Getting started with C3D Modeler

LESSON 2

Two-dimensional curves

https://c3dlabs.com
Moscow
2019

Table of Contents

1. Introduction

Classes for representing curves in C3D Modeler belong to a group of geometric objects. Curves can be used to construct drawings and solve problems of computational geometry. They are also often used to construct surfaces and shells - the most high-level elements of geometric models of solids. The important role of curves in the modeling of solids based on the B-Rep boundary representation is also related to the fact that they are used to represent the edges bounding fragments of surfaces (faces) of models of solids. Computer graphics literature often deals with polygonal models with flat edges. In geometric modeling systems for CAD, polygonal models are also used, but in the general case, the faces of the bodies are fragments of curvilinear surfaces, and the edges bounding them are segments of arbitrary curves.

In the software implementation, C3D Modeler curve class objects are used in different ways: they can be used explicitly to perform geometric calculations or as parameters of higher-level operations, as well as implicitly when performing other kernel operations (for example, when calculating edge edges).

C3D Modeler tools associated with working with curves can be divided into several categories:

- 2D curves (MbCurve base class and a set of classes inherited from it).
- 3D curves (MbCurve3D base class and classes inherited from it).
- Functions for constructing two-dimensional curves. They implement geometric algorithms, as a result of which some two-dimensional curve is formed.
- Functions for constructing 3D curves.
- Functions of geometric algorithms that are not related to the creation of new curves. These include operations with curves and points, operations with curves in two-dimensional space, operations with curves and surfaces.

This paper discusses the elements of C3D Modeler for operations with simplest curves in a two-dimensional space [1]. In particular, these include: straight lines, straight line segments, circles and ellipses and their arcs. When working with curves in C3D Modeler, it is necessary to have an idea of the different ways of creating curves, which for a specific task are chosen either taking into account the requirements for their mathematical description, or the requirements specified in the form of user restrictions. You must also be able to perform operations on the calculation of intersection points, distances between curves and other geometric objects. Operations of this type allow you to perform geometric constructions, the results of which can also be used for subsequent actions (similar to auxiliary constructions, which are often performed when working interactively in typical CAD systems).

2. Classes for representation of curves in two-dimensional space

Part of the C3D class diagram associated with the representation of two-dimensional curves is shown in Fig. 1. Classes TapeBase and MbRefItem are classes for the implementation of service functionality, Classes MbPlaneItem and MbCurve implement the important concepts of the kernel - a geometric object and a curve in two-dimensional space. These classes are abstract (objects of such classes are not created, classes are used to build more complex classes through inheritance). To represent curves of specific types, it is required to create classes inherited from MbCurve. The kernel includes more than 10 such classes, and if necessary, you can create new ones. Some of the curves

[1] The mathematical foundations of the curves are described in the book Geometric Modeling: The mathematics of shapes Paperback – December 24, 2014 by Nikolay Golovanov, Chapter 1 https://www.amazon.com/Geometric-Modeling-mathematics-Nikolay-Golovanov/dp/1497473195 .

inherited from MbCurve are also abstract classes, for example, MbPolyCurve (curve defined by control points). Such classes are basic for a family of curves. In particular, members of the MbPolyCurve family are the inherited MbBezier, MbPolyline classes and several others.

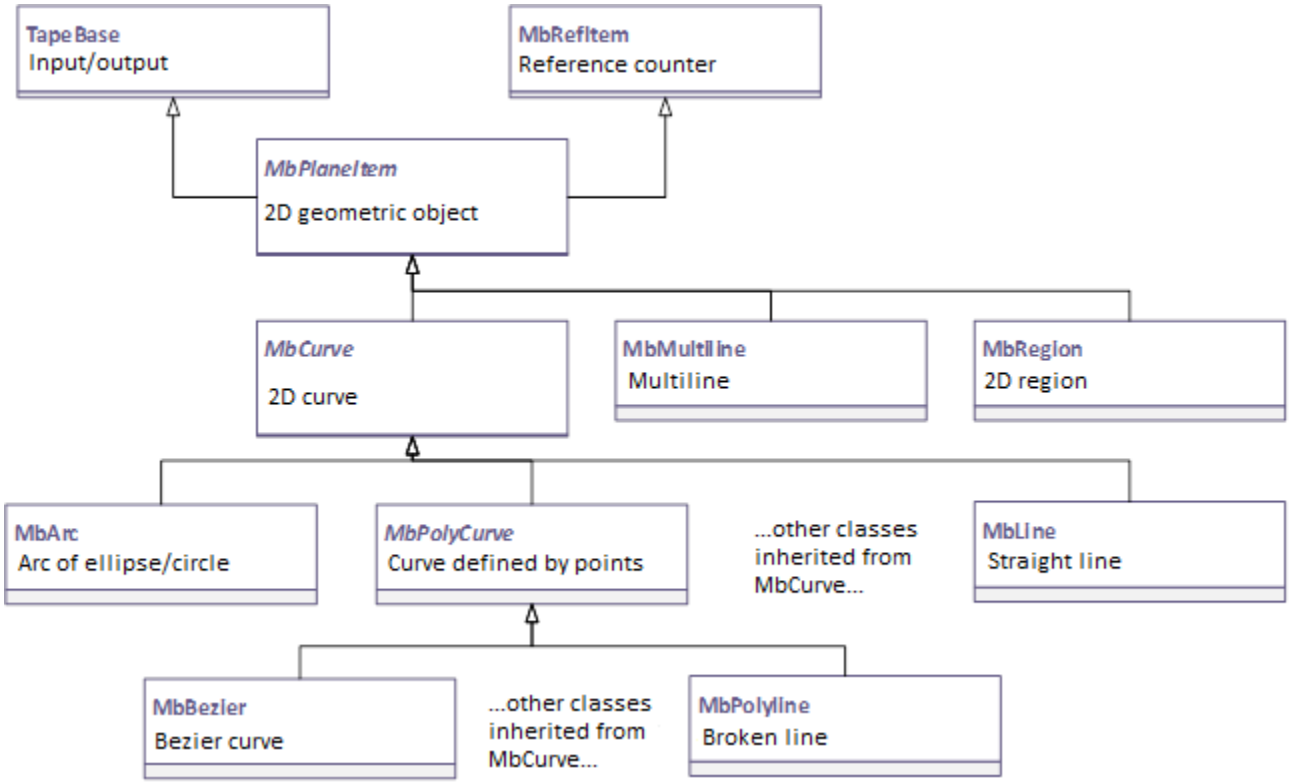


Fig. 1. Fragment of a class diagram for representing curves in two-dimensional space. The names of the abstract base classes MbPlaneItem, MbCurve and MbPolyCurve are in italics.

2.1 The base class of two-dimensional geometric objects MbPlaneItem

The MbPlaneItem class contains virtual functions that form a common interface of geometric objects in two-dimensional space. In addition to curves MbCurve, such objects are multi-lines MbMultiline (sets of curves) and regions MbRegion (sets of specially organized contours MbContour) (Fig. 1).

The MbPlaneItem class inherits from two classes: MbRefItem and TapeBase. These two classes are intended to add support features typical for kernel classes to MbPlaneItem.

The TapeBase class implements basic functions for writing / reading attributes of objects from streams (for example, associated with files). This mechanism is necessary to provide long-term storage of objects, for example, in a file with data about a geometric model. To support it in classes of concrete geometric objects, it is required to provide a number of methods, the main ones of them - Read and Write.

The MbRefItem class implements a reference counting mechanism. This is a common way to ensure the correct deletion of dynamically created objects. It is based on using a couple of methods: AddRef() and Release(). When a class begins to use a dynamically created object, the reference count of this object is increased by calling AddRef(). When the object is no longer needed, the reference count is decreased by calling Release(). In case if the next call to Release() decreases the reference count to 0, then the object is automatically deleted (by calling its destructor). In a sense, this mechanism resembles the usual Open / Close functions for accessing various resources (for example, files).

The MbPlaneItem class is abstract and its purpose is to provide for the inheritance of an interface that is common to all geometric objects in a two-dimensional space. The methods of this interface can be divided into several groups:

1) Obtaining information about the type of geometric object.
2) Check for equality, uniformity, equating and copying geometric objects.
3) Geometric transformations on the plane.
4) Computational operations.
5) Access to named properties of a geometric object.
6) Getting service points of the object: base points and anchor points.
7) Support read / write threads.

