УЧЕБНЫЙ КУРС
по геометрическому ядру C3D

РАБОТА 3

Составные кривые и сплайны в двумерном пространстве

https://c3dlabs.com
Москва
2018

Содержание

1. Введение

В работе №2 были рассмотрены классы для представления простейших кривых в двумерном пространстве, которые описывались с помощью известных аналитических выражений. Для точного представления произвольных кривых этих классов часто оказывается недостаточно.

Некоторые кривые можно представить в виде совокупности сегментов – последовательно соединяющихся частей различных известных кривых. Для этих целей в ядре C3D применяются контуры (класс MbContour).

Еще одним инструментом для моделирования формы кривых являются сплайны – гладкие кривые, форма которых зависит от набора контрольных точек. Сплайны состоят из соединяющихся сегментов. В отличие от контуров, все сегменты сплайнов описываются одинаковыми математическими выражениями – полиномиальными функциями (многочленами). В математике известны различные виды сплайнов, в уравнениях которых используются многочлены разных видов. В геометрическом моделировании оказываются удобны не все сплайны – например, сплайны на базе многочленов высокой степени могут сильно колебаться между контрольными точками. Чаще требуется построить кривую, не только гладкую, но и «не содержащую чрезмерного количества изгибов и скачков» между контрольными точками. Для этой цели лучше подходят сплайны на базе полиномов невысокого порядка, в первую очередь, кубические сплайны (классы MbCubicSpline и MbHermit).

В геометрическом моделировании широко применяется особый вид сплайнов – NURBS-сплайны (и их частный случай – кривые Безье). Они удобны тем, что позволяют в унифицированном виде представить любую кривую. Важная роль NURBS-кривых также объясняется тем, что они применяются в ядре для представления линий пересечения произвольных поверхностей (в частности, ребер в B-Rep моделях твердых тел). В C3D для представления кривых Безье и NURBS-сплайнов предназначены классы MbBezier и MbNurbs.

2. Составные кривые

2.1 MbContour – контур в двумерном пространстве

Контур представляет собой составную кривую, образованную последовательно соединяющимися сегментами. Каждый сегмент может быть фрагментом любой другой кривой (за исключением другого контура). Контуры могут быть разомкнутыми и замкнутыми. Сегменты контура могут соединяться гладко (с выполнением требования равенства производных в точках соединения) или с изломами. Примеры контуров показаны на рис. 1. На этом рисунке у разомкнутого контура имеется 5 сегментов с 4-мя стыками (из них один гладкий), у замкнутого – 11 сегментов с 11-тью стыками (из них 2 гладких).

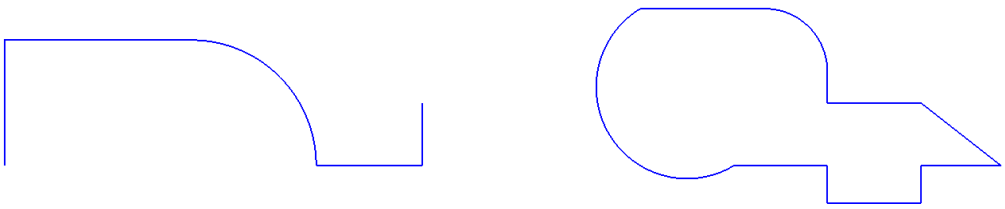


Рис. 1. Примеры разомкнутого и замкнутого контуров, построенных на базе отрезков и дуг окружностей.

Для представления контуров предназначен класс MbContour, унаследованный от класса кривой в двумерном пространстве MbCurve. В классе MbContour реализованы методы, опре-

деленные общим интерфейсом кривой, а также добавлен ряд методов, специфических для контура. Ниже приведен фрагмент интерфейса MbContour (заголовочный файл cur_contour.h) с атрибутами и методами, предназначенными для операций с отдельными сегментами контура.

```
class MbContour : public MbCurve {
protected :
  RPArray<MbCurve> segments;     // Массив сегментов контура
  bool             closed;       // Признак замкнутого контура

  // Вспомогательные атрибуты
  // Параметрическая длина контура (величина диапазона значений параметра t)
  mutable double   paramLength;
  // Метрическая длина контура (суммарная длина сегментов)
  mutable double   metricLength;
  // Габаритный прямоугольник
  mutable MbRect   rect;

public :
  // ГРУППА #1: КОНСТРУКТОРЫ
  // Конструктор пустого контура (для последующего вызова метода Init)
  MbContour();
  // Конструктор контура на базе одного начального сегмента
  MbContour( const MbCurve& init, bool same );
  // Конструктор контура по массиву сегментов
  MbContour( const RPArray<MbCurve>& init, bool same );

  // ГРУППА #2: МЕТОДЫ ОТЛОЖЕННОЙ ИНИЦИАЛИЗАЦИИ
  // Инициализация контура по набору кривых
  bool Init( List<MbCurve>& curves );
  // Инициализация с копированием существующего контура
  void Init( const MbContour& other );
  // Инициализация замкнутого контура по массиву точек
  void Init( SArray<MbCartPoint>& points );

  // ГРУППА #3: МЕТОДЫ, УНАСЛЕДОВАННЫЕ ОТ MbPlaneItem и MbCurve (не показаны)

  // ГРУППА #4: ИНФОРМАЦИОННЫЕ МЕТОДЫ КОНТУРА
  // Проверка, являются ли стыки контура гладкими
  virtual bool   IsSmoothConnected() const;
  // Вычисление метрической длины разомкнутого контура между точками, соответствующими
  // значениям параметра t1 и t2
  virtual double CalculateLength( double t1, double t2 ) const;
  // Проверка, является ли контур замкнутым и непрерывным (у замкнутого контура
  // первая и последняя точки должны совпадать с точностью до величины eps.)
  bool IsClosedContinuousC0( double eps = 5.0 * PARAM_NEAR ) const;
  // Проверка, есть ли в контуре криволинейный сегмент
  bool IsAnyCurvilinear() const;
  // Проверка, есть ли у двух контуров идентичные сегменты
  bool IsSameSegments( const MbContour& cntr ) const;

  // ГРУППА #5: МЕТОДЫ ДЛЯ РАБОТЫ С СЕГМЕНТАМИ
  // Получить количество сегментов контура
  int      GetSegmentsCount() const { return segments.Count(); }
  // Получить сегмент контура по индексу (значения индексов начинаются с 0)
  const MbCurve* GetSegment( int i ) const { return segments[i]; }
  // Получение всех сегментов контура
  void     GetSegments( RPArray<MbCurve>& segms ) const;
  // Получение начальной точки i-го сегмента
  bool     GetBegSegmentPoint( int i, MbCartPoint& p ) const;
  // Получение конечной точки i-го сегмента
  bool     GetEndSegmentPoint( int i, MbCartPoint& p ) const;
```

```
  // Добавление сегмента в контур
  bool     AddSegment( MbCurve* );
  MbCurve* AddSegment( const MbCurve* pBasis, double t1, double t2, int sense = 1 );
  // Вставка сегмента перед сегментом контура с индексом index
  bool     AddAtSegment( MbCurve* newSegment, int index );
  // Вставка сегмента после сегмента контура с индексом index
  bool     AddAfterSegment( MbCurve* newSegment, int index );
  // Удаление всех сегментов контура
  void     DeleteSegments();
  // Удаление сегмента с заданным индексом
  void     DeleteSegment( int ind );

  // ГРУППА #6: МЕТОДЫ ДЛЯ ПОСТРОЕНИЯ СКРУГЛЕНИЙ И ФАСОК В СТЫКАХ КОНТУРА
  // Скругление контура с заданным радиусом
  bool Fillet( double rad );
  // Скругление двух соседних сегментов
  bool FilletTwoSegments( int& index, double rad );
  // Вставка фаски
  bool Chamfer( double len, double angle, bool type );
  // Вставка фаски между двумя соседними сегментами
  bool ChamferTwoSegments( int& index, double len, double angle, bool type,
                           bool firstSeg=true );
  // Удаление скругления или фаски
  MbeState RemoveFilletOrChamfer( const MbCartPoint& pnt );

  // Другие методы MbContour...
};
```

Применение класса MbContour для представления контура пластины демонстрируется в примере 2.1. Этот контур (рис. 2) состоит из разомкнутой ломаной линии и дуги окружности, гладко замыкающей разрыв ломаной. Обозначения размеров и точек на рис. 2 приведены для пояснения построений в программном фрагменте и не входят в состав контура. Вершины ломаной при построении перечисляются против часовой стрелки, от точки A до точки B. Координаты точек задаются относительно начала координат, выбранного в левой нижней точке ломаной (т. O).

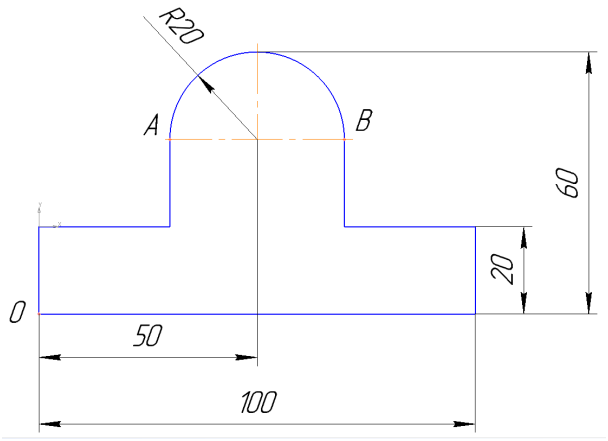


Рис. 2. Замкнутый контур, состоящий из ломаной линии и дуги окружности (пример 2.1).

Пример 2.1. Построение контура (рис. 2).

```
#include "cur_contour.h"        // MbContour - контур в двумерном пространстве
#include "cur_polyline.h"       // MbPolyline - Ломаная линия
#include "cur_arc.h"            // MbArc - Эллиптическая дуга
```

```
void MakeUserCommand0()
{
  MbPlacement3D pl; // Локальная СК (по умолчанию совпадает с мировой СК)

  // Вершины ломаной
  SArray<MbCartPoint> arrPnts(8);
  arrPnts.Add( MbCartPoint(30, 40) );
  arrPnts.Add( MbCartPoint(30, 20 ) );
  arrPnts.Add( MbCartPoint(0, 20) );
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(100, 0) );
  arrPnts.Add( MbCartPoint(100, 20) );
  arrPnts.Add( MbCartPoint(70, 20) );
  arrPnts.Add( MbCartPoint(70, 40) );

  // Ломаная линия с вершинами arrPnts
  MbPolyline* pPolyline = new MbPolyline( arrPnts, false /* Флаг незамкнутой линии */);

  // Дуга окружности для замыкания ломаной.
  // При построении указывается центр, радиус, начальная и конечная точки и
  // направление обхода дуги между этими точками (значение initSense>0 соответствует
  // обходу против часовой стрелки, а initSense<0 – по часовой стрелке).
  MbCartPoint arcCenter( 50, 40 );
  const double RADIUS = 20;
  MbArc* pArc = new MbArc( arcCenter, RADIUS, arrPnts[7], arrPnts[0], 1 /*initSense*/);

  // Контур из двух сегментов
  MbContour* pContour= new MbContour( *pPolyline, true );
  pContour->AddSegment( pArc );

  // Отображение контура
  if (pContour )
    viewManager->AddObject( Style( 1, RGB(0,0,255) ), pContour, &pl );

  // Вызовы информационных методов контура
  size_t segmentsCount = pContour->GetSegmentsCount();          // 2
  double tMin = pContour->GetTMin();                            // 0.0
  double tMax = pContour->GetTMax();                            // 10.14159
  // Параметр IsSmoothConnected - допуск для классификации угловых точек на контуре
  bool   isSmooth = pContour->IsSmoothConnected(0.001);         // false
  double length = pContour->CalculateLength( tMin, tMax );      // 302.83
  bool   isClosed = pContour->IsClosedContinuousC0();           // true
  bool   isCurvilinear = pContour->IsAnyCurvilinear();          // true

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pPolyline );
  ::DeleteItem( pArc );
  ::DeleteItem( pContour );
}
```

В тексте примера 2.1 обратите внимание на порядок перечисления точек при построении дуги окружности. Дуга окружности добавляется в контур после ломаной линии. На момент добавления дуги последней точкой контура является вершина последнего сегмента ломаной – отрезок с координатами вершин arrPnts[6] и arrPnts[7]. Поэтому замыкающая дуга должна начинаться в точке B (arrPnts[7]) и заканчиваться в точке A (arrPnts[0]), а не наоборот.

В тексте примера также приведены вызовы информационных методов класса MbContour, а в комментариях указаны возвращаемые значения. По этим значениям видно, что построенный контур состоит из двух сегментов (ломаная при добавлении в контур на отрезки не раскладывается и добавляется как один сегмент). Контур является замкнутым и содержит криволинейный сегмент, метрическая длина контура равна 302.83, а значение параметра $t \in [0.0,$

10.14159]. Диапазон значений *t* складывается из диапазона [0.0, 7.0] для ломаной из 7 отрезков и диапазона [0, π] для дуги – полуокружности.

2.2 Применение контуров для построения трехмерных объектов

Контуры могут применяться в качестве параметров операций для построения трехмерных объектов, например, поверхностей и твердых тел (классы ядра для представления трехмерных геометрических объектов подробнее будут рассмотрены в последующих работах).

На рис. 3 приведены примеры использования контура, показанного на рис. 2, для построения поверхности выдавливания и твердого тела выдавливания. При этих построениях контур используется в качестве образующей кривой, а направляющей является прямолинейный отрезок заданной длины.

Для построения поверхности выдавливания выполняются следующие действия:

1) Построение двумерного контура образующей (класс MbContour).
2) Построение трехмерного контура посредством размещения двумерного контура на заданной плоскости в трехмерном пространстве (класс MbContourOnPlane).
3) Вызов функции-алгоритма ::ExtrusionSurface для выполнения операции выдавливания с указанием трехмерного контура-образующей и направляющего вектора. Эта функция возвращает динамически созданный объект-поверхность MbSurface.

Эта схема реализована в примере 2.2, который является дополнением примера 2.1. Для выполнения шага 1 в начало примера 2.2 следует поместить текст из примера 2.1 от начала до первого вызова функции ::DeleteItem (удалять объект-контур pContour до построения поверхности нельзя).

