
# Getting started with C3D Modeler

**LESSON 5**

## Construction of swept surfaces


https://c3dlabs.com
Moscow
2019

# Table of Contents

# 1. Introduction

Swept surfaces form one of the most commonly used surfaces in geometric modeling. In particular, they are convenient for constructing surface models with the properties of rotational or mirror symmetry. Such models are used to represent the shape of many engineering parts, for example, manufactured through the use of metalworking technologies based on the impact of cutting tools moving along specified trajectories.

In this lesson, we use methods for creating the most common types of surfaces: extrusion, rotation and kinematic surfaces. C3D Modeler demonstrates the construction of surface models consisting of several fragments of various surfaces. In many cases, instead of surface models, solid models are used. This lesson also demonstrates the construction of thin-walled solid-state models.

# 2. Swept surfaces

Table 1 lists the C3D Modeler classes for representing the swept surfaces and the sets of properties that are specified when creating objects of these classes. All the classes listed are inherited from the abstract parent class MbSweptSurface. The inheritance hierarchy and examples of classes for representing the swept surfaces were considered in Lesson No. 4, paragraph 4.2.

**Table 1.** Classes to represent swept surfaces.

| № | Class | Header file | Surface code (MbSurface::IsA()) | Sets of values for different ways of construction |
|---|---|---|---|---|
| 1 | MbExtrusionSurface (an extrusion surface) | surf_extrusion_surface.h | st_ExtrusionSurface | 1) Generator and directional vector. |
| 2 | MbRevolutionSurface (a rotation surface) | surf_revolution_surface.h | st_RevolutionSurface | 1) Generator, origin of local SC, Z axis of local SC, angle of rotation.<br>2) Generator, axis and angle of rotation.<br>3) Generator, axis of rotation, minimum and maximum angle. |
| 3 | MbExpansionSurface (plane-parallel kinematic surface) | surf_expansion_surface.h | st_ExpansionSurface | 1) Generator and guide curve.<br>2) Point, generator and guide curve. |
| 4 | MbSpiralSurface (a spiral surface) | surf_spiral_surface.h | st_SpiralSurface | 1) Generator, local SC, radius and step of the spiral.<br>2) Generator and spiral curve – guide curve. |
| 5 | MbEvolutionSurface (a kinematic surface) | surf_evolution_surface.h | st_EvolutionSurface | 1) Generator and guide curve. |
| 6 | MbExactionSurface (a kinematic surface with adaptation) | surf_exaction_surface.h | st_ExactionSurface | 1) Generator, guide curve and normal vectors to the side faces at the end points of the guide. |

In Table 1, it can be noted that the methods of construction of all the swept surfaces are of the same type — either the generator curve and the guide curve are indicated explicitly, or the geometrical objects that uniquely define the generator curve and guide line are indicated. In addition to explicitly calling the constructors of classes of swept surfaces, you can use utility functions to construct these surfaces, which are described in the action_surface.h file. These functions are listed in Table 2. Each

of these functions returns the MbResultType value — in the case of successful surface construction, this value equals rt_Success. The constructed surface is returned as a dynamically created object in the form of a pointer to the base class MbSurface. If necessary, using the C ++ operator of the dynamic_cast <> () operator, MbSurface* pointer can be converted to a pointer to the corresponding specific surface class. However, it is usually not necessary to perform such an operation, since all basic operations with surfaces are provided by the interface methods of the parent class MbSurface.

**Table 2.** Functions utilities for constructing swept surfaces (action_surface.h)

| № | Class | Function | Input parameters of the function |
|---|---|---|---|
| 1 | MbExtrusionSurface (an extrusion surface) | ::ExtrusionSurface() | Generating curve and directional vector (the length of the vector sets the length og the directional line segment). |
| 2 | MbRevolutionSurface (a rotation surface) | ::RevolutionSurface() | Generating curve, axis and angle of rotation. |
| 3 | MbExpansionSurface (a plane-parallel kinematic surface) | ::ExpansionSurface() | Generating and guiding curves. |
| 4 | MbSpiralSurface (a spiral surface) | ::SpiralSurface() | Generating curve, local SC, step of the spiral. |
| 5 | MbEvolutionSurface (a kinematic surface) | ::EvolutionSurface() | Generating and guiding curves. If the guide is the spiral curve MbCurveSpiral, then the spiral surface will be constructed. |
| 6 | MbExactionSurface (a kinematic surface with adaptation) | Requires the use of class constructors. | |

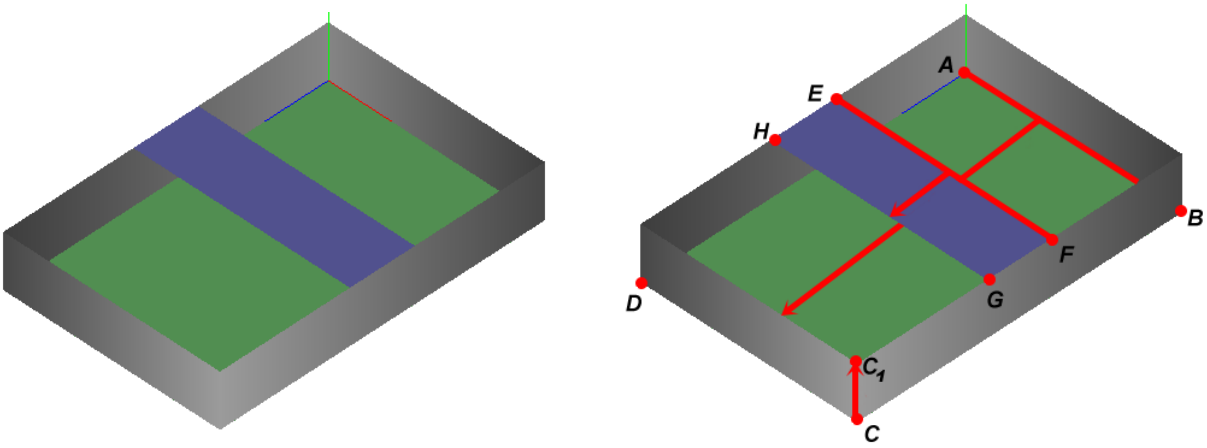
# 3. MbExtrusionSurface – an extrusion surface

Extrusion surfaces are convenient for constructing surface models consisting of fragments with pairs of parallel rectilinear boundaries. To describe the shape of an object, you may need several different fragments with common borders. When constructing individual fragments using classes of surfaces, the correctness of their conjugation is required to ensure explicitly. In contrast to this approach, when building models of solids, the necessary surface fragments and their conjugation along common borders (edges) is provided by the geometrical kernel automatically.

Consider the use of the extrusion surfaces MbExtrusionSurface for constructing a surface model of a rectangular box with a crossbar (Fig. 1). The constructed model is shown in Fig. 1 (a). There are marked with auxiliary elements explaining the construction in fig. 1 (b). The box is defined by three parameters: width AB, height $CC_1$ and depth BC. The surface model consists of three extrusion surfaces:

1) Side surface with a rectangle ABCD as generating curve and a vector $\mathbf{CC_1}$ as a guide curve.
2) The bottom of the box - a rectangle ABCD - with a generating segment AB and a guide in the form of a vector with a length BC, directed along the Z axis.
3) Transverse - rectangle EFGH - with the generating segment EF and the guide in the form of a vector with length EH, directed along the Z axis.

Methods for selecting generators and guides for constructing the listed surfaces are not the only possible ones. For example, to build the box bottom (rectangle ABCD), we could take the vector AD and the vector length AB, parallel to the X axis, as the side AD. We cannot use the elementary MbPlane surface to represent rectangular fragments, since MbPlane is an infinitely long plane. The MbPlane class is convenient for performing calculations and auxiliary constructions, but is not suitable for representing rectangles.




(a) (b)

**Fig. 1.** Constructing a surface model from three extrusion surfaces MbExtrusionSurface. (a) Model box with cross bar. (b) Auxiliary constructions explaining the choice of generators and guides.

The program fragment for constructing the model shown in Fig. 1 (a) is given in Example 3.1. To construct extrusion surface objects, use the utility function ::ExtrusionSurface().

*Note:* when constructing the side surface of the model, a rectangle was used. In the general case, it is undesirable to specify generators with angular points, i.e. with gaps in the meaning of the first derivative. For example, such surfaces are not suitable for constructing surfaces bounded by curves on their base. Therefore, in cases where the extrusion surface is used as an intermediate step for building more complex surfaces, it must be constructed using smooth generators. This usually requires splitting the surface into several parts. For example, the side surface in Example 3.1 could be constructed as a set of four rectangular fragments.

