *stepData*.*stepType* = ist_MipStep и *stepData.angle* = *deviateAngle*. Параметр *deviateAngle* определяет максимально допустимый угол между нормалями соседних треугольников и четырёхугольников триангуляции поверхности.

Метод

```
double CalculateArea( const MbFace & face,
                      double   deviateAngle )
```

вычисляет площадь поверхности одной грани.

Входными параметрами метода являются:

- **face** – грань
- *deviateAngle* – параметр управления точностью расчёта

Метод возвращает площадь грани **face**.

Метод объявлен в файле *tri_face.h*.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан.

Метод

```
double CalculateArea( const MbSurface & surface,
                      double      deviateAngle )
```

вычисляет площадь поверхности.

Входными параметрами метода являются:

- **surface** – поверхность
- *deviateAngle* – параметр управления точностью расчёта

Метод возвращает площадь поверхности **surface**.

Метод объявлен в файле *mip_solid_area_volume.h*.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан.

Метод

```
double CalculateAreaCentre( const MbFace &          face,
                            double          deviateAngle,
                            bool            byOuter,
                            VERSION         version,
                            MbCartPoint3D &  centre )
```

вычисляет площадь поверхности и центр масс грани.

Входными параметрами метода являются:

- **face** – грань
- *deviateAngle* – параметр управления точностью расчёта
- *byOuter* – параметр игнорирования внутренних вырезов грани
- version – параметр версионирования расчёта

Выходным параметром метода является **centre** – центр масс грани.

Метод возвращает площадь грани.

Метод объявлен в файле *mip_solid_area_volume.h*.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан.

Параметр *byOuter* позволяет исключить внутренние вырезы грани при расчёте. Если *byOuter* = true, то в расчёте считается, что внутренние вырезы грани отсутствуют. Для нормального расчёта площади поверхности и центра масс грани следует положить *byOuter* = false. Параметр version необходим для поддержания старых версий расчёта.

Метод

```
double CalculateAreaCentre( const MbFaceShell &    shell,
                            double           deviateAngle,
                            MbCartPoint3D &  centre )
```

вычисляет площадь поверхности и центр масс набора граней.

Входными параметрами метода являются:

- **shell** – набор граней
- *deviateAngle* – параметр управления точностью расчёта

Выходным параметром метода является **centre** – центр масс набора граней.

Метод возвращает площадь набора граней.

Метод объявлен в файле *mip_solid_area_volume.h*.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан.

3.22.6 Вычисление объёма тела

Метод

```
double CalculateVolume( const MbSolid & solid,
                        double    deviateAngle )
```

вычисляет объём тела.

Входными параметрами метода являются:

- **solid** – тело
- **deviateAngle** – параметр управления точностью расчёта

Метод возвращает объём тела.

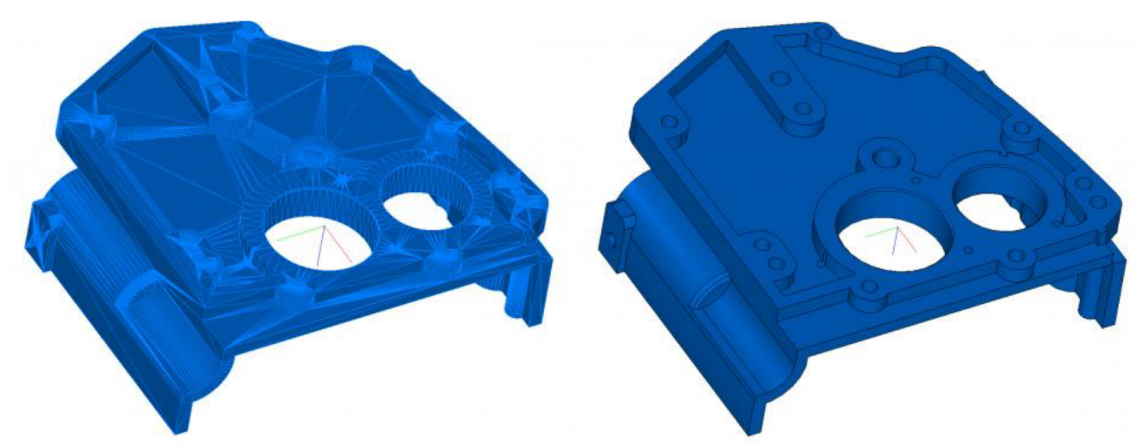
Метод объявлен в файле *mip_solid_area_volume.h*.

Параметр *deviateAngle* управляет точностью расчёта, и от его значения зависит время расчёта. Параметр *deviateAngle* должен находиться в пределах 0.01 <= *deviateAngle* <= 0.35 радиан.

ГЛАВА 4

C3D B-Shaper

Модуль C3D B-Shaper преобразует полигональные модели в твердотельные с граничным представлением (B-Rep). В результате выполнения алгоритмов пользователь может редактировать полученную модель привычными инструментами CAD-системы, значительно уменьшив сложность и объем данных.



Области применения C3D B-Shaper:

- работа в CAD и BIM-системах с полигональными моделями из каталогов готовых и типовых моделей деталей, элементов зданий и сооружений;
- распознавание поверхностей в полигональных моделях, полученных в результате 3D-сканирования или топологической оптимизации в CAE-системах;
- алгоритмы сглаживания сетки, децимации и сжатия в компьютерной графике.



Уникальный алгоритм модуля C3D B-Shaper разбивает полигональную сетку на сегменты — прообразы предполагаемых граней. После этого выделенные области могут быть распознаны как элементарные поверхности (плоскость, цилиндр, конус, сфера, тор), поверхность вращения или NURBS-поверхность. Между соседними сегментами вычисляются кривые пересечения, на основе которых в дальнейшем строятся ребра граней тела.

Работа с модулем ведется через API. Предусмотрены два варианта его использования: полностью автоматический и интерактивный. Необходимо отметить, что работа в автоматическом режиме распознавания оболочки в общем случае не подходит для моделей, полученных в результате 3D-сканирования или топологической оптимизации. Данный режим предназначен для применения на полигональных моделях, имеющих в своей основе некоторые модели, созданные в CAD-приложениях.

Для активации лицензии на модуль необходимо вызвать функцию EnableMathModules() с соответствующими параметрами.

4.1 Автоматический режим распознавания оболочки по полигональной сетке

Интерфейс автоматического преобразования C3D B-Shaper представлен двумя функциями: ConvertMeshToShell() и ConvertCollectionToShell().

Метод

```
MbResultType ConvertMeshToShell(       MbMesh &                mesh,
                                       MbFaceShell *&          shell,
                                 const MbMeshProcessorValues & params =␣
↪MbMeshProcessorValues() );
```

выполняет создание оболочки в граничном представлении (B-Rep), соответствующей модели, заданной полигональной сеткой. Алгоритм модуля в автоматическом режиме распознаёт и реконструирует грани, соответствующие элементарным поверхностям (плоскость, цилиндр, конус, сфера, тор), поверхностям вращения или NURBS поверхностям.

Входными параметрами метода являются:

- *mesh* – полигональная сетка модели;
- *params* – настройки преобразования.

Выходным параметром метода является построенная оболочка *shell*.

При удачной работе алгоритма метод возвращает значение *rt_Success*, в противном случае метод возвращает код ошибки из перечисления MbResultType.

Метод объявлен в файле *action_b_shaper.h*.

К настройкам преобразования относится значение точности распознавания, то есть максимально допустимого отстояния вершин полигональной сетки в границах данного сегмента до распознаваемой поверхности. Эта точность может быть абсолютной или относительной: при использовании относительной точности отклонение граней тела от сетки проверяется относительно размера модели. Также пользователю доступна опция переключения режимов распознавания, которая позволяет управлять типами поверхностей при реконструкции. Эти параметры находятся в полях класса MbMeshProcessorValues, который содержит:

- *useRelativeTolerance* – флаг использования относительной точности;
- *tolerance* – точность распознавания;
- *surfReconstructMode* – режим распознавания поверхностей, из перечисления:
  - *srm_All* – строить все поверхности;
  - *srm_NoGrids* – не строить поверхности на базе триангуляции;
  - *srm_CanonicOnly* – строить только элементарные поверхности;
  - *srm_Default* – по умолчанию.

Метод

```
MbResultType ConvertCollectionToShell(       MbCollection &           collection,
                                             MbFaceShell *&           shell,
                                       const MbMeshProcessorValues & params =␣
↪MbMeshProcessorValues() );
```

выполняет создание оболочки в граничном представлении (B-Rep), соответствующей коллекции элементов, содержащих полигональную сетку. Алгоритм модуля в автоматическом режиме распознаёт и реконструирует грани, соответствующие элементарным поверхностям (плоскость, цилиндр, конус, сфера, тор).

Входными параметрами метода являются:

- *collection* – коллекция элементов, содержащая полигональную сетку;
- *params* – настройки преобразования.

Выходным параметром метода является построенная оболочка *shell*.

При удачной работе алгоритма метод возвращает значение *rt_Success*, в противном случае метод возвращает код ошибки из перечисления MbResultType.

Метод объявлен в файле *action_b_shaper.h*.

Тестовое приложение выполняет построение оболочки в граничном представлении командой меню «Создать→Тело→На базе полигональной модели→Преобразовать в тело с распознаванием поверхностей».

4.2 Класс MbMeshProcessor – создание оболочки

Класс MbMeshProcessor предназначен для создания оболочки по полигональной сетке с пользовательскими настройками. Класс предоставляет расширенные возможности по управлению процессами сегментации и распознавания поверхностей.

```
class MbMeshProcessor {

public:
    // Создание экземпляра процессора по коллекции элементов.
    MbMeshProcessor * Create( const MbCollection & collection );

    // Деструктор
    ~MbMeshProcessor();

    // Управление точностью распознавания
    void   SetRelativeTolerance( double tolerance );
    void   SetTolerance( double tolerance );
    double GetTolerance() const;

    // "Лечение" сетки.
    void SetUseMeshSmoothing( bool useSmoothing );

    // Управление сегментацией сетки.
    const MbCollection & GetSegmentedMesh();
    MbResultType         SegmentMesh( bool createSurfaces = true );
    void                 ResetSegmentation();
    void                 UniteSegments( size_t firstSegmentIdx,
                                        size_t secondSegmentIdx );
    MbResultType         SegmentMeshBySeparators( const std::vector<std::vector<uint>>
↪ & sep );

    // Управление распознаванием поверхностей.
    void             SetReconstructionMode( MbeSurfReconstructMode mode );
    void             FitSurfaceToSegment( size_t idxSegment );
    void             FitSurfaceToSegment( size_t idxSegment, MbeSpaceType surfaceType␣
↪);
    const MbSurface * GetSegmentSurface( size_t idxSegment ) const;

    // Построение B-rep модели.
    MbResultType CreateBRepShell( MbFaceShell *& pShell );

    ..
};
```

Описание класса находится в файле *action_b_shaper.h*.

Далее будут рассмотрены более подробные описания методов данного класса.

4.3 Управление точностью распознавания

Одной из ключевых настроек преобразования полигональной сетки в модель с граничным представлением является значение точности распознавания. Точность работы метода определяется максимальным допустимым отклонением распознанных поверхностей от вершин полигональной сетки. Пользователь может задать требуемую в расчётах точность, или будет использоваться значение по умолчанию. В случае, когда параметр отклонения граней от сетки заранее не известен, можно использовать относительную точность, которая будет рассчитана исходя из размеров исходного тела. Таким образом, в случае неудовлетворительного результата действия алгоритма пользователь может влиять на него, изменяя параметр точности распознавания.

Для управления значением точности используются несколько функций класса MbMeshProcessor.

Метод

```
void SetRelativeTolerance( double tolerance );
```

устанавливает значение относительной точности. При использовании относительной точности отклонение граней тела от сетки проверяется относительно габаритов текущей сетки. Входным параметром метода является:

- *tolerance* – значение относительной точности.

Для получения абсолютной погрешности распознавания значение *tolerance* будет умножено на диагональ габаритного куба полигональной сетки или коллекции элементов модели. Таким образом, при заданной относительной погрешности, равной 1.0, значение абсолютной погрешности будет посчитано, исходя из размеров модели.

Метод

```
void SetTolerance( double tolerance );
```

устанавливает абсолютную точность распознавания поверхностей и расширения сегментов сетки. Входным параметром метода является:

- *tolerance* – значение абсолютной погрешности.

Данный метод должен быть вызван перед вызовом метода SegmentMesh(). По умолчанию абсолютная точность равна 0.1.

Метод

```
double GetTolerance() const;
```

передаёт значение текущей абсолютной точности, используемой при распознавании поверхностей и расширения сегментов сетки. Метод возвращает абсолютную погрешность значением типа *double*.

4.4 Редактирование сегментации полигональной сетки

Часто полигональные сетки моделей, являясь результатом 3D-сканирования, имеют сложное внутреннее устройство с наличием «шумов» – случайных выбросов точек за пределы габаритов грани. Преобразование таких сеток предполагает более плотное интерактивное взаимодействие с пользователем, так как автоматическое распознавание затруднено. Для коррекции результатов сегментации предусмотрены несколько инструментов.

Метод

```
MbResultType SegmentMesh( bool createSurfaces = true );
```

выполняет сегментирование заданной полигональной сетки. Входным параметром метода является:

- *createSurfaces* – флаг создания поверхностей на сегментах.

При удачной работе алгоритма метод возвращает значение *rt_Success*, в противном случае метод возвращает код ошибки из перечисления MbResultType.

Метод :

```
void UniteSegments( size_t firstSegmentIdx,
                    size_t secondSegmentIdx );
```

объединяет два сегмента в текущей сегментации полигональной сетки. Входными параметрами метода являются:

- *firstSegmentIdx* – индекс первого сегмента для объединения;
- *secondSegmentIdx* – индекс второго сегмента для объединения.

Результат объединения доступен через коллекцию, возвращаемую методом GetSegmentedMesh().

Метод

```
MbResultType SegmentMeshBySeparators( const std::vector<std::vector<uint>> &␣
↪separators );
```

выполняет сегментацию полигональной сетки по разделителям сегментов. Входным параметром метода является:

- *separators* – массив разделителей, каждый из которых содержит путь по вершинам сетки, рёбра которого разделяют сегменты.

При удачной работе алгоритма метод возвращает значение *rt_Success*, в противном случае метод возвращает код ошибки из перечисления MbResultType.

4.5 Реконструкция поверхности на сегменте

При использовании C3D B-Shaper разработчикам доступна реконструкция поверхности определённого типа на сегменте. Для управления распознаванием поверхностей используются следующие методы.

Метод

```
void FitSurfaceToSegment( size_t idxSegment );
```

распознаёт поверхность по сегменту сетки с заданным индексом и вписывает её в сегментацию. Распознанная поверхность может быть получена с помощью вызова метода GetSegmentSurface(). Входной параметр *idxSegment* передаёт индекс сегмента полигональной сетки.

Метод с указанием типа поверхности

```
void FitSurfaceToSegment( size_t    idxSegment,
                          MbeSpaceType surfaceType );
```

выполняет построение поверхности заданного типа, аппроксимирующей сегмент сетки с заданным индексом. Распознанная поверхность может быть получена с помощью вызова метода GetSegmentSurface(). Входной параметр *idxSegment* передаёт индекс сегмента полигональной сетки.

ГЛАВА 5

C3D PolyShaper

Модуль C3D PolyShaper предназначен для редактирования, анализа и устранения дефектов полигональных моделей (сеток), а также для их преобразования в гладкие поверхности.

Полигональное представление — это описание геометрии совокупностью вершин, ребер и плоских фасетов, объединенных в многогранный трехмерный объект. Полигональное моделирование применяется в реверс-инжиниринге, топологической оптимизации, при проведении инженерных расчетов, в медицине, при подготовке моделей к 3D-печати.

В основе модуля лежит топологическое представление сетки в виде двусвязного списка рёбер, или Halfedge структуры. Логическое разбиение сетки на части, или сегментация, поддерживается тоже в виде Halfedge и по сути является аналогом граничного представления тел. Работа с модулем осуществляется через API. API включает в себя классы MbPolymesh и MbPolymeshTopo, являющиеся представлением полигонального объекта, и набор функций и классов для работы с ними.

Объект MbPolymesh может быть сконструирован вручную, получен из объекта MbMesh или из объекта MbSolid путём линеаризации. Также возможно обратное преобразование в MbMesh (с потерей топологических данных). Обратное преобразование нужно в случае, если отрисовка реализована через объект MbMesh. Большая часть функционала модуля предназначена для работы с MbPolymesh, но некоторые функции доступны также для MbMesh.

Для активации лицензии на модуль необходимо вызвать функцию EnableMathModules() с соответствующими параметрами.

5.1 Особенности реализации двусвязного списка рёбер

Структура Halfedge имеет давнюю историю и является одним из самых удобных способов представления топологии сеток. В модуле C3D PolyShaper она реализована классическим способом с некоторыми особенностями. Реализацией является класс MbHalfedge, который находится в заголовочном файле *halfedge.h*. Это просто интерфейс для топологии сетки, который предоставляется для пользователя классом MbPolymeshTopo.

Контейнер MbHalfedge содержит массив рёбер MbHalfedgeEdge, массив вершин MbHalfedgeVertex, массив фасетов MbHalfedgeFacet и массив атрибутов сетки MbHalfedgeAttrBase (не путать с MbAttribute).

Ребро имеет ссылки на следующее ребро, на предыдущее ребро, на ребро-близнец, на фасет, которому ребро принадлежит, и на вершину, из которой оно исходит (как показано на рис. 1). Если ребро является граничным, то ссылка на ребро-близнец не определена.

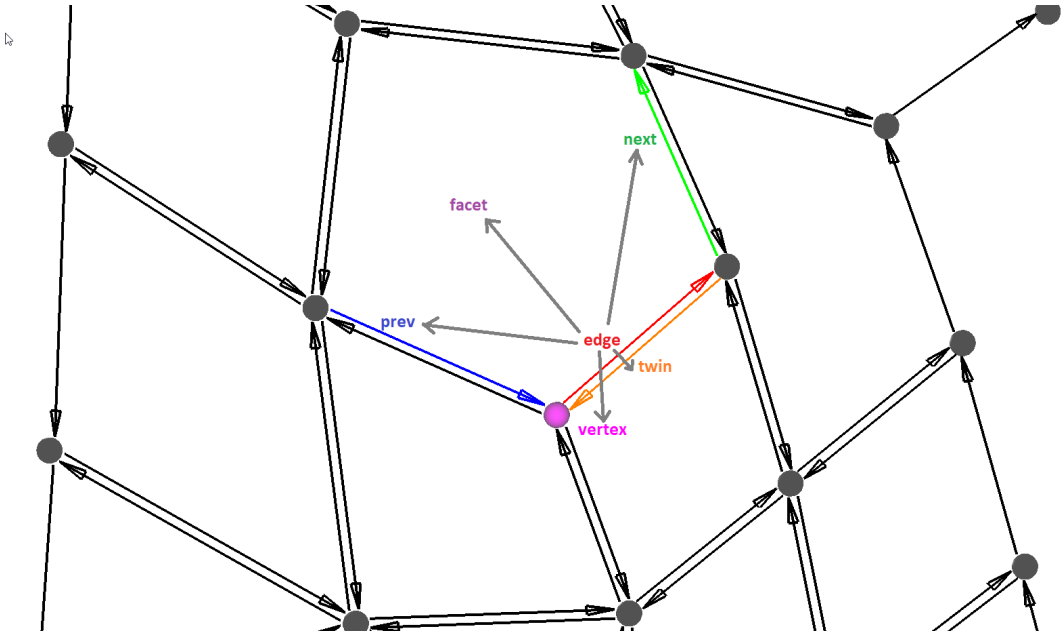


Рис. 1

Вершина имеет ссылку на одно из рёбер, которые из неё исходят (как показано на рис. 2). Для внутренней вершины это может быть любое ребро из веера. Для граничной это должно быть обязательно граничное ребро (такое ребро единственное в веере).

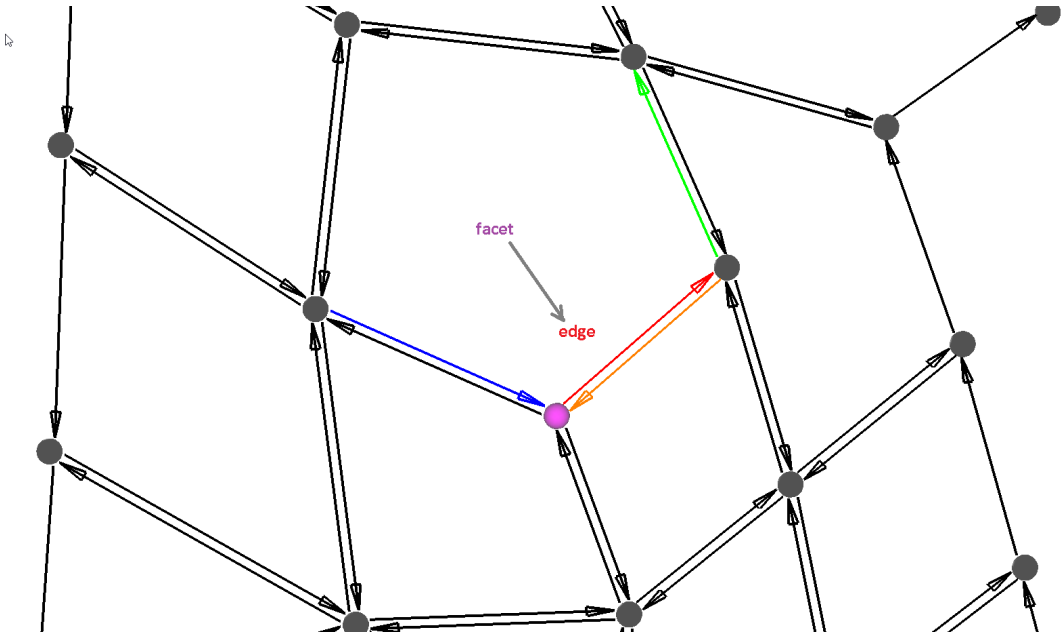


Рис. 2

Фасет имеет ссылку на одно любое из рёбер, которые его образуют (как показано на рис. 3). Рёбра следуют друг за другом в направлении против часовой стрелки. Внутренних циклов для фасета не предусмотрено. Фасет может состоять из любого количества рёбер.

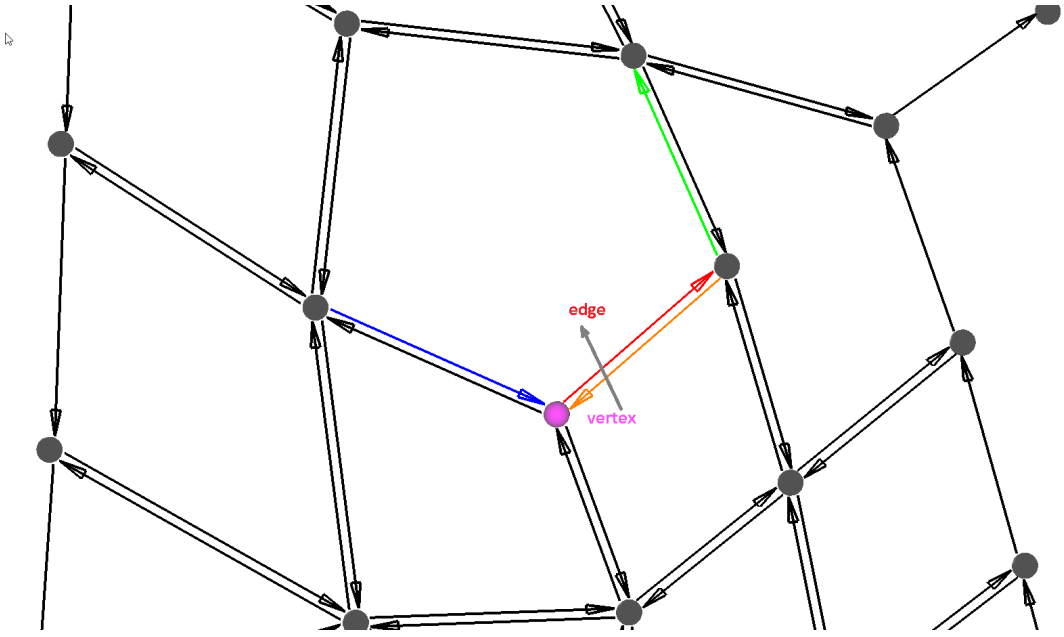


Рис. 3

Любой элемент сетки может быть в активном и неактивном состояниях. Например, при удалении фасета из сетки он не удаляется из массива фасетов, а переводится в неактивное состояние, а размер массива не изменяется. При добавлении фасета сначала делается проверка наличия неактивных фасетов. Если неактивные фасеты есть, то один из них

становится новым. Если же неактивных фасетов нет, то массив фасетов увеличивается на соответствующее количество элементов.

Совокупность элементов сетки образует исключительно топологию сетки. Учёт геометрических или иных свойств сетки осуществляется через добавление в сетку соответствующего атрибута – наследника класса MbHalfedgeAttrBase. Ссылки элементов сетки друг на друга сделаны не через указатели, а через индексы. Это позволяет ассоциировать с элементами сетки целые числа, вещественные числа, 3D или 2D точки, векторы или более сложные структуры. Доступные для пользователя типы атрибутов находятся в заголовочных файлах *mesh_halfedge_attr.h* и *mesh_segm_halfedge_attr.h*. Возможность создания пользовательских атрибутов не предусмотрена.

Например, для работы с 3D точками в сетке должен существовать атрибут типа MbHalfedgeAttrPoint3D. И уже его API позволяет получить точку, соответствующую вершине сетки, нормаль или площадь треугольника и так далее.

5.2 Сеточные итераторы

Для работы с топологией сетки предусмотрено несколько типов итераторов. Они находятся в заголовочных файлах *halfedge_iterators.h* и *halfedge_range_iterators.h*:

1. EdgeFanIterator – итератор для обхода рёбер веера вершины в направлении против часовой стрелки (см. рис. 1). Пример использования:

```
for( EdgeFanIterator it( halfedge, iEd ); !it.Done(); ++it );

или::

for( EdgeFanIterator it( halfedge, iVx, true ); !it.Done(); ++it ) {

    if( it.Edge() < SYS_MAX_T ) {
        ...
    }
    if( it.Twin() < SYS_MAX_T ) {
        ...
    }
}
```

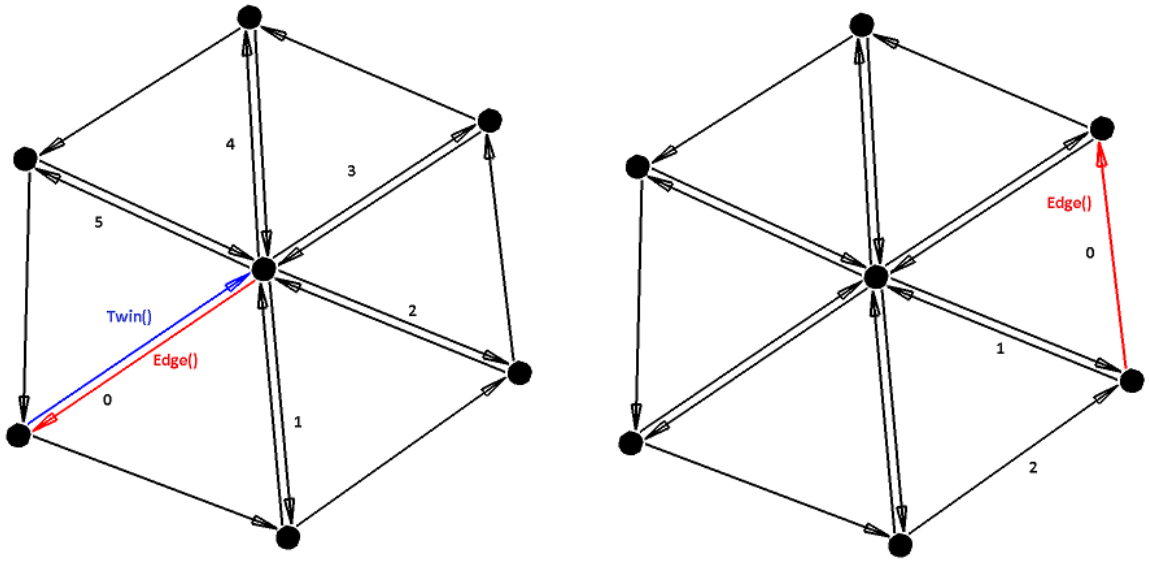


Рис. 1

2. EdgeFacetIterator – итератор для обхода рёбер фасета, начиная с ребра фасета, в направлении против часовой стрелки (см. рис. 2).

Пример использования:

```
for( EdgeFacetIterator it( halfedge, iFacet ); !it.Done(); ++it ) {

    const MbHalfedgeEdge & ed = halfedge.Edge( it.Edge() );
}
```

3. EdgesAroundFacetBegEndPair AllEdgesAroundFacet(*size_t* iFacet) const;

Диапазон для обхода рёбер фасета, начиная с ребра фасета в направлении против часовой стрелки (см. рис. 2). Пример использования:

```
for( const MbHalfedgeEdge & ed : halfedge.AllEdgesAroundFacet( iFacet ) ) {
    ...
}
```

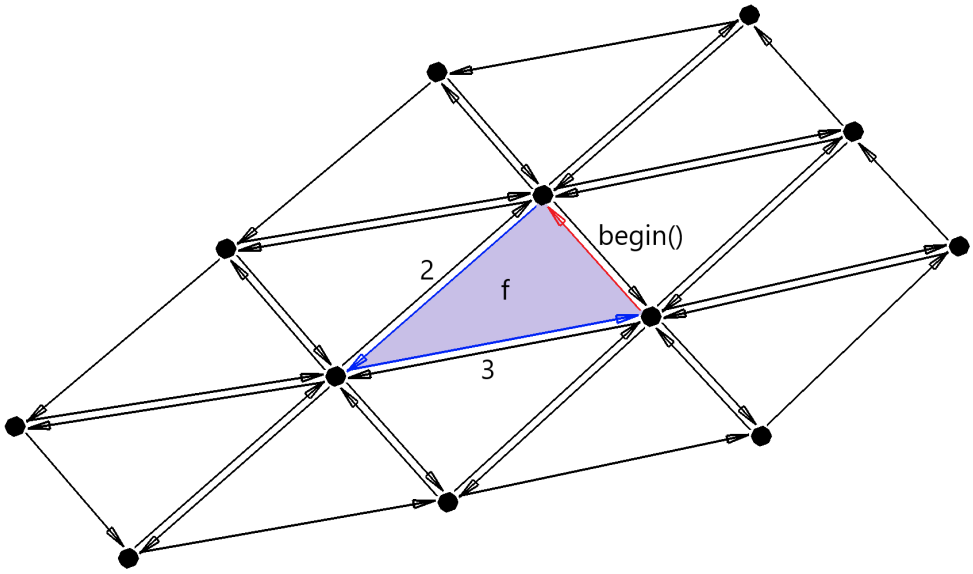


Рис. 2

4. VerticesAroundFacetBegEndPair AllVerticesAroundFacet(*size_t* iF) const;

Диапазон для обхода всех вершин вокруг фасета, начиная с вершины, принадлежащей ребру, определяющему фасет, против часовой стрелки (см. рис. 3). Пример использования:

```
for( const MbHalfedgeVertex & v : halfedge.AllVerticesAroundFacet( iFacet ) ) {
    ...
}
```

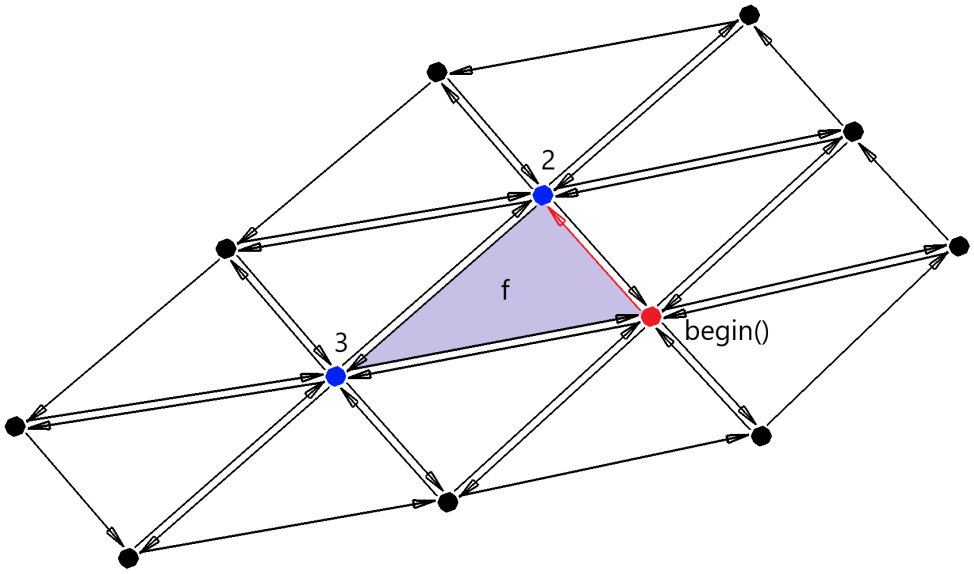


Рис. 3

5. EdgesAroundVertexBegEndPair AllEdgesAroundVertex(*size_t* iV) const;

Диапазон для обхода всех рёбер (входящих и исходящих), смежных с вершиной, начиная с ребра, определяющего вершину (см. рис. 4). Порядок обхода – против часовой стрелки, с чередованием входящих и исходящих рёбер.

Пример использования:

```
for( const MbHalfedgeEdge & ed : halfedge.AllEdgesAroundVertex( iVx ) ) {
    ...
}
```

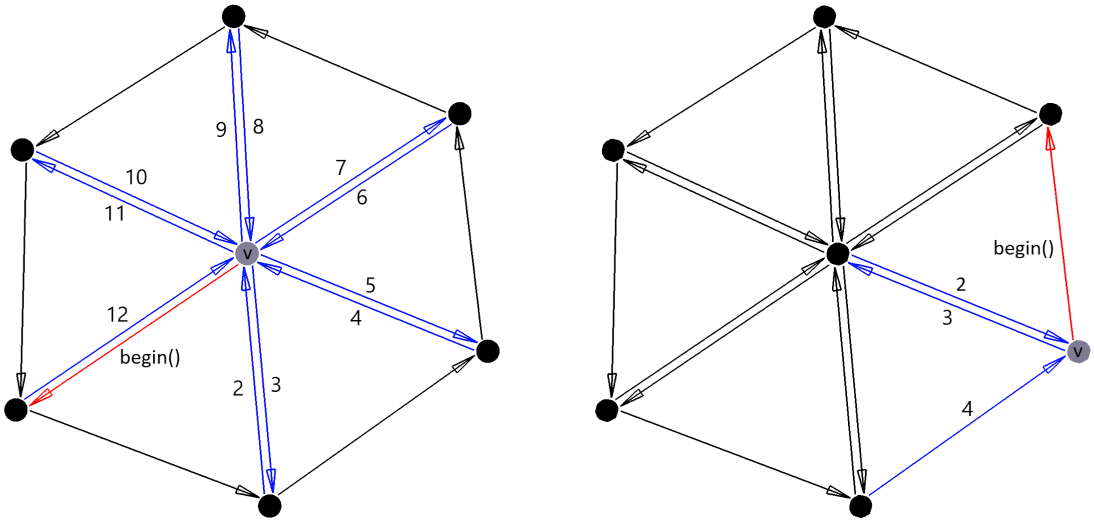


Рис. 4

6. VerticesAroundVertexBegEndPair AllVerticesAroundVertex(*size_t* iV) const;

Диапазон для обхода всех вершин, смежных с данной вершиной (см. рис. 5). Порядок обхода – против часовой стрелки, первая вершина смежна по ребру, определяющему вершину. Пример использования:

```
for( const MbHalfedgeVertex & v : halfedge.AllVerticesAroundVertex( iVx ) ) {
    ...
}
```

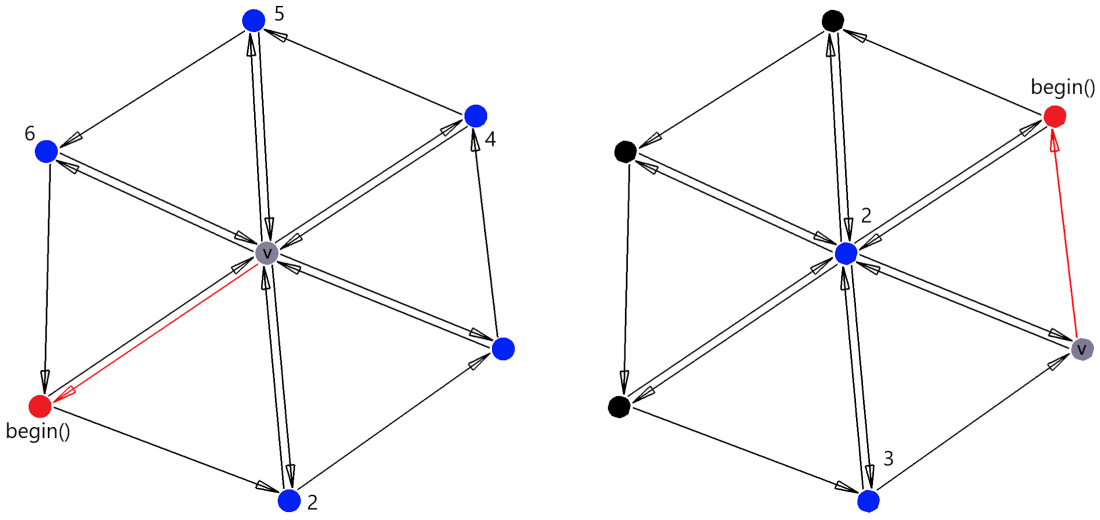


Рис. 5

7. FacetsAroundVertexBegEndPair AllFacetsAroundVertex(*size_t* iV) const;

Диапазон для обхода всех фасетов, включающих данную вершину (см. рис. 6). Порядок обхода – против часовой стрелки, первый фасет включает ребро, определяющее вершину. Пример использования:

```
for( const MbHalfedgeFacet & f : halfedge.AllFacetsAroundVertex( iVx ) ) {
    ...
}
```

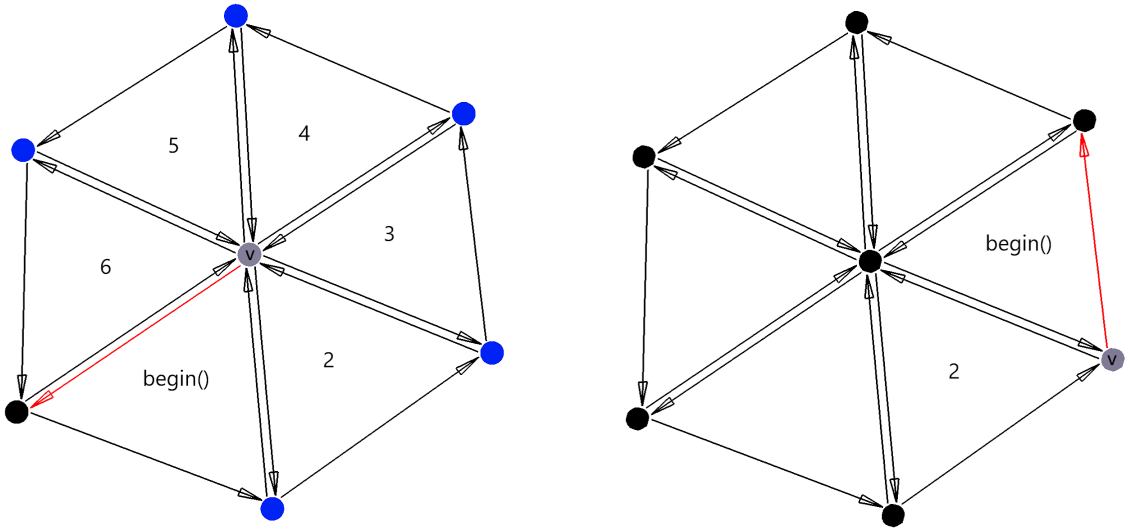


Рис. 6

5.3 Сегментация и сегментационные итераторы

Сегментация над полигональной сеткой также организована в виде Halfedge структуры. Для её представления используются следующие элементы, которые находятся в заголовочном файле *halfedge_element.h*. Каждый из сегментационных элементов может быть в активном и неактивном статусе.

Подрегион (MbSegmentationSubRegion) – связная группа фасетов.

Регион – группа из нескольких подрегионов, как правило, из одного. Отдельного класса, описывающего регион, нет.

Ребро сегментации – последовательность рёбер сетки, такая, что каждое следующее ребро исходит из вершины, в которую приходит предыдущее ребро. Ребро сегментации может быть замкнутым. Самопересечения рёбер не допускаются. Аналогично ребру сетки ребро сегментации имеет ссылки на следующее ребро сегментации, предыдущее ребро сегментации, ребро сегментации-близнеца и вершину, за счёт чего формируются циклы. Если ребро сегментации принадлежит открытой границе, ссылка на близнеца отсутствует. Сеточные рёбра, входящие в ребро сегментации и его близнеца, тоже являются близнецами (рис. 1):

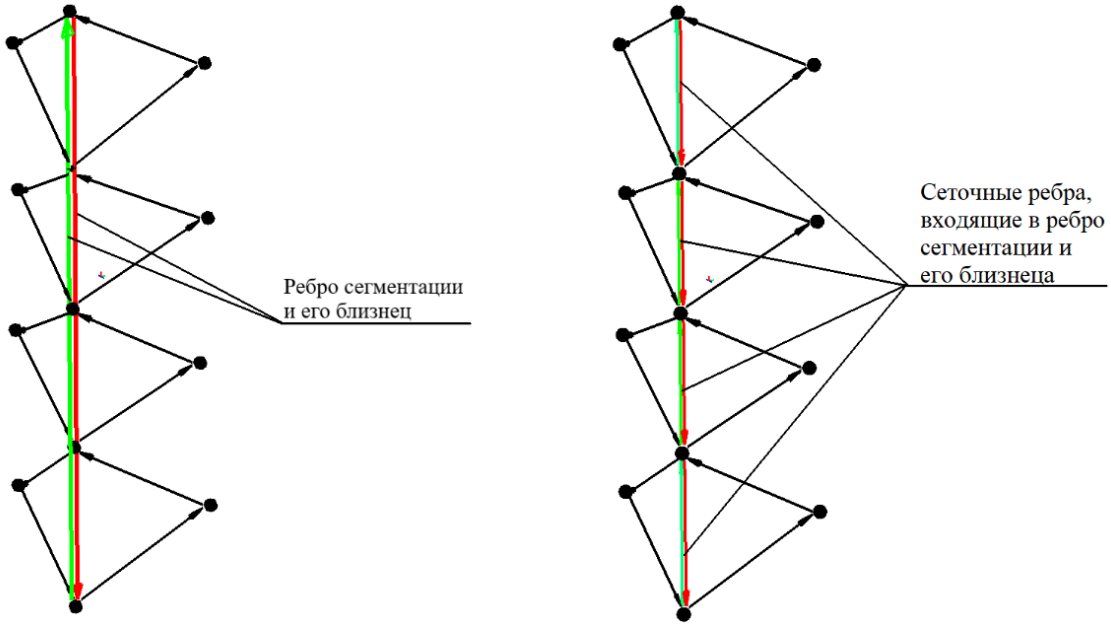


Рис. 1

Вершина сегментации – соответствует вершине сетки, в которой сходятся несколько рёбер сегментации.

Цикл подрегиона (MbSegmentationLoop), состоящий из последовательности рёбер сегментации. Подрегион может иметь сколько угодно циклов или вообще ни одного, если он представляет из себя замкнутую группу фасетов. Имеет такой же смысл, как цикл грани в граничном представлении, с некоторыми отличиями. Как правило, на сетке нельзя разделить циклы на внешний и внутренние, поэтому все циклы равноправны. Также цикл может иметь «внутренние» рёбра.

Пример разделения сетки на четыре региона, каждый из которых состоит из одного подрегиона, показан на рис. 2:



Рис. 2

Пример для той же сетки, но теперь в сегментации два региона, один из которых состоит из одного подрегиона, а другой из трёх, показан на рис. 3:

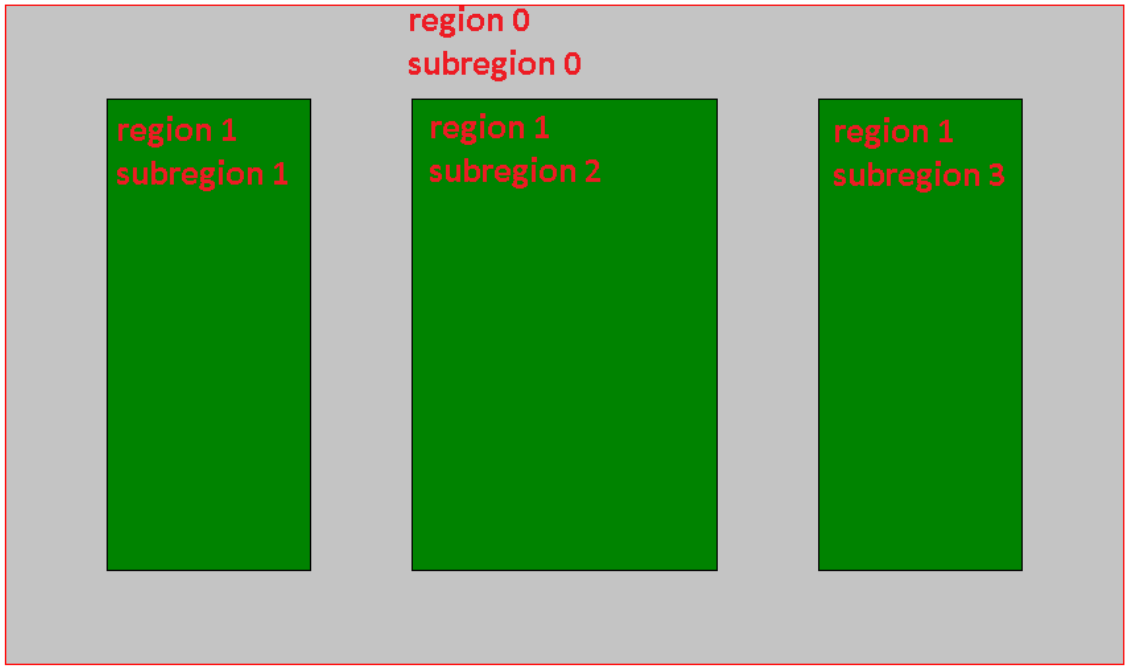


Рис. 3

Пример подрегиона с двумя циклами показан на рис. 4. Зелёный цикл полностью состоит из внутренних рёбер, в красном внутренними являются рёбра 1, 2, 4–9.

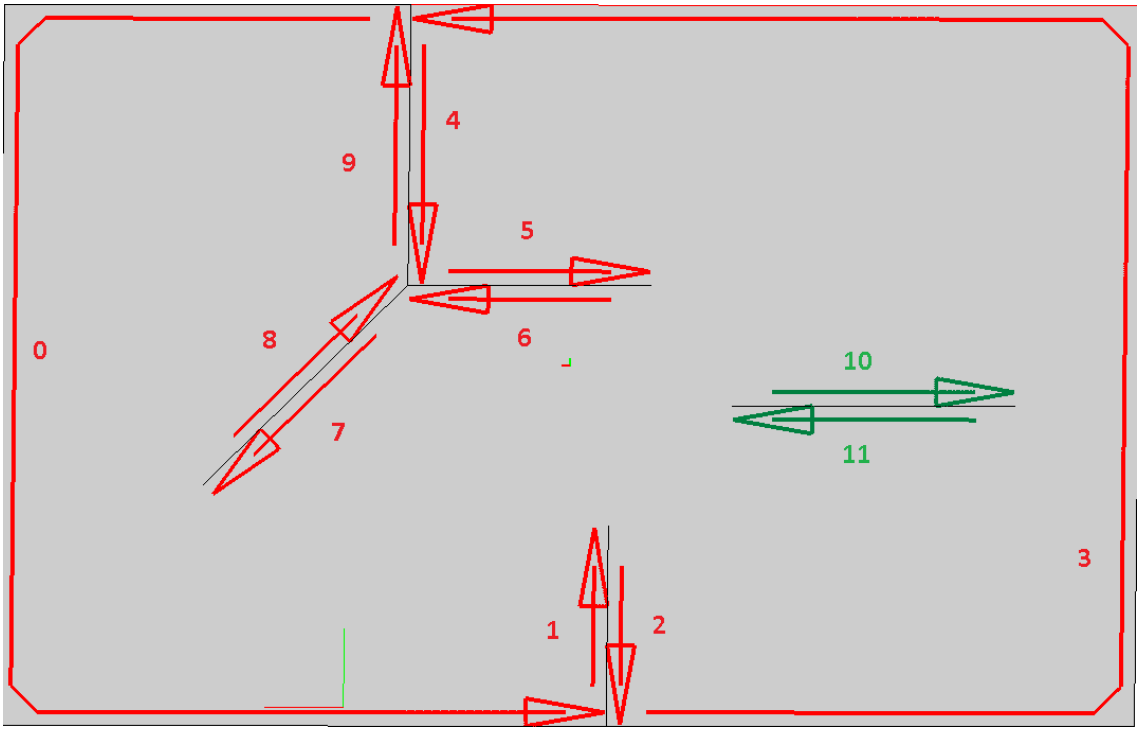


Рис. 4

Для работы с топологией сегментации предназначены сегментационные итераторы, которые находятся в заголовочном файле *halfedge_segm_iterators.h*:

- EdgeLoopIterator – итератор по рёбрам цикла подрегиона. В зависимости от флага bExcludeInnerEdges возвращает последовательность всех рёбер цикла либо только внешних.
- EdgeEdgesIterator – итератор по сеточным рёбрам ребра сегментации. В зависимости от флага bReverse возвращает последовательность сеточных рёбер в прямом либо обратном порядке.
- SubregionRegionIterator – итератор по подрегионам региона.
- LoopSubregionIterator – итератор по циклам подрегиона.