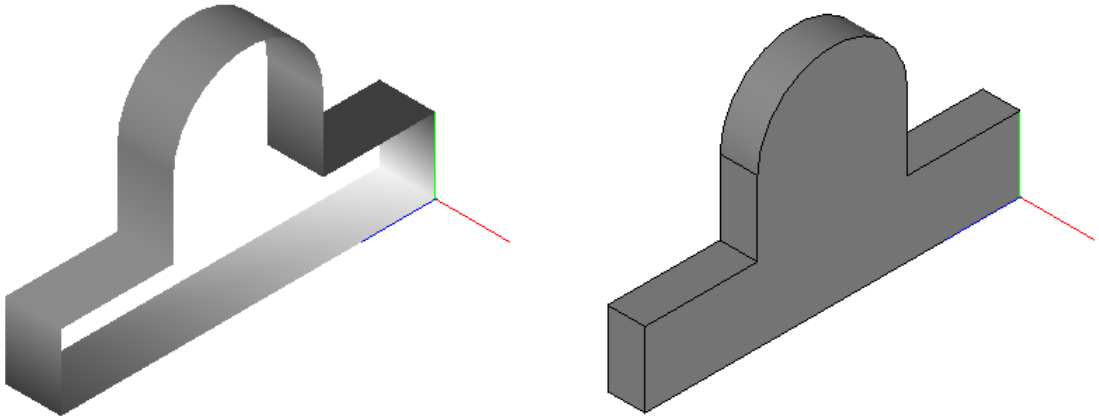


Рис. 3. Поверхность и твердое тело выдавливания, построенные с использованием качестве образующей контура с рис. 2 (примеры 2.2 и 2.3).

Пример 2.2. Построение поверхности выдавливания на основе двумерного контура.

```
// Заголовочные файлы, дополнительные к примеру 2.1
#include "surf_plane.h"                // MbPlane - плоскость
#include "cur_contour_on_plane.h"      // MbContourOnPlane - Контур на плоскости
                                       // (кривая в трехмерном пространстве)
#include "action_surface.h"            // Функции- алгоритмы построения поверхностей

void MakeUserCommand0()
{
  // ... сюда следует поместить часть примера 2.1 до первого вызова ::DeleteItem
  // Из скопированной части удалите фрагмент, связанный с отображением геометрических
  // объектов (вызов viewManager->AddObject() и предшествующую проверку if).

  // ПОСТРОЕНИЕ ПОВЕРХНОСТИ ВЫДАВЛИВАНИЯ
```

```
  // Контур-образующую разместим в плоскости ZY глобальной системы координат.
  // Эта плоскость представляется в виде объекта MbPlane и строится по точке-началу
  // координат и двум векторам, задающим оси локальной системы координат
  // плоскости – в данном случае это оси Z и Y глобальной СК.
  MbPlane* pPlaneZY = new MbPlane( MbCartPoint3D(0,0,0), MbVector3D(0,0,1),
                                   MbVector3D(0,1,0) );
  // Трехмерная кривая "контур на плоскости", которая будет использоваться в качестве
  // образующей для операции выдавливания
  MbContourOnPlane* pc = new MbContourOnPlane( *pPlaneZY, *pContour, true );

  // Вектор, задающий направление и величину выдавливания (на 15 единиц в
  // отрицательном направлении оси X глобальной СК)
  MbVector3D dir( -15, 0, 0 );

  // Вызов функции для построения поверхности выдавливания
  MbSurface* pSurf = NULL;
  MbResultType res = ::ExtrusionSurface(
                *pc, dir, true /* разрешение на упрощение поверхности */, pSurf );

  // Отображение построенной поверхности
  if ( res == rt_Success )
    viewManager->AddObject( Style(1, LIGHTGRAY), pSurf );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pPlaneZY );
  ::DeleteItem( pc );
  ::DeleteItem( pSurf );

  ::DeleteItem( pPolyline );
  ::DeleteItem( pArc );
  ::DeleteItem( pContour );
}
```

Применение контура в качестве образующей для построения твердого тела выдавливания (рис. 3, справа) показано в примере 2.3. При построении тела, по сравнению с поверхностью, производится автоматическое формирование граней и расчет ребер (показаны отрезками на рис. 3, справа).

Порядок построения тела выдавливания следующий:

1) Построение двумерного контура образующей (класс MbContour).
2) Создание объектов для хранения параметров операции выдавливания: образующая для операции выдавливания (класс MbSweptData), направляющий вектор (MbVector3D), параметры движения образующей вдоль направляющего вектора (класс ExtrusionValues).
3) Создание объекта для именования граней твердого тела.
4) Вызов функции-алгоритма для построения твердого тела методом выдавливания.

Перечисленный порядок действий более сложный, чем в случае построения поверхности выдавливания. Использование нескольких различных классов для представления параметров одной операции допускает большое количество вариантов построения, а также позволяет однотипным образом применять различные операции построения твердых тел (они рассматриваются в следующих работах).

Пример 2.3. Построение твердого тела выдавливания на основе двумерного контура.

```
// Заголовочные файлы, дополнительные к примеру 2.1
#include "solid.h"              // MbSolid - твердое тело
#include "creator.h"            // Классы для построения геометрических моделей
#include "action_solid.h"       // Функции-алгоритмы построения твердых тел
```

```
void MakeUserCommand0()
{
  // ... сюда следует поместить часть примера 2.1 до первого вызова ::DeleteItem
  // Из скопированной части удалите фрагмент, связанный с отображением геометрических
  // объектов (вызов viewManager->AddObject() и предшествующую проверку if).

  // ПОСТРОЕНИЕ ТЕЛА ВЫДАВЛИВАНИЯ

  // Локальная СК в трехмерном пространстве, у которой ось X совпадает с
  // осью Z глобальной СК. Строится по двум осям X и Y и началу координат.
  MbPlacement3D plZY( MbVector3D(0,0,1), MbVector3D(0,1,0), MbCartPoint3D(0,0,0) );

  // Объект, хранящий параметры образующей (контур и локальная СК).
  // Контур размещается на плоскость XY локальной СК plZY.
  MbSweptData sweptData( plZY, *pContour );

  // Направляющий вектор для операции выдавливания
  MbVector3D dirV( -1, 0, 0 );

  // Параметры операции выдавливания, задающие свойства тела для построения -
  // расстояние выдавливания в прямом и в обратном направлении вдоль
  // направляющего вектора
  const double HEIGHT_FORWARD = 10.0, HEIGHT_BACKWARD = 0.0;
  ExtrusionValues extrusionParam( HEIGHT_FORWARD, HEIGHT_BACKWARD );

  // Служебный объект для именования элементов модели твердого тела
  MbSNameMaker operNames( ct_CurveExtrusionSolid, MbSNameMaker::i_SideNone, 0 );
  PArray<MbSNameMaker> cNames( 0, 1, false );

  // Построение твердого тела выдавливания
  MbSolid* pSolid = NULL;
  MbResultType resCode = ::ExtrusionSolid( sweptData, dirV, NULL, NULL, false,
                                           extrusionParam, operNames, cNames, pSolid);

  // Отображение построенного тела
  if ( resCode == rt_Success )
    viewManager->AddObject( Style(1, LIGHTGRAY), pSolid );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pPolyline );
  ::DeleteItem( pArc );
  ::DeleteItem( pContour );
  ::DeleteItem( pSolid );
}
```

2.3 Сопряжение сегментов контура

Типичной задачей при построении составных кривых является организация сопряжений в точках соединения сегментов. На рис. 4 показаны три варианта сопряжений применительно к замкнутой ломаной линии – совпадение вершин, скругления и фаски. Сопряжение с использованием скруглений можно рассматривать как построение контура посредством добавления сопрягающих сегментов в виде дуг окружностей, а сопряжение с использованием фасок – как построение ломаной или контура с добавлением сопрягающих отрезков. Скругления обеспечивают гладкое сопряжение, фаски – сопряжение с угловыми точками.

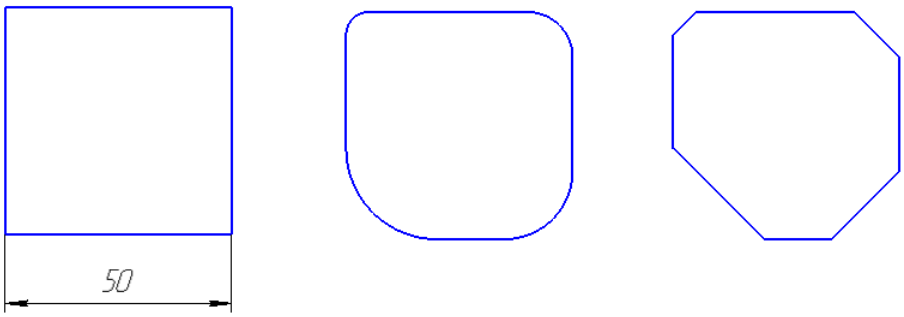


Рис. 4. Типы сопряжений сегментов ломаной линии, изображающей квадрат (пример 2.4). (слева) Совпадение вершин. (центр) Скругления радиусом 5, 10, 15 и 20. (справа) Фаски длиной 5, 10, 15 и 20 с одинаковым направлением 45°.

Операции построения скруглений и фасок применяются часто, поэтому в ядре предусмотрены средства для упрощения подобных построений. Они реализованы в виде функций-алгоритмов для построений в двумерном пространстве и в виде методов классов MbPolyline и MbContour. Большинство функций-алгоритмы (описаны в заголовочном файле alg_curve_fillet.h) создают новые объекты, представляющие либо сопрягающие дугу/отрезок, либо модифицированные ломаную/контур:

- Функция ::Fillet для построения скругления между двумя произвольными кривыми. Создает объект MbArc, представляющий дугу скругления.
- Функция ::Chamfer для построения фаски между двумя произвольными кривыми. Создает новый объект MbLineSegment, представляющий отрезок фаски.
- Функция ::FilletPolyContour для построения скругления в вершинах ломаных или контуров. Создает новый объект MbContour, представляющий контур с добавленным скруглением.
- Функция ::ChamferPolyContour для построения фасок в вершинах ломаных или контуров. Модифицирует исходную кривую посредством добавления в нее одного прямолинейного сегмента.

При построении контуров получать в явном виде сопрягающие дуги и отрезки требуются не всегда, чаще бывает достаточно добавить сегмент к существующей кривой. Для этого могут применяться методы, предусмотренные в классе MbContour (см. выше фрагмент интерфейса класса MbContour, методы в группе #6).

Построение скруглений и фасок демонстрируется в примере 2.4 для построения кривых, показанных на рис. 4. При построении нескольких скруглений и фасок с указанием индексов сегментов в методах MbContour необходимо правильно указывать значения индексов, учитывая, что ранее добавленные скругления и фаски увеличивают количество сегментов контура. На рис. 4 контур со скруглениями и ломаная с фасками содержат по 8 сегментов.

Пример 2.4. Построение скруглений и фасок (рис. 4).

```
#include "cur_polyline.h"       // MbPolyline - ломаная линия
#include "cur_contour.h"        // MbContour - контур
#include "alg_curve_fillet.h"   // Функции для построения скруглений и фасок

void MakeUserCommand0()
{
  MbPlacement3D pl; // Локальная СК (по умолчанию совпадает с мировой СК)

  // Ломаная линия, изображающая квадрат.
  // Вершины перечисляются по часовой стрелке, начиная с левого нижнего угла.
  const double SQUARE_SIDE = 50;
  SArray<MbCartPoint> arrPnts(4);
```

```
arrPnts.Add( MbCartPoint(0, 0) );
arrPnts.Add( MbCartPoint(0, SQUARE_SIDE) );
arrPnts.Add( MbCartPoint(SQUARE_SIDE, SQUARE_SIDE) );
arrPnts.Add( MbCartPoint(SQUARE_SIDE, 0 ) );
MbPolyline* pSquare = new MbPolyline( arrPnts, true /* Флаг замкнутой линии */);

// Построение нового контура со скруглениями на основе ломаной-квадрата.
// В качестве входных параметров указывается радиус скругления,
// флаг построения только в одной вершине, и точка, задающая вершину для скругления.
// Функция возвращает новую кривую-контур со скруглением в виде
// выходного параметра pFillets.
MbContour* pFillets = NULL;
::FilletPolyContour(pSquare, 5, false, arrPnts[1], pFillets );
// Построение еще трех скруглений с помощью методов MbContour.
// Указывается индекс первого из скругляемых сегментов и
// радиус скругления. При указании индексов учитывается, что
// каждое скругление добавляет один сегмент в контур.
// После вызова метода для построения скругления значения индексов
// изменяются и соответствуют построенным сегментам-дугам.
ptrdiff_t idxSideTop    = 2;         // Индекс верхней стороны квадрата
ptrdiff_t idxSideRight = 4;          // Индекс правой стороны
ptrdiff_t idxSideBottom = 6;         // Индекс нижней стороны
pFillets->FilletTwoSegments( idxSideTop,    10 /* радиус скругления */ );
pFillets->FilletTwoSegments( idxSideRight,  15 );
pFillets->FilletTwoSegments( idxSideBottom, 20 );
// Сдвиг контура по горизонтали, чтобы он не накладывался на исходный квадрат
pFillets->Move( MbVector( SQUARE_SIDE*3/2, 0 ) );

// Создание копии ломаной-квадрата для построения ломаной с фасками
MbPolyline* pChamfers = new MbPolyline( *pSquare );
// Сдвиг ломаной, чтобы она не накладывалась на уже построенные кривые.
pChamfers->Move( MbVector( 2 * SQUARE_SIDE*3/2, 0 ) );
// Длины фасок (начиная с левого нижнего угла квадрата)
const double arrLen[] = { 20, 5, 10, 15 };
// Построение фасок в углах ломаной, начиная с левого нижнего угла
const double DEG_TO_RAD = M_PI/180.0;
for (int i = 0; i < 4; i++)
{
  // Индекс очередной вершины, в которой строится фаска. Множитель 2 нужен для
  // учета того, что каждая фаска добавляет один сегмент к ломаной.
  int idxPnt = i*2;
  // Координаты очередной вершины квадрата.
  MbCartPoint pntVert;
  pChamfers->GetPoint( idxPnt, pntVert );
  // Функция построения фаски модифицирует исходную кривую посредством
  // добавления отрезка длиной arrLen[i] с направлением 45 градусов.
  ::ChamferPolyContour(pChamfers, arrLen[i], 45*DEG_TO_RAD, true, false, pntVert );
}

// Отображение ломаных и контура
if ( pSquare )
  viewManager->AddObject( Style( 1, RGB(0,0,255) ), pSquare, &pl );
if ( pFillets )
  viewManager->AddObject( Style( 1, RGB(0,0,255) ), pFillets, &pl );
if ( pChamfers )
  viewManager->AddObject( Style( 1, RGB(0,0,255) ), pChamfers, &pl );

// Пример запроса количества сегментов у построенных кривых
int segCount_Square   = pSquare->GetSegmentsCount();          // 4
int segCount_Fillets  = pFillets->GetSegmentsCount();         // 8
int segCount_Chamfers = pChamfers->GetSegmentsCount();        // 8
```

```
  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSquare );
  ::DeleteItem( pFillets );
  ::DeleteItem( pChamfers );
}
```

Рассмотрим пример построения твердого тела выдавливания с использованием образующей, показанной на рис. 5. Образующая является совокупностью из контура и двух окружностей. Контур состоит из 6 дуг окружностей и двух отрезков. Четыре дуги в составе контура построены в виде скруглений.