***Example 3.1. Construction of extrusion surfaces (Fig. 1).***

```
#include "surf_extrusion_surface.h"
#include "cur_line_segment3d.h"
#include "cur_polyline3d.h"
#include "action_surface.h"

void MakeUserCommand0()
{
  // Parameters of the model of the box
  //   Width, height and depth are the dimensions along the axes X, Y и Z
  const double BOX_WIDTH = 10, BOX_HEIGHT = 2.5, BOX_DEPTH = 15;
  //   Crossbar width - 20% of the box depth
  const double BAR_SIZE = BOX_DEPTH/5;

  // 1) Construction of the side surface
  MbSurface* pSurfBox = NULL;
  // Vertices of the Rectangle ABCD (box bottom)
  std::vector<MbCartPoint3D> arrPolyPntsBox = { { 0, 0, 0 }, { BOX_WIDTH, 0, 0 },
                                { BOX_WIDTH, 0, BOX_DEPTH }, { 0, 0, BOX_DEPTH } };
  // Generator to construct a side surface is the contour of the bottom of the box in
  // the form of a broken line
  MbPolyline3D* pGenCurveBox = new MbPolyline3D( arrPolyPntsBox, true );
  // Guide vector is parallel to the Y axis, length is equal to the height of the
  // box CC1
  MbVector3D vecDirBox( 0, BOX_HEIGHT, 0 );
  // Calling a utility function to construct an extrusion surface
  MbResultType resBox = ::ExtrusionSurface( *pGenCurveBox, vecDirBox, true, pSurfBox );
```

```
  // 2) Constructing the bottom of the box - rectangle ABCD
  MbSurface* pSurfBottom = NULL;
  // Generating AB segment
  MbLineSegment3D* pGenCurveBottom =
            new MbLineSegment3D( arrPolyPntsBox[0], arrPolyPntsBox[1]  );
  // The guide vector is parallel to the Z axis, the length is equal to the depth of
  // the box BC
  MbVector3D vecDirBottom( 0, 0, BOX_DEPTH );
  MbResultType resBottom = ::ExtrusionSurface( *pGenCurveBottom,
                                               vecDirBottom, true, pSurfBottom );

  // 3) Construction of the transverse element
  MbSurface* pSurfBar = NULL;
  // The generator - the segment EF - is constructed by means of the parallel transfer
  // of the segment AB to the vector AE
  MbLineSegment3D* pGenCurveBar =
            new MbLineSegment3D( arrPolyPntsBox[0], arrPolyPntsBox[1]  );
  pGenCurveBar->Move( MbVector3D(0, BOX_HEIGHT, (BOX_DEPTH - BAR_SIZE)/2 ) );
  // The guide vector is parallel to the Z axis, the length is equal to the width of
  // the transverse element
  MbVector3D vecDirBar( 0, 0, BAR_SIZE );
  MbResultType resBar = ::ExtrusionSurface( *pGenCurveBar, vecDirBar, true, pSurfBar );

  // Display surface
  viewManager->AddObject(Style(1, LIGHTGRAY), pSurfBox );
  viewManager->AddObject(Style(1, GREEN), pSurfBottom );
  viewManager->AddObject(Style(1, BLUE), pSurfBar );

  // Reducing the reference count of dynamically created kernel objects
  ::DeleteItem( pGenCurveBox);
  ::DeleteItem( pSurfBox );
  ::DeleteItem( pGenCurveBottom );
  ::DeleteItem( pSurfBottom );
  ::DeleteItem( pGenCurveBar );
  ::DeleteItem( pSurfBar );
}
```

The construction of extrusion surfaces is often one of the intermediate modeling operations. Consider the problem of constructing a model of a bounded surface, shown in Fig. 2. A surface fragment with a hole can be represented as a surface bounded by curves MbCurveBoundedSurface (Work No. 4, Section 4.6). The extrusion surface will be used as the base. The possible order of construction consists of the following steps:

1) Construction of the base extrusion surface.
2) Construction of an internal circular contour on a basic surface.
3) Getting a contour representing the outer boundary of the base surface.
4) Construction of the surface bounded by two curves: the outer contour of the base surface and the contour of the hole.

The execution of these actions is shown in Example 3.2.

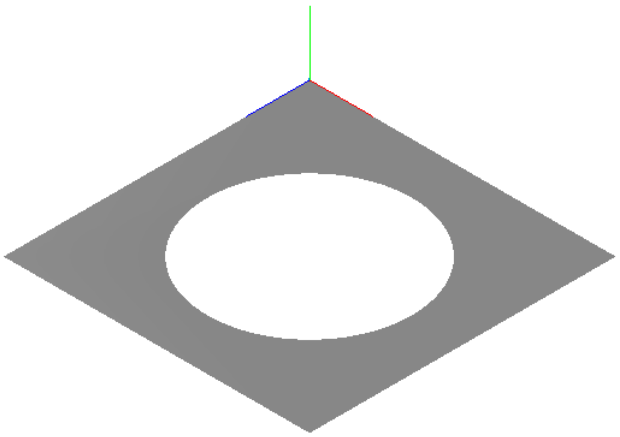


**Fig. 2.** Surface model based on extrusion surface.

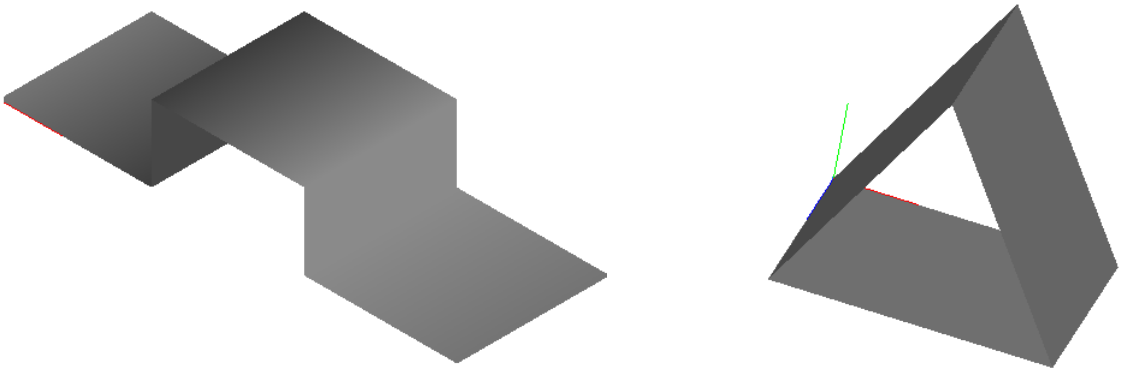
***Example 3.2. Extrusion surface with a hole (Fig. 2).***

```
#include "surf_extrusion_surface.h"
#include "cur_line_segment3d.h"
#include "cur_contour.h"
#include "cur_arc3d.h"
#include "action_surface.h"
#include "action_curve.h"

void MakeUserCommand0()
{
  // The size of the side of the surface fragment in the form of a square
  const double SIZE_SQUARE = 5;
  // Inner center hole radius
  const double HOLE_RADIUS = SIZE_SQUARE/3;

  // 1) CONSTRUCTION OF THE BASIC EXTRUSION SURFACE
  MbSurface* pSurfBase = 0;
  {
    // Generator: a segment on the X axis with a length of SIZE_SQUARE
    MbLineSegment3D* pGenCurve =
        new MbLineSegment3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(SIZE_SQUARE,0,0));
  // Guide vector - pointing towards + Z
  MbVector3D vecDir( 0, 0, SIZE_SQUARE );
  // Calling a function to construct an extrusion surface
  ::ExtrusionSurface( *pGenCurve, vecDir, true, pSurfBase );
  }

  // 2) BUILDING A CIRCLE CONTOUR ON SURFACE
  MbContour* pArcContour = 0;
  {
    // Center hole and a pair of points on the circle in the global SC
    MbCartPoint3D pc( SIZE_SQUARE/2, 0, SIZE_SQUARE/2 );
    MbCartPoint3D p1 = pc, p2 = pc;
    p1.x += HOLE_RADIUS;
    p2.z += HOLE_RADIUS;

    // Auxiliary space curve - hole contour
    MbArc3D* pArcCurve = new MbArc3D( pc, p1, p2, 0, true );
    // Projecting a space curve to the surface to get a contour on the surface
    ::SurfaceBoundContour( *pSurfBase, *pArcCurve, Math::DefaultMathVersion(),
                           pArcContour );
    ::DeleteItem( pArcCurve );
  }
```

```
  // 3) OBTAINING THE CONTOUR - THE EXTERNAL BORDER OF THE BASIC SURFACE
  MbContour* pExtContour = &pSurfBase->MakeContour(true);

  // 4) CONSTRUCTION OF THE SURFACE LIMITED BY A PAIR OF CONTOURS
  MbSurface* pSurf = 0;
  {
    RPArray<MbCurve> arrBounds;
    arrBounds.Add( pExtContour );
    arrBounds.Add( pArcContour );
    // Constructing a surface pSurf based on an extrusion surface pSurfBase
    ::BoundedSurface( *pSurfBase, arrBounds, pSurf );
  }

  // Display surface
  viewManager->AddObject(Style(1, LIGHTGRAY), pSurf );

  // Reducing reference counts for dynamically created kernel objects
  ::DeleteItem( pSurfBase );
  ::DeleteItem( pSurf );
  ::DeleteItem( pArcContour );
  ::DeleteItem( pExtContour );
}
```

## *3.1 Tasks*

1) Develop a function for constructing a fragment of a cylindrical surface in the form of an extrusion surface with a generatrix - a circular arc. Using this function, build a cylindrical surface, as well as a surface with a quarter and a third in the shape of a circular arc.
2) Construct the surface of the elliptical cylinder as an extrusion surface with an ellipse generator. Arrange these actions as a separate function and construct using it several adjacent cylindrical surfaces with parallel axes of symmetry.
3) Construct the extrusion surface with a cosine shaped form with three periods. To represent the generatrix, use a Bezier or NURBS curve constructed from a set of points.
4) Construct a surface model consisting of five rectangular fragments (Fig. 3 (a)). Build a rectangle implement as a separate function.
5) Construct a surface model of rectangular fragments that form a triangular shape (Fig. 3 (b)). To build rectangles, use the function developed for task 4.