Для итерации по вееру вершин сегментации нужно использовать итератор EdgeFanIterator, как показано в примере ниже.

Примеры использования сегментационных итераторов:

```
MbPolymesh polymesh;

const MbPolymeshTopo & topo = *polymesh->GetTopo();

const MbHalfedge & halfedge = topo.GetMeshTopo();

for( size_t r = 0; r < topo.RegionCount(); ++r ) {

    if( !topo.IsRegionActive( r ) )
        continue;
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
    for( SubregionRegionIterator itSub( topo, r ); !itSub.Done(); ++itSub ) {

        for( LoopSubregionIterator itLoop( topo, itSub.Subregion() ); !itLoop.Done();␣
↪++itLoop ) {

            for( EdgeLoopIterator itEdge( topo, itLoop.Loop() ); !itEdge.Done();␣
↪++itEdge ) {

                for( EdgeEdgesIterator it( topo, itEdge.Edge() ); !it.Done(); ++it ) {

                    const MbHalfedgeEdge & ed = halfedge.Edge( it.Edge() );
                    ...
                }
            }
        }
    }
},

for( EdgeFanIterator it( topo.GetSegmTopo(), iVxSegm, true ); !it.Done(); ++it ) {

    const MbHalfedgeEdge & edSegm = topo.GetSegmTopo().Edge( it.Edge() );
}
```

5.4 Топология полигонального объекта MbPolymeshTopo

Полигональный объект *MbPolymesh* является наследником *MbItem* и содержит внутри топологию MbPolymeshTopo (по аналогии с *MbSolid*, который содержит внутри *MbFaceShell*). Топология включает в себя все методы для работы как с сеткой, так и с сегментацией. Далее будут рассмотрены более подробные описания методов данного класса.

5.4.1 Методы общего назначения

Метод

```
static MbPolymeshTopo * Create();
```

создаёт пустой экземпляр класса.

Метод

```
static MbPolymeshTopo * Create( const MbMesh & mesh,
                                double   toleranceSew );
```

создаёт экземпляр класса по экземпляру MbMesh. Точность сшивки *toleranceSew* используется для объединения почти совпадающих вершин в одну.

Метод

```
bool GetMesh( MbMesh & mesh ) const;
```

создаёт экземпляр MbMesh с потерей топологической информации, где каждому подрегиону соответствует отдельный объект MbGrid. При этом внутри этого экземпляра создаётся (в виде MapPolymeshToMbMesh) отображение между индексами MbGrid и индексами подрегионов, так как в общем случае они будут иметь различную нумерацию.

Метод

```
MbPolymeshTopo * Clone() const;
```

создаёт копию топологии.

Метод

```
void CalculateGabarit( const MbMatrix3D & matrInto,
                             MbCube &     bb ) const;
```

рассчитывает габарит объекта.

Метод

```
void Invert();
```

инвертирует всю топологию, включая разворот циклов вокруг фасетов.

Методы трансформации

```
void Transform( const MbMatrix3D & matr );
void Move     ( const MbVector3D & to   );
void Rotate   ( const MbAxis3D   & axis, double angle );
```

трансформируют все составляющие полигонального объекта, для которых эта операция имеет смысл (3D точки, нормали, направления кривизны и т.д.).

Метод

```
void DeleteInactive();
```

удаляет все неактивные элементы, уменьшая тем самым размер полигонального объекта (рёбра, вершины, фасеты сетки, рёбра сегментации, вершины сегментации, регионы, подрегионы и циклы сегментации).

Метод

```
double FindNearest( const MbCartPoint3D & pnt,
                          MbCartPoint3D & pntProj,
                          size_t &        iFacet ) const;
```

находит ближайшую к указанной точку и индекс фасета, на который она попала. Возвращает расстояние от указанной точки до ближайшей на полигональном объекте.

Метод

```
bool FindLineIntersection( const MbAxis3D &      ray,
                                 MbCartPoint3D & pntCross,
                                 double &        tRes,
                                 size_t &        iFacet ) const;
```

находит ближайшее пересечение с прямой. Выходными параметрами являются точка пересечения, параметр луча и индекс фасета, с которым это пересечение произошло.

Метод

```
const MbFaceShell * GetSubstrateShell() const;
```

обеспечивает доступ к оболочке-подложке (если она существует), из которой полигональный объект был создан.

Метод

```
void DeleteSubstrateShell();
```

удаляет оболочку-подложку, из которой полигональный объект был создан, а также все сохранённые данные, обеспечивающие отображение от регионов и рёбер сегментации к граням и рёбрам оболочки.

5.4.2 Методы для работы с сеткой

Метод

```
const MbHalfedge & GetMeshTopo() const;
```

возвращает топологию сетки в виде объекта MbHalfedge. Это нужно, например, для конструирования сеточных итераторов.

Метод

```
void Reserve( size_t nVx,
              size_t nEd,
              size_t nF );
```

резервирует память под вершины, рёбра и фасеты для сетки и всех атрибутов, которые она содержит.

Метод

```
size_t CloneVertex( size_t iVx );
```

клонирует вершину сетки и возвращает индекс склонированной вершины.

Методы добавления элементов

```
size_t AddVertex();
size_t AddEdge();
size_t AddFacet();
```

добавляют активный элемент сетки и возвращают его индекс. Если в сетке есть неактивные элементы, один из них переводится в активное состояние и возвращается его индекс.

Методы деактивации элементов

```
void MakeVertexInactive( size_t iVx );
void MakeEdgeInactive   ( size_t iEd );
void MakeFacetInactive  ( size_t iF  );
```

переводят элемент сетки в неактивное состояние. Использование должно быть крайне осторожным, так как без дополнительных действий эти методы приведут к невалидности топологии сетки.

Метод

```
void SewEdges( size_t iEd0,
               size_t iEd1 );
```

делает два указанных рёбра сетки близнецами.

Метод

```
size_t AddFacet( const c3d::IndicesVector & vertices );
```

конструирует структуру фасета по указанным вершинам и возвращает его индекс. Вершины на момент вызова должны уже существовать. Метод создаёт фасет, все необходимые рёбра и устанавливает между ними (кроме близнецов) все нужные ссылки.

Метод

```
void SewByVertices();
```

устанавливает ссылки на близнецов для всех рёбер сетки, используя вершины, из которых исходят рёбра. Используется для окончательного формирования топологии при ручном конструировании, после того как все нужные фасеты были добавлены методом AddFacet().

Метод

```
void DeleteFacets( const c3d::IndicesVector & facets );
```

удаляет указанные фасеты и обеспечивает валидность топологии после этой операции. Удаление неактивных элементов не производится. Пример действий, необходимых для поддержания валидности топологии, показан на рис. 1: в исходном состоянии вершина A одна и та же для всех рёбер веера. После удаления выделенных фасетов веер оказывается разрывным, что не допускается в структуре Halfedge. Поэтому вершина будет склонирована, и части оставшегося веера будет соответствовать вершина A, а другой части её клон B. В этом месте топологически будет разрыв сетки, хотя геометрически эти две вершины совпадают.

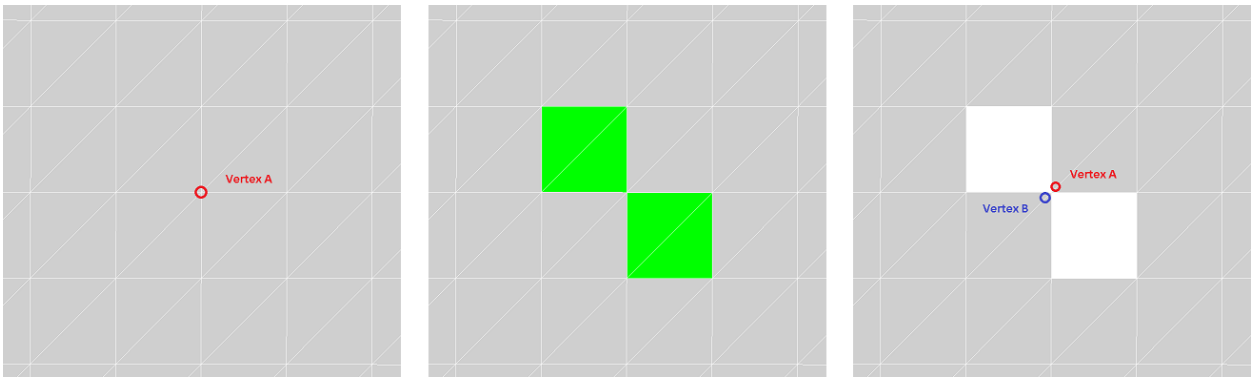


Рис. 1

Метод

```
bool FlipEdge( size_t iEd );
```

делает переброску ребра в триангуляции. Нельзя перебрасывать открытые рёбра и рёбра, принадлежащие рёбрам сегментации. Пример работы метода показан на рис. 2:

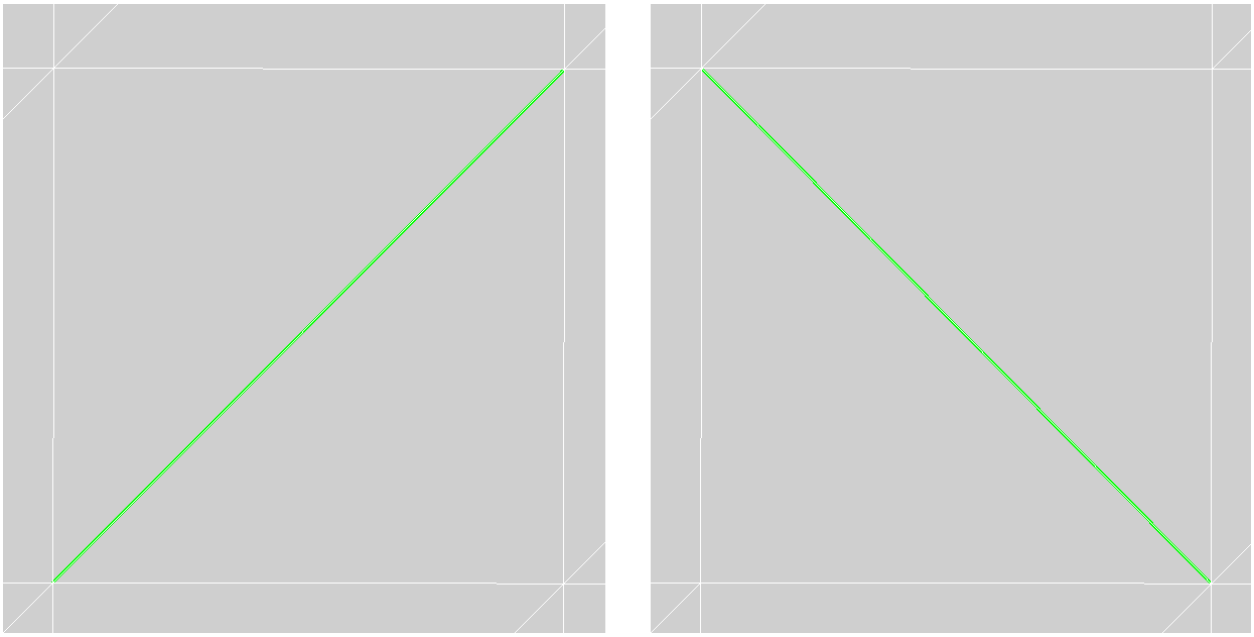


Рис. 2

Метод

```
size_t SplitEdge( size_t iEd,
                  double prm );
```

разрезает ребро в указанном параметре, значение которого должно быть в интервале от 0 до 1. Возвращает индекс созданной в этом месте двухвалентной вершины. Количество рёбер для смежных фасетов увеличится на единицу. На рис. 3 показан результат после триангуляции двух четырёхугольных смежных фасетов:

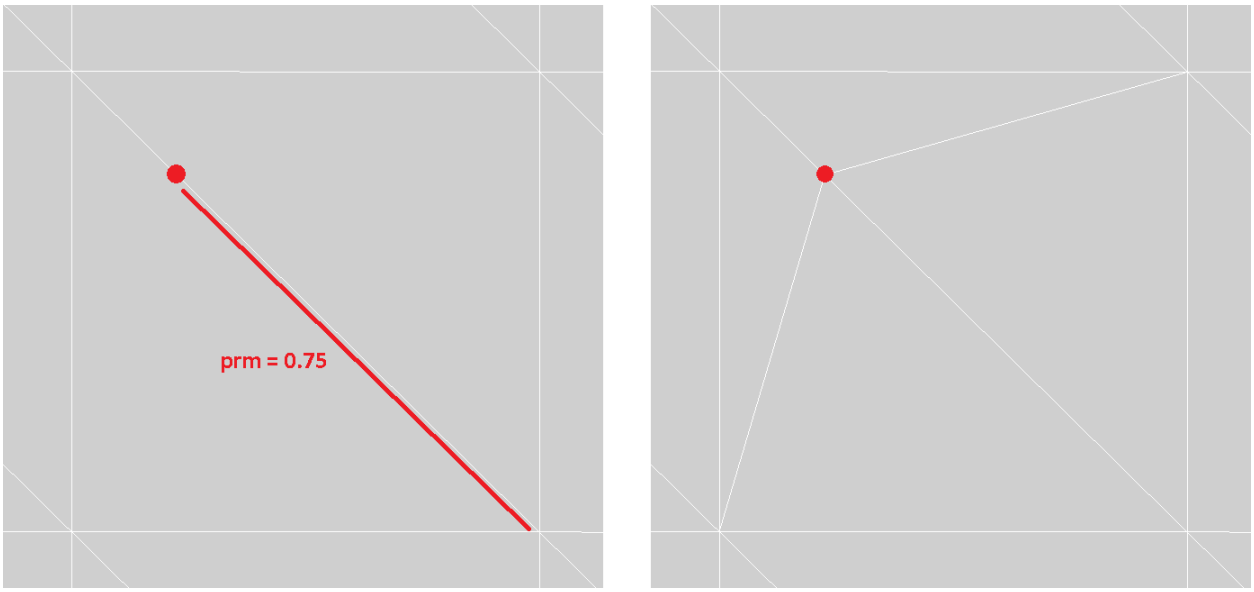


Рис. 3

Метод

```
void SplitFacet( size_t              eVx0,
                 size_t              eVx1,
                 c3d::IndicesPair  & eNew );
```

разрезает фасет на два путём вставки пары рёбер между вершинами *v0* и *v1*, из которых исходят указанные рёбра *eVx0*, *eVx1*, и возвращает индексы этих вставленных рёбер. Входные рёбра должны принадлежать одному фасету, который и будет разрезаться. Пример работы показан на рис. 4:

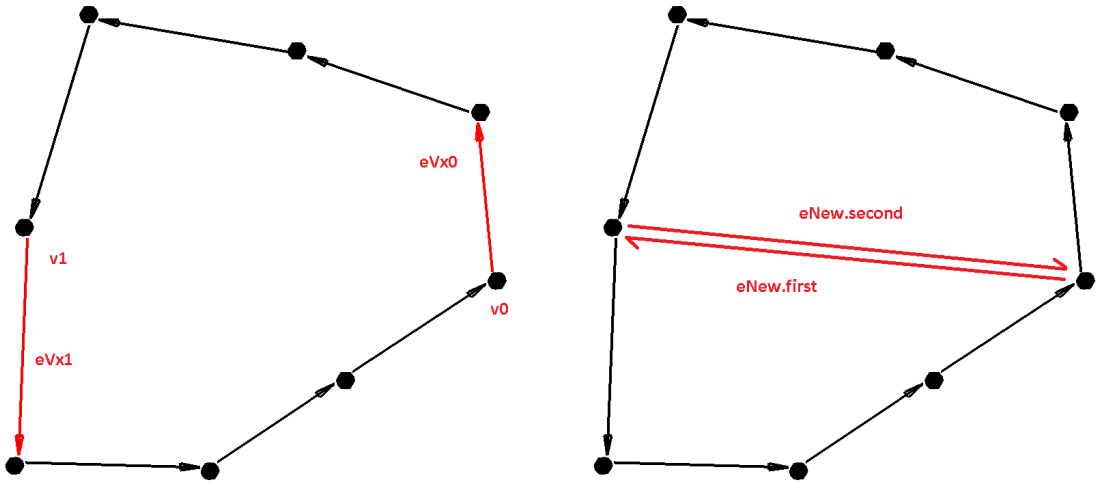


Рис. 4

Метод

```
bool JoinEdgeFacets( size_t iEd );
```

объединяет два фасета, смежных ребру, в один. Это операция, обратная методу SplitFacet(). Нельзя объединить фасеты, если в результате получится фасет, касающийся сам себя ребром или вершиной.

Метод

```
bool RemoveVertexValence2( size_t iVx );
```

удаляет двухвалентную вершину сетки. Операция, обратная методу SplitEdge().

Метод

```
size_t InsertVertexTriangle( size_t iTr,
                             double u,
                             double v );
```

вставляет новую вершину в треугольник и разрезает его на три. Возвращает индекс вставленной вершины. Параметры *u* и *v* – барицентрические координаты, которые определяют новую вершину внутри треугольника. Значения атрибутов для новой вершины (3D точка, 2D параметр, нормаль, кривизна и т.д.) рассчитываются как линейная комбинация из величин в вершинах треугольника согласно барицентрическим координатам.

Метод

```
size_t InsertVertexFacet( const c3d::IndicesVector &      edgesSplit,
                          const std::array<size_t, 3> &   edgesInterpolate,
                                double                    u,
                                double                    v );
```

вставляет новую вершину в фасет и разрезает его на несколько новых фасетов путём соединения рёбрами вставленной вершины с вершинами, из которых исходят рёбра из массива *edgesSplit*. Рёбра из этого массива должны принадлежать одному фасету. Массив *edgesInterpolate* определяет «виртуальный» треугольник из вершин фасета, относительно которого задаются барицентрические координаты *u* и *v*.

Пример использования показан на рисунке: на рисунке a) чёрным показан фасет, который нужно разбить синей вершиной и фиолетовыми рёбрами; на рисунке b) зелёным цветом показаны рёбра фасета, индексы которых должны быть указаны в массиве *edgesSplit* (они должны быть расположены в порядке следования внутри фасета); на рисунке c) зелёным цветом показан «виртуальный» треугольник, относительно которого задаются барицентрические координаты, а красным – индексы рёбер, которые нужно поместить в массив *edgesInterpolate*. Расчёт значений атрибутов для новой вершины делается аналогично методу InsertVertexTriangle().

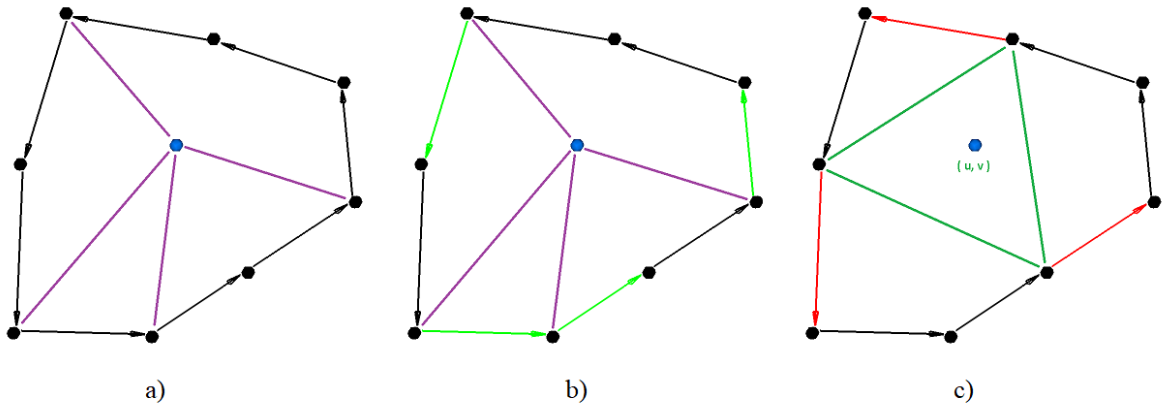


Рис. 5

5.4.3 Методы для работы с сегментацией

Метод

```
bool IsSegmented() const;
```

проверяет, инициализирована ли сегментация. Рекомендуется всегда инициализировать сегментацию.

Метод

```
void Initialize( const MbHalfedgeBoundaryCriterionBase & crit );
```

инициализирует сегментацию согласно указанному критерию границы. Критерий должен быть реализован пользователем как наследник базового класса MbHalfedgeBoundaryCriterionBase, в котором должна быть заложена логика того, считать

ли ребро сетки разделительным или нет. Например, можно предварительно посчитать двугранные углы для рёбер сетки, а в критерии проверять, превышает ли угол заданный порог или нет. Тогда получится сегментация сетки по острым сломам.

Метод

```
void Initialize();
```

инициализирует сегментацию из одного региона. Сетка разбивается на связные части, каждой из которых назначается подрегион. Сегментационные рёбра образуются из открытых рёбер сетки, если таковые имеются. Рекомендуется всегда инициализировать так сегментацию при ручном создании полигонального объекта.

Метод

```
const MbHalfedge & GetSegmTopo() const;
```

открывает доступ к константной топологии сегментации в виде Halfedge. Нужен для конструирования некоторых сегментационных итераторов.

Метод

```
size_t LoopsCount() const;
```

возвращает количество циклов в сегментации (включая неактивные).

Метод

```
const MbSegmentationLoop & Loop( size_t iLoop ) const;
```

получает цикл сегментации по его индексу.

Метод

```
size_t RegionCount() const;
```

возвращает количество регионов в сегментации (включая неактивные).

Метод

```
bool IsRegionActive( size_t iRegion ) const;
```

проверяет, является ли регион активным. То же самое можно сделать с помощью метода IsFacetActive() с тем же индексом для топологии сегментации, возвращаемой методом GetSegmTopo().

Метод

```
size_t SubRegionsCount() const;
```

возвращает количество подрегионов в сегментации (включая неактивные).

Метод

```
size_t SubRegionsCount( size_t iRegion ) const;
```

возвращает количество подрегионов, входящих в указанный регион.

Метод

```
const MbSegmentationSubRegion & SubRegion( size_t iSub ) const;
```

возвращает подрегион сегментации по его индексу.

Метод

```
size_t EdgeSegmCount() const;
```

возвращает количество рёбер в сегментации (включая неактивные).

Метод

```
const MbHalfedgeEdge & EdgeSegm( size_t iEdSegm ) const;
```

возвращает ребро сегментации по его индексу.

Метод

```
bool IsInnerEdge( size_t iEdSegm ) const;
```

проверяет, является ли ребро сегментации внутренним.

Метод

```
size_t GetSegmEdgeId( size_t iEdTopo ) const;
```

возвращает индекс сегментационного ребра, которому принадлежит указанное сеточное ребро.

Метод

```
void GetFacetRegionId( size_t iFacet,
                       size_t & iRegion,
                       size_t & iSubRegion ) const;
```

возвращает индексы региона и подрегиона, которым принадлежит указанный фасет сетки.

Метод

```
size_t VertexSegmCount() const;
```

получает количество вершин в сегментации (включая неактивные).

Метод

```
const MbHalfedgeVertex & VertexSegm( size_t iVxSegm ) const;
```

возвращает вершину сегментации по её индексу.

Метод

```
void GetRegion( size_t                r,
                c3d::IndicesVector & facets ) const;
```

возвращает индексы всех фасетов сетки, входящих в указанный регион.

Метод

```
void GetSubRegion( size_t                iSub,
                   c3d::IndicesVector &  facets,
                   bool                  bAdd = false ) const;
```

возвращает индексы всех фасетов сетки, входящих в указанный подрегион.

Метод

```
void DeleteRegion( size_t iRegion );
```

удаляет указанный регион. При этом поддерживается валидность топологии и сетки, и сегментации, аналогично удалению фасетов.

Метод

```
void SplitByEdgesTopo( const c3d::IndicesVector & splitSet,
                       bool                       bKeepRegions );
```

разбивает существующую сегментацию набором рёбер сетки. Это может привести к появлению новых связанных групп фасетов. Для каждого из них создаётся новый подрегион. По флагу bKeepRegions эти подрегионы либо остаются в составе старого региона, либо из каждого из них будет создан новый регион. Также эта операция может привести к появлению внутренних рёбер сегментации.

Метод

```
void MergeRegions( const c3d::IndicesVector & regions,
                   bool                       bRemoveValence2 );
```

объединяет указанные регионы в один. Его индекс равен первому индексу из массива *regions*. При этом могут образоваться последовательности рёбер сегментации, разделённые двухвалентными сегментационными вершинами. По флагу bRemoveValence2 эти последовательности объединяются в рёбра сегментации.

Метод

```
void SplitByFacetsTopo( const c3d::IndicesVector & splitSet,
                        bool                       bMakeOneRegion,
                        bool                       bKeepRegions );
```

разбивает существующую сегментацию набором фасетов сетки. Этому набору соответствует один или несколько связных групп фасетов, относящихся к старым регионам и каждой из которых назначается новый подрегион. Комбинация этих групп в новые регионы и подрегионы зависит от двух управляющих флагов bMakeOneRegion и bKeepRegions. Пример показан на рис. 6, пусть сегментация состоит из четырёх простых регионов, которые нужно разбить выделенными ярко зелёным фасетами:

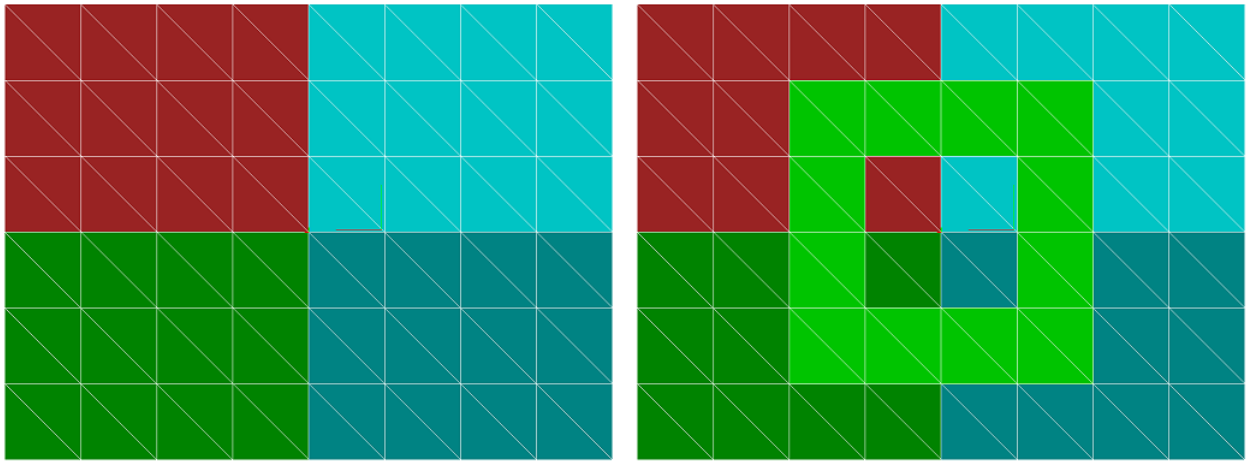


Рис. 6

1. bMakeOneRegion = true, bKeepRegions = true.

Выделенные фасеты образовали один новый регион (жёлтый). Старые регионы остались, но разделились каждый на два подрегиона. Теперь в сегментации 5 регионов (рис. 7).

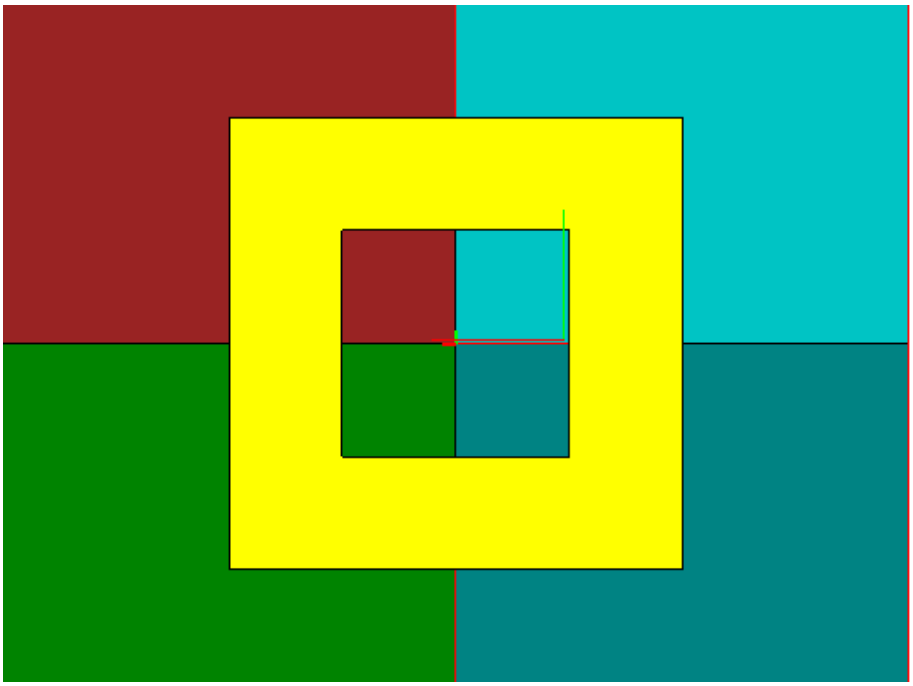


Рис. 7

2. bMakeOneRegion = true, bKeepRegions = false.

Выделенные фасеты образовали один новый регион (жёлтый). Каждый из старых регионов распался на две несвязные части, каждая из которых образовала новый отдельный регион. Всего теперь в сегментации 9 регионов (рис. 8).

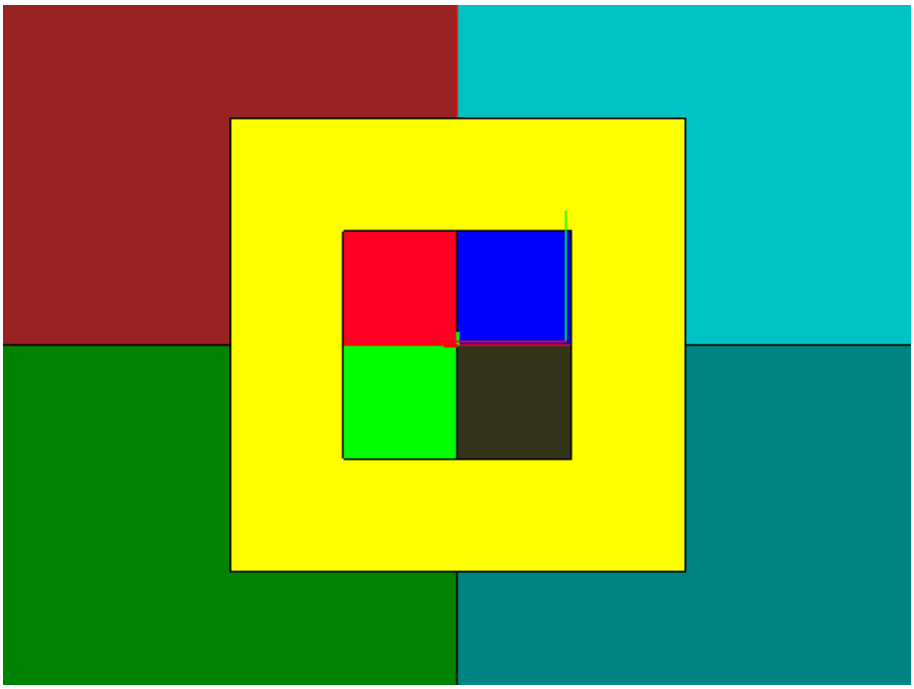


Рис. 8

3. bMakeOneRegion = false, bKeepRegions = true.

Выделенные фасеты разбили каждый из старых регионов на 3 части. Регионов в сегментации по-прежнему 4, но каждый из них теперь состоит из трёх подрегионов (рис. 9).

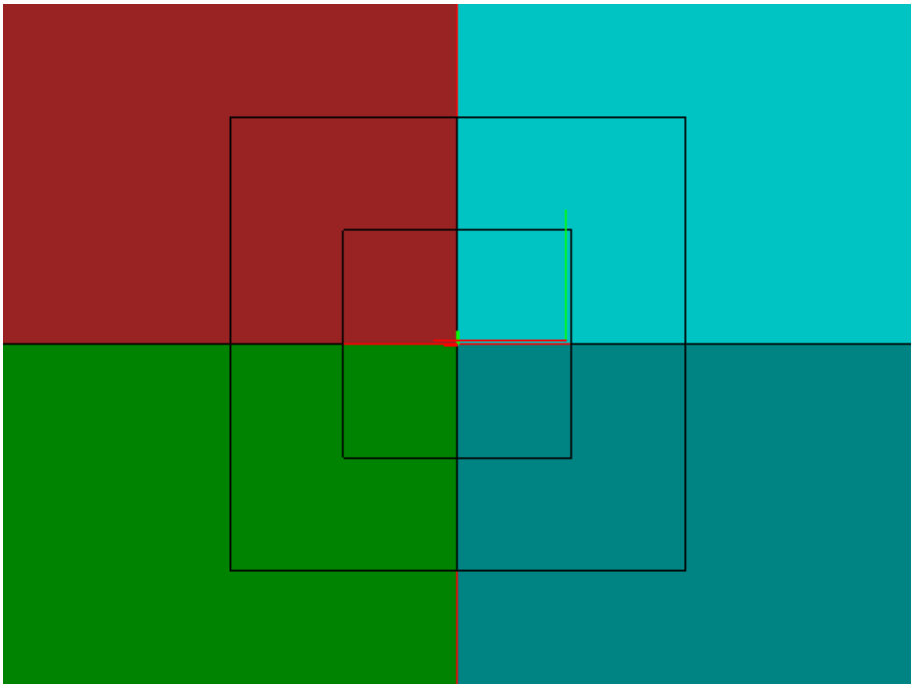


Рис. 9

4. bMakeOneRegion = false, bKeepRegions = false.

Выделенные фасеты разбили каждый из старых регионов на 3 части. Каждой такой части назначен новый регион. Всего теперь в сегментации 12 регионов, состоящих из одного подрегиона (рис. 10).

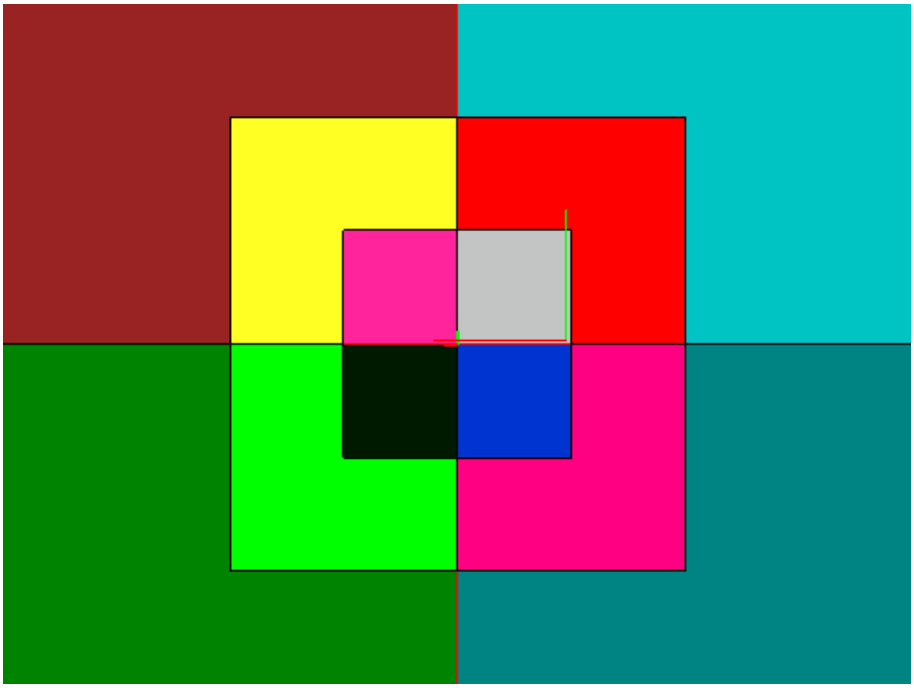


Рис. 10

Метод

```
void SplitRegionBySubregions( size_t iReg );
```

выделяет подрегионы региона как отдельные регионы.

Метод

```
c3d::IndicesPair SplitSegmentationEdge( size_t iEdSegm,
                                        size_t iEdTopo,
                                        size_t iVxSegmSplit = SYS_MAX_T );
```

разрезает ребро сегментации с индексом *iEdSegm* (и его близнеца) в вершине сетки, из которой исходит сеточное ребро *iEdTopo*. Ребро *iEdTopo* должно принадлежать сегментационному ребру и не должно совпадать с самым первым сеточным ребром для сегментационного ребра. В этом месте возникнет двухвалентная вершина сегментации, существующая с индексом *iVxSegmSplit*, если он задан, либо будет добавлена новая. Возвращает индексы новых рёбер сегментации, на которые распалось старое ребро сегментации и его близнец (если он существует). Топологию сетки метод не изменяет. Новые рёбра сегментации наследуют атрибуты от старых, из которых они были получены.

Метод

```
bool ExtractSubRegionAsRegion( size_t iSub );
```

выделяет указанный подрегион в отдельный регион. Если старый регион состоял только из указанного подрегиона, никаких действий выполнено не будет.

Метод

```
void RemoveSegmVertexValence2( size_t iVx );
```

удаляет вершину сегментации с валентностью 2. Метод, обратный к SplitSegmentationEdge().

Метод

```
size_t GetTopoVertex( size_t iVxSegm ) const;
```

возвращает индекс вершины сетки, в которой находится указанная вершина сегментации.

Метод

```
size_t GetSegmEdgeRegionId( size_t iEdSegm ) const;
```

возвращает индекс региона, которому принадлежит ребро сегментации.

Метод

```
size_t GetSegmEdgeSubRegionId( size_t iEdSegm ) const;
```

возвращает индекс подрегиона, которому принадлежит ребро сегментации.

Метод

```
size_t GetLoopRegionId( size_t iLoop ) const;
```

возвращает индекс региона, которому принадлежит цикл.

Метод

```
size_t GetLoopSubRegionId( size_t iLoop ) const;
```

возвращает индекс подрегиона, которому принадлежит цикл.

Метод

```
size_t GetNeighborRegionId( size_t iEdSegm ) const;
```

возвращает индекс региона с другой стороны ребра сегментации.

Метод

```
size_t GetNeighborSubRegionId( size_t iEdSegm ) const;
```

возвращает индекс подрегиона с другой стороны ребра сегментации.

5.4.4 Методы для работы с атрибутами сетки и сегментации

Метод

```
size_t MeshHalfedgeAttributeCount() const;
```

возвращает количество сеточных атрибутов.

Метод

```
size_t SegmHalfedgeAttributeCount() const;
```

возвращает количество сегментационных атрибутов.

Метод

```
template <class T>
T * AddMeshHalfedgeAttribute();
```

создаёт сеточный атрибут указанного типа и возвращает указатель на него.

Метод

```
template <class T>
T * AddSegmHalfedgeAttribute();
```

добавляет сегментационный атрибут указанного типа и возвращает указатель на него.

Метод

```
template <class T>
T * GetMeshHalfedgeAttribute();
```

и константный вариант

```
template <class T>
const T * GetMeshHalfedgeAttribute() const;
```

находят в полигональном объекте сеточный атрибут указанного типа (первый подходящий в массиве атрибутов) и возвращают указатель на него.

Метод

```
template <class T>
T * GetSegmHalfedgeAttribute();
```

и константный вариант

```
template <class T>
const T * GetSegmHalfedgeAttribute() const;
```

находят в полигональном объекте сегментационный атрибут указанного типа (первый подходящий в массиве атрибутов) и возвращают указатель на него.

Метод

```
template <class T>
void DeleteMeshHalfedgeAttribute();
```

удаляет сеточный атрибут указанного типа (первый подходящий в массиве атрибутов).

Метод

```
template <class T>
void DeleteSegmHalfedgeAttribute();
```

удаляет сегментационный атрибут указанного типа (первый подходящий в массиве атрибутов).

Методы

```
MbHalfedgeAttrBase * GetMeshHalfedgeAttributeByIndex( size_t ind );

const MbHalfedgeAttrBase * GetMeshHalfedgeAttributeByIndex( size_t ind ) const;
```

возвращают указатель на сеточный атрибут с заданным индексом в массиве атрибутов.

Методы

```
MbHalfedgeAttrBase * GetSegmHalfedgeAttributeByIndex( size_t ind );

const MbHalfedgeAttrBase * GetSegmHalfedgeAttributeByIndex( size_t ind ) const;
```

возвращают указатель на сегментационный атрибут с заданным индексом в массиве атрибутов.

Метод

```
void DeleteMeshHalfedgeAttribute( MbHalfedgeAttrBase * pAttr );
```

удаляет указанный сеточный атрибут.

Метод

```
void DeleteSegmHalfedgeAttribute( MbHalfedgeAttrBase * pAttr );
```

удаляет указанный сегментационный атрибут.

Метод

```
MbAttributeContainer * GetSegmRegionModelAttributes( size_t iRegion,
                                                     bool   bCreate );
```

возвращает контейнер атрибутов объекта геометрического ядра для региона. Если он отсутствует, то по флагу bCreate будет создан, иначе возвращается нулевой указатель.

Метод

```
const MbAttributeContainer * GetSegmRegionModelAttributes( size_t iRegion ) const;
```

возвращает контейнер атрибутов объекта геометрического ядра для региона.

Метод

```
MbAttributeContainer * GetSegmEdgeModelAttributes( size_t iEdSegm,
                                                   bool   bCreate );
```

возвращает контейнер атрибутов объекта геометрического ядра для ребра сегментации. Если он отсутствует, то по флагу *bCreate* будет создан, иначе возвращается нулевой указатель.

Метод

```
const MbAttributeContainer * GetSegmEdgeModelAttributes( size_t iEdSegm ) const;
```

возвращает контейнер атрибутов объекта геометрического ядра для ребра сегментации.

Метод

```
MbAttributeContainer * GetSegmVertexModelAttributes( size_t iVxSegm,
                                                     bool   bCreate );
```

возвращает контейнер атрибутов объекта геометрического ядра для вершины сегментации. Если он отсутствует, то по флагу bCreate будет создан, иначе возвращается нулевой указатель.

Метод

```
const MbAttributeContainer * GetSegmVertexModelAttributes( size_t iVxSegm ) const;
```

возвращает контейнер атрибутов объекта геометрического ядра для вершины сегментации.

5.5 Атрибуты сетки и сегментации

Как было сказано выше, атрибуты сетки и сегментации нужны для ассоциации разного рода величин с топологическими элементами полигонального объекта – вершинами, рёбрами и фасетами сетки, регионами, рёбрами и вершинами сегментации.

Некоторые типы атрибутов являются уникальными, то есть во всех алгоритмах предполагается, что в полигональном объекте есть только один атрибут такого типа, либо его нет вообще. К таким атрибутам относятся:

- MbHalfedgeAttrPoint3D – 3D точки для вершин;
- MbHalfedgeAttrParameters – 2D параметры для вершин;
- MbHalfedgeAttrParamByEdge – реброзависимые 2D параметры;
- MbHalfedgeAttrNormalsVertex – нормали;
- MbHalfedgeAttrCurvaturesVertex – кривизны;
- MbHalfedgeAttrDensityVertex – целевой размер фасета для вершин;
- MbHalfedgeAttrDihedralAngles – двугранные углы для рёбер;
- MbHalfedgeAttrStrains – напряжения в треугольниках;
- MbHalfedgeAttrSurfaces – поверхности для регионов.

Остальные атрибуты, как правило, являются вспомогательными и могут присутствовать в любом количестве в зависимости от конкретной задачи. К ним относятся:

- MbHalfedgeAttrBoolFacet;
- MbHalfedgeAttrBoolEdge;
- MbHalfedgeAttrBoolVertex;
- MbHalfedgeAttrUintFacet;
- MbHalfedgeAttrUintEdge;
- MbHalfedgeAttrUintVertex;
- MbHalfedgeAttrIntFacet;
- MbHalfedgeAttrIntEdge;
- MbHalfedgeAttrIntVertex;
- MbHalfedgeAttrSizetFacet;

- MbHalfedgeAttrSizetEdge;
- MbHalfedgeAttrSizetVertex;
- MbHalfedgeAttrDoubleFacet;
- MbHalfedgeAttrDoubleEdge;
- MbHalfedgeAttrDoubleVertex;
- MbHalfedgeAttrFloatFacet;
- MbHalfedgeAttrFloatEdge;
- MbHalfedgeAttrFloatVertex;
- MbHalfedgeAttrDirectionEdge3D;
- MbHalfedgeAttrNormalsFacet;
- MbHalfedgeAttrDirectionVertex3D.

Создание пользователем собственного типа атрибута не предусмотрено.

5.6 Реброзависимые атрибуты

Предположим, мы хотим получить полигональный объект путём линеаризации параметрической оболочки и сохранить в вершинах 2D параметры на соответствующих гранях. Каждой грани будет соответствовать регион в сегментации. Но нумерация вершин в полигональном объекте сквозная, единая на всю сетку. И для вершин сетки, которые лежат на рёбрах и в вершинах сегментации, получается неоднозначная ситуация. Если такую вершину рассматривать по отношению к одной грани – ей должен соответствовать некий 2D параметр, а по отношению к другой – другой параметр. Таким образом, становится невозможным описывать 2D параметры простым атрибутом, ставящим в соответствие каждой вершине один параметр. То же самое относится к таким величинам, как нормали и кривизны.

Для поддержки таких ситуаций предусмотрены реброзависимые атрибуты – наследники базового класса MbHalfedgeAttrEdgeDependentVertexBase. Таких атрибутов три:

- MbHalfedgeAttrParamByEdge;
- MbHalfedgeAttrNormalsVertex;
- MbHalfedgeAttrCurvaturesVertex.

Устроены они следующим образом (на примере нормалей). Нормали лежат отдельным массивом, и их количество и нумерация в этом массиве никак не связаны с вершинами сетки. Каждому ребру сетки ставится в соответствие индекс в массиве нормалей. Если это внутренняя для региона вершина, то все рёбра в её веере будут ссылаться на одну и ту же нормаль. Если эта вершина лежит на ребре сегментации или в вершине сегментации, то разным рёбрам веера соответствуют разные нормали.

Рассмотрим подробнее пример простой сетки из четырёх треугольников, показанной на рис. 1. Нормаль в вершине будем рассчитывать усреднением нормалей треугольников, входящих в её веер. Рассмотрим два случая:

- первый – когда вся сетка образует одно целое (см. рисунок c)),
- второй – когда она сегментирована на два плоских региона (см. рисунок a)).

Направление нормалей показано красными отрезками. В веере ближайшей угловой вершины есть три рёбра, исходящих из неё. Справа все они ссылаются на одну и ту же нормаль. Слева два красных рёбра (средний рисунок) ссылаются на нормаль N1, а одно зелёное – на нормаль N2.

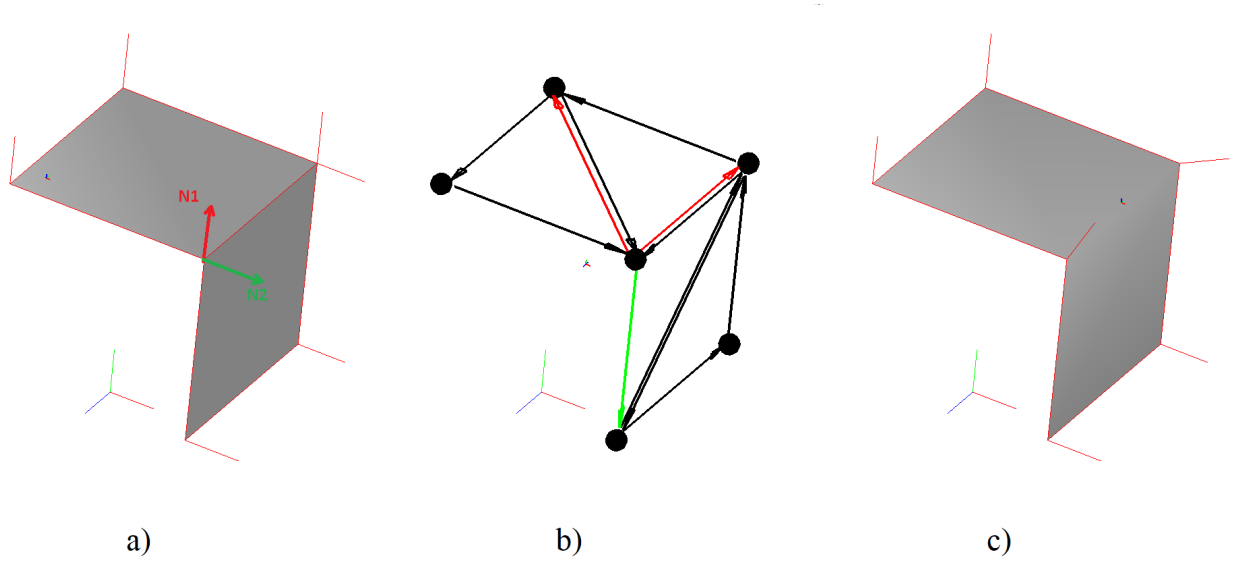


Рис. 1

Таким образом, при наличии нормалей визуализация сегментированной и несегментированной сетки существенно отличается, как показано на рис. 2:

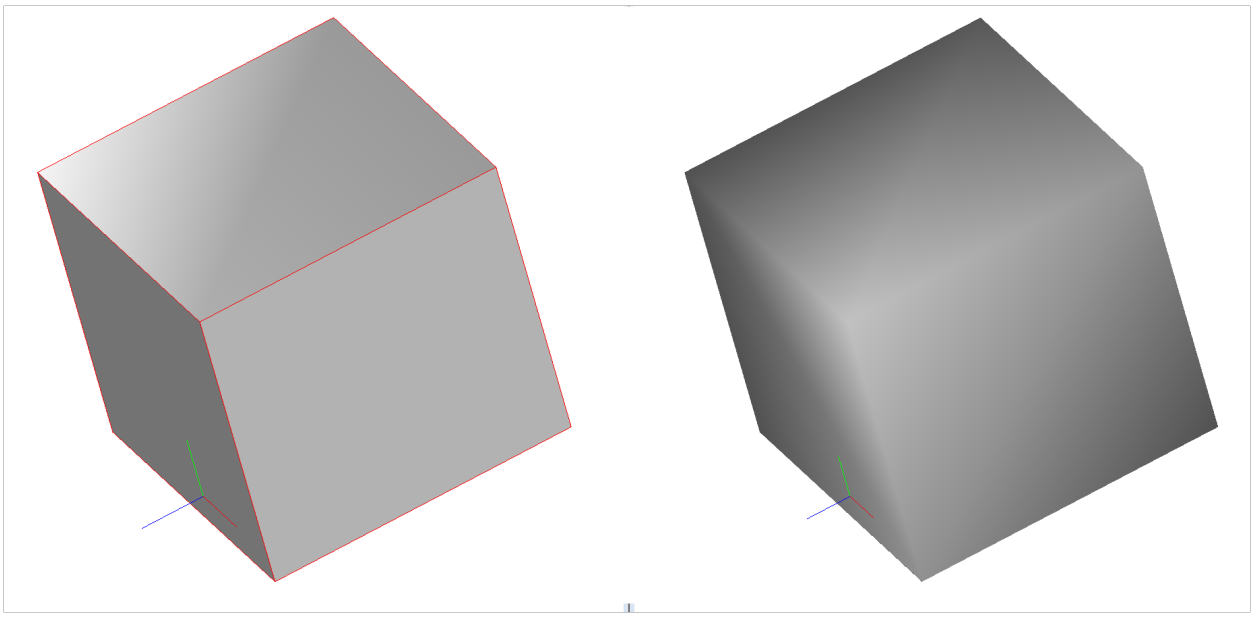


Рис. 2

Не следует интерпретировать метод атрибута нормалей

```
const MbVector3D & GetNormal( size_t iNm ) const;
```

как метод доступа к нормали вершины и использовать в качестве аргумента индекс вершины. Доступ к нормалям вершины должен быть организован через рёбра, которые из неё исходят, с помощью метода

```
const MbVector3D & GetNormalEdge( size_t iEd ) const;
```

Все реброзависимые атрибуты не предполагаются к созданию пользователем. Для их создания нужно вызывать функции ядра для расчёта этих величин. В этих функциях будет задана структура атрибутов, соответствующая текущей сегментации. При любых изменениях в топологии сегментации (объединение регионов, разбивка рёбрами сетки или фасетами и т.д.) структура имеющихся реброзависимых атрибутов будет также изменена согласно новой сегментации.