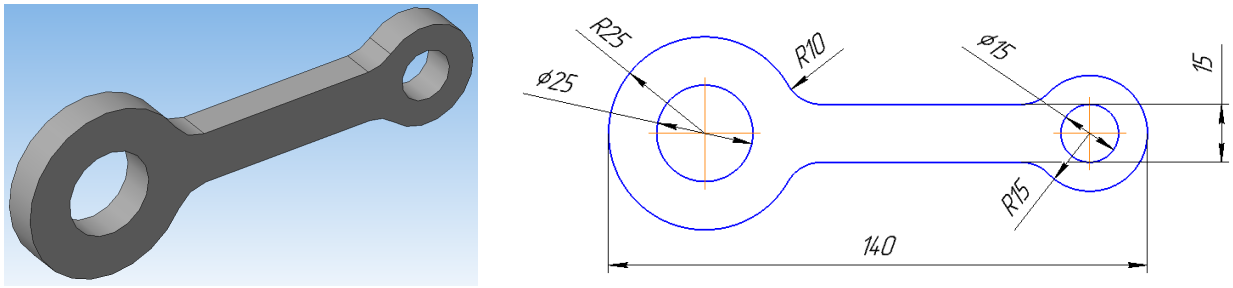


Рис. 5. Тело выдавливания на основе образующей, состоящей из трех кривых (контур MbContour и две окружности MbArc).

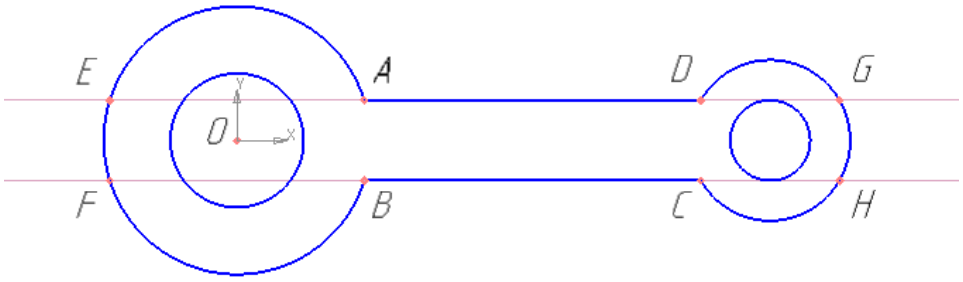


Рис. 6. Вспомогательные построения для нахождения концевых точек дуг окружностей для контура образующей. В качестве начала локальной СК образующей выбрана точка O.

Исходный текст для построения твердого тела (рис. 5) приведен в примере 2.5. Контур строится с применением вспомогательных построений, показанных на рис. 6. Пара параллельных вспомогательных прямых применяется для нахождения точек пересечения отрезков контура с окружностями радиусом 25 (точки A, B, E и F) и 15 (точки C, D, G, H). Половина из этих точек используется в качестве концевых точек для вычисления требуемых дуг окружностей. Порядок построения контура следующий: дуга AB (направление построения против часовой стрелки), отрезок BC (слева-направо), дуга CD (против часовой стрелки), отрезок DA (справа-налево).

Для повышения удобочитаемости пример 2.5 разделен на две части. Для построения кривых, входящих в состав образующей, предназначена отдельная функция CreateSketch. Эта функция в виде массива возвращает указатели на динамически создаваемые внешний контур и две внутренних окружности. Окружности также представляются в виде контуров, состоящих из одного сегмента (для удобства передачи функции, выполняющей построение тела выдавливания). Затем на базе образующей строится тело выдавливания. Построенные образующая и тело выдавливания показаны на рис. 7.

Пример 2.5. Использование контура со скруглениями в качестве образующей тела выдавливания (рис. 5,6,7).

```
#include "cur_contour.h"        // MbContour - контур
#include "cur_arc.h"            // MbArc - дуга эллипса/окружности
#include "cur_line.h"           // MbLine - прямая в двумерном пространстве
#include "cur_line_segment.h"   // MbLineSegment - класс двумерного отрезка
#include "mb_cross_point.h"     // MbCrossPoint - точка пересечения пары кривых
#include "surf_plane.h"         // MbPlane - плоскость
#include "solid.h"              // MbSolid - твердое тело
#include "creator.h"            // Классы для построения геометрических моделей
#include "alg_curve_fillet.h"   // Функции для построения скруглений и фасок
#include "action_point.h"       // Функции-алгоритмы для операций с точками
#include "action_solid.h"       // Функции- алгоритмы построения твердых тел

// ФУНКЦИЯ ПОСТРОЕНИЯ КРИВЫХ ДЛЯ ОБРАЗУЮЩЕЙ ТЕЛА ВЫДАВЛИВАНИЯ.
// Три кривые возвращаются в массиве в виде указателей на контуры.
void CreateSketch( RPArray<MbContour>& _arrContours )
{
  // Центры и радиусы окружностей, дуги которых входят в контур
  const MbCartPoint centerCircleLeft( 0, 0 );
  const MbCartPoint centerCircleRight( 100, 0 );
  const double RAD_LEFT  = 25;
  const double RAD_RIGHT = 15;

  // Вспомогательные горизонтальные прямые, содержащие отрезки контура
  // Значения горизонтальных координат могут быть любыми.
  MbLine* pLineTop = new MbLine(MbCartPoint( 0,  7.5), MbCartPoint( 10,  7.5 ) );
  MbLine* pLineBtm = new MbLine(MbCartPoint( 0, -7.5), MbCartPoint( 10, -7.5 ) );

  // Вычисление точек пересечения вспомогательных прямых с окружностями (рис.6)
  MbCrossPoint pntsEA[2];       // Пересечения верхней прямой и левой окружности
  MbCrossPoint pntsDG[2];       // Верхняя прямая и правая окружность
  MbCrossPoint pntsFB[2];       // Нижняя прямая и левая окружность
  MbCrossPoint pntsCH[2];       // Нижняя прямая и правая окружность
  ::LineCircle( *pLineTop, centerCircleLeft,  RAD_LEFT,  pntsEA );
  ::LineCircle( *pLineTop, centerCircleRight, RAD_RIGHT, pntsDG );
  ::LineCircle( *pLineBtm, centerCircleLeft,  RAD_LEFT,  pntsFB );
  ::LineCircle( *pLineBtm, centerCircleRight, RAD_RIGHT, pntsCH );

  // Упорядочение точек пересечения по горизонтали, чтобы было pnts[0].x < pnts[1].x
  if (pntsEA[0].p.x > pntsEA[1].p.x )
    std::swap(pntsEA[0], pntsEA[1] );
  if (pntsDG[0].p.x > pntsDG[1].p.x )
    std::swap(pntsDG[0], pntsDG[1] );
  if (pntsFB[0].p.x > pntsFB[1].p.x )
    std::swap(pntsFB[0], pntsFB[1] );
  if (pntsCH[0].p.x > pntsCH[1].p.x )
    std::swap(pntsCH[0], pntsCH[1] );

  // Дуги окружностей для контура
  MbArc* pArcAB = new MbArc(centerCircleLeft,  RAD_LEFT,  pntsEA[1].p, pntsFB[1].p, 1);
  MbArc* pArcCD = new MbArc(centerCircleRight, RAD_RIGHT, pntsCH[0].p, pntsDG[0].p, 1);

  // Отрезки для контура
  MbLineSegment* pSegBC = new MbLineSegment(pntsFB[1].p, pntsCH[0].p);
  MbLineSegment* pSegDA = new MbLineSegment(pntsDG[0].p, pntsEA[1].p );

  // Построение контура
  MbContour* pContour = new MbContour();
  pContour->AddSegment( pArcAB );
  pContour->AddSegment( pSegBC );
```

```
  pContour->AddSegment( pArcCD );
  pContour->AddSegment( pSegDA );

  // Построение скруглений во всех узлах контура (содержит четыре узла)
  pContour->Fillet( 10 );

  // Построение внутренних окружностей образующей и преобразование их в контуры
  const double INNER_RAD_LEFT  = 12.5;
  const double INNER_RAD_RIGHT = 7.5;
  MbArc* pCircleLeft  = new MbArc(centerCircleLeft,  INNER_RAD_LEFT);
  MbArc* pCircleRight = new MbArc(centerCircleRight, INNER_RAD_RIGHT);
  MbContour* pContourCircleLeft  = new MbContour( *pCircleLeft, true );
  MbContour* pContourCircleRight = new MbContour( *pCircleRight, true );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pLineTop );
  ::DeleteItem( pLineBtm );
  ::DeleteItem( pArcAB );
  ::DeleteItem( pArcCD );
  ::DeleteItem( pSegBC );
  ::DeleteItem( pSegDA );

  // Сохранение указателей на возвращаемые контуры в выходном массиве
  // У возвращаемых объектов счетчик ссылок уменьшать не надо - это должно быть
  // сделано в месте вызова, после того, как эти объекты больше не будут нужны.
  _arrContours.push_back( pContour );
  _arrContours.push_back( pContourCircleLeft );
  _arrContours.push_back( pContourCircleRight );
}


// ОБРАБОТЧИК ПОЛЬЗОВАТЕЛЬСКИЙ КОМАНДЫ ТЕСТОВОГО ПРИЛОЖЕНИЯ
void MakeUserCommand0()
{
  MbPlacement3D pl;    // Локальная СК (по умолчанию совпадает с мировой СК)
  // СОЗДАНИЕ КОНТУРОВ ДЛЯ ОБРАЗУЮЩЕЙ
  RPArray<MbContour> arrContours;
  CreateSketch( arrContours );

  // Отображение образующей (в плоскости XY глобальной СК)
  for (int i = 0; i<arrContours.size(); i++)
    viewManager->AddObject( Style( 1, RGB(0,0,255) ), arrContours[i], &pl );

  // ПОСТРОЕНИЕ ТЕЛА ВЫДАВЛИВАНИЯ
  // Образующая размещается на плоскости XY глобальной СК.
  // Важное замечание: объект-плоскость должен создаваться динамически,
  // поскольку он продолжает использоваться в объекте-твердом теле после
  // выхода из данной функции.
  MbPlane* pPlaneXY = new MbPlane( MbCartPoint3D(0,0,0), MbCartPoint3D(1,0,0),
                                   MbCartPoint3D(0,1,0) );
  // Объект, хранящий параметры образующей
  MbSweptData sweptData( *pPlaneXY, arrContours );
  // Направляющий вектор для операции выдавливания
  MbVector3D dir( 0, 0, -1 );

  // Параметры операции выдавливания, задающие свойства тела для построения:
  // расстояние выдавливания в прямом и в обратном направлении вдоль
  // направляющего вектора
  const double HEIGHT_FORWARD = 10.0, HEIGHT_BACKWARD = 0.0;
  ExtrusionValues extrusionParam( HEIGHT_FORWARD, HEIGHT_BACKWARD );

  // Служебный объект для именования элементов модели твердого тела
```

```
  MbSNameMaker operNames( ct_CurveExtrusionSolid, MbSNameMaker::i_SideNone, 0 );
  PArray<MbSNameMaker> cNames( 0, 1, false );

  // Построение твердого тела выдавливания
  MbSolid* pSolid = NULL;
  MbResultType res = ::ExtrusionSolid( sweptData, dir, NULL, NULL, false,
                                       extrusionParam, operNames, cNames, pSolid);

  // Отображение построенного тела
  if (res == rt_Success)
  {
    // Смещение тела по оси Y, чтобы при отображении оно не накладывалось на образующую
    pSolid->Move(MbVector3D(0,80,0));
    viewManager->AddObject( Style(1, LIGHTGRAY), pSolid );
  }

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSolid );
  ::DeleteItem( pPlaneXY );
  for (int i = 0; i<arrContours.size(); i++)
    ::DeleteItem( arrContours[i] );
}
```

В примере 2.5 в функции MakeUserCommand0() обратите внимание на динамическое создание объекта-плоскости pPlaneXY. Это объект класса MbPlane, поддерживающий механизм подсчета ссылок (т.е. унаследован от MbRefItem). Целесообразно придерживаться следующего подхода: если некоторый объект ядра используется в составе других объектов, то следует создавать этот используемый объект динамически и при выходе из области видимости динамического объекта вызвать для него ::DeleteItem. Тот факт, что объект pPlaneXY продолжает использоваться после выхода из функции MakeUserCommand0(), не вполне очевиден: как часть объекта с параметрами образующей sweptData он запоминается во внутреннем журнале построения твердого тела pSolid.

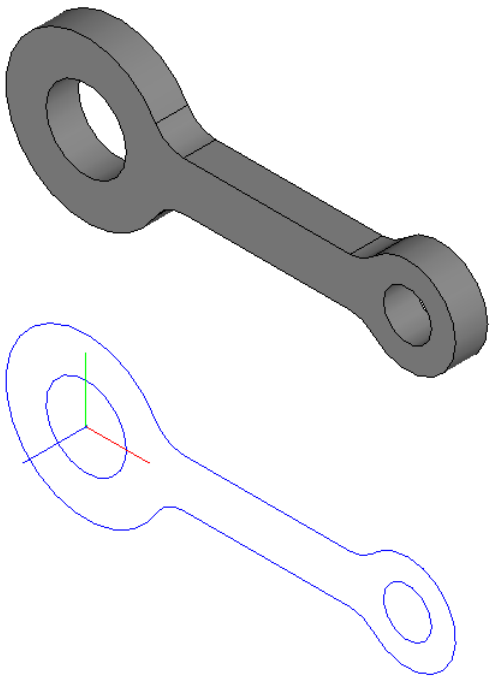


Рис. 7. Результаты, отображаемые в окне тестового приложения при выполнении примера 2.5.

2.4 Задания

1) Выполните примеры 2.1-2.5. С помощью окон свойств в тестовом приложении выясните, какие объекты входят в геометрическую модель в каждом из этих примеров (контур, ломаная линия, поверхность, твердое тело). Для твердого тела из примера 2.5 в окне свойств найдите журнал построения, а в нем – контур из 8 сегментов, соответствующий внешнему контуру образующей (рис. 5). Последовательность вызова окон геометрической модели показана на рис. 8: тело, журнал построения, тело выдавливания, сечение 1.