The methods related to the listed categories are shown below in the fragment description of the MbPlaneItem class. Please note that most of the methods are abstract virtual functions (marked with equality zero). Such functions must be implemented in inherited classes. There are no attributes in the MbPlaneItem class.

```
class MbPlaneItem : public TapeBase, public MbRefItem {
  // Macro for prohibiting C++ class elements generated by the compiler automatically:
  // copy constructor and assignment operator
  OBVIOUS_PRIVATE_COPY( MbPlaneItem )

protected :
  // Default constructor, not used.
  // Inherited classes should have their own constructors,
  // the parameters of which take into account the peculiarities of these classes.
  MbPlaneItem();

public :
  // Destructor. It usually is not explicitly called.
  // It's used in conjunction with the reference counting mechanism
  virtual ~MbPlaneItem();

  // GROUP #1: OBTAINING INFORMATION ABOUT THE TYPE OF GEOMETRIC OBJECT
  // Class code inherited from MbRefItem
  virtual MbeRefType  RefType() const;
  // Type code
  virtual MbePlaneType IsA() const = 0;
  // Group type code denoting parent class
  virtual MbePlaneType Type() const = 0;
  // Family code denoting the base class inherited from MbPlaneItem
  virtual MbePlaneType Family() const = 0;

  // GROUP #2: COMPARISON AND COPYING OF GEOMETRIC OBJECTS
  // Creation of a copy of an object with the possibility of using a service registrar
  // to prevent multiple copying of parts of composite objects
  virtual MbPlaneItem& Duplicate( MbRegDuplicate* iReg = NULL ) const = 0;
  // Verification of objects for equality (their attributes must have the same values)
  virtual bool IsSame( const MbPlaneItem& item ) const = 0;
  // Verification of objects of the same type (they must have the same type of
  // attributes and the same type of component parts, if any)
  virtual bool IsSimilar( const MbPlaneItem& item ) const;
  // Equating an object (valid for objects of the same type)
  virtual bool SetEqual ( const MbPlaneItem& item ) = 0;

  // GROUP #3: GEOMETRIC TRANSFORMATIONS
  // Arbitrary transformation specified in matrix form.
  virtual void Transform( const MbMatrix& matr, MbRegTransform* iReg = NULL,
                          const MbSurface* newSurface = NULL ) = 0;
  // Parallel transfer to a given vector
```

```
  virtual void Move( const MbVector& to, MbRegTransform* iReg = NULL,
                     const MbSurface* newSurface = NULL ) = 0;
  // Rotate around a point at a given angle (in radians)
  virtual void Rotate( const MbCartPoint& pnt, const MbDirection& angle,
                  MbRegTransform* iReg = NULL, const MbSurface* newSurface=NULL ) = 0;
  void Rotate( const MbCartPoint& pnt, double angle, MbRegTransform* iReg = NULL,
               const MbSurface* newSurface = NULL );

  // GROUP #4: COMPUTATIONAL OPERATIONS
  // Expansion of the dimensional rectangle to include this object in it
  virtual void AddYourGabaritTo( MbRect& r ) const = 0;
  // Checking whether an object includes in a specified dimensional rectangle
  virtual bool IsVisibleInRect( const MbRect& rect, bool exact = false ) const = 0;
  // Calculate distance to point
  virtual double DistanceToPoint( const MbCartPoint& to ) const = 0;
  virtual bool   DistanceToPointIfLess( const MbCartPoint& to, double& d ) const = 0;

  // GROUP #5: NAMED PROPERTIES OF THE OBJECT
  /// Create your own property with its name.
  virtual MbProperty& CreateProperty( MbePrompt name ) const = 0;
  // Get the internal properties of the object to view and modify them.
  virtual void GetProperties( MbProperties& properties ) = 0;
  // Modifying the internal properties of an object
  // by copying the values from the given properties.
  virtual void SetProperties( MbProperties& properties ) = 0;

  // GROUP #6: SERVICE POINTS
  // Get base points
  virtual size_t GetBasisPoints( RPArray<MbCartPoint>& s ) = 0;
  // Get anchor points
  virtual void GetFastenPoints( SArray<MbCartPoint>& s ) const = 0;

  // GROUP #7: SUPPORT READING / WRITING THREADS
  // Registration of an object to prevent its multiple writing.
  void PrepareWrite();

  // Macro to add to the class methods provided by the base class TapeBase
  DECLARE_PERSISTENT_CLASS( MbPlaneItem )
};
```

The MbPlaneItem class interface is designed to provide unified processing in C3D Modeler for all geometric objects in two-dimensional space (of course, if they are correctly inherited from this class).

2.2 Parent class of two-dimesional curves MbCurve

The parent class MbCurve, inherited from the base class MbPlaneItem, is intended to describe curves in two-dimensional space. In the curve class, the interface of the geometry object MbPlaneItem is inherited, and several more groups of methods are added (in the following list, these are groups 2-5):

1) General functions of a two-dimensional geometric object (this is a part of the interface inherited from MbPlaneItem);
2) Description of the curve definition area.
3) Methods for calculating the differential (defined in points) characteristics of the curve (these are the coordinates of the points, the values of the derivatives, the normal vector and the tangent vector).
4) Functions of moving along a curve;

5) General functions of the curve. It is the most extensive group that includes functions for performing typical calculations with a curve defined using parametric equations.

The interface of the MbCurve class is quite extensive, therefore, we will provide its fragments below with the selection of the main methods from the listed groups.

```
class MbCurve : public MbPlaneItem {
public :

  // GROUP #1: GENERAL FUNCTIONS OF TWO-DIMENSIONAL GEOMETRIC OBJECT
  // This group includes methods inherited from MbPlaneItem. Some of them are
  // implemented in MbCurve, some - in inherited classes
  virtual MbePlaneType IsA()  const = 0;
  virtual MbePlaneType Type() const;
  // ... and other methods of MbPlaneItem

  // GROUP #2: DESCRIPTION OF THE CURVE DEFINITION AREA
  // Getting the limit values of the curve parameter
  virtual double GetTMin() const = 0;
  virtual double GetTMax() const = 0;
  // Check whether the curve is closed
  virtual bool   IsClosed() const = 0;
  // Check whether a closed curve is periodic
  virtual bool   IsPeriodic() const;
  // Getting the period for a periodic curve
  virtual double GetPeriod() const;

  // GROUP #3: CALCULATION OF THE CURVE CHARACTERISTICS
  // Calculation of coordinates of a point of a curve, derivatives, tangent and
  // normal vectors (with automatic adjustment of the parameter t when going beyond
  // the definition domain)
  virtual void PointOn  ( double& t, MbCartPoint& p ) const = 0;
  virtual void FirstDer ( double& t, MbVector& v ) const = 0;
  virtual void SecondDer( double& t, MbVector& v ) const = 0;
  virtual void ThirdDer ( double& t, MbVector& v ) const = 0;
          void Tangent  ( double& t, MbVector& v ) const;
          void Tangent  ( double& t, MbDirection& d ) const;
          void Normal   ( double& t, MbVector& v ) const;
          void Normal   ( double& t, MbDirection& d ) const;
  // Calculation of coordinates of a point of a curve, derivatives, tangent and
  // normal vectors without control of falling of the value of the parameter t
  // into the domain of definition. When going beyond the domain of definition,
  // the characteristics of a point will be calculated on the curve extension.
  virtual void _PointOn  ( double t, MbCartPoint& p ) const;
  virtual void _FirstDer ( double t, MbVector& v ) const;
  virtual void _SecondDer( double t, MbVector& v ) const;
  virtual void _ThirdDer ( double t, MbVector& v ) const;
          void _Tangent  ( double t, MbVector& v ) const;
          void _Tangent  ( double t, MbDirection& d ) const;
          void _Normal   ( double t, MbVector& v ) const;
          void _Normal   ( double t, MbDirection& d ) const;

  // GROUP #4: FUNCTIONS OF MOVING ALONG A CURVE
  // Calculation of the parameter step in the vicinity of the point with
  // the parameter t, so that the deviation of the curve from its polygon
  // does not exceed the threshold value sag
  virtual double Step( double t, double sag ) const;
  // Calculation of the parameter step in the vicinity of the point t,
  // so that the angular deviation of the tangent curve does not exceed
  // the threshold angle ang.
  virtual double DeviationStep( double t, double ang ) const;
```

```
  // GROUP #5: GENERAL CURVE FUNCTIONS
  // Curvature Calculation
  virtual double Curvature( double t ) const;
  // Calculation of the derivative of curvature on the parameter
          double CurvatureDerive( double t ) const;
  // Calculation of the signed radius of curvature of a curve
          double CurvatureRadius( double t ) const;
  // ... and more than 50 methods for implementing algorithms for working with curves
};
```

Like the base class MbPlaneItem, the class MbCurve is abstract (Fig. 1). Virtual function implementations are present in the classes inherited from MbCurve to represent curves of specific types (Table 1). Among the classes listed in table 1, MbPolyCurve occupies a special place. This is the parent class of a family of curves defined by a set of control points. This family includes a class for representing a polyline and classes for representing various splines.

