Getting started with C3D Modeler

LESSON 6
Basic operations for constructing solids

https://c3dlabs.com
Moscow
2019

Table of Contents

1. Introduction

The main purpose of the C3D Modeler geometric kernel is to build mathematical models of solids and perform operations with these models. In previous lessons, methods for constructing curves and surfaces in two-dimensional and three-dimensional space were considered. Curves and surfaces in C3D Modeler are represented using parametric equations. Such equations are convenient for calculating points belonging to geometric objects, as well as characteristics of curves and surfaces at these points (for example, normal vectors). Unlike the curves and surfaces considered in previous lessons, the majority of solids cannot be represented using a single mathematical equation. Therefore, in C3D Modeler, solids, compared to curves and surfaces, are represented as more complex objects — these objects are composite.

In C3D Modeler, an approach based on the boundary representation (B-Rep) of solids is used for building solids, the most common in the field of CAD and computer graphics systems. In the framework of this approach, the shape of a simulated three-dimensional object is represented as a set of fragments of surfaces that are joined to each other along straight and curvilinear fragments of curves. Such fragments of surfaces are called faces, and fragments of curves are called edges. The edges joined to each other together represent a composite boundary surface. It separates the internal volume of the object being modeled from the rest of the space. C3D Modeler contains a set of operations that allow you to build the boundary surface of the object being modeled without explicitly specifying the surface fragments and the shape of the edges (the corresponding calculations can be quite cumbersome, for example, see lesson 5, paragraph 4.2).

This lesson discusses the basic operations of solid modeling: the construction of elementary solids (these operations were already mentioned in Chapter 1), the construction of extrusion and revolution bodies. The names of these operations resemble the corresponding surface construction operations (see Lesson 5), but the execution result turns out to be different - in this case, C3D Modeler forms a model of a solid body as an object of class MbSolid. This class is essential for representing solid models. The MbSolid class inherits from the MbItem class “Object of a Geometric Model” and contains a construction log — a list of core operations that lead to the construction of a solid body. This is an important difference from the objects of curves and surfaces. In classes of curves and surfaces, C3D Modeler stores the parameters of mathematical equations, and in the build log, a list of core operations and their parameters. To obtain a model of a solid body, the kernel performs operations according to this journal. The list and types of parameters for different operations may vary. In particular, both parameters and surfaces can be used as parameters. More complex operations of constructing solids, as well as actions with them, will be considered in the following lessons.

2. MbSolid – a solid body

As applied to solid modeling operations in C3D Modeler (and in the area of B-Rep boundary modeling in general), a number of specific terms are used. The following terms are listed with the corresponding C3D classes. The reader can find detailed consideration and explanatory illustrations in the book: Nikolay Golovanov. Geometric modeling *https://c3dlabs.com/en/info/book/* .

The face (MbFace class) is a fragment of a certain surface to which the normal direction is attributed. Specifying the direction of the normal is very important, since this direction is used to distinguish the two sides of the face — the outer and the inner. The outer side of the face is the one that the observer is facing towards the face normal. The reverse side of the face is called the inner.

An edge (MbEdge class) is a fragment of some curve with which a direction is assigned. Often there is a case when the edges are straight (segments). But, in general, the edges can be fragments of any curves without self-intersections. An edge has a direction sign, which takes a positive value if the edge is considered to be directed along the curve, or a negative direction if the edge is directed against the curve.

Fragments of surfaces representing faces are bounded by boundary curves of faces. Along these boundary curves, the faces join each other, and it is these curves that are used to construct the edges. Faces can be closed (for example, as the lateral surface of a cylindrical body). In this case, the face has a boundary curve, along which the face meets with itself. Such a fragment of a boundary curve and an edge constructed on it, along which a face meets with itself, is called ***a seam***.

A vertex (MbVertex class) is a point at which two or more edges join.

A face loop (MbLoop class) is a sequence of edges describing a certain boundary of a face. The face loop is always closed and directed so that the face is always to the left of the observer as it moves along the loop from the outside of the face. A face may have several loops. For example, a rectangular face with one circular opening will have two loops — one corresponds to the outer boundary, and the other to the boundary of the internal opening. When constructing and processing models of bodies using the functions of C3D Modeler, the direction of loops and edges is calculated automatically.

A Shell (MbFaceShell class) is a surface in the form of a set of faces joined to each other along common edges.