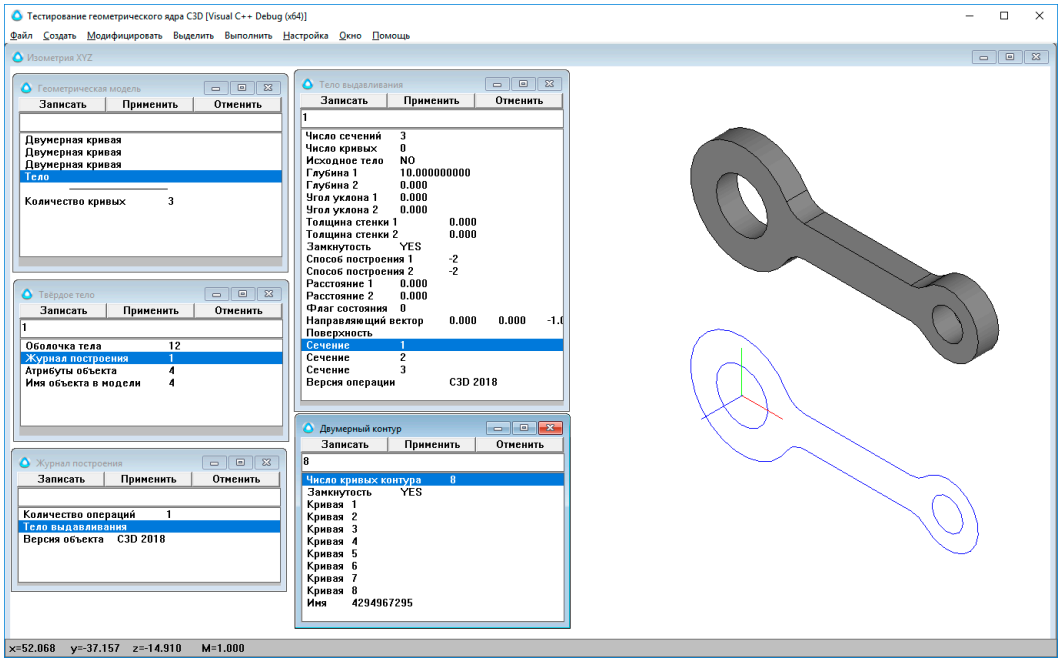


Рис. 8. Окна свойств геометрической модели, которые надо вызвать для просмотра свойств внешнего контура образующей (задание 1).

2) На основе примера 2.5 постройте твердое тело выдавливания на базе образующей из двух контуров и четырех окружностей:

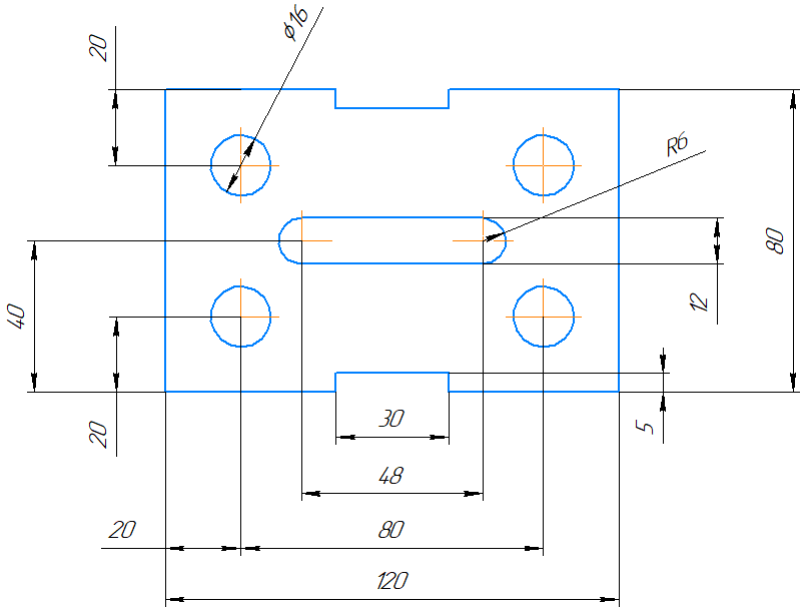


Рис. 9. Образующая для задания 2 состоит из 6 кривых: внешняя ломаная линия, 4 окружности и внутренний контур.

Для решения этой задачи можно модифицировать функцию CreateSketch из примера 2.5 так, чтобы эта функция возвращала образующую, показанную на рис. 9, в виде массива из 6 контуров. В отладочных целях образующую можно формировать постепенно: сначала построить твердое тело на базе образующей из одного внешнего контура, а затем дополнить ее внутренним контуром и окружностями.

3) Выполните построение твердого тела выдавливания с образующей, состоящей из внешнего контура и трех внутренних окружностей (рис. 10).

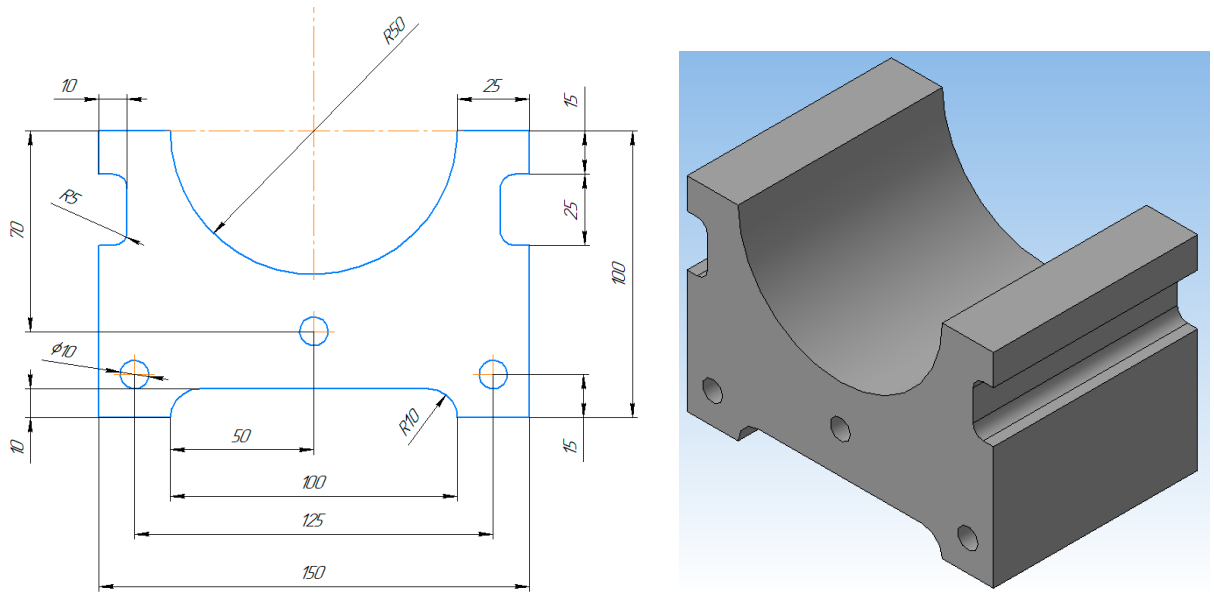


Рис. 10. Образующая и твердое тело выдавливания (задание 3).

При построении контура образующей начало ее локальной СК можно выбрать в левом нижнем углу. Допустим, что вершины ломаной для образующей указываются в порядке **A**, **B**, **C** и т.д. (против часовой стрелки, см. рис. 11). В таком случае в качестве координат точек можно указать значения: (25, 100), (0, 100), (0, 85), (10, 85), (10, 60), (0, 60), (0, 0), (25, 0), (25, 10), (125, 10), (125, 0), (150, 0), (150, 60), (140, 60), (140, 85), (150, 85), (150, 100), (125, 100). При построении скруглений количество сегментов может изменяться (рис. 11), поэтому при вызовах MbContour::FilletTwoSegments следует указать подходящие значения индексов.

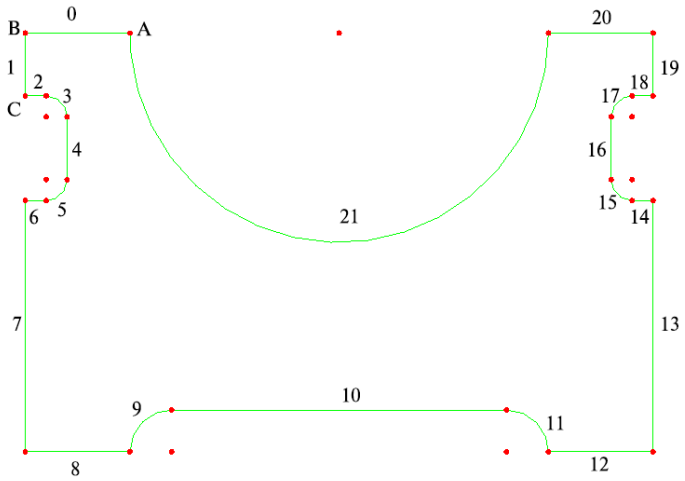


Рис. 11. Индексы сегментов контура образующей после построения скруглений (при построении по точкам с координатами, указанными в задании 3).

3. Сплайны

В C3D реализованы 4 класса для представления сплайновых кривых: кубический сплайн (MbCubicSpline), кубический сплайн Эрмита (MbHermit), кривая Безье (MbBezier) и NURBS-кривая MbNurbs. Кубические сплайны удобны для построения кривых, проходящих через известное множество точек (для решения задач интерполяции). Вычислительная реализация кубических сплайнов относительно проста и эффективна, но для некоторых задач геометрического моделирования они оказываются не слишком удобны: требуется указывать все контрольные точки в начале построения сплайна (это оказывается не всегда возможно в процессе интерактивного построения кривых), необходимо задавать производные в контрольных точках, построенную кривую не удается локально модифицировать (например, при смещении одной из контрольных точек может измениться форма всех сегментов сплайна). Для подобных применений, когда форма кривой точно не известна, а подбирается пользователем с учетом трудно формализуемых критериев (например, эстетических), удобнее применять другие типы сплайнов, в частности, кривые Безье.

Кубические сплайны и кривые Безье позволяют решать задачи двух видов – построение кривых по точкам и построение кривой подбором формы. В геометрическом моделировании есть еще одна важная проблема – унифицированное представление геометрических объектов. Эта проблема решается с использованием сплайнов NURBS. В виде таких сплайнов возможно точно представить все кривые, применяемые в геометрическом моделировании (прямые, окружности, эллипсы и любые другие кривые произвольной формы). Такие сплайны широко применяются в алгоритмах для автоматических построений, например, для описания ребер твердых тел, вычисляемых как кривые пересечения произвольных поверхностей. В силу важности представления в виде NURBS-сплайнов, в родительском классе кривой MbCurve имеется набор методов для преобразование любой кривой в эту универсальную форму.

Сплайны состоят из однотипных сегментов, поэтому их классы в C3D унаследованы от родительского класса MbPolyCurve (кривая, построенная по набору точек). В этом классе предусмотрены атрибуты для хранения набора контрольных точек и набор метод для работы с сегментами кривой. (Класс MbPolyCurve и унаследованный от него класс MbPolyline рассматривались в Работе №2.) Кроме набора контрольных точек, в классах сплайнов предусмотрено хранение дополнительных параметров (например, значения производных в контрольных точках) и реализованы методы, зависящие от типа сплайна.

3.1 MbCubicSpline – кубический сплайн

Кубический сплайн задается множеством контрольных точек и множеством вторых производных в этих точках. Сплайн проходит через все контрольные точки, а значения производных позволяют управлять формой сплайна. Все сегменты сплайна являются полиномами третьей степени, коэффициенты которых зависят от значений параметра t_i в контрольных точках $\mathbf{p}_i$.

Ниже приведен фрагмент интерфейса класса MbCubicSpline (файл cur_cubic_spline.h), с перечнем атрибутов, конструкторами класса и методами для получения свойств сплайна. Оставшаяся часть интерфейса содержит методы, унаследованные от базового MbCurve и родительского MbPolyCurve.

```
class MbCubicSpline : public MbPolyCurve {
protected :
  SArray<double>   tList;       // Массив значений параметра t в контрольных точках.
  SArray<MbVector> vectorList;  // Массив вторых производных в контрольных точках.

public :
```

```
  // КОНСТРУКТОРЫ
  // Построение сплайна по заданной кривой
  MbCubicSpline( const MbCurve& other );
  // Построение по контрольным точкам с указанием признака замкнутости сплайна
  MbCubicSpline( const SArray<MbCartPoint>& points, bool cls );
  // Построение с указанием контрольных точек, массива вторых производных и
  // признака замкнутости
  MbCubicSpline( const SArray<MbCartPoint>& points,
                 const SArray<MbVector>& seconds, bool cls );
  // Построение по контрольным точкам, значениям параметра t в контрольных точках и
  // признака замкнутости
  MbCubicSpline( const SArray<MbCartPoint>& points,
                 const SArray<double>& params, bool cls );
  // Построение по контрольным точкам, значениям параметра t, значениям вторых
  // производных в контрольных точках и признака замкнутости
  MbCubicSpline( const SArray<MbCartPoint>& points,
                 const SArray<MbVector>& seconds,
                 const SArray<double>& params, bool cls );

  // ИНФОРМАЦИОННЫЕ МЕТОДЫ
  // Получение размера массива параметров.
  int  GetTListCount() const { return tList.Count(); }
  // Получение массив значений параметров в контрольных точкаxx.
  void GetTList( SArray<double> & params ) const { params = tList; }
  // Получить значение параметра для контрольной точки с индексом i.
  const double& GetTList( size_t i ) const { return tList[i]; }
  // Получение количества элементов в массиве векторов вторых производных
  int  GetVectorListCount() const { return vectorList.Count(); }
  // Получение массива вторых производных в контрольных точках.
  void GetVectorList( SArray<MbVector>& vectors ) const { vectors = vectorList; }
  // Получить вектор второй производной в контрольной точке с индексом i.
  const MbVector& GetVectorList( int i ) const { return vectorList[i]; }
  // Получение значений вторых производных на концах незамкнутого сплайна
  void CreateEndS( MbVector&, MbVector& );

  // Другие методы MbCubicSpline, в т.ч. унаследованные от MbCurve и MbPolyCurve
};
```

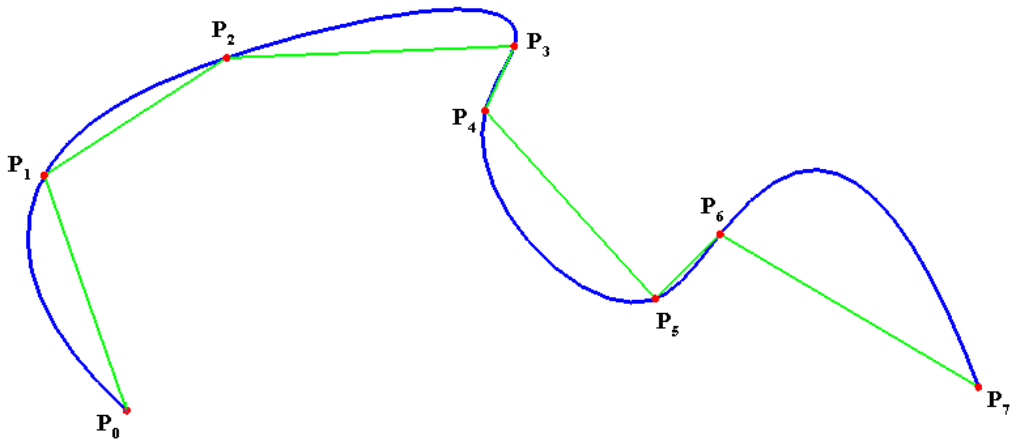


Рис. 12. Кубический сплайн и ломаная линия, построенные по совпадающему набору контрольных точек (пример 3.1).