(a) (b)

**Fig. 3.** Samples of surface models for task 4 (model (a)) and for task 5 (model (b)).

# 4. MbRevolutionSurface – a rotation surface

Surfaces of revolution are intended to represent surface models with rotational symmetry. Such models are widely distributed, in particular, for the representation of machine-building parts manufactured by turning. The MbRevolutionSurface class is used to represent such surfaces. This class has several constructors that allow you to specify a generator, axis and angle of rotation in various ways. However, as in other similar cases, it is more convenient to use a utility function that monitors the correctness of the passed parameters. The description of this function for constructing the rotation surface is contained in the header file action_surface.h:

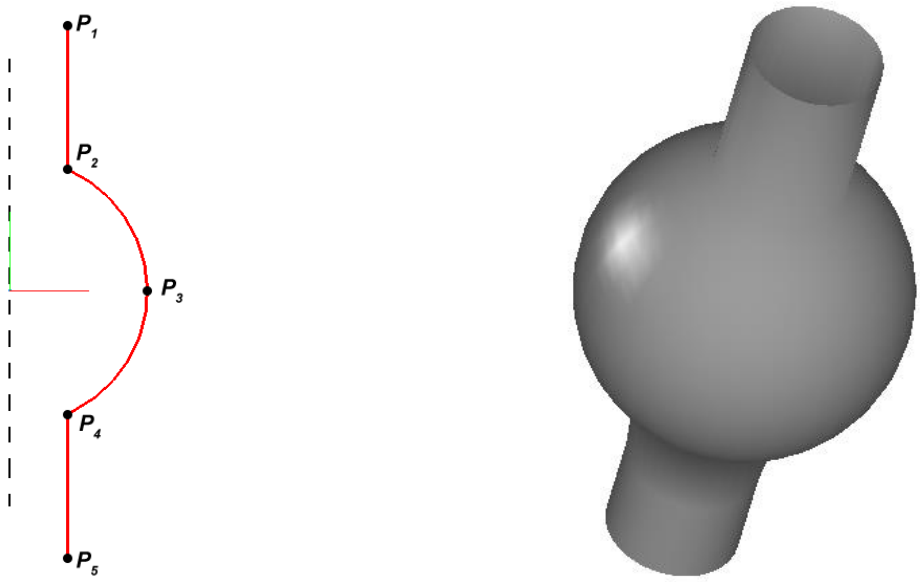
```
MbResultType ::RevolutionSurface( MbCurve3D& curve, const MbCartPoint3D& origin,
             const MbVector3D& axis, double angle, bool simplify, MbSurface*& result );
```

The generator curve is passed as a curve object. The rotation axis is given by two input parameters, origin and axis. The axis object alone is not enough, because the MbVector3D class represents a free vector in three-dimensional space. The combination of the origin and axis parameters can be considered as a fixed vector (directional segment) that uniquely determines the axis of rotation. Thus, the rotation axes are defined by a pair of vertices of the directed segment origin and (origin.x + axis.x, origin.y + axis.y, origin.z + axis.z). The angle of rotation of the generatrix angle around the axis is specified in radians. The simplify parameter is used to enable automatic simplification of the surface model.

The output parameter of the ::RevolutionSurface function is the result object representing the constructed rotation surface. For consistency, the returned object is represented as a pointer to the MbSurface parent class. As a rule, there is no need to convert the result from the MbSurface* type pointer to MbRevolutionSurface* - the main features of the rotation surface class are concentrated in the implementation of the virtual methods of the base class. MbRevolutionSurface's specific new methods are just a small set of methods for obtaining class attributes — parameters for constructing a surface of revolution (generator and axis of rotation).

To check the correctness of the function call ::RevolutionSurface, you can check the returned object - the result pointer should not be zero, as well as the return value of a function of the MbResultType type - in the case of a successful call it should be equal to rt_Success.

Consider the use of the surface of revolution to construct a model in the form of a combination of intersecting spherical and cylindrical surfaces (Fig. 4 (b)). We will assume that the axis of the cylinder passes through the center of the sphere, and that the centers of gravity of the cylinder and the sphere coincide. In this case, the desired surface will have the form of a surface of revolution with a generator consisting of a circular arc and segments - that is, half of the surface section of the plane containing the axis of the cylinder. The constructed surface and its forming curve are shown in Fig. 4. Program calls that implement the construction are shown in Example 4.1.




(a) (b)

**Fig. 4.** The surface of rotation (b), constructed using the represented generatrix curve (a) (Example 4.1).

***Example 4.1. The surface of rotation in the form of a combination of spherical and cylindrical surfaces (Fig. 4).***

```
#include "cur_line_segment3d.h"
#include "cur_arc3d.h"
#include "cur_contour3d.h"
#include "action_surface.h"
#include "surf_revolution_surface.h"

// Construction of the surface of revolution in the form of a combination of a cylinder
// and a sphere with coincident centers of gravity.
MbSurface* CreateCylSphereSurface()
{
  // Cylinder and Sphere Parameters
  const double CYL_RAD    = 2.0;
  const double CYL_HEIGHT = 20.0;
  const double SPHERE_RAD = 5.0;

  // Generating curve is a contour consisting of two segments and a circular arc
  // between them
  // The points are the vertices of the generatrix segments (Fig. 4)
  MbCartPoint3D p1( CYL_RAD, CYL_HEIGHT/2, 0 );
  MbCartPoint3D p2( CYL_RAD, sqrt( SPHERE_RAD*SPHERE_RAD - CYL_RAD*CYL_RAD ), 0 );
  MbCartPoint3D p3( SPHERE_RAD, 0, 0 );
  MbCartPoint3D p4( p2.x, -p2.y, p2.z );
  MbCartPoint3D p5( p1.x, -p1.y, p1.z );
  // Generating curve segments
  MbLineSegment3D* pSeg1 = new MbLineSegment3D( p1, p2 );
  MbArc3D* pArc = new MbArc3D( p2, p3, p4, 1, false );
  MbLineSegment3D* pSeg2 = new MbLineSegment3D( p4, p5 );
  // Construction of a generatrix curve in the form of a contour from three segments
  RPArray<MbCurve3D> arrCurves;
  arrCurves.Add( pSeg1 );
  arrCurves.Add( pArc );
  arrCurves.Add( pSeg2 );
  MbContour3D* pGenContour = new MbContour3D( arrCurves, true );

  // Parameters of the axis of rotation - it coincides with the axis of the cylinder
```

```
  // and the axis Y of the local SC, in which the vertices of the segments of the
  // generator curve p1-p5 were set.
  MbCartPoint3D axOrg( 0, 0, 0 );
  MbVector3D axDir( 0, 1, 0 );

  // Construction of the rotation surface
  MbSurface* pSurf = NULL;
  MbResultType res = ::RevolutionSurface( *pGenContour, axOrg, axDir,
                                          2*M_PI, true, pSurf );
  if ( res != rt_Success )
  {
    // Return in case of error when constructing the surface
    ::DeleteItem( pGenContour );
    ::DeleteItem( pSurf );
    return NULL;
  }

  ::DeleteItem( pGenContour );
  return pSurf;
}


void MakeUserCommand0()
{
  // Surface construction is implemented in a separate function
  MbSurface* pSurf = CreateCylSphereSurface();

  // Display surface
  viewManager->AddObject( Style(1, LIGHTGRAY), pSurf );

  // Reducing reference counts for dynamically created kernel objects
  ::DeleteItem( pSurf );
}
```

## *4.1 Construction of surface model from several parts*

Based on Example 4.1, we construct a more complex surface model. We assume that the surface of rotation shown in Fig. 4 (b) is located in the coordinate system, the center of which coincides with the center of the sphere, and the Y axis - with the axis of the cylinder (in this SC in example 4.1, the surface was constructed). Suppose that this surface is complemented by a side surface of a rectangular prism with a square base. The model that you want to construct, compared with the model in Fig. 4 (b), complemented by fragments of the surface of the prism lying outside the sphere, and a pair of square holes on the surface of the sphere. Part of the surface of the prism that falls inside the sphere is excluded from the model.

To construct the required model (Fig. 5 (a)) the following basic actions will be performed:

1) Construction of a part of the model as a combination of spherical and cylindrical surfaces (Fig. 5 (b)). To do this, use the CreateCylSphereSurface() function, which was implemented in Example 4.1.
2) Construction of two contours, limiting the part of the prism outside the scope. One circuit is the square base of the prism, which lies in a plane parallel to ZY and offset in the -X direction by half the height of the prism. The second contour is the line of intersection of the surface of the prism and the spherical surface. These outlines are shown in Fig. 5 (c).
3) Construction of the MbRuledSurface ruled surface using two curves - contours, calculated in step (2). This surface will represent one of two fragments of the surface of the prism lying outside the surface of the sphere. The resulting surface is shown in Fig. 5 (d). This surface cannot be

constructed as an extrusion surface, since it cannot be used to indicate a generating curve of constant shape, which could be moved to construct along any path.

4) Repeating steps (2) and (3) to construct a ruled surface representing the second fragment of a prism outside the sphere.
5) In the fragment of a spherical surface it is necessary to construct holes corresponding to the lines of intersection of the surfaces of the sphere and the prism. To do this, on the basis of the surface constructed in step (1), the surface is bounded by contours. As a pair of bounding contours, contours are specified that correspond to the intersection lines of the prism surface and the sphere (they were calculated in the process of performing steps (2) and (4)). The procedure for constructing a surface with a hole is similar to Example 3.2.