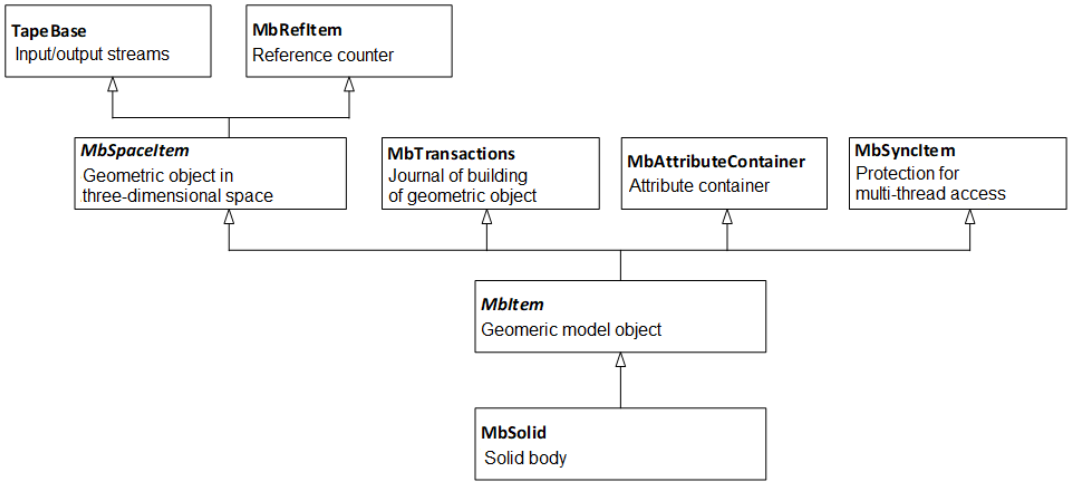

Fig. 1. Fragment of the class diagram showing inheritance relations for the “Solid body” class MbSolid.

For the representation of solids in C3D, the MbSolid class (header file solid.h) is intended. The inheritance diagram for this class is shown in Fig. 1. Operations for constructing solids are similar to the methods for creating surfaces: creating elementary bodies at given vertices, creating extrusion bodies, rotating, sweeping (kinematic operation), constructing over flat sections. On the class diagram in fig. 1 note that the MbSolid class inherits from the abstract class MbItem. MbItem is the base class for objects in the C3D geometry model. These objects represent concepts expanded in comparison with mathematical concepts, due to the storage of parameters (properties) necessary to perform computational geometric operations, as well as displaying and computational operations. These capabilities are provided both by the implementation of the MbItem class and by inheriting from the service classes C3D - MbTransactions, MbAttributeContainer and MbSyncItem. Especially important is the inheritance from the MbTransactions class. The geometry build log in C3D is an internal representation of the list of construction operations. This journal resembles a construction tree, which is displayed explicitly in most CAD systems when a user works with 3D solid models. In C3D, in the build log, operations are stored as class objects inherited from MbCreator. These classes, representing construction operations, are called “creators of geometric model objects”, or, briefly, “creators”.

Once again, we note that in C3D Modeler, geometric objects (for example, curves and surfaces considered in previous lessons) and objects of a geometric model are different concepts. Geometric models do not have a build log. In fig. 1 note that the MbItem class is inherited from the class of a geometric object in the three-dimensional space of MbSpaceItem. This ensures that the solid operations are performed in the core for three-dimensional geometric objects.

In object-oriented programming, domain concepts are represented as classes. In addition to the inheritance relationship, other relationships can be defined between classes (and, accordingly, between the relevant domain concepts). In addition to inheritance (and, perhaps, more often than inheritance), an aggregation relation is used between classes (also called composition in another way). Inheritance allows you to express in the programming language the relationship between a general concept and its particular cases, and aggregation - the relationship between the whole (composite) concept and its constituent parts. The different types of relationships between classes can be displayed both on one diagram and on several, depending on the convenience of the presentation. To simplify the class diagrams, the relations of inheritance and aggregation for the MbSolid class in this section are shown in separate diagrams in fig. 1 and fig. 2.

In fig. 2 connecting lines with a rhomb designate that an object of one class is part of another. A rhomb is marked with a compound object. If an object of a certain class is included in another in a single copy, then no marks are made at the other end of the connecting line. If the number of the same type parts can be arbitrary, the mark “n” is made on the diagram, if the number of parts is known in advance, it is indicated explicitly, for example, “2”. In the diagram in fig. 2, near the connecting lines are the names of the attributes of the classes in which the corresponding objects are represented in C3D. For example, you can see that the MbSolid class contains one MbFaceShell in the form of an object called outer. The shell contains the set of MbFace faces as an array of faceSet (the fact that faceSet is an array is not clearly indicated on the diagram). Arrays of objects that are parts of composite objects usually contain arrays of the RPArray class in C3D. Details on how to store parts of composite objects, as well as the internal names of the corresponding attributes, relate to the kernel implementation and are not explicitly used to work with it. In the diagram in fig. 2 (as well as in the C3D documentation), this information is provided to help the user navigate the kernel programming interface (for example, to determine which chain of calls can be used to access certain elements of MbSolid or some other composite object).

In fig. 2, it can be noted that the MbEdge “Edge” class is missing in the aggregation diagram. This class is abstract. It inherits the MbCurveEdge “Face edge” class. From it, in turn, the class MbOrientedEdge “Oriented Edge” is inherited to add the orientation attribute. MbLoop face loops (sets of MbOrientedEdge edges) are stored inside the MbFace “Face” class along with the face surface as an object of MbSurface class.