ГЛАВА 6

C3D Collision Detection

На этапе проектирования сборочного изделия необходимо убедиться в том, что между частями сборки отсутствуют пространственные наложения — коллизии.

Модуль обнаружения столкновений C3D Collision Detection позволяет инженерному приложению (CAD, BIM и т. п.) проверить модели на коллизии, чтобы на раннем этапе обеспечить собираемость модели или устранить препятствия движению частей механизма. Раннее выявление ошибок моделирования при проверке проекта на коллизии снижает издержки на этапе изготовления изделия.

Модуль предоставляет два способа оценки столкновения твердотельных объектов:

- Первый способ основан на алгоритмах булевой операции и предназначен для точной оценки пересечения двух неподвижных тел с вычислением ребер границы пересечения.
- Второй способ основан на работе с полигональным представлением модели и предназначен для динамических сцен, заданных набором тел. Данный вариант рекомендуется использовать, если требуется многократная оценка столкновения твердотельных моделей, изменяющих свое положение в режиме реального времени.

Для активации лицензии на модуль необходимо вызвать функцию EnableMathModules() с соответствующими параметрами.

6.1 Определение пересечения двух тел

Метод

```
MbResultType InterferenceSolids(MbSolid &                    solid1,
                                MbSolid &                    solid2,
                                std::vector<MbCurveEdge*> * edges,
                                std::vector<ptrdiff_t> (*faceNumbers)[4])
```

выполняет оценку пересечения двух тел.

Входными параметрами метода являются:

- **solid**1 – первое тело;
- **solid**2 – второе тело;
- **edges** – ребра пересечения тел (может быть 0);
- (*faceNumbers)[4] – контейнер номеров граней (может быть 0).

Выходными параметрами метода являются:

- ребра пересечения **edges** граней тел **solid**1 и **solid**2;
- номера пересекшихся граней (*faceNumbers)[4].

Метод возвращает rt_Intersect, если тела пересекаются, или rt_NoIntersect, если тела не пересекаются.

Метод объявлен в файле *cdet_bool.h*.

Метод модифицирует ребра тел **solid**1 и **solid**2, поэтому если требуется сохранить тела неизменными, необходимо передавать в метод копии, сделанные с помощью MbSolid::Duplicate(). Модифицирование заключается в резке ребер тел ребрами пересечения.

Возвращаемое значение rt_Intersect означает, что пара тел имеет некоторый объем пересечения или некоторую площадь соприкосновения. Функция вернет rt_NoIntersect, если тела не имеют объема пересечения. Касание в одной точке не считается пересечением, так как имеет нулевой объем пересечения.

Если указатель на контейнер рёбер **edges** не равен NULL, то в **edges** будут помещены ребра пересечения тел **solid**1 и **solid**2.

Если указатель на контейнер номеров граней faceNumbers не равен NULL, то в faceNumbers[0] и faceNumbers[1] будут помещены номера пересекшихся граней соответственно первого и второго тела, кроме того, в faceNumbers[2] и faceNumbers[3] будут помещены номера соприкасающихся граней соответственно первого и второго тела.

6.2 Определение столкновений в наборе тел

Данный способ оценки столкновений применяется в случае, когда необходимо осуществить серию тестов на столкновение для одного и того же набора тел, меняющих свое положение во времени. На рис. 1.2.1 изображен механизм, представляющий собой сборочную единицу из семи деталей, положение которых подчинено кинематическим связям. Требуется оценить границы, в пределах которых детали не мешают друг другу при работе механизма. На рис. 1 приведено положение деталей механизма, при котором возникают столкновения между ними. Красным цветом на рис. 2 показаны грани тел, встретившихся друг с другом при выходе за границы свободного движения.

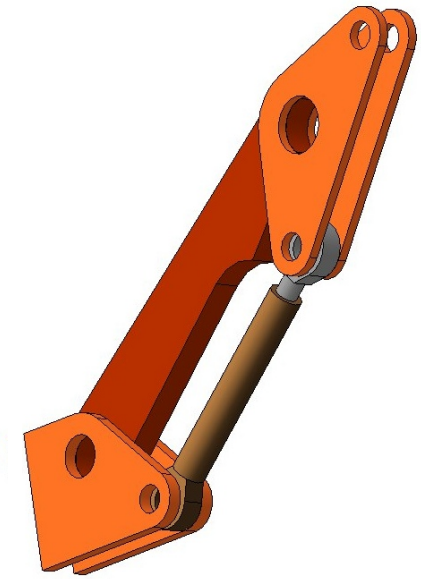


Рис. 1

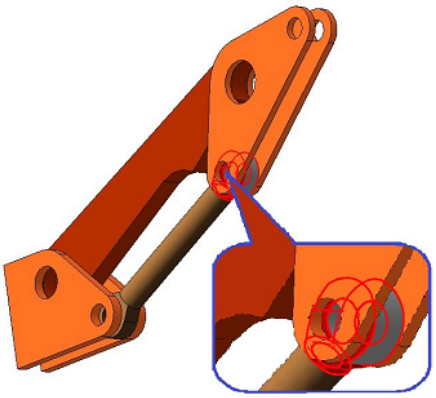


Рис. 2

Для контроля столкновения между телами из некоторого набора необходимо завести экземпляр класса MbCollisionDetectionUtility, с помощью метода **AddItem**() добавить тела в набор для контроля столкновений. В момент времени, когда одно или несколько тел изменили положение, приложение сообщает телам набора новое положение вызовом **Reposition** и определяет, имеются ли в геометрической модели коллизии с помощью вызова **CheckCollisions**.

Класс для контроля столкновений MbCollisionDetectionUtility объявлен в файле *cdet_utility.h*, вспомогательные типы данных объявлены в файле *cdet_data.h*.

Метод

```
cdet_item MbCollisionDetectionUtility::AddItem (const MbSolid &        solid,
                                                const MbPlacement3D & place,
                                                      cdet_app_item   appItem =␣
↪CDET_APP_NULL)
```

добавляет твердое тело **solid** в набор для контроля столкновений.

Входными параметрами метода являются:

- **solid** – твердотельная модель;
- **place** – локальная система координат, задающая положение тела **solid** в пространстве;
- *appItem* – геометрический объект клиентского приложения, которому принадлежит тело.

Метод возвращает дескриптор объекта для контроля столкновений.

Параметр *appItem* используется для идентификации объектов модели клиентского приложения при их столкновении. Если параметр *appItem* не определен, то идентификация осуществляется только на основе дескриптора (тип cdet_item), с которым тело было зарегистрировано в наборе. Предполагается, что одному объекту *appItem*, может принадлежать не одно тело MbSolid, то есть для одного и того же объекта с *appItem* метод **AddItem** может быть вызван более одного раза. В этом случае тела из группы, добавленной для одного и того же *appItem,* не проверяются на столкновения между собой.

Метод

```
cdet_item MbCollisionDetectionUtility::RemoveItem(cdet_item cdItem)
```

удаляет твердое тело из набора для контроля столкновений.

Входным параметром метода является *cdItem* – дескриптор объекта для контроля столкновений, ранее добавленный методом **AddItem**().

Метод возвращает дескриптор объекта.

Метод

```
cdet_result MbCollisionDetectionUtility::CheckCollisions(cdet_query cdQuery =␣
↪defaultQuery)
```

проверяет столкновения между объектами из контрольного набора тел.

Входным параметром метода является **cdQuery** – структура запроса деталей столкновения.

Метод возвращает код результата поиска столкновений cdet_result, который может содержать следующие коды:

- CDM_RESULT_None – результат неопределенного типа;
- CDM_RESULT_NoCollisionDetected – столкновений не выявлено: пересечений нет или поиск был прерван;
- CDM_RESULT_CollisionDetected – выявлено хотя бы одно объемное пересечение (коллизия);
- CDM_RESULT_Error – неизвестная ошибка или внутрисистемная ошибка.

Метод **CheckCollisions**() ищет столкновения между телами, добавленными в контрольный набор с помощью вызова **AddItem**(). Метод вернет значение CDM_RESULT_CollisionDetected, если в данном наборе тел найдено столкновение. Если столкновений не обнаружено, то метод вернёт CDM_RESULT_NoCollisionDetected. Код cdet_result является основным результатом работы алгоритма **CheckCollisions**, однако с помощью структуры **cdQuery** можно заказать более детальную информацию о результатах поиска столкновений, а также ограничить или даже прервать поиск, когда это

требуется. Если параметр **cdQuery** не определен, то алгоритм поиска не тратит время на выявление всех подробностей столкновения и останавливается, как только будет найдена первая пара пересекшихся граней.

Если для одного и того же объекта cdet_app_item добавлены два и более тел, то столкновения в пределах такой группы не проверяется.

6.3 Запросы на поиск столкновений

Кроме самого факта столкновения тел иногда требуется получить список граней, которые участвуют в столкновении, как показано на рис. 25.2.2, или выделить группы тел, между которыми искать столкновения не требуется. Для этого предусмотрен набор типовых классов, наследников от cdet_query, настроенных на запрос подробностей столкновения:

- Структура cdet_query_result – самый простой и быстрый вариант, отвечающий на вопрос имеются ли столкновения в контрольной группе тел. Именно эта структура используется в качестве параметра по умолчанию для **CheckCollisions**.
- Структура cdet_first_collided – выдает первую найденную пару граней столкновения.
- Структура cdet_collided_faces – выдает список граней, участвующих в столкновениях, в виде упорядоченного множества пар std::pair<cdet_app_item, const MbRefItem*>, где первый элемент пары –твердотельная модель клиентского приложения, второй элемент пары – грань, принадлежащая модели. Также с помощью метода cdet_collided_faces::**Group** можно сгруппировать объекты контрольного набора тел так, что бы искать только пересечения между группами.

Метод cdet_collided_faces::**Group**() объединяет в группу пару тел; для объединения в группу ***n*** тел необходимо вызвать cdet_collided_faces::**Group** ***n***-1 раз для первого и всех остальных тел группы.

В параграфе *Настройка запроса на поиск столкновений* описано, как с помощью наследования от cdet_query пользователь геометрического ядра может настроить собственные запросы, однако, в большинстве случаев необходимые запросы можно осуществить с помощью класса cdet_collided_faces. Для этого необходимо завести экземпляр cdet_collided_faces и применить его в качестве параметра **CheckCollisions**. Перед этим можно настроить фильтр поиска с помощью методов cdet_collided_faces::**Group** и cdet_collided_faces::**ExcludeGroup**. При необходимости фильтр поиска можно сбросить с помощью вызова cdet_collided_faces::Reset и перенастроить фильтр заново.

Метод

```
cdet_collided_faces::Group(cdet_app_item fst, cdet_app_item snd)
```

объявляет группу тел, элементы которой не проверяются на столкновение друг с другом.

Входными параметрами метода являются *fst* и *snd* – пара тел, которые уже состоят в группах или впервые объединяются в группу.

Данный метод объединяет в одну группу два тела или присоединяет одно тело к группе другого тела. Таким образом, для того, чтобы объединить ***n*** тел в группу необходимо вызвать cdet_collided_faces::Group ***n***-1 раз для первого и всех остальных тел группы.

Метод

```
cdet_collided_faces::ExcludeGroup(cdet_app_item member)
```

исключает все элементы группы из поиска на столкновения.

Входным параметром метода является *member* – любое тело, участник группы, элементы которой исключаются.

Данный метод отключает группу из любых проверок на столкновение. Его рекомендуется применять для временного отключения проверки тел на столкновения без полного удаления из набора с помощью MbCollisionDetectionUtility::**RemoveItem** так как последующее добавление с помощью **AddItem** займет больше вычислительного времени, чем отменить исключение с помощью cdet_collided_faces::**Reset**. Для исключения набора тел из проверки на столкновения допускается многократно вызвать метод **ExcludeGroup** для всех элементов набора либо предварительно объединить все исключаемых тела в группу, а затем один раз вызвать **ExcludeGroup** для любого тела группы.

Метод

```
cdet_collided_faces::Reset()
```

отменяет результаты всех вызовов **Group** и **ExcludeGroup**. Метод применяется, если требуется перенастроить группы и исключения для однажды заданной структуры запросов cdet_collided_faces.

6.4 Настройка запроса на поиск столкновений

Если возможности настроек структуры cdet_collided_faces не достаточны, пользователь геометрического ядра C3D может реализовать собственный запрос. Для этого необходимо определить наследника класса cdet_query и реализовать для него специальную функцию обратного вызова (call-back функцию) вида:

```
typedef cback_res (*cback_func)(cdet_query *, message, cback_data &).
```

Ниже приведен пример реализации на языке программирования C++ структуры cdet_query_result, являющейся наследником структуры запроса cdet_query. Пример задает поиск до первого найденного пересечения и определяет, есть ли хотя бы одно пересечение в контрольном наборе тел. Пример можно найти в файле **cdet_data.h**.

```
//------------------------------------------------------------
struct cdet_query_result: public cdet_query
{
cdet_result result;
cdet_query_result()
    : cdet_query(QueryFunc)
    , result(CDM_RESULT_NoCollisionDetected) {}
private:
static cback_res QueryFunc(cdet_query * query, message code, cback_data &)
{
    cdet_query_result * q = static_cast<cdet_query_result*>(query);
    switch( code )
    {
        case CDET_QUERY_STARTED: // The collision query is started for all solids of␣
↪the set
        {
            q->result = CDM_RESULT_NoCollisionDetected;
            return CBACK_VOID_RESULT;
        }
        case CDET_INTERSECTED: // First intersection is founded.
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
        {
            q->result = CDM_RESULT_CollisionDetected;
            return CBACK_SUFFICIENT;
        }
        case CDET_FINISHED: // A pair of solids is finished.
        return (q->result == CDM_RESULT_CollisionDetected) ?␣
↪CBACK_BREAK:CBACK_VOID_RESULT;
        default:
        return CBACK_VOID;
        }
    }
};
```

В основе запроса лежит функция обратного вызова

```
static cback_res QueryFunc(cdet_query * query, message code, cback_data & cData),
```

которую пользователь может реализовать на свое усмотрение. Данная функция передается в алгоритм **CheckCollisions** через структуру cdet_query и будет вызываться при наступлении одного из следующих событий (enum cdet_query::message):

- CDET_QUERY_STARTED (начат поиск столкновений в контрольной группе тел),
- CDET_STARTED (начат поиск столкновений для пары объектов),
- CDET_FINISHED (завершен поиск столкновений для пары объектов),
- CDET_INTERSECTED (найдено пересечение граней от пары объектов),
- CDET_TOUCHED (найдены соприкасающиеся грани пары объектов).

Входными параметрами функции QueryFunc являются:

- **query** – указатель на структуру запроса,
- *code* – целочисленное значение, кодирующее событие поиска,
- **cData** – структура передачи данных.

Функция возвращает одно из значений перечисления cback_res:

- CBACK_SUFFICIENT (Реакция клиентского приложения на события CDET_INTERSECTED или CDET_TOUCHED. Прерывает поиск для данной пары тел, но продолжает поиск столкновений других пар из контрольного набора тел),
- CBACK_SKIP (Пропустить поиск для данной пары тел по событию CDET_STARTED: Поиск для других пар из набора будет продолжен),
- CBACK_BREAK (Прерывает дальнейший поиск пересечений для всех событий),
- CBACK_VOID (Не оказывает влияния на запрос столкновений, продолжает поиск), которое передает клиентское приложение ядру C3D для настройки алгоритма поиска.

Целочисленное значение *code* определяет событие поиска и может принимать одно из значений:

- CDET_QUERY_STARTED,
- CDET_STARTED,
- CDET_FINISHED,

- CDET_INTERSECTED,
- CDET_TOUCHED,

Структура данных **cData** предназначена для передачи пары элементов типа cdet_query::geom_element и должна иметь вид:

```
struct cback_data
    {
        geom_element first, second;
    };
```

где элемент пары cdet_query::geom_element – это структура деталей поиска, содержащая следующие:

- geom_element::**appItem** – геометрический объект клиентского приложения,
- geom_element::**refItem** – геометрический объект типа MbRefItem, как правило это грань MbFace,
- geom_element::**wMatrix** – матрица перехода в общую, мировую СК из локальной СК тела.

Различная реализация метода обратного вызова **QueryFunc** позволяет выполнять следующие настройки процесса поиска:

- Собрать все пары граней, участвующих в столкновении тел;
- В любой момент исключать из рассмотрения определенные тела или пары тел;
- Различать случай пересечения и касания граней;
- Прерывать поиск всех пересечений при достижении определенного результата;
- Прерывать поиск при обнаружении пересечения или касания;
- Группировать тела, чтобы в пределах группы пересечения не искались.

6.5 Группировка тел из контрольного набора

При контроле столкновений часто требуется искать столкновения не для всех пар из контрольного набора тел, а только между группами тел. Например, можно повысить быстродействие, если не тратить время на тесты пересечения, когда известно, что тела внутри группы не пересекаются, либо по какой-то причине проверки внутри группы не требуется. Организовать такую группировку можно двумя способами.

Первый способ – передавать одно и то же значение *appItem* при добавлении тел одной группы в метод **AddItem**. Этот способ удобен, когда тела группируются естественным образом по их принадлежности некоторому «монолитному» объекту на стороне пользовательского приложения.

Второй способ – настроить запрос cdet_query, позволяющий пропускать тест пересечения для пары тел, принадлежащих одной и той же группе. Пример этого можно увидеть в исходном коде класса cdet_collided_faces. Для этого нужно в методе обратного вызова вернуть CBACK_SKIP по событию CDET_STARTED для пары тел, принадлежащей одной и той же группе.

ГЛАВА 7

C3D Solver

Ограничения определяют все связи между геометрическими объектами в 2D и 3D-приложениях. Они могут быть как размерными, задающими углы и расстояния, так и логическими, задающими совпадение, параллельность, перпендикулярность, касание и т.д..

Используя параметрический решатель C3D Solver, разработчики включают размерные и логические ограничения в свои приложения для создания связей между геометрическими объектами в 2D и 3D-моделях. C3D Solver сохраняет и поддерживает ограничения, когда пользователи вносят изменения в геометрию. Например, когда изменяют размер, C3D Solver мгновенно пересчитывает все зависящие от него геометрические объекты с сохранением существующих связей.

Модуль C3D Solver позволяет значительно снизить затраты на разработку программного обеспечения с функциями 2D и 3D моделирования за счет использования готовых решений поддержки геометрических и параметрических ограничений. Параметрическое ядро C3D Solver может быть использовано разработчиками САПР как на стадии первоначальной разработки программного обеспечения, так и во время любой другой из последующих стадий развития продукта.

Для активации лицензии на модуль необходимо вызвать функцию EnableMathModules() с соответствующими параметрами.

7.1 ДВУМЕРНЫЙ ГЕОМЕТРИЧЕСКИЙ РЕШАТЕЛЬ

Любой чертеж с геометрической точки зрения можно представить набором объектов плоскости – это точки, прямые, отрезки, дуги, эллипсы, сплайновые кривые, но не всякий произвольный набор плоской геометрии есть чертеж. В чертеже геометрические объекты всегда подчинены взаимосвязям, задающим положения одних объектов относительно других. Такие взаимосвязи предполагаются всегда, вне зависимости от того, чертим ли мы эскиз на бумаге с помощью карандаша или создаем его на компьютере. Если речь идет о компьютерном черчении, то такие взаимосвязи задаются с помощью *геометрических ограничений,* которые включают в себя *логические ограничения* такие, как совпадение, параллельность/перпендикулярность, касание, горизонтальность/вертикальность,

симметричность, и *размерные ограничения*, задающие разного рода *угловые и линейные размеры*, а также размеры, задающие длину кривой или радиус. Эскиз или чертеж, имеющий хотя бы одно размерное ограничение, называется *параметрическим*.

Решатель двумерных геометрических ограничений позволяет вычислять положение геометрических объектов, удовлетворяющее всем заданным ограничениям и размерам эскиза. Процесс черчения, при котором положение объектов, подчиняется заданным ограничениям и размерам называется *параметрическим черчением*.

7.1.1 Назначение геометрического решателя GCE

В составе геометрического ядра C3D решатель двумерной геометрии имеет внутреннее техническое название – GCE (*сокр. Geometric Constraint Engine*). GCE является компонентом модуля C3D Solver, включающего в себя еще один компонент – решатель трехмерных сопряжений (GCM). Основное предназначение компонента GCE – это удовлетворение системы ограничений, заданных над геометрическими объектами плоского чертежа (эскиза). Предметом вычислений решателя GCE являются геометрические объекты на плоскости: точка, прямая, окружность, эллипс, сплайн или параметрическая кривая, на которые пользователь накладывает связи в виде ограничений из заданного набора типов, включающий логические ограничения: параллельность, перпендикулярность, касание, точка на кривой, симметрия и разного рода размерные ограничения, задающие расстояния, длины и углы. Решатель также может работать с кривыми, имеющими концы (bounded curves), как например, отрезки (line segments) и дуги окружности (arcs).

Вся функциональность геометрического решателя GCE доступна через заголовочные файлы *gce_types.h* и *gce_api.h* и состоит из набора простых структур данных и функций. Система типов решателя отражает основные понятия для формулировки задачи геометрических ограничений чертежа и манипулирования состоянием геометрических объектов. Проблемная область геометрического решателя включает такие понятия, как геометрический объект, геометрическое ограничение, размер, система ограничений, числовая переменная и т.д. Взаимодействие приложения с решателем ограничений осуществляется на основе простых структур данных C++, все они объявлены в файле *gce_types.h*.

7.1.2 Встраивание в приложение

Геометрический решатель GCE построен, как общецелевой компонент параметрического черчения. Это значит, что его можно встроить в любое приложение, имеющее дело с плоской (planar) геометрией, чертежами или эскизами. Предполагается, что такое приложение уже имеет свои структуры данных, представляющие геометрию эскиза. Перед программистом, встраивающим компонент GCE в свое приложение, фактически ставится задача передать под управление решателя свои геометрические объекты, подчиненные системе ограничений. Программный интерфейс решателя имеет свою собственную отвлеченную (abstract) систему типов данных, составляющую необходимый минимум того, чтобы сформулировать задачу удовлетворения ограничений. Поэтому, чтобы подключить геометрический решатель к управлению эскизом, необходимо реализовать специальную обертку, которая, выполняя роль моста между C3D Solver и приложением, будет передавать данные о геометрии и ограничениях в решатель, и наоборот, применять результаты вычислений решателя к геометрии эскиза. На рис. 1 изображена примерная схема взаимодействия решателя ограничений с эскизом. Обертка **GCE wrapper** обслуживает параметрическую модель эскиза и выполняет следующие функции:

- загружает в решатель геометрическую модель эскиза, выраженную в типах данных модуля C3D Solver;
- обрабатывает запросы редактора такие, как добавление/удаление данных системы ограничений, вычисление эскиза, драггинг геометрических объектов;

- управляет состоянием геометрии эскиза, подчиненной ограничениям;
- обновляет данные решателя по актуальному состоянию геометрии эскиза;
- скрывает (адаптирует) API решателя с его математическими типами данных и предоставляет более удобное API с типами данных приложения;

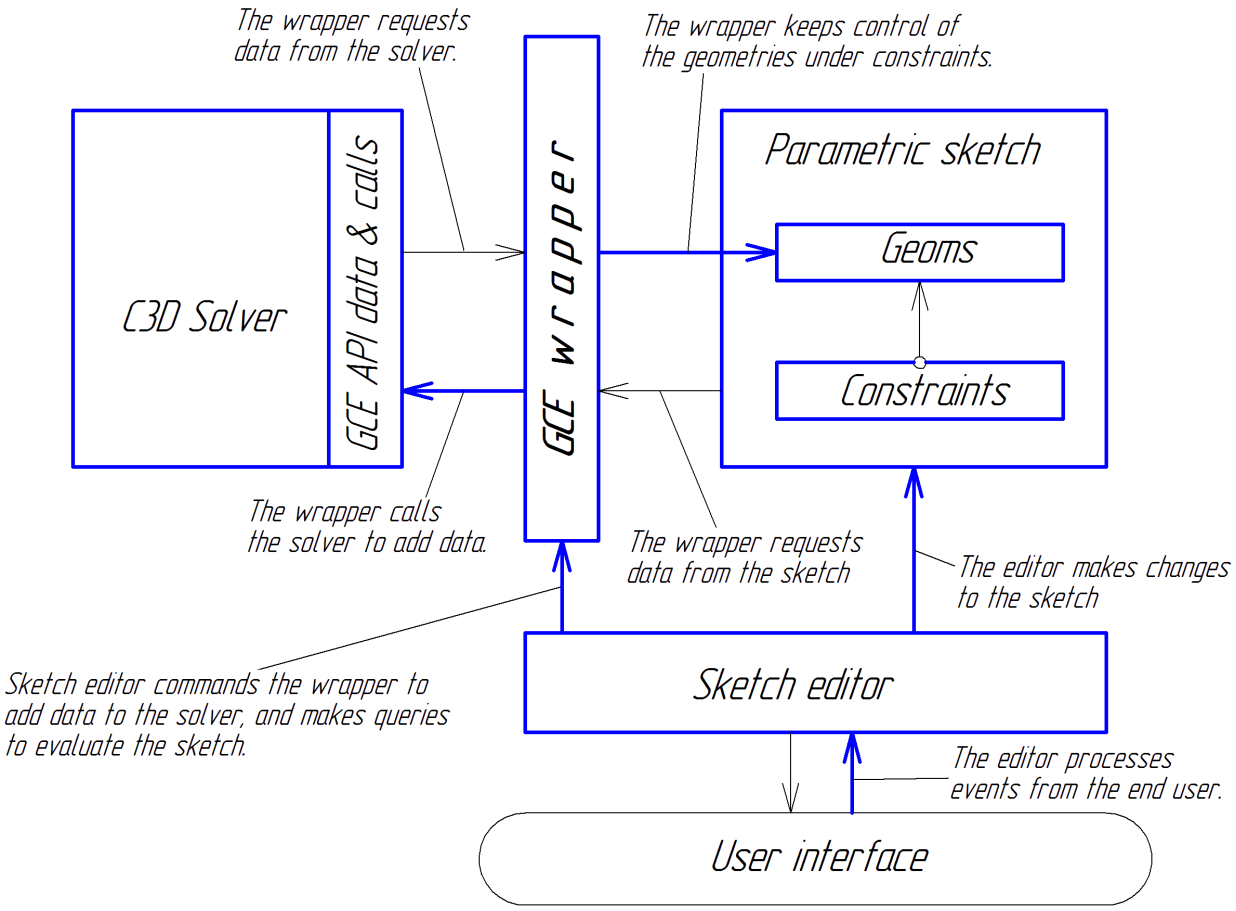


Рис. 1

Схема на данном рисунке не является непременным руководством по встраиванию решателя, а только демонстрирует пример одного из способов интеграции C3D Solver в приложение.

Также компонент GCE не позволяет сохранять данные пользователя в файл документа, поэтому разработчик приложения должен позаботиться о чтении/записи системы ограничений своего эскиза в документе приложения. Кроме того, следует принимать во внимание, что набор типов геометрического решателя не может перечислить всё многообразие геометрических типов приложения. Например, в решателе нет типа «прямоугольник», однако он может быть представлен, как четверка точек и прямых, связанных определенными ограничениями. Возможное программное решение предусматривает, что каждый чертежный объект будет иметь свой метод, формулирующий его описание в решателе.

7.1.3 Типы поддерживаемой геометрии

Геометрический решатель GCE поддерживает базовый набор типов геометрии, достаточный, чтобы на его основе строить разнообразные чертежные объекты эскиза. Базовый набор включает следующие типы геометрии:

- точка (point) – задается двумя декартовыми координатами X, Y;
- прямая (line) – задается точкой и вектором нормали;
- окружность (circle) – задается точкой центра и радиусом;
- эллипс (ellipse) – задается точкой центра, направляющим вектором и двумя значениями полуосей;
- сплайн (spline) – задается структурой данных, определяющих NURBS;
- параметрическая кривая (parametric curve) – задается объектом кривой общего типа MbCurve;

Также решатель позволяет создавать отдельные куски или дуги на основе бесконечных кривых. Для этого предусмотрены тип bounded curve и специально выделенный её частный случай – отрезок прямой (line segment):

- ограниченная кривая (bounded curve) – задается одной базовой кривой и двумя концевыми точками;
- отрезок прямой (line segment) – задается парой точек;

7.1.4 Типы геометрических ограничений и размеров

Любое геометрическое ограничение связывает между собой геометрические объекты, называемые *аргументами* ограничения. Аргументом ограничения может быть любой объект, относящийся к одному из типов геометрии, перечисленных в параграфе *Типы поддерживаемой геометрии*. Например, окружность может быть аргументом ограничения **касание**. По числу геометрических аргументов ограничения различают, как унарные, бинарные и тернарные. Они связывают соответственно один, два или три объекта. Примером тернарного ограничения является **симметрия**, в которой участвуют три объекта: две точки и ось симметрии. В общем случае ограничение может иметь N аргументов. *Логические ограничения* предусматривают только зависимости между геометрическими объектами. *Размерные ограничения* устанавливают связь между геометрическими объектами и числовым параметром, означающим какой-то линейный или угловой размер, поэтому у размерных ограничений последний аргумент всегда числовой, называемый также *скаляром*. Типы ограничений, которые поддерживает геометрический решатель GCE приведены в таблице 1:

Таблица 1 - Поддерживаемые типы ограничений

Логические ограничения	Размерные ограничения	Специальные ограничения
фиксация	длинна отрезка	управляющий параметр
горизонтальность	радиус	фиксация производной сплайна
вертикальность	расстояние	точка на кривой с позицией по параметру кривой
длина отрезка	направленное расстояние	линейное уравнение вида $a1 * v1 + a2 * v2 + ... + an * vn + c = 0$
совпадение	угол	
точка на кривой		
выравнивание точек		
коллинеарность		
равенство длин		
равенство радиусов		
параллельность		
перпендикулярность		
касание		
средняя точка		
зеркальная симметрия		
биссектриса		

Более подробно каждое из этих ограничений будет рассмотрено ниже. Логические ограничения рассматриваются в главе *ДВУМЕРНЫЕ ЛОГИЧЕСКИЕ ОГРАНИЧЕНИЯ*. Размерные ограничения и вопросы параметризации эскиза рассматриваются в главе *ДВУМЕРНЫЕ РАЗМЕРЫ*.

7.1.5 Базовые типы данных API решателя GCE

Взаимодействие приложения с решателем ограничений осуществляется на основе простых структур данных C++, все они объявлены в файле *gce_types.h*. Среди них центральное место занимают дескрипторные типы данных, которые служат для идентификации любых объектов, находящихся под контролем решателя, см. табл. 1:

Таблица 1 - Дескрипторные типы данных

Тип данных решателя	Реализация	Интерпретация
GCE_system	void *	дескриптор системы ограничений
geom_item	size_t	дескриптор геометрического объекта
constraint_item	size_t	дескриптор ограничения
var_item	size_t	дескриптор числовой переменной

Типы данных, которые перечисляют конечные наборы значений, приведены в табл. 2

Таблица 2 - Перечислительные типы данных

Тип данных решателя	Интерпретация
geom_type	тип геометрического объекта
constraint_type	тип ограничения
point_type	тип запрашиваемой контрольной точки
coord_name	имя координаты геометрического объекта
GCE_result	диагностический код выполнения функции
GCE_bisec_variant	выбор между решениями биссектрисы

Структуры данных, перечисленные ниже в табл. 3, служат для передачи набора параметров объекта. Например, с помощью структуры GCE_ellipse в решатель передаются данные эллипса: координаты центра, размеры полуосей и направляющий вектор.

Таблица 3 - Структуры данных

Тип данных решателя	Интерпретация
GCE_vec2d	координаты двумерного вектора
GCE_point	координаты точки на плоскости
GCE_point_dof	запись степени свободы точки
GCE_line	координаты прямой
GCE_circle	координаты окружности
GCE_ellipse	координаты эллипса
GCE_spline	координаты и характеристики сплайна
GCE_dim_pars	параметры размерного ограничения
GCE_adim_pars	параметры углового размерного ограничения
GCE_ldim_pars	параметры линейного размерного ограничения
GCE_dragging_point	контрольная точка драггинга

7.1.6 Система геометрических ограничений

Система геометрических ограничений это набор геометрических объектов и ограничений, связывающих эти объекты. Типы поддерживаемых геометрических объектов и ограничений приведены соответственно в параграфах *Типы поддерживаемой геометрии*, *Типы геометрических ограничений и размеров*. Предполагается, что параметрический эскиз, который создает приложение, с его объектами и ограничениями, имеет свою систему ограничений. С программной точки зрения это означает, что каждый параметрический эскиз ассоциируется с системой ограничений через дескриптор типа GCE_system, а каждый его геометрический объект и ограничение представлены своими уникальными дескрипторами типов geom_item и constraint_item.

Перед тем как начать работать с геометрическими ограничениями эскиза, необходимо объявить для него систему ограничений с помощью вызова функции

```
GCE_system GCE_CreateSystem().
```

Функция вернет систему ограничений в виде дескриптора GCE_system, который представляет собой указатель на экземпляр внутренней структуры данных геометрического решателя. Все дальнейшие манипуляции с системой ограничений будут осуществляться через данный дескриптор. Например, если мы хотим объявить точку в эскизе, необходимо для его системы ограничений вызвать функцию

```
geom_item GCE_AddPoint(GCE_system gSys, GCE_point pVal).
```

Результатом функции является дескриптор геометрической точки, принадлежащей системе ограничений **gSys**. Параметр **pVal** задает начальные координаты <X,Y> точки.

При завершении работы с эскизом обязательно следует вызвать функцию удаления системы ограничений

```
void GCE_RemoveSystem(GCE_system gSys),
```

которая полностью освободит память, занимаемую системой ограничений со всеми данными объектов и ограничений. После вызова **GCE_RemoveSystem** дескриптор системы ограничений становится недействительным, т.е. дальнейшее использование такого дескриптора может привести к аварийному прекращению работы приложения.

Функция

```
void GCE_ClearSystem(GCE_system gSys)
```

делает систему ограничений пустой, освобождая память только от объектов и ограничений, но оставляет систему ограничений действительной для дальнейшей работы.

7.1.7 Представление геометрических объектов

Решатель геометрических ограничений работает с определенной формой представления геометрических объектов, проиллюстрированной на рис. 1. Все объекты выражаются через координаты точек, вектора и числа (скаляры).

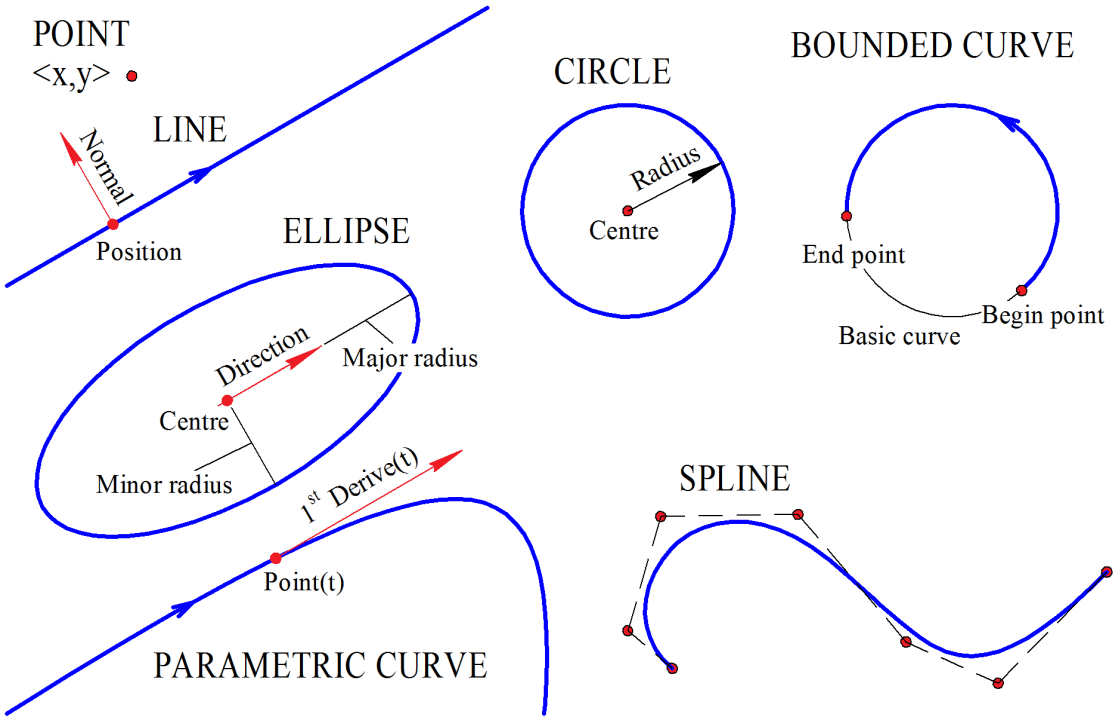


Рис. 1

Приложение может иметь свое представление геометрических объектов, отличное от решателя, однако передача данных о состоянии объектов в решатель и получение обратно результатов вычислений основываются на том, что каждый тип геометрии имеет свое представление:

- **Точка** представлена парой декартовых координат <X,Y>.
- **Прямая** задается точкой позиции и вектором нормали. Предполагается, что у прямой есть направляющий вектор, который равен вектору нормали повернутому по часовой стрелке на 90 градусов. Другими словами, направляющему вектору с координатами <X,Y> будет соответствовать нормаль с координатами <Y,-X>.
- **Окружность** задается точкой центра и радиусом. C3D Solver поддерживает окружности нулевого или положительного радиуса. Имеется возможность допустить или запретить нулевой радиус для окружности в индивидуальном порядке, для чего следует воспользоваться функцией обратного вызова **GCE_allow_zero_radius**, разрешающей нулевой радиус.

  Замечание: Необходимость опции нулевого радиуса возникает при решении следующих задач:

  – построение параметрического контура с исчезающими и вновь появляющимися дугами окружности;

  – построение параметрических 3D-моделей с переменной топологией на основе 2D-контуров;

  – построение поверхности по траектории переменного сечения.

  Одним из примеров применения опции нулевого радиуса является моделирование поверхности воздуховода переменного сечения, а именно прямоугольное сечение со скруглениями переходит в прямоугольник без скруглений.
- **Эллипс** задается точкой центра, радиусами по «главной» (major) и по «минорной» (minor) полуосям и направляющим вектором главной полуоси. Также эллипс параметризуется величиной с периодом от ***0*** до ***2***π, пробегающей по эллипсу против часовой стрелки, начиная от точки на «главной» полуоси.
- **Параметрическая кривая** передается в решатель в виде класса MbCurve. Такая кривая рассматривается как неподвижная, а вычисления связанные с этой кривой, строятся на виртуальных функциях: MbCurve::PointOn, MbCurve::FirstDer и MbCurve::SecondDer, выдающие соответственно точку на кривой по параметру, первую и вторую производную в этой точке. Подробнее о параметрических кривых см. параграф *Параметрические кривые общего вида*

Замечание: Пока для создания параметрической кривой в системе ограничений потребуется создать экземпляр класса MbCurve, однако в будущем будет реализован альтернативный вариант, основанный на простых функциях, реализуемых самим пользователем. Также пользователь ядра C3D имеет возможность реализовать своего наследника для MbCurve.

- **Сплайн** основан на NURBS-представлении, в основе которого лежит список контрольных точек. Более подробно работа со сплайном будет рассмотрена в главе *ДВУМЕРНЫЕ СПЛАЙНЫ И ПАРАМЕТРИЧЕСКИЕ КРИВЫЕ*
- **Ограниченная кривая** представляет собой участок кривой, ограниченный концевыми точками. Задается тройкой: базовая кривая, начальная точка участка, конечная точка участка. C3D Solver поддерживает опцию нулевой длины для ограниченной кривой и отрезка. Для того, чтобы допустить или запретить нулевую длину кривой в индивидуальном порядке необходимо воспользоваться функцией обратного вызова **GCE_allow_zero_length**. По умолчанию опция нулевой длины отключена.

7.1.8 Степень свободы

Каждому типу геометрии свойственна степень свободы, равная минимальному числу координат, необходимых для определения состояния геометрического объекта. Например, степень свободы двумерной точки равна 2. Окружность имеет степень свободы 3, поскольку она полностью определяется тройкой значений <X,Y,R> – координатами центра и радиусом. Согласно параграфу *Представление геометрических объектов* Представление геометрических объектов, прямая представляется точкой позиции и вектором нормали, однако это удобное, но избыточное представление; минимально-достаточным представлением прямой могла бы быть пара значений смещения и угла наклона, поэтому прямая имеет степень свободы 2. Также эллипс имеет степень свободы равную 5. Степень свободы сплайна равна сумме степеней свобод его контрольных точек. Параметрическая кривая полностью определена на стороне приложения, т.е. её степень свободы равна 0. В таблице 1 приведены степени свободы всех типов, поддерживаемых решателем.

Таблица 1 - Степень свободы геометрических объектов

Тип геометрии	Степень свободы
Точка	2
Прямая	2
Окружность	3
Эллипс	5
Сплайн	2 · Количество контрольных точек
Параметрическая кривая	0
Ограниченная кривая	2 + Степень свободы базовой кривой

Каждый геометрический объект, участвующий в системе, добавляет определенное количество своих степеней свобод к общей степени свободы эскиза. С другой стороны, каждое добавляемое ограничение отнимает определенную степень свободы. Для того, чтобы определить состояние всех геометрических объектов эскиза, требуется определенное количество ограничений, отнимающее все степени свободы объектов.

Большинство ограничений отнимает одну степень свободы, это такие ограничения, как параллельность, перпендикулярность, горизонтальность/вертикальность, равенство радиусов, равенство длин, точка на прямой, касание и большая часть размерных ограничений. Другие ограничения снимают две степени свободы: средняя точка, коллинеарность, симметрия, биссектриса.

Можно сказать, что задача параметрического черчения состоит в том, чтобы полностью определить геометрические объекты эскиза. Количество геометрических ограничений, которые необходимы для полного определения эскиза, соответствуют сумме степеней свобод всех объектов, входящих в эскиз.

7.1.9 Добавление и удаление геометрических объектов

Основным предметом вычислений геометрического решателя являются геометрические объекты, поэтому формирование системы ограничений начинается с того, что приложение объявляет в системе ограничений геометрические объекты, которым предстоит стать аргументами ограничений.

Каждый геометрический объект, объявленный в системе ограничений, получает свой уникальный идентификатор – дескриптор типа geom_item, а его геометрический тип (geom_type) остается неизменным в течении жизни объекта. Ниже приведены вызовы API C3D Solver, добавляющие геометрические объекты в систему.

Точка добавляется с помощью метода

```
geom_item GCE_AddPoint(GCE_system gSys, GCE_point pVal).
```

Результатом функции является дескриптор геометрической точки, принадлежащей системе ограничений **gSys**. Параметр **pVal** задает начальные значения координат точки <X,Y>.

Прямая добавляется с помощью метода

```
geom_item GCE_AddLine(GCE_system gSys, GCE_line lVal).
```

Результатом функции является дескриптор прямой на плоскости, принадлежащей системе ограничений **gSys**. Параметр **lVal** задает начальные значения позиции **lVal.p** и нормали **lVal.norm** прямой.

Окружность добавляется с помощью метода

```
geom_item GCE_AddCircle(GCE_system gSys, GCE_circle cVal).
```

Результатом функции является дескриптор окружности, принадлежащей системе ограничений **gSys**. Параметр **cVal** задает начальные значения центра **cVal.centre** и радиуса **cVal.radius**.

Эллипс добавляется с помощью метода

```
geom_item GCE_AddEllipse(GCE_system gSys, GCE_ellipse eVal).
```

Результатом функции является дескриптор эллипса, принадлежащего системе ограничений**gSys**. Параметр **eVal** определяет начальные значения эллипса:

<**eVal**.centre, **eVal**.direct, **eVal**.majorR, **eVal**.minorR>, означающие соответственно центр, направляющий вектор и радиусы по главной и минорной осям.

Сплайн добавляется с помощью метода

```
geom_item GCE_AddSpline(GCE_system gSys, GCE_spline spl).
```

Результатом функции является дескриптор сплайна, принадлежащего системе ограничений **gSys**. Структура данных **spl** определяет сплайн в начальном состоянии. Подробности, связанные со сплайновой кривой и способами его определения, рассмотрены в параграфе *Сплайновые кривые*.

Параметрическая кривая общего вида добавляется с помощью метода

```
geom_item GCE_AddParametricCurve(GCE_system gSys, const MbCurve & crv).
```

Результатом функции является дескриптор параметрической кривой, добавленной в систему ограничений **gSys**. Объект **crv** полностью определяет абстрактное математическое описание кривой в параметрической форме. Подробнее класс двумерной кривой общего вида обсуждается в параграфе *Двумерная кривая MbCurve*. Надо заметить, что тип данных MbCurve входит в иерархию классов геометрического ядра C3D, а значит его время жизни регулируется счетчиком ссылок. Другими словами система ограничений гарантирует действительность экземпляра MbCurve, пока параметрическая кривая не будет удалена из системы. Подробнее о работе с параметрической кривой рассказывается в параграфе *Параметрические кривые общего вида*.

Ограниченная кривая добавляется с помощью метода

```
geom_item GCE_AddBoundedCurve(GCE_system gSys, geom_item curve, geom_item p[2]).
```

Результатом функции является дескриптор ограниченной кривой, принадлежащей системе ограничений **gSys**. Созданная кривая основывается на кривой **curve** и будет представлять её участок, ограниченный концевыми точками **p**[0] и **p**[1]. Данный метод фактически связывает три объекта – базовую кривую и две точки в единый объект и автоматически обеспечивает два ограничения «точка на кривой» для концевых точек. Фрагмент кода ниже показывает пример создания **дуги окружности**:

```
GCE_system gSys = GCE_CreateSystem();
GCE_circle circPars; // Circle values
GCE_point endP1, endP2; // Coordinates of the end points.
// ...
// It is required to insert the code that assigns start values of the circle and its␣
↪endpoints.
// ...
geom_item p[2] = { GCE_AddPoint(gSys, endP1), GCE_AddPoint(gSys, endP2) };
geom_item circItem = GCE_AddCircle( gSys, circPars ); // Create a circle. It's a base␣
↪curve of the arc.
geom_item arc = GCE_AddBoundedCurve( gSys, circItem, p ); // Create the circular arc.
// ...
GCE_RemoveSystem( gSys );
```

Аналогично с помощью вызова **GCE_AddBoundedCurve**() можно создавать дуги эллипса, участки на параметрической кривой, отрезки прямой и т.д. Ограниченная кривая, созданная на основе прямой, называется **отрезком**, а все ограничения действующие для прямой также применимы для отрезка.

Любой геометрический объект, добавленный одним из вышеперечисленных методов, свободный от ограничений или размеров, может быть удален вызовом метода

```
bool GCE_RemoveGeom(GCE_system gSys, geom_item g).
```

Данный метод удаляет геометрический объект из системы и делает его дескриптор **g** не действительным. Функция вернет **true** в случае успешного выполнения. Функция вернет значение **false**, а удаление не будет исполнено, если геометрический объект на момент вызова является аргументом одного из ограничений. Чтобы узнать доступен ли геометрический объект для удаления методом **GCE_RemoveGeom**, можно воспользоватьсяь функцией **GCE_IsConstrainedGeom**. Допускается удаление геометрического объекта, на котором основаны другие объекты, раньше других. Например, точки концов или базовая кривая могут быть удалены раньше граничной кривой, которая на них основана, однако их неизбежное удаление откладывается до тех пор, пока не будет удален последний из объектов, в которых они участвуют.