Table 1. Classes for representing different two-dimensional curves inherited from the MbCurve class.

№	Class	Description
1	MbLine	Straight line.
2	MbLineSegment	Straight line segment.
3	MbArc	An arc of an ellipse or a circle.
4	MbCosinusoid	Cosinusoid.
5	MbPolyCurve	Curve given by control points. This is an abstract parent class for the curve family, which includes: MbPolyline (polyline), MbBezier (Bezier curve), MbCubicSpline (cubic spline), MbHermit (composite Hermite cubic spline) and MbNurbs (NURBS curve).
6	MbContour	Contour (composite curve consisting of joined segments). There is a kind of contour in the kernel – MbContourWithBreaks - contour with breaks inherited from MbContour.
7	MbOffsetCurve	Equidistant curve for a given base curve.
8	MbCharacterCurve	Curve with the assignment of coordinate functions in symbolic form.
9	MbPointCurve	Curve degenerated to a point.
10	MbProjCurve	Projection curve.
11	MbReparamCurve	Reparameterized curve for a given base curve.
12	MbTrimmedCurve	Truncated curve for a given base curve.

3. MbLine – a straight line in two-dimensional space

The MbLine class is intended to represent a straight line defined by a vector parametric equation:

$$\mathbf{r}(t) = \mathbf{p_0} + t\mathbf{s}, \text{ где } t\in(-\infty, +\infty)$$

In this equation: $\mathbf{p_0}$ – position vector of some point belonging to the line; $\mathbf{s}$ – direction vector of a straight line, t – parameter. Each value of the parameter t corresponds to a position vector $\mathbf{r}(t)$ of a certain point of the line. A point with a position vector $\mathbf{p_0}$ will be called a starting point, since $\mathbf{r}(t)=\mathbf{p}_0$ at $t=0$.

There are 2 attributes for storing equation parameters in the MbLine class:

```
class MbLine : public MbCurve {
```

```
private :
  MbCartPoint origin; // Starting point
  MbDirection direct; // Direction vector
```

The MbLine class has 6 constructors, 5 of which are intended to create a straight line with an indication of the parameters in various forms. Also in the MbLine class there are Init methods with similar constructors of parameter sets that allow changing attributes of existing MbLine objects.

```
// Construction of a straight line by the starting point and the direction vector
MbLine( const MbCartPoint& initP, const MbDirection& initDirect );
MbLine( const MbCartPoint& initP, const MbVector& initV );
// Construction of a straight line by a point and angle (in radians)
MbLine( const MbCartPoint& initP, double angle );
// Construction of a line by two points
MbLine( const MbCartPoint& p1, const MbCartPoint& p2 );
// Construction of a straight line by the coefficients of the general equation
// of a line Ax + By + C = 0
MbLine( double a, double b, double c );
```

In Example 3.1, a straight line is drawn that runs at an angle of 45° to the horizontal axis in the XY plane of the world coordinate system. After constructing a straight line as an object pLine, it calls some functions of the MbPlaneItem and MbCurve interface to get the properties of the direct object. The comments to the method calls, starting with the pLine-> IsA() call, show the returned values (you can check them by setting a stop point on this line, and then executing the program step by step using the Visual Studio debugger). From the property values of the object pLine, it is clear that this object is a non-periodic non-closed straight line, the domain of definition of the parameter t is the range [–50000000, 50000000] (the infinitely long straight line is a mathematical abstraction, therefore in the practical implementation of C3D the range of t values is limited).

Example 3.1 Construction of a straight line in two-dimensional space

```
#include "cur_line.h"           // MbLine – Straight line in two-dimensional space

void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Factor to convert angle values from degrees to radians
  const double DEG_TO_RAD = M_PI/180.0;

  // CONSTRUCTING TWO-DIMENSIONAL STRAIGHT LINE – BY A POINT AND A DIRECTION VECTOR
  MbCartPoint p1(0, 0);
  MbDirection dir( DEG_TO_RAD * 45 );
  // Dynamically creating a straight line object by calling the MbLine constructor
  MbLine* pLine = new MbLine( p1, dir);
  // Display of a straight line
  if ( pLine != NULL )
    viewManager->AddObject( Style( 1, RGB(255,0,0)), pLine, &pl );

  // Getting information about the type of geometric object
  MbePlaneType type_obj = pLine->IsA();      // pt_Line
  // Group type code denoting parent class
  MbePlaneType type_parent = pLine->Type();  // pt_Curve
  // Family code denoting the base class inherited from MbPlaneItem
  MbePlaneType type_family = pLine->Family();// pt_Curve

  // Characterization of the parameter definition domain t
  // obtaining the limit values of the curve parameter
  double tMin = pLine->GetTMin();                  // -50000000
```

```
  double tMax = pLine->GetTMax();                    // 50000000
  // Check whether the curve is closed
  bool bClosed = pLine->IsClosed();                  // false
  // Check whether a closed curve is periodic
  bool bIsPeriodic = pLine->IsPeriodic();            // false
  // Getting the period for a periodic curve
  double tPeriod = pLine->GetPeriod();               // 0.0
  // Calculate the metric length of the curve
  double length = 0.0;
  bool bHasLength = pLine->HasLength( length );      // false
  // Check whether the curve is bounded
  bool bIsBounded = pLine->IsBounded();              // false
  // Check whether the curve is straight
  bool bIsStraight = pLine->IsStraight();            // true

  // Reducing the reference count of a dynamic object
  ::DeleteItem( pLine );
}
```

Example 3.2 demonstrates a method for calculating points belonging to a straight line. This method is the same for all classes inherited from MbCurve - the curve object is passed the value of the parameter t for which it is necessary to calculate the Cartesian coordinates of the point belonging to the curve. In this example, the PNT_CNT points are calculated and plotted, the parameters of which are selected from the range [T1, T2] with equal steps. When using the range [–50, +50], the points are on opposite sides of the starting point of the straight line (which was passed to the MbLine constructor). The points corresponding to negative and positive values of t are displayed in blue and red, respectively (Fig. 2). In fig. 2 that equal intervals of t values correspond to the same intervals between the corresponding points of the line. Vector-parametric equations do not have this property for all curves. If necessary, it is possible to determine the correspondence between the values of t and the distances between points of a curve using the methods of the class MbCurve.

Example 3.2 Calculation of points belonging to a straight line

```
#include "cur_line.h"           // MbLine – Straight line in two-dimensional space
void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Factor to convert angle values from degrees to radians
  const double DEG_TO_RAD = M_PI/180.0;

  // CONSTRUCTING TWO-DIMENSIONAL STRAIGHT LINE – BY A POINT AND A DIRECTION VECTOR
  MbCartPoint p1(0, 0);
  MbDirection dir( DEG_TO_RAD * 30 );
   // Dynamically creating an object by calling the MbLine constructor
  MbLine* pLine = new MbLine( p1, dir);
  // Display of a straight line
  if ( pLine != NULL )
    viewManager->AddObject( Style( 1, RGB(255,0,0)), pLine, &pl );

  // Calculation and display of PNT_CNT points of a straight line corresponding to
  // uniformly distributed values of the parameter t from the range [T1, T2]
  const double T1 = -50;
  const double T2 = +50;
  const int PNT_CNT = 15;
  double dt = (T2 - T1)/(PNT_CNT - 1);
  MbCartPoint pnt;
  for (int i = 0; i < PNT_CNT; i++)
  {
```

```
    double t = T1 + dt * i;
    pLine->PointOn( t, pnt );
    viewManager->AddObject( Style(5, t<0 ? RGB(0, 0, 255) : RGB(255,0,0)), pnt, &pl );
  }

  // Reducing the reference count of a dynamic object
  ::DeleteItem( pLine );
}
```

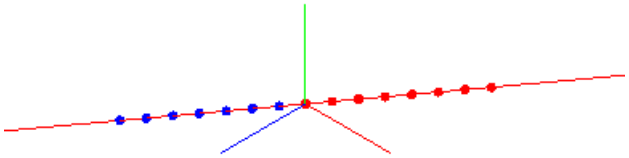


Fig. 2. Straight line MbLine, passing at an angle of 30° to the axis OX in the XY plane of the world SC. Constructed 15 points located on opposite sides of the starting point of the straight line (coincides with the origin) (Example 3.2).