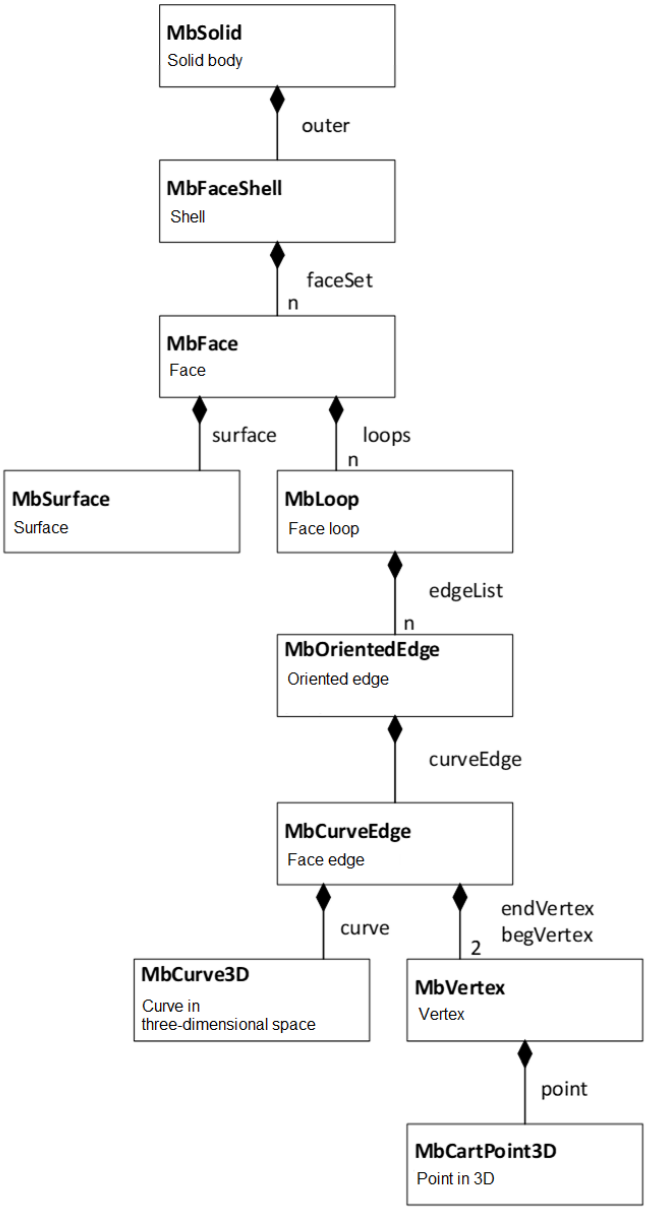

Fig. 2. Class diagram showing aggregation relationships for the MbSolid “Solid body” class.

Below, in Example 2.1, is a fragment of the interface of the class MbSolid. A significant part of the capabilities of the MbSolid class is provided by inheriting from MbItem, MbSpaceItem and MbTransactions (Fig. 1). Also in the interface of the MbSolid class there is a new section that implements functions specific to solids. For example, in this section, methods are provided for getting the components of a geometric model of a solid body — faces and edges (they are shown in Fig. 2).

Example 2.1. Fragment of the interface of the class MbSolid "Solid body" (Fig. 1, 2).

```
class MATH_CLASS MbSolid : public MbItem {
  // ...

public :
  // CONSTRUCTORS
  explicit MbSolid( MbFaceShell* shell, MbCreator* creator );
```

```
  explicit MbSolid( MbFaceShell& shell, MbCreator& creator );
  MbSolid( MbFaceShell& shell, RPArray<MbCreator>& creators, bool sameCreators,
           MbRegDuplicate* iReg );
  MbSolid( MbFaceShell& shell, std::vector< SPtr<MbCreator> >& creators,
           bool sameCreators, MbRegDuplicate* iReg );
  MbSolid( MbFaceShell& shell, const MbSolid& solid, MbCreator& creator );
  MbSolid( MbFaceShel & shell, const MbSolid& solid, MbCreator* creator );
  virtual ~MbSolid();

public :
  // GENERAL FUNCTIONS OF THE GEOMETRIC OBJECT IN THREE-DIMENSIONAL SPACE
  // (These methods are inherited from MbSpaceItem via MbItem.)
  virtual MbeSpaceType  IsA()  const; // Getting the type of a geometric object
  // Create a copy of a geometric object
  virtual MbSpaceItem & Duplicate( MbRegDuplicate* iReg = NULL ) const;
  // Perform a geometric transformation specified by a matrix
  virtual void Transform( const MbMatrix3D&, MbRegTransform* iReg = NULL );
  // Shift by a given vector
  virtual void Move( const MbVector3D&, MbRegTransform* iReg = NULL );
  // Revolution around a given axis at a given angle
  virtual void Rotate( const MbAxis3D&, double angle, MbRegTransform* iReg = NULL );
  // Checking objects for equality
  virtual bool IsSame( const MbSpaceItem& init ) const;
  // Checking the similarity of objects (can one equate another?)
  virtual bool IsSimilar( const MbSpaceItem& init ) const;
  // Make objects equal
  virtual bool SetEqual ( const MbSpaceItem & init );
  // Calculate distance to point
  virtual double DistanceToPoint( const MbCartPoint3D & ) const;
  // Add the size of the solid to the dimensional cube
  virtual void AddYourGabaritTo( MbCube& ) const;
  // Build a polygonal copy of a solid
  virtual void CalculateWire( double sag, MbMesh& mesh ) const;
  // Creating a new property
  virtual MbProperty& CreateProperty( MbePrompt n ) const;
  // Obtaining solid properties
  virtual void GetProperties( MbProperties& properties );
  // Setting solid property values
  virtual void SetProperties( const MbProperties& properties );
  // Get base objects
  virtual void GetBasisItems( RPArray<MbSpaceItem>& s );
  // Get control points
  virtual void GetBasisPoints( MbControlData3D& ) const;
  // Change object by control points
  virtual void SetBasisPoints( const MbControlData3D& );
  // Rebuild solid by build log
  virtual bool RebuildItem( MbeCopyMode copyMode, RPArray<MbSpaceItem>* items,
                            IProgressIndicator* progInd );
  // Finish the body for the last outstanding operation in the build log
  virtual bool FinishItem();
  // Build a polygonal object - a simplified version of the solid
  virtual MbItem* CreateMesh( const MbStepData& stepData,
                              const MbFormNote& note, MbRegDuplicate* iReg ) const;

  // FUNCTIONS OF SOLID BODY
  // Calculate dimensional cube of a solid
  virtual bool CalculateGabarit( MbCube& cube ) const;
  // Add solid state builders to the sent array of creators
  virtual bool GetCreators( RPArray<MbCreator>& creators ) const;
  // Replace the body shell by the sent one
  void         SetShell( MbFaceShell* shell );
  // Detach the shell
```

```
  MbFaceShell* DetachShell();
  // Get the shell
  MbFaceShell* GetShell() const;
  // Check if the shell was built
  bool         IsShellBuild() const;
  // Reinstall the pointers of the edges on the faces they connect
  void         MakeRight();
  // Check whether the pointers to the faces they connect are correctly positioned in
  // the edges
  bool         IsRight() const;
  // Get the number of faces
  size_t       GetFacesCount() const;

  // Get solid vertices
  void    GetVertices( RPArray<MbVertex>& ) const;
  // Get oriented edges of the solid
  void    GetEdges( RPArray<MbCurveEdge>& ) const;
  // Get vertices and edges of the solid
  void    GetItems( RPArray<MbVertex>&, RPArray<MbCurveEdge>& ) const;
  // Get faces of the solid
  void    GetFaces( RPArray<MbFace>& ) const;
  // Get the parts of the body - vertices, edges and faces
  void    GetItems( RPArray<MbTopologyItem>& ) const;
  // Get a vertex by its number
  MbVertex* GetVertex( size_t index ) const;
  // Get an edge by its number
  MbCurveEdge* GetEdge( size_t index ) const;
  // Get a face by its number
  MbFace* GetFace( size_t index ) const;
  // Get the number of the vertex
  size_t  GetVertexIndex( const MbVertex& ) const;
  // Get the number of the edge
  size_t  GetEdgeIndex( const MbCurveEdge& ) const;
  // Get the number of the face
  size_t  GetFaceIndex( const MbFace& ) const;
  // Get the number of connected shells of the body
  size_t  GetShellCount() const;

  // ...
};
```