Пример 3.1. Построение кубического сплайна и ломаной (рис. 12).

```
#include "cur_polyline.h"              // MbPolyline - Ломаная линия
```

```
#include "cur_cubic_spline.h"          // MbCubicSpline - кубический сплайн в 2D

void MakeUserCommand0()
{
  // Контрольные точки сплайна
  SArray<MbCartPoint> arrPnts(8);
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(-7, 20) );
  arrPnts.Add( MbCartPoint(8.5, 30) );
  arrPnts.Add( MbCartPoint(33, 31) );
  arrPnts.Add( MbCartPoint(30.5,25.5) );
  arrPnts.Add( MbCartPoint(45, 9.5) );
  arrPnts.Add( MbCartPoint(50.5, 15) );
  arrPnts.Add( MbCartPoint(72.5, 2) );

  // Построение незамкнутого кубического сплайна по контрольным точкам
  MbCubicSpline* pSpline = new MbCubicSpline( arrPnts, false );

  // Построение ломаной линии по контрольным точкам сплайна
  MbPolyline* pPolyline = new MbPolyline( arrPnts, false );

  // Отображение контрольных точек
  MbPlacement3D pl; // Локальная СК (по умолчанию совпадает с мировой СК)
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPnts[i], &pl );
  // Отображение ломаной
  viewManager->AddObject( Style(2, RGB(0,255,0)), pPolyline, &pl );
  // Отображение сплайна
  viewManager->AddObject( Style(3, RGB(0,0,255)), pSpline, &pl );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSpline );
  ::DeleteItem( pPolyline );
}
```

На рис. 12 приведены кубический сплайн и ломаная, построенные по 8 контрольным точкам. Соответствующий программный текст представлен в примере 3.1. Для этого сплайна параметр $t \in [0, 7]$. В отличие от ломаной линии, параметрическая длина сегментов сплайна не всегда равна 1. Значения параметров t_i в контрольных точках могут вычисляться автоматически или задаваться в качестве исходных данных для расчета сплайна, в зависимости от используемого конструктора MbCubicSpline. Изменяя положение контрольных точек, значения параметров и вторых производных можно изменять форму сплайна. Однако данный тип сплайна не слишком удобен для контроля формы и для обеспечения гладкого сопряжения с другими кривыми в концевых точках. Наилучшим образом MbCubicSpline подходит для интерполяции набора известных точек гладкой кривой.

Для сопряжения кубического сплайна с другими кривыми можно применять построение скруглений для контуров. На рис. 13 показано построение сопряжений сплайна с двумя отрезками (пример 3.2). Угловые точки были заменены на гладкое сопряжение дугой окружности, однако при этом составная кривая перестала проходить через концевые контрольные точки сплайна.

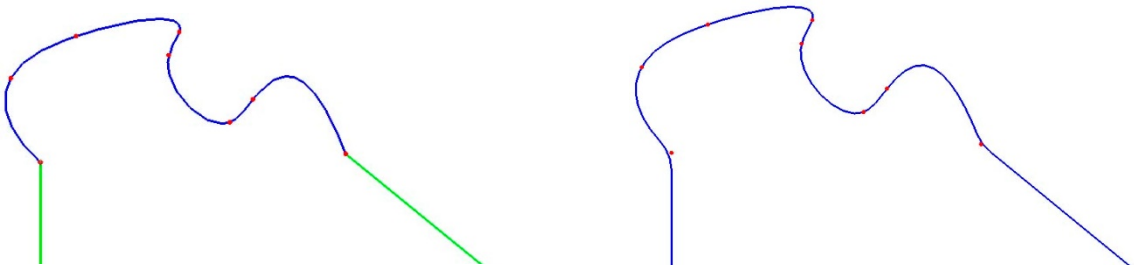


Рис. 13. Сопряжение кубического сплайна с отрезками с использованием скруглений. (пример 3.2).

Пример 3.2. Применение скруглений для сопряжения кубического сплайна с другими кривыми (рис. 13).

```
#include "cur_line_segment.h"          // MbLineSegment - класс двумерного отрезка
#include "cur_cubic_spline.h"          // MbCubicSpline - кубический сплайн в 2D
#include "cur_contour.h"               // MbContour - контур

void MakeUserCommand0()
{
  // Контрольные точки сплайна
  SArray<MbCartPoint> arrPnts(8);
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(-7, 20) );
  arrPnts.Add( MbCartPoint(8.5, 30) );
  arrPnts.Add( MbCartPoint(33, 31) );
  arrPnts.Add( MbCartPoint(30.5, 25.5) );
  arrPnts.Add( MbCartPoint(45, 9.5) );
  arrPnts.Add( MbCartPoint(50.5, 15) );
  arrPnts.Add( MbCartPoint(72.5, 2) );

  // Построение незамкнутого кубического сплайна по контрольным точкам
  MbCubicSpline* pSpline = new MbCubicSpline( arrPnts, false );

  // Два отрезка, соединяющихся со сплайном в концевых точках
  MbLineSegment* pSeg1 = new MbLineSegment(MbCartPoint(0, -50), MbCartPoint(0, 0));
  MbLineSegment* pSeg2 = new MbLineSegment(MbCartPoint(72.5,2), MbCartPoint(130,-45));

  // Построение контура из отрезков и сплайна
  MbContour* pContour = new MbContour();
  pContour->AddSegment( pSeg1 );
  pContour->AddSegment( pSpline );
  pContour->AddSegment( pSeg2 );

  // Скругление в угловых точках контура
  const int FILLET_RAD = 10;
  ptrdiff_t idxSegLeft = 0;
  pContour->FilletTwoSegments( idxSegLeft, FILLET_RAD );
  ptrdiff_t idxSegRight = pContour->GetSegmentsCount()-2;
  pContour->FilletTwoSegments( idxSegRight, FILLET_RAD );

  // Отображение контрольных точек
  MbPlacement3D pl; // Локальная СК (по умолчанию совпадает с мировой СК)
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPnts[i], &pl );
  // Отображение контура
  viewManager->AddObject( Style(3, RGB(0,0,255)), pContour, &pl );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSpline );
```

```
  ::DeleteItem( pSeg1 );
  ::DeleteItem( pSeg2 );
  ::DeleteItem( pContour );
}
```

3.2 MbHermit – кубический сплайн Эрмита

Кубический сплайн Эрмита задается опорными точками, параметрами и первыми производными. Аналогично кубическому сплайну, сплайн Эрмита проходит через все контрольные точки и может применяться для задач интерполяции. Возможность задать первые производные позволяет удобнее контролировать форму сплайна по сравнению с классом MbCubicSpline. На форму сплайна MbHermit влияет не только направление, но и величина первой производной в контрольных точках, определяющая, «насколько сильно» сегмент кривой «притягивается» к соответствующей контрольной точке.

Интерфейс класса MbHermit аналогичен MbCubicSpline, однако он дополнен еще несколькими методами, рассчитанными на модификацию сплайна посредством добавления в него новых точек или изменения существующих. Эти методы показаны ниже во фрагменте интерфейса MbHermit (заголовочный файл cur_hermit.h)

```
class MbHermit : public MbPolyCurve {
protected :
  SArray<MbVector> vectorList;   // Массив первых производных в контрольных точках
  SArray<double>   tList;        // Массив значений параметров в контрольных точках

public :
  // КОНСТРУКТОРЫ
  // Конструкторы и методы отложенной инициализации аналогичны MbCubicSpline

  // МЕТОДЫ МОДИФИКАЦИИ СПЛАЙНА
  // Добавление контрольной точки в конец массива
  virtual void AddPoint( const MbCartPoint& pnt );
  // Вставка точки в заданную позицию массива контрольных точек
  virtual void InsertPoint( int index, const MbCartPoint& pnt );
  // Вставка точки с заданным значением параметра или замена существующей,
  // если вставляемая точка лежит в пределах допуска вблизи существующей.
  virtual void InsertPoint(double t, const MbCartPoint& pnt, double xEps, double yEps);
  // Вставка/замена точки с указанием производной
  virtual void InsertPoint( double t, const MbCartPoint& pnt, const MbVector& v,
                            double xEps, double yEps );
  // Замена точки с заданным индексом в массиве контрольных точек
  virtual void ChangePoint( int index, const MbCartPoint& pnt );
  // Удаление точки с заданным индексом из массива контрольных точек
  virtual void RemovePoint( int index );
  // Добавление набора точек и значений параметров t в конец кривой.
  bool AddPoints( SArray<double>& params, SArray<MbCartPoint>& points );
  // Вставка набора точек и значений параметров в начало кривой.
  bool InsertPoints( SArray<double>& params, SArray<MbCartPoint>& points );
  // Задать значения производных во всех контрольных точках
  bool SetTangentVectors( const SArray<MbVector>& tVectors );
  // Получение максимального индекса массива параметров слева от точки с параметром t.
  int  GetIndex( double t ) const;

  // Другие методы MbHermit, в т.ч. унаследованные от MbCurve и MbPolyCurve
};
```

Рассмотрим построение сплайна Эрмита и его последующую модификацию для сопряжения сплайна с отрезками аналогично рис. 13. В примере 3.3 создается объект MbHermit,

построенный по тем же контрольным точкам, которые использовались в примере 3.2. После построения в концевых точках сплайна производится модификация значений производных (с учетом того, что направление отрезков известно – один вертикальный, второй направлен под углом –45 градусов к горизонтальной оси). Результат показан в центре на рис. 14(б).

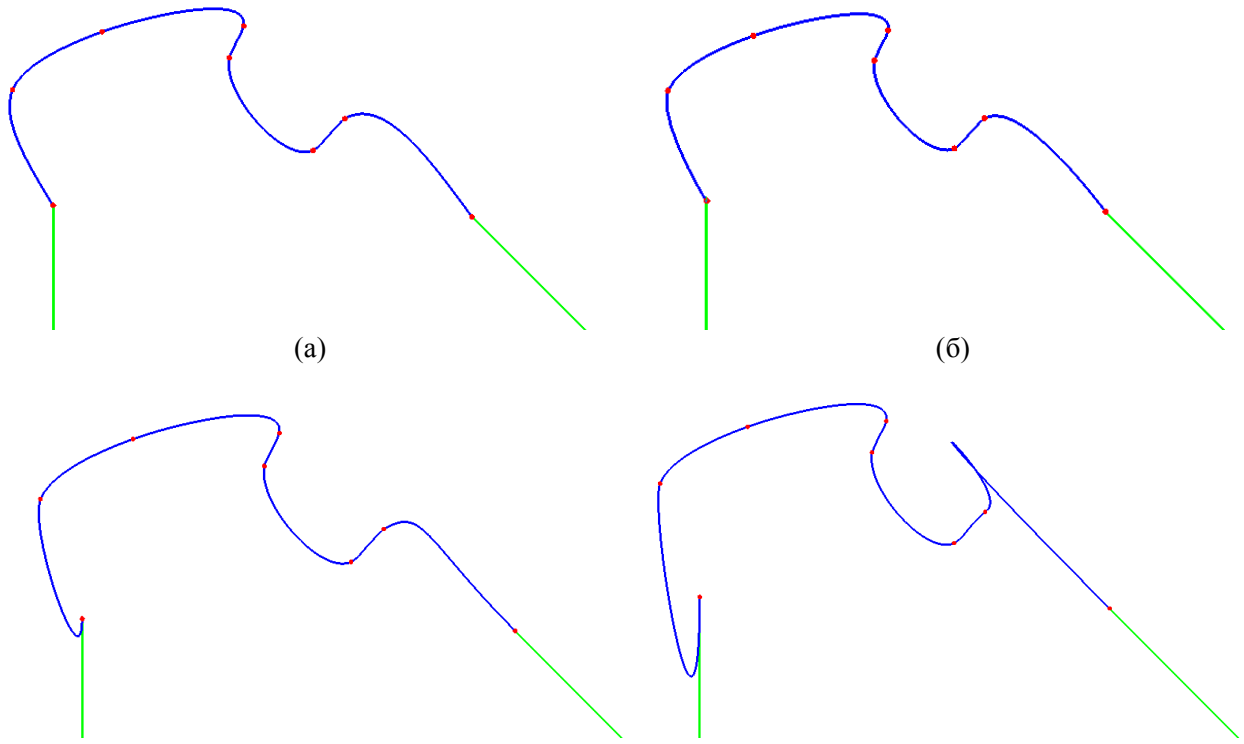


Рис. 14. Сопряжение кубического сплайна Эрмита с отрезками посредством указания значений производных в контрольных точках. (а) Исходный сплайн и пара отрезков с вершинами в концевых точках сплайна. (б) В концевых точках сплайна в качестве производных заданы касательные вектора в направлении отрезков. (в, г) Значения производных в концевых точках увеличены.

Пример 3.3. Модификация кубического сплайна Эрмита (рис. 14).

```
#include "cur_line_segment.h"          // MbLineSegment - отрезок
#include "cur_hermit.h"                // MbHermit - кубический сплайн Эрмита

void MakeUserCommand0()
{
  // Контрольные точки сплайна
  SArray<MbCartPoint> arrPnts(8);
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(-7, 20) );
  arrPnts.Add( MbCartPoint(8.5, 30) );
  arrPnts.Add( MbCartPoint(33, 31) );
  arrPnts.Add( MbCartPoint(30.5, 25.5) );
  arrPnts.Add( MbCartPoint(45, 9.5) );
  arrPnts.Add( MbCartPoint(50.5, 15) );
  arrPnts.Add( MbCartPoint(72.5, 2) );

  // Построение незамкнутого кубического сплайна Эрмита
  MbHermit* pSpline = new MbHermit( arrPnts, false );

  // Два отрезка, соединяющихся со сплайном в концевых точках
  // pSeg1 - вертикальный отрезок
  // pSeg2 - отрезок под углом -45 градусов к горизонтальной оси
```

```
  MbLineSegment* pSeg1 = new MbLineSegment(MbCartPoint(0, -50), arrPnts[0]);
  MbLineSegment* pSeg2 = new MbLineSegment(arrPnts[7], MbVector(1, -1), 0, 50 );

  // Модификация контрольных точек сплайна: положение не изменяется
  // (указываются координаты существующих концевых точек сплайна),
  // но задаются новые значения производных.
  const double TH = 0.01;       // Допуск по расстоянию, в пределах которого
                                // точки считаются совпадающими
  pSpline->InsertPoint( 0.0, arrPnts[0], MbVector(0, -1), TH, TH );
  pSpline->InsertPoint( 7.0, arrPnts[7], MbVector(1, -1), TH, TH );

  // Отображение контрольных точек
  MbPlacement3D pl;
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPnts[i], &pl );
  // Отображение сплайна
  viewManager->AddObject( Style(3, RGB(0,0,255)), pSpline, &pl );
  // Отображение отрезков
  viewManager->AddObject( Style(3, RGB(0,255,0)), pSeg1, &pl );
  viewManager->AddObject( Style(3, RGB(0,255,0)), pSeg2, &pl );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSpline );
  ::DeleteItem( pSeg1 );
  ::DeleteItem( pSeg2 );
}
```

Существенных изменений между исходными кривыми и кривыми на рис. 14(а) и (б) не заметно, хотя в концевых точках сплайна производные имеют требуемые значения. При увеличении значений производных (с сохранением прежнего направления) можно добиться приближения сегментов сплайна к направлению отрезков. Однако, при этом форма сплайна может существенно измениться с образованием нежелательных изгибов и петель. Кривая, показанная на рис. 14(в) построена с использованием следующих вызовов для модификации концевых точек сплайна:

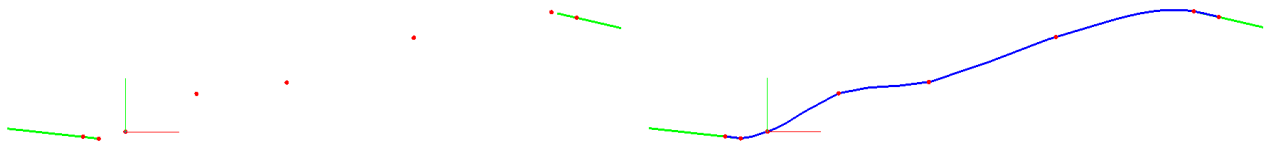
```
pSpline->InsertPoint( 0.0, arrPnts[0], MbVector(0, -30), TH, TH );
pSpline->InsertPoint( 7.0, arrPnts[7], MbVector(30, -30), TH, TH );
```

Сплайн, показанный на рис 12(г), был построен с использованием в качестве производных в концевых точках векторов `MbVector(0, -100)` и `MbVector(100, -100)`.