Software implementation of the above actions is given in Example 4.2.

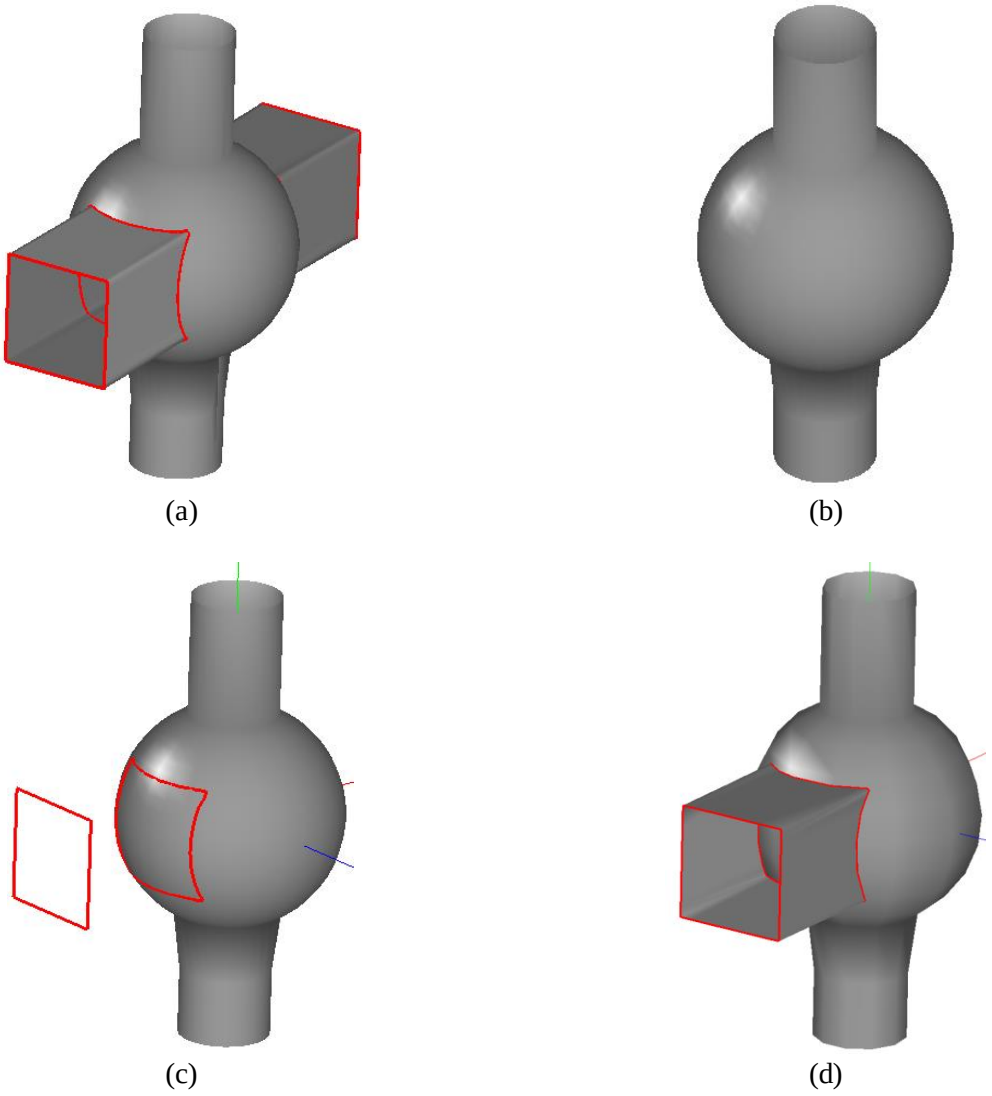



**Fig. 5.** Building a surface model of four surfaces: two surfaces of revolution (representing a spherical and cylindrical surfaces) and two ruled surfaces (representing fragments of the surface of a prism lying outside the sphere). (a) Built model. (b) A part of the model is a surface of revolution constructed similarly to Example 4.1. (c) Contours for constructing the first ruled surface — the base of the prism and its projection onto the surface of the sphere. (d) The ruled surface, constructed on the contours shown in fig. (c).

***Example 4.2. Model of two surfaces of revolution and two ruled surfaces (fig. 5).***

```
#include "cur_line_segment3d.h"
#include "cur_polyline3d.h"
#include "cur_contour3d.h"
#include "cur_arc3d.h"
#include "cur_contour.h"
#include "action_surface.h"
#include "surf_revolution_surface.h"
#include "action_surface_curve.h"
#include "action_curve.h"

void MakeUserCommand0()
{
  // Parameters of a rectangular prism that will complement the pCylSphere surface
  const double PRISM_SIDE = 5;         // Side of the square base of the prism
  const double PRISM_HEIGHT = 20;      // Prism height

  // 1) CONSTRUCTION OF A PART OF A MODEL AS A COMBINATION OF SPHERICAL AND CYLINDRICAL
  // SURFACES (the function implemented in Example 4.1 is called)
  MbSurface* pCylSphere = CreateCylSphereSurface();
  MbAxis3D axVert( MbVector3D( 0, 1, 0 ) );
  pCylSphere->Rotate( axVert, M_PI/2 );

  // 2) CONSTRUCTION OF THE COUPLE OF CONTOURS, LIMITING ONE OF THE PARTS OF PRISM
  // OUTSIDE SPHERES
  // The coordinates of the vertices of the base of the prism (in a plane parallel to
  // ZY, and shifted to half the height in the direction -X)
  std::vector<MbCartPoint3D> arrPnts_Base1 = {
            { -PRISM_HEIGHT/2,  PRISM_SIDE/2, -PRISM_SIDE/2 },
            { -PRISM_HEIGHT/2,  PRISM_SIDE/2,  PRISM_SIDE/2 },
            { -PRISM_HEIGHT/2, -PRISM_SIDE/2,  PRISM_SIDE/2 },
            { -PRISM_HEIGHT/2, -PRISM_SIDE/2, -PRISM_SIDE/2 }
  };

  // Contour - closed polyline to represent the base of a prism
  MbPolyline3D* pContour_Base1 = new MbPolyline3D( arrPnts_Base1, true );

  // Projection of the pContour_Base1 space contour onto the pCylSphere surface
  RPArray< MbCurve3D > arrProjCurves_Base1;
  MbVector3D dirProj1( +1, 0, 0 );
  ::CurveProjection( *pCylSphere, *pContour_Base1, &dirProj1, false, false,
                     arrProjCurves_Base1 );

  // 3) CONSTRUCTION OF THE FIRST FRAGMENT OF THE SURFACE OF PRISM
  // The pSurfPrism1 surface is constructed as a ruled surface defined by a pair
  // of curves.
  MbSurface* pSurfPrism1 = 0;
  ::RuledSurface( *pContour_Base1, *arrProjCurves_Base1[0], true, pSurfPrism1 );

  // 4) CONSTRUCTION OF THE SECOND FRAGMENT OF THE SURFACE OF PRISM
  // The second base of the prism is offset from the first base by the height
  // of the prism
  MbPolyline3D* pContour_Base2 = new MbPolyline3D( arrPnts_Base1, true );
  pContour_Base2->Move( MbVector3D( PRISM_HEIGHT, 0, 0) );

  // Projecting the pContour_Base2 space contour onto the pCylSphere surface
  RPArray< MbCurve3D > arrProjCurves_Base2;
  MbVector3D dirProj2( -1, 0, 0 );
```

```
  ::CurveProjection( *pCylSphere, *pContour_Base2, &dirProj2, false, false,
                     arrProjCurves_Base2 );

  // Construction of the second fragment of the prism surface as a ruled surface,
  // given by a pair of curves
  MbSurface* pSurfPrism2 = 0;
  ::RuledSurface( *pContour_Base2, *arrProjCurves_Base1[0], true, pSurfPrism2 );

  // 5) CONSTRUCTION OF HOLES ON THE BASIC SURFACE pCylSphere ACCORDING TO THE LINES OF
  //    CROSSING THE PRISM WITH SPHERE
  // Construction of the surface bounded by three contours
  MbSurface* pSurfWithHoles = 0;
  RPArray<MbCurve> arrBounds;
  {
    // Projection of space curves - the lines of intersection of a prism with a sphere
    // - on the base surface to obtain two-dimensional contours on the base surface
    MbContour* pProjContour1 = NULL;
    :SurfaceBoundContour( *pCylSphere, *arrProjCurves_Base1[0],
                          Math::DefaultMathVersion(), pProjContour1 );

    MbContour* pProjContour2 = NULL;
    ::SurfaceBoundContour( *pCylSphere, *arrProjCurves_Base2[0],
                          Math::DefaultMathVersion(), pProjContour2 );

    // Getting the contour - the outer edge of the base surface
    MbContour* pExtContour = &pCylSphere->MakeContour(true);

    // Construct a surface with pSurfWithHoles holes based on pCylSphere
    arrBounds.Add( pExtContour );
    arrBounds.Add( pProjContour1 );
    arrBounds.Add( pProjContour2 );
    ::BoundedSurface( *pCylSphere, arrBounds, pSurfWithHoles );
  }

  // 6) DISPLAY OF GEOMETRIC OBJECTS
  viewManager->AddObject( Style(1, LIGHTGRAY), pSurfWithHoles );
  viewManager->AddObject( Style(3, LIGHTRED),  pContour_Base1 );
  viewManager->AddObject( Style(3, LIGHTRED),  arrProjCurves_Base1[0] );
  viewManager->AddObject( Style(3, LIGHTRED),  pContour_Base2 );
  viewManager->AddObject( Style(3, LIGHTRED),  arrProjCurves_Base2[0] );
  viewManager->AddObject( Style(3, LIGHTGRAY), pSurfPrism1 );
  viewManager->AddObject( Style(3, LIGHTGRAY), pSurfPrism2 );

  // Reducing reference counts for dynamically created kernel objects
  ::DeleteItem( pCylSphere );
  ::DeleteItem( pContour_Base1 );
  for (int i=0; i<arrProjCurves_Base1.size(); i++)
    ::DeleteItem(arrProjCurves_Base1[i]);
  ::DeleteItem( pSurfPrism1 );
  ::DeleteItem( pContour_Base2 );
  for (int i=0; i<arrProjCurves_Base2.size(); i++)
    ::DeleteItem(arrProjCurves_Base2[i]);
  ::DeleteItem( pSurfPrism2 );
  for (int i=0; i<arrBounds.size(); i++)
    ::DeleteItem(arrBounds[i]);
  ::DeleteItem( pSurfWithHoles );
}
```