The MbSolid class interface has a set of constructors that allow you to perform explicit construction of solids based on shells, other solids and builders. As in the case of surfaces, for the construction of solids it is usually more convenient to use not constructors, but utility functions that perform all the necessary actions, as well as verification of construction errors. In more detail these functions will be considered further. One of these is used in Example 2.2, which demonstrates access to the constituent parts of a solid. The result of executing example 2.2 is shown in fig. 3. The faces of the four-sided pyramid with a square base are assigned different colors. To change the color attribute of each face.

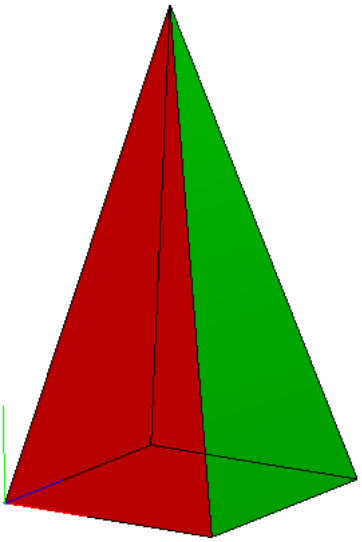

Fig. 3. A solid model of a pyramid with faces that are assigned various color attributes (Example 2.2).

Example 2.2. Building a solid and changing the color attributes of the faces (fig. 3).

```
#include "action_solid.h"
#include <vector>

using namespace std;
using namespace c3d;
using namespace TestVariables;


//
// Auxiliary function for the construction of an elementary body - a symmetric pyramid
// with a square base.
// The base of the pyramid lies in the XZ plane of the global SC.
//
// Input parameters:
//   baseSize - base side length,
//   height - pyramid height.
// Output parameters:
//   Pointer to a dynamically created geometric model object - solid
MbSolid* CreatePyramid( const double baseSize, const double height )
{
  // Default maker of names of faces
  MbSNameMaker names( 1 , MbSNameMaker::i_SideNone, 0);

  // Pointer to a solid under construction
  MbSolid* pSolid = NULL;

  // An array of points to build a pyramid - 4 points of the base and vertex
  vector<MbCartPoint3D> vecVerts = { { 0, 0, 0 }, { baseSize, 0, 0 },
                                     { baseSize, 0, baseSize}, { 0, 0, baseSize },
                                     { baseSize/2, height, baseSize/2 } };
  SArray<MbCartPoint3D> arrVerts( vecVerts );

  // The call of the utility function for the construction of the elementary body –
  // the pyramid
  ::ElementarySolid(arrVerts, et_Pyramid, names, pSolid);
```

```
  return pSolid;
}


void MakeUserCommand0()
{
  // Building a pyramid with an auxiliary function
  const double PYR_BASE   =  50.0;          // Base side length
  const double PYR_HEIGHT = 100.0;          // Pyramid height
  MbSolid* pPyr = CreatePyramid( PYR_BASE, PYR_HEIGHT );

  // Changing the color attributes of faces
  if ( pPyr )
  {
    // An array of colors to assign to the faces of the pyramid
    vector<COLORREF> arrColors = { RGB(0xFF, 0, 0), RGB(0, 0xFF, 0), RGB(0, 0, 0xFF),
                                   RGB(0xFF, 0xFF, 0), RGB(0xFF, 0, 0xFF) };
    // Enumerate all faces of a solid
    const size_t faceCnt = pPyr->GetFacesCount();
    for (size_t faceIdx = 0; faceIdx < faceCnt; faceIdx++)
    {
      // Getting a pointer to the next face with the index faceIdx
      MbFace* pFace = pPyr->GetFace(faceIdx);

      // Assigning the color attribute of the face
      if ( pFace )
        if ( faceIdx < arrColors.size() )
          pFace->SetColor( arrColors[faceIdx] );
    }
  }

  // Display the constructed body
  if ( pPyr )
    viewManager->AddObject(TestVariables::SOLID_Style, pPyr);

  // Reducing the reference count of a dynamically created kernel object
  ::DeleteItem(pPyr);
}
```