7.1.10 Фиксация и заморозка геометрических объектов

Любой геометрический объект, созданный в системе ограничений, изначально свободен, т.е. имеет полную степень свободы, свойственную его типу. При вычислениях C3D Solver может менять состояние геометрических объектов, когда это требуется для удовлетворения ограничений. Иногда требуется обездвижить часть геометрии так, чтобы решатель оставлял положение геометрического объекта без изменений. Для этого в API GCE предусмотрены вызовы, которые могут **заморозить** или **зафиксировать** геометрический объект. Состояние замороженных или фиксированных геометрических объектов может поменять только приложение с помощью вызовов:

GCE_SetPointXY и **GCE_SetCoordValue**.

Заморозку геометрических объектов удобно применять, когда требуется обеспечить неподвижность какой-то части чертежа. Например, в системе САПР часть геометрии чертежа может быть получена проецированием из трехмерной модели, и поэтому навсегда связана с ней односторонней ассоциативной связью: «объемная модель → плоское отображение». Заморозить геометрический объект можно с помощью метода

```
bool GCE_FreezeGeom(GCE_system gSys, geom_item g).
```

Метод вернет значение **true**, если геометрический объект **g** стал фактически замороженным. Надо заметить, что заморозка не считается ограничением, поэтому метод **GCE_IsConstrainedGeom** не принимает во внимание заморожен объект или нет. Степень свободы замороженного объекта равна 0.

Альтернативный способ сделать геометрический объект неподвижным для решателя – это задать ограничение фиксация объекта с помощью метода

```
constraint_item GCE_FixGeom(GCE_system gSys, geom_item g).
```

Функция зафиксирует объект **g**, принадлежащий системе ограничений **gSys**, и вернет дескриптор вновь добавленного ограничения «фиксация объекта». В отличие от **GCE_FreezeGeom** данный метод создает ограничение, которое в любой момент можно удалить методом **GCE_RemoveConstraint** и тем самым освободить объект. Ограничение фиксация рассмотрена также в параграфе *Унарные ограничения: горизонтальность/вертикальность и варианты фиксации*

7.1.11 Контрольные точки геометрических объектов

Среди геометрических типов, поддерживаемых решателем (см. параграф *Типы поддерживаемой геометрии*) особенное место занимает **точка**. Во-первых, точка самый элементарный геометрический объект. Во-вторых, через точки, называемые *контрольными*, выражаются все остальные типы геометрии. Например, окружность выражается структурой <C,R>, где C – точка центра, R – скаляр, числовое значение радиуса. В-третьих, контрольные точки могут самостоятельно участвовать в ограничениях, в качестве их аргументов. Например, можно задавать ограничения, как для самой окружности, так и отдельно для её точки, скажем, расстояние от центра окружности до прямой, см. рис. 1.

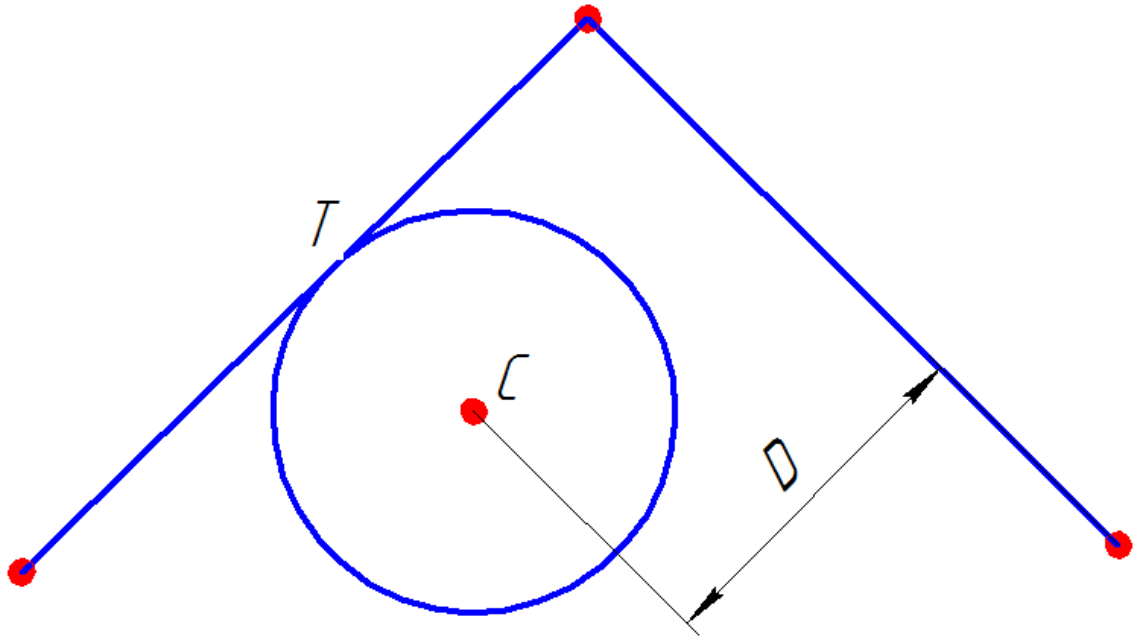


Рис. 1

C – контрольная точка центра

T – касание окружности и прямой

D – расстояние от центра до прямой

Чтобы работать с контрольной точкой, как с самостоятельной точкой, необходимо запросить её дескриптор у геометрического объекта с помощью функции:

```
geom_item GCE_PointOf(GCE_system gSys, geom_item g, point_type pnt).
```

Результатом функции является дескриптор контрольной точки геометрического объекта **g**, принадлежащего системе ограничений **gSys**. Параметр **pnt** определяет какая контрольная точка объекта запрашивается и принимает одно из следующих значений:

- GCE_FIRST_END – начальная точка кривой (первый конец);
- GCE_SECOND_END – конечная точка кривой (второй конец);
- GCE_CENTRE – центр окружности, эллипса, а также их дуги (bounded curve);
- GCE_Q1 – квадрантная точка эллипса (3 часа);
- GCE_Q2 – квадрантная точка эллипса (12 часов);
- GCE_Q3 – квадрантная точка эллипса (6 часов);
- GCE_Q4 – квадрантная точка эллипса (9 часов).

Запросить контрольные точки сплайновой кривой позволяет метод

```
geom_item GCE_SplinePoint(GCE_system gSys, geom_item spl, size_t pntIdx).
```

Функция вернет дескриптор контрольной точки сплайна **spl**, взятой по номеру **pntIdx**, принимающий значения от 0 до N-1, где N – число контрольных точек сплайна. На рис. 2

изображен сплайн, имеющий N = 7 контрольных точек (красные кружки), с нумерацией от 0.

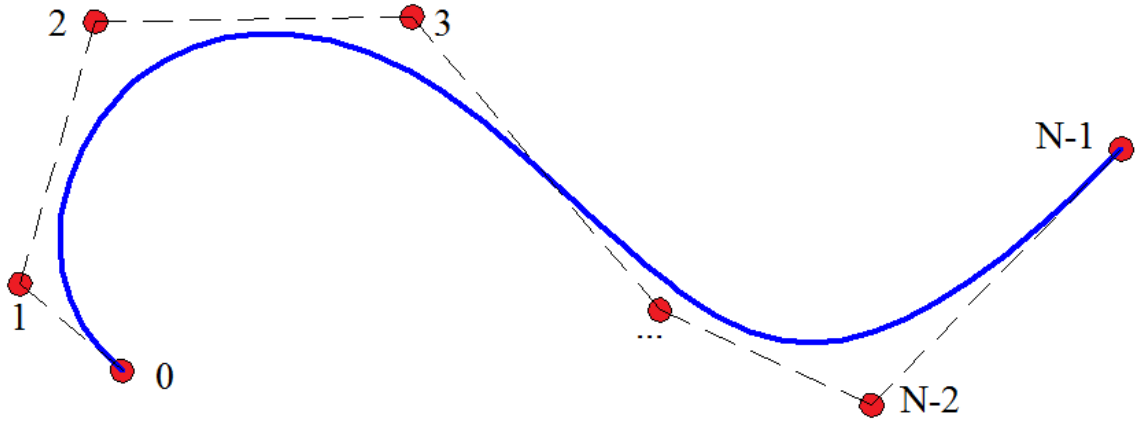


Рис. 2

7.1.12 Скалярная переменная

Основным предметом вычисления модуля GCE являются геометрические объекты двумерной плоскости, подчиненные ограничениям и размерам. Эти типы данных по своей природе имеют векторное представление. Параметрическое черчение предполагает связь геометрии с управляющими либо вариационными числовыми параметрами. Поэтому дело не обходится без еще одного типа – **числовой переменной** или **скаляра** со значением, заданным, как правило, в единицах длины или угла.

В параметрической системе ограничений могут участвовать переменные такого рода:

- Числовая переменная, ассоциированная с размером – **размерная переменная**. В решенном состоянии размерная переменная равна измеренному значению размера. Размерная переменная может служить как управляющим, независимым параметром эскиза, так и вариационным или измерительным. Подробнее об использовании переменных для создания управляющих и вариационных размеров обсуждается в параграфе *Управляющие и вариационные размеры*.
- Вспомогательный параметр геометрического ограничения, например, параметр, указывающий локализацию точек касания для кривых сложной формы. Подробности в параграфе *Касание*.
- Переменная, не ассоциированная с геометрическим ограничением и размером, может участвовать в скалярных линейных или нелинейных уравнениях, устанавливающих законы связи одних величин с другими.

В API модуля GCE переменную представляет тип данных var_item. А создать переменную можно вызовом метода

```
var_item GCE_AddVariable(GCE_system gSys, double val).
```

Функция вернет дескриптор скалярной переменной с начальным значением **val**. В дальнейшем переменную по данному дескриптору можно применять для определения размеров, ограничений и уравнений.

Любая переменная может быть **зависимой** или **независимой**. Зависимая переменная подчинена измерительным размерам или ограничениям, а её значение вычисляется решателем. Независимая переменная не может быть изменена решателем, а её значение

задается приложением. Чтобы переменная стала независимой, достаточно вызвать метод API

```
constraint_item GCE_FixVariable(GCE_system gSys, var_item var),
```

который создаст ограничение «управляющий параметр» и вернет дескриптор ограничения, фиксирующий её значение. Независимая переменная доступна для варьирования с помощью вызова **GCE_ChangeDrivingDimension**. Ограничение «управляющий параметр» вместе с другими типами управляющих размеров позволяют создать параметрический эскиз, геометрия которого может меняться по определенному пользователем закону варьированием независимых параметров. Про создание управляющих параметров и размеров см. параграф *Управляющие и вариационные размеры*.

7.1.13 Линейное уравнение

Линейное уравнение выражается формулой $a1v1 + a2v2 + a3v3 + \ldots + anvn + \ = 0$, где a1, a2, a3, ..., an – постоянные коэффициенты, v1, v2, v3, ...,vn – скалярные переменные уравнения. Линейное уравнение создается вызовом метода

```
constraint_item GCE_AddLinearEquation(      GCE_system gSys,
                                      const double *   a,
                                      const var_item * v,
                                            size_t     n,
                                            double     c),
```

который вернет дескриптор линейного уравнения, зарегистрированного в системе **gSys**.

Исходные данные вызова:

- **gSys** – система ограничений;
- **a** – массив постоянных коэффициентов при переменных;
- **v** – массив переменных линейного уравнения;
- **n** – количество переменных линейного уравнения;
- **c** – постоянный коэффициент «c» без переменной;

7.1.14 Журналирование вызовов API

C3D Solver позволяет для конкретной системы ограничений GCE_system записать всю историю вызовов API, включая исходные параметры и возвращаемые значения вызванных функций. Результатом записи будет файл журнала, который в дальнейшем может быть использован разработчиками геометрического ядра C3D для отладки и тестирования функций C3D Solver. ***Устройство журнала***

Файл журнала имеет текстовый формат, основанный на простом синтаксисе S-выражений. Любая функция API, вызванная для конкретной системы ограничений, фиксируется в журнале в следующем виде:

```
(GCE_Func (arg list)(returned value)),
```

где *GCE_Func* – имя вызванной GCE-функции, (*arg list*) – список аргументов функции, (*returned value*) – возвращаемое значение.

Например, так выглядит запись вызова, создающего окружность с центром в начале координат и радиусом 2.5:

```
(GCE_AddCircle ((0.0 0.0) 2.5) #1)
```

Окружность получила символический идентификатор #1, который является возвращаемым значением функции GCE_AddCircle.

Журнал фактически является записью сеанса работы с конкретной системой ограничений на языке, который может быть интерпретирован специальными средствами, которыми располагает разработчик C3D Solver. Также журнал может быть отредактирован вручную или упрощен, если для достижения определенной ситуации не требуются все вызовы.

Включение режима журналирования

Чтобы включить режим записи журнала системы ограничений, необходимо вызвать функцию

```
bool GCE_SetJournal(GCE_system gSys, const char * fName),
```

которая для системы ограничений **gSys** назначит файл журнала **fName**. В случае успешной инициализации режима функция вернет **true**.

Входными параметрами метода являются:

- **gSys** – пустая система ограничений, созданная методом **GCE_CreateSystem**,
- **fName** – строка, задающая полный путь к файлу. Например: «*..Journals\sample.jrn*»

После включения журналирования будет осуществляться запись информации о всех обращениях к API C3D Solver, адресованных системе ограничений **gSys**.

Внимание: файл журнала будет готов только после завершения сеанса работы с системой ограничений, а именно сразу после вызова **GCE_RemoveSystem**().

Пример кода демонстрирующего запись сеанса работы системы ограничений в журнал приведен ниже:

```
GCE_system gSys = GCE_CreateSystem();
// Switch on journalling for the constraint system.
GCE_SetJournal(gSys, "C:\\Logs\\gce_sample.jrn");
// Make some calls of C3D Solver ...
GCE_point p1Val, p2Val; // Coordinates of two points = (1,1) and (2,2)
p1Val.x = p1Val.y = 1.0;
p2Val.x = p2Val.y = 2.0;
geom_item pnt[2] = { GCE_AddPoint(gSys, p1Val), GCE_AddPoint(gSys, p2Val) };
GCE_AddCoincidence(gSys, pnt);
// ...
// ...
GCE_Evaluate(gSys);
// Finalize the constraint system
GCE_RemoveSystem(gSys);
// The journal file is ready!
```

Замечание

Журналирование нужно только для отладочных целей, поэтому рекомендуется включать её только в режиме отладки. Не следует оставлять журналирование включенным для продуктовой (release) или рабочей версии приложения, т.к. это может приводить к напрасному расходованию ресурсов времени и памяти.

Отладка и тестирование

Журналирование дает возможность разработчику геометрического ядра отлаживать функций C3D Solver путем воспроизведения журнала вызовов, записанного при работе в приложении. Данный способ использует тот факт, что внутреннее состояние системы ограничений полностью определяется последовательностью вызовов API, поэтому достаточно повторить ту же последовательность вызовов с теми же данными, чтобы полностью воспроизвести проблему, возникшую при эксплуатации решателя приложением.

Обширная коллекция журналов образует базу тестовых примеров, которая является неотъемлемым инструментом для развития и контроля качества модуля C3D Solver. Каждая ревизия C3D Solver проверяется на тестовой базе журналов, что гарантирует сохранение результатов всех ранее сделанных исправлений и усовершенствований.

Таким образом, чтобы получить консультацию или решить проблемную ситуацию, связанную со встраиванием C3D Solver в приложение, достаточно временно включить режим журналирования, выполнить интересующий сценарий действий и переслать полученный файл с необходимыми комментариями или вопросами в техническую поддержку C3D Labs. После исправления ошибки соответствующий журнал добавляется в базу тестовых примеров.

7.1.15 Функции обратного вызова

Обратные вызовы – это отдельный вид функций API C3D Solver, особенность которых в том, что они вызываются изнутри C3D Solver и могут быть определены на стороне приложения. Необходимость в таких функциях возникает, когда требуется предоставить решателю доступ к данным, хранящимся на стороне приложения, определить свойства геометрических объектов, а также в порядке обратной связи обеспечить решатель запросами, влияющими на ход вычислений. Обратные вызовы расширяют возможности для интеграции решателя с приложением, предоставляя еще один канал взаимодействия. Все функции и типы данных для организации обратных вызовов собраны в отдельном заголовочном файле *gce_callback.h*.

Для того, чтобы функции обратного вызова, определенные пользователем, заработали для данной системы ограничений, необходимо их зарегистрировать с помощью вызова

```
GCE_result GCE_Register(      GCE_system            gSys,
                        const GCE_callback_table &  cbTable)
```

Входные данные:

- **gSys** – система ограничений, которую будут обслуживать обратные вызовы;
- **cbTable** – таблица обратных вызовов, определяющая следующий набор функций:
  - **gRegister** – функция вида `void (*GCE_geom_registered)(GCE_app_geom ag)`, вызывается решателем, когда геометрический объект пользователя регистрируется в системе **gSys** с помощью вызова **GCM_Bind**;
  - **gUnregister** – функция вида `void (*GCE_geom_unregistered)(GCE_app_geom ag)`, вызывается для геометрического объекта **ag** в тот момент времени, когда геометрический решатель освобождает объект **ag**, прекращая с ним всякую работу. Данное событие наступает при таких прямых вызовах, как **GCE_RemoveGeom**, **GCE_RemoveSystem** или **GCE_ClearSystem**.
  - **abortFunc** – функция запроса на прерывание вычислений. Это булевозвратная функция вида bool (***GCE_abort**)(), которая время от времени вызывается в решателе в процессе работы **GCE_Evaluate** и прекращает вычисления, если функция вернет значение **true**. Функция **GCE_Evaluate**, работа которого была прервана таким образом, вернет код ошибки **GCE_RESULT_Aborted**.

- **allowZeroRadius** – функция обратного вызова вида `bool (*GCE_allow_zero_radius)(GCE_app_geom ag)`, которая сообщает системе ограничений о том, что окружность **ag** допускает решение, при котором ее радиус равен нулю.
- **allowZeroLength** – функция обратного вызова вида `bool (*GCE_allow_zero_length)(GCE_app_geom ag)`, которая сообщает системе ограничений о том, что отрезок или ограниченная кривая **ag** допускает решение, при котором их длина равна нулю. Например, приложение САПР с помощью опции нулевой длины может запретить отрезок нулевой длины, если он изображает на чертеже ось зеркальной симметрии. Для такого запрета приложению следует вернуть false при вызове allowZeroLength.
- **getLSegDirection** – функция обратного вызова вида `GCE_vec2d(*GCE_get_lseg_direction)(GCE_app_geom ag)`, позволяет получить направляющий вектор отрезка **ag**. Она предназначена для того, чтобы приложение не забывало куда смотрит отрезок L=0, когда потребуется снова вернуть ему длину. А также, чтобы избежать эффекта выворачивания отрезка при участии в таких ограничениях, как касание и на расстоянии.

Возвращаемое значение:

- **GCE_RESULT_Ok**, если таблица обратных вызовов была успешно зарегистрирована для системы ограничений;
- **GCE_RESULT_None**, если регистрация обратных вызовов не выполнена.

Замечание:

Для корректной работы решателя вызов ***GCE_Register*** следует вызвать однократно для одной и той же системы ограничений, в тот момент, пока она еще пуста, например сразу после GCE_CreateSystem.

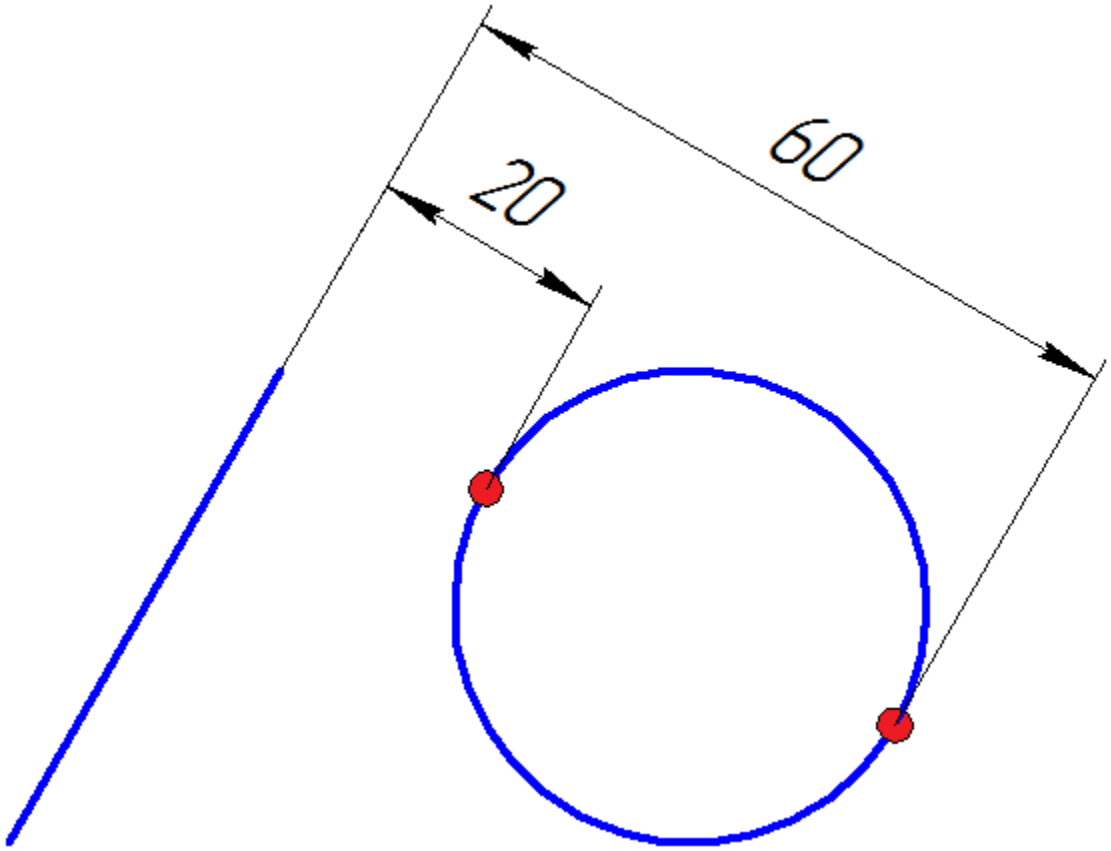
7.2 ДВУМЕРНЫЕ РАЗМЕРЫ

Размеры представляют собой ограничения, связывающие геометрические объекты с числовым параметром (значением размера). Каждый размер измеряет какую-то числовую характеристику геометрии, например, расстояние между точками, угол между прямыми, радиус окружности. Логические ограничения отличаются от размеров тем, что не имеют ни одного числового аргумента. Каждый размер всегда имеет, кроме геометрических аргументов, один числовой (скаляр).

О том, как с помощью параметров и размеров можно управлять геометрией эскиза, обсуждается в параграфе *Управляющие и вариационные размеры*.

7.2.1 Вспомогательные точки линейного размера

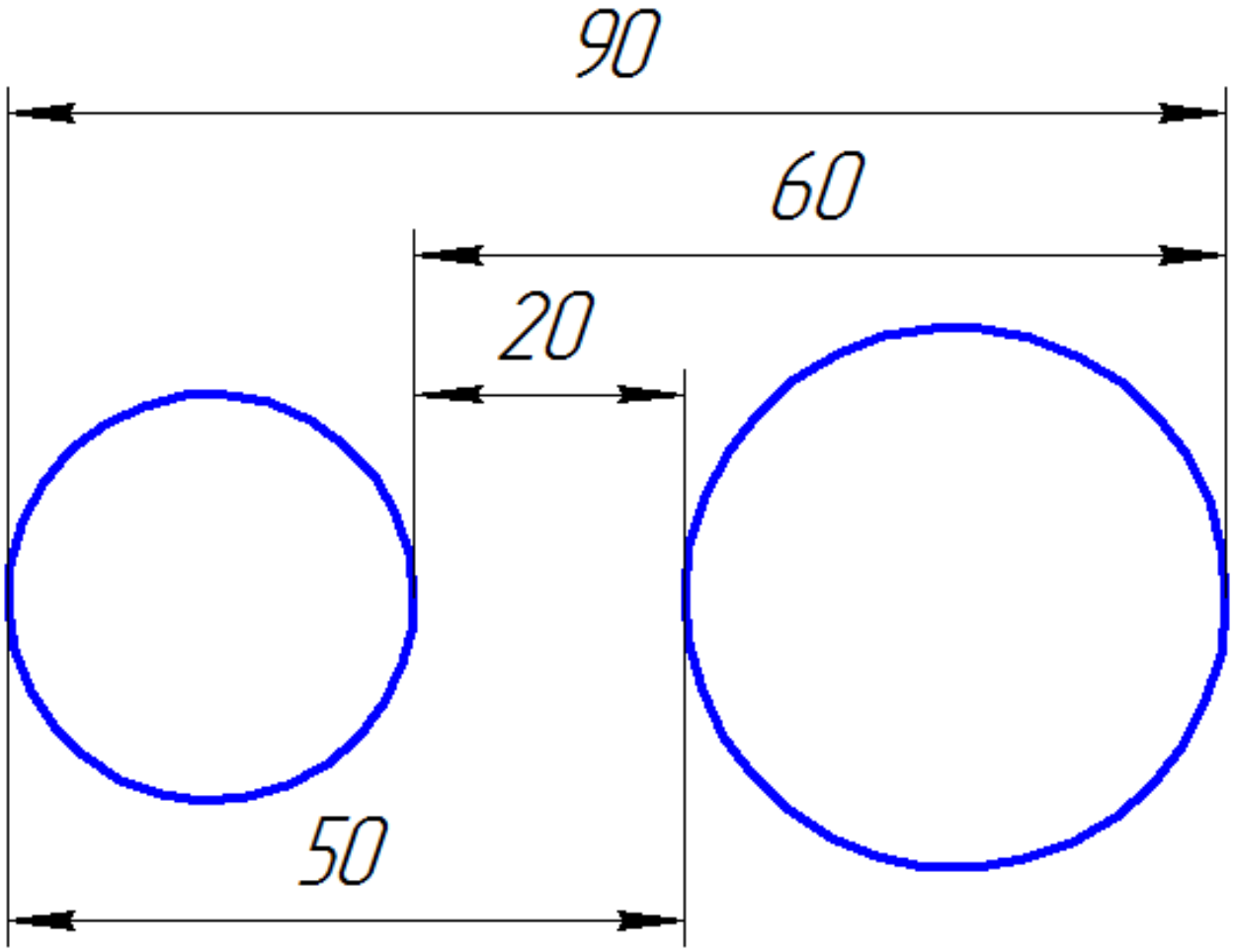
Линейные размеры (distance dimension) для некоторых типов объектов, а также диаметральные размеры не могут быть однозначно заданы без дополнительного указания точек приложения размера. Например, расстояние от окружности до отрезка прямой может измеряться двумя способами: от «дальней» точки приложения к окружности до отрезка либо от «ближней» точки до отрезка, см. рис. 1 . Понятия дальний и ближний взяты в кавычки, т.к. ситуация может поменяться при варьировании размера и тогда дальняя точка станет ближе к отрезку, чем ближняя>.



{width=»50%»}

Рис. 1

Для однозначного определения размера потребуется с помощью вызова **GCE_AddPoint**() создать вспомогательную точку, из которой начинается выносная линия размера со стороны окружности. Дескриптор этой точки будет использован в качестве вспомогательного при создании линейного размера. В примере, приведенном на рис. 2, требуется только одна точка – на стороне окружности. В других случаях, таких как расстояние окружность-окружность, потребуется две вспомогательные точки, поскольку для пары окружностей возможны четырех варианта определения размера. На рис. 3 размеры на 30, 50, 60 и 90 показывают альтернативные варианты измерения расстояний между двумя окружностями и соответственно комбинации выбора пары вспомогательных точек на окружностях.



{width=»50%»}

Рис. 2

7.2.2 Управляющие и вариационные размеры

Размеры отличаются от логических ограничений (параллельность, перпендикулярность и т.д.) тем, что они связаны с числовой величиной (скаляром) – *значением размера*. Значение размера можно задать двумя способами:

- **Управляющий размер**. Значение размера задается независимым числовым параметром, который дается пользователем и может меняться только из вне через API решателя (вызов **GCE_ChangeDrivingDimension**). В этом случае размер не зависит от геометрии, а наоборот, геометрия подчиняется размеру также как геометрическому ограничению. Размеры с таким способом задания требуются для того, чтобы управлять геометрией с помощью размерных параметров, и называются соответственно **управляющими**.
- **Вариационный размер**. Значение размера задается числовой переменной, которая является таким же предметом вычислений решателя, как и геометрия. Фактически текущее значение будет ассоциировано с переменной, которая специально создается с помощью вызова **GCE_AddVariable**, и может быть повторно применена в других размерах или уравнениях. Значение переменой размера может меняться под влиянием геометрии. Такой способ задания применяется в тех случаях, когда требуется не установить размер, а измерить его. Размеры, связанные с варьируемой переменной, называются **вариационными**.

Иногда требуется создать *вариационный* размер, действующий, как *управляющий*, особенно когда необходимо управлять одновременно двумя и более размерами, завязанными на один числовой параметр. В этом случае можно создать один или несколько вариационных размеров с одной переменной, которую можно сделать *управляющим параметром* с помощью вызова:

```
constraint_item GCE_FixVariable(GCE_system gSys, var_item var),
```

где **var** – дескриптор переменной размера.

Функция вернет ограничение *управляющий параметр*, фиксирующее переменную **var** с её текущим значением. После вызова **GCE_FixVariable** переменная фактически становится независимым параметром системы ограничений и может меняться только извне, с помощью вызова **GCE_ChangeDrivingDimension**, примененным к дескриптору полученного ограничения.

Варьирование управляющими размерами и параметрами осуществляется с помощью метода

```
GCE_result GCE_ChangeDrivingDimension(GCE_system gSys, constraint_item dItem
                                      , double dVal).
```

Входными параметрами метода являются:

- **gSys** – система ограничений параметрического эскиза;
- **dItem** – дескриптор управляющего размера или управляющего параметра;
- **dVal** – требуемое значение управляющего размера или управляющего параметра.

Функция вернет код GCE_RESULT_Ok, если операция выполнена успешно. Если функция вернет код ошибки (отличный от GCE_RESULT_Ok), это может быть связано с одной из следующих причин:

- недействительные дескрипторы системы или ограничения **gSys** и **dItem**;
- присланное ограничение не является управляющим размером или управляющим параметром, например, если ограничение – вариационный размер.

Следует учитывать, что данная функция не выполняет вычислений, а только подготавливает изменение управляющего размера или параметра. Чтобы изменение вступило в силу, необходимо вызвать вычислительную функцию **GCE_Evaluate**. Если требуется изменить одновременно два и более управляющих размеров, следует сначала сделать серию подготовительных вызовов **GCE_ChangeDrivingDimension** на каждый управляющий параметр или размер, а затем одним вызовом **GCE_Evaluate** вычислить новое положение геометрии, подчиненное новому состоянию управляющих размеров или параметров.

7.2.3 Нулевые и знакопеременные размеры

Все типы линейных размеров (distance dimension) имеют непрерывную область, включающую нуль, отрицательные и положительные значения. На рис. 1 изображен эскиз с размером, позиционирующим геометрию, как в положительной, так и в отрицательной области значений. Слева эскиз имеет отрицательное значения размера, справа изображен тот же эскиз, которому поменяли знак размера с отрицательного на положительный.

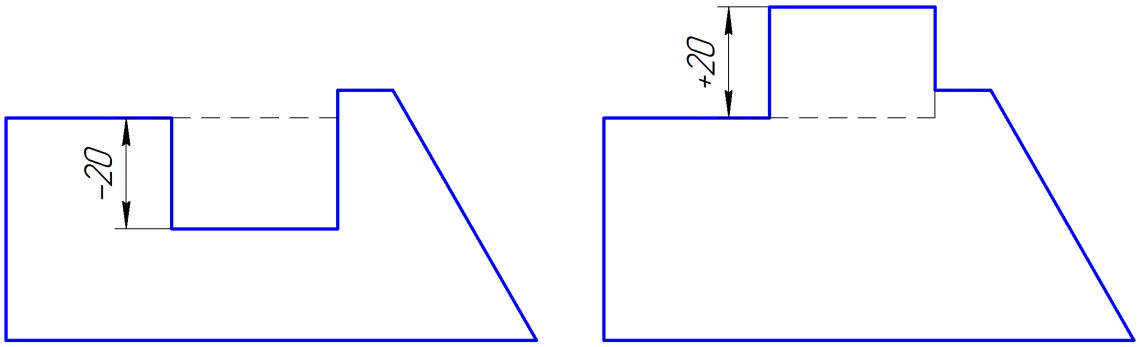


Рис. 1

Нулевые значения допустимо принимать всем видам линейных и угловых размеров, штриховой линией изображена геометрия в нулевом положении, когда значение размера равно нулю.

Знакопеременность свойственна прежде всего для размеров, заданных для ориентированного объекта, такого, как прямая, имеющая вектор нормали. Прямая делит пространство на две полуплоскости, поэтому, к примеру, линейный размер между прямой и точкой будет иметь разные знаки в зависимости от того, на какой стороне от прямой лежит точка. Точка, лежащая на прямой, будет соответствовать нулевому размеру.

Заметим, что расстояние от точки до точки по природе является неотрицательной величиной; несмотря на это, линейный размер, заданный для двух точек, может быть отрицательным или нулевым. Это вполне имеет практический смысл, когда мы имеем дело с параметрическим эскизом. Например, знакопеременный размер, проиллюстрированный на левом и правом эскизе из рис. 27.3.1, можно задать, как ограничение расстояние от точки до прямой, так и расстояние от точки до точки. В обоих случаях его поведение будет одинаковым.

7.2.4 Линейный размер (расстояние)

Линейный размер задается для пары геометрических объектов. В таблице 1 приведены пары типов геометрии, которые поддерживает C3D Solver применительно к линейному размеру.

Таблица 1 - Пары геометрических объектов, поддерживаемых для линейного размера

	Точка	Прямая	Окружность	Эллипс	Сплайн	Парам. кривая
Точка	Y	Y	Y			
Прямая	Y	Y	Y			
Окружность	Y	Y	Y			
Эллипс						
Сплайн						
Парам.кривая						

Любой линейный размер, вне зависимости от типов кривых, для которой он применен, измеряет расстояние между точками, лежащими на кривых. Обычно это точки с минимальным расстоянием между объектами. Однако можно выбрать другие пары точек для измерения размера. На рис 27.1.2 из параграфа *Вспомогательные точки линейного размера* показан пример четырех вариантов линейного размера для одной и той же пары окружностей.

Линейный размер для пары геометрических объектов задается методом

```
constraint_item GCE_AddDistance(GCE_system gSys, geom_item g[2]
                                , const GCE_ldim_pars & dPars);
```

Возвращаемое значение функции – дескриптор линейного размера, заданного для объектов **g**[0] и **g**[1], принадлежащих системе ограничений **gSys**.

Структура параметров **dPars** определяет следующие настройки линейного размера:

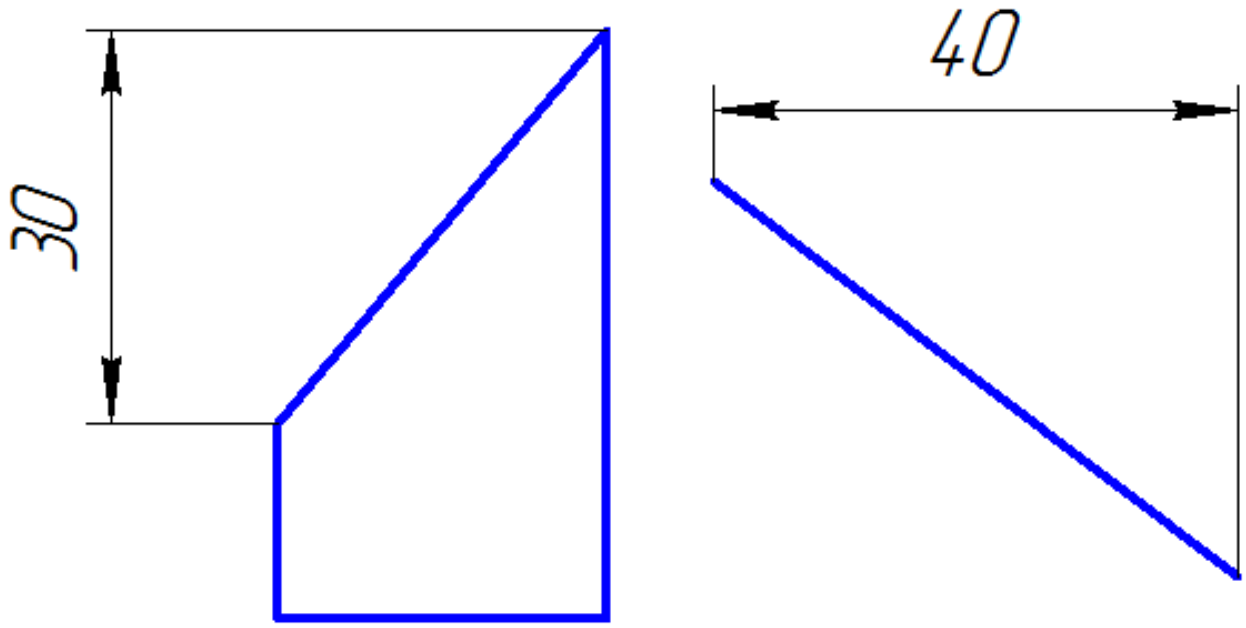
- **dPars.dPars** – задает значение размера, который может быть, как независимым параметром, так и переменной;
- **dPars.hp**[2] – пара вспомогательных точек, задающих вариант измерения, когда это требуется (подробнее см. параграф *Вспомогательные точки линейного размера*).

Параметр **dPars.dPars** фактически определяет источник значения размера. Если **dPars.dPars.var =** GCE_NULL_V, то значение размера определяется независимым числовым параметром **dPars.dPars.dimValue**, что делает размер управляющим. Если в **dPars.dPars.var** задан дескриптор переменной, то размер является вариационным. Подробнее об управляющих и вариационных размерах обсуждалось в параграфе *Управляющие и вариационные размеры*.

Пара дескрипторов вспомогательных точек **dPars.hp**[2] не является обязательной, если вариант измерения размера однозначен, как например, расстояние от точки до прямой, либо оставляется на выбор решателя, который подберет вариант измерения наиболее близкий к текущим расположению объектов и значению размера. В этом случае по умолчанию **dPars.hp**[2] равен паре нулевых дескрипторов {GCE_NULL, GCE_NULL}. Если одна или обе вспомогательные точки **dPars.hp**[0] и **dPars.hp**[1] заданы, то координаты этих точек будут указывать линейному размеру вариант измерения. Подробнее о выборе варианта измерения обсуждается в параграфе *Вспомогательные точки линейного размера*.

7.2.5 Направленный линейный размер

Обычный линейный размер измеряет расстояние между точками, лежащими на геометрических объектах. Направленный линейный размер также задается для пары точек, однако расстояние измеряется между проекциями точек вдоль прямой с конкретным направлением. В основном данный тип размера используется для создания вертикальных и горизонтальных размеров, которые задают расстояние между точками в проекции на оси X или Y. На рис. 1 показаны примеры вертикального и горизонтального размеров для пары точек. В общем случае направление размера может быть произвольным.



{width=»50%»}

Рис. 1

Также направленный размер предусматривает определенное поведение при геометрической трансформации эскиза методом *GCE_Transform*(). Направленный размер повернется, если матрица трансформации будет содержать ротационный компонент.

Направленный размер создается с помощью метода

```
constraint_item GCE_AddDirectedDistance(GCE_system gSys, geom_item p[2]
                                      , const GCE_ldim_pars & dPars)
```

Возвращаемое значение функции – дескриптор наклонного размера.

Входными параметрами метода являются:

- **gSys** – система ограничений параметрического эскиза;
- **p**[2] – дескрипторы двух точек;
- **dPars** – структура данных, в которой кроме обычных настроек, свойственных линейным размерам (см. аналогичные настройки для метода **GCE_AddDistance**()), дополнительно задается угловое направление размера **dPars.dirAngle**. Например, если требуется задать *горизонтальный* размер, то **dPars.dirAngle** = 0; если требуется задать *вертикальный* размер, то **dPars.dirAngle** = $\pi/2$. В общем случае значение **dPars.dirAngle** берется от 0 до 2π.

Замечание. В текущей версии C3D Solver направленный размер можно задать только для точек.

7.2.6 Расстояние от точки до отрезка

Расстояние от точки до отрезка – это линейные размер, вовлекающий тройку точек, из которых две точки задают прямолинейный отрезок, от которого измеряется расстояние до третьей точки. Данный тип размера создается с помощью метода

```
constraint_item GCE_AddDistancePLs(GCE_system gSys, geom_item p[3]
                                                  , const GCE_dim_pars & dPars).
```

Особенность данного ограничения в том, что он требует ненулевую длину отрезка. Это означает, что линейный размер перестанет работать, если две точки, задающие отрезок, совпадут. Размер может быть применен, когда требуется задавать линейный размер только по точкам, без наличия линейного объекта, однако если этого не требуется, то рекомендуется использовать функцию **GCE_AddDistance**(), которой будет достаточно для создания практически всех вариантов линейных размеров.

7.2.7 Угловые размеры

Угловой размер задается для двух геометрических объектов, имеющих направляющие вектора. Значение углового размера задается в радианах с областью значений от 0 до 2π и определяется углом, измеряемым против часовой стрелки, между направляющими векторами первого и второго объектов. Угловой размер с таким способом измерения показан на рис.1 на примере двух пересекающихся прямых с данными направлениями. Справа на рис.2 показана пара прямых, имеющих тот же угол между векторами, однако размер охватывает обратный сектор угла. Значение такого размера определяется по формуле $D = 2\pi - \alpha$, где α – измерительное значение угла между векторами, а D – значение размера. Данный размер также называется сопряженным (conjugate), по другому его можно получить, просто поменяв местами аргументы углового размера.

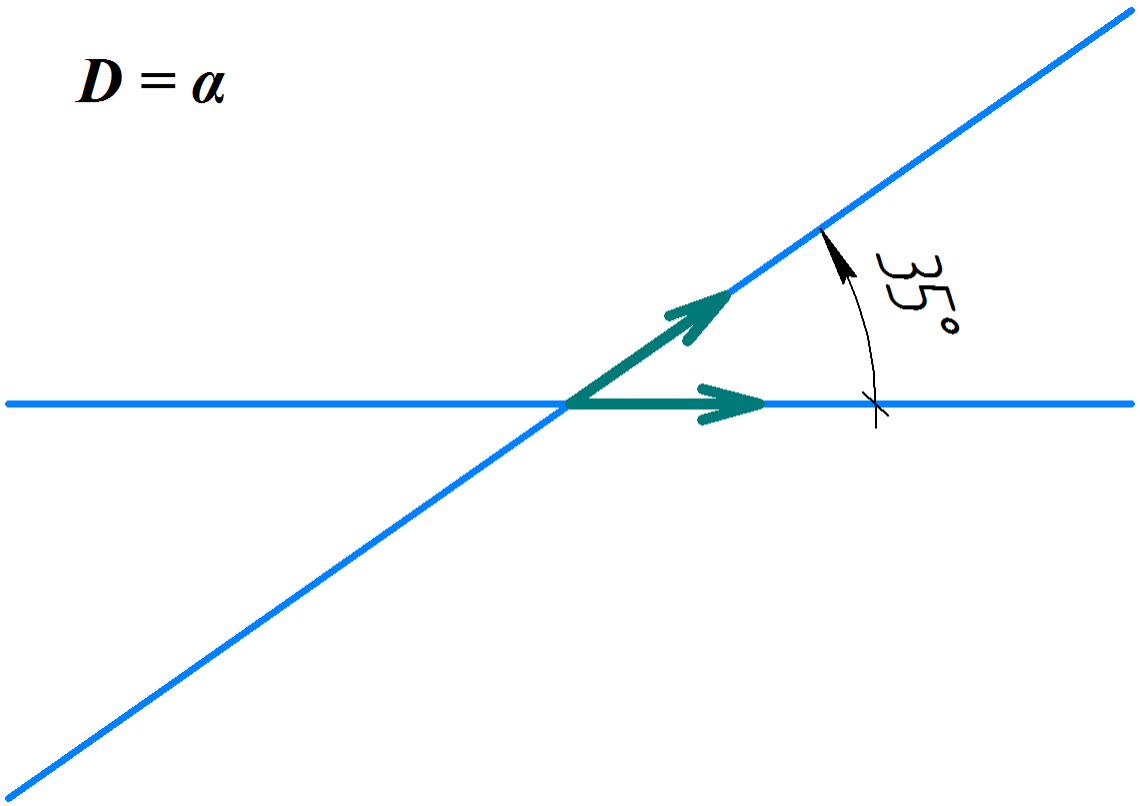


Рис. 1

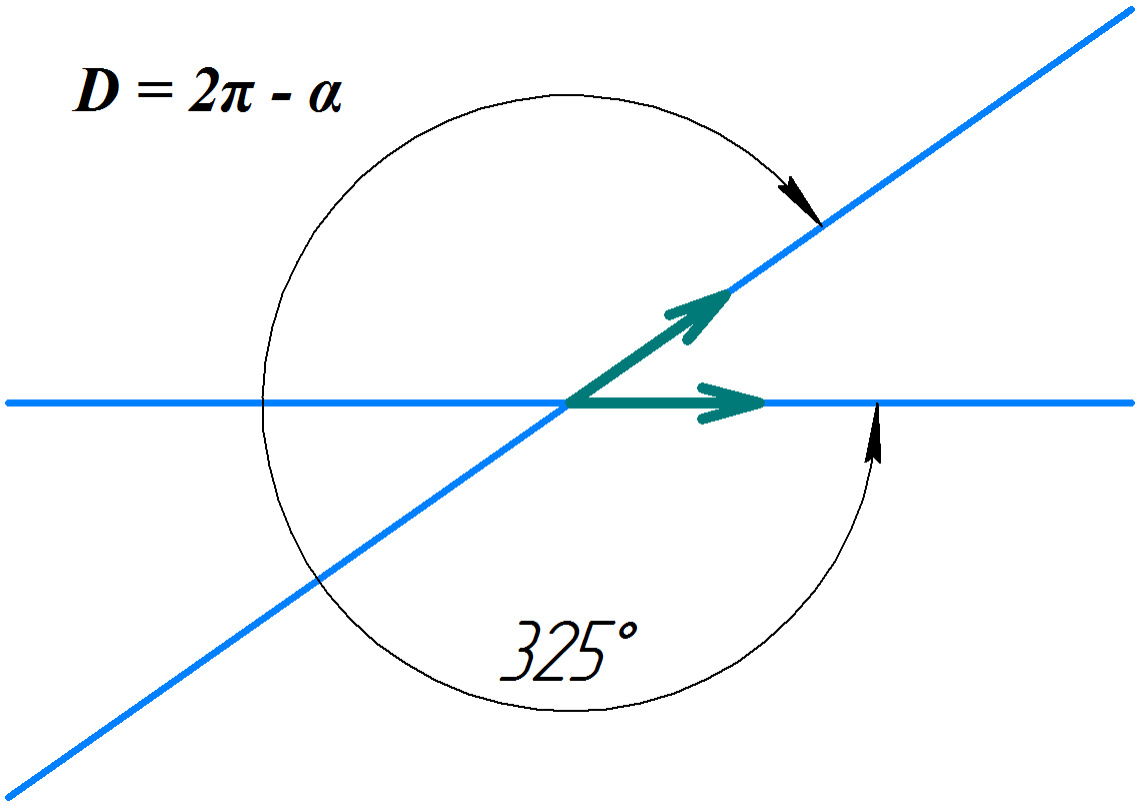


Рис. 2

Всего для одной и той же пары направлений, включая два рассмотренных выше, возможны четыре варианта задания углового размера. На рис.3 изображен размер, значение которого определяется по формуле смежного (adjacent) угла $D = \pi - \alpha$. На рис.4 справа показан альтернативный вариант смежного угла, который определяется суммой $D = \pi + \alpha$.