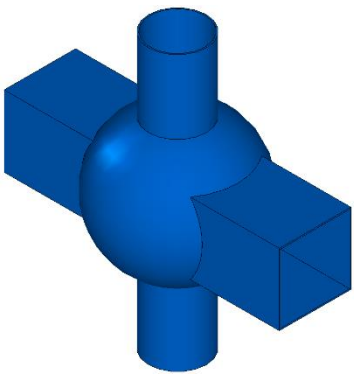
Combining surfaces in C3D Modeler requires explicitly ensuring that the required intermediate geometric operations are performed. By this is meant the explicit creation of fragments of surfaces that form the required geometric model, with an indication of the necessary boundary

curves. In Example 4.2, the geometric objects that make up the model shown in Fig. 5 (a) are not connected to each other in any data structure - they are stored independently of each other as a set of separate geometric objects.

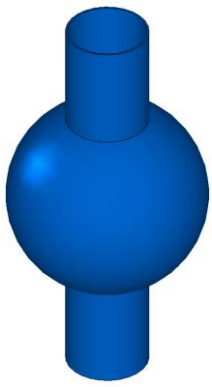
## *4.2 Construction of a thin-wall solid model*

In contrast to surface models, when constructing models of solids in C3D Modeler, a large number of intermediate constructions are performed automatically (in particular, the construction of edges at the faces of solids). Therefore, to construct geometrically consistent models, it is preferable to use the operations of constructing solid models. To illustrate this approach, we consider the construction of a model similar to that in Fig. 5(a), in the form of a thin-walled solid model. When constructing it, the following actions will be performed (they are explained in Fig. 6):

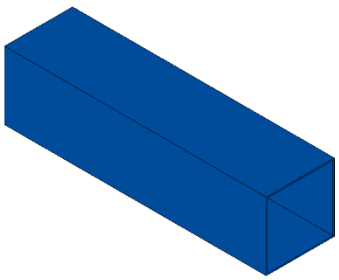
1) Construction of a thin-walled solid body of revolution whose surface is a combination of a spherical and cylindrical surface (Fig. 6 (b)). To build this body, the same generator is used, which was used to build the model in fig. 4.
2) Construction of a thin-walled extrusion solid whose surface coincides with the surface of a prism with a square base (Fig. 6 (c)).
3) Execution of the Boolean operation “body difference” for splitting the extrusion body into three parts (Fig. 6 (d)). Two parts located outside the body of revolution (Fig. 6 (e)) are selected for further constructions.
4) Execution of the Boolean operation “difference of bodies” for splitting the body of revolution (Fig. 6 (b)) into three parts (Fig. 6 (e)). There are three parts, since the body is a thin-walled rotation, and in the area of each square hole (one in Fig. 6 (e) is not visible) a separate thin-walled part is formed. The part shown in fig. 6 (g) is selected to construct the resultant body (Fig. 6 (a)). The parts corresponding to the square holes in the body of rotation are not involved in the further processing.
5) Double execution of the “union of bodies” Boolean operation for combining selected fragments of the extrusion body and the rotation body. As a result, the target thin-walled body is formed (Fig. 6 (a)).



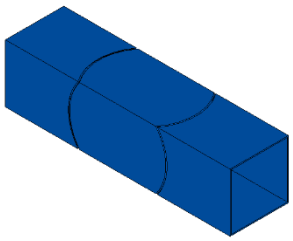
(a)



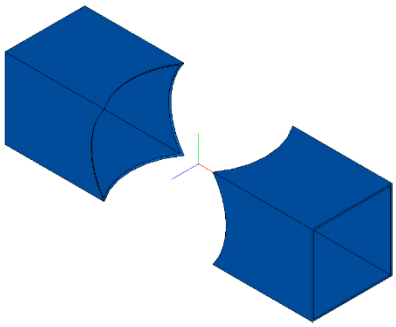
(b)



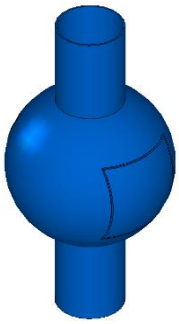
(c)



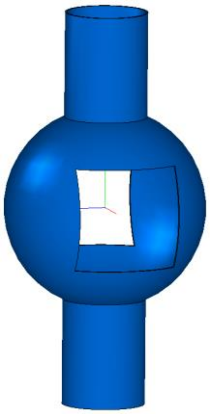
(d)



(e)



(f)



(g)

**Fig. 6.** The construction of a thin-walled solid-state model of three fragments: one fragment of the body of revolution and two fragments of the body of extrusion. (a) Built model. (b) Auxiliary solid body of rotation T1. (c) Auxiliary extrusion solid T2. (d) Difference of bodies T2 – T1. (e) Selected extrusion body fragments. (e) Difference of bodies T1 – T2. (g) Selected fragment of the body of revolution.

Software implementation of the construction of a solid-state model in fig. 6 is presented in Example 4.3. When comparing with the surface model from Example 4.2, it can be noted that in solid-state modeling, complex and numerically unstable conjugation operations of model fragments can be performed automatically using C3D Modeler functions that implement Boolean solid-state modeling operations. The entire sequence of actions to construct the body is stored in the auxiliary data structure of the kernel - in the build log.