3.1 Tasks

1) Construct 5 parallel straight lines using MbLine constructor with indication of a point and a direction vector.
2) Construct 5 parallel lines. To build the first straight line, use the MbLine constructor with two straight points. Construct the remaining straight lines as copies of the first one (MbLine constructor with the source straight line to be copied) followed by parallel transfer (MbLine::Move method).
3) Construct a set of lines with a common point and equal angular pitch. Set the number of lines using a constant parameter. To construct lines, use the MbLine constructor by a point and an angle.
   Note. In the test application, when viewing straight lines in the vicinity of the intersection point, do not use too large a scale. For example, if the scale factor is more than 100000 (the value of the coefficient M is displayed in the status bar), an approximate representation of screen coordinates may appear during the display. This effect is a consequence of a simplified way of displaying geometric objects in a test application and is not related to the accuracy of calculations in C3D Modeler.
4) Construct a straight line, calculate the normal vector to the straight line (using the MbCurve::Normal method, which is inherited by the MbLine class from MbCurve) and construct a perpendicular to the straight line, using the calculated normal vector as a direction vector when calling the constructor of a straight-perpendicular.
5) Construct 4 intersecting straight lines (two pairs of parallel straight lines) whose intersection points are the vertices of the rhombus. Calculate and display the intersection points. To calculate the intersection points of lines, use the LineLine function from the set of operations with points (header file action_point.h).

4. MbLineSegment – a line segment in two-dimensional space

The MbLineSegment class represents a line segment defined by a vector parametric equation:

$$\mathbf{r}(t) = (1-t)\mathbf{p_0} + t\mathbf{p_1}, \text{ где } t \in [0, 1]$$

In this equation: $\mathbf{p}_0$ и $\mathbf{p}_1$ are position vectors of line segment ends, *t* is a parameter whose values belong to a single line segment. The equation of a line segment looks like a linear combination of its vertices, and the value of the parameter t indicates the proportion in which the position vectors of the ends are added to calculate the corresponding point of the line segment. Limit values t correspond to the ends of the segment.

The MbLineSegment class is described in the cur_line_segment.h header file. Given the structure of the line segment equation, there are 2 attributes in the MbLineSegment class:

```
class MbLineSegment : public MbCurve {
private :
  MbCartPoint point1; // Starting point
  MbCartPoint point2; // End point
```

The MbLineSegment interface contains a set of constructors and Init() initialization methods, implemented MbCurve parent class methods and a set of MbLineSegment-specific methods (for example, Get / Set methods for getting and changing segment ends).

In fig. 3 there are two examples of constructions performed using the MbLineSegment segments. With the help of segments, a fractal figure was constructed - the Sierpinski triangle. The construction of this triangle is carried out by repeating half division of the sides of an equilateral triangle. Each triangle is divided into 4 parts. The central triangle is excluded from the further partitioning procedure. In example 4.1, a similar construction is implemented using segments constructed by two end points.

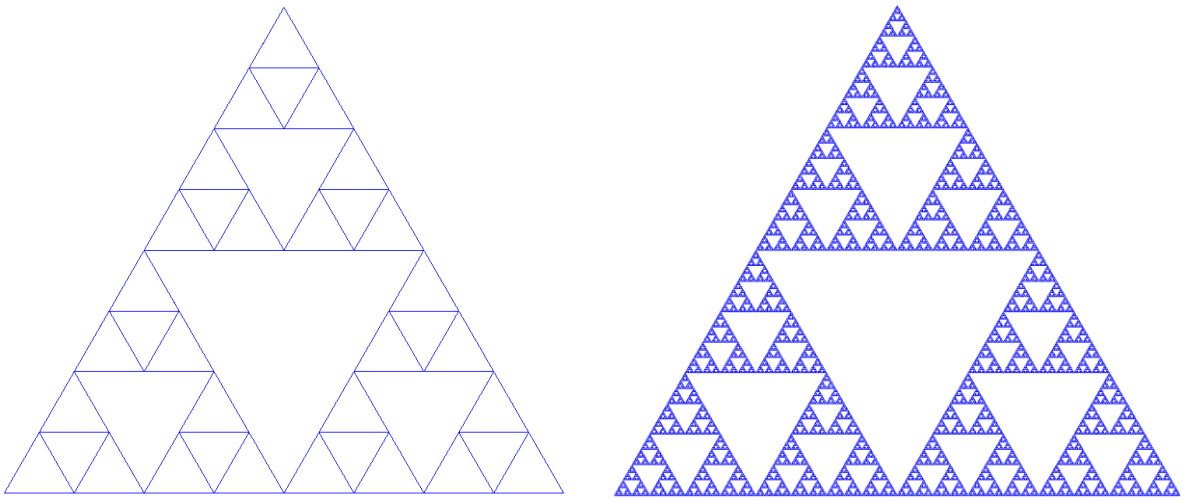


Fig. 3. The Sierpinski fractal triangle (on the left: the depth of the partition is 4, the number of triangles built is 40; on the right: the depth of the partition is 9, the number of triangles is 9841) (Example 4.1).

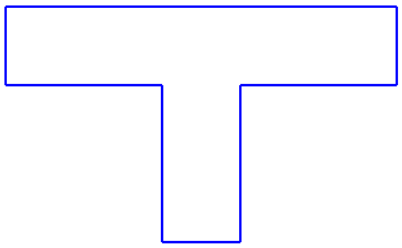
Example 4.1. Construction of a Sierpinski triangle with a given depth of partition

```
#include "cur_line_segment.h"    // MbLineSegment – 2D line segment
// std::queue – a container adapter of standard C ++ library
#include <queue>

void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Auxiliary structure for describing a triangle using the coordinates
  // of its vertices
  struct Triangle {
```

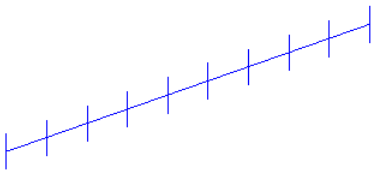
```
    MbCartPoint p1, p2, p3;
    Triangle(const MbCartPoint& _p1, const MbCartPoint& _p2, const MbCartPoint& _p3) :
          p1(_p1), p2(_p2), p3(_p3) {}
  };

  // Original equilateral triangle to split
  const double SIDE_SIZE = 10;         // The size of the side of the original triangle
  // Vertices of the original triangle (listed clockwise)
  MbCartPoint p1( 5, 5 );
  MbCartPoint p2( p1.x + SIDE_SIZE*cos(M_PI/3.0), p1.y + SIDE_SIZE*sin(M_PI/3.0 ) );
  MbCartPoint p3( p1.x + SIDE_SIZE, p1.y );
  Triangle tr( p1, p2, p3 );

  // The queue for storing triangles for later splitting
  std::queue<Triangle> arrTr;
  arrTr.push( tr );

  // Depth of splitting
  const int DIV_CNT = 4;
  // The total number of triangles as a result of splitting
  // It's calculated as the sum of a geometric progression 3^0 + 3^1 + ...3^n,
  // Sn = (q^n - 1)/(q - 1), где q = 3, n = DIV_CNT
  const int TR_CNT = ( int(pow(3, DIV_CNT)) - 1)/2;

  // Display and division of triangles
  for ( int i = 0; i < TR_CNT; i++ )
  {
    // Selection of the next triangle from the queue
    const Triangle& t = arrTr.front();

    // Segments - sides of a triangle t
    MbLineSegment* pS1 = new MbLineSegment( t.p1, t.p2 );
    MbLineSegment* pS2 = new MbLineSegment( t.p2, t.p3 );
    MbLineSegment* pS3 = new MbLineSegment( t.p3, t.p1 );

    // Calculating the midpoints of the sides of a triangle t
    MbCartPoint mid1, mid2, mid3;
    pS1->GetMiddlePoint( mid1 );
    pS2->GetMiddlePoint( mid2 );
    pS3->GetMiddlePoint( mid3 );

    // Keeping in the queue of the triangles resulting from the triangle splitting t
    arrTr.push( Triangle( t.p1, mid1, mid3 ) );
    arrTr.push( Triangle( mid1, t.p2, mid2 ) );
    arrTr.push( Triangle( mid3, mid2, t.p3 ) );

    // Display of the processed triangle t
    viewManager->AddObject( LINE_SEG_Style, pS1, &pl );
    viewManager->AddObject( LINE_SEG_Style, pS2, &pl );
    viewManager->AddObject( LINE_SEG_Style, pS3, &pl );

    // Remove the processed triangle t from the queue
    arrTr.pop();

    // Decrease the reference count of side segments t
    ::DeleteItem( pS1 );
    ::DeleteItem( pS2 );
    ::DeleteItem( pS3 );
  }
}
```

4.1 Tasks

1) Make method calls to get attributes of an object (segment) and determine the values returned by these methods (as in Example 3.1, starting with the pLine-> IsA ()). You can use the debugger in step-by-step mode to check return values.
2) Using a segment as an object, construct a T-shape:

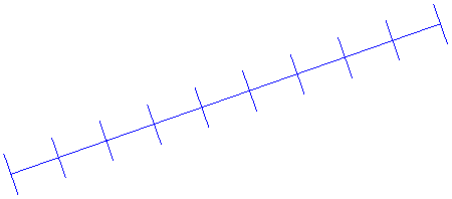


3) Split the segment with evenly spaced smaller vertical segments:



Set the number of vertical segments using a constant parameter. The location of the centers of the vertical segments can be calculated using the MbLineSegment::PointOn() method called on the split segment.