Для управления формой сплайна Эрмита удобнее совместно изменять местоположение контрольных точек и значения производных. Контрольные точки желательно плотнее располагать вблизи участков сплайна, где существенно изменяется его направление. На участках, где направление сплайна примерно постоянно, контрольные точки целесообразно располагать равномерно. При сопряжении сплайна с другими кривыми в окрестностях концевых точек можно специально предусмотреть сегменты, направление которых будет близко к направлению соседних кривых в точках сопряжения. Для демонстрации этого приема рассмотрим построение сплайна Эрмита по контрольным точкам, показанным на рис. 15. Сплайн применяется для построения сопряжения между двумя прямолинейными участками дороги и проходит через несколько промежуточных точек, положение которых зафиксировано на перекрестках с второстепенными дорогами.

Рис. 15. Спутниковый снимок (maps.google.ru) с фрагментом дороги (1590x520 пикселей). На нем отмечены прямолинейные участки дороги для сопряжения сплайном Эрмита. На перекрестках размещены пять промежуточных контрольных точек, через которые должен проходить сплайн.

Контрольные точки для построения сплайна по данным рис. 15, показаны на рис. 16(а). На нем также изображены сопрягаемые прямолинейные отрезки. Кроме 5 контрольных точек, соответствующих круговым меткам на рис. 15, добавлены еще 3 контрольных точки. Они обеспечивают добавление к сплайну концевых сегментов, начальная и конечная точки которых лежат на двух отрезках (с правой стороны достаточно одной дополнительной точки, чтобы сегмент сплайна был направлен примерно параллельно отрезку). Координаты контрольных точек заданы относительно точки O. Построенный сплайн показан на рис. 16(б), соответствующий программный текст приведен в примере 3.4.



(а) (б)

Рис. 16. (а) Расположение контрольных точек для построения сплайна и сопрягаемых прямолинейных отрезков. (б) Построенный сплайн MbHermit.

Пример 3.4. Кубический сплайн Эрмита (рис. 15).

```
#include "cur_line_segment.h"          // MbLineSegment - отрезок
#include "cur_hermit.h"                // MbHermit - кубический сплайн Эрмита

void MakeUserCommand0()
{
  // Контрольные точки сплайна (соответствуют координатам пикселей на
  // спутниковом снимке). Две точки в начале и одна в конце массива попадают на
  // отрезки, для сопряжения которых строится сплайн.
  std::vector<MbCartPoint> stdArrPnts = { {-106, -12}, {-67, -17}, { 0, 0 }, { 178, 95 },
                                          { 404, 123}, {722, 235}, {1067, 299}, {1130, 285} };
  SArray<MbCartPoint> arrPnts( stdArrPnts );

  // Построение незамкнутого кубического сплайна Эрмита
  MbHermit* pSpline = new MbHermit( arrPnts, false );

  // Два отрезка, изображающие прямолинейные участки дороги.
  // Координаты концевых точек, как и контрольных точек сплайна,
  // получены измерением координат пикселей на исходном изображении.
  MbLineSegment* pSeg1 = new MbLineSegment(MbCartPoint(-296, 8),
                                           MbCartPoint(-67, -17));
```

```
  MbLineSegment* pSeg2 = new MbLineSegment(MbCartPoint(1081, 296),
                                           MbCartPoint(1242, 259));

  // Вектора, параллельные отрезкам pSeg1 и pSeg2
  MbDirection dir1 = pSeg1->GetDirection();
  MbDirection dir2 = pSeg2->GetDirection();

  // Расчет производных в концевых точках сплайна: направлены параллельно отрезкам,
  // величина задана для "притяжения" сегмента сплайна к отрезку.
  MbVector vect1( dir1.ax, dir1.ay );
  MbVector vect2( dir2.ax, dir2.ay );
  vect1 *= 10;
  vect2 *= 10;

  // Модификация производных в концевых контрольных точках сплайна
  const double TH = 0.01;       // Допуск по расстоянию, в пределах которого
                                // точки считаются совпадающими
  pSpline->InsertPoint( 0.0, arrPnts[0], vect1, TH, TH );
  pSpline->InsertPoint( 7.0, arrPnts[7], vect2, TH, TH );

  // Отображение контрольных точек
  MbPlacement3D pl;
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPnts[i], &pl );
  // Отображение сплайна
  viewManager->AddObject( Style(3, RGB(0,0,255)), pSpline, &pl );
  // Отображение отрезков
  viewManager->AddObject( Style(3, RGB(0,255,0)), pSeg1, &pl );
  viewManager->AddObject( Style(3, RGB(0,255,0)), pSeg2, &pl );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSpline );
  ::DeleteItem( pSeg1 );
  ::DeleteItem( pSeg2 );
}
```

Сплайн, показанный на рис. 16(б), при наложении на спутниковый снимок оказывается неточным представлением криволинейного участка дороги (рис. 17). Для того, чтобы добиться лучшего совпадения сплайна с видимой кривой, необходимо добавить контрольные точки – разместить их в точках перегиба этой кривой. Эти три точки показаны на рис. 18 квадратными маркерами. Координаты этих точек (267, 120), (490, 131) и (880, 305). Если добавить их в качестве контрольных точек сплайна с индексами 4, 6 и 8, то в результате будет построен сплайн, показанный на рис. 18. Таким образом, для уточнения желаемой формы сплайна потребовалось увеличить количество контрольных точек, располагая их вдоль кривой с учетом особенностей требуемой формы.

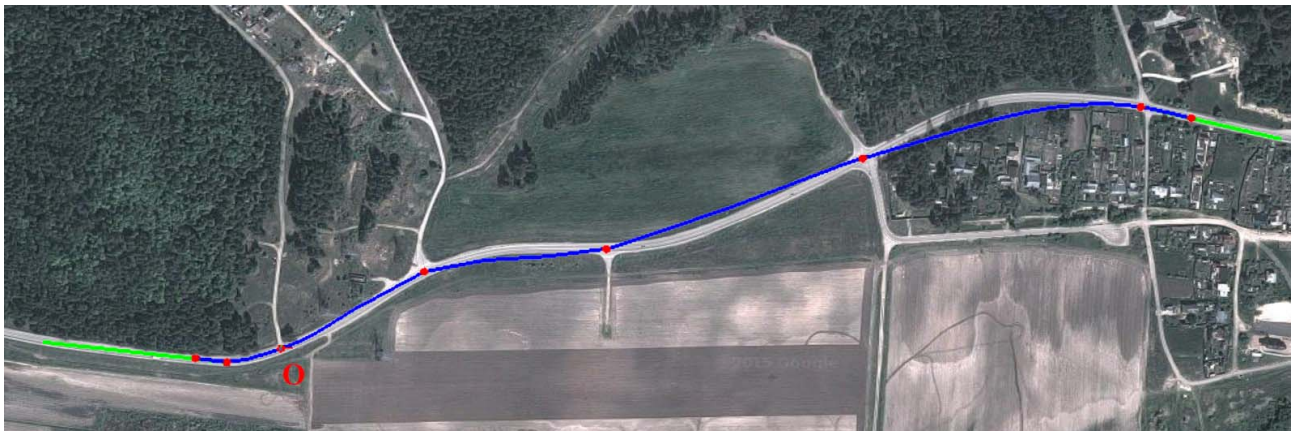


Рис. 17. Сплайн Эрмита, построенный по 8 контрольным точкам.

Рис. 18. Сплайн Эрмита, построенный по 11 контрольным точкам.

3.3 MbBezier – кривая Безье

Рассмотренные выше кубические сплайны удобны для решения задач, когда требуется построить кривую, проходящую через набор известных точек. Однако форму таких сплайнов контролировать сложно, поскольку связь формы кривой с направлением и величиной производных в контрольных точках не очевидна и задание параметров сплайна оказывается непростой задачей.

Сплайны Безье предоставляют более простой способ контроля формы кривой, но имеют свои особенности. Они уже не обязательно проходят через все контрольные точки. Ломаная, соединяющая контрольные точки, называется контрольной ломаной кривой Безье. Форма этой ломаной определяет форму кривой. Связь между ними удовлетворяет известному набору правил[1]:

- Основа формы кривой повторяет очертания контрольной ломаной.
- Первая и последняя точки кривой совпадают с концевыми точками контрольной ломаной.
- Касательные векторы в концах кривой по направлению совпадают с первым и последним сегментами контрольной ломаной.
- Сплайн лежит внутри выпуклой оболочки контрольных точек (внутри наибольшего многоугольника, построенного по этим точкам).

[1] Математическое описание сплайнов, в т.ч. кривых Безье и NURBS, можно найти в книгах: Голованов Н.Н. Геометрическое моделирование. М.: ИНФРА-М, 2016 и Роджерс Д., Адамс Дж. Математические основы машинной графики. М.: Мир, 2001.

- Кривая Безье пересекает любую прямую не чаще, чем многоугольник, построенный по ее контрольным точкам.

В целом, кривая Безье менее склонна к образованию петель и изгибов, чем ранее рассмотренные сплайны. Управляя формой ломаной, можно контролировать форму кривой без явного указания значений производных в контрольных точках.

Кривые Безье могут иметь различный порядок, он связан с тем, полиномы какой степени используются в параметрическом представлении этой кривой. В классе MbBezier (заголовочный файл cur_bezier.h) реализованы кубические кривые Безье. Для построения кривой Безье третьей степени необходимы 4 контрольных точки, задающие контрольную ломаную. При использовании класса MbBezier обычно строятся сплайны с указанием большего числа контрольных точек. В таком случае кривая MbBezier строится как составная кривая, состоящая из гладко соединяющихся кубических кривых Безье.

На рис. 19 показаны 4 кубических сплайна Безье, их контрольные точки и контрольные ломаные. Местоположение пары контрольных точек изменяется, и это изменение явным образом влияет на форму сплайна. Построение кривой, показанной на рис. 19, приведено в примере 3.5. Из четырех контрольных точек, необходимых для кривой Безье третьего порядка, две точки (P_1 и P_4) задают местоположение концевых точек сплайна. Отрезки P_1P_2 и P_4P_3 задают направление и величину касательных векторов в концевых точках.

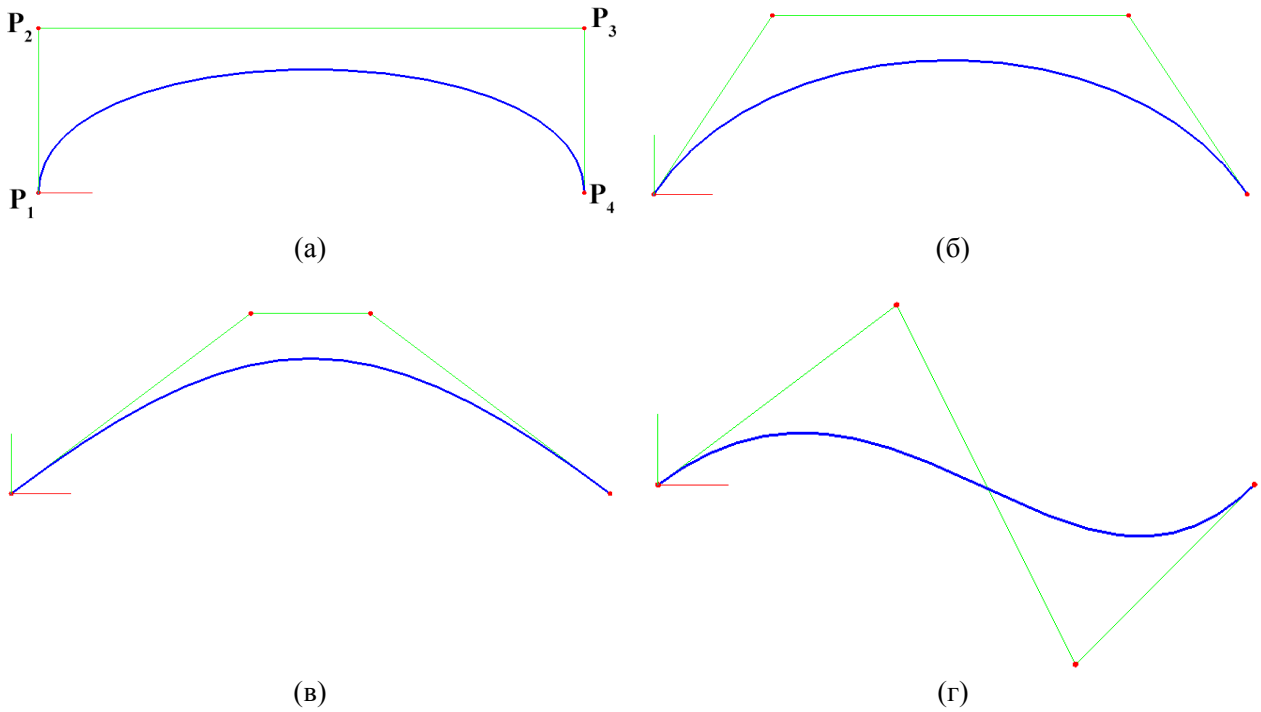


Рис. 19. Кривые Безье третьего порядка. Местоположение верхней пары контрольных точек (P_2 и P_3) изменяется. Таким образом меняется величина и направление касательных векторов в концевых точках, влияющих на форму сплайна.