***Example 4.3. Thin-walled solid model in the form of a combination of fragments of bodies of revolution and extrusion (Fig. 6).***

```
#include "cur_polyline3d.h"
#include "cur_arc3d.h"
#include "action_solid.h"

// Auxiliary function.
// Construction of a thin-walled solid body of revolution in the form of a combination
// of a cylinder and a sphere with centers of gravity at the origin.
MbSolid* CreateCylSphereSolid( double sphereRad, double cylRad, double cylHeight,
                               double thwall /* wall thickness */ )
{
  // Generating curve is a contour consisting of two segments and a circular arc
  // between them
  // Points - the vertices of the segments of the generator
  MbCartPoint3D p1( cylRad, cylHeight/2, 0 );
  MbCartPoint3D p2( cylRad, sqrt( sphereRad*sphereRad - cylRad*cylRad ), 0 );
```

```
  MbCartPoint3D p3( sphereRad, 0, 0 );
  MbCartPoint3D p4( p2.x, -p2.y, p2.z );
  MbCartPoint3D p5( p1.x, -p1.y, p1.z );
  // Generating curve segments
  MbLineSegment3D* pSeg1 = new MbLineSegment3D( p1, p2 );
  MbArc3D* pArc = new MbArc3D( p2, p3, p4, 1, false );
  MbLineSegment3D* pSeg2 = new MbLineSegment3D( p4, p5 );
  // Construction of a generating curve in the form of a contour from three segments
  RPArray<MbCurve3D> arrCurves;
  arrCurves.Add( pSeg1 );
  arrCurves.Add( pArc );
  arrCurves.Add( pSeg2 );
  MbContour3D* pGenContour = new MbContour3D( arrCurves, true );

  // Formation of the object sweptData
  RPArray< MbContour3D > arrContours;
  arrContours.Add( pGenContour );
  MbSweptData sweptData(arrContours);

  // Parameters of the axis of rotation - the axis is directed along the Y axis
  MbCartPoint3D axOrg( 0, 0, 0 );
  MbVector3D axDir( 0, 1, 0 );
  MbAxis3D axRot( axOrg, axDir );

  // Namer of faces of rotation solid
  MbSNameMaker operNames( ct_CurveRevolutionSolid, MbSNameMaker::i_SideNone, 0 );
  operNames.SetVersion( MbNameVersion() );
  PArray<MbSNameMaker> cNames( 0, 1, false );

  // Parameters of the rotation operation for constructing a thin-walled body
  RevolutionValues params;
  params.shape = 0;                          // Construction of solid “Sphere”
  params.side1.scalarValue = 360*M_PI/180;   // Angle of rotation of generator
  params.thickness1 = thwall;                // Wall thickness

  // The call of the operation of constructing a rotation body pSolid
  MbSolid* pSolid = NULL;
  MbResultType res = ::RevolutionSolid( sweptData, axRot, params,
                                        operNames, cNames, pSolid );
  if ( res != rt_Success )
  {
    // Return in case of error when building a solid
    ::DeleteItem( pGenContour );
    ::DeleteItem( pSolid );
    return NULL;
  }

  // Return constructed solid
  return pSolid;
}


// Auxiliary function.
// Construction of a thin-walled extrusion solid representing the lateral surface of a
// prism with a square base
MbSolid* CreatePrismSolid( double prismSide, double prismHeight, double thwall )
{
  // The coordinates of the vertices of the base of the prism
  // (in the ZY plane - in the middle section of the prism)
  std::vector<MbCartPoint3D> arrPnts_Base1 = {
      { 0,  prismSide/2, -prismSide/2 },
      { 0,  prismSide/2,  prismSide/2 },
```

```
        { 0, -prismSide/2,  prismSide/2 },
        { 0, -prismSide/2, -prismSide/2 }
    };

    // Contour is a closed polyline to represent the base of a prism
    MbPolyline3D* pContour_Base1 = new MbPolyline3D( arrPnts_Base1, true );

    // Construction of generator curve
    RPArray<MbCurve3D> arrCurves;
    arrCurves.Add( pContour_Base1 );
    MbContour3D* pGenContour = new MbContour3D( arrCurves, true );

    // Formation of the object sweptData
    RPArray< MbContour3D > arrContours;
    arrContours.Add( pGenContour );
    MbSweptData sweptData(arrContours);

    // Extrusion direction
    MbVector3D vecDir( 1, 0, 0 );

    // Namer of faces of rotation solid
    MbSNameMaker operNames( ct_CurveExtrusionSolid, MbSNameMaker::i_SideNone, 0 );
    operNames.SetVersion( MbNameVersion() );
    PArray<MbSNameMaker> cNames( 0, 1, false );

    // Extrusion Operation Parameters.
    // Extrusion is performed symmetrically on both sides of the generator.
    ExtrusionValues params( prismHeight/2, prismHeight/2 );
    params.thickness1 = thwall;          // Wall thickness

    // Calling an extrusion operation to construct a solid pSolid
    MbSolid* pSolid = NULL;
    MbResultType res = ::ExtrusionSolid( sweptData, vecDir, NULL, NULL, false,
                                         params, operNames, cNames, pSolid );
    if ( res != rt_Success )
    {
      // Return in case of error when building a solid
      ::DeleteItem( pGenContour );
      ::DeleteItem( pSolid );
      return NULL;
    }

    // Return built solid
    return pSolid;
}


// Main function.
// Building a thin-walled solid model in the form of a combination of bodies of
// rotation and extrusion
void MakeUserCommand0()
{
    // 1) CONSTRUCTION OF THE FIRST AUXILIARY THIN-WALL SOLID BODY OF ROTATION -
    // COMBINATION OF A CYLINDER AND SPHERE
    const double CYL_RAD    = 2.0;       // The radius of the cylindrical part
    const double CYL_HEIGHT = 20.0;      // The height of the cylindrical part
    const double SPHERE_RAD = 5.0;       // Spherical radius
    const double THICKNESS  = 0.1;       // Thin wall thickness
    MbSolid* pCylSphereSolid = CreateCylSphereSolid( SPHERE_RAD, CYL_RAD,
                                                     CYL_HEIGHT, THICKNESS );
    // Possible debugging call to verify step (1)
    // if ( pCylSphereSolid )
```

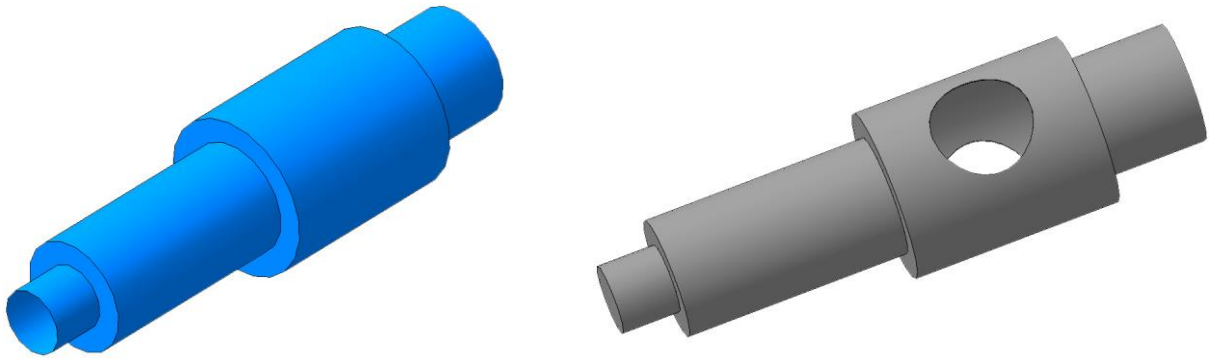
```
//   TestVariables::viewManager->AddObject( TestVariables::SOLID_Style,
//                                          pCylSphereSolid );


// 2) CONSTRUCTION OF THE SECOND AUXILIARY THIN-WALL SOLID BODY OF EXTRACT –
// SIDE SURFACE OF PRISM WITH SQUARE BASE
const double PRISM_SIDE = 5;         // Side of the square base of the prism
const double PRISM_HEIGHT = 20;      // Prism height
MbSolid* pPrismSolid = CreatePrismSolid( PRISM_SIDE, PRISM_HEIGHT, THICKNESS );
// Possible debugging call to verify step (2)
// if ( pPrismSolid )
//  TestVariables::viewManager->AddObject(TestVariables::SOLID_Style, pPrismSolid);


// 3) Dividing the prismatic body of pPrismSolid IN THREE PARTS:
//    TWO PARTS OUTSIDE AND ONE INSIDE pCylSphereSolid.
//    EXTERNAL PARTS ARE SELECTED FOR USE WHEN CONSTRUCTING A RESULT BODY
MbSolid* pPrismPart1 = NULL;
MbSolid* pPrismPart2 = NULL;

// Face name maker to build a body with a Boolean operation
MbSNameMaker operBoolNames( ct_BooleanSolid, MbSNameMaker::i_SideNone, 0 );
// Operation flags: building a closed body with the union of similar faces and edges
MbBooleanFlags flagsBool;
flagsBool.InitBoolean( true );
flagsBool.SetMergingFaces( true );
flagsBool.SetMergingEdges( true );

{
  // Boolean operation - body difference: pSolidDiff = pPrismSolid - pCylSphereSolid
  // To accomplish the difference, an auxiliary body pCylSphereSolidSmall is
  // generated, whose radius of the spherical part is smaller than that of
  // pCylSphereSolid by the wall thickness.
  // This is done to ensure the subsequent merging of body parts in the future, in
  // step (5).
  MbSolid* pCylSphereSolidSmall = CreateCylSphereSolid( SPHERE_RAD - THICKNESS,
                                        CYL_RAD, CYL_HEIGHT, THICKNESS );
  MbSolid* pSolidDiff = NULL;
  MbResultType res = ::BooleanResult( *pPrismSolid, cm_Copy, *pCylSphereSolidSmall,
                     cm_Copy, bo_Difference, flagsBool, operBoolNames, pSolidDiff );
  ::DeleteObject( pCylSphereSolidSmall );

  // Possible debugging call to verify step (3)
  // if ( pSolidDiff )
  //   TestVariables::viewManager->AddObject( TestVariables::SOLID_Style,
  //                                          pSolidDiff );

  // PSolidDiff composite body splitting into three bodies: two outside and one
  // inside pCylSphereSolid
  if ( res == rt_Success )
  {
    RPArray<MbSolid> parts;
    MbSNameMaker detachNames( ct_DetachSolid, MbSNameMaker::i_SideNone, 0 );

    // When pSolidDiff body is divided, the largest part remains in pSolidDiff, while
    // smaller parts (partsCnt pieces) are placed in the parts array (in descending
    // order of overall size)

    size_t partsCnt = ::DetachParts( *pSolidDiff, parts, true, detachNames );
    if ( partsCnt == 2 )
    {
      // Selection of the outer parts of the body pSolidDiff
```

```
      pPrismPart1 = parts[0];
      pPrismPart2 = parts[1];
      // Removing the excess part that is inside the extrusion body
      ::DeleteItem( pSolidDiff );
    }
  }
}


// 4) CUTTING SQUARE OPENINGS FROM pCylSphereSolid
MbSolid* pCylSphereHolesSolid = NULL;
{
  // Boolean operation - body difference: pSolidDiff = pCylSphereSolid - pPrismSolid
  MbSolid* pSolidDiff = NULL;
  MbResultType res = ::BooleanResult( *pCylSphereSolid, cm_Copy, *pPrismSolid,
                       cm_Copy, bo_Difference, flagsBool, operBoolNames, pSolidDiff );
  // Possible debugging call to verify step (4)
  // if ( pSolidDiff )
  //   TestVariables::viewManager->AddObject(TestVariables::SOLID_Style, pSolidDiff);

  // PSolidDiff composite body splitting into three bodies
  if ( res == rt_Success )
  {
    RPArray<MbSolid> parts;
    MbSNameMaker detachNames( ct_DetachSolid, MbSNameMaker::i_SideNone, 0 );
    size_t partsCnt = ::DetachParts( *pSolidDiff, parts, true, detachNames );

    if ( partsCnt == 2 )
    {
      // Saving the part that will be used for the resulting model
      pCylSphereHolesSolid = pSolidDiff;
      // Removing parts corresponding to square holes in the body of rotation
      ::DeleteItem( parts[0] );
      ::DeleteItem( parts[1] );
     }
  }
}


// 5) CONSTRUCTION OF A RESULTING BODY pResSolid: UNITING THREE BODIES
//    pCylSphereHolesSolid, pPrismPart1 И pPrismPart2
MbSolid* pResSolid = NULL;
{
  // Building an intermediate body: pSolid1 = pCylSphereHolesSolid + pPrismPart1
  MbSolid* pSolid1 = NULL;
  ::BooleanResult( *pCylSphereHolesSolid, cm_Copy, *pPrismPart1, cm_Copy, bo_Union,
                flagsBool, operBoolNames, pSolid1 );

  // Construction of the resulting body: pResSolid = pSolid1 + pPrismPart2
  ::BooleanResult( *pSolid1, cm_Copy, *pPrismPart2, cm_Copy, bo_Union,
                  flagsBool, operBoolNames, pResSolid );

  // Reducing the reference count of an intermediate body that is no longer required
  ::DeleteItem( pSolid1 );
}

// DISPLAYING THE RESULTING SOLID BODY
if ( pResSolid )
  TestVariables::viewManager->AddObject( TestVariables::SOLID_Style, pResSolid );


// REDUCTION OF COUNTER LINKS DYNAMICALLY CREATED NUCLEAR OBJECTS
```

```
  ::DeleteItem( pCylSphereSolid );
  ::DeleteItem( pPrismSolid );
  ::DeleteItem( pPrismPart1 );
  ::DeleteItem( pPrismPart2 );
  ::DeleteItem( pCylSphereHolesSolid );
  ::DeleteItem( pResSolid );
}
```