4) Modify the solution from task (3) so that the dividing segments are directed perpendicular to the one being broken. To obtain the slope of the breakable segment, use the MbLineSegment::GetAngle() method, and to rotate the breaking segments, use the MbLineSegment::Rotate(pnt, MbDirection(angle)) method, where pnt is the center of the segment (center of rotation) and angle is the angle of rotation (in radians).



5. MbArc – ellipses, circles and their arcs

The MbArc class (cur_arc.h header file) is intended to represent elliptic arcs using the following equation:

$$\mathbf{r}(t) = \mathbf{p} + a\cos t\,\mathbf{i_x} + b\sin t\,\mathbf{i_y},\ \text{where}\ \ t_{min} \le t \le t_{maxH}$$

Here **p** is the position vector of the origin of the local coordinate system of the ellipse, $\mathbf{i_x}$ and $\mathbf{i_y}$ are basis vectors of the local SC of the ellipse, a and b are ellipse semi-axes, t is a parameter specifying the point of the ellipse. The origin of the local coordinate system (point **p**) is located at the center of the ellipse, its axes ($\mathbf{i_x}$ и $\mathbf{i_y}$) are parallel to the axes of the ellipse.

The range $[t_{min}, t_{max}]$, to which the parameter t belongs, sets the start and end point of the arc. For example , the range $[0, 2\pi]$ corresponds to a full ellipse (or a circle if a = b), and $[0, \pi/4]$ corresponds to the arc of the ellipse in the first quadrant of the coordinate system.

In the above equation, there are quite a few characteristics — the location and orientation of the local SC of the ellipse, the dimensions of its semi-axes, and the boundaries of the range of the parameter t. The constructor indicating all these characteristics is cumbersome and not always convenient to use, therefore MbArc provides a set of constructors and initialization methods Init()

suitable for creating four typical varieties of elliptical arcs: circles, arcs of circles, ellipses and arcs of ellipses.

Consider the construction of several concentric circles and circular arcs (Fig. 4). In Example 5.1, MbArc constructors are called to construct a circle along the center and radius, along a radius and a point on the circle, and also arcs of circles in several ways. The constructed MbArc objects are shown in Fig. 4. In the program text of Example 5.1, they are represented by objects with the names pc1-pc7 (for the pc1 object, the radius is the largest, and for the pc7 - the smallest).

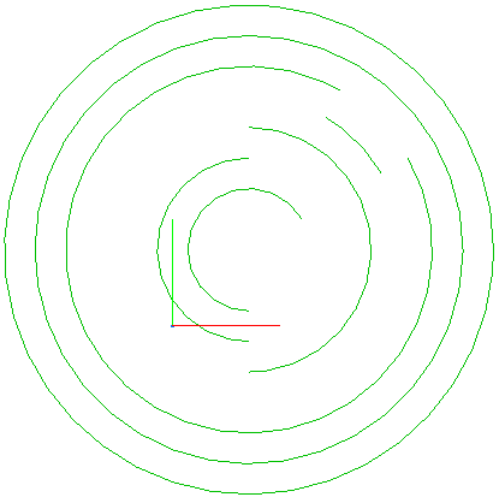


Fig. 4. Examples of circles and arcs constructed using objects of the MbArc class (Example 5.1).

Example 5.1. Construction of concentric circles and circular arcs

```
#include "cur_arc.h"      // MbArc - Elliptical arc in two-dimensional space
#include <vector>  // std::vector - dynamic array from standard C ++ library

void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Center of circles and arcs in the local SC
  MbCartPoint cnt( 5, 5 );

  // Construction of the circle by the center and radius
  const double RAD1 = 16;
  MbArc* pc1 = new MbArc(cnt, RAD1);

  // Circle by the center and the point on the circle
  const double RAD2 = 14;
  MbArc* pc2 = new MbArc( cnt, MbCartPoint(cnt.x + RAD2, cnt.y) );

  // Circular arc: set the center, radius, point-ends of the arc and the direction
  // of the walk from the first to the second end point (-1 - clockwise, +1 - against).
  // The coordinates of the point-ends of the arc are calculated using
  // coordinate functions:
  //                      x(t) = R*cos(t), y(t) = R*sin(t)
  // The position vectors of the ends of the arc are selected at an angle of 30 and 60
  // degrees to the horizontal axis..
  // Arc pc3 is built in a clockwise direction, arc pc4 is against
  const double RAD3 = 12;
  MbCartPoint p31(cnt.x + RAD3*cos(M_PI/6), cnt.y + RAD3*sin(M_PI/6) );
  MbCartPoint p32(cnt.x + RAD3*cos(M_PI/3), cnt.y + RAD3*sin(M_PI/3) );
  MbArc* pc3 = new MbArc( cnt, RAD3, p31, p32, -1 );
```

```
  const double RAD4 = 10;
  MbCartPoint p41(cnt.x + RAD4*cos(M_PI/6), cnt.y + RAD4*sin(M_PI/6) );
  MbCartPoint p42(cnt.x + RAD4*cos(M_PI/3), cnt.y + RAD4*sin(M_PI/3) );
  MbArc* pc4 = new MbArc( cnt, RAD4, p41, p42, 1 );

  // Circular arc: set the center, radius, parameters of the point-ends of the arc
  // and the direction of the arc: -1 - clockwise, +1 - against
  const double RAD5 = 8;
  MbArc* pc5 = new MbArc( cnt, RAD5, M_PI/2, 3*M_PI/2, -1 );
  const double RAD6 = 6;
  MbArc* pc6 = new MbArc( cnt, RAD6, M_PI/2, 3*M_PI/2, 1 );

  // Arc of a circle by three points.
  // The coordinates of p71, p72 and p73 points belonging to the arc of a circle
  // are calculated in the local SC of a circle.
  const double RAD7 = 4;
  MbCartPoint p71( RAD7*cos(M_PI/6), RAD7*sin(M_PI/6) );
  MbCartPoint p72( RAD7*cos(M_PI/2), RAD7*sin(M_PI/2) );
  MbCartPoint p73( RAD7*cos(3*M_PI/2), RAD7*sin(3*M_PI/2) );
  // The moving of the coordinates of points to move the center of the SC of the circle
  // to the point cnt
  p71.Move( cnt );
  p72.Move( cnt );
  p73.Move( cnt );
  MbArc* pc7 = new MbArc( p71, p72, p73 );

  // Placing pointers to circles in an array for display using a cycle
  std::vector<MbArc*> arrArcs = { pc1, pc2, pc3, pc4, pc5, pc6, pc7 };
  for (int i = 0; i < arrArcs.size(); i++)
  {
    viewManager->AddObject( ARC_Style, arrArcs[i], &pl );
    ::DeleteItem( arrArcs[i] );
  }
}
```

The construction of ellipses and elliptic arcs is performed similarly to circles. Consider the use of the MbArc class for constructing a collection of elliptical arcs, ellipses and circles that resemble the central element of an artistic parquet (Fig. 5). This pattern is based on the rosette of ellipses, for which clipping is performed along two borders - along the outer ellipse and along the inner circle. The number of ellipses in Example 5.2 is given by the constant ELLIPSE_CNT. The first ellipse for building a rosette is shown in fig. 5b. The remaining ellipses are obtained by rotating the first at a given angle around the origin, so that the poles of all ellipses of the rosette are located at the origin.