Пример 3.5. Построение кубической кривой Безье по контрольным точкам (рис. 19).

```
#include "cur_bezier.h"                // MbBezier - кубическая кривая Безье
#include "cur_line_segment.h"          // MbLineSegment - отрезок

void MakeUserCommand0()
{
  // Контрольные точки
  SArray<MbCartPoint> arrPnts(4);

  // Для рис. 19(а)
```

```
  /*
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(0,30) );
  arrPnts.Add( MbCartPoint(100, 30) );
  arrPnts.Add( MbCartPoint(100, 0) );
  */

  // Для рис. 19(б)
  /*
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(20, 30) );
  arrPnts.Add( MbCartPoint(80, 30) );
  arrPnts.Add( MbCartPoint(100, 0) );
  */

  // Для рис. 19(в)
  /*
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(40, 30) );
  arrPnts.Add( MbCartPoint(60, 30) );
  arrPnts.Add( MbCartPoint(100, 0) );
  */

  // Для рис. 19(г)
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(40, 30) );
  arrPnts.Add( MbCartPoint(70, -30) );
  arrPnts.Add( MbCartPoint(100, 0) );

  // Построение незамкнутого кубического сплайна Безье
  // Используется конструктор для построения по четырем контрольным точкам
  MbBezier* pSpline = new MbBezier( arrPnts );

  // Отображение контрольных точек
  MbPlacement3D pl;
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPnts[i], &pl );

  // Отображение сплайна
  viewManager->AddObject( Style(3, RGB(0,0,255)), pSpline, &pl );

  // Отображение контрольной ломаной сплайна
  for (int i = 0; i < arrPnts.size()-1; i++)
  {
    MbLineSegment* pSeg = new MbLineSegment(arrPnts[i], arrPnts[i+1]);
    viewManager->AddObject( Style(1, RGB(0,255,0)), pSeg, &pl );
    ::DeleteItem( pSeg );
  }

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSpline );
}
```

Количество контрольных точек кривой Безье связано с порядком полиномов, на базе которых строится математическое описание сплайна. Чтобы указать большее количество контрольных точек, необходимо повышать степень кривой Безье. Однако вычисление точек кривых Безье высокой степени связано с увеличением вычислительных затрат. Кроме того, промежуточные контрольные точки кривых Безье высокой степени не имеют такого же очевидного геометрического смысла, как пары концевых точек, задающие направление и величину касательных векторов.

Для представления произвольных кривых с помощью сплайнов Безье в ядре C3D применяется не изменение степени базовых полиномов, а представление требуемой кривой в виде

последовательности сегментов. Каждый из этих сегментов представляет собой кубическую кривую Безье. Такое разбиение может быть выполнено автоматически в конструкторе класса MbBezier. В классе MbBezier предусмотрен конструктор, позволяющий вместо явного указания контрольных точек указать точки, через которые должна проходить кривая Безье. Эти точки называются «полюсами». Контрольные точки сплайна Безье, обеспечивающие его прохождение через полюса, вычисляются автоматически. Координаты вычисленных контрольных точек хранятся в массиве точек, которые наследует классом MbBezier от родительского класса MbPolyCurve (как и все рассматриваемые классы сплайнов, класс MbBezier унаследован от класса кривой, построенной по набору точек).

Т.о., класс MbBezier позволяет задать форму сплайна столь очевидным образом, как и в случае интерполяционных кубических сплайнов – указанием набора точек, принадлежащих кривой (с соблюдением аналогичных правил их расположения – приблизительно равномерно с добавлением точек в точках перегиба требуемой кривой). Но вместе с этим сохраняются возможности легкого управления направлением кривой, характерные для кривых Безье. Каждый полюс оказывается концевой точкой одного из кубических сегментов Безье, подобных показанным на рис. 19. Для изменения направления касательной в любой из таких точек достаточно изменить направление соответствующих контрольных точек, при этом форма кривой Безье изменится локально – максимум у двух смежных кубических сегментов.

Применение класса MbBezier показано в примере 3.6. В этом примере строится сплайн, проходящий через точки, показанные на рис. 14(а). На рис. 20 показана результирующая кривая вместе с точками-полюсами и контрольными точками. По контрольным точкам построена контрольная ломаная кривой Безье. Отрезки этой ломаной (одиночные у концевых точек сплайна и парные у полюсов) направлены по касательной к сплайну. Их местоположение соответствует точкам стыковки сегментов составной кривой Безье – они совпадают с полюсами.

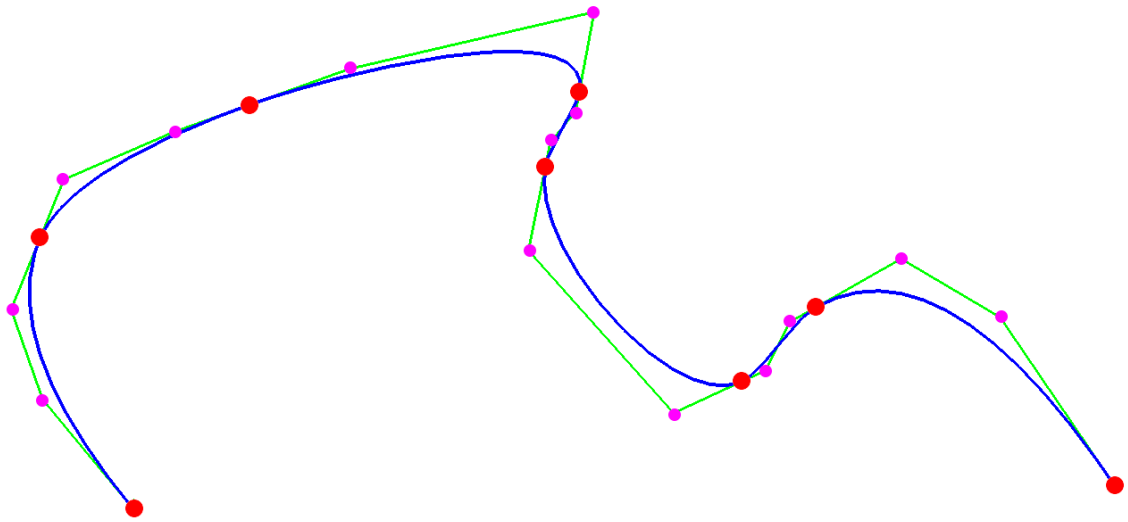


Рис. 20. Составная кубическая кривая Безье, состоящая из 7 сегментов. Красными круговыми маркерами отмечены полюса, пурпурными – контрольные точки (не совпадающие с полюсами). Контрольная ломаная показана зеленым цветом.

Пример 3.6. Построение составной кубической кривой Безье по точкам-полюсам (рис. 20).

```
#include "cur_bezier.h"                    // MbBezier - кубическая кривая Безье
#include "cur_line_segment.h"              // MbLineSegment – отрезок

void MakeUserCommand0()
{
  // Точки-полюса для составной кубической кривой Безье
  SArray<MbCartPoint> arrPoles(8);
  arrPoles.Add( MbCartPoint(0, 0) );
```

```
  arrPoles.Add( MbCartPoint(-7, 20) );
  arrPoles.Add( MbCartPoint(8.5, 30) );
  arrPoles.Add( MbCartPoint(33, 31) );
  arrPoles.Add( MbCartPoint(30.5, 25.5) );
  arrPoles.Add( MbCartPoint(45, 9.5) );
  arrPoles.Add( MbCartPoint(50.5, 15) );
  arrPoles.Add( MbCartPoint(72.5, 2) );

  // Построение незамкнутого кубического сплайна Безье по полюсам
  MbBezier* pSpline = new MbBezier( arrPoles, false );

  // (*) - в этом месте пример будет дополнен для сопряжения сплайна с отрезками

  // Получение массива вычисленных контрольных точек сплайна.
  // У составной кривой Безье в C3D есть особенность реализации – первые и
  // последние пары точек в массиве контрольных точек совпадают.
  // Они могут использоваться для сопряжения этой кривой с другими кривыми.
  SArray<MbCartPoint> arrPnts;
  pSpline->GetPointList( arrPnts );

  // Отображение сплайна (синим цветом)
  MbPlacement3D pl;
  viewManager->AddObject( Style(4, RGB(0,0,255)), pSpline, &pl );

  // Отображение полюсов (красный цвет)
  for (int i = 0; i<arrPoles.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPoles[i], &pl );

  // Отображение контрольных точек сплайна (пурпурный цвет)
  for (int i = 0; i < arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,255)), arrPnts[i], &pl );

  // Отображение контрольной ломаной сплайна (зеленый цвет)
  for (int i = 0; i < arrPnts.size()-1; i++)
  {
    MbLineSegment* pSeg = new MbLineSegment(arrPnts[i], arrPnts[i+1]);
    viewManager->AddObject( Style(3, RGB(0,255,0)), pSeg, &pl );
    ::DeleteItem( pSeg );
  }

  ::DeleteItem( pSpline );
}
```

Для выполнения сопряжения сплайна Безье с другими кривыми, направление которых известно, можно указать такое положение контрольных точек сплайна, чтобы касательные к сегментам сплайна были направлены требуемым образом. Для сопряжения сплайна, построенного в примере 3.6, с парой отрезков, показанных на рис. 14(б), можно выполнить перечисленные в примере 3.7 программные вызовы. В этом фрагменте контрольные точки для первого и последнего сегментов сплайна помещаются в вершины отрезков так, чтобы касательные к сегментам оказались параллельны отрезкам, с которыми сопрягается сплайн. В результате будет получен сплайн Безье, показанный на рис. 21.

Пример 3.7. Изменение направления касательных в концевых точках кривой Безье (рис. 21).

```
void MakeUserCommand0()
{
  // ... все приведенные программные вызовы надо добавить в пример 3.6 в
  // позицию, отмеченную комментарием (*).
```

```
  // Два отрезка, соединяющихся со сплайном в концевых точках
  // pSeg1 - вертикальный отрезок
  // pSeg2 - отрезок под углом -45 градусов к горизонтальной оси
  MbLineSegment* pSeg1 = new MbLineSegment(arrPoles[0], MbCartPoint(0, -50));
  MbLineSegment* pSeg2 = new MbLineSegment(arrPoles[7], MbVector(1, -1), 0, 50 );

  // Метод MbBezier::ChangePoint позволяет явно задать положение контрольной точки
  // по ее индексу (начиная с 0).

  // Первая вершина отрезка pSeg1 совпадает с концевой точкой сплайна и с первой парой
  // контрольных точек (с индексами 0 и 1). Совместим контрольную точку с индексом 0
  // со второй вершиной отрезка pSeg1 так, чтобы касательный вектор в концевой точке
  // сплайна был направлен параллельно этому отрезку.
  MbCartPoint seg1p2;
  pSeg1->GetPoint2(seg1p2);
  pSpline->ChangePoint( 0, seg1p2 );

  // Произведем аналогичное построение для последней контрольной точки –
  // поместим ее во вторую вершину отрезка pSeg2, тогда касательный вектор в
  // концевой точке сплайна будет направлен параллельно этому отрезку.
  MbCartPoint seg2p2;
  pSeg2->GetPoint2(seg2p2);
  pSpline->ChangePoint( pSpline->GetPointListCount()-1, seg2p2 );

  // Отображение отрезков, с которыми сопрягается сплайн (красный цвет)
  MbPlacement3D pls;
  viewManager->AddObject( Style(3, RGB(255,0,0)), pSeg1, &pls );
  viewManager->AddObject( Style(3, RGB(255,0,0)), pSeg2, &pls );

  //...
}
```

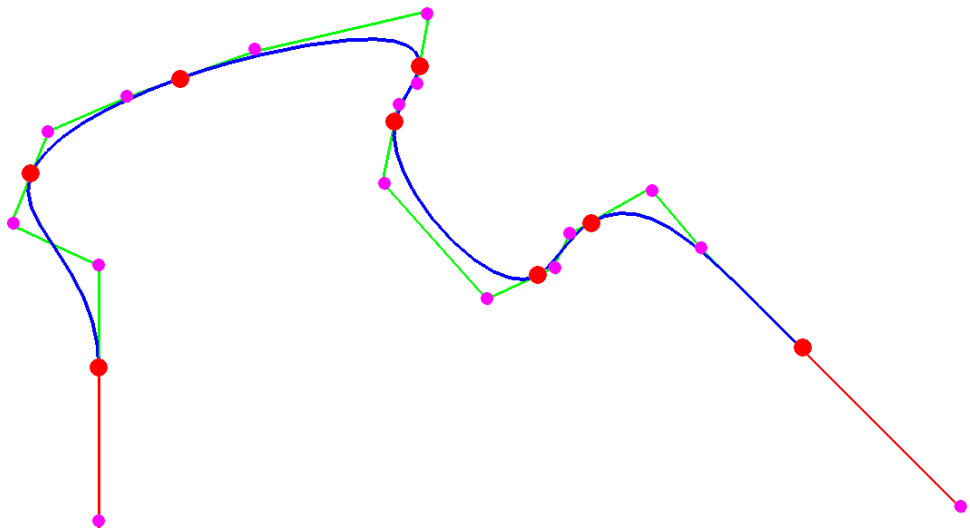


Рис. 21. Сопряжение составной кубической кривой Безье с парой отрезков путем указания местоположения пары контрольных точек (пример 3.7).

Сравните рис. 21 и рис. 20 и обратите внимание, что изменилась форма только двух сегментов кривой Безье – первого и последнего. На остальные сегменты смещенные контрольные точки не влияют. Удаление контрольных точек от полюсов сплайна не сильно влияет на форму сплайна. Результат, аналогичный рис. 21, будет получен и при небольшом смещении контрольных точек в направлении требуемой касательной.

3.4 MbNurbs – кривая NURBS

В геометрическом моделировании сплайны NURBS применяются в качестве универсального способа представления произвольных кривых. Как и в случае кривых Безье, форму NURBS-сплайнов можно задавать с помощью контрольных точек. Эти сплайны также проходят через концевые контрольные точки и аналогичным образом в них задается направление касательных. В отличие от кривых Безье (являющихся частным видом NURBS-сплайнов), у NURBS-сплайнов нет жесткого ограничения на количество контрольных точек в зависимости от степени сплайна. Кубическую кривую Безье можно построить только по 4 контрольным точкам, а кубический NURBS-сплайн может задаваться произвольным количеством контрольных точек. У NURBS-сплайнов есть еще ряд управляющих параметров: весовые коэффициенты и узловые значения. Весовые коэффициенты – это действительные числа, сопоставленные контрольным точкам. Они определяют, насколько сильно соответствующая контрольная точка влияет на форму сплайна. Минимальное нулевое значение означает, что контрольная точка на сплайн не влияет. По мере увеличения весового коэффициента сплайн начинает «сильнее притягиваться» к контрольной точке. Форма NURBS-сплайна зависит не только от местоположения контрольных точек, но и от их весовых коэффициентов. Узловые значения также влияют на форму сплайна. Это набор значений параметра t из диапазона допустимых значений $[t_{min}, t_{max}]$. Выбор этих значений влияет на то, как при вычислении точки сплайновой кривой учитываются контрольные точки и их веса. Используя повторяющиеся узловые значения, можно формировать на сплайновой кривой угловые точки и «резкие изгибы». Использование четырех способов настройки формы сплайна (порядок сплайна, контрольные точки, весовые коэффициенты, узловые значения) позволяет с помощью одной сплайновой кривой точно представить любую кривую.