### *4.3 Tasks*

1) Construct the surface of a circular cone by rotating the generatrix.
2) Construct the surface of a single-cavity hyperboloid. Generating curve - hyperbola - can be represented as a NURBS-curve passing through a set of points.
3) Run Example 4.3. Call the build log for the formed solid (Fig. 6(h)) and find in it the parameters of the operations of constructing solids called in the function MakeUserCommand0() of this example.
4) Construct the surface of rotation in the form of a side surface of the shaft (Fig. 7(a)).
5) Construct a thin-walled solid, whose lateral surface is similar to the surface from the previous task (Fig. 7(a)).
6) Modify the thin-walled solid, constructed in task (5), by adding a through circular hole in one of its sections (Fig. 7(b)).



(a) (b)

**Fig. 7.** Samples of models for setting 4 (surface of rotation (a)) and for setting 6 (solid thin-walled model (b)).

## 5. Kinematic surfaces

The kinematic surface of a general form is specified by specifying two curves — the generator and the guide. A special case of kinematic surfaces is the surface of a plane-parallel motion - when they are constructed, the orientation of the generatrix does not change during the motion. Such surfaces are also called shift surfaces. For their representation in C3D Modeler, the MbExpansionSurface class is used. The shift surface differs from the extrusion surface MbExtrusionSurface (p. 3) in that it allows you to explicitly specify a guide of an arbitrary type (in the case of the MbExtrusionSurface, a straight line segment is used as a guide).

An example of a shift surface is shown in Fig. 8. A circular contour is used as a generator, and a plane spline curve is used as a guide. On this surface, the shape of the generator is constant. To build a shift surface, follow these steps:

1) Construction of generatrix in the form of an object of a class MbCurve3D.
2) Construction of a guide as an object of a class MbCurve3D.

3) Constructing a surface of a plane-parallel motion in the form of a MbSurface object or MbExpansionSurface (using the utility function from the action_surface.h file or using an explicit call to the surface class constructor).

Often the generator and guide turn out to be plane curves. But when building a surface of a plane-parallel motion, it is necessary to transfer them in the form of three-dimensional curves. In this case, the generator and the guide can be constructed in two stages: first, to form a two-dimensional curve, and then on its base - a three-dimensional curve of the same type indicating the location in space using the MbPlacement3D object (local three-dimensional coordinate system). This technique is used in the following examples 5.1-5.3.

To avoid self-intersections of the surface, the generating and guiding curves should not contain sections parallel to each other. The starting point of the guideline must be aligned with the corresponding point of the generator curve.

In the following example 5.1, to construct a surface of a plane-parallel motion resembling a curved tube of constant circular cross section, the ExpansionSurface utility function from the action_surface.h header file is used:

```
MbResultType ::ExpansionSurface(const MbCurve3D& curve, const MbCurve3D& spine,
                                MbCurve3D* curve1, MbSurface*& result);
```

When calling this function, it is necessary to specify the forming curve, the spine guide curve and the output parameter to get the resultant motion surface. Additionally, the second curve1 curve generator can be specified. In the case of surfaces of constant section, null value is passed as curve1 (as in Example 5.1).

***Example 5.1 Constructing a surface of shifting of a constant cross section (Fig. 8).***

```
#include <cur_bezier.h>
#include <cur_bezier3d.h>

void MakeUserCommand0()
{
  const double DEG_TO_RAD = M_PI/180.0;
  MbPlacement3D plArc;          // SC to construct a generator (global SC)
  MbPlacement3D plSpine;        // SC to construct a guide

  // Construction of a two-dimensional generator curve - a circle
  const double RAD = 10;
  const MbCartPoint arcCenter(0, 0);
  // Construction of a circle on the plane by the center and radius
  MbArc* pArc2D = new MbArc(arcCenter, RAD);
  // Construction of a circle in three-dimensional space
  MbArc3D* pArc = new MbArc3D(*pArc2D, plArc);

  // Construction of a two-dimensional guide - Bezier curve
  SArray<MbCartPoint> arrPnts(4);
  arrPnts.Add(MbCartPoint(0, 0));
  arrPnts.Add(MbCartPoint(40, 30));
  arrPnts.Add(MbCartPoint(70, -30));
  arrPnts.Add(MbCartPoint(100, 0));
  // Constructor by four control points is used
  MbBezier* pSpine = new MbBezier(arrPnts);

  // Rotate the local coordinate system around the Y axis of the world coordinate
  // system until the XY plane of the local SC is aligned with the XZ world SC plane
  plSpine.Rotate(MbAxis3D(MbVector3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(0, 1, 0))),
                 -90*DEG_TO_RAD);
```

```
  // Construction of a three-dimensional Bezier curve based on a two-dimensional one
  // using a constructor that takes as input a two-dimensional curve and a local
  // coordinate system of the three-dimensional curve under construction
  MbBezier3D* pBezier = new MbBezier3D(*pSpine, plSpine);

  // Calling the function of constructing the surface
  MbSurface* pSurface = NULL;
  ::ExpansionSurface(*pArc, *pBezier, NULL, pSurface);

  // Display of the resulting surface
  viewManager->AddObject(Style(1, RGB(0, 255, 0)), pSurface);

  ::DeleteItem(pSurface);
  ::DeleteItem(pBezier);
  ::DeleteItem(pSpine);
  ::DeleteItem(pArc);
  ::DeleteItem(pArc2D);
}
```

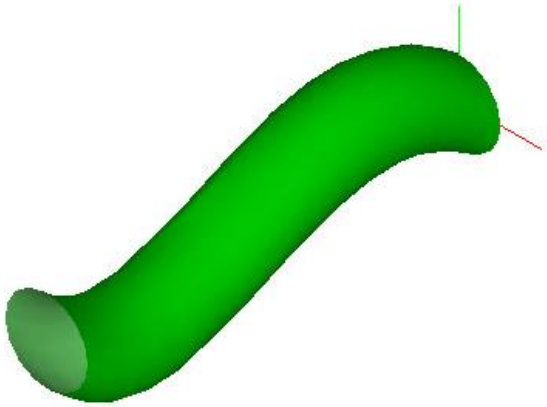


**Fig. 8.** A surface of a plane-parallel motion with a constant cross section (Example 5.1).

The ExpansionSurface function allows you to construct not only the surfaces of a plane-parallel motion of a constant section, but also the surfaces whose cross-sectional shape changes as you move along the guide curve. The change in cross section is achieved by explicitly specifying the shape of the generator curve at the start and end points of the guide curve. In this case, not one is created, as discussed above in Example 5.1, but two separate forming curves. In this case, both contours of the guide curves should be aligned with the end points of the guide - curve with the starting point of the guide, and curve1 - with the end point. This method of calling ExpansionSurface is shown in Example 5.2. In this example, both generators are circles. The radius of the second generator is 3 times the first. To align the second generator with the end point of the guide center, the local SC of this generator is transferred to the end point of the Bezier guide curve.