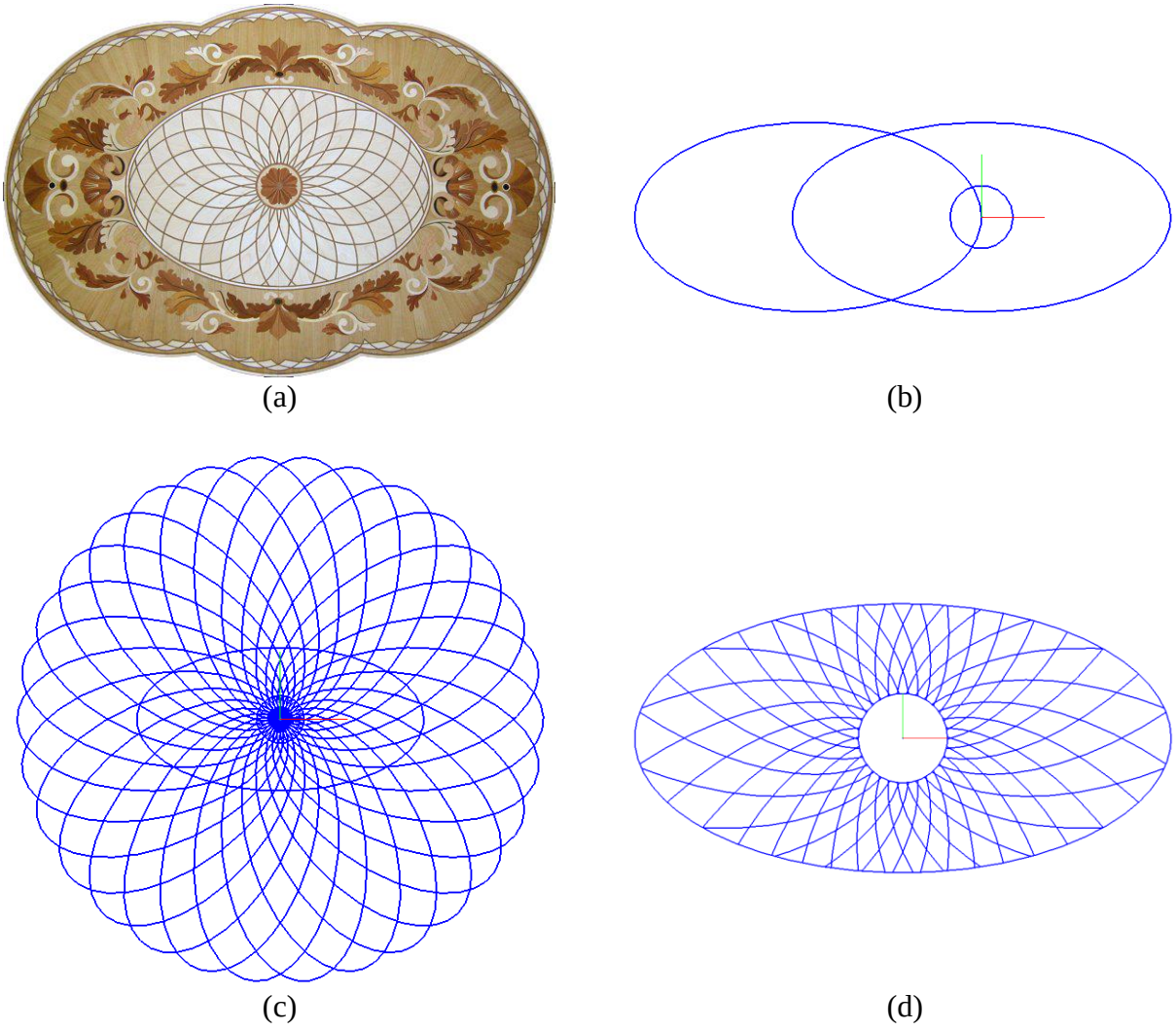


Fig. 5. The pattern of elliptical arcs (Example 5.3). (a) An example of a similar pattern on the central element of the parquet. (b) One ellipse and two borders to build a pattern. (c) Rosette of ellipses before clipping with boundary curves. (d) Pattern after clipping boundary curves.

Example 5.2. Building a pattern based on an ellipse socket

```
#include "cur_arc.h"            // MbArc - Elliptical arc in two-dimensional space
#include "mb_cross_point.h"     // MbCrossPoint - The intersection point of two curves
#include "action_point.h"       // Utilities for operations with points

void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Border 1: outer ellipse
  // The semi-axes of the ellipse are 120 and 60.
  // The local CS pl1 coincides with the system of the XOY plane of
  // the coordinate system pl.
  // In the constructor of pl1, the origin and the angle of rotation of the CS are
  // specified (as an object of the MbDirection class).
  MbPlacement pl1( MbCartPoint(0,0), MbDirection(0.0) );
  MbArc* pBound1 = new MbArc( 120, 60, pl1 );
  viewManager->AddObject( Style( 2, RGB(0,0,255) ), pBound1, &pl );

  // Border 2: Inner Circle
```

```
  MbArc* pBound2 = new MbArc( MbCartPoint(0,0), 20 );
  viewManager->AddObject( Style( 2, RGB(0,0,255) ), pBound2, &pl );

  // The first ellipse to build an rosette
  // The local CS for this ellipse is placed so that its right pole is located at
  // the point (0, 0), and the major axis lies on the axis OX
  const double RADA = 110, RADB = 60;
  MbPlacement plEllipse( MbCartPoint( -RADA, 0 ), MbDirection(0.0) );
  MbArc* pEllipse = new MbArc( RADA, RADB, plEllipse );

  // Construction of ellipses rosettes.
  // They are placed on a circle with equal angular step.
  const int ELLIPSE_CNT = 30;
  for (int i = 0; i < ELLIPSE_CNT; i++)
  {
    // The next ellipse pEl is obtained by rotating the first ellipse by angle
    MbArc* pEl = new MbArc( *pEllipse );
    double angle = 2*M_PI/ELLIPSE_CNT*i;
    pEl->Rotate(MbCartPoint(0, 0), MbDirection(angle));

    // Calculate the intersection points of pEl with the outer boundary ellipse pBound1
    // The MbCrossPoint class stores the coordinates of the intersection point,
    // the parametric coordinates of this point on both curves,
    // and the type (intersection or tangency).
    SArray<MbCrossPoint> arrExt;
    int pntCntExt = ::IntersectTwoCurves( *pEl, *pBound1, arrExt );

    // Calculate the intersection points of pEl with the internal boundary –
    // circle pBound2
    SArray<MbCrossPoint> arrInt;
    int pntCntInt = ::IntersectTwoCurves( *pEl, *pBound2, arrInt );

    // If the estimated number of intersection points is calculated (must be 2 on each
    // border), then arcs of the ellipse pEl with the ends at the found points
    // are built
    if ( pntCntExt == 2 && pntCntInt == 2 )
    {
      MbArc* pElArc1 = new MbArc( pEl->GetRadiusA(), pEl->GetRadiusB(),
                        pEl->GetPlacement(), arrInt[0].p, arrExt[0].p, 1 );
      MbArc* pElArc2 = new MbArc( pEl->GetRadiusA(), pEl->GetRadiusB(),
                        pEl->GetPlacement(), arrInt[1].p, arrExt[1].p, -1 );

      viewManager->AddObject( Style( 2, RGB(0,0,255) ), pElArc1, &pl );
      viewManager->AddObject( Style( 2, RGB(0,0,255) ), pElArc2, &pl );

      ::DeleteItem( pElArc1 );
      ::DeleteItem( pElArc2 );
    }

    // The pEl object was not transferred for display, so the next time it is called,
    // it will be deleted (because its reference count will decrease to 0)
    ::DeleteItem( pEl );
  }

  ::DeleteItem( pBound1 );
  ::DeleteItem( pBound2 );
  ::DeleteItem( pEllipse );
}
```

5.1 Tasks

1) In the cur_arc.h header file, find the MbArc class description and see which attributes are provided in it. Some of the attributes correspond to the elements of the vector-parametric equation of this geometric object, and some are intended to increase the efficiency of internal class calculations. (To quickly move to a class description in Visual Studio, in any line declaring an object on a class name, you can call the context menu and select the **Go To Definition** command in it).
2) For any object of the MbArc class, call methods to get attributes of this geometric object: IsA(), Type(), Family(), GetTMin(), GetTMax(), IsClosed(), IsPeriodic(), GetPeriod(), IsBounded( ), IsStraight() (analogous to example 3.1). Check the return values using the Visual Studio debugger in step-by-step mode.
3) Construct an ellipse and a line intersecting at two points. Calculate the intersection points (either by using the IntersectTwoCurves function, as in Example 5.2, or using the MbArc::EllipticIntersect method). Construct an arc of the ellipse with the ends at the calculated points. Find out the dependence of the result obtained on the order in which the ends of the ellipse are specified in the constructor and on the value of the traversal direction parameter.
4) Construct an ellipse and display points on it, calculated with equal step on the value of the parameter t in the range from 0 to 2π (using the MbArc::PointOn() method). Make the construction for 5 and 20 points. Using the MbArc::CalculateLength() method, calculate the distance between adjacent points along the curve and determine whether the parameterization is natural (for natural parameterization, the arc length is used as a parameter).
5) While performing task 4) it turns out that the distance between points of the curve MbArc with a uniform change of the parameter is not constant. Construct an ellipse and a set of points evenly distributed along it. To calculate the coordinates of such points, use the MbArc::GetPointsByEvenLengthDelta() method.
6) Calculating the coordinates of the points, as in task 4), construct at these points segments representing the tangent vectors and the normal vectors to the ellipse. To calculate the tangent vector and the normal vector, use the MbArc class methods inherited from MbCurve: MbCurve::Tangent() and MbCurve::Normal().