На рис. 22 показаны незамкнутые NURBS-сплайны различного порядка, построенные по набору из 9 контрольных точек. Программные вызовы для построения показаны в примере 2.8. NURBS-кривая 2-го порядка совпадает с контрольной ломаной. У кривой 3-го порядка заметно, что сегменты ломаной являются касательными к сплайну. У кривых любого порядка первый и последний сегменты контрольной ломаной являются касательными в концевых точках сплайна (как и у кривой Безье).

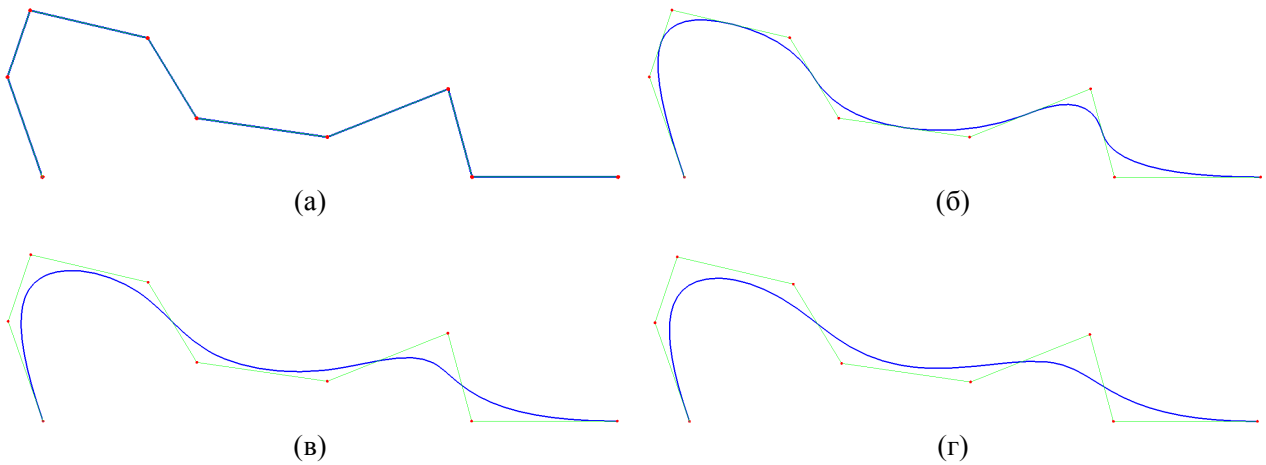


Рис. 22. NURBS-сплайны различного порядка, построенные по одинаковым контрольным точкам. (а) Порядок 2 – сплайн совпадает с контрольной ломаной. (б) Порядок 3. (в) Порядок 4. (г) Порядок 5.

Пример 3.8. Построение NURBS-сплайна по контрольным точкам (Рис. 22).

```
#include "cur_nurbs.h"                  // MbNurbs - NURBS-сплайн
#include "cur_line_segment.h"           // MbLineSegment – отрезок
```

```
void MakeUserCommand0()
{
  // Контрольные точки сплайна
  SArray<MbCartPoint> arrPnts(9);
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(-8.5, 24.3) );
  arrPnts.Add( MbCartPoint(-3, 40.5) );
  arrPnts.Add( MbCartPoint(25.6, 33.8) );
  arrPnts.Add( MbCartPoint(37.5, 14.3) );
  arrPnts.Add( MbCartPoint(69.3, 9.7) );
  arrPnts.Add( MbCartPoint(98.7, 21.4) );
  arrPnts.Add( MbCartPoint(104.5, 0) );
  arrPnts.Add( MbCartPoint(140, 0) );

  // Порядок сплайна: может быть больше 1 и не более числа контрольных точек
  ptrdiff_t degree = 2;

  // Построение незамкнутого NURBS-сплайна по контрольным точкам с помощью вызова
  // статического метода MbNurbs::Create.
  MbNurbs* pSpline = MbNurbs::Create( degree, arrPnts, false /* флаг замкнутости */ );

  // Отображение контрольных точек
  MbPlacement3D pl;
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(1, RGB(255,0,0)), arrPnts[i], &pl );

  // Отображение контрольной ломаной
  for (int i = 0; i<arrPnts.size()-1; i++)
  {
    MbLineSegment* pSeg = new MbLineSegment(arrPnts[i], arrPnts[i+1]);
    viewManager->AddObject( Style(1, RGB(0,255,0)), pSeg, &pl );
    ::DeleteItem( pSeg );
  }

  // Отображение сплайна
  viewManager->AddObject( Style(3, RGB(0,0,255)), pSpline, &pl );

  // Уменьшение счетчиков ссылок динамически созданных объектов ядра
  ::DeleteItem( pSpline );
}
```

Влияние весовых коэффициентов на форму NURBS-сплайна третьего порядка показано на рис. 23. Одна из контрольных точек, использовавшихся в примере 3.8, была значительно смещена по вертикали. С увеличением весового коэффициента NURBS-сплайн «сильнее притягивается» к соответствующей контрольной точке. Построение сплайна, показанного на рис. 23(г), приведено в примере 3.9.

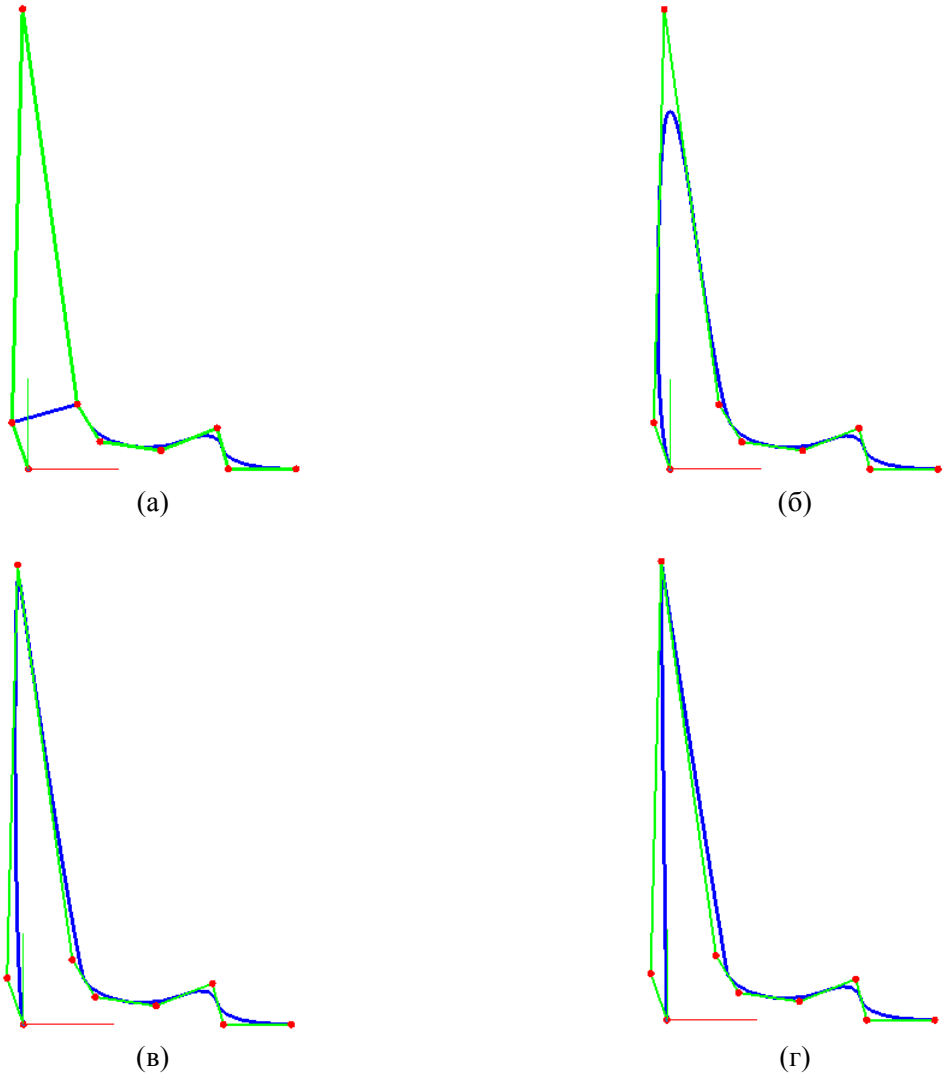


Рис. 23. NURBS-сплайны третьего порядка с изменением весового коэффициента третьей слева контрольной точки. (а) Вес точки равен 0.0 – контрольная точка не влияет на форму кривой. (б) Вес равен 1.0 (значение по умолчанию). (в) Вес равен 10.0 – кривая сильнее изгибается в сторону контрольной точки. (г) Вес равен 1000.0 – смещение увеличивается, так что сплайн стремится пройти через контрольную точку.

Пример 3.9. Указание весового коэффициента контрольной точки NURBS-сплайна (рис. 23).

```
#include "cur_nurbs.h"                  // MbNurbs - NURBS-сплайн
#include "cur_line_segment.h"           // MbLineSegment - отрезок

void MakeUserCommand0()
{
  // Контрольные точки сплайна
  SArray<MbCartPoint> arrPnts(9);
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(-8.5, 24.3) );
  arrPnts.Add( MbCartPoint(-3, 240.5) );
  arrPnts.Add( MbCartPoint(25.6, 33.8) );
  arrPnts.Add( MbCartPoint(37.5, 14.3) );
  arrPnts.Add( MbCartPoint(69.3, 9.7) );
  arrPnts.Add( MbCartPoint(98.7, 21.4) );
  arrPnts.Add( MbCartPoint(104.5, 0) );
  arrPnts.Add( MbCartPoint(140, 0)) ;
```

```
  // Весовые коэффициенты контрольных точек
  SArray<double> arrWeights(9);
  arrWeights.Add(1.0);
  arrWeights.Add(1.0);
  arrWeights.Add(1000.0);
  arrWeights.Add(1.0);
  arrWeights.Add(1.0);
  arrWeights.Add(1.0);
  arrWeights.Add(1.0);
  arrWeights.Add(1.0);
  arrWeights.Add(1.0);

  // Порядок сплайна: больше 1 и не больше числа контрольных точек
  ptrdiff_t degree = 3;

  // Построение незамкнутого NURBS-сплайна по контрольным точкам и их весам
  MbNurbs* pSpline = MbNurbs::Create( degree, arrPnts, false, &arrWeights );

  // Отображение контрольных точек
  MbPlacement3D pl;
  for (int i = 0; i<arrPnts.size(); i++)
    viewManager->AddObject( Style(5, RGB(255,0,0)), arrPnts[i], &pl );

  // Отображение контрольной ломаной
  for (int i = 0; i<arrPnts.size()-1; i++)
  {
    MbLineSegment* pSeg = new MbLineSegment(arrPnts[i], arrPnts[i+1]);
    viewManager->AddObject( Style(2, RGB(0,255,0)), pSeg, &pl );
   ::DeleteItem( pSeg );
  }

  // Отображение сплайна
  viewManager->AddObject( Style(3, RGB(0,0,255)), pSpline, &pl );

  ::DeleteItem( pSpline );
}
```

Построение NURBS-сплайнов может выполняться несколькими способами, в т.ч. могут применяться следующие средства:

1) Статические методы MbNurbs::Create. Эти методы заменяют вызов конструкторов класса и упрощают обработку ошибок построения сплайнов (поскольку из конструктора вернуть значение нельзя, а из метода Create можно – нулевое значение означает ошибку при построении сплайна). Использование двух подобных методов было показано в примерах 3.8 и 3.9. При вызове методов MbNurbs::Create необходимо указывать характеристики сплайна – порядок, контрольные точки, весовые коэффициенты и узловые значения.
2) Методы MbNurbs::CreateNURBS4 для построения сплайнов 4-го порядка с указанием различных наборов характеристик сплайна.
3) Функция ::SplineCurve (заголовочный файл action_curve.h) для построения NURBS-кривой четвертого порядка, проходящей через заданный набор точек.
4) Функции ::NurbsConics (заголовочный файл action_curve.h) для построения в виде NURBS-сплайнов кривых - конических сечений с указанием разных наборов параметров.
5) Метод родительского класса MbCurve::NurbsCurve, позволяющий получить любую кривую в виде NURBS-сплайна.

3.5 Задания

1) Постройте кривую Безье для сопряжения отрезков, показанных на рис. 16 (координаты точек для использования в качестве полюсов содержатся в примере 3.4). Постройте с наложением два сплайна одновременно (кривую Безье и сплайн Эрмита из примера 3.4) и сравните их форму.
2) Постройте NURBS-кривую, проходящую через заданные точки, с помощью функции ::SplineCurve (заголовочный файл action_curve.h). В качестве координат точек используйте координаты контрольных точек из примера 3.8 или 3.9. Функция ::SplineCurve не позволяет указывать порядок сплайна и выполняет построение NURBS-кривых четвертого порядка. Обратите внимание на отличие построенного сплайна от сплайна, показанного на рис. 4(в). Точки, передававшиеся в примерах 3.8 и 3.9 методам MbNurbs::Create, влияли на форму сплайна за счет направления касательных. При передаче этих же точек функции ::SplineCurve формируется сплайн, который проходит через эти точки.
3) Постройте NURBS-кривую, аппроксимирующую окружность. Сначала постройте окружность в виде объекта класса MbArc. Затем у этого объекта вызовите метод MbCurve::NurbsCurve для получения кривой в виде NURBS-сплайна. Вызов может осуществляться следующим образом (в предположении, что pArc – объект, представляющий окружность):

```
MbNurbsParameters parms;     // Структура с характеристиками сплайна
parms.degree = 3;            // Порядок сплайна
// Получение кривой pArc в виде NURBS-сплайна с заданными характеристиками
MbCurve* pCurve = pArc->NurbsCurve( parms );
```

   Отобразите этот сплайн в тестовом приложении и с помощью окна свойств выясните, сколько контрольных точек задают этот сплайн. Повторите построение для сплайнов порядка 2, 3, 4, 5, 6. Уменьшается или увеличивается количество контрольных точек сплайна с увеличением порядка?
4) Для кривой Безье из задания (2) постройте две эквидистантные кривые, которые могли бы изображать края дороги на рис. 15. Для представления эквидистантной кривой используйте класс MbOffsetCurve (заголовочный файл cur_offset_curve.h) и его конструктор с указанием базовой кривой и величины смещения. Отобразите в тестовом приложении обе эквидистантных кривых и их базовую кривую Безье.

4. Заключение

В данной работе обсуждались классы для описания составных кривых и сплайнов. Если кривую можно представить в виде последовательности соединяющихся сегментов разнотипных кривых, то в таком случае составная кривая может быть представлена в виде контура – объекта класса MbContour. Контуры используются как для геометрических построений на плоскости, так и в качестве параметров операций твердотельного моделирования. В качестве примера в работе рассматривалось применение контуров в качестве образующей для операции выдавливания и построения поверхностей и тел выдавливания.

Также в работе были рассмотрены основные классы для представления сплайновых кривых в двумерном пространстве: кубические сплайны MbCubicSpline и MbHermit, кубические кривые Безье MbBezier и NURBS-сплайны MbNurbs. В работе были рассмотрены типовые способы построения сплайнов по набору контрольных точек.