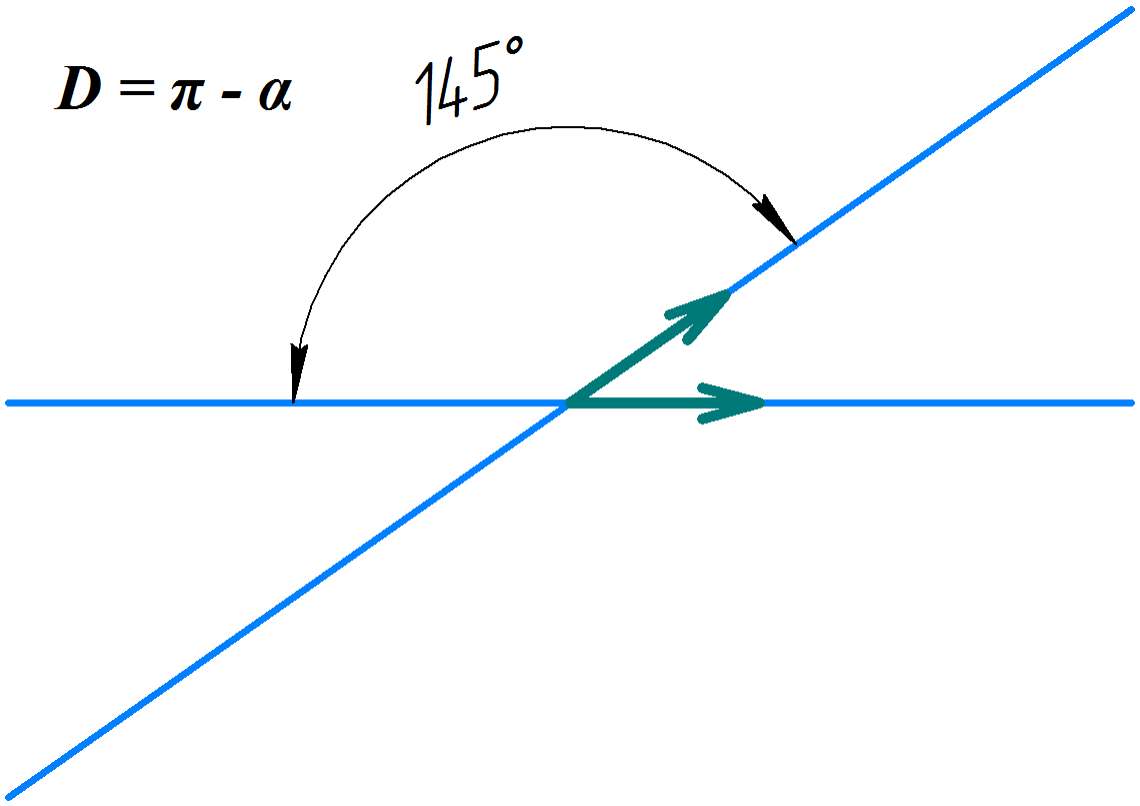


Рис. 3

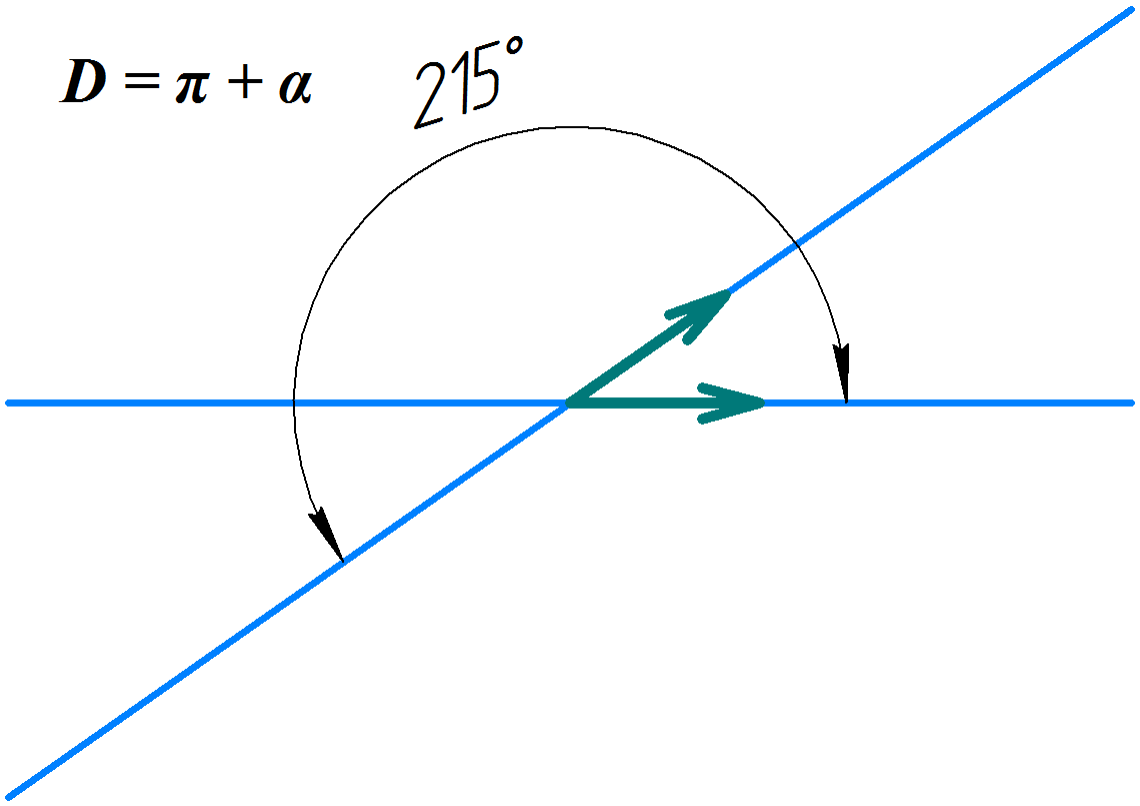


Рис. 4

Еще одна полезная закономерность, связанная с четырьмя способами задания углового размера, состоит в том, что размер со значением $D = 2\pi - \alpha$ получается из размера $D = \alpha$, если просто поменять местами первый и второй объекты размерного ограничения. Аналогично, переменой местами аргументов, получается 4-й вариант $D = \pi + \alpha$ из 3-го варианта $D = \pi - \alpha$.

Угловой размер для пары направленных объектов, таких, как отрезок, прямая или эллипс, задается методом

```
constraint_item GCE_AddAngle(GCE_system gSys, geom_item l1
                                            , geom_item l2
                                            , const GCE_adim_pars & dPars).
```

Функция создаст в системе ограничений **gSys** новый угловой размер, заданный для геометрических объектов **l1** и **l2**. Значение размера и способ его задания определяется структурой **dPars**.

Возвращаемое значение – дескриптор нового углового размера.

Структура **dPars** определяет значение углового размера, заданного через структуру **dPars.dPars** (**GCE_dim_par**), в которой значение размера хранится в переменной **dPars.dPars.var** либо, в случае **dPars.dPars.var = GCE_NULL_V**, значение размера задано независимым числовым параметром **dPars.dPars.dimValue**, что делает размер управляющим. Подробнее об управляющих и варьируемых разделах рассмотрено в параграфе *Управляющие и вариационные размеры*.

Если параметр **dPars.adjacent = false**, то значение размера будет определяться формулой $D = f \cdot \alpha$. Множитель ***f*** в данной формуле задается параметром **dPars.factor** и поз-

воляет задавать угловой размер в других единицах. Например, если **dPars.factor** = $180/\pi$, то приложение получает возможность задавать угловой размер в градусах. Что бы задать угловой размер со значением $D = f \cdot (2\pi - \alpha)$, необходимо поменять местами аргументы **l1** и **l2**.

Для определения угловых размеров со смежным вариантом угла, показанных на рис. 27.7.3 и рис. 27.7.4, необходимо задать флаг **dPars.adjacent = true**, в этом случае значение размера будет определять по формулам $D = f \cdot (\pi - \alpha)$ или $D = f \cdot (\pi + \alpha)$. Вторая формула получается, если поменять местами аргументы **l1** и **l2**. Множитель ***f*** также задается параметром **dPars.factor**.

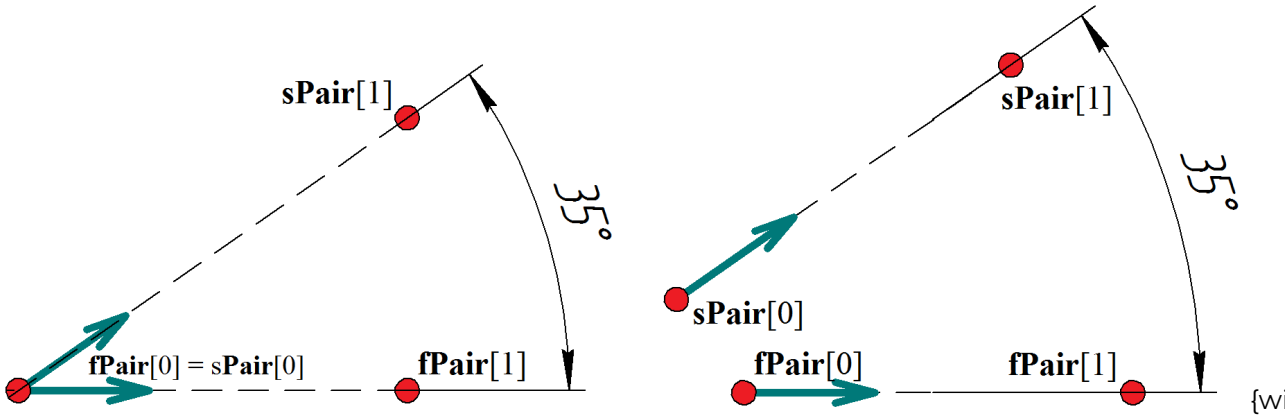
7.2.8 Угловой размер по трем или четырем точкам

C3D Solver также предоставляет еще один способ определения углового размера, для которого направляющие вектора определяются парами точек. Этот же способ позволяет задавать угловой размер по трем точкам, которые определяют угол измерительного треугольника.

Угловой размер по тройке или четверке точек создается с помощью метода

```
constraint_item GCE_AddAngle4P(GCE_system gSys, geom_item fPair[2]
                                              , geom_item sPair[2]
                                              , const GCE_adim_pars & dPars ).
```

Пары точек, заданные дескрипторами **fPair** и **sPair**, определяют измерительные вектора углового размера. Вектор, построенный от точки **fPair**[0] до точки **fPair**[1], определяет первую сторону угла, вектор от **sPair**[0] до **sPair**[1], определяет вторую сторону угла. На рис.1 показаны измерительные углы размера для трех точек (слева) и четырех точек (справа). Чтобы задать размер по трем точкам, необходимо точку вершины угла задать дважды, в первой и во второй паре, например, **fPair**[0]=**sPair**[0].



{width=»50%»}

Рис. 1

Остальные параметры функции **GCE_AddAngle4P** задаются аналогично **GCE_AddAngle**.

7.2.9 Радиальные и диаметральные размеры

Данные типы размеров действуют для окружностей или дуг и задают соответственно радиус окружности или диаметр, равный удвоенному значению радиуса. Радиальные и диаметральные размеры, как и линейные так и угловые, могут быть ассоциированы с независимым числовым параметром или скалярной переменной, т.е. быть *управляющими* или *вариационными*, что настраивается структурой `GCE_dim_pars`.

Радиальный размер создается методом:

```
constraint_item GCE_AddRadiusDimension(GCE_system gSys, geom_item cir
                                       , GCE_dim_pars dPar).
```

Диаметральный размер создается методом:

```
constraint_item GCE_AddDiameter(GCE_system gSys, geom_item cir, GCE_dim_pars dPar).
```

Оба метода имеют одинаковые исходные данные:

- **gSys** – система геометрических ограничений, где регистрируется новый размер;
- **cir** – дескриптор окружности или дуги окружности;
- **dPars** – структура, которая определяет значение размера, заданного в единицах длины.

Значение размера хранится в переменной **dPars.var** либо, в случае **dPars.var = GCE_NULL_V**, значение размера задано независимым числовым параметром **dPars.dimValue**, что делает размер управляющим. Подробнее управляющие и варьируемые размеры рассмотрены в параграфе *Управляющие и вариационные размеры*.

7.3 ДВУМЕРНЫЕ ЛОГИЧЕСКИЕ ОГРАНИЧЕНИЯ

Логические ограничения, в отличие от размеров, не имеют числовых параметров и задаются только для геометрических объектов. C3D Solver позволяет задавать для геометрических объектов такие логические ограничения, как горизонтальность/вертикальность, совпадение, параллельность/перпендикулярность, касание, равенство радиусов и т.д.

7.3.1 Совпадение точки и другого объекта

В случае, когда два геометрических объекта имеют один и тот же тип, ограничение совпадение подразумевает равенство двух геометрических объектов. Однако, в настоящей версии C3D Solver данное ограничение действительно только, когда один из аргументов является точкой.

Если один из аргументов совпадения точка, а другой – кривая, то ограничение определит точку, лежащую на кривой. Ограничение совпадение задается вызовом метода

```
constraint_item GCE_AddCoincidence(GCE_system gSys, geom_item g[2]).
```

Функция добавит в систему **gSys** ограничение совпадения двух геометрических объектов **g**[0] и **g**[1] и вернет его дескриптор.

7.3.2 Выравнивание точек

Данный тип ограничения чаще всего используется для выравнивания пары точек по горизонтали или по вертикали. В общем случае данное ограничение будет размещать точки на воображаемой прямой с заданным направлением. Заметим, что оно не гарантирует сохранение порядка точек выравнивания вдоль заданного направления.

Выравнивание точек с константным направлением выравнивания задается методом

```
constraint_item GCE_AddAlignPoints(GCE_system gSys, geom_item p[2], double ang);
```

Метод принимает систему ограничений **gSys**, пару точек **p[2]** и константный угол выравнивания **ang**, а возвращает дескриптор нового ограничения.

Параметры пары **p[0]** и **p[1]** задают дескрипторы выравниваемых точек, а параметр **ang** задается в радианах и определяет направление, вдоль которого будут выравниваться точки. Например, угловое направление $ang = 0$ будет соответствовать выравниванию точек по горизонтали, значение $ang = \pi/2$ будет соответствовать выравниванию по вертикали.

Выравнивание точек с варьируемым направлением выравнивания задается методом

```
constraint_item GCE_AddAlignPoints(GCE_system gSys, geom_item p[2], geom_item dir);
```

Аналогично методу с константным направлением выравнивания, эта реализация принимает систему ограничений **gSys** и пару точек **p[2]**.

Параметр **dir** – любой зарегистрированный в решателе объект, для которого корректен вызов **GCE_GetVectorValue**(gSys, dir, GCE_DIRECTION).

Выравнивание произойдёт вдоль направляющего вектора (**GCE_DIRECTION**) линейного объекта или главной полуоси эллипса.

Это ограничение можно использовать вместе с **GCE_AddLinearDimension** для задания линейного размера вместо метода **GCE_AddDistance**. Данный способ предпочтительнее, поскольку дает возможность более точно настроить привязки вспомогательных точек в соответствии с ожиданиями пользователя.

7.3.3 Параллельность/перпендикулярность

Ограничения параллельность и перпендикулярность задаются для пары прямых или отрезков с помощью методов:

```
constraint_item GCE_AddParallel(GCE_system gSys, geom_item g[2]),
```

```
constraint_item GGCE_AddPerpendicular(GCE_system gSys, geom_item g[2]),
```

которые принимают следующие исходные данные:

- **gSys** – система геометрических ограничений,
- **g**[2] – пара прямолинейных кривых.

Возвращаемый результат – дескриптор геометрического ограничения, зарегистрированного в системе **gSys**.

7.3.4 Коллинеарность

Коллинеарность подразумевает, что геометрические объекты, участвующие в данном ограничении, лежат на одной прямой. Коллинеарность можно задать как для двух геометрических объектов, так и для трех точек. Коллинеарность для пары геометрических объектов создается вызовом метода

```
constraint_item GCE_AddColinear(GCE_system gSys, geom_item g[2]),
```

который требует, чтобы один из аргументов ограничения был прямолинейным, а другой являлся либо прямой, либо точкой. Геометрические аргументы задаются парой **g**[2]. Функция вернет дескриптор нового ограничения, зарегистрированного в системе **gSys**.

Коллинеарность для трех точек создается вызовом метода

```
constraint_item GCE_AddColinear3Points(GCE_system gSys, geom_item p[3]).
```

Через параметр **p**[3] передаются дескрипторы трех точек, которые требуется разместить на одной прямой. Функция вернет дескриптор нового ограничения, зарегистрированного в системе **gSys**.

7.3.5 Виды равенств характеристик кривых

C3D Solver поддерживает четыре вида равенств характеристик кривых: равенство длин GCE_AddEqualLength, равенство радиусов GCE_AddEqualRadius, равенство смещений эквидистантных кривых GCE_AddEqualOffset и равенство кривизны двух кривых в заданных точках GCE_AddEqualCurvature. Последовательно рассмотрим каждый из них.

Равенство длин двух кривых задается методом

```
constraint_item GCE_AddEqualLength(GCE_system gSys, geom_item curve1,
                                                    geom_item curve2);
```

Метод принимает систему ограничений **gSys** и пару кривых – **curve1** и **curve2**, а возвращает дескриптор нового ограничения.

Равенство длин двух кривых можно накладывать на кривые, которые имеют концы (т.е. у них можно запросить GCE_FIRST_END и GCE_SECOND_END), либо являются замкнутыми. Поддерживается для любых комбинаций кривых, для которых поддерживается ограничение *GCE_AddLength*.

Равенство радиусов для двух кривых задается методом

```
constraint_item GCE_AddEqualRadius(GCE_system gSys, geom_item c1,
                                                    geom_item c2);
```

Метод принимает систему ограничений **gSys** и пару кривых – **c1** и **c2**, а возвращает дескриптор нового ограничения.

Равенство радиусов для двух кривых можно накладывать на кривые, у которых можно запросить GCE_RADIUS.

Равенство смещений эквидистантных кривых задаётся методом

```
geom_item GCE_AddEqualOffset(GCE_system gSys, geom_item curve1,
                                              geom_item curve2);
```

Метод принимает систему ограничений **gSys** и пару эквидистантных кривых – **curve1** и **curve2**, а возвращает дескриптор нового ограничения.

Равенство смещений эквидистантных кривых можно накладывать на кривые, у которых можно запросить GCE_OFFSET.

Равенство кривизны двух кривых в заданных точках задается методом

```
constraint_item GCE_AddEqualCurvature(GCE_system gSys, geom_item curves[2],
                                                       var_item tPars[2]);
```

Метод принимает систему ограничений **gSys**, пару кривых **curves** и параметры параметрических кривых (если требуются), в которых должно выполняется равенство кривизны. Возвращает дескриптор нового ограничения.

Если тип кривой GCE_CIRCLE или GCE_BOUNDED_CURVE, базовой кривой которой является окружность, то соответствующее этой кривой значение tPars[i] должно равняться GCE_NULL_V, так как кривизна окружности одинакова во всех её точках. Если тип кривой GCE_LINE или GCE_LINE_SEGMENT, то равенство кривизны можно задать только в паре с GCE_SPLINE.

7.3.6 Унарные ограничения: горизонтальность/вертикальность и варианты фиксации

Такие ограничения, как горизонтальность, вертикальность, фиксация объекта, фиксация длины и фиксация направления действуют для одного геометрического объекта, а значит являются ***унарными***. Для создания одного из этих ограничения необходимо вызвать функцию

```
constraint_item GCE_AddUnaryConstraint(GCE_system gSys, constraint_type cType,␣
↪geom_item geom).
```

Функция добавит в систему **gSys** унарное ограничение для геометрического объекта **geom**. Параметр **cType** задает перечислительный тип унарного ограничения и может принимать одно из значений, приведенных в таблице 1: Таблица 1 - Типы унарных ограничений

Тип ограничения	Описание
GCE_FIX	**Фиксация геометрического объекта** делает объект неподвижным. Решатель не может поменять состояние данного объекта при вычислениях. Фиксированный геометрический объект может быть изменен только приложением с помощью вызовов GCE_SetPointXY и GCE_SetCoordValue.
GCE_HO GCE_VEF	**Горизонтальность и вертикальность** применяются для типов прямая и отрезок прямой. Эти ограничения выравнивают линейный объект с системой координат эскиза: соответственно горизонтальность по оси OX, вертикальность по оси OY. Заметим, что данные ограничения допускают переворот направляющего вектора прямой на 180 градусов.
GCE_ANG	Данное ограничение фиксирует угловое положение прямой или отрезка. Ограничение типа GCE_ANGLE_OX подробнее рассмотрено в параграфе *Ограничение угловое положение.*
GCE_LEN	Данное ограничение фиксирует длину отрезка прямой. Подробнее см. параграф *Фиксация длины.*
GCE_RAD	Данное ограничение подробно рассмотрено в параграфе *Фиксация радиуса.* Применительно к вызову GCE_UnaryConstraint данный тип ограничения действует только для окружности. Для применения к эллипсу следует вызывать GCE_FixRadius.

7.3.7 Длина кривой

Длина кривой задается методом

```
constraint_item GCE_AddLength(GCE_system gSys, geom_item curve,
                                               GCE_dim_pars dPar);
```

Метод принимает систему ограничений **gSys**, кривую **curve** и параметры линейного размера. Функция вернет дескриптор нового ограничения.

Ограничение поддерживается для следующих видов кривых: отрезка, окружности, сплайна, а также ограниченной кривой (GCE_BOUNDED_CURVE), созданной на основе прямой, окружности или сплайна.

Зафиксировать длину кривой можно с помощью метода

```
constraint_item GCE_FixLength(GCE_system gSys, geom_item curve);
```

Функция вернет дескриптор нового ограничения. Параметр **curve** задает дескриптор кривой.

Ограничение поддерживается для тех же кривых, для которых поддерживается ограничение GCE_AddLength.

Фиксация длины является размерным ограничением, поэтому допускается изменение значения длины методом *GCE_ChangeDrivingDimension*.

7.3.8 Фиксация радиуса

Ограничение фиксация радиуса применяется для окружности или эллипса. C3D Solver не может менять радиус окружности или эллипса, если для объекта задано ограничение фиксация радиуса. Приложение может работать с данным ограничением, как с управляющим размером, допускающим варьирование радиуса методом **GCE_ChangeDrivingDimension**(). Применительно к окружности данное ограничение является аналогом управляющего радиального размера, рассмотренного в параграфе *Радиальные и диаметральные размеры*. Фиксация радиуса осуществляется вызовом метода

```
constraint_item GCE_FixRadius(GCE_system gSys, geom_item g
                              , coord_name cName = GCE_RADIUS ).
```

Исходные данные вызова:

- **gSys** – система геометрических ограничений;
- **g** – дескриптор окружности или эллипса;
- **cName** – тип радиуса, принимающий следующие значения:
  - GCE_RADIUS – значение по умолчанию для окружности;
  - GCE_MAJOR_RADIUS или GCE_MINOR_RADIUS, означающие фиксацию радиуса на главной или минорной оси эллипса.

Возвращаемый результат – дескриптор ограничения, фиксирующего радиус окружности или эллипса.

7.3.9 Ограничение углового положения

Ограничение угловое положение применяется для типов прямая и отрезок прямой. Фактически данный тип ограничения является размерным и создает управляющий размер, фиксирующий угловое положение линейного объекта в его начальном состоянии относительно оси OX общей системы координат эскиза. Угловое положение измеряется в диапазоне от 0 до 2π. Чтобы создать фиксацию направления, необходимо вызвать **GCE_AddUnaryConstraint**, задав тип ограничения GCE_ANGLE_OX. В отличие от GCE_HORIZONTAL и GCE_VERTICAL, данный тип ограничения не допускает пе-

реворот направляющего вектора прямой на 180 градусов. Поскольку данное ограничение размерное, угловое положение прямой можно менять с помощью вызова **GCE_ChangeDrivingDimension**.

7.3.10 Касание

Касание – это геометрическое ограничение, позиционирующее пару кривых таким образом, что они буду касаться в одной точке. В таблице 1. приведены поддерживаемые пары кривых касания.

Таблица 1 - Пары кривых, поддерживаемых ограничением касания

	Прямая	Окружность	Эллипс	Сплайн	Параметрическая кривая
Прямая		+	+	+	+
Окружность	+	+	+	+	+
Эллипс	+	+		+	
Сплайн	+	+	+	+	
Параметрическая кривая	+	+			

newline

Выбор варианта касания. На рис.1 изображены два примера касания для пар окружность-окружность и окружность-прямая. Штриховой линией изображены окружности с альтернативным вариантом касания. Рисунок наглядно показывает, что любое касание может иметь два решения. Например, окружности могут касаться, оставаясь снаружи друг друга, либо одна внутри другой; в другом примере, на рис.2 справа показано, что при касании прямой и окружности возможны два варианта их взаимного размещения, «слева» и «справа» от прямой.

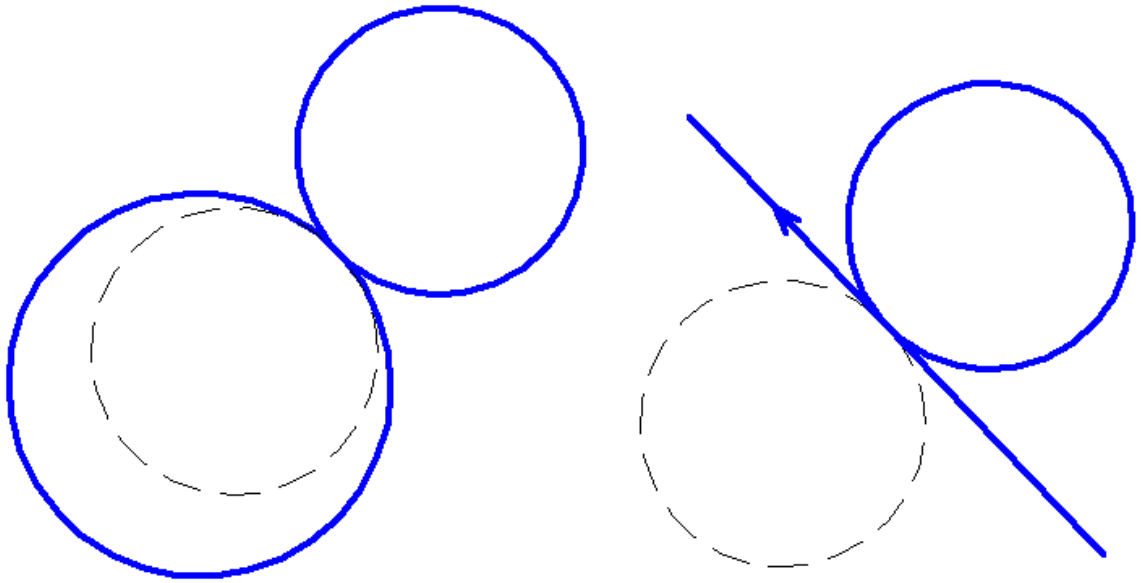


Рис. 1

Геометрический решатель GCE осуществляет выбор одного из двух вариантов касания,

руководствуясь принципом «близости к желаемому решению». Фактически сами объекты касания указывают на выбор одного из двух вариантов взаимного размещения. В момент создания ограничения решатель «запоминает» текущее взаимное размещение объектов и сохраняет его при дальнейших изменениях геометрии эскиза (см. параграф *Взаимное размещение объектов*).

Вспомогательные параметры касания. Рассмотренный выше подход к выбору варианта касания на практике достаточен, если мы имеем дело только с аналитическими кривыми (прямая, окружность, эллипс). Для кривых с произвольной формой, таких как сплайн или параметрическая кривая, кроме выбора «стороны» взаимного размещения, необходимо указать локализацию точки касания вдоль кривых.

На рис.3 показан пример, где уже сделан выбор взаимного размещения прямой и сплайна (сонаправленность кривых в точке касания), однако этот выбор оставляет другие варианты локализации точек соприкосновения. Переменные t1 и t2, взятые на параметрической области сплайна, указывают две альтернативные точки касания сплайна и отрезка, один конец которого закреплен.

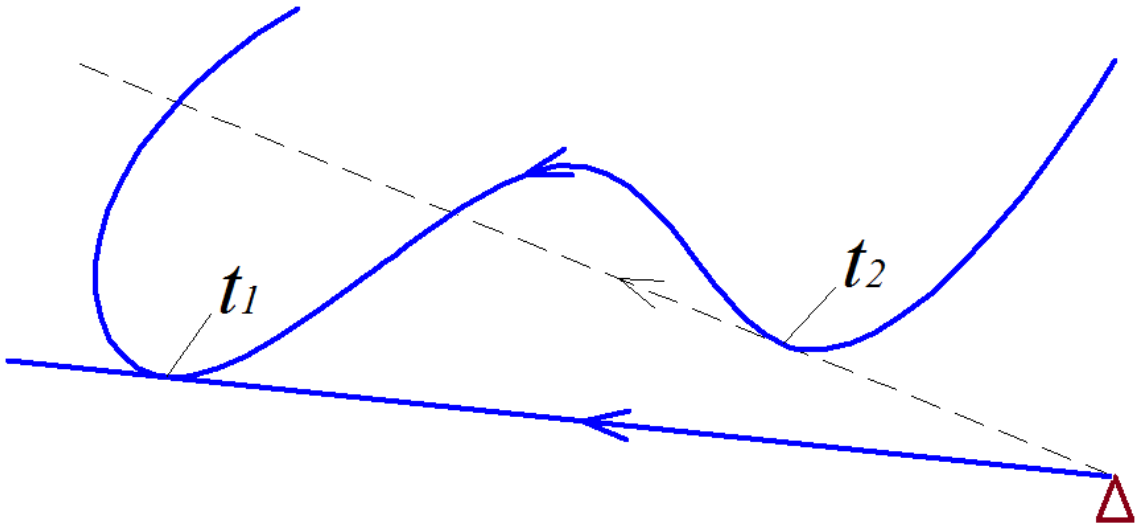


Рис. 2

Для локализации точек касания на сплайне или параметрической кривой заводят вспомогательные переменные, указывающие в параметрической области кривой предполагаемую точку касания. В конечном приложении пользователя это обычно реализуется так, что пользователь выбирает «указателем» объект касания, а координаты указателя рассматриваются, как приблизительная точка предполагаемого касания. Зная координаты указателя, можно оценить приблизительное значение параметра касания (параметр ближайшей точки на кривой).

Создание ограничения касания. Ограничение касания задается вызовом метода

```
constraint_item GCE_AddTangent(GCE_system gSys, geom_item g[2], var_item tPar[2]).
```

Входными параметрами метода являются:

- **gSys** – система ограничений параметрического эскиза;
- **g**[2] – дескрипторы двух кривых;
- **tPar**[2] – дескрипторы переменных, вспомогательные параметры касания для первой и второй кривой.

Функция возвращает дескриптор нового ограничения касания, созданного для данной пары кривых. Значения вспомогательных параметров **tPar**[0], **tPar**[1] принимаются во внимание только, если соответственно первая или вторая кривая является сплайном или параметрической кривой. Для аналитических кривых (прямая, окружность, эллипс) следует передать пустой дескриптор GCE_NULL_V. Например, в **tPar** можно передать массив {GCE_NULL_V,GCE_NULL_V}, если обе кривые аналитические. Если касание задается для сплайна или параметрической кривой, то для ограничения заводится вспомогательная переменная с помощью вызова **GCE_AddVariable**. Начальное значение переменной будет указывать на предполагаемую точку касания в параметрической области кривой. Надо заметить, что любая переменная, как и геометрические объекты, является предметом вычислений, поэтому её значение будет обновляться при всех вычислительных запросах решателя (GCE_Evaluate, см. параграф *Вычисление системы ограничений*), т.е. фактически решатель будет отслеживать актуальное значение параметра касания. Если для сплайна или параметрической кривой **tPar**[idx] = GCE_NULL_V, то локализация точки касания будет выполнена автоматически, по принципу близости к начальному положению объектов.

7.3.11 Множественные и концевые касания

Геометрический решатель GCE поддерживает множественные касания, т.е. имеется возможность задать два и более ограничений касания для одной и той же пары кривых, в которых участвуют сплайны или параметрические кривые. На рис.1 изображены горизонтальная прямая и сплайн, для которых заданы два ограничения касания, имеющих различные точки соприкосновения:

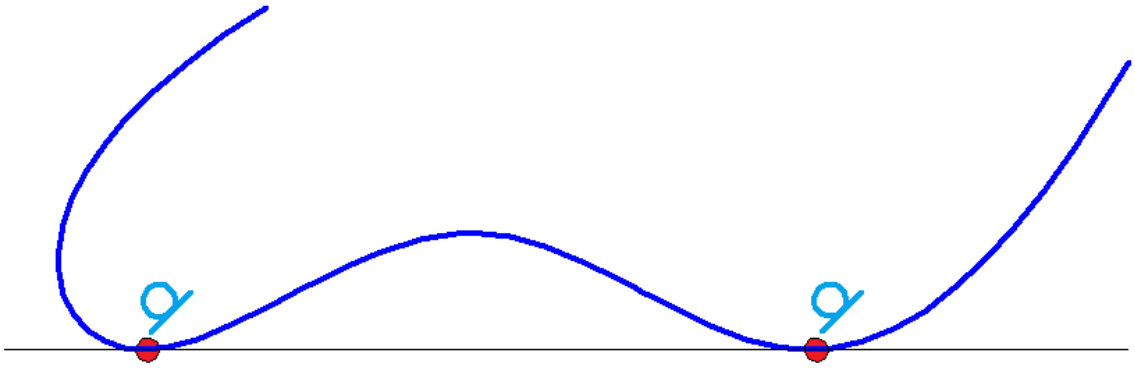


Рис. 1

Заметим, что в случае, когда пара касания состоит только из аналитических кривых (прямая, окружность, эллипс), возможно задать только единственное ограничение касания для одной и той же пары.

Концевое касание с непрерывностью G1 – характерный случай касания с участием сплайна, когда точка соприкосновения совпадает с одним из концов сплайна. В параметрическом черчении концевые касания позволяют создать гладкую стыковку двух кривых, с непрерывностью типа **G1**. На рис.2 приведены случаи гладкой стыковки сплайна с другими кривыми:

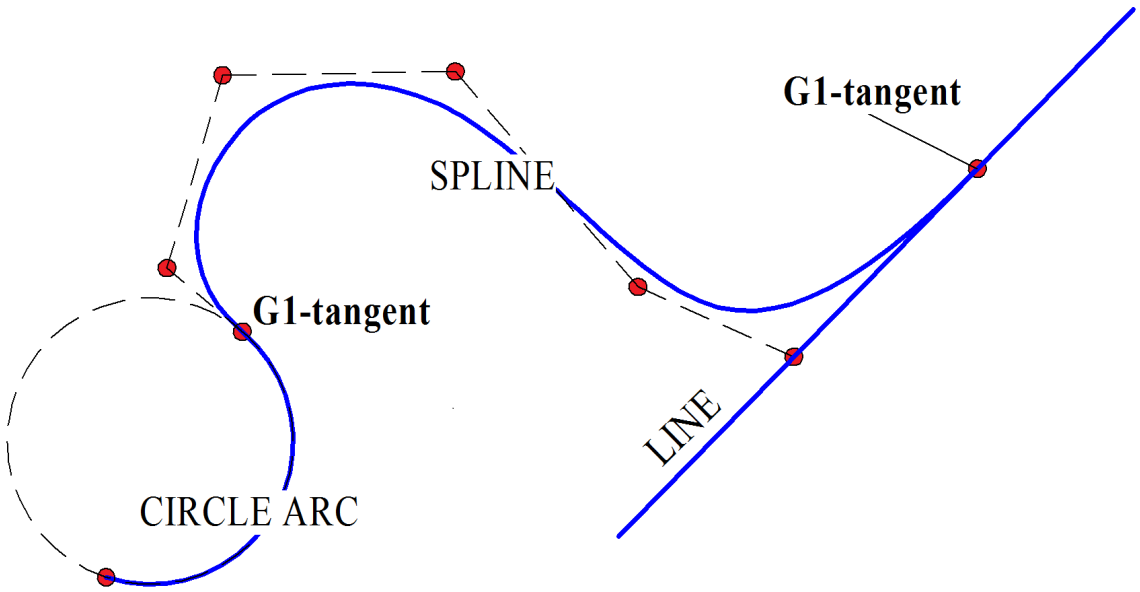


Рис. 2

Метод **GCE_AddTangent**() воспринимает касание на конце незамкнутого сплайна, как особый случай, и автоматически добавляет к касанию условие принадлежности конца сплайна к другой кривой касания. Таким образом, если в общем случае, точка касания имеет свободу перемещения вдоль сплайна, то в случае концевого касания, точка касания прикрепляется к одному из концов сплайна.

Концевое касание можно создать одним из следующих способов:

- Передать в метод **GCE_AddTangent**() вспомогательный параметр сплайна **tPar**[splineIdx], равный параметру начала или конца сплайна.
- Вызвать метод **GCE_AddTangent** без вспомогательного параметра касания, т.е **tPar**[splineIdx]=GCE_NULL_V, но концевая точка касания сплайна должна лежать на другой кривой касания. Это можно обеспечить в момент создания сплайна или предварительно выполнить совпадение конца сплайна с другой кривой с помощью вызова **GCE_AddIncidence** и последующего вызова **GCE_Evaluate**.

7.3.12 Зеркальная симметрия

Зеркальная симметрия задается для тройки геометрических объектов. Первый и второй объекты являются зеркальным отражением друг друга, относительно третьего объекта – прямой, определяющей ось симметрии.

Ограничение симметрии создается вызовом метода

```
constraint_item GCE_AddSymmetry(GCE_system gSys, geom_item g[2], geom_item lObj).
```

7.3.13 Биссектриса

Ограничение биссектриса применяется для тройки линейных объектов, из которых первые два делят плоскость на четыре сектора, а третий объект – ось, делит выбранный сектор пополам. Биссектриса задается с помощью метода

```
constraint_item GCE_AddAngleBisector(GCE_system gSys, geom_item l1
                                   , geom_item l2
                                   , geom_item bl
                                   , GCE_bisec_variant variant)
```

Исходные данные метода:

gSys – система геометрических ограничений;

l1, l2 – пара прямолинейных объектов, делящих плоскость на секторы;

bl – биссектриса, прямая или отрезок прямой, делящей сектор между прямыми **l1** и **l2** пополам;

variant – выбор из трех вариантов, определяющий сектор биссектрисы:

- GCE_BISEC_CLOSEST выбирает один из секторов биссектрисы по относительному размещению объектов в начальном состоянии (см. параграф *Начальное приближение*);
- GCE_BISEC_PLUS – биссектриса делит сектор, на который указывает векторная сумма направлений **l1** и **l2**;
- GCE_BISEC_MINUS – биссектриса делит сектор, на который указывает векторная разница направлений **l1** и **l2**.

Функция возвращает дескриптор ограничения, которое объявляет, что объект **bl** является биссектрисой для сектора, образованного прямыми **l1** и **l2**.

Также следует заметить, что данное ограничение не утрачивает смысл, если прямые **l1** и **l2** не пересекаются. В этом случае биссектриса делит пополам часть плоскости, заключенной между прямыми **l1** и **l2**, располагаясь от них на равных расстояниях.

7.3.14 Средняя точка

Ограничение *средняя точка* задается для тройки точек, которые связаны тем, что третья точка располагается на середине отрезка, образованного первой и второй точками. Данное ограничение создается вызовом метода

```
constraint_item GCE_AddMiddlePoint(GCE_system gcSys, geom_item pnt[3]).
```

7.3.15 Взаимное размещение объектов

В параграфе *Касание* приведен пример касания окружности и прямой, которые могут иметь два варианта взаимного размещения, см. рис. 28.10.1. C3D Solver запоминает взаимную ориентацию в момент создания ограничения и стремиться сохранить её при последующих изменениях. Аналогичный выбор может стоять для линейных и угловых размеров. Например, линейный размер для точки и прямой, предусматривает, что при одном и том же знаке размера точка будет всегда оставаться с одно и той же стороны от прямой (относительно нормали).

Сохранение взаимного размещения объектов поддерживается для таких ограничений, как *касание*, *линейный* и *угловой* размеры. Остальные ограничения, такие как *параллельность*, *перпендикулярность*, *коллинеарность* и прочие допускают смену взаимного размещения. Например, перпендикулярность двух прямых может сменить взаимное размещение с левой пары векторов на правую, так что угол между их направлениями может быть как 90, так и 270 градусов.

7.4 ВЫЧИСЛЕНИЕ И ДИАГНОСТИКА ДВУМЕРНЫХ ОГРАНИЧЕНИЙ

7.4.1 Вычисление системы ограничений

Основная задача C3D Solver, удовлетворение системы ограничений эскиза, выполняется с помощью функции API

```
GCE_result GCE_Evaluate(GCE_system gSys)
```

которая возвращает код результат вычислений системы ограничений **gSys**.

Функция не будет затрачивать время на вычисления, если на момент её вызова все ограничения удовлетворены.

Возвращаемые значения:

- **GCE_RESULT_Ok** означает успешное решение системы ограничений, решатель присвоил всем геометрическим объектам новое состояние, удовлетворяющее все заданные ограничения системы.
- **GCE_RESULT_Overconstrained** означает, что при попытке вычисления обнаружены ограничения противоречащие друг другу, т.е. в системе ограничений присутствует подмножество, в котором ограничения не могут быть удовлетворены одновременно. Состояние геометрии не меняется.
- **GCE_RESULT_Not_Satisfied** означает, что функция не нашла решения, удовлетворяющего всем заданным ограничениям. Состояние геометрии не меняется. Данный диагностический код возвращается при следующих ситуациях:
  - размер вышел за пределы области значений, допускающих существование решения;
  - положения фиксированных или замороженных точек вышли за область, допускающую существования решения;
  - в редких случаях – плохое начальное приближение. Удовлетворение ограничений требует значительные изменение параметрических координат;
  - сильные взаимные влияния. Например заданы такие условия, что изменение угла на один градус приводит к смещению объекта на значительные расстояние;
  - в редких случаях – численная неустойчивость, обусловленная большим порядком разницы между объектами одного и того же чертежа, например размеры наименьшего и наибольшего объектов эскиза различаются более чем на 7 порядков (Например, один эллипс имеет размер 1 мм, а другой – 10 км);
  - в общем случае – иные ситуации, когда решение не существует или не может быть вычислено.
- **GCE_RESULT_InvalidGeometry**. Найдено решение, которое приводит к вырождению геометрических объектов. Например, отрезок или дуга становятся нулевой длины, а окружность или эллипс – нулевого радиуса. Состояние геометрии при этом не изменяется. Такая ситуация возникает, если функции обратного вызова **GCE_allow_zero_radius**/**GCE_allow_zero_length** запрещают геометрическому объекту (окружности, отрезку или ограниченной кривой) принимать нулевой радиус или длину. Также это может происходить в общем случае, если структура с функциями обратного вызова `GCE_callback_table` используется с настройками по умолчанию (без переопределения по вызову GCE_Register).

7.4.2 Изменение или получение состояния геометрии

Во время работы с геометрическими ограничениями приложение может осуществлять запрос состояния геометрии, вычисленное решателем, а решатель, в свою очередь, может передавать состояние координат объектов приложению.

Запрос состояния геометрии выполняют функции:

```
GCE_point GCE_GetPointXY(GCE_system gSys, geom_item g
                                        , point_type pName = GCE_PROPER_POINT)
```

```
double GCE_GetCoordValue(GCE_system gSys, geom_item g, coord_name cName)
```

```
double GCE_GetVarValue(GCE_system gSys, var_item var)
```

Для передачи состояния координат объектов из приложения в решатель используются функции:

```
bool GCE_SetPointXY(GCE_system gSys, geom_item g, point_type pName, GCE_point xyVal)
```

```
bool GCE_SetCoordValue(GCE_system gSys, geom_item g, coord_name cName, double crdVal)
```

```
bool GCE_SetVarValue(GCE_system gSys, var_item var, double val)
```

7.4.3 Начальное приближение

Создание параметрического чертежа – это пошаговое внесение изменений, добавление или удаление объектов и ограничений, варьирование размеров. Каждый новый запрос на вычисление системы ограничений опирается на начальное состояние геометрических объектов, оставшееся в результате предыдущего запроса **GCE_Evaluate**.

Важно понимать, что начальное состояние геометрии является неотъемлемым условием задачи удовлетворения ограничений. Два эскиза, имеющие одинаковый набор геометрических объектов и ограничений, но отличающиеся начальным состоянием, могут иметь различные решения. C3D Solver принимает во внимание начальное состояние и стремится удовлетворить все заданные ограничения с минимальными отклонениями от начального приближения при минимальном количестве изменяемых объектов.

Начальное приближение имеет определяющее значение не только при запросе на вычисление, но и в момент создания ограничений. Взаимное размещение геометрических объектов показывает вариант формулировки ограничения. Например, если в момент создания касания двух окружностей (**GCE_AddTangent**), одна окружность располагается внутри другой, то решатель сформулирует касание, при котором одна окружность всегда будет оставаться внутри другой. Подробнее о взаимном размещении геометрических объектов см. параграф *Взаимное размещение объектов*.

7.4.4 Недоопределенные системы ограничений

В предыдущем параграфе обсуждалась ситуация переопределения, когда в системе ограничений присутствует больше ограничений, чем необходимо. В данном параграфе обсуждается другая ситуация, естественная для процесса черчения, когда в эскизе остаются геометрические объекты, не полностью определенные ограничениями. В начале черчения эскиз может вообще не иметь ограничений, а все геометрические объекты будут иметь полную степень свободы. Соответственно, чем больше логических ограничений

и размеров добавляется в эскиз, тем меньше степени свободы остается объектам. У полностью определенного эскиза не остается и одной степени свободы. Заметим, что в одном и том же эскизе одновременно могут быть обе ситуации – часть эскиза переопределена, остальная часть может быть недоопределена, и в тоже время, какая-то часть геометрии может быть определена полностью.

Большая часть методов решения, реализованных в C3D Solver, ориентирована именно на недоопределенные системы ограничений. Кроме того, используются некоторые преимущества недоопределенных случаев для разбиения системы ограничений в целом на последовательность подсистем размером меньше, вычисляемых одна после другой. В тех или иных ситуациях недоопределенности решатель выбирает одно из оптимальных решений, основываясь на следующих принципах:

- для удовлетворения ограничений меняется как можно меньше геометрических объектов;
- те геометрические объекты, которые затрагиваются решением, получают минимальное смещение от своего начального приближения (см.параграф *Начальное приближение*).

7.4.5 Анализ степеней свободы

В процессе параметрического черчения важно контролировать, какие из геометрических объектов эскиза уже полностью определены, а какие нуждаются в дополнительных ограничениях. В параграфе *Недоопределенные системы ограничений* обсуждались принципы выбора решения для тех геометрических объектов, которым не хватает ограничений, чтобы однозначно определить свое состояние. Показать пользователю (визуализировать) недоопределенные геометрические объекты помогают функции анализа степеней свободы.

Одна из них вычисляет степени свободы точки:

```
GCE_point_dof GCE_PointDOF(GCE_system gSys, geom_item pnt)
```

Данный метод позволяет выяснить текущую степень свободы точки, которая может быть контрольной точкой любого геометрического объекта, как например центр окружности или эллипса, контрольная точка сплайна, один из концов отрезка и т.д.

Еще одна функция вычисляет степень свободы геометрической координаты:

```
GCE_point_dof GCE_CoordDOF(GCE_system gSys, geom_item g, coord_name cName)
```

Данный метод позволяет выяснить текущую степень свободы геометрической координаты, например, координаты точки или вектора, угол нормали прямой, угол наклона эллипса, радиус окружности и т.д.

7.4.6 Информационные запросы

В данном параграфе приведен ряд функций, с помощью которых приложение может узнать различные сведения об объектах, зарегистрированных в системе ограничений, или результаты диагностики и вычислений. Эти функции предназначены для осуществления справочных запросов, поэтому их вызов не приведет к изменению состояния системы ограничений или отдельных её данных.

Тип геометрии. Тип геометрического объекта, зарегистрированного в системе ограничений, можно узнать с помощью функции

```
geom_type GCE_GeomType(GCE_system gSys, geom_item g),
```

значением которой является перечислительный тип геометрического объекта **g**, зарегистрированного в системе **gSys**.

Возвращаемые значения:

- GCE_POINT – двумерная точка;
- GCE_LINE – двумерная прямая;
- GCE_CIRCLE – двумерная окружность;
- GCE_ELLIPSE – двумерный эллипс;
- GCE_SPLINE – двумерный сплайн;
- GCE_PARAMETRIC_CURVE – двумерная параметрическая кривая общего вида;
- GCE_BOUNDED_CURVE – кривая, ограниченная концевыми точками.

Ниже приведен перечень функций для информационных запросов:

Тип базовой кривой возвращает функция

```
geom_type GCE_BaseCurveType(GCE_system gSys, geom_item crv)
```

Связность с ограничениями оценивает функция

```
bool GCE_IsConstrainedGeom(GCE_system gSys, geom_item g)
```

Удовлетворенность ограничения оценивает функция

```
bool GCE_IsSatisfied(GCE_system gSys, constraint_item cItem)
```

Вырожденные геометрические объекты возвращает функция

```
std::vector<geom_item> GCE_DiagnoseGeometry(GCE_system gcSys)
```

Текущее заданное значение размера (не измеренное) возвращает функция

```
double GCE_DimensionParameter(GCE_system gSys, constraint_item dItem)
```

Оценку соответствия координат точки заданным ограничениям выдает функция

```
bool GCE_CheckPointSatisfaction(GCE_system gSys, geom_item pnt
                                , point_type cp
                                , double px
                                , double py)
```

Состояние решения данного ограничения возвращает функция

```
GcConState GCE_GetConstraintState(GCE_system, constraint_item gc_item)
```

Состояние системы ограничений по результатам последнего решения возвращает функция

```
GcConstraintStatus GCE_GetConstraintStatus(GCE_system gSys)
```

7.4.7 Драггинг геометрических объектов

В данном параграфе приведены функции, обслуживающие режим для интерактивного манипулирования недоопределенным эскизом, называемым драггинг.

Инициализировать режим драггинга контрольной точки объекта:

```
GCE_result GCE_PrepareDraggingPoint(GCE_system gSys, GCE_dragging_point drgPnt
                                                   , GCE_point curXY)
```

Инициализировать режим драггинга контрольной точки множества объектов:

```
GCE_result GCE_PrepareDraggingPoint(GCE_system gSys,
                                    const std::vector<GCE_dragging_point> & cPntArr,
                                    GCE_point curXY)
```

Инициализировать режим перетаскивания множества объектов:

```
GCE_result GCE_PrepareMovingGeoms(GCE_system gSys,
                                  std::vector<geom_item> & geoms,
                                  GCE_point curXY)
```

Переместить указатель драггинга:

```
GCE_result GCE_MovePoint(GCE_system gcSys, GCE_point curXY)
```

7.4.8 Геометрическая трансформация

C3D Solver позволяет осуществить геометрическое преобразование параметрического эскиза по матрице, которая может содержать сдвиг, поворот и масштабирование. Для выполнения указанной трансформации необходимо для системы ограничений эскиза вызвать метод

```
GCE_result GCE_Transform(      GCE_system  gSys,
                         const MbMatrix &  mat).
```

Следует принимать во внимание, что данный метод осуществляет преобразование, как геометрии, так и ограничений. Например, линейные размеры могут менять свои значения, если матрица трансформации содержит масштабирование.