Example 2.3 demonstrates how to perform a computational operation using the MbShell shell class. In fig. 4 shows a pyramid constructed similarly to Example 2.2. There are 4 points around the pyramid - they are obtained by moving the midpoints of the sides of the base vertically up half the height of the pyramid. These reference points are projected on the faces of a pyramid. The resulting projections in Fig. 4 are also displayed as points. The original points and their projections are connected by segments. To calculate the projection of a three-dimensional point on the face of a solid, the MbFaceShell::DistanceToBound method of the shell class is used. This method returns an object of class MbPntLoc, which stores data on the calculated projection of the original point to the nearest shell face, as well as a numerical code of the MbeItemLocation type, characterizing the location of the original point relative to the shell (possible options are outside, on the shell or inside).

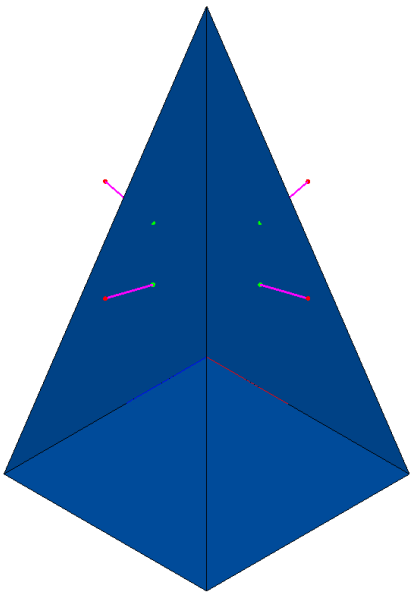

Fig. 4. An example of performing a computational operation using the "Shell of a solid body" class: projecting three-dimensional points on the faces of a pyramid (Example 2.3).

Example 2.3. Access to the elements of a solid body and the use of methods of the class "Shell" MbFaceShell (Fig. 4).

```
void MakeUserCommand0()
{
  // Build a pyramid using the auxiliary function CreatePyramid used in Example 2.2.
  const double PYR_BASE   =  50.0;           // Base side length
  const double PYR_HEIGHT = 100.0;           // Pyramid height
  MbSolid* pPyr = CreatePyramid( PYR_BASE, PYR_HEIGHT );
  if ( !pPyr )
    return;

  // Display solid body – a pyramid
  viewManager->AddObject(TestVariables::SOLID_Style, pPyr);

  // Access to the shell of a solid
  MbFaceShell* pShell = pPyr->GetShell();
  if ( !pShell )
    return;

  // The coordinates of the four original points outside the pyramid (chosen
  // arbitrarily, in this example, these points are built as the midpoints of the sides
  // of the base, raised half the height of the pyramid.)
  vector<MbCartPoint3D> vecPntsSrc = {
            { PYR_BASE/2, PYR_HEIGHT/2, 0 }, { PYR_BASE, PYR_HEIGHT/2, PYR_BASE/2 },
            { 0, PYR_HEIGHT/2, PYR_BASE/2 }, { PYR_BASE/2, PYR_HEIGHT/2, PYR_BASE } };

  // Calculation of three-dimensional points –
  // projections of the original points on the faces of the pyramid.
  vector<MbCartPoint3D> vecPntsProj;
  const size_t pntSrcCnt = vecPntsSrc.size();

  for ( size_t i = 0; i < pntSrcCnt; i++ )
  {
    // Calling the shell class method to project a three-dimensional point onto
    // a solid shell.
    // The MbFaceShell::DistanceToBound method returns data on the projection of
```

```
    // a point onto the nearest face of the shell and information about the location
    // of the point relative to the shell.
    MbPntLoc finFaceData;
    MbeItemLocation rShell;

    bool bProjOK = pShell->DistanceToBound( vecPntsSrc[i], EPSILON,
                                            finFaceData, rShell );

    // If the projection of the point vecPntsSrc[i] is successfully calculated, the
    // three-dimensional coordinates of this projection in the global SC are placed
    // into the array vecPntsProj.
    if ( bProjOK )
      vecPntsProj.push_back( finFaceData.GetPoint() );
  }

  // Display of source points, their projections and projecting segments.
  if ( vecPntsSrc.size() == vecPntsProj.size() )
  {
    // The original points and their projections will be displayed as
    // an object "Point frame"
    MbPointFrame* pPntFrame = new MbPointFrame;

    for (int i = 0; i<pntSrcCnt; i++ )
    {
      // Original points are displayed in red.
      // Since the original points and their projections are added to the point frame
      // in turn, the next pair of points will be stored in the frame
      // with indices (2*i, 2*i+1).
      pPntFrame->AddVertex( vecPntsSrc[i] );
      pPntFrame->GetVertex( 2*i )->SetColor( RGB(0xFF, 0, 0 ) );
      pPntFrame->GetVertex( 2*i )->SetWidth( 10 );

      // The projections of the original points are displayed in green.
      pPntFrame->AddVertex( vecPntsProj[i] );
      pPntFrame->GetVertex( 2*i+1 )->SetColor( RGB(0, 0xFF, 0 ) );
      pPntFrame->GetVertex( 2*i+1 )->SetWidth( 10 );

      // Display of the projecting segment for the i-th pair of points.
      // Segments are displayed in purple, with a line thickness of 3.
      MbLineSegment3D* pSeg = new MbLineSegment3D( vecPntsSrc[i], vecPntsProj[i] );
      Style segStyle( 3, RGB(0xFF, 0, 0xFF ) );
      viewManager->AddObject( segStyle, pSeg );
    }

    // Display point frame
    viewManager->AddObject(TestVariables::POINT_Style, pPntFrame);
  }

  // Reducing the reference count of a dynamically created kernel object
  ::DeleteItem(pPyr);
}
```