***Example 5.2 Construction of the surface of shifting of a variable cross section (Fig. 9).***

```
void MakeUserCommand0()
{
  const double DEG_TO_RAD = M_PI / 180.0;
  MbPlacement3D plArc1;   // SC to construct the first generator (global SC)
  MbPlacement3D plSpine;  // SK to construct a guide (calculated below)
  MbPlacement3D plArc2;   // SK to construct the second generatrix (calculated below)

  // Construction of a two-dimensional generator curve - a circle
  const double RAD = 10;
```

```
  const MbCartPoint arcCenter(0, 0);
  // Construction of a circle on the plane by the center and radius
  MbArc* pArc2D_1 = new MbArc(arcCenter, RAD);
  // Construction of a circle in three-dimensional space
  MbArc3D* pArc1 = new MbArc3D(*pArc2D_1, plArc1);

  // Construction of a two-dimensional guide - Bezier curve
  SArray<MbCartPoint> arrPnts(4);
  arrPnts.Add(MbCartPoint(0, 0));
  arrPnts.Add(MbCartPoint(40, 30));
  arrPnts.Add(MbCartPoint(70, -30));
  arrPnts.Add(MbCartPoint(100, 0));
  // Used constructor to construct by four control points.
  MbBezier* pSpine = new MbBezier(arrPnts);

  // Rotate the local coordinate system around the Y axis of the world coordinate
  // system until the XY plane of the local SC is aligned with the XZ world SC plane
  plSpine.Rotate(MbAxis3D(MbVector3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(0, 1, 0))),
                 -90 * DEG_TO_RAD);

  // Construction of a three-dimensional Bezier curve based on a two-dimensional
  // curve using a constructor that takes as input a two-dimensional curve and a local
  // coordinate system of the three-dimensional curve under construction
  MbBezier3D* pBezier = new MbBezier3D(*pSpine, plSpine);

  // Transfer of the local coordinate system of the second circle to the end point of
  // the guide curve
  plArc2.Move(MbVector3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(0, 0, 100)));

  // Construction of the second generator - a circle at the end of the
  // resulting surface
  MbArc* pArc2D_2 = new MbArc(arcCenter, RAD*3);
  MbArc3D* pArc2 = new MbArc3D(*pArc2D_2, plArc2);

  // Calling the function of constructing the surface
  MbSurface* pSurface = NULL;
  ::ExpansionSurface(*pArc1, *pBezier, pArc2, pSurface);

  // Display of the resulting surface
  viewManager->AddObject(Style(1, RGB(0, 0, 255)), pSurface);

  ::DeleteItem(pSurface);
  ::DeleteItem(pBezier);
  ::DeleteItem(pSpine);
  ::DeleteItem(pArc2D_1);
  ::DeleteItem(pArc1);
  ::DeleteItem(pArc2D_2);
  ::DeleteItem(pArc2);
}
```

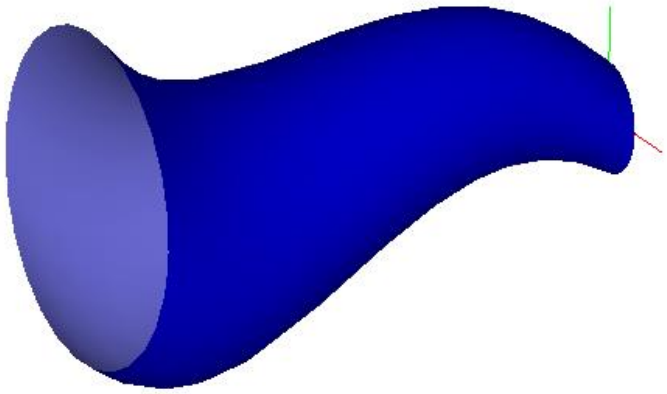


**Fig. 9.** Surface of shifting of a variable cross section (example 5.2).

The EvolutionSurface function is used to construct general kinematic surfaces.:

```
MbResultType EvolutionSurface( const MbCurve3D& curve, const MbCurve3D& spine,
                               MbSurface*& result);
```

This function forms the surface as an object of the MbEvolutionSurface class (or MbSpiralSurface in the particular case if a conical spiral curve is passed as a guide). The surface is constructed by moving the curve curve along the spine guide while maintaining orientation relative to it. The shape of the generatrix during the movement does not change. The call to this function is shown in Example 5.3, in which a kinematic surface is constructed using a circle as a generator curve and a circular arc as a guide curve.

***Example 5.3 Construction of the kinematic surface (Fig. 10).***

```
void MakeUserCommand0()
{
  const double DEG_TO_RAD = M_PI/180.0;
  MbPlacement3D plArc;    // SC to construct the first generator (global SC)
  MbPlacement3D plCurve;  // SK to construct a guide (calculated below)

  // Rotate the local SC of the guide curve from the XY plane to the XZ plane of the
  // global coordinate system
  plCurve.Rotate(MbAxis3D(MbVector3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(1, 0, 0))),
                 90*DEG_TO_RAD);

  // Construction of a two-dimensional generator curve - a circle
  const double RAD = 10;
  const MbCartPoint arcCenter(0, 0);
  // Construction of a circle on the plane by the center and radius
  MbArc* pArc2D = new MbArc(arcCenter, RAD);
  // Construction of a circle in three-dimensional space
  MbArc3D* pArc = new MbArc3D(*pArc2D, plArc);

  // Construction of the guide curve - arc of a circle
  // First, an arc of a two-dimensional circle is constructed along the center of
  // the circle, radius, initial and final points
  MbArc*  pCurve2D = new MbArc(MbCartPoint(-50,0), 50, MbCartPoint(0, 0),
                               MbCartPoint(-50, 50), 1);
  // Construction of a three-dimensional circle arc
  MbArc3D* pCurve = new MbArc3D(*pCurve2D, plCurve);
```

```
  // Call of function of construction of a kinematic surface
  MbSurface* pSurface = NULL;
  ::EvolutionSurface(*pArc, *pCurve, pSurface);

  // Display constructed surface
  viewManager->AddObject(Style(1, RGB(0, 0, 255)), pSurface);

  ::DeleteItem(pSurface);
  ::DeleteItem(pCurve2D);
  ::DeleteItem(pCurve);
  ::DeleteItem(pArc2D);
  ::DeleteItem(pArc);
}
```

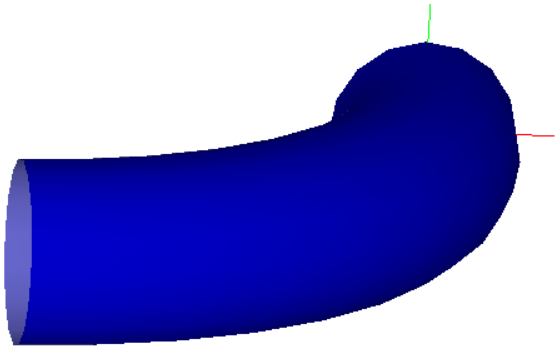
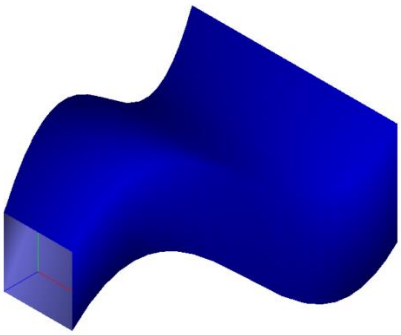


**Fig. 10.** Kinematic surface of circular section with a guide - a circular arc (Example 5.3).

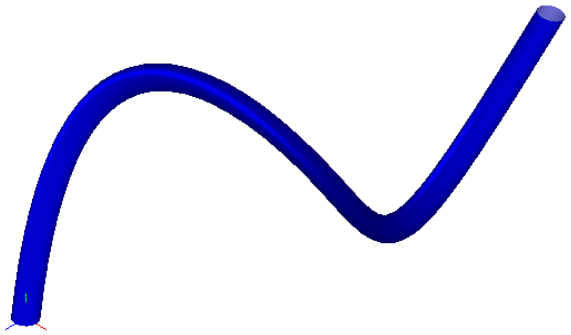
## *5.1 Tasks*

1) Construct the surface in Example 5.3 using the ExpansionSurface function, not EvolutionSurface. Compare the resulting surface with the kinematic surface in fig. 10.
2) Construct a surface of shifting with a variable section using the ExpansionSurface function. The section at the starting point of the guide curve is a square with a side of 20, in the final one there is a rectangle with sides 60 and 30. The forming contour-square lies in the XY plane of the global CS, and its center coincides with the origin. The guide in its local SC passes through the points (0,0), (60.90), (100, -80), (150.0), and the global SC is directed opposite to the Z axis. The resulting surface should look like this:



3) Using the EvolutionSurface function, construct the kinematic surface shown in the following figure. The guide is defined as a fourth-order MbNurbs3D three-dimensional curve. This curve, which does not lie in the same plane, passes through the points (0, 0, 0), (10, 155, -80), (300, 250, 0), (450, 350, 80), (500, 500, 0). Use a circle with a radius of 10 as a cross section. To construct

a three-dimensional NURBS curve, use the function ::SplineCurve (the same function was used to construct a NURBS curve on a plane in work No. 3, item 3.5, problem 2).



## 6. Conclusion

In this lesson, we considered methods for constructing surfaces of motion of the most common types: extrusion, rotation, and kinematic surfaces. To represent these surfaces in C3D Modeler, the MbExtrusionSurface, MbRevolutionSurface, MbExpansionSurface, and MbEvolutionSurface classes are used. To construct the surfaces of each of the listed types in C3D Modeler there are utility functions described in the header class action_surface.h. The use of these functions is preferable to direct calling constructors, since they provide for the handling of construction errors.

Often surface models require the combination of fragments of several different types of surfaces. The lesson presents examples of the construction of such models. It can be noted that in this case it is necessary to pay special attention to ensure the joining of fragments of different surfaces. These operations can be quite time consuming. Therefore, for constructing geometric models of complex shape, solid models are usually more convenient. When they are constructed in C3D Modeler, the conjugation of the surface fragments (the docking of the faces of solid bodies along the edges) is performed automatically. Methods for constructing solid models will be considered in the following lessons.