6. MbCharacterCurve – a curve defined in a symbolic form

The MbCharacterCurve class (cur_character_curve.h file) allows you to construct a curve, indicating, in a symbolic form, the coordinate functions of this curve. The curve in this class is represented using two coordinate functions x (t) and y (t):

$$\mathbf{r}(t) = [x(t) \quad y(t)], \text{ where } t_{min} \le t \le t_{max}$$

To represent scalar functions of one parameter in C3D Modeler, a special class MbFunction is intended. Both coordinate functions *x(t)* and *y(t)* must be represented as MbFunction objects and passed to the MbCharacterCurve constructor. This constructor has the following form:

```
MbCharacterCurve( MbFunction& x, MbFunction& y, MbeLocalSystemType cs,
                  const MbPlacement& place, double tmin, double tmax );
```

In addition to coordinate functions, the type of local coordinate system used (Cartesian or polar), the local coordinate system *place*, and the boundaries of the parameter value range are indicated.

The use of the MbCharacterCurve class allows us to represent curves with a known analytical expression without the need to design and implement a separate class inherited from MbCurve.

Example 6.1 demonstrates the construction of a circle using the MbCharacterCurve class. To obtain coordinate functions in the form of MbFunction objects, the function generator class MbFunctionFactory is used.

Example 6.1. Construction of a circle using coordinate functions in a symbolic form

```
// MbCharacterCurve – a curve defined in a symbolic type
#include "cur_character_curve.h"
#include "function.h"           // MbFunction - single variable scalar function
#include "function_factory.h"   // MbFunctionFactory – function object generator

void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Object is a function generator (parses the expression
  // of the function and prepares utility data structures to
  // calculate function values).
  MbFunctionFactory ff;

  // Coordinate functions in symbolic form for a circle with a center at the origin
  // and a radius of 10
  c3d::string_t sXFunc = _T("10*cos(t)");
  c3d::string_t sYFunc = _T("10*sin(t)");
  // Limits of t
  const double T_MIN = 0.0, T_MAX = 2*M_PI;

  // Coordinate functions in the form of MbFunction objects
  MbFunction* pXFunc = ff.CreateAnalyticalFunction( sXFunc, _T("t"), T_MIN, T_MAX );
  MbFunction* pYFunc = ff.CreateAnalyticalFunction( sYFunc, _T("t"), T_MIN, T_MAX );

  if ( pXFunc != NULL && pYFunc != NULL )
  {
    // Local two-dimensional SC of a curve that coincides with the
    // XOY three-dimensional system pl
    MbPlacement plCurve( MbCartPoint(0,0), MbDirection(0.0) );
    // Object - curve
    MbCharacterCurve* pCurve = new MbCharacterCurve( *pXFunc, *pYFunc, ls_CartesSystem,
                                                     plCurve, T_MIN, T_MAX );

    // Display of a curve
    if ( pCurve )
      viewManager->AddObject( Style( 1, RGB(0,0,255) ), pCurve, &pl );

    ::DeleteItem( pCurve );
  }

  // Removing dynamically created objects
  ::DeleteItem( pXFunc );
  ::DeleteItem( pYFunc );
}
```

Run Example 6.1 and verify that the circle was successfully constructed using this method. Open the function.h header file and familiarize yourself with the interface of the MbFunction class (or see its description in the help system). Pay attention to the similarity of the MbFunction interface and MbCurve interface in the part of calculating the values of the function and its derivatives.

6.1 Tasks

1) Based on Example 6.1, construct curves defined by the following pairs of coordinate functions in the range of values [t_{min}, t_{max}]:

№	Curve	$x(t)$	$y(t)$	t_{min}	t_{max}
1	Limaçon of Pascal	(1 + 2*cos(t))*cos(t)	(1 + 2*cos(t))*sin(t)	0	2π
2	Cardioid (a special case of limaçon of Pascal)	2*cos(t)-cos(2*t)	2*sin(t)-sin(2*t)	0	2π
3	Cycloid	5*(t-sin(t))	5*(1 - cos(t))	0	4π
4	Logarithmic spiral	exp(0.1*t)*cos(t)	exp(0.1*t)*sin(t)	0	10π
5	Beats (addition of vibrations with similar frequencies)	t	10*(sin(t) + sin(0.9*t))	0	50π

7. MbPolyLine – a polygonal chain

In C3D Modeler, there is a set of classes to represent curves constructed from a set of points. The parent class for such curves is the MbPolyCurve class (Fig. 1). This is an abstract class from which the following classes are inherited: MbPolyline(polygonal chain), MbBezier (Bezier curve), MbCubicSpline (cubic spline), MbHermit (composite Hermite cubic spline) and MbNurbs (NURBS curve). All of these curves play an important role in geometric modeling, in particular, the NURBS curve, which is the main way of unifying arbitrary curves. (In this paper, the MbPolyline class is considered, classes for splines will be discussed in work 3).

The MbPolyCurve class is inherited from MbCurve. For curves constructed by points, it defines a set of attributes and methods. The points along which the curve is constructed are called control points. Their location affects the shape of the curve, but the curve does not necessarily pass through all these points. The relationship between the characteristic points and the shape of the curve depends on its type. For example, the MbPolyline polyline passes through all control points, and the Bezier curve passes through the first and the last. However, regardless of the type of curve, the class stores the coordinates of all control points and provides methods for working with an array of these points.

Below is a fragment of the MbPolyCurve parent class listing some methods for control point operations.

```
class MbPolyCurve : public MbCurve {
protected :
  SArray<MbCartPoint> pointList;    // Control point array
  // Other attributes...

public :
  // Methods inherited from MbPlaneItem and MbCurve...

  // SET OF GENERAL METHODS OF POLYGONAL CURVES

  // Getting the number of control points
  Virtual size_t GetPointsCount() const;
  // Get the coordinates of the control point by its index
  virtual void   GetPoint( int index, MbCartPoint& pnt ) const;
  const MbCartPoint& GetPointList( size_t i ) const { return pointList[i]; }
  // Get the number of control points
  size_t         GetPointListCount() const { return pointList.Count(); }
  // Get the maximum index of the control points array
  int            GetPointListMaxIndex() const { return pointList.MaxIndex(); }
  // Get a copy of the control point array
  void           GetPointList( SArray<MbCartPoint>& pnts ) const { pnts = pointList; }
  // Getting the index of the control point closest to the inputed point
  virtual int    GetNearPointIndex( const MbCartPoint& pnt ) const;

  // Remove control point with given index
```

```
  virtual void   RemovePoint( int index );
  // Remove all control points
  virtual void   RemovePoints();

  // Add point to end of control point array
  virtual void   AddPoint( const MbCartPoint& pnt );
  // Insert a point in the array of control points after the point with the index.
  virtual void   AddAfter( const MbCartPoint& pnt, int index );

  // Insert a point into the array of characteristic points at the position with the
  // inputed index
  virtual void   InsertPoint(int index, const MbCartPoint& pnt ) = 0;
  // Insert into the array of control points a point that will correspond to the
  // parameter t of the curve
  virtual void   InsertPoint( double t, const MbCartPoint& pnt,
                              double xEps, double yEps ) = 0;
  // Insert a point and its derivative into the array of control points that will
correspond to the parameter t of the curve
  virtual void   InsertPoint( double t, const MbCartPoint & pnt, const MbVector& v,
                              double xEps, double yEps );

  // Change a point with the given index
  virtual void   ChangePoint(int index, const MbCartPoint & pnt );
  // Replace the array of control points
  // (the number of points must match the existing one)
  virtual bool   ChangePointsValue( const SArray<MbCartPoint>& pntList );

  // Get the number of intervals of a curve
  size_t GetSegmentsCount() const;
  // Getting the parametric coordinates of the interval,
  // which affects the control point with a given index
  virtual void GetRuleInterval( int index, double &t1, double &t2 ) const = 0;

  // Rebuilding curve
  virtual void Rebuild() = 0;

  // Other MbPolyCurve methods

};
```

With reference to a polygonal chain, control points have a simple geometric meaning: a polygonal chain consists of straight-line segments. These segments consistently connect control points. Thus, it is convenient to use polygonal chains to represent images consisting of connecting segments. Such a representation is preferable to individual MbLineSegment objects, since it allows you to avoid re-specifying matching vertices for adjacent segments. The polygonal chain can be closed, in which case the first and the last control points should coincide.

In fig. 6 there is an example of an unclosed polygonal chain obtained by copying a single rectangular pulse. The source text is presented in Example 7.1.