2.1 Tasks

1) Run Example 2.3. Try changing the size parameters of the pyramid and make sure that the model is built correctly. Increase the number of original points to project onto the pyramid's shell - in the array of original points, put the coordinates of 20 points evenly distributed around the pyramid at half of its height and another point above its top.
2) Delete three faces from the pyramid of Example 2.2 and display the resulting solid body. Notice how the outer and inner sides of the faces are displayed in the test application. Call the

properties window in it and make sure that the shell of the pyramid contains only two faces. To build, get a solid shell and delete faces using the MbFaceShell::DeleteFace(size_t index) method, where index is the index of the deleted face. After deleting a face with a certain index, this index will correspond to the face following the deleted one. Therefore, in this task, you can make a three-fold removal of a face with zero index. In order to avoid ambiguity when calling several variants of the MbFaceShell::DeleteFace method, explicitly select the called method by specifying the type of its parameter:

```
pShell->DeleteFace( (size_t)0 );
```

3) In the test application, display the edges of the pyramid constructed using the CreatePyramid function from Example 2.2. Build all the edges of this solid body in the form of curvilinear segments of thickness 5. You can use the MbSolid::GetEdge method to obtain the edges. This method for valid edges returns pointers to objects of class. Go through all the edges of the body, increasing the index of the edge until this method returns a null pointer. To get the three-dimensional curve on which the edge is constructed, use the MbCurveEdge::GetCurve method. Calls to display the next pEdge edge (assuming this is the pointer MbCurveEdge* pEdge) may look like this:

```
MbCurve3D* pCurve = (MbCurve3D*)(&pEdge->GetCurve());
viewManager->AddObject( Style( 5, LIGHTRED ), pCurve );
```

3. Elementary solids

When building solid models, the MbSolid class is usually used in conjunction with additional tools - builder classes, utility functions, and functions of geometric operations. In the fragment of the MbSolid interface, shown in Example 2.1, you can see that the constructors of the MbSolid class have parameters of the MbCreator class. This is an abstract base builder class, from which classes are inherited to implement various operations for constructing geometric model objects. For building shells of elementary solids in C3D, the MbElementarySolid building class is intended (Fig. 5).

The MbElementarySolid builder constructs the order of 10 different body shells, which in C3D are elementary. The type of shell to build is specified in the MbElementarySolid constructor in the form of an elementary solid type code. The number of reference points and their geometric meaning depend on the type of body to be built. Valid values for the types of elementary solids and the number of reference points are given in Table 1. The MbElementarySolid constructor has the following prototype:

```
MbElementarySolid:: MbElementarySolid(
            const Points& points,               // Basic points
            ElementaryShellType solidType,      // Elementary solid type
            const MbSNameMaker& names );        // Operation name maker
```