Надо заметить, что в результате трансформации возможно, что часть ограничений станет неудовлетворенной, о чем можно запросить с помощью вызова **GCE_IsSatisfied**. Ограничения, которые сохранили удовлетворенность, называются *инвариантными* относительно данной трансформации.

7.4.9 Тест избыточности

C3D Solver предусматривает различные способы, позволяющие оценить избыточность ограничений. Методы

```
GCE_result GCE_DeviateDimension(GCE_system gSys, constraint_item dItem, double delta)␣
↪
```

```
GCE_result GCE_DeviationTest(GCE_system gSys, constraint_item dItem, double delta)
```

оценивают избыточность ограничений на основе отклонения размерных ограничений.

7.5 ДВУМЕРНЫЕ СПЛАЙНЫ И ПАРАМЕТРИЧЕСКИЕ КРИВЫЕ

В этой главе рассматривается работа с такими кривыми, как сплайн, параметрическая кривая общего типа, а также эллипс. Эти кривые объединяет то, что C3D Solver принимает во внимание не только геометрию, но и параметризацию данных кривых.

7.5.1 Сплайновые кривые

C3D Solver использует NURBS-кривые в качестве математической основы сплайновых кривых. NURBS-представление стало фактически промышленным стандартом для представления кривых свободной формы и поддерживается практически всеми САПР. Общепризнанным каноническим трудом на данную тему является книга The NURBS Book авторов Les Piegl и Wayne Tiller (издательство Springer).

C3D Solver предоставляет два способа определения сплайна:

- сплайн по набору характеристических точек, заданных значениями их координат;
- сплайн, проходящий через набор интерполяционных точек, заданных дескрипторами, ранее зарегистрированных геометрических объектов.

Сплайн на основе NURBS-кривой создается с помощью метода

```
geom_item GCE_AddSpline(GCE_system gSys, GCE_spline spl).
```

Функция принимает структуру данных GCE_spline, которая содержит в себе информацию, определяющую сплайн: начальные значения контрольных точек, массив интерполяционных точек (при необходимости), веса , признак замкнутости и т.д.

Специальное ограничение, фиксирующее вектор производной сплайна, создается методом

```
constraint_item GCE_FixSplineDerivative(GCE_system  gSys,
                                        geom_item   spline,
                                        double      par,
                                        uint        derOrder,
                                        GCE_vec2d * fixVal = NULL );
```

7.5.2 Параметрические кривые общего вида

Применения параметрической кривой. Одно из полезных применений параметрической кривой связано с участием в эскизе неподвижных кривых с не поддерживаемым типом геометрии. Например, планарный эскиз может быть создан в контексте трехмерной модели САПР, откуда в эскиз могут приходить произвольной формы проекции 3D-кривых. Проекционные кривые, участвующие в системе ограничений в виде параметрических кривых, дают возможность привязывать геометрию эскиза к проекционным объектам. Другое применение это оформлении чертежей, фрагменты которых получены проецированием геометрии из 3D-моделей и сохраняющими с ней ассоциативную связь.

7.5.3 Ограничения, основанные на параметризации кривых

Для создания специальных ограничений кривых в параметрическом представлении, а также сплайна и эллипса, используются методы:

```
constraint_item GCE_AddPointOnPercent(GCE_system gSys,
                                      geom_item  curve,
                                      geom_item  pnt[3],
                                      double     k)
```

```
constraint_item GCE_AddPointByMetricPercent(GCE_system gSys, geom_item curve
                                                           , geom_item pnt[3]
                                                           , double k)
```

```
constraint_item GCE_AddFixCurvePoint(GCE_system gSys, geom_item curve, geom_item pnt)
```

```
constraint_item GCE_AddPointOnParEllipse(GCE_system gSys, geom_item pnt
                                                        , geom_item ellipse
                                                        , double t)
```

7.6 ТРЕХМЕРНЫЙ ГЕОМЕТРИЧЕСКИЙ РЕШАТЕЛЬ

Данный раздел рассматривает еще один программный компонент в составе модуля C3D Solver – трехмерный геометрический решатель. Трехмерный геометрический решатель применяется для геометрического моделирования, когда объекты 3D-пространства требуется организовать в схему взаимосвязей, а именно подчинить их геометрическим ограничениям и размерам. Данная потребность проистекает из таких задач:

- Объединение твердотельных деталей в сборки;
- Кинематический анализ, включая обратную кинематическую задачу;
- Анимация твердотельных сборок;
- Моделирование трехмерных каркасных конструкции (wireframes).

В сочетании с компонентом поиска столкновений (см. главу *Определение столкновений тел*) геометрический решатель помогает пользователю САПР оценивать границы движения деталей механизмов.

7.6.1 Термины и определения

Локальная система координат (ЛСК) – система координат, определяющая положение объектов 3D-пространства относительно *мировой системы координат*. ЛСК задается точкой **начала координат** и тройкой векторов – осями Z, X, Y, которые применительно к задачам C3D Solver всегда ортонормированны. С понятием ЛСК связана такое понятие, как **Трансформация**.

Геометрический объект – основной предмет вычислений модуля C3D Solver, представляющий объект 3D-пространства, такой как точка, кривая, поверхность или ЛСК. Модуль C3D Solver поддерживает определенный набор типов 3D-геометрии, приведенных в параграфе *Типы поддерживаемой геометрии*. Заметим, что для C3D Solver абстракция твердого тела представлена не топологической структурой, как в модуле C3D Modeler, а группой геометрических объектов, объединенных в **Кластер** с общей ЛСК. Подробнее об этом см. в параграфе *Кластеризация геометрической сцены, моделирование сборок*.

Стандартное положение – это ЛСК, у которой начало и оси совпадают с *мировой системой координат*.

Трансформация – применительно к задачам модуля C3D Solver, это преобразование геометрического объекта только с помощью комбинации перемещений и вращений. Математически трансформация представляется в виде квадратной матрицы размером 4×4, называемой Матрица трансформации. С помощью матрицы трансформации удобно представлять ЛСК, имея ввиду, что любую ЛСК можно получить трансформацией из *стандартного положения* (см. **Стандартное положение**). Заметим, что C3D Solver имеет дело только с трансформациями, сохраняющими расстояние между любыми точками геометрического объекта. Таким образом, в C3D Solver запрещены трансформации, искажающие ортогональность или единичность векторов ЛСК, например масштабирование.

Жесткое множество или Кластер – подмножество геометрических объектов, объединенных общей ЛСК, которая рассматривается C3D Solver, как вычисляемый геометрический объект. В геометрической модели с помощью кластеров представляют твердые тела и геометрически-жесткие подсборки. В решателе любой геометрический объект с типом ЛСК может быть кластером.

Геометрическое ограничение – отношение, задающее связь между двумя, тремя и более геометрическими объектами, называемыми *аргументами ограничения*. Примеры геометрических ограничений: прямая L параллельна плоскости P, цилиндр C касается плоскости P, расстояние от плоскости P1 до плоскости P2 равно 10.0 и т.д.

Аргумент ограничения – геометрический объект или числовой параметр, связанный геометрическим ограничением или размером.

Система геометрических ограничений – это набор взаимосвязанных геометрических объектов и ограничений. Система геометрических ограничений формулирует задачу удовлетворения геометрических ограничений (ЗУГО, англ. GCSP – geometric constraint satisfaction problem), ставящей цель найти положение геометрических объектов трехмерного пространства, удовлетворяющее всем заданным ограничениям.

Начальное приближение – значения координат всех геометрических объектов и переменных системы ограничений, которые принимаются решателем, как близкое к желаемому решению состояние. Начальное приближение обычно удовлетворяет большей части ограничений и предлагается как стартовое условие ЗУГО. Также от начального приближения зависит выбор решения, которое сделает C3D Solver для задач с несчетным множеством решений (см. Недоопределенная система ограничений). Для одной и той же системы ограничений C3D Solver может давать разные решения при разных начальных приближениях.

Задача удовлетворения геометрических ограничений (ЗУГО) – задача, ставящая цель трансформировать набор геометрических объектов, имеющих предварительную оценку (**Начальное приближение**), в положение, удовлетворяющее всем заданным ограничениям. Надо заметить, что кроме списка объектов и ограничений, формулировка ЗУГО включает и Начальное приближение, которое также влияет на результат решения.

Трехмерный геометрический решатель – программный компонент, отвечающий за решение задачи удовлетворения ограничений для трехмерных объектов. В составе модуля C3D Solver данный компонент имеет техническое название GCM (*сокр. **G**eometric **C**onstraint **M**anager*). Другой компонент, с обозначением GCE, отвечает за вычисление планарных объектов (см. *ДВУМЕРНЫЙ ГЕОМЕТРИЧЕСКИЙ РЕШАТЕЛЬ)*.

Полностью определенная система ограничений – это система геометрических ограничений, имеющая единственное или конечное число возможных решений.

Несовместная система ограничений – система ограничений, не имеющая решений. Соответственно система ограничений, имеющая хотя бы одно решение, называется **Совместной**.

Недоопределенная система ограничений – это система геометрических ограничений, допускающая бесконечное множество решений.

Переопределенная система ограничений – это система ограничений, в которой имеются ограничения, удаление которых не меняет множества решений или несовместную систему делает совместной. Переопределенность системы ограничений обычно говорит о наличии лишних ограничений, удаление которых либо ничего не меняет либо устраняет противоречия.

Размерное ограничение или Размер – геометрическое ограничение, одним из аргументов которого является числовым параметром, измеряющим расстояние или угол между двумя геометрическими объектами. Численная величина этого параметра называется **Значением размера**.

Линейный размер – размерное ограничение, измеряющее расстояние от точки до точки, принадлежащие двум геометрическим объектам. Например, расстояние от точки до плоскости измеряется, как расстояние от точки до ее ближайшей проекции на плоскости. Значение линейного размера всегда задается в единицах расстояния (метр, миллиметр, дюйм и т.д.) и может принимать как положительные, так и отрицательные значения, включая нуль.

Угловой размер – размерное ограничение, измеряющее угол между двумя направлениями, взятыми от пары геометрических объектов. Другими словами значение углового размера равно углу, на который следует повернуть вектор первого объекта до совмещения с вектором второго объекта. Например угол между прямой и плоскостью определяется углом поворота прямой до совпадения с плоскостью по наикратчайшему пути. Значения угловых размеров задаются в радианах.

Логическое ограничение – это геометрическое ограничение, измеряющее булевскую величину, принимающую два значения: **true**, если ограничение выполнено (удовлетворено) и **false**, если ограничение не выполнено. Примерами логических ограничения являются параллельность двух прямых, касание цилиндра и плоскости, соосность двух конусов и т.д. Заметим, что к логическим ограничениям не относятся размеры.

Параметрическая модель – это *геометрическая модель*, которая позволяет получать отличающиеся друг от друга экземпляры этой модели путем изменения независимых (управляющих) числовых параметров. Система геометрических ограничений являться параметрической моделью, если в ней задан хотя бы один *управляющий размер* (см. *Управляющие размеры*).

7.6.2 Назначение геометрического решателя GCM

В составе геометрического ядра C3D трехмерный геометрический решатель имеет внутреннее техническое обозначение – GCM (*сокр. **G**eometric **C**onstraint **M**anager*). Префикс **GCM_** в именах функций и структур данных говорит о принадлежности к API трехмерного решателя. Компонент GCM вычисляет положение и координаты геометрических объектов, удовлетворяющих заданному набору ограничений и размеров, тем самым обеспечивает целостность геометрической модели приложения. Геометрический решатель GCM имеет дело с такими геометрическими объектами пространства, как точка, прямая, плоскость, окружность, сфера, цилиндр, конус, тор или их объединение в жесткие множества (Кластер). Положение геометрических объектов может быть подчинено, зависимостям в виде ограничений из определенного набора типов, включающего логические ограничения: параллельность, перпендикулярность, касание, совпадение, соосность, симметрия и разного рода размеры в единицах расстояния и угла.

Базовая функциональность геометрического решателя GCM доступна через заголовочный файл **gcm_api.h**. Взаимодействие приложения с решателем ограничений осуществ-

ляется на основе простых структур данных C++, все они объявлены в файле **gcm_types.h**.

7.6.3 Встраивание в приложение компонента GCM

Геометрический решатель GCM построен, как общецелевой компонент трехмерного параметрического моделирования. Это значит, что его можно встроить в любое приложение, где требуется снабдить трехмерную геометрическую модель функциональностью размеров и ограничений. Главная особенность в том, что интеграция с C3D Solver не требует ничего менять в структурах данных приложения.

Программный интерфейс C3D Solver имеет свою собственную отвлеченную (abstract) систему типов данных, никак не связанную с типами данных модуля C3D Modeler. Поэтому для работы с решателем приложение может использовать или не использовать структуры данных модуля C3D Modeler. Чтобы подключить геометрический решатель к управлению геометрической моделью, необходимо обеспечить следующую цепочку взаимодействий приложения с решателем:

1. **Создать систему геометрических ограничений** (см. *GCM_CreateSystem*). Это первое что необходимо сделать, что бы начать работу с C3D Solver.
2. **Добавить объекты в систему ограничений**. Нужно добавить геометрические объекты в систему с помощью вызовов **GCM_AddGeom**, **GCM_SubGeom** (подробнее в *Добавление и удаление геометрических объектов*). Дескрипторы (см. *Таблица 31.6.1. Дескрипторные типы данных*), которые выдадут эти функции, используются для хранения ссылок объектов приложения на геометрические объекты C3D Solver. Для оптимизации памяти не рекомендуется добавлять объекты, пока они не связаны ограничениями.
3. **Добавить геометрические ограничения в систему.** Добавить ограничения с помощью вызовов, рассмотренных в *Добавление и удаление геометрических ограничений*. Ограничения приложения ссылаются на ограничения C3D Solver с помощью специальных дескрипторов (см. *Таблица 31.6.1. Дескрипторные типы данных*).
4. **Решить систему ограничений.** С помощью вызова *GCM_Evaluate* C3D Solver вычисляет новое состояние геометрических объектов, удовлетворяющее объявленным ранее ограничениям.
5. **Применить результаты вычислений**. Необходимо применить вычисленные координаты объектов из системы ограничений C3D Solver к геометрическим объектам приложения. Результат вычислений хранится внутри C3D Solver, поэтому приложение должно обновить состояние своих объектов, запросив у решателя новые значения координат.
6. **Удаление объектов и ограничений**. При завершении сеанса работы с системой ограничений приложение вызывает метод **GCM_RemoveSystem**. Также в процессе жизни системы ограничений могут удаляться отдельно ограничения и объекты, как только в них отпадает необходимость (см. *Добавление и удаление геометрических объектов*).

Для адаптации геометрического решателя к нативным типам данных приложения следует организовать интерфейс между приложением и компонентом GCM. На рис. 1 изображена примерная схема взаимодействия решателя ограничений с приложением, где предлагается на стороне приложения реализовать специальный Constraint Manager, отвечающий следующим задачам:

- скрывает за собой API C3D Solver и предоставляет для приложения интерфейс, более удобный, выраженный в нативных данных приложения;

- загружает в решатель данные об объектах и ограничениях 3D-модели (формулировка геометрической задачи);
- обрабатывает командные запросы такие, как добавление/удаление данных системы ограничений, запросы на вычисления;
- обеспечивает обратную связь с решателем, применяет результаты вычислений C3D Solver к геометрической модели приложения;
- обновляет данные решателя для синхронизации с моделью приложения.

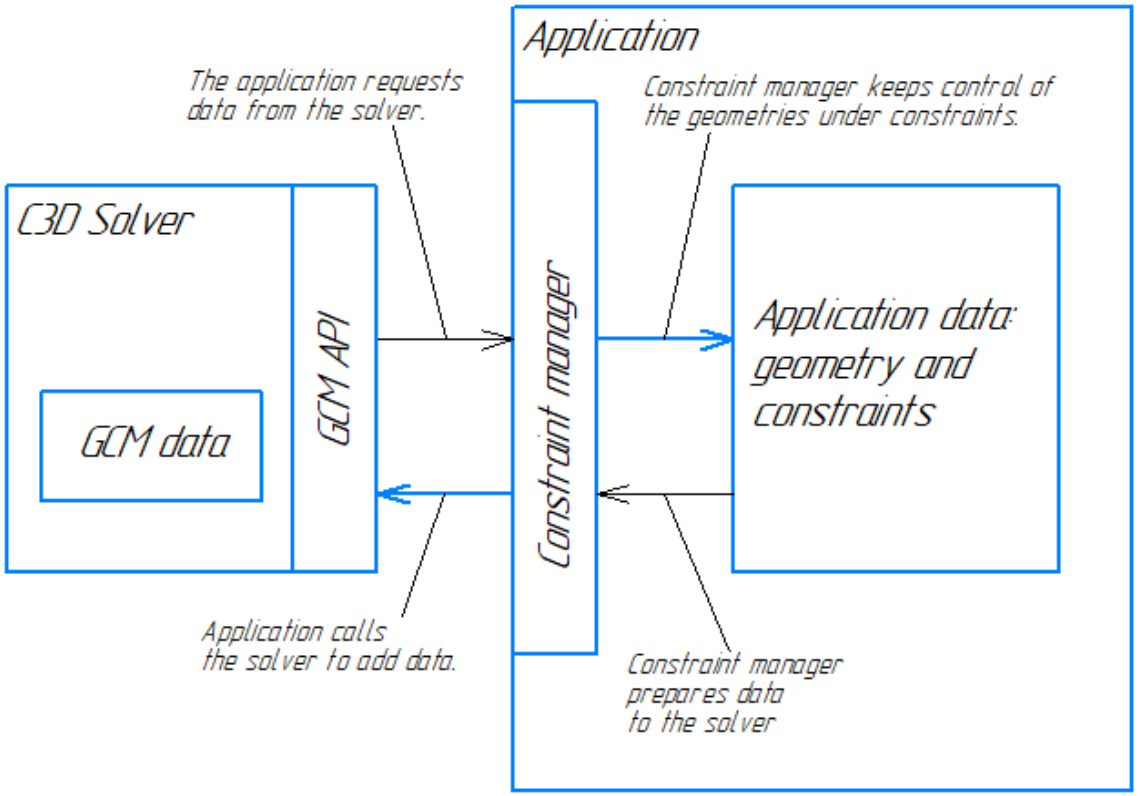


Рис. 1

Схема на рис. 1 не является обязательной, а только демонстрирует один из возможных путей интеграции C3D Solver с приложением.

Также компонент GCM не позволяет сохранять данные пользователя в файл документа, поэтому разработчик приложения должен позаботиться о чтении/записи системы ограничений 3D-модели в документе приложения.

7.6.4 Типы поддерживаемой геометрии

Геометрическая сцена под управлением GCM, формируется в виде набора геометрических объектов, принадлежащих одному из следующих типов:

- Точка
- Прямая
- Плоскость
- Цилиндр или Конус

- Сфера
- Тороид
- Окружность
- Локальная система координат (ЛСК)

Геометрические объекты перечисленных выше типов могут группироваться образуя геометрически-жесткие объединения – кластеры, которые обычно в приложении играют роль твердого тела или подсборки. В системе ограничений кластеры регистрируются под типом ЛСК (значение enum GCM_LCS).

О типе любого геометрического объекта, зарегистрированного в системе ограничений, можно узнать с помощью функции

```
GCM_g_type GCM_GeomType(GCM_system gSys, GCM_geom g)
```

которая имеет следующие входные параметры:

- **gSys** – дескриптор системы геометрических ограничений;
- **g** – дескриптор геометрического объекта, принадлежащего системе **gSys**.

Функция вернет одно из следующих значений, перечисляемых enum GCM_g_type:

GCM_NULL_GTYPE	для пустого объекта (g = GCE_NULL)
GCM_POINT	для точки
GCM_LINE	для прямой
GCM_PLANE	для плоскости
GCM_CYLINDER	для цилиндра
GCM_CONE	для конуса
GCM_SPHERE	для сферы
GCM_TORUS	для тороида
GCM_CIRCLE	для окружности
GCM_LCS	для локальной системы координат
GCM_MARKE	для маркера в виде тройки <O,Z,X>
GCM_SPLINE	для сплайна (пока не поддерживается)

7.6.5 Типы поддерживаемых ограничений

Любое геометрическое ограничение связывает между собой геометрические объекты, называемые *аргументами*ограничения. Аргументом ограничения может быть любой объект, относящийся к одному из типов геометрии, перечисленных в параграфе *Типы поддерживаемой геометрии*. Например, цилиндр может быть аргументом ограничения **касание**. По числу геометрических аргументов ограничения различают, как *унарные*, *бинарные* и *тернарные*. Они связывают соответственно один, два или три объекта. Примером тернарного ограничения является **симметрия**, в которой участвуют два объекта, отражающие друг друга, и плоскость зеркальной симметрии. В общем случае ограничение может иметь N аргументов. *Логические ограничения* предусматривают только зависимости между геометрическими объектами. *Размерные ограничения* устанавливают связь между геометрическими объектами и числовым параметром (значением размера), измеряющим расстояние или угол, поэтому у размерных ограничений последний аргумент всегда числовой (скалярный). Типы ограничений, которые поддерживает геометрический решатель GCM приведены в таблице 1:

Таблица 1 - Типы геометрических ограничений (enum GCM_c_type)

Ограничение	Арность(число аргументов)	Обозначение в API
Логические ограничения		
Совпадение		GCM_COINCIDENT
Концентричность		GCM_CONCENTRIC
Параллельность		GCM_PARALLEL
Перпендикулярность		GCM_PERPENDICULAR
Касание		GCM_TANGENT
Зеркальная симметрия		GCM_SYMMETRIC
Зависимость определенная пользователем	N	GCM_DEPENDENT
Размерные ограничения		
Расстояние	+1	GCM_DISTANCE
Угол	+1	GCM_ANGLE
Радиус	+1	GCM_RADIUS
Ограничения паттернов		
Подчинение паттерну	+1	GCM_PATTERNED
Линейный паттерн (трансляционный)		GCM_LINEAR_PATTERN
Угловой паттерн (ротационный)		GCM_ANGULAR_PATTERN
Механические передачи		
Механическая передача	+2	GCM_TRANSMITTION
Кулачковый механизм	+2	GCM_CAM_MECHANISM

7.6.6 Базовые типы данных API решателя GCM

Взаимодействие приложения с решателем ограничений осуществляется на основе простых структур данных C++, все они объявлены в файле *gcm_types.h*. Среди них центральное место занимают дескрипторные типы данных, которые служат для идентификации любых объектов, находящихся под контролем решателя (таблица 1):

Таблица 1 - Дескрипторные типы данных

Тип данных решателя	Реализация	Интерпретация
GCM_system	void *	дескриптор системы ограничений
GCM_object	struct { size_t id; }	дескриптор вычислительного объекта
GCM_geom	GCM_object	дескриптор геометрического объекта
GCM_constraint	GCM_object	дескриптор ограничения
GCM_pattern	GCM_object	дескриптор паттерна

Типы данных, которые перечисляют конечные наборы значений, приведены в табл. 2:

Таблица 2 - Перечислительные типы данных

Тип данных решателя	Интерпретация
GCM_g_type	тип геометрического объекта
GCM_c_type	тип ограничения
GCM_alignment	вариант выравнивания
GCM_tan_choice	вариант касания
GCM_result	диагностический код
GCE_scale	опция масштабируемости паттерна

Структуры данных, перечисленные ниже (табл. 3), служат для передачи набора параметров при вызове функций API. Например, с помощью структуры GCM_g_record в решатель передаются тип геометрического объекта и его позиционирующая ЛСК.

Таблица 3 - Структуры данных

Тип данных решателя	Интерпретация
GCM_vec3d	координаты трехмерного вектора
GCM_point	координаты трехмерной точки
GCM_g_record	запись геометрического объекта
GCM_extra_param	параметр вызова GCM_dependent_func
GCM_c_arg	аргумент ограничения геометрический или числовой
GCM_c_record	запись ограничения, тип и его аргументы

7.6.7 Система геометрических ограничений

Система геометрических ограничений это набор геометрических объектов, взаимосвязанных друг с другом через ограничения и размеры. Типы поддерживаемых геометрических объектов и ограничений приведены соответственно в параграфах *Типы поддерживаемой геометрии*, *Типы поддерживаемых ограничений*. Предполагается, что геометрическая модель, которая создается в приложении, имеет свою систему ограничений. С программной точки зрения это означает, что каждая модель ассоциируется с системой ограничений через дескриптор типа GCM_system, а каждый его геометрические объект и ограничение представлены своими уникальными дескрипторами типов GCM_geom и GCM_constraint.

Перед тем как начать работать с геометрическими ограничениями необходимо объявить для модели систему ограничений с помощью вызова:

```
GCM_system GCM_CreateSystem()
```

Функция вернет пустую систему ограничений в виде дескриптора GCM_system, который представляет собой указатель на экземпляр внутренней структуры данных геометрического решателя. Все дальнейшие манипуляции с системой ограничений будут осуществляться через данный дескриптор. Например, если мы хотим объявить точку, необходимо для созданной системы ограничений вызвать функцию:

```
GCM_geom GCM_AddPoint(GCM_system gSys, MbCartPoint3D pVal)
```

Результатом функции является дескриптор геометрической точки, принадлежащей системе ограничений **gSys**. Параметр **pVal** задает начальные координаты <x,y,z> точки.

При завершении работы с геометрической моделью, обязательно следует вызвать функцию удаления ее системы ограничений:

```
void GCM_RemoveSystem(GCM_system gSys)
```

которая полностью освободит память, занимаемую системой ограничений со всеми данными объектов и ограничений. После вызова **GCM_RemoveSystem** дескриптор системы ограничений становится недействительным, т.е. дальнейшее использование такого дескриптора может привести к аварийному прекращению работы приложения.

Функция

```
void GCM_ClearSystem(GCM_system gSys)
```

делает систему ограничений пустой, освобождая память только от объектов и ограничений, но оставляет систему ограничений действительной для дальнейшей работы.

Также с помощью вызова

```
bool GCM_ReadSystem(GCM_system gSys, reader & in)
```

можно сохранить систему ограничений в поток, а вызовом

```
bool GCM_WriteSystem(GCM_system gSys, writer & out)
```

полностью восстановить состоянии системы. Функции сохранения и воспроизведения из потока гарантируют сохранность значений дескрипторов всех ограничений и объектов.

Внимание! Функции чтения/записи системы ограничений сохраняют только те данные об объектах и ограничениях, которые загружены в C3D Solver. Все исходные данные об объектах и ограничениях модели хранятся на стороне приложения, сохранность которых – забота разработчика приложения. Функции **GCM_ReadSystem**, **GCM_WriteSystem** могут оказаться полезными в следующих случаях:

- передача данных от разработчика приложения разработчику C3D Solver для отладки, в целях технической поддержки. См. также параграф *Журналирование вызовов API решателя GCM*, где рассмотрено журналирование вызовов GCM в целях отладки и техподдержки;
- конвертирование нативных форматов в C3D-файл, с некоторой потерей данных;
- временное сохранение состояния системы ограничений в различных целях (возможна частичная потеря данных для таких ограничений, как механические передачи и зависимости GCM_DEPENDENT).

7.6.8 Представление геометрических объектов

Трехмерный геометрический решатель GCM работает с определенным представлением данных геометрических объектов, проиллюстрированной в таблице 1. Все объекты выражены в координатах точек, векторов и чисел (скаляров), записанных в определенной для каждого типе форме.

Записи всех перечисленных геометрических объектов можно унифицировать в виде кортежа значений: <$OZXYR_AR_B$>

где четверка значений O Z X Y – задает размещение объекта в пространстве, а именно его ЛСК, заданное точкой начала СК и тремя осям Z, X, Y, а два скалярных значения R_A, R_B задают параметры радиусных объектов, таких как окружность, цилиндр, конус, тороид, сфера.

Считается, что геометрический объект любого типа имеет ассоциированную с ним локальную систему координат. Даже точка или сфера может избыточно представлена в виде ЛСК, для которых безразлично направление осей Z, X, Y, но имеют значение координаты точки начала O. Аналогично плоскость имеет свою ЛСК, начало которой задает положение плоскости, а ось Z задает нормаль плоскости. При этом значения осей X, Y для плоскости игнорируются.

В программном интерфейсе трехмерного решателя унифицированная форма записи в виде <$OZXYR_AR_B$> используется в структуре GCM_g_record, поля данных которой приведены в таблице 2:

Таблица 2 - Описание полей данных структуры GCM_g_record.

Тип данных	Поле данных	Интерпретация (геометрический смысл разъясняется в таблице 31.8.1.)
GCM_g_	type	дескриптор системы ограничений
GCM_pc	origin	Точка позиционирования геометрического объекта (начало ЛСК, центр окружности или сферы и т.д.)
GCM_ve	axisZ	Ось «Z» локальной системы координат (например, в случае прямой ось axisZ – это её направляющий вектор).
	axisX	Ось «X» локальной системы координат
	axisY	Ось «Y» локальной системы координат
double	radiusA	Задает радиус в случае окружности, сферы или цилиндра, для конуса задает радиус основания, для тора задает «Большой» радиус.
	radiusB	Задает «Малый» радиус в случае тора или конуса

GCM_g_record обобщает запись любой геометрии, поэтому в зависимости от типа геометрии некоторые поля данных GCM_g_record могут не использоваться. Например, поля axisX, axisY нужны только для случая type = GCM_LCS, а для таких объектов, как плоскость, цилиндр, тор не имеют значения. Поле axisZ не используется для точки, и сферы, однако в остальных случаях она задает ориентацию объекта в пространстве.

В API решателя GCM предусмотрены вспомогательные функции **GCM_Point**, **GCM_Line**, **GCM_Plane**, **GCM_Cone**, **GCM_Cylinder**, **GCM_Circle**, **GCM_Torus**, **GCM_Sphere**, **GCM_SolidLCS**, позволяющие правильно заполнить структуру GCM_g_record. Подробнее о данных функциях см. параграф *Добавление и удаление геометрических объектов*.

7.6.9 Добавление и удаление геометрических объектов

Основным предметом вычислений геометрического решателя являются геометрические объекты, поэтому формирование системы ограничений начинается с того, что приложение объявляет в системе ограничений геометрические объекты, которым предстоит стать аргументами ограничений.

Каждый геометрический объект, объявленный в системе ограничений, получает свой уникальный идентификатор – дескриптор типа GCM_geom, а его геометрический тип (GCM_g_type) остается неизменным в течении жизни объекта. Ниже приведены вызовы API C3D Solver, добавляющие геометрические объекты в систему.

Точка добавляется с помощью вызова

```
GCM_geom GCM_AddPoint(GCM_system gSys, const MbCartPoint3D & pVal)
```

Результатом функции является дескриптор геометрической точки с типом **GCM_POINT**, принадлежащей системе ограничений ***gSys*** .Параметр ***pVal*** задает начальные значения координат точки <x,y,z>.

Для создания геометрических объектов других типов, включая и точку, необходимо заполнить структуру данных GCM_g_record, представляющую собой унифицированную запись для геометрии любого типа, поддерживаемого решателем. Значение данной структуры принимает на вход функция

```
GCM_geom GCM_AddGeom(GCM_system gSys, const GCM_g_record & gRec)
```

Результатом вызова является дескриптор геометрического объекта, зарегистрированного в системе ***gSys***. Параметры геометрического объекта задаются полями структуры GCM_g_record:

- **gRec.type** задает геометрический тип объекта (GCM_g_type);
- **gRec.origin** задает положение объекта, начало его ЛСК;
- **gRec.axisZ** задает вектор оси Z ЛСК;
- **gRec.axisX** задает вектор оси X ЛСК;
- **gRec.axisY** задает вектор оси Y ЛСК;
- **gRec.radiusA** и **gRec.radiusB** задают скалярные параметры радиусных объектов.

Подробнее о представлении данных в структуре GCM_g_record изложено в параграфе *Представление геометрических объектов*.

Для удобства и правильного заполнения структуры GCM_g_record в API компонента GCM предусмотрены вспомогательные функции, приведенные ниже:

Функция

```
GCM_g_record GCM_Point(const MbCartPoint3D &)
```

фактически конвертирует значение координат трехмерной точки из типа данных MbCartPoint3D в GCM_g_record, воспринимаемой функцией GCM_AddGeom.

```
GCM_g_record GCM_Line(const MbCartPoint3D & org, const MbVector3D & axisZ)
```

возвращает запись прямой по точке и направляющему вектору.

```
GCM_g_record GCM_Plane(const MbCartPoint3D & org, const MbVector3D & axisZ)
```

возвращает запись о плоскости, заданной точкой и вектором нормали.

Функция для получения записи конуса:

```
GCM_g_record GCM_Cone(const MbCartPoint3D & centre, const MbVector3D & axis
                                  , double radiusA
                                  , double radiusB)
```

Входные параметры:

- **centre** – центр окружности-основания конуса,
- **axis** – направляющий вектор оси конуса,
- **radiusA** – радиус основания конуса,
- **radiusB** – радиус сечения конуса (малый радиус).

Предполагается, что параметры конуса описывают воображаемый усеченный конус, высота которого всегда равна единице длины, т.е. имеет единичное расстояние между поперечным сечением радиуса **radiusA** и сечением радиуса radiusB (см. иллюстрацию в таблице 1).

Функция для получения записи цилиндра:

```
GCM_g_record GCM_Cylinder(const MbCartPoint3D & centre, const MbVector3D & axis
                                  , double radius)
```

Входные параметры:

- **centre** – точка на оси цилиндра;
- **axis** – направляющий вектор оси конуса;
- **radius** – радиус цилиндра.

Функция для получения записи окружности:

```
GCM_g_record GCM_Circle(const MbCartPoint3D & centre, const MbVector3D & axis
                                    , double radius)
```

Входные параметры:

- **centre** – координаты центра окружности;
- **axis** – направляющий вектор оси окружности;
- **radius** – радиус окружности.

Функция для получения записи тора:

```
GCM_g_record GCM_Torus(const MbCartPoint3D & centre, const MbVector3D & axis
                                   , double majorR
                                   , double minorR)
```

Входные параметры:

- **centre** – координаты центра тора;
- **axis** – направление оси вращения тора;
- **majorR** – расстояние от цента тора до центра образующей окружности тора;
- **minorR** – радиус образующей окружности тора.

Функция для получения записи сферы:

```
GCM_g_record GCM_Sphere(const MbCartPoint3D & centre, double radius)
```

Входные параметры:

- **centre** – центр сферы;
- **radius** – радиус сферы.

Функция для получения записи прямоугольной правой (right-handed) ЛСК:

```
GCM_g_record GCM_SolidLCS(const MbCartPoint3D & org, const MbVector3D & axisZ
                                   , const MbVector3D & axisX)
```

Входные параметры:

- **org** – начало системы координат;
- **axisZ** – орт Z;
- **axisX** – орт X.

Результатом функции является запись ЛСК в виде четверки <O Z X Y>, где ось Z ориентирована по векторному произведению X□Y, что соответствует правосторонней ориентации тройки X-Y-Z (right-handed rule). Запись ЛСК имеет геометрический тип **GCM_LCS**.

Другой метод создания записи ЛСК имеет тоже имя, но принимает в качестве аргумента значение MbPlacement3D:

```
GCM_g_record GCM_SolidLCS(const MbPlacement3D &)
```

Данный метод допускает создание как правосторонней, так и левосторонней СК.

Результат, который возвращает функция **GCM_SolidLCS**, используется для задания в системе ограничений твердого тела (кластера) с помощью вызовов **GCM_AddGeom** или **GCM_SubGeom**.

Для получения пустой записи GCM_g_record, которая может быть использована для конструирования значения по умолчанию, можно применить вызов:

```
GCM_g_record GCM_NullGeom(void).
```

Ниже приведён фрагмент кода, демонстрирующий создание системы, включающей сферу **sph** с центром в начале координат и радиусом 10.0, а также цилиндр **cyl** с центром в точке <10, 0, 0>, ориентированный вдоль оси Z и имеющий тот же радиус.

```
GCM_system gSys = GCM_CreateSystem();
const MbCartPoint3D cylPos( 10.0, 0.0, 0.0 );
/*
Register two geoms, sphere and cylinder.
*/
GCM_geom sph = GCM_AddGeom(gSys, GCM_Sphere(MbCartPoint3D::origin, 10.0));
GCM_geom cyl = GCM_AddGeom(gSys, GCM_Cylinder(cylPos, MbVector3D::zAxis, 10.0));
...
...
/*
Finalize the constraint system to free its located memory.
*/
GCM_RemoveSystem(gSys);
```

Кроме вызова **GCM_AddGeom** в решателе предусмотрен другой метод для регистрации геометрических объектов, предусматривающий абстракцию твердого тела, а именно геометрически-жесткие множества, имеющие собственную ЛСК. Такие подмножество называется *кластером*. Для создания *кластера* необходимо объявить его ЛСК с помощью запроса:

```
GCM_geom lcs = GCM_AddGeom(gSys, GCM_SolidLCS(org, axisZ, axisX))
```

Значения ***org***, ***axisZ***, ***axisX*** задают позицию и ориентацию кластера (его ЛСК). Полученный дескриптор **lcs** будет задавать ЛСК геометрически-жесткого множества, элементы которого объявляются с помощью вызова:

```
GCM_geom GCM_SubGeom(GCM_system gSys, GCM_geom lcs, const GCM_g_record & gRec)
```

Входные параметры:

- **gSys** – система геометрических ограничений;
- **lcs** – дескриптор ЛСК кластера;
- **gRec** – запись параметров геометрического объекта, заданного в ЛСК кластера.

Функция возвращает дескриптор геометрического объекта, принадлежащего кластеру. Характерная особенность объекта, принадлежащего кластеру в том, что его положение в

пространстве подчинено ЛСК кластера, таким образом подчиненный объект меняет свое положение вместе с ЛСК кластера. Это основное его отличие от геометрических объектов, созданных методом **GCM_AddGeom**. Более подробно о работе с кластерами рассмотрено в параграфе *Кластеризация геометрической сцены, моделирование сборок*.

Геометрические объекты, возвращаемые функциями **GCM_SubGeom**, как и **GCM_AddGeom**, могут участвовать в ограничениях и размерах, т.е. могут быть аргументами ограничения.

Удаление геометрических объектов осуществляется вызовом

```
void GCM_RemoveGeom(GCM_system gSys, GCM_geom g)
```

Данный метод удаляет геометрический объект из системы и делает его дескриптор **g** не действительным. Если геометрический объект на момент вызова является аргументом одного из ограничений, то он фактически остается в системе, но его неизбежное удаление откладывается до тех пор, пока не будет удалено последнее из ограничений, в которых он участвует. Таким образом, пользователь API GCM может освобождать геометрический объект, как только он стал не нужен для приложения, не заботясь о том есть ли ограничения, которые его удерживают.

7.6.10 Добавление и удаление геометрических ограничений

Для добавления большинства геометрических или размерных ограничений можно воспользоваться вызовом:

```
GCM_constraint GCM_AddConstraint(GCM_system gSys, const GCM_g_record & cRec)
```

который требует заполнения структуры данных **cRec**, несущей данные о типе ограничения **cRec.type**, и массив аргументов ограничения **cRec.args**. Функция вернет дескриптор нового ограничения, зарегистрированного в системе **gSys**.

Сразу приведем пример добавления ограничения – совпадение точки и плоскости:

```
GCM_system gSys = GCM_CreateSystem();
GCM_c_record cRec; // The record of point and plane coincidence.
cRec.type = GCM_COINCIDENT;
/*
Some point and plane as arguments of the coincidence.
*/
cRec.args[0] = GCM_AddPoint(gSys, MbCartPoint3D::origin);
cRec.args[1] = GCM_AddGeom(gSys, GCM_Plane(MbCartPoint3D(0,0,1), MbVector3D::zAxis));
cRec.args[2] = GCM_NO_ALIGNMENT; // The alignment option does't matter in point-plane
↪case.
/*
Add coincidence to the system.
*/
GCM_constraint cItem = GCM_AddConstraint(gSys, cRec);
/*
Finalize the constraint system.
*/
GCM_RemoveConstraint(gSys, cItem);
GCM_RemoveSystem(gSys);
```

Для каждого типа ограничения принята определенная последовательность заполнения аргументов **args** в структуре GCM_c_record. В примере выше показано, что для ограниче-

ния GCM_COINCIDENT первые два элемента cRec.args[0] и cRec.args[1] – это дескрипторы аргументов ограничения Совпадение, а третий элемент cRec.args[2] задает опцию выравнивания.

Устройство структуры GCM_c_record. В таблице ниже приведена схема заполнения структуры GCM_c_record, передаваемой в метод **GCM_AddConstraint**, для различных типов ограничений.

Таблица 1 - Типы данных аргументов для различных ограничений (заполнение GCM_c_record)

Тип ограничения	**Типы аргументов**				
type	args[0]	args[1]	args[2]	args[3]	args[4]
GCM_COINCIDENT	GCM_geom	GCM_ge	GCM_alignment		
GCM_CONCENTRIC	GCM_geom	GCM_ge	GCM_alignment		
GCM_PARALLEL	GCM_geom	GCM_ge	GCM_alignment		
GCM_PERPENDICULA	GCM_geom	GCM_ge	GCM_alignment		
GCM_TANGENT	GCM_geom	GCM_ge	GCM_alignment	GCM_tan_ch	
GCM_SYMMETRIC	GCM_geom	GCM_ge	CM_geom	GCM_alignn	
GCM_DISTANCE	GCM_geom	GCM_ge	double	GCM_alignn	
GCM_ANGLE (планарный угол)	GCM_geom	GCM_ge	GCM_geom (ось вращения)	double	GCM_alignment
GCM_ANGLE (3D угол)	GCM_geom	GCM_ge	GCM_NULL (без оси)	double	GCM_alignment
GCM_RADIUS	GCM_geom	GCM_ge			

Значение опции выравнивания GCM_alignment рассматривается в параграфе *Опция выравнивания GCM_alignment*. Значения GCM_tan_choice будут рассмотрены в разделах, где обсуждается ограничение касание (GCM_TANGENT). В таблице 31.10.1. приведены только правила интерпретации типов данных, который принимает элемент типа GCM_c_arg из массива GCM_c_record::args для того или иного типа ограничения.

Таким образом структура данных GCM_c_record представляет собой унифицированный способ задания разного типа ограничений, имея только один метод **GCM_AddConstraint**. Однако для удобства разработчика в API GCM имеются альтернативные методы добавления ограничений, не требующие заполнения структуры GCM_c_record.

Для добавления любых **бинарных** ограничений, связывающих пару геометрических объектов, можно воспользоваться функцией:

```
GCM_constraint GCM_AddBinConstraint(GCM_system gSys, GCM_c_type cType
                                    , GCM_geom g1
                                    , GCM_geom g2
                                    , GCM_alignment aVal
                                    , GCM_tan_choice tVar)
```

Входные параметры функции:

- **gSys** – дескриптор системы ограничений;
- **cType** – значение типа из следующего списка: GCM_COINCIDENT, GCM_PARALLEL, GCM_PERPENDICULAR, GCM_TANGENT, GCM_CONCENTRIC;
- **g1**, **g2** – пара дескрипторов геометрических объектов, аргументы бинарного ограничения;

- **aVal** – опция выравнивания, позволяющая выбрать взаимную ориентацию для ориентируемых объектов (подробнее см. параграф *Опция выравнивания GCM_alignment*);
- **tVar** – опция выбора касания, позволяющего выбирать способы исполнения ограничения с типом GCM_TANGENT.

Результатом вызова является дескриптор бинарного ограничения, связывающего объекты **g1**, **g2**, с выбранными условиями выравнивания/касания **aVal**, **tVar**.

Размерное ограничение типа GCM_DISTANCE, определяющее расстояние между объектами, создаётся вызовом:

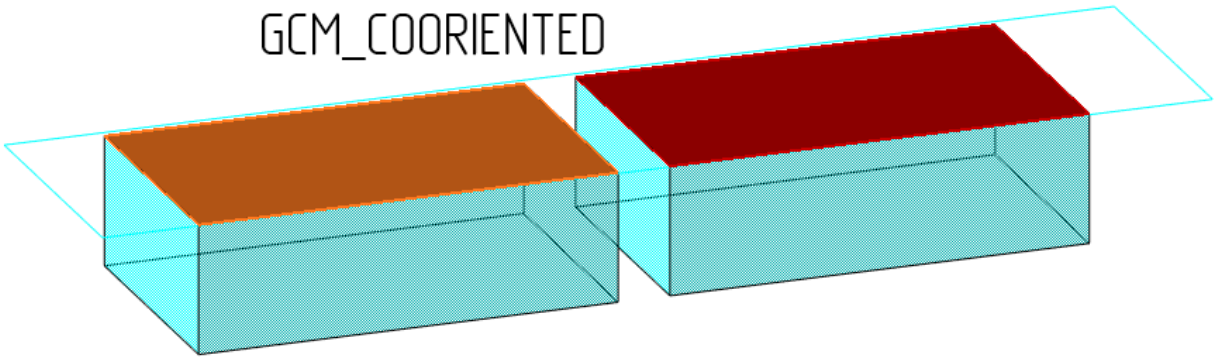
```
GCM_constraint GCM_AddDistance(GCM_system gSys, GCM_geom g1
                               , GCM_geom g2
                               , double dVal
                               , GCM_alignment aVal)
```

который принимает входные данные **g1**, **g2**, **aVal**, такие же, как у функции **GCM_AddBinConstraint**, числовое значение **dVal** – задает значение размера.

Аналогично работают вызовы, задающие угловой размер, зеркальную симметрию и радиусный размер: **GCM_AddAngle**, **GCM_AddSymmeric**,**GCM_FixRadius**. Они также не требуют заполнения транзитной структуры GCM_c_record.

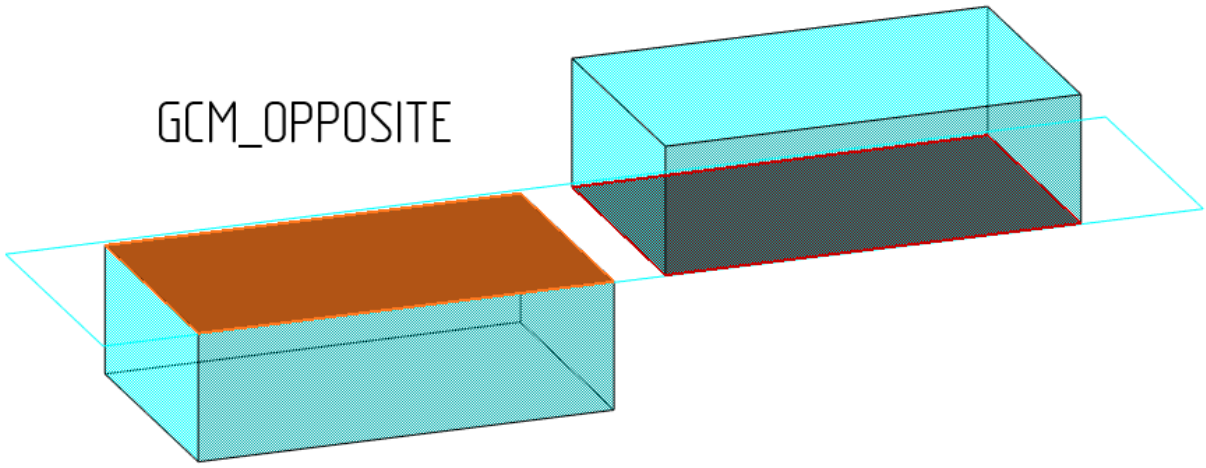
7.6.11 Опция выравнивания GCM_alignment

Тип ограничения (GCM_c_type) является основной характеристикой ограничения, достаточной что бы сказать удовлетворяет ли положение объектов данному типу ограничения или нет, однако участие в ограничении ориентируемых объектов, таких, как плоскость или прямая, допускают разные исполнения одного и того же типа GCM_c_type. Например, если взять две грани тела, лежащие в одной плоскости, мы можем сказать, что они удовлетворяют ограничению совпадения (GCM_COINCIDENT), однако это утверждение допускает два варианта совпадения, когда нормальные вектора граней смотрят в одну сторону или направлены в разные стороны. На рисунках 1, 2 показаны альтернативные варианты выравнивания двух тел с помощью ограничения GCM_COINCIDENT, связывающего пару цветных граней:



{width=»50%»}

Рис. 1



{width=»50%»}

Рис. 2

По умолчанию решатель выполняет совпадение, оставляя ориентацию геометрических объектов как можно ближе к исходному состоянию (см. *Начальное приближение*). Поэтому выбор из двух вариантов выполняется автоматически по принципу минимального поворота. Но если требуется, можно заказать определенную взаимную ориентацию векторов. Для этого в методах **GCM_AddBinConstraint**, GCM_AddDistance, GCM_AddConstraint предусмотрена опция выравнивания GCM_alignment, значения которой приведены в таблице.

GCM_alignment дополняет тип ограничения GCM_c_type условием, исключающим всякую неоднозначность при выборе решения для ограничения. Значения GCM_COORIENTED и GCM_OPPOSITE действуют на такие ограничения, которые подразумевают параллельность векторов Z своих аргументов. Например перпендикулярность плоскости и прямой подразумевают параллельность нормального вектора плоскости и прямой. Или пример совпадения двух плоскостей, который делает нормали граней сонаправленными (Рис. 1) либо противонаправленными (Рис. 2). Другие значения опции выравнивания действуют в более сложных случаях, совместно с такими ограничениями, как касание, линейные или угловые паттерны, где выбирать приходится из более чем 2 решений. Например касание двух тороидов по окружности предусматривает восемь решений, которые кроме сонаправленности векторов, различаются по стороне касания относительно осей Z, размещению одного тора снаружи или внутри другого.

Таблица 1 - Значения опции выравнивания GCM_alignment.