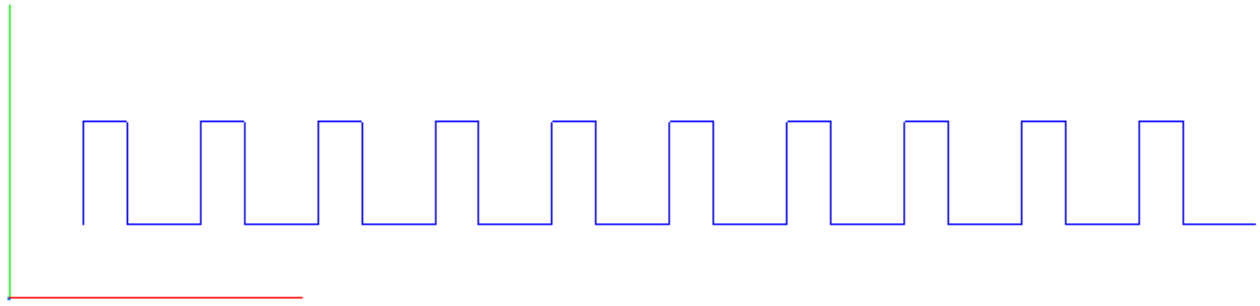


Fig. 6. A polygonal chain in the form of a sequence of 10 rectangular pulses (Example 7.1).

Example 7.1. Construction of a polyline in the form of a sequence of rectangular pulses

```
#include "cur_polyline.h"              // MbPolyline

void MakeUserCommand0()
{
  MbPlacement3D pl;// Local SC (default coincides with the global SC)

  // Characteristics of a square pulse
  const double PERIOD = 8;             // Period
  const double DURATION  = 3;          // Duration
  const double AMPL = 7;               // Amplitude

  // An array of 5 points - vertices of a polyline representing
  // one rectangular pulse with the beginning at the point (0, 0)
  SArray<MbCartPoint> arrPnts(5);
  arrPnts.Add( MbCartPoint(0, 0) );
  arrPnts.Add( MbCartPoint(0, AMPL ) );
  arrPnts.Add( MbCartPoint(DURATION, AMPL) );
  arrPnts.Add( MbCartPoint(DURATION, 0) );
  arrPnts.Add( MbCartPoint(PERIOD, 0) );

  // Polyline with vertices arrPnts
  MbPolyline* pPolyline = new MbPolyline( arrPnts, false /* Unclosed line flag */);

  // Adding impulse instances to the COPY_CNT polyline
  const int COPY_CNT = 9;
  for (int i = 0; i < COPY_CNT; i++)
  {
    // Each point in the next copy is shifted by one period.
    // These shifts are accumulated in the arrPnts array.
    for (int j = 0; j < arrPnts.size(); j++)
    {
      arrPnts[j].Move( PERIOD, 0 );
      pPolyline->AddPoint( arrPnts[j] );
    }
  }

  // The shift of the polyline so that its first point is located at (5, 5)
  pPolyline->Move( MbVector(5, 5) );

  // Display tha polyline
  viewManager->AddObject( Style( 1, RGB(0,0,255) ), pPolyline, &pl );

  ::DeleteItem( pPolyline );
}
```

The construction of a closed polyline is demonstrated in Example 7.2. It draws the border of a polygon inscribed in a circle (Fig. 7). The coincidence of the initial and final control points is provided by specifying the flag of the closed line in the polyline construstor.

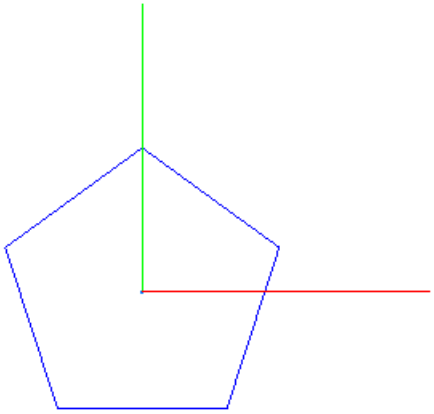


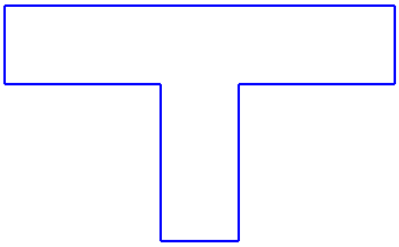
Fig. 7. A polyline representing the boundary of a pentagon (Example 7.2).

Example 7.2. Construction of the boundary of a polygon inscribed in a circle

```
#include "cur_polyline.h"       // MbPolyline

void MakeUserCommand0()
{
  MbPlacement3D pl;       // Local SC (default coincides with the global SC)

  // The number of sides of the polygon
  const int SIDE_CNT = 5;
  // The radius of the polygon circumscribing circle
  const double RAD = 5.0;

  // Array to store the vertices of the polyline
  SArray<MbCartPoint> arrPnts(SIDE_CNT);

  // Calculation of polyline vertices by uniform division of a circle
  for (int i = 0; i < SIDE_CNT; i++)
  {
    // The angular position of the i-th vertex on the circumscribing circle.
    // The angular position of the initial vertex - M_PI/2 (this vertex is located
    // on the vertical axis).
    double angle = M_PI/2 + 2*M_PI/SIDE_CNT*i;
    MbCartPoint pnt( RAD*cos(angle), RAD*sin(angle) );
    arrPnts.Add( pnt );
  }

  // Closed polyline with vertices stored in array arrPnts
  MbPolyline* pPolyline = new MbPolyline( arrPnts, true /* Unclosed line flag */ );

  // Display the polyline
  if ( pPolyline )
    viewManager->AddObject( Style( 1, RGB(0,0,255) ), pPolyline, &pl );

  ::DeleteItem( pPolyline );
}
```

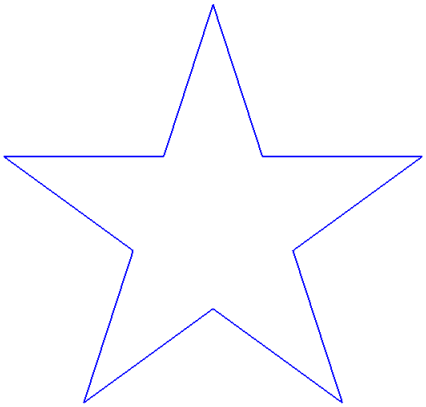
7.1 Tasks

1) Familiarize yourself with the MbPolyline class interface in C3D Modeler help system and in the cur_polyline.h header file. Find a list of MbPolyline::Init() constructors and methods representing various ways of initializing a new or existing polyline object.
2) Construct a closed polyline depicting the border of a T-shaped figure (this line was previously constructed as a set of segments in Section 4.1):



   Call the properties window for the constructed shape in the test application. Make sure that this is a single geometric object containing an array of points, the number of which coincides with the number of vertices of the polyline.
3) Construct an open closed polyline of the following form:

4) Using Example 7.2, draw polygons with different numbers of sides: 3, 4, 5, 6 7, 9, 50.
5) For a closed polyline representing a polygon, call methods to get attributes of this geometric object: IsA(), Type(), Family(), GetTMin(), GetTMax(), IsClosed(), IsPeriodic(), GetPeriod(), IsBounded(), IsStraight() (similar to example 3.1). Check the returned values using the Visual Studio debugger in step-by-step mode. Note the difference in the values returned by the MbPolyline::Type () and MbPolyline::Family () methods.
6) For polyline representing pentagon boundaries (Example 7.2), calculate the coordinates of the polyline points corresponding to the integer values of the parameter t in the range from MbPolyline::GetTMin() to MbPolyline::GetTMax() using the MbPolyline::PointOn() method. Display these points.
7) Construct a closed star shaped line. (The end vertices of the polyline coincide with the vertices of a regular pentagon. To calculate the location of the internal vertices, you can use an auxiliary construction — calculate the intersection points of lines containing the sides of a star.)



8. Conclusion

In this paper, we considered the methods of representing the simplest curves in two-dimensional space: straight lines, segments, polygonal chains, ellipses, circles, arcs of ellipses and circles. Were also described how to set curves using coordinate functions in a symbolic form.

All curves in two-dimensional space in C3D Modeler are constructed on the basis of the common parent class MbCurve, calculated on the representation of curves using vector-parametric equations. This unification allows one-way image to calculate the points of the curves using the PointOn() method, as well as the values of derivatives, the tangent and normal vectors for an arbitrary value of the parameter t from the domain of definition. The inheritance from MbCurve classes representing different curves allows to use these curves in the same type in functions that implement algorithms of computational geometry (for example, to determine the intersection points of curves).

In this paper, not all features of C3D Modeler for representation and operations with curves are considered. A significant role in geometric modeling is played by spline curves, as well as composite curves constructed from fragments of different curves of the same type or of different types. These issues are addressed in the following paper.