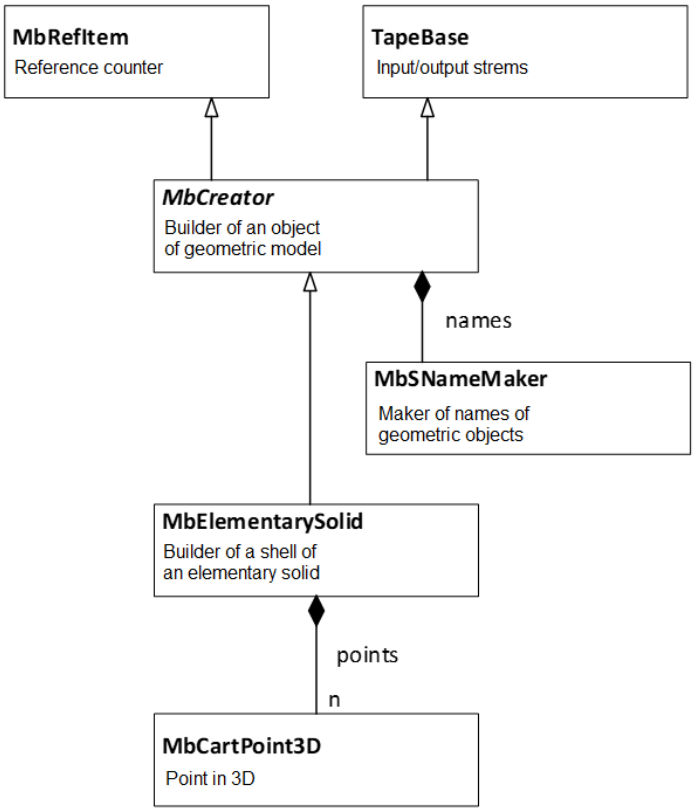

Fig. 5. The class diagram for the builder class of MbElementarySolid elementary solids shells. This diagram shows two types of relationships between classes — inheritance and aggregation.

Table 1. Types of elementary solids

№	Solid type	Shell type code solidType	Number of control points when constructing
1.	Sphere	et_Sphere	3 points
2.	Torus	et_Torus	3 points
3.	Cylinder	et_Cylinder	3 points
4.	Cone	et_Cone	3 points
5.	Block	et_Block	4 points
6.	Wedge	et_Wedge	4 points
7.	Prism	et_Prism	Number of base points + 1 point
8.	Pyramid	et_Pyramid	Number of base points +1 point
9.	Plate with rounded ends	et_Plate	4 points

The coordinates of the control points when constructing elementary solids are transferred to the builder class as an array of three-dimensional points. Points in this array have the following meaning:

1) points[0] sets the origin of the local coordinate system.
2) For a sphere, torus, cylinder and cone:
   Position points[1] specifies the direction of the Z axis of the local SC.
   Position points[2] specifies the direction of the X axis of the local SC.
3) For block, wedge and plate:
   points[1] specifies the direction of the X axis of the local SC.
   points[2] specifies the direction of the Y axis of the local SC.

Thus, points[0], points[1] and points[2] determine the location and orientation of an elementary solid body. But they also set the size of the body to build. The following rules are used for this:

1) points [1] defines the height of the cylinder, the height of the cone, the large radius of the torus, the block length, the length of the wedge, the plate thickness (to specify the radius of the sphere points[1] is not used).
2) points [2] defines the radius of the cylinder, the radius of the cone, the radius of the sphere, the small radius of the torus, the width of the block, the width of the wedge, the width of the plate.
3) The last point determines the height of the block, wedge, plate, prism, the top of the pyramid.

The assignment of parameters for the construction of elementary solids resembles the parameters used for the construction of elementary surfaces (Lesson No. 4, "Main types of surfaces", p. 5).

Using the builder class to build a solid model of a cylinder (Fig. 6) is shown in Example 3.1. The header file for this class is cr_elementary_solid.h.

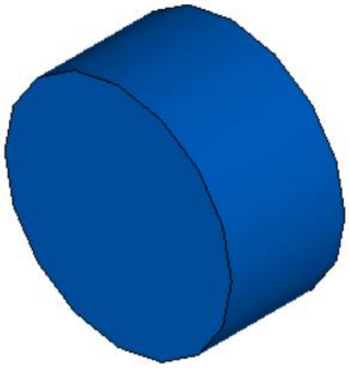

Fig. 6. Elementary solid – a cylinder (examples 3.1 и 3.2).

Example 3.1. The construction of an elementary body - a cylinder - using the explicit use of the shell builder class (Fig. 6).

```
#include "action_solid.h"
#include "cr_elementary_solid.h"
#include <vector>

using namespace std;
using namespace c3d;
using namespace TestVariables;

void MakeUserCommand0()
{
  // Default name maker
  MbSNameMaker names(1, MbSNameMaker::i_SideNone, 0);

  // An array of control points for building a cylinder
  SArray<MbCartPoint3D> points(3);
  // The local cylinder SC is shifted by 50 units along the Y axis of the global SC.
  points.Add(MbCartPoint3D(0, 50, 0));
  // The direction of the Z axis of the local SC coincides with the direction of the Z
  // axis of the global SC, the height of the cylinder is 10 units.
  points.Add(MbCartPoint3D(0, 50, 10));
```

```
  // The direction of the X axis of the local SC coincides with the direction of the
  // X axis of the global SC, the cylinder radius is 10 units.
  points.Add(MbCartPoint3D(10, 50, 0));

  // Creating a builder to build the shell of an elementary body - a cylinder
  MbElementarySolid* pCylCreator = new MbElementarySolid( points, et_Cylinder, names );
  // Build a cylinder shell using a builder object
  MbFaceShell* pShell = NULL;
  if ( pCylCreator )
    pCylCreator->CreateShell( pShell, cm_Same );
  // Building a solid
  MbSolid* pCyl = NULL;
  if ( pShell )
    pCyl = new MbSolid( pShell, pCylCreator );

  // Display of the constructed body
  if ( pCyl )
    viewManager->AddObject(TestVariables::SOLID_Style, pCyl);

  // Reducing the reference count of dynamically generated kernel objects
  ::DeleteItem(pCylCreator);
  ::DeleteItem(pShell);
  ::DeleteItem(pCyl);
}
```

When directly calling class constructors in Example 3.1, three steps were required:

1) create a builder of a MbElementarySolid class;
2) call the MbElementarySolid::CreateShell builder method to build a shell;
3) call the constructor of the MbSolid class and transfer to it the construction object and the shell that this object built.