Имя	Значение
GCM_NO_ALI	Неопределенная ориентации. Значение GCM_NO_ALIGNMENT применяется в тех случаях, когда ограничение не чувствительно к ориентации геометрических объектов (например, совпадение двух точек) или ориентацию объектов требуется оставить свободной. Например, в угловых или линейных паттернах, если указать GCM_NO_ALIGNMENT, то ориентация «образца» никак не будет влиять на «экземпляр копирования».
GCM_CLOSES	Решение выбирается автоматически, как можно ближе к стартовому положению геометрических объектов.
GCM_COORIE	Задает сонаправленность векторов Z аргументов ограничения.
GCM_OPPOSI	Задает противонаправленность векторов Z аргументов ограничения.
GCM_ALIGNE GCM_ALIGNE GCM_ALIGNE GCM_ALIGNE	Варианты решения ограничений с сонаправленными векторами Z аргументов ограничения (подварианты GCM_COORIENTED).
GCM_REVERS GCM_REVERS GCM_REVERS GCM_REVERS	Варианты решения ограничений с противонаправленными векторами Z аргументов ограничения (подварианты GCM_OPPOSITE).
GCM_ALIGNE	Создает одинаковую ориентацию ЛСК объектов (сонаправленность по „Z“ и по „X“) для таких ограничений, как GCM_SYMMETRIC и GCM. Т.е. ограничения симметрия, угловые и линейные паттерны с данной опцией будут ориентировать «экземпляр копирования» также, как и «образец» при том, что их позиционирование будет подчиняться соответствующим законам копирования.
GCM_ROTATE	Действует для ограничения GCM_PATTERNED в случае углового паттерна GCM_ANGULAR_PATTERN. Означает, что ЛСК «образца копирования» и «экземпляра» полностью совмещаются поворотом вокруг оси паттерна.
GCM_ALIGN_	Применяется в качестве дополнительной опции угловых и линейных петтернов в функциях GCM_AddLinearPattern, GCM_AddAngularPattern.

Подводя итог, можно сказать, что GCM_alignment является дополнительным условием в тех ситуациях, когда GCM_c_type допускает не единственный вариант исполнения ограничения.

7.6.12 Кластеризация геометрической сцены, моделирование сборок

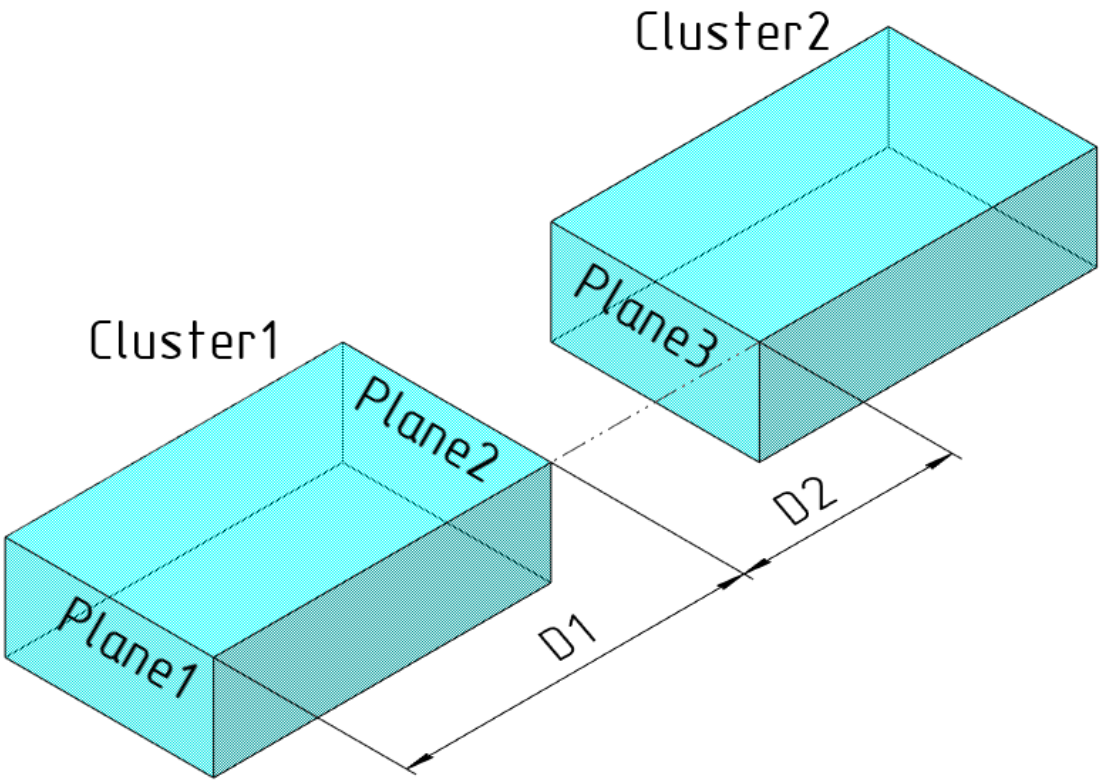
В основе геометрической модели системы ограничений лежит принцип иерархичности. Этот принцип находит свое выражение, в том что геометрические объекты могут группироваться в виде геометрически-жестких множеств, называемых кластерами (Кластер). Кластер отвечает такой абстракции геометрической модели САПР, как твердое тело или подсборка. Кластеры могут образовывать древо-подобные иерархии, зависимости в которых можно проследить с помощью функции

```
GCM_geom GCM_Parent(GCM_system gSys, GCM_geom subGeom)
```

которая вернет для объекта **subGeom**, зарегистрированного в системе ограничений **gSys**, кластер, которому принадлежит **subGeom**. Если функция вернет нулевой дескриптор **GCM_NULL**, это означает что запрашиваемый объект не содержится ни в одном кластере и был зарегистрирован в системе ограничений вызовом **GCM_AddGeom**. Геометриче-

ские объекты, подчиненные какому-либо кластеру, получаются с помощью другого вызова, **GCM_SubGeom** (подробности см. *Добавление и удаление геометрических объектов*).

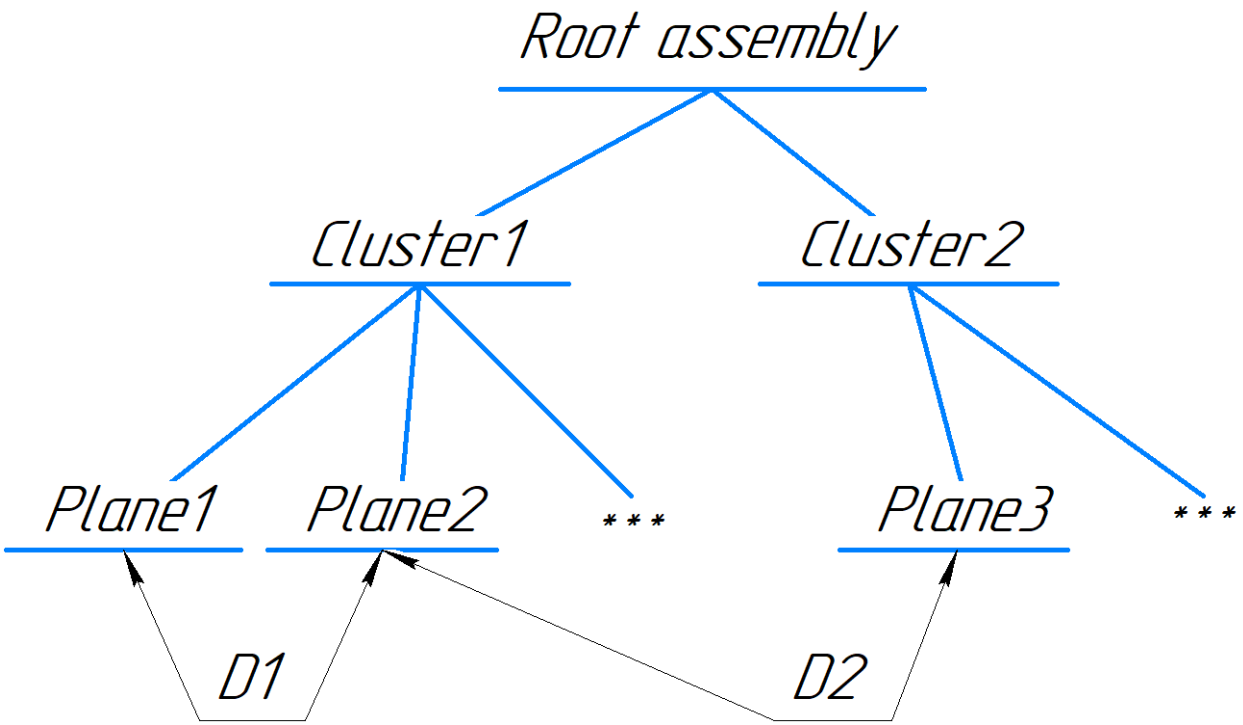
Принадлежность ограничений кластерам. Геометрические ограничения и размеры также, как и геометрические объекты имеют свое место в иерархии кластеров. Уровень, на котором будет вычисляться то или иное ограничение, не требуется указывать явно, он определяется в зависимости от того, каким кластерам принадлежат его аргументы. При добавлении ограничения решатель выбирает минимальную по размеру подсистему, включающую все его аргументы. Другими словами, уровень геометрического ограничения определяется минимальным поддеревом иерархии, включающим все его аргументы. На рис. 1 изображен пример сборки, где имеются два размера, которые будут вычисляться на разных уровнях иерархии. Сборка состоит из двух параллелепипедов, представленных кластерами Cluster1 и Cluster2, каждый из которых объединяет плоскости, ребра и вершины каждого из параллелепипедов.



{width=»50%»}

Рис. 1

Таким образом, вся иерархия состоит из трех подсистем ограничений, вычисляемых отдельно друг от друга снизу вверх (см. рис. 2). В сборке есть два размера: D1 – расстояние, заданное между плоскостями Plane1 и Plane2, принадлежащими одному и тому же кластеру Cluster1, D2 – расстояние, заданное между плоскостями Plane2 и Plane3, принадлежащими разным кластерам (Cluster1, Cluster2). В этому случае размер D1 в иерархии кластеров будет вычисляться в кластере Cluster1, а размер D2 будет вычисляться после D2 вместе со всеми ограничениями корневого уровня сборки. Так результаты вычисления D1 будут определять габариты только одного параллелепипеда, в то время, как размер D2 будет влиять на позиционирование двух параллелепипедов относительно друг друга.



{width=»50%»}

Рис. 2

Итак уровень ограничения определяется минимальным поддеревом кластеров, охватывающим все его аргументы. В частности, если аргументы ограничения лежат в одном и том же кластере, то ограничение будет принадлежать тому же кластеру. Если аргументы ограничения принадлежат разным кластерам, то выбирается уровень, образующий минимальное поддерево, включающее все аргументы ограничения.

Ниже приведен исходный код, который воспроизводит ситуацию, изображенную на рис. 1. Данный код приведен в качестве примера, показывающего, как в решателе формируется иерархия кластеров, представленная на рис. 2.

```
GCM_system gSys = GCM_CreateSystem();
/*
Add LCS of the first cluster, it represents the first parallelepiped.
Also its two planes will be added.
*/
GCM_geom cluster1 = GCM_AddGeom(gSys, GCM_SolidLCS(MbCartPoint3D::origin));
GCM_FixGeom(gSys, cluster1); // First cluster will have a fixed LCS.
const MbCartPoint3D org1(50.0, 0, 0); // Position of 'Plane1' in LCS of the first␣
↪cluster.
GCM_geom plane1 = GCM_SubGeom(gSys, cluster1, GCM_Plane(org1, MbVector3D::xAxis));
const MbCartPoint3D org2(-112.0, 0, 0); // Position of 'Plane2' in LCS of the first␣
↪cluster.
GCM_geom plane2 = GCM_SubGeom(gSys, cluster1, GCM_Plane(org2, -MbVector3D::xAxis));
/*
Set a distance dimension D1 between 'Plane1' and 'Plane2'.
(opposite sides of the parallelepiped, see figure 31.12.1)
*/
GCM_constraint D1 = GCM_AddDistance(gSys, plane1, plane2, -162.0);
/*
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
Add LCS of 'Cluster2' (the second box).
*/
const MbCartPoint3D pos2(-262.0, 0, 0);// Position of the second cluster.
GCM_geom cluster2 = GCM_AddGeom(gSys, GCM_SolidLCS(pos2));
const MbCartPoint3D org3(50.0, 0, 0); // Position of 'Plane3' in LCS of 'Cluster2'.
GCM_geom plane3 = GCM_SubGeom(gSys, cluster2, GCM_Plane(org3, MbVector3D::xAxis));
/*
Set distance dimension D2 between 'Plane2' and 'Plane3'.
(D2 defines a distance between the boxes, see figure 31.12.1)
*/
GCM_constraint D2 = GCM_AddDistance(gSys, plane2, plane3, 100.0);
/*
Dimension D2 can change the position of 'Cluster2' relative to 'Cluster1'.
*/
GCM_ChangeDrivingDimension(gSys, D2, 120.0);
GCM_Evaluate(gSys);
GCM_RemoveSystem(gSys);
```

Порядок вычисления кластеров. Обратим внимание, что схема кластеров, которая образовалась в результате комбинации вызовов **GCM_AddGeom** и **GCM_SubGeom** строго определяет порядок вычисления ограничений. Ограничения одного и того же кластера вычисляются одновременно, но не раньше, чем будут вычислены ограничения вложенных кластеров. Таким образом все кластеры вычисляются последовательно снизу вверх, от подчиненных к вышестоящим, включая корневой уровень.

7.6.13 Вычисление системы ограничений

Основную задачу трехмерного геометрического решателя (см. Задача удовлетворения геометрических ограничений) выполняет функция

```
GCM_result GCM_Evaluate(GCM_system gSys)
```

которая вычислит новое состояние геометрии и раздаст диагностических коды тем ограничениям, которые не удалось удовлетворить. Функция не тратит время на вычисления, если на момент её вызова все ограничения удовлетворены.

GCM_Evaluate вернет код GCM_RESULT_Ok, если найдено успешное решение для всех заданных ограничений. Любое другое возвращаемое значение говорит о том, что удовлетворить систему ограничений целиком не удалось. Таким образом, возвращаемое значение **GCM_Evaluate**, показывает либо успешный результат, либо диагностический код одного из не решенных ограничений, либо иное код, показывающий причину неудачного решения. Расшифровка диагностических кодов приведена в параграфе *Диагностические коды решения*.

После вызова **GCM_Evaluate** каждому геометрическому ограничению присваивается диагностический статус, который можно узнать с помощью вызова

```
GCM_result GCM_EvaluationResult(GCM_system gSys, GCM_constraint cItem)
```

который для каждого ограничения выдаст одно из возможных значений:

- GCM_RESULT_None означает, что статус ограничения не определен, т.е. ограничение ни разу не вычислялось;
- GCM_RESULT_Satisfied означает, что ограничение решено;

- GCM_RESULT_Overconstrained, означает что ограничение участвует в группе противоречащих друг другу ограничений, которые не могут быть удовлетворены одновременно. Обычно удаление одного из таких ограничений исправляет ситуацию;
- GCM_RESULT_Not_Satisfied, означает, что ограничение не удовлетворено (предусмотрено для случаев, когда не удается диагностировать причину неудачного решения);
- GCM_RESULT_Unsolvable, означает, что ограничение не может быть решено. Возможные причины: противоречие ограничений друг другу, отсутствие необходимых ограничений, позиционирующих объекты (например, в кулачковом механизме), плохое начальное приближение;
- GCM_RESULT_IncompatibleArguments, означает, что ограничение не применимо для объектов с данными типами геометрии. Например, совпадение плоскости и цилиндра не возможно выполнить; Подробнее о всех значениях GCM_result см. в параграфе *Диагностические коды решения*.

В результате вызова **GCM_Evaluate** меняется состояние геометрических объектов. Чтобы приложение могло применить результаты расчетов, необходимо запросить с помощью функции **GCM_Placement** значения ЛСК, позиционирующие объекты.

Функция

```
MbPlacement3D GCM_Placement(GCM_system gSys, GCM_geom g)
```

возвращает положение (ЛСК) геометрического объекта **g** в соответствии с текущим состоянием геометрической модели. В параграфе *Представление геометрических объектов* уже говорилось, что свою ЛСК имеет каждый геометрический объект, даже если это примитив в роде точки. ЛСК удобна тем, что с ее помощью можно выразить положение объектов всех типов, включая не только твердые тела, но и простые объекты, такие как точка, прямая и плоскость. Также с помощью вызова **GCM_Origin** можно запросить точку начало только координат ЛСК. Это удобно, когда мы имеем дело с неориентированными объектами, такими как точка и сфера.

Другая функция, применительно к радиусным объектам, таким как окружность, сфера или цилиндр вернет значение радиуса:

```
double GCM_Radius(GCM_system gSys, GCM_geom g)
```

Также с помощью функций:

```
double GCM_RadiusA(GCM_system gSys, GCM_geom g)
```

```
double GCM_RadiusB(GCM_system gSys, GCM_geom g)
```

можно узнать большой и малый радиусы тороида и конуса.

Приложение имеет возможность не только запрашивать текущее состояние геометрии, но и менять его, если по каким-либо причинам произошли изменения на стороне CAD-модели, требующие синхронизировать геометрический решатель. Для этого предусмотрен вызов

```
void GCM_SetPlacement(GCM_system gSys, GCM_geom g, const MbPlacement3D & place)
```

который меняет текущее значение ЛСК объекта.

Исходные данные вызова:

- **gSys** – система геометрических ограничений;

- **g** – дескриптор геометрического объекта;
- **place** – новое положение (ЛСК), заданной в мировой системе координат.

Заметим, эта функция только придает объекту новое состояние без переоценки системы ограничений. Вызов **GCM_Evaluate** может поменять заданное состояние, если имеются не удовлетворенные ограничения.

В любой момент времени у каждого ограничения можно спросить удовлетворено оно или нет с помощью функции

```
bool GCM_IsSatisfied(GCM_system gSys, GCM_constraint cItem)
```

которая вернет true, если ограничение **cItem**, удовлетворено в текущем состоянии геометрии. Результат вызова **GCM_IsSatisfied** зависит только от текущего состояния геометрических аргументов ограничения и не связано с его диагностическим статусом, возвращаемым по запросу **GCM_EvaluationResult**.

7.6.14 Диагностические коды решения

В таблице ниже перечислены коды, сообщающие результат вычисления или выполнения функции API компонента GCM. Данные значения являются результатом таких вызовов, как **GCM_Evaluate**, **GCM_EvaluationResult**, **GCM_SolveReposition**.

Таблица 1 - Значения перечислительного типа GCM_result

Результирующие коды выполнения функций API решателя	
GCM_RESULT_None	**Нет результата.** Для вызова **GCM_EvaluationResult** это означает, что ограничение ни разу не вычислялось.
GCM_RESULT_Ok	**Успешно выполнено.** Результат успешного выполнения операции или успешного вычисления ограничений.
GCM_RESULT_ItsNotDrivingDimension	**Размер не управляющий.** Неудачная попытка вызова **GCM_ChangeDrivingDimension**, для ограничения, не являющегося управляющим размером.
GCM_RESULT_Unregistered	**Дескриптор объекта или ограничения не действителен.** Ситуация: Обращение к вызову API с дескриптором, не принадлежащем системе ограничений.
GCM_RESULT_Aborted	Вычисления прерваны по запросу пользователя.
GCM_RESULT_InternalError **GCM_RESULT_Error**	**Программная ошибка** (не математическая). Предусмотрена для ситуаций, связанных с некорректным завершением. операции или некорректными данными.
Результаты вычисления системы ограничений	
GCM_RESULT_Satisfied	**Ограничение или система ограничений удовлетворены**.

продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Результирующие коды выполнения функций API решателя	
GCM_RESULT_Overconstrained	**Ограничение переопределяет модель.** Ошибка выявляется в ситуациях, когда имеются противоречащие ограничения (несовместная система ограничений) в переопределенной системе ограничений.
GCM_RESULT_Unsolvable	**Не решаемые ограничения.** Ошибка выявляется в различных случаях, когда решение не возможно. Обычно это связано с тем, что есть набор ограничений, которые не могут быть удовлетворены одновременно (несовместная система ограничений), или когда размер вышел из области допустимых значений.
GCM_RESULT_Not_Satisfied	**Ограничение не удовлетворено.** Код ошибки возвращается в ситуациях, когда не удалось решить ограничения. Возможные причины неудачи: - Размер вышел за пределы значений, допускающих существование решения; - Положения фиксированных или замороженных объектов вышли за область, допускающую существования решения. - В редких случаях – плохое начальное приближение. Удовлетворение ограничений требует значительные изменение параметрических координат; - Сильные взаимные влияния. Например, заданы такие условия, что изменение угла на один градус приводит к смещению объекта на значительные расстояние. - В редких случаях – численная неустойчивость, обусловленная большим разбросом порядка вычисляемых координат. Например в геометрической модели одновременно встречаются микроскопические объекты (1e-3) и крупные объекты (1e+6). - В общем случае – иные ситуации, когда решение не существует или не может быть найдено.
GCM_RESULT_MatedFixation	**Задано геометрическое ограничение между фиксированными или замороженными объектами**.
GCM_RESULT_InvalidArguments	**Аргументы ограничения не определены.** Ситуация некорректно определенного ограничения, когда его аргументы пустые (дескриптор аргумента = GCM_NULL).

продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Результирующие коды выполнения функций API решателя	
GCM_RESULT_IncompatibleArguments	**Несовместимые аргументы для данного типа ограничения.** Ситуация некорректно определенного ограничения, когда его аргументы не совместимы друг другу или данное сочетание аргументов не поддерживается. Например, попытка задать совпадение для плоскости и цилиндра.
GCM_RESULT_InappropriateArgument	**Тип аргумента не подходит для данного ограничения.** Ситуация некорректно определенного ограничения, когда один из его аргументов не подходит к данному типу ограничения. Например, попытка задать параллельность для точек или сфер очевидно лишена смысла.
GCM_RESULT_Duplicated	**Ограничение дублирует имеющееся.** Код сообщает, что в системе ограничений находятся два одинаковых ограничения. В принципе данная ситуация не мешает вычислению ограничений.
GCM_RESULT_DraggingFailed	
GCM_RESULT_InappropriateAlignment	
GCM_RESULT_InconsistentAngleType	
GCM_RESULT_InconsistentAlignment	
mtResCode_UnsupportedTangencyChoice	
mtResCode_IsNoPossibleForCircTanChoice	
GCM_RESULT_InconsistentPlanarAngle	
Диагностические ситуации для ограничения GCM_DEPENDED	
GCM_RESULT_DependentConstraintUnsolved	
GCM_RESULT_CyclicDependence	
GCM_RESULT_MultiDependedGeom	
GCM_RESULT_OverconstrainingDependedGe	
mtResCode_InvalidDependenceForFixGeom	
Диагностические коды для механических передач	
mtResCode_CoaxialMtGearTransmissionIsNot	
mtResCode_NoSeparatedSolutionForCamGea	
mtResCode_CyclicDependenceForTwoOrMore	
mtResCode_InconsistentFollowerAxis	

7.6.15 Управляющие размеры

Управляющие размеры – это размерные ограничения, которые задают расстояние или угол между двумя геометрическими объектами и тем самым наделяют геометрическую сцену числовыми параметрами, управляющими положением ее объектов. Варьируя параметры размеров, можно управлять состоянием модели или получать экземпляры параметрической модели в разном геометрическом исполнении.

Управляющие размеры можно создать с помощью вызовов GCM_AddDistance,

GCM_AddAngle и GCM_FixRadius. Основное предназначение управляющих размеров – это управлять положением геометрических объектов с помощью числового параметра длины, радиуса или угла. Заметим, что ограничения паттернов, созданных методом **GCM_AddGeomToPattern**(), также относятся к управляющим размерам, поскольку элементы паттерна, характеризуются числовыми параметрами, задающими их линейные или угловые позиции.

Функция

```
GCM_result GCM_ChangeDrivingDimension(GCM_system gSys, GCM_constraint dItem
                                      , double dVal)
```

позволяет менять значение любого размера **dItem** и тем самым управлять состояние параметрической модели.

Входные параметры вызова:

- **gSys** – система ограничений параметрической модели;
- **dItem** – дескриптор размерного ограничения;
- **dVal** – новое значение размера, заданное в единицах длины для линейных размеров или в радианах для угловых размеров.

Возвращаемые значения:

- GCM_RESULT_Ok, если операция выполнена успешно;
- GCM_RESULT_ItsNotDrivingDimension означает, что присланное ограничение не является управляющим размером или управляющим параметром, например, если ограничение – вариационный размер;
- GCM_RESULT_None или GCM_RESULT_Unregistered возвращаются, если присланные дескрипторы **gSys** или **dItem** не действительны.

Следует учитывать, что данная функция не выполняет вычислений, а только подготавливает изменение управляющего размера. Чтобы изменение вступило в силу, необходимо вызвать **GCM_Evaluate**. Если требуется изменить одновременно два и более управляющих размеров, следует сначала сделать серию вызовов **GCM_ChangeDrivingDimension** на каждый из них, затем один раз вызвать **GCM_Evaluate**.

7.6.16 Журналирование вызовов API решателя GCM

C3D Solver дает возможность записать историю вызовов API GCM для дальнейшего воспроизведения с целью отладки и тестирования. Например, чтобы получить консультацию или решить проблемную ситуацию, возникшую в приложении при работе с C3D Solver, достаточно временно включить режим записи журнала, выполнить интересующий сценарий действий. Записанный журнал пересылается в техническую поддержку C3D Labs с описанием ситуации. После исправления ошибки журнал добавляется в базу тестовых примеров.

Журналирование вызовов API трехмерного геометрического решателя осуществляется аналогично журналированию 2D-решателя (*Журналирование вызовов API*). При включении режима журналирования с помощью вызова

```
bool GCM_SetJournal(GCM_system gSys, const char * fName)
```

решатель осуществляет запись истории вызовов API применительно к системе ограничения **gSys**. В дальнейшем записанный журнал может быть использован разработчикам C3D Solver для анализа или пополнения базы тестов.

Внимание! Файл журнала будет готов только после завершения сеанса работы с системой ограничений, а именно сразу после вызова **GCE_RemoveSystem**. Для корректной записи журнала вызов **GCM_SetJournal** должен следовать сразу после **GCM_CreateSystem**. Например:

```
GCM_system gSys = GCM_CreateSystem();
#ifdef C3D_DEBUG
GCM_SetJournal( gSys, "d:\\Logs\\gcm\_sample2.jrn");
#endif // C3D_DEBUG
/*
Vertexes of the triangle A-B-C.
*/
GCM_geom A = GCM_AddPoint(gSys, MbCartPoint3D(138.0, -38.0, -31.0));
GCM_geom B = GCM_AddPoint(gSys, MbCartPoint3D(67.0, -44.0, 51.0));
GCM_geom C = GCM_AddPoint(gSys, MbCartPoint3D(20,30,10));
/*
Set distance constraints of the sides of the triangle.
*/
GCM_constraint d1 = GCM_AddDistance(gSys, A, B, 100.0, GCM_CLOSEST);
GCM_constraint d2 = GCM_AddDistance(gSys, B, C, 100.0, GCM_CLOSEST);
GCM_constraint d3 = GCM_AddDistance(gSys, C, A, 100.0, GCM_CLOSEST);
GCM_Evaluate(gSys);
/*
Finalize the constraint system. The journal file is written.
*/
GCM_RemoveSystem(gSys);
```

Внимание! Журналирование нужно только для отладочных целей, поэтому рекомендуется включать её только в режиме отладки. Не рекомендуется оставлять журналирование включенным для продуктовой (release) или рабочей версии приложения, т.к. это может приводить к наоптимальному расходованию ресурсов времени и памяти.

ГЛАВА 8

C3D Converter

Модуль C3D Converter обеспечивает конвертацию, чтение и запись 3D-моделей в различных обменных форматах и отвечает за обмен данными с другими системами. Первая ключевая задача — импорт: чтение файла, буфера в памяти или потока сторонней геометрической модели в одном из поддерживаемых форматов и построение на основе этих данных геометрической модели C3D. Вторая задача — экспорт: представление геометрической модели C3D в одном из обменных форматов и запись этого представления в указанный файл, буфер в памяти или поток.

Вместе со сведениями о геометрической форме моделируемого объекта через конвертер может передаваться максимальный набор данных:

- Информация о форме, которая может быть описана с помощью тел, поверхностей, проволочных каркасов и групп точек;
- Информация о структуре моделей: выделены составляющие её компоненты, которые могут быть многократно использованы в модели;
- Информация об изделиях: идентификатор, комментарии, сведения об авторах и пр;
- Атрибуты: визуальные свойства, элементарные атрибуты;
- Элементы аннотации: размеры, технические требования, обозначения.

Геометрическое ядро C3D поддерживает текстовые форматы ACIS, IGES, STEP, X_T, STL, VRML, OBJ, SAT и бинарные форматы X_B, STL, JT, 3MF, SAB. Форматы ACIS, IGES, STEP, X_T, SAT, X_B и SAB передают информацию о геометрической форме моделируемого объекта с помощью граничного представления (Boundary representation), а форматы STL, VRML, OBJ, JT и 3MF — в полигональном представлении (Polygonal representation). Формат JT может содержать модель как в граничном, так и в полигональном виде.

Конвертер 3D-моделей имеет гибкий набор настроек: можно управлять автоматической сшивкой поверхностей в оболочки, ограничивать передачу объектов определённых типов или включать специальные режимы для анализа и отладки. Для приложений, где файловый ввод-вывод может стать узким местом, предусмотрен обмен данными через буфер в памяти и многопоточный режим работы конвертеров STEP и Parasolid, что позволяет существенно повысить производительность.

Для активации лицензии на модуль необходимо вызвать функцию EnableMathModules() с соответствующими параметрами.

8.1 Принципы работы конвертера

Работа конвертера предполагает использование интерфейсов, которые можно разделить на две группы. В первую входят интерфейсы, которые имеют реализацию на стороне C3D, во вторую — те интерфейсы, которые должны быть реализованы пользователем. Кроме того, геометрическое ядро C3D предоставляет стандартную реализацию всех интерфейсов, необходимых для обмена данными.

Интерфейсы соответствуют следующим понятиям:

- Конвертер (интерфейс IConvertor3D) — специальный объект, который используется для передачи данных. Реализован на стороне C3D.
- Свойства конвертера (интерфейс IConvertorProperty3D) — служат для передачи настроек обмена в конвертер. Интерфейс имеет стандартную реализацию.
- Документ (интерфейс ItModelDocument) — объект, которому соответствует передаваемая модель в целом. Интерфейс имеет стандартную реализацию.
- Деталь (интерфейс ItModelPart) и сборка (интерфейс ItModelAssembly) — части модели, которые могут быть представлены как детали или как сборочные единицы.
- Вставка (интерфейс ItModelInstance) — объект, отвечающий за позиционирование и/или повторное использование компонента в модели.

Обобщающим понятием детали и сборки является **компонент**. Предполагается, что **компонент** может быть минимальной единицей, которой может соответствовать отдельный файл, с которым работает конечное приложение. Использование компонентов и вставок позволяет сформировать дерево модели, в котором компоненты могут быть многократно использованы.

Внутри конвертера осуществляется преобразование внутренней модели, исходя из специфики формата. Например, у детали может быть зеркальное исполнение, и этому соответствует левая система координат компонента, а в формате STEP левые системы координат не предусмотрены. Также в формате STEP визуальные свойства нельзя назначить на вставку детали и для перекрашенных деталей создаются дубли. Основных критериев преобразования два — левая система координат и визуальные свойства (в каких-то случаях допустимо назначать визуальные свойства на вставке, например цвет, а в каких-то — нет).

Схема преобразования между моделями данных базируется на двух сущностях:

- типы элементов (*MbAssembly* или формосодержащие *MbSolid*, *MbMesh*, *MbWireFrame*, *MbSpaceInstance*, *MbPlaneInstance*);
- атрибуты (в первую очередь *MbProductInfo*).

8.2 Порядок работы с конвертером

Конвертеры предоставляют средства для экспорта и импорта моделей MbModel, MbItem и пользовательских реализаций документов ItModelDocument.

Группа функций, предназначенных для обмена данными через файлы, размещена в файле *conv_model_exchange.h* и объявлена в пространстве имён c3d:

Функция

```
MbeConvResType c3d::ImportFromFile(       MbModel &               model,
                                  const path_string &           fileName,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

позволяет прочитать файл обменного формата в модель.

Функция

```
MbeConvResType c3d::ImportFromFile(       c3d::ItemSPtr &         item,
                                  const path_string &           filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

позволяет прочитать файл обменного формата в элемент.

Функция

```
MbeConvResType c3d::ImportFromFile(       ItModelDocument &       mDoc,
                                  const path_string &           filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

позволяет прочитать файл обменного формата в модель.

Функция

```
MbeConvResType c3d::ExportIntoFile(       MbModel &               model,
                                  const path_string &           filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

позволяет записать модель в файл обменного формата.

Функция

```
MbeConvResType c3d::ExportIntoFile(       MbItem &                item,
                                  const path_string &           filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

позволяет записать элемент в файл обменного формата.

Функция

```
MbeConvResType c3d::ExportIntoFile(       ItModelDocument &       mDoc,
                                  const path_string &           filePath,
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr␣
↪);
```

позволяет записать модельный документ в файл обменного формата.

Аргументами функций являются:

- **model** – экспортируемая или импортируемая геометрическая модель;
- **item** – элемент модели;
- **mDoc** – модельный документ;
- **fileName (filePath)** – полный путь к файлу;
- *prop* – реализация интерфейса свойств конвертера;
- *indicator* – индикатор прогресса.

При импорте обменный формат определяется на основе расширения файла.

Группа функций, реализующих передачу моделей в обменные форматы через оперативную память:

```
MbeConvResType c3d::ImportFromBuffer(       MbModel &              model,
                                      const C3DExchangeBuffer &    buffer,
                                            MbeModelExchangeFormat modelFormat,
                                            IConvertorProperty3D * prop = nullptr,
                                            IProgressIndicator *   indicator =␣
↪nullptr );
```

```
MbeConvResType c3d::ImportFromBuffer(       c3d::ItemSPtr &        item,
                                      const C3DExchangeBuffer &    buffer,
                                            MbeModelExchangeFormat modelFormat,
                                            IConvertorProperty3D * prop = nullptr,
                                            IProgressIndicator *   indicator =␣
↪nullptr );
```

```
MbeConvResType c3d::ImportFromBuffer(       ItModelDocument &      mDoc,
                                      const C3DExchangeBuffer &    buffer,
                                            MbeModelExchangeFormat modelFormat,
                                            IConvertorProperty3D * prop = nullptr,
                                            IProgressIndicator *   indicator =␣
↪nullptr );
```

```
MbeConvResType c3d::ExportIntoBuffer( MbModel &              model,
                                      MbeModelExchangeFormat modelFormat,
                                      C3DExchangeBuffer &    buffer,
                                      IConvertorProperty3D * prop = nullptr,
                                      IProgressIndicator *   indicator = nullptr );
```

```
MbeConvResType c3d::ExportIntoBuffer( MbItem &               item,
                                      MbeModelExchangeFormat modelFormat,
                                      C3DExchangeBuffer &    buffer,
                                      IConvertorProperty3D * prop = nullptr,
                                      IProgressIndicator *   indicator = nullptr );
```

```
MbeConvResType c3d::ExportIntoBuffer( ItModelDocument &      item,
                                      MbeModelExchangeFormat modelFormat,
                                      C3DExchangeBuffer &    buffer,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
                                        IConvertorProperty3D * prop = nullptr,
                                        IProgressIndicator *   indicator = nullptr );
```

Аргументами функций являются:

- **model** – экспортируемая или импортируемая геометрическая модель;
- **item** – элемент модели;
- **mDoc** – модельный документ;
- **modelFormat** – формат обменного файла;
- **buffer** – буфер;
- **data** – адрес буфера в памяти;
- *prop* – настройки конвертера;
- *indicator* – индикатор прогресса.

Функции ImportFromFile(), ImportFromBuffer() выполняют импорт геометрических моделей:

– форматов STL, OBJ;

– формата SAT/SAB до 22.0 версии включительно;

– формата IGES версии 5.3;

– формата JT версий 8.0 — 11.x;

– формата Parasolid до версии 25.0 включительно;

– формата STEP с поддерживаемыми прикладными протоколами 203, 214;

– формата VRML версии 2.0.

Функции ExportIntoFile(), ExportInfoBuffer() выполняют экспорт геометрических моделей:

– форматов STL, OBJ;

– формата SAT/SAB до 22.0 версии включительно;

– формата IGES версии 5.3;

– формата JT версии 9.5;

– формата Parasolid до версии 10.0 включительно;

– формата STEP с поддерживаемыми прикладными протоколами 203, 214;

– формата VRML версии 2.0.

Результат FullFilePath() аргумента *prop* используется для определения формата обменного файла только в том случае, если значение modelFormat равно *mxf_autodetect*.

Ответственность за удаление буфера, который создаёт и наполняет функция ExportIntoBuffer(), лежит на пользователе. Для освобождения памяти рекомендуется использовать оператор *delete*[].

Другим способом импорта и экспорта модели является обращение к методам конвертера IConverter3D.

Для начала работы с конвертером необходимо получить его экземпляр. Определённая в глобальной области видимости функция IConverter3D * GetConvertor3D() создает конвертер, который может выполнять импорт и экспорт моделей в любом из поддерживаемых

форматов. Функция не имеет параметров. В случае успеха функция возвращает конвертер, в противном случае возвращает нулевой указатель.

Функция объявлена в файле *conv_model_exchange.h*.

После создания конвертера обмен данными осуществляется путем вызова методов конвертера. Методы конвертера (устаревшие, рекомендуется использовать ImportFromFile(), ImportFromBuffer(), ExportInfoFile(), ExportInfoBuffer()):

```
MbeConvResType SATRead( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        std::iostream *        stream,
                        IProgressIndicator *   indicator = 0 ),
```

```
MbeConvResType SATWrite( IConvertorProperty3D & property,
                         ItModelDocument &      document,
                         std::iostream *        stream,
                         IProgressIndicator *   indicator = 0 ),
```

```
MbeConvResType SATRead( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0,
                        MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType SATWrite( IConvertorProperty3D & property,
                         ItModelDocument &      document,
                         IProgressIndicator *   indicator = 0,
                         MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType IGSRead( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0,
                        MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType IGSWrite( IConvertorProperty3D & property,
                         ItModelDocument &      document,
                         IProgressIndicator *   indicator = 0,
                         MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType JTRead( IConvertorProperty3D & property,
                       ItModelDocument &      document,
                       IProgressIndicator *   indicator = 0,
                       MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType JTWrite( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0,
                        MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType XTRead( IConvertorProperty3D & property,
                       ItModelDocument &      document,
                       IProgressIndicator *   indicator = 0,
                       MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType XTWrite( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0,
                        MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType STEPRead( IConvertorProperty3D & property,
                         ItModelDocument &      document,
                         IProgressIndicator *   indicator = 0,
                         MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType STEPWrite( IConvertorProperty3D & property,
                          ItModelDocument &      document,
                          IProgressIndicator *   indicator = 0,
                          MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType STLRead( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0,
                        MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType STLWrite( IConvertorProperty3D & property,
                         ItModelDocument &      document,
                         IProgressIndicator *   indicator = 0,
                         MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType VRMLRead( IConvertorProperty3D & property,
                         ItModelDocument &      document,
                         IProgressIndicator *   indicator = 0,
                         MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType VRMLWrite( IConvertorProperty3D & property,
                          ItModelDocument &      document,
                          IProgressIndicator *   indicator = 0,
                          MbRefItem *            qeuryStitch = 0 );
```

```
MbeConvResType OBJRead( IConvertorProperty3D & property,
                        ItModelDocument &      document,
                        IProgressIndicator *   indicator = 0)
```

выполняют импорт и экспорт модели в форматах SAT, IGES, JT, X_T, STEP, STL, VRML, OBJ (только импорт) соответственно.

Входными параметрами методов являются:

- **property** – свойства конвертера;
- **stream** – поток для получения или записи модели обменного формата;
- **document** – документ, передающий геометрическую модель и другую информацию;
- *indicator* – индикатор хода процесса чтения или записи;
- *qeuryStitch* – запросчик сшивки поверхностей или единиц измерения при импорте (устаревший параметр, не рекомендуется использовать, для управления сшивкой следует пользоваться свойствами конвертера).

В случае успеха методы возвращают *cnv_Success* или *cnv_PartialSuccess*, в противном случае возвращается код результата операции из перечисления *MbResultType*.

По окончании работы полученный экземпляр конвертера необходимо удалить.

Конвертер наследует интерфейс класса IConverter3D, который объявлен в файле *conv_model_exchange.h*.

В классе IConverter3D определены следующие методы:

- Метод

```
virtual void SetConfigurationSelector( SPtr<IConfigurationSelector>␣
↪configuration_selector )
```

  устанавливает обработчик для выбора конфигураций. Если обработчик установлен и в импортируемом файле есть более одной конфигурации (исполнения), то в процессе чтения будет вызван установленный обработчик для выбора необходимой конфигурации (см. *Импорт конфигураций модели*).

  Входной параметр **configuration_selector** – указатель на устанавливаемый обработчик.

- Метод

```
virtual void SetMetadataTransferCallback( SPtr<IConverterMetadataReceiver>␣
↪metadataTransferCallback )
```

  устанавливает передачу метаданных (см. *Импорт метаданных строительных форматов*).

- Метод

```
virtual MbeConvResType LoadForeignReader( const c3d::path_string &           ␣
↪pluginName,
                                          const c3d::optionNameValuePairs_t &␣
↪pluginSpecificSettings )
```

  загружает плагин получения данных для построения модели. Используется для явного задания используемого плагина для импорта проприетарных форматов, расположенного по указанному пути. Это альтернативный методу c3d::ImportFromFile() метод использования плагинов, когда путь до плагинов явно не задается, предполагая, что плагины лежат в одном каталоге с ядром.

  Входные параметры:

  - **pluginName** – имя подключаемого файла;
  - **pluginSpecificSettings** – специфические для плагина настройки (описание специфичных для плагина настроек следует получить у поставщика компонента).

- Метод

```
virtual MbeConvResType ReleaseForeignReader()
```

  отключает загруженный плагин получения данных для построения модели.

- Метод

```
virtual MbeConvResType ImportForeign( const c3d::path_string &     path,
                                      ItModelDocument &      idoc,
                                      IConvertorProperty3D* prop = 0,
                                      IProgressIndicator*   indicator = 0 )
```

читает файл с использованием плагина, заданного в LoadForeignReader().

Входные параметры:

- **path** – путь к файлу, который нужно прочитать;
- **idoc** – реализация интерфейса документа;
- **prop** – реализация интерфейса свойств конвертера;
- **indicator** – индикатор хода процесса.

8.3 Особенности обменных форматов

Основные особенности обменных форматов показаны в таблице ниже:

Формат	C3D	3MF	OBJ	STL	VRML	JT	X_T/X_B	SAT	IGES	STEP
Основное назначение	Формат экосистемы C3D	3D-печать	Визуализация	3D-печать	Визуализация	Обмен данными в Parasolid CAD-PLM	Формат экосистемы Parasolid	Формат экосистемы ACIS	Обмен данными, хранение	Хранение данных, машиностроение
Единица длины	мм	Задаются в файле	-	-	-	На компонент	м	На весь файл	На весь файл	На компонент
Особенности физического устройства	Бинарный	Набор XML-файлов	-	Есть	Текстовый	Бинарный	bin+txt	bin+txt	Текстовый	Текстовый
Ссылки на другие файлы	Нет	Нет	Нет	Нет	Нет	Есть	Нет	Нет	Есть	Есть
Особенности сборочных моделей	-	-	-	Одна сетка на модель	Простр-ые трансформации	-	-	Развертывается в лин-й список	Разные сп. групп-ки объектов	только правые СК
Визуальные свойства	Есть	Есть	Есть	Нет	Есть	Есть	Есть	Есть	Есть	Есть
GD&T	Есть	Нет	Нет	Нет	Нет	Есть	Нет	Нет	Нет	Есть
PMI	Есть	Нет	Нет	Нет	Нет	Есть, тесселяция	Нет	Нет	Есть в 2D	Есть
Полигональное представление без топологии	Есть	Есть	Есть	Есть	Есть	Есть	Нет	Нет	Нет	Есть
Полигональное представление с топологией	Есть	Есть	-	Нет	Нет	Есть	Есть	Нет	Нет	Есть
Точки	Есть	Нет	Нет	Нет	Нет	Есть	Нет	Нет	Есть	Есть
Кривые	Есть	Нет	-	Нет	Нет	Есть	Есть	Есть	Есть	Есть
Поверхности без топологии	Есть	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Есть	Есть
Граничное представление формы	Есть	Нет	Нет	Нет	Нет	Есть	Есть	Есть	Есть	Есть

8.4 Свойство конвертера IConvertorProperty3D

Свойство конвертера IConvertorProperty3D является одним из параметров методов обмена данными. Поскольку один и тот же интерфейс IConvertorProperty3D используется для управления работой конвертеров всех форматов, то он содержит как методы, общие для

всех форматов, так и специфичные для того или иного формата. Кроме того, отличается набор данных, которые управляют работой экспорта и импорта.

Методы, которые вызываются при экспорте в любой формат, отвечают за выдачу ряда сведений о документе в целом, о системе координат, относительно которой геометрическая модель должна быть ориентирована, а также запрашивают, следует ли принудительно трансформировать объекты обменного формата так, чтобы сохранялась их форма и взаимное расположение, но использовались только правые системы координат и масштабный множитель единиц длины, на который при экспорте умножаются все линейные размеры:

```
bool GetPropertyString( MbeConverterStrings nString, std::string & propertyString );

MbPlacement3D GetOriginLocation();

bool ReplaceLocationsToRight() (для форматов с левыми системами координат);

double LengthUnitsFactor().
```

Метод

```
void SetIoPermission( MbeIOPermiss nPermission, bool setF )
```

позволяет задавать фильтр передаваемых объектов.

Метод, который вызывается при импорте из любого формата и служит для запроса масштабного коэффициента единиц длины в пользовательской модели, на который умножаются все линейные размеры:

```
double AppLengthUnitsFactor();
```

Метод, который вызывается при импорте из формата STEP и служит для управления импортом атрибутов вставок:

```
virtual bool ImportInstanceAttributes() const { return true; }
```

Метод, который вызывается при импорте из форматов SAT, X_T, X_B, STEP и IGES, а также при импорте проприетарных форматов через плагины, выдает запрет или разрешение на автоматическую сшивку отдельных поверхностей в тела и указывает точность сшивки:

```
bool EnableAutoStitch( double& stitchPrecision );
```

При импорте из IGES и SAT вызывается метод, который разрешает или запрещает конвертеру обратиться к запросчику:

```
bool CanShowMessages();
```

При экспорте в формат STL вызывается метод, определяющий, будут ли данные предоставляться в текстовом или двоичном виде:

```
bool IsFileAscii();
```

При экспорте в STEP вызывается метод, имеющий реализацию по умолчанию и служащий для задания представления текста в элементах аннотации и прикладного протокола STEP, который будет использоваться:

```
eTextForm GetAnnotationTextRepresentation();
```

При экспорте в IGES вызывается метод, который определяет будет ли экспортироваться информация о топологии:

```
bool IsOutOnlySurfaces();
```

При экспорте в полигональные форматы (STL и VRML) вызываются методы, имеющие реализацию по умолчанию и управляющие параметрами расчёта триангуляции:

```
MbStepData TesselationParameters();
```

```
bool DualSeams();
```

```
void DualSeams( bool dSeams );
```

При экспорте в STEP и SAT вызывается метод, имеющий реализацию по умолчанию и возвращающий версию формата. Для задания версии рекомендуется использовать предопределенные значения.

```
long int GetFormatVersion();
```

При экспорте в формат ACIS вызывается метод, имеющий реализацию по умолчанию и управляющий удалением швов из циклов:

```
virtual bool ExportWithSeamRemoval() const { return false; }
```

Метод

```
virtual bool ImportMeshesWithTopology()
```

импортирует полигональные объекты как объекты с топологической информацией.

Метод

```
virtual bool RepairInconsistentMesh( MbRepairInconsistentMeshParams & )
```

включает постобработку сеток.

Геометрическое ядро C3D предоставляет стандартную реализацию интерфейса IConvertorProperty3D.

Класс ConvConvertorProperty3D представляет собой стандартную реализацию свойства конвертера. Свойство наследует интерфейс класса IConvertorProperty3D.

В своих полях, к которым открыт доступ, класс ConvConvertorProperty3D хранит следующую информацию:

- **fileName** – путь к файлу обменного формата (по умолчанию пустая строка);
- **fileASCII** – признак экспорта в текстовый файл STL (по умолчанию *true*);
- **formatVersion** – версия формата при экспорте (по умолчанию EXPORT_DEFAULT);
- **exportIGESTopology** – флаг экспорта топологии в формат IGES (по умолчанию *true*);
- **ioPermission** – фильтр объектов по типам (обеспечивает разрешение экспортировать и импортировать объекты любого типа);

- **propertyStrings** – особые сведения о документе (пустой контейнер);
- **annotTextReprSTEP** – представление текста аннотационных элементов (по умолчанию exf_TextOnly);
- **originLocation** – локальная система координат модели;
- **enableAutostitch** – флаг включения автосшивки при импорте модели (по умолчанию *true*);
- **autostitchPrecision** – точность сшивки граней (по умолчанию 0.3);
- **tesseleationStepData** – параметры триангуляции;
- **showMessages** – управление обращением к запросчику (по умолчанию *false*);
- **dualSeams** – флаг сохранения совпадающих точек швов (для конвертера STL и VRML);
- **lengthUnitsFactor** – масштабный коэффициент единиц длины при импорте и экспорте;
- **appUnitsFactor** – масштабный коэффициент единиц длины модели пользовательского приложения;
- **logRecords** – контейнер сообщений для пользователя во время импорта/экспорта.

Интерфейс IConvertorProperty3D и класс ConvConvertorProperty3D объявлены в файле *conv_exchange_settings.h*.

Преобразование кодировок

За преобразование кодировок отвечает интерфейс IC3DCharEncodingTransformer, реализацию которого возвращает метод GetUserCharEncodingTransformer(). Метод имеет штатные реализации:

- IC3DCharEncodingTransformerUTF8 - преобразование по правилам utf-8;
- IC3DCharEncodingTransformerWindows1251 – преобразование, соответствующее кодировке cp-1251 (кириллица Windows);
- C3DCharEncodingTransformerLocale - преобразование по правилам заданной локализации;
- IC3DCharEncodingTransformerStep - преобразование по нотации STEP.

Если преобразование явно не задано, то для формата STEP по умолчанию используется преобразование IC3DCharEncodingTransformerStep, а для остальных форматов C3DCharEncodingTransformerLocale.

8.5 Документ модели ItModelDocument

Интерфейс ItModelDocument входит в число интерфейсов, использование которых позволяет передавать данные, которые имеют сложную древовидную многокомпонентную структуру, в которой допускается многократное использование компонентов. Документ предоставляет два способа работы с моделью:

- Прямое использование модели, формируемой с использованием объектов геометрического ядра C3D.
- Пользователь реализует интерфейсы ItModelPart, ItModelAssembly, ItModelInstance и методы, которые работают с реализациями. Конвертер, в свою очередь, проводит обращение к интерфейсным методам возвращаемых объектов.

Методы интерфейса группируются следующим образом:

а) Группа методов, предназначенных для работы с моделью напрямую:

1. Метод задания корневого элемента модели:

```
virtual void C3dModelDocument::SetContent( MbItem* content );
```

2. Метод для получения наполнения модели:

```
virtual MbItem* GetContent();
```

3. Метод (устаревший)

```
virtual map_of_visual_items C3dModelDocument::GetAnnotationItems( eTextForm form );
```

позволяет получить элементы аннотации, соответствующие элементам геометрической модели.

Аргументы:

eTextForm – форма представления текста.

4. Метод (устаревший)

```
virtual void SetAnnotationItems( const map_of_visual_items& visItems );
```

позволяет задать размеры.

б) Группа методов, предназначенных для формирования модели при импорте:

1. Метод

```
virtual ModelAssemblyPtr C3dModelDocument::CreateAssembly(
    const std::ItemsSPtrVector& componentItems,
    const c3d::string_t&        fileName
);
```

позволяет создать документ с новой сборкой при импорте.

Аргументы:

fileName – имя сборки;

componentItems – тела, добавляемые в сборку.

2. Метод

```
virtual ModelPartPtr C3dModelDocument::CreatePart(
    const c3d::ItemsSPtrVector& componentItems,
    const c3d::string_t&        fileName
);
```

позволяет создать документ с новой деталью при импорте и увеличить счётчик ссылок результирующего документа на 1.

Аргументы:

componentItems – тела, добавляемые в деталь.

fileName – имя детали.

3. Метод

```
virtual bool C3dModelDocument::FinishImport( IProgressIndicator * indicator );
```

позволяет завершить импорт и сохранить документ. Возвращает *true*, если операция прошла успешно, *false* в противном случае.

Аргументы:

indicator – объект для отображения хода процесса.

в) Группа методов, предназначенных для получения связанных компонент модели при экспорте:

1. bool IsAssembly(); // является ли модель сборкой
2. bool IsEmpty(); // является ли модель пустой (не содержит ни деталь ни сборку)
3. Метод

```
virtual ModelAssemblyPtr C3dModelDocument::GetInstanceAssembly();
```

позволяет получить сборку для экспорта, ассоциированную с документом модели.

4. Метод

```
virtual ModelPartPtr C3dModelDocument::GetInstancePart();
```

позволяет получить деталь для экспорта, ассоциированную с документом модели.

5. Метод

```
bool C3dModelDocument::MakeAssembliesUnique();
```

позволяет преобразовать все сборки и вставки в уникальные компоненты.

Поскольку заранее неизвестно, какая реализация будет применяться для передачи данных, то при импорте вызывается как метод SetContent(), так и один из методов CreateAssembly() или CreatePart(), в зависимости от результата импорта. Соответственно, при экспорте сначала вызываются методы GetInstanceAssembly() или GetInstancePart() в зависимости от результата IsAssembly(), и, только если конвертеру не удаётся сформировать модель, вызывается метод GetContent().

В геометрическом ядре C3D имеется стандартная реализация интерфейса ItModelDocument: C3DModelDocument. При импорте модель формируется с использованием стандартных реализаций интерфейсов ItModelPart, ItModelAssembly, ItModelInstance. При экспорте можно либо задать модель с помощью метода SetContent, либо с использованием реализаций интерфейсов ItModelPart, ItModelAssembly, ItModelInstance.

Интерфейс ItModelDocument и класс C3DModelDocument объявлены в файле *conv_model_document.h*.

8.6 Индикатор процесса выполнения IProgressIndicator

Индикатор процесса выполнения (или просто индикатор) обеспечивает обратную связь конвертера с пользователем. В ходе обмена данными конвертер обращается к различным методам.

Для установки диапазона шагов, размера шага и отображаемых сведений о процессе служит метод

```
bool Initialize( size_t range, size_t delta, IStrData & massage ).
```

Для сообщения о выполнении некоторого числа шагов служит метод

```
bool Progress( size_t n ).
```

Метод возвращает признак того, следует ли продолжать операцию.

Для получения сведений о том, следует ли прерывать операцию по требованию пользователя, служит метод

```
bool IsCancel().
```

Для информирования пользователя о том, что операция завершена, служит метод

```
void Success().
```

Для информирования пользователя об аварийной остановке процесса служит метод

```
void Stop().
```

Пользователь может определить способ передачи информации о том, следует ли выполнить принудительную остановку процесса. В качестве стандартного способа пользователю предлагается реализовать метод

```
void SetCancel( bool cancel );
```

Интерфейс IProgressIndicator объявлен в файле *alg_indicator.h*.

В тестовом приложении класс ProgressIndicatorImp представляет собой одну из возможных реализаций индикатора.

Класс ProgressIndicatorImp объявлен и реализован в файле *test_converter.cpp* тестового приложения геометрического ядра C3D.

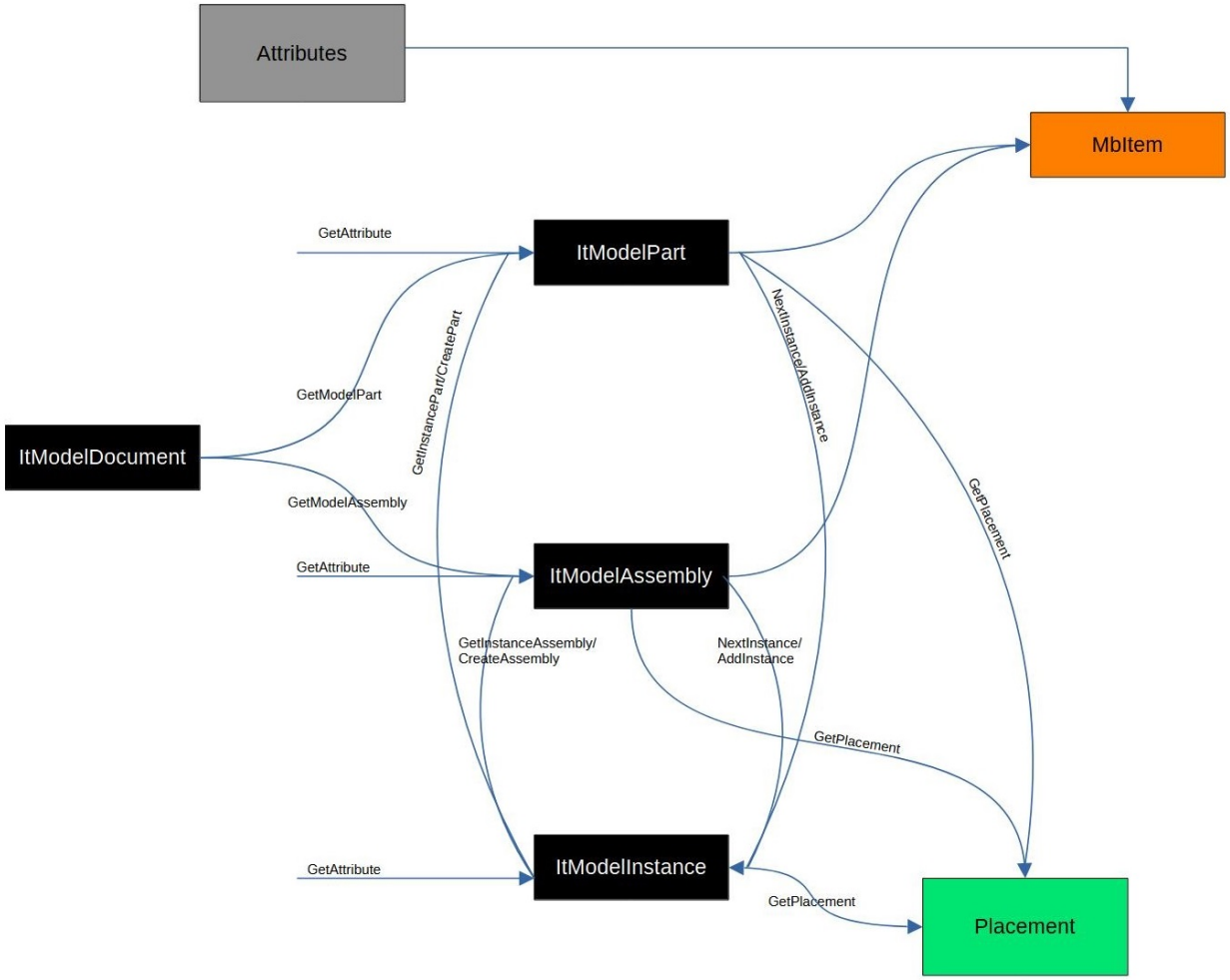
8.7 Архитектура дерева модели

Для модели, сформированной с использованием объектов геометрического ядра C3D, приняты следующие соглашения:

- Документ эквивалентен корневому компоненту (компоненту высшего уровня).
- Компоненту всегда соответствует объект типа MbAssembly в случае формата с граничным представлением. В случае формата с полигональным представлением в качестве компонента допустим объект типа MbMesh. Ожидается, что на компонент назначены свойства, и это выражается в том, что компонент обладает атрибутами типа MbProductInfo и MbPersonOrganizationInfo.
- Собственными объектами компонента могут быть объекты типа MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance (как с объектами геометрии внутри, так и содержащие элементы аннотаций MbPMI) и MbPlaneInstance для форматов с граничным представлением, либо MbMesh для полигональных форматов, которые лежат в MbAssembly.
- Вложенность компонент и их позиционирование осуществляется с помощью объектов типа MbInstance. MbInstance лежит в компоненте верхнего уровня и ссылается на компонент нижнего уровня. Циклические зависимости недопустимы.

Подход, согласно которому модель для обмена данными формируется средствами геометрического ядра C3D, имеет ряд недостатков. Наиболее существенный из них заключается в том, что использование конкретных объектов и конкретной конфигурации затрудняет развитие функционала - внесение любых изменений приведёт к необходимости внесения ответных изменений у всех пользователей. Использование интерфейсов модели, компонента и вставки, реализуемых на стороне пользователя, даёт последним независимость от конкретной реализации.

В терминах интерфейсов архитектура дерева модели выглядит следующим образом:



- Документ ItModelDocument() содержит в себе компонент высшего уровня.
- Компоненты бывают двух типов: детали ItModelPart и сборки ItModelAssembly. Разница между ними лишь в интерпретации, оба типа могут ссылаться на компоненты более низкого уровня.
- Собственными объектами компонентов могут быть объекты типа MbSolid, MbWireFrame, MbPointFrame, MbSpaceInstance и MbPlaneInstance.
- Вложенность компонент и их позиционирование осуществляется с помощью объектов ItModelInstance. Циклические зависимости недопустимы.

ItModelInstanceProperties — базовый класс, предок интерфейсов ItModelPart,

ItModelAssembly, ItModelInstance. В интерфейсе объявлены методы:

```
virtual bool SetAttributes( const c3d::AttrSPtrVector & /*attributes*/ );

virtual c3d::AttrSPtrVector GetAttributes();

virtual void GetRequirements( AnnotationSptrVector &, eTextForm ); (устаревший)

virtual void SetRequirements( const AnnotationSptrVector & ); (устаревший)
```

Интерфейсы ItModelPart и ItModelAssembly имеют одинаковый набор методов, поэтому их можно считать идентичными с точностью до интерпретации. Назначение свойств компонентов, визуальных свойств и технических требований реализуется посредством методов базового класса ItModelInstanceProperties.

Метод PureFileName() вызывается при экспорте и возвращает строку, смысл которой – имя файла компонента.

Методы для работы с собственными объектами:

```
void GetItems( std::vector< SPtr<MbItem> > & items,
               MbeGettingItemType            itemType,
               bool                          includeInvisible );

void AddItems( const std::vector< SPtr<MbItem> > & items );

GetItems();

AddItems();
```

Метод GetItems() предназначен для выдачи элементов в контейнер с использованием фильтров типа и видимости, которые формируются на основе значений, возвращаемых методом IConvertorProperty3D:: GetIoPermission().

Метод AddItems() используется при импорте и предназначен для добавления элементов в компонент.

Методы для работы со вставками:

```
SPtr<ItModelInstance> PrepareInstance();

SPtr<ItModelInstance> NextInstance( bool includeInvisible );

PrepareInstance();

NextInstance();
```

Метод PrepareInstance() предназначен для осуществления вставки компонента при импорте.

Метод NextInstance() представляет собой итератор вставок при экспорте. Предполагается, что итератор готов к выдаче вставок с начала. Признаком окончания является возврат нулевого указателя. Значение фильтра видимости в процессе обхода соответствует результату вызова метода IConvertorProperty3D:: GetIoPermission(iop_wInvisible).

Метод GetAnnotationItems(eTextForm **textForm**) является устаревшим.

Методы для импорта через плагины:

- Метод

```
virtual void ItModelPart::AddMappedItem( MbItem &                    item,
                                         MbPlacement3D const &       placement,
                                         c3d::AttrSPtrVector const & attributes );
```

позволяет добавить объект с заданной локальной системой координат в корень сборки при импорте. Метод актуален только для формата IFC (см. параграф *Импорт IFC*).

Аргументы:

item – объект, добавляемый в модель.

placement – локальная система координат.

- Метод

```
virtual SPtr<MbInstance> ItModelPart::NextMappedItem() const;
```

возвращает объект с заданной локальной системой координат или nullptr. Метод актуален только для формата IFC (см. параграф *Импорт IFC*).

- Метод

```
virtual void ItModelPart::AddAxesGrid( c3d::ItemSPtr const & );
```

позволяет добавить группу конструкционных осей. Метод актуален только для формата IFC (см. параграф *Импорт IFC*).

- Метод

```
virtual c3d::ItemSPtr ItModelPart::NextAxesGrid() const;
```

возвращает очередную группу конструкционных осей или нуль в случае исчерпания. Метод актуален только для формата IFC (см. параграф *Импорт IFC*).

- Метод

```
virtual c3d::ItemSPtr NextSpatialAxis() const { return c3d::ItemSPtr{}; }
```

возвращает очередную пространственную ось или нуль в случае исчерпания.

- Метод

```
virtual void AddSpatialAxis( c3d::ItemSPtr const & );
```

позволяет добавить очередную пространственную ось.

- Метод

```
virtual c3d::ItemSPtr ItModelPart::NextBoundaryItem();
```

возвращает следующий объект, ограничивающий площадь или объем. Метод актуален только для формата IFC (см. параграф *Импорт IFC*).

- Метод

```
virtual SPtr<MbAssistingItem> ItModelPart::NextCoordinateSystem();
```

возвращает очередную систему координат или нуль в случае исчерпания. Включение и выключение метода при импорте и экспорте возможно с помощью фильтрации SetIoPermission(iop_rCS / iop_wCS).

Методы для работы с модельным документом и сборками:

```
virtual void ItModelPart::AddBoundaryItem( c3d::ItemSPtr const &);
```

добавляет объект, ограничивающий площадь или объем.

```
virtual void ItModelPart::AddCoordinateSystem( SPtr<MbAssistingItem> const &);
```

позволяет добавить систему координат. Включение и выключение метода при импорте и экспорте возможно с помощью фильтрации SetIoPermission(iop_rCS / iop_wCS).

Интерфейсы ItModelPart и ItModelAssembly объявлены в файле *conv_model_document.h*.

Вставки ItModelInstance

Наследование интерфейса вставки ItModelInstance от ItModelInstanceProperties сложилось исторически и сохранено для совместимости. В рамках сложившегося подхода корректным является только назначение собственных визуальных свойств вставке.

Метод void * GetId() позволяет различать вставки эквивалентных компонентов. Если различные вставки возвращают одинаковое значение, то эквивалентность определяется по результатам анализа содержания. В процессе экспорта конвертер стремится к оптимизации модели за счёт многократного использования эквивалентных компонент. Пользователь может предоставить информацию о том, какие компоненты точно не являются эквивалентными для сокращения вычислительных затрат при экспорте.

При экспорте для получения такой информации о вставке как локальная система координат вставки, содержимое вставки, отсутствие наполнения вставки используются методы:

```
bool GetPlacement( MbPlacement3D & place );

bool IsAssembly();

bool IsEmpty();
```

Если вставка не пуста, то в зависимости от результата метода IsAssembly() вызывается один из методов:

```
SPtr<ItModelAssembly> GetInstanceAssembly();

SPtr<ItModelPart> GetInstancePart();
```

Эти методы возвращают интерфейс компонента, на который ссылается вставка.

Для создания компонентов при импорте предназначены методы:

```
SPtr<ItModelAssembly> CreateAssembly(const MbPlacement3D &              place,
                                     const std::vector<SPtr<MbItem> > &␣
↪componentItems,
                                     const c3d::path_string &           fileName );
```

```
SPtr<ItModelPart> CreatePart(      MbPlacement3D &              place,
                             const std::vector<SPtr<MbItem> > & componentItems,
                             const c3d::path_string &           fileName );
```

Результатом работы этих методов является интерфейс компонента, размещение которого в компоненте верхнего уровня описано локальной системой координат **place**, компонент наполняется элементами из контейнера **componentItems**, ему соответствует имя файла **fileName**.

Для размещения во вставке ранее созданных компонентов используются методы:

```
bool SetAssembly( const MbPlacement3D &   place,
                  const ItModelAssembly * existing );
```

```
bool SetPart( const MbPlacement3D & place,
              const ItModelPart *   existing );
```

Размещение компонента **existing** определяется локальной системой координат **place**.

Интерфейс ItModelInstance объявлен в файле *conv_model_document.h*.

8.8 Атрибуты изделия

Сведения об изделии, которые передаются через конвертер, включают в себя:

- Информация о компоненте: тип (деталь или сборочная единица), идентификатор, название, комментарий;
- Визуальные свойства, свойства материала, элементарные атрибуты;
- Информация об авторе и организации.

Указанная информация передаётся посредством атрибутов MbProductInfo и MbPersonOrganizationInfo и любых элементарных атрибутов (MbElementaryAttribute) соответственно.

Информация о компоненте MbProductInfo

Класс MbProductInfo нужно назначать на элемент модели MbItem, который нужно выделить как компонент, либо на элемент модели типа MbInstance в том случае, когда наименование, обозначение или комментарий вставки отличаются от соответствующих данных используемого компонента.

Основной конструктор класса MbProductInfo имеет следующую сигнатуру:

```
MbProcuctInfo( const c3d::string_t & initId,
               const c3d::string_t & initName,
               const c3d::string_t & initDesc,
                     bool            isAssm );
```

Аргумент **initId** используется для передачи идентификатора (обозначения) компонента, **initName** – наименование компонента, **initDesc** – комментарии, **isAssm** – признак того, является ли компонент сборочной единицей.

Основным способом получения строковых данных об изделии является вызов метода

```
void GetData( c3d::string_t & oId,
              c3d::string_t & oName,
              c3d::string_t & oDesc ) const;
```

Входные параметры **oId**, **oName**, **oDesc** используются для передачи идентификатора, наименования и комментария соответственно.

Для получения информации о том, является ли компонент сборочной единицей, используется метод

```
bool IsAssembly() const;
```

Класс MbProductInfo содержит также методы, принимающие строки типа *std::string*. Эти методы осуществляют преобразования с помощью функций c3d::ToSTDstring() и c3d::ToC3Dstring().

Класс MbProductInfo содержит конструктор:

```
MbProductInfo(       bool          isAssm,
              const std::string & initId,
              const std::string & initName,
              const std::string & initDesc );
```

Получение информации об изделии осуществляется методом

```
void GetDataStd( std::string & oId,
                 std::string & oName,
                 std::string & oDesc ) const.
```

8.9 Импорт конфигураций модели

При импорте моделей формата c3d, а также моделей доступных через плагины SolidWorks, NX, Creo, Inventor, Solid Edge есть возможность прочитать модель в разных конфигурациях (исполнениях). Импорт конфигураций из Catia не реализован, поскольку конфигурации в Catia сохранены в параметрическом виде, который C3D Converter не поддерживает.

Существует два способа задания выборщика конфигураций:

- Передать указатель на IConfigurationSelector в метод LoadForeignReader() при загрузке плагина для последующего использования в методах импорта через явно указанный плагин.
- Использовать метод IConverter3D::SetConfigurationSelector() для последующего использования в методах импорта через автоматически подгруженный плагин.

Для работы с этими методами необходимо реализовать интерфейс IConfigurationSelector (*conv_requestor.h*).

Методы интерфейса IConfigurationSelector:

```
// Добавить конфигурацию для выбора пользователем.
virtual void AddConfiguration( const c3d::string_t & configurationName ) = 0;

// Добавить конфигурацию для выбора пользователем, не содержащую модель.
virtual void AddEmptyConfiguration( const c3d::string_t & configurationName )
{
    AddConfiguration( configurationName );
}

// Добавить конфигурацию для выбора пользователем, требующую обновления.
virtual void AddNeedsUpdateConfiguration( const c3d::string_t & configurationName )
{
    AddConfiguration( configurationName );
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
}

// Указать индекс активной конфигурации.
virtual void SetActiveConfiguration( const size_t index ) = 0;

// Получить индекс конфигурации, выбранной пользователем.
virtual size_t GetConfiguration() const = 0;

// Индекс конфигурации для прекращения чтения по запросу пользователя.
static const size_t configurationIdToCancel = SYS_MAX_T;
```

- Без явного указания GetConfiguration() будет загружена конфигурация по умолчанию, заданная через SetActiveConfiguration().
- Методы AddEmptyConfiguration() и AddNeedsUpdateConfiguration() носят прежде всего информативный характер, т.к. в отличие от метода AddConfiguration() они не позволяют получить полноценную модель в силу ограничений сохранения модели из исходной САПР.

8.10 Плагины C3D Converter

Плагины позволяют читать модели из форматов, не поддерживаемых ядром C3D непосредственно, в первую очередь это проприетарные форматы. Технически плагин представляет собой библиотеку, реализующую заданный API и явно загружаемую ядром по запросу от приложения.

Плагин является частью C3D Toolkit и максимально интегрирован с геометрическим ядром. В частности это проявляется в том, что ядро автоматически загружает плагин и активирует его единым ключом, если:

1. Ключ содержит требуемый формат ([nx], [sw], [inv], [cat], [se], [cre], [ifc]),
2. Бинарные файлы плагина располагаются в том же каталоге, что и *c3d.dll*.

Некоторые особенности работы с плагинами:

- Плагины можно обновлять независимо от геометрического ядра C3D.
- В качестве упрощенной схемы с собственными плагинами можно работать через метод ImportFromFile() (при наличии лицензии и расположении бинарных файлов плагина в том же каталоге, что и *c3d.dll*).
- Если необходимо передавать какие либо настройки в экземпляр менеджера конвертации, который возвращает GetConverter3D, то схема использования плагина 2-х ступенчатая: GetConverter3D() + ImportFromFile() (если нужны специфичные объекты обратного вызова для настройки (выбор конфигурации)).
- Если необходимо передавать дополнительные настройки, такие как работа с конфигурациями модели или с метаданными, то необходимо использовать функцию GetConverter3D() для задания специфичных объектов обратного вызова, после чего вызвать ImportFromFile().
- Через плагины нельзя читать геометрические модели, используя метод ImportFromBuffer().

- Для использования сторонних плагинов либо для использования c3d плагинов, расположенных в каталоге, отличном от каталога с *c3d.dll*, используется последовательность вызовов:

GetConverter3D() - LoadForeignReader() - ImportForeign().

Список поддерживаемых версий c3d plugin:

Для форматов nwd, rvt, rvm, sp существуют партнерские плагины, совместимые с c3d_plugin, работа с ними аналогична работе с форматом ifc (см. *Вспомогательные методы конвертеров*).

8.11 Импорт IFC

При импорте IFC есть возможность получить специфичные для IFC типы данных:

- именованные плоские оси (NextAxesGrid());
- объекты, ограничивающие объём (NextBoundaryItem());
- многократно используемые объекты (NextMappedItem(), см. Интерфейсы ItModelPart и ItModelAssembly).

Чтобы эти данные получить, модель необходимо импортировать в ItModelDocument. В случае импорта в MbItem или MbModel вся специфика предметной области учитываться не будет, и указанные типы объектов преобразуются в обычные геометрические объекты.

Использование многократно используемых объектов похоже на использование обычных вставок в сборке, с той разницей, что в структуре сборок каждый вставляемый объект представляет собой отдельную логически завершённую деталь с атрибутом MbProductInfo, тогда как многократно используемый объект — это только геометрический блок (например, фрагмент забора, который при множественном инстанцировании образует целый забор).

Объекты, ограничивающие объём, по сути не являются геометрическими объектами. Это набор геометрических данных, который ограничивает объём, например комнаты или террасы. Такие объекты можно фильтровать при импорте, используя установку SetIoPermission(iop_rBounds).

Группы конструкционных осей представляют собой плоские объекты, отображающие оси и имеющие атрибуты, в том числе имена осей.

При импорте IFC, как и других строительных форматов (nwd, rvt, rvm, sp и др.), есть возможность импортировать метаданные (см. *Импорт метаданных строительных форматов*).

8.12 Импорт метаданных строительных форматов

Под метаданными строительных форматов подразумеваются разнотипные данные, которые не могут быть представлены в терминах геометрического ядра в виду чрезмерной сложности и специфики предметной области, а потому обрабатываются отдельно. К ним относятся сложноструктурированные свойства объектов, их взаимосвязи и деревья представления модели. Импорт метаданных доступен для формата *ifc* из *c3d_plugin*, а так же для форматов *nwd*, *rvt*, *rvm*, *sp*, работа с которыми так же производится через *c3d_plugin*.

8.13 Вспомогательные методы конвертеров

Для получения сведений о поддерживаемых форматах и загруженных плагинах C3D Converter предусмотрен ряд вспомогательных методов.

Получение списка поддерживаемых форматов

Получить список поддерживаемых форматов можно как через интерфейс IConverter3D, так и с помощью отдельной API-функции. Список включает как обменные форматы, доступные по стандартной лицензии [cnv], так и форматы, подключаемые через плагины.

Через интерфейс IConverter3D для этого предназначены методы:

```
virtual std::vector<c3d::FormatInfo> GetSupportedImportFormats();
```

```
virtual std::vector<c3d::FormatInfo> GetSupportedImportForeignFormats();
```

Метод GetSupportedImportFormats() возвращает список поддерживаемых форматов с учётом лицензионного ключа и наличия формат-специфичных библиотек конвертера. Если какой-либо плагин не поддерживает получение списка форматов, то его форматы не будут включены в результат.

Метод GetSupportedImportForeignFormats() возвращает список форматов, поддерживаемых для импорта через ImportForeign(). Список будет пустым, если метод LoadForeignReader() не вызывался или завершился неуспешно. Если какой-либо плагин не поддерживает получение списка форматов, то его форматы не будут включены в результат.

Для получения списка поддерживаемых форматов также предусмотрена отдельная API-функция:

```
CONV_FUNC( std::vector<FormatInfo> ) GetSupportedFormats();
```

Функция GetSupportedFormats() возвращает список поддерживаемых форматов с учётом лицензионного ключа и наличия формат-специфичных библиотек конвертера. Если какой-либо плагин не поддерживает получение списка форматов, то его форматы не будут включены в результат.

Структура FormatInfo содержит краткие сведения о формате: его название и список поддерживаемых расширений файлов.

Получение сведений о загруженных плагинах

Для получения сведений о загруженных и активированных плагинах используется функция:

```
CONV_FUNC( std::vector<PluginVersionInfo> ) GetLoadedPluginsVersions();
```

Функция GetLoadedPluginsVersions() возвращает версии загруженных плагинов. Значение версии определяется реализацией конкретного плагина. Если плагин не сообщает свою версию, то вместо неё используется строка «n/a».

Структура PluginVersionInfo содержит полный путь к плагину и версию плагина.

Версии плагинов различаются для разных форматов. Для механических форматов (*catia*, *creo*, *nx*, *sw*, *se*) и *ifc* используется единая версия плагина, например 22.0.29.0. Партнёрские плагины строительных форматов (*nwd*, *rvt*, *rvm*, *sp*) имеют собственные схемы нумерации версий, например 1.4.0.0, поскольку развивались независимо.

8.14 Создание тела по пользовательским объектам

Функция ImportSolid() предназначена для создания тела по данным, размещённым в контейнере пользовательских объектов ObContainer. Функция формирует импортируемое тело solid на основе набора ob-структур геометрии и топологии и возвращает код завершения операции типа MbeConvResType. Интерфейс функции:

```
CONV_FUNC( MbeConvResType ) ImportSolid( c3d::SolidSPtr &                    solid,
                                         std::shared_ptr<ObContainer const> ␣
↪obContainer,
                                         IConvertorProperty3D *              prop,
                                         IProgressIndicator *               ␣
↪indicator = nullptr );
```

Параметры функции:

solid – импортируемое твердое тело, которое будет создано по данным из контейнера;

obContainer – контейнер пользовательских объектов ObContainer;

prop – указатель на реализацию интерфейса свойств конвертера IConvertorProperty3D;

indicator – указатель на индикатор хода процесса IProgressIndicator (может быть равен *nullptr*, если отслеживание прогресса не требуется).

Контейнер ObContainer заполняется пользовательскими данными, представленными набором ob-структур для описания геометрии и топологии тела. После заполнения контейнер передаётся в функцию ImportSolid(), которая по исходным данным создаёт твердотельную модель. Функция исправляет некоторые типовые ошибки, характерные для импорта из различных форматов, а также выполняя сшивку и объединение подобных граней; поведение настраивается через IConvertorProperty3D стандартным образом.

При возникновении вопросов или ошибок на стороне пользователя исходные данные можно передать в техническую поддержку в виде отладочного текстового файла, формируемого методом ObContainer::WriteIntoFile().

ГЛАВА 9

Справочные материалы

9.1 Коды результата операции

9.1.1 Перечисление MbResultType

rt_Success = 0 – Нормальная работа.

Типы ошибок образующих контуров

rt_Empty – Пустой результат.

rt_ToManyAxis – Слишком много осей.

rt_ToFewAxis – Не хватает осей.

rt_ToManyContours – Слишком много контуров.

rt_Stars – Есть «звезда».

rt_SelfIntersection – Самопересечение контура.

rt_SelfIntWhenExtended – Самопересечение в продолжении контура.

rt_Intersection – Пересечение контуров.

rt_NoIntersectSolid – Образующий контур не пересекает тела.

rt_NoIntersectSection – Образующий контур не пересекает сечения (для операции построения тела по сечениям).

rt_LibNotFound – Фрагмент не найден в библиотеке.

rt_MustBeClosed – Должен быть замкнут.

rt_MustBeOpen – Должен быть разомкнут.

rt_AxisIntersection – Пересечение с осью.

rt_DegenerateAxis – Вырожденная ось.

Типы ошибок тела

rt_MultiSolid – Тело состоит из отдельных частей.

rt_CurveError – Ошибочная кривая.

rt_ContourError – Ошибочный контур.

rt_SurfaceError – Ошибочная поверхность.

rt_SolidError – Ошибочное тело.

rt_ParameterError – Ошибочный параметр.

rt_ThicknessError – Неправильно задана толщина.

rt_NoSequenceCurveAndSections – Не последовательное расположение сечений вдоль кривой (для операции построения тела по сечениям).

rt_SelfIntersect – Объект самопересекается.

rt_NoIntersect – Объекты не пересекаются.

rt_OffsetIntersectError – Невозможно построить эквидистанту с данными параметрами.

rt_BooleanError – Ошибка в булевой операции.

rt_NoEdges – Ребра не найдены.

rt_PrepareError – Ошибка при подготовке операции.

rt_ChamferError – Ошибка при создании фаски ребра.

rt_FilletError – Ошибка при создании скругления ребра.

rt_PartlyChamfer – Созданы фаски не на всех ребрах.

rt_PartlyFillet – Скруглены не все ребра.

rt_ChamferSurfaceError – Ошибка при создании поверхности фаски ребра.

rt_FilletSurfaceError – Ошибка при создании поверхности скругления ребра.

rt_TooLargeChamfer – Слишком большие катеты фаски.

rt_TooLargeFillet – Слишком большой радиус скругления.

rt_SemiChamfer – Фаски построены не для всех ребер.

rt_SemiFillet – Скруглены не все ребра.

rt_CuttingError – Ошибка резки поверхностью.

rt_ThinError – Ошибка при создании тонкостенного тела.

rt_OffsetError – Слишком большая толщина стенки при создании тонкостенного тела.

rt_FaceError – Ошибочная грань.

rt_RibError – Неизвестная ошибка постановки ребра жесткости.

rt_DraftError – Неизвестная ошибка уклона граней тела.

rt_NoObjectForDirection – В выбранном направлении отсутствует поверхность.

rt_AbsorptionSolid – Локальное тело поглощает результат.

rt_Error – Неизвестная ошибка.

rt_None – Нет сообщений.

rt_Intersect – Объекты пересекаются.

rt_InvalidType – Ненадлежащий тип кривой.

rt_NoConvertTextToNurbs – Преобразование текста выполнить невозможно.

rt_SplitWireNotSplitFace – Контур не разбивает ни одну из граней или совпадает с кромкой грани.

rt_SplitWireNotIntersectFace – Контур не пересекает выбранное множество граней или совпадает с кромкой грани.

rt_MustBeOnlyOnePoint – Для данной операции в эскизе должна быть только одна точка.

rt_InvalidPoleUsage – Эскиз с одной точкой может использоваться только для крайнего сечения.

rt_ThinWithPole – Построение тонкой стенки невозможно, если одно из сечений представляет собой точку.

rt_TopologyError – Ошибочная топология.

rt_NoObjectInDirNormal – Нет объекта в прямом направлении.

rt_NoObjectInDirReverse – Нет объекта в обратном направлении.

rt_SolidAffectedByBoolean – Тела изменены булевой операцией.

rt_HaveDegenerateSegment – Кривая содержит сегменты нулевой длины.

rt_NotAllSourcesFound – Не найдены одна или несколько исходных операций.

rt_InvalidEmptyContour – Контур состоит из 2 отрезков, проходящих друг по другу.

rt_UnnecessaryVariables – Избыточное количество переменных.

rt_DomainMismatch – Область определения не соответствует заданным значениям.

rt_UnknownTranslatorError – Неизвестная ошибка при работе транслятора.

rt_UnknownParserError – Неизвестная ошибка при работе синтаксического анализатора.

rt_UnknownSymbol – В строке присутствует неизвестный символ.

rt_NoClosingBracket – Не хватает закрывающей скобки.

rt_NoOpeningBracket – Не хватает открывающей скобки.

rt_ImpossibleOperation – Невозможная операция.

rt_LostObject – Операция потеряла опорные объекты.

rt_NotAllBendsProcessed – Не все сгибы согнуты/разогнуты.

rt_ValueScalingError – Масштабирование с заданным коэффициентом невозможно.

rt_RatioScalingError – Масштабирование с заданным соотношением коэффициентов невозможно.

rt_MultiSolidDeflected – Данная операция не применима к телам из частей.

rt_MultiSolidDefused – Результатом данной операции не может быть тело из частей.

rt_TransitionError – Перестало выполняться граничное условие сопряжения.

rt_MeshCrossingError – Точки пересечения кривых не образуют регулярной сетки.

rt_ClosedError – Нельзя выполнить замыкание.

rt_TooGreatCurve – Кривая, через которую происходит сопряжение, больше, чем сама поверхность.

rt_ClosedOrUnClosedError – Все кривые должны быть либо замкнуты, либо разомкнуты.

rt_SketchNoSheet – Эскиз должен располагаться только на внешней или внутренней плоской грани листового тела.

rt_BadSketch – Эскиз не удовлетворяет требованиям операции.

rt_ConnectionError – Нарушена связность объектов.

rt_MeshSmoothError – В точках пересечения кривые из противоположных семейств касаются.

rt_NoIntersectContour – Нет контуров пересечения.

rt_ErShMtHeight – Полный размер высоты должен быть больше толщины листового материала.

rt_DegenerateSurface – Поверхность вырождена.

rt_SurfaceEdgesIntersect – Невозможно создать поверхность: нарушен порядок внутренних ребер разбиения.

rt_TooLargeExtension – Величина удлинения слишком велика.

rt_TooSmallExtension – Величина удлинения слишком мала.

rt_VertexExtensionError – Не удалось продлить поверхность до данной вершины.

rt_SurfaceExtensionError – Не удалось продлить поверхность до данной поверхности.

rt_NoEdgesConection – Ребра не образуют связную цепочку.

rt_TooManyPoints – Большое количество точек. Уменьшите количество.

rt_TooManyPoints_1 – Слишком большое количество точек.

rt_DirectionExtensionError – Не удалось продлить поверхность в данном направлении.

rt_ExtensionPoleError – Грань содержит полюс: укажите другую кромку или выберите другой тип продления.

rt_MustBeOpenOrClosed – Контуры должны быть либо все замкнуты, либо все разомкнуты.

rt_TooComplicatedItemsSet – Слишком сложный набор элементов для обработки.

rt_NoAxesIntersection – Оси не пересекаются.

rt_TooFarItems – Объекты слишком далеко.

rt_ProcessIsStopped – Процесс остановлен.

rt_ContourSweptError – Контур невозможно использовать для заданного перемещения.

rt_SomeContourError – Один из контуров невозможно использовать для заданного построения.

rt_SplitWireNotAllFaces – Линии разъема (ребра пересечения) созданы не на всех выбранных гранях или не доходят до границ граней.

rt_GeneratrixColinearGuide – В некоторых точках образующая параллельна направляющей.

rt_NotEnoughMemory – Недостаточно памяти.

rt_BorderColinearCurve – Направление боковой границы параллельно касательной на конце образующей кривой.

rt_ObjectNotFound – Объект не найден.

rt_PoleBrokenError – В сплайновой поверхности часть точек из полюса передвинута, часть осталась совпадающей.

rt_ApproxError – Аппроксимация не выполнена.

rt_AccuracyError – Не выполнены условия по точности построения.

rt_BadVariable – Ошибочное значение переменной.

rt_BadEdgesForChamfer – Невозможно построить фаску на указанных ребрах.

rt_RevokeStopFillet – Остановка скругления невозможна, скругление выполнено без остановки.

rt_AdjacentTransitionError – Не согласованы сопряжения на смежных границах.

rt_ObjectAccessDenied – Доступ к объекту запрещен.

rt_TooManySegments – Количество сегментов слишком велико.

rt_GapShiftError – Недопустимое положение зазора обечайки.

rt_CutBySilhouetteError – Силуэтная грани линия не разрезает грань.

rt_DegeneratedProjection – Вырожденная проекция опорного объекта.

rt_NotAllContoursUsed – Использованы не все контура (кривые).

rt_ChangedParameter – Параметр операции был изменен.

rt_NotAllObjectsUsed – Использованы не все объекты.

rt_ZeroJumperError – Ошибка из-за образования перемычки нулевой толщины.

rt_FullFilletError – Ошибка при создании скругления грани.

Ошибки построения плавных сплайнов

rt_IncorrectData – Некорректные данные.

rt_IncorrectPolylines – Некорректные формы ломаных / направления касательных.

rt_IncorrectStructure – Некорректная структура ломаной с прямолинейными участками.

rt_TooFewPoints – Слишком мало точек.

rt_CoincidentPoints – Совпадение точек.

rt_TooSharpAngle – Слишком острый угол между сегментами ломаной.

rt_ReverseMotion – Обратный ход сегмента.

rt_SharpTorsion – Резкое кручение ломаной.

rt_IncorrectFirstDirection – Некорректное направление касательной в начальной точке.

rt_IncorrectLastDirection – Некорректное направление касательной в конечной точке.

rt_FirstTangentVector – Первая касательная задана к прямолинейному участку.

rt_LastTangentVector – Последняя касательная задана к прямолинейному участку.

rt_TangentVectorsUnsuitable – Касательные векторы не корректны для локально-выпуклого участка.

rt_IncorrectBezier – Некорректная структура исходной кривой Безье.

rt_NoIntersection – Нет пересечения касательных. Некорректная структура ломаной.

rt_CriticalConfiguration – Критическая конфигурация для B-сплайновой аппроксимации.

rt_IncorrectInitialData – Некорректные исходные данные.

rt_NotImplemented – Данная команда не реализована.

rt_InsufficientMemory – Недостаточно памяти.

rt_IncorrectFirstTangent – Некорректное направление первой касательной.

rt_IncorrectLastTangent – Некорректное направление последней касательной.

rt_StraightenLastSite – Конечный прямолинейный участок. Касательный вектор не нужен.

rt_IncorrectPolylineStructure – Некорректная структура ломаной с прямолинейными участками.

rt_IncorrectDataStructure – Некорректная структура данных в модели.

rt_ObjectIsNotPolyline – Объект - не пространственная ломаная.

rt_MissingKnots – Отсутствует вектор узлов.

rt_ClosedSpline – Замкнутый сплайн во внешнем файле.

rt_ObjectNotSelected – Не выбран нужный объект.

rt_SharpChangeCurvature – Резкое изменение кривизны на участке перегиба.

rt_CloseToStraightness – Близость к прямолинейности.

rt_TooFewStartPoints – Слишком мало точек на выпуклом участке начального участка.

rt_BanClosedConfiguration – Запрет на замкнутую конфигурацию при количестве точек < 5.

rt_IncorrectDeterminant – Некорректная структура исходного геометрического определителя.

rt_DimensionsPointsArrays – Размеры массивов точек.

rt_TangentsNotIntersect – Касательные не пересекаются.

rt_PointsCoincided – Точки совпадают.

rt_FewPointsForClosed – Мало точек для замкнутой ломаной.

rt_TooFewPointsOnSite – Слишком мало точек на выпуклом участке.

rt_PointsOnLine – Точки на прямой.

rt_TooManyApexes – Слишком много точек.

rt_SiteNotConvex – Трехсегментный участок не выпуклый.

rt_CurvatureRange – Кривизна за пределами допустимого диапазона.

rt_ObjectNotHermiteGD – Объект не ГО Эрмита.

rt_ObjectNotNURBS – Объект не NURBS.

rt_DerivativeContinuityErr – Ошибка обеспечения непрерывности производной.

rt_NotEnoughBuildData – Недостаточно данных для построения.

rt_InternalError – Неизвестная ошибка.

rt_DerivativeGap = 198 – Отсутствует непрерывность производной кривой по параметру.

rt_DerivativeGapU – Отсутствует непрерывность производной поверхности по первому параметру.

rt_DerivativeGapV – Отсутствует непрерывность производной поверхности по второму параметру.

rt_ContourGapError – В контуре есть разрывы между сегментами.

rt_ContourSegmentsOverlapError – В контуре есть наложения сегментов.

rt_ContourSegmentsNoTangentJoint – В контуре есть сегменты с не гладкой стыковкой.

rt_ExtensionOutOfRange – Удлинение не может быть ограничено заданной поверхностью.

rt_BuildSheetBySolidError – Ошибка построения листового тела по произвольному телу.

rt_BendEdgeError – Ребро не может быть использовано как ребро сгиба.

rt_StampToolPositionError – Ошибочное положение инструмента относительно листового тела.

Ошибки построения поверхности переменного сечения

rt_NoEdgesForBuild – Отсутствуют рёбра для построения.

rt_NoCurvesForBuild – Отсутствуют кривые для построения.

rt_NotCorrectDataForBuild – Некорректные данные.

rt_NotEnoughDataForBuild – Не хватает данных для построения.

rt_ReferenceCurveError – Ошибочная опорная кривая.

rt_GuideCurveError – Ошибочная направляющая кривая.

rt_ApexCurveError – Ошибочная вершинная кривая.

rt_DiscriminantFunctionError – Ошибочная дискриминантная функция.

rt_DiscriminantCurveError – Ошибочная дискриминантная кривая.

rt_DiscriminantSurfaceError – Ошибочная дискриминантная поверхность.

rt_AngleFunctionError – Ошибочная функция угла.

rt_ReferenceCurveBuildFailed – Опорную кривую не удалось построить.

rt_GuideCurveBuildFailed – Направляющую кривую не удалось построить.

rt_ApexCurveBuildFailed – Вершинную кривую не удалось построить.

rt_VertexCurveBuildFailed – Контрольную кривую не удалось построить.

rt_ApexDiscriminantBuildFailed – Вершинную кривую и дискриминант не удалось построить.

rt_DiscriminantBuildFailed – Дискриминантную функцию не удалось построить.

rt_PatternCurveError – Ошибочная образующая кривая.

rt_SynchronizeParameterError – Не удалось синхронизировать параметры.

rt_ReparamBuildFailed – Не удалось выполнить репараметризацию.

rt_TriangleSectionFailed – Треугольник сечения поверхности не может быть определен.

rt_DiscriminantFunctionWrong – Функция дискриминанта для задания формы сечения не обеспечивает построение поверхности.

rt_DiscriminantCurveWrong – Кривая для задания формы сечения не обеспечивает построение поверхности.

rt_DiscriminantSurfaceWrong – Касательная поверхность для задания формы сечения не обеспечивает построение поверхности.

rt_SectionMovementWrong – Движение сечения не совместимо с направляющими.

rt_DiscriminantSurfaceFar – Дискриминантная поверхность далеко от сечения.

rt_CrossOrderError – Нарушен порядок взаимного пересечения кривых.

rt_WrongExtensionLength – Некорректная длина продления.

rt_WrongExtensionWayValue – Неизвестный способ продления.

rt_StampToolHoleError – Ошибка, в области штамповки не должно быть вырезов.

rt_CurveClosedAtStart – Кривая замкнулась в начале.

rt_CurveClosedAtEnd – Кривая замкнулась в конце.

rt_CurveClosedBothSides – Кривая замкнулась с двух сторон.

rt_BeyondLimitsExtension – Продленная поверхностная кривая вышла за границы поверхности.

rt_NonBijectiveFunc – Отображение множества значений параметра функции во множество значений параметра кривой не является взаимно однозначным.

rt_LicenseError – Ошибка лицензии: неправильная или истекшая лицензия.

rt_NoRayIntersection – В выбранном направлении нет пересечения луча со вторым объектом.

rt_ExtremesRequestExceeded – Количество экстремумов превышает количество измерений.

rt_ConstraintSurfaceDomainError – В области определения элемента обнаружена недопустимая ситуация относительно внешнего объекта ограничения - поверхности.

rt_ConstraintCurveEmptyDomain – Пустая область определения относительно внешнего объекта ограничения - кривой.

rt_ConstraintCurveDomainError – В области определения элемента обнаружена недопустимая ситуация относительно внешнего объекта ограничения - кривой.

rt_ConstraintCurveNoBijection – Нарушение биекции между траекторией и внешним объектом ограничения - кривой.

rt_FacesGroupsOverlapError – Массивы заменяющих и заменяемых граней пересекаются.

rt_StitchShellsError – Ошибка сшивки оболочек.

rt_NoFacesForModification – Отсутствуют грани для модификации.

rt_NoReplacementFaces – Отсутствуют заменяющие грани.

rt_ErrorTotal – Общая ошибка операции.

9.1.2 Перечисление MbeStitchResType

```
enum MbeStitchResType {
    stch_Success = 0,
    stch_PrepareError,
    stch_CoorientFaceError,
    stch_SomeEdgesUnstitched,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
    stch_OutwardOrientError,
    stch_NoEdgeWasStitched,
    stch_SeparatePartsResult,
    stch_EdgeStitchError,
    stch_InputTopologyError
};
```

stch_Success = 0 – Нет ошибок.

stch_PrepareError – Ошибка при подготовке операции сшивки.

stch_CoorientFaceError – Невозможно выставить согласованную ориентацию граней.

stch_SomeEdgesUnstitched – Некоторые рёбра остались несшитыми.

stch_OutwardOrientError – Не удалось установить нормали граней наружу тела.

stch_NoEdgeWasStitched – Не было сшито ни одного ребра.

stch_SeparatePartsResult – После сшивки остались несвязанные между собой куски.

stch_EdgeStitchError – Ошибка сшивки ребра.

stch_InputTopologyError – Критические ошибки топологии во входных оболочках.

9.2 Способы построения поверхности сопряжения (скругления или фаски)

```
enum MbeSmoothForm
{
    st_Unknown     = -3, // Неопределенное сопряжение
    st_Ellipse     = -2, // Скругление дугой эллипса с пересчетом радиусов
    st_Span        = -1, // Скругление с заданной хордой
    st_Fillet      =  0, // Скругление с заданными радиусами
    st_Chamfer     =  1, // Фаска с заданными катетами
    st_Slant1      =  2, // Фаска по катету и углу (катет distance2 рассчитан для␣
↪прямого угла между гранями и определяет прилегающий к катету distance1 угол)
    st_Slant2      =  3, // Фаска по углу и катету (катет distance1 рассчитан для␣
↪прямого угла между гранями и определяет прилегающий к катету distance2 угол)
    st_ByCurvature =  4  // Скругление с непрерывной кривизной на границе с опорными␣
↪поверхностями
};
```