When dynamically creating the objects listed above, Example 3.1 provides for minimal error handling - checking the values of pointers returned when dynamically creating objects using the new operator.

To reduce the typical call sequences for creating the necessary builders and shells, C3D Modeler has a set of utility functions that simplify the construction of solids. In particular, elementary bodies can be constructed using the utility-function ::ElementarySolid (header file action_solid.h). This function has already been used previously in examples 2.2 and 2.3 to build a pyramid. A prototype of this function is shown below:

```
MbResultType ElementarySolid(const SArray<MbCartPoint3D>& points,
      ElementaryShellType solidType, const MbSNameMaker & names, MbSolid *& result);
```

The first three input parameters are similar to the constructor parameters for the MbElementarySolid builder class. The last output parameter, result, is used to obtain the constructed solid body. In the case of a successful build, the function returns the rt_Success value, in the case of an error, one of the values of the MbResultType type (analysis of this value allows to clarify the cause of the error). Using the utility function to build a cylinder is shown in Example 3.2.

Example 3.2. Construction of an elementary solid using the utility function ::ElementarySolid (fig. 6).

```
void MakeUserCommand0()
{
  // Default name maker
  MbSNameMaker names(1, MbSNameMaker::i_SideNone, 0);

  // An array of control points for building a cylinder
  SArray<MbCartPoint3D> points(3);
```

```
  // The local cylinder SC is shifted by 50 units along the Y axis of the global SC
  points.Add(MbCartPoint3D(0, 50, 0));
  // The direction of the Z axis of the local SC coincides with the direction of
  // the Z axis of the world SC, the height of the cylinder is 10 units
  points.Add(MbCartPoint3D(0, 50, 10));
  // The direction of the X axis of the local SC coincides with the direction of
  // the X axis of the world SC, the cylinder radius is 10 units
  points.Add(MbCartPoint3D(10, 50, 0));

  // Construction of an elementary body - a cylinder - by three points
  MbSolid* pCyl = NULL;
  ::ElementarySolid(points, et_Cylinder, names, pCyl);

  // Display of the constructed body
  if ( pCyl )
    viewManager->AddObject(TestVariables::SOLID_Style, pCyl);

  // Reducing the reference count of a dynamically created kernel object
  ::DeleteItem(pCyl);
}
```

3.1 Tasks

1) In C3D Modeler developer's guide (found in the C3D_Manual_English.pdf file that comes with the kernel) read section M.1.1. "Constructing an elementary body." It presents illustrations explaining the geometric meaning of the control points used to construct various elementary bodies.
2) In the online documentation for C3D Modeler, find the section "The Geometric Modelling Module \ Geometric Construction Methods \ Solid modeling" (this section is available at the address https://c3d.ascon.net/doc/math/group___solid___modeling.html). Review the list of utility functions listed in this section. They implement various methods of constructing solids through the implicit use of the corresponding builder classes. The parameter sets of these functions depend on the operation being implemented.
3) Implement a function (for example, call it DrawVertsAndEdges) to display all edges and vertices of an arbitrary solid body that should be passed to this function as a link to an object of the MbSolid class. Use this function to display the edges and vertices of the cylinder from Example 3.2. When solving this problem, you can use the results from task 2.1.3, in which you wanted to display the edges of the pyramid in the test application. To display the vertices, use "the Point-Frame" MbPointFrame class, and for sorting the vertices of a solid body, use the MbSolid::GetVertex method.
4) Using the ::ElementarySolid utility function, construct an elementary sphere solid and display the edges and vertices of this solid using the function from the previous task. Similarly, build elementary solids of other types: a wedge, a prism, a plate with rounded ends.

4. Construction of extrusion solids

To build extrusion solids, use the utility function ::ExtrusionSolid from the action_solid.h header file. This function uses the MbCurveExtrusionSolid builder class. After construction, the construction object is saved in the body building log.

The function ::ExtrusionSolid has a fairly large number of parameters. The prototype of this function and the parameters, divided by purpose into three groups, are given below.

```
MbResultType ExtrusionSolid(
                  const MbSweptData& sweptData,
```

```
const MbVector3D& direction,
const MbSolid* pSolid1, const MbSolid* pSolid2,
bool checkIntersection,
const ExtrusionValues& params,
const MbSNameMaker& operNames,
const PArray<MbSNameMaker>& contoursNames,
MbSolid*& result );
```

Required input parameters of the function:

1) sweptData – generating curve data;
2) direction – extrusion direction vector;
3) params – extrusion operation parameters (in particular, extrusion depth values in the forward and reverse directions);
4) operNames – operation name maker;
5) contoursNames – name maker of the segments of the generating contour;

Optional input parameters to limit the extrusion depth to external bodies:

1) pSolid1 – bounding body in the forward direction. If it is set, then extrusion will be performed in the forward direction to the nearest faces of this body.
2) pSolid2 – bounding body in the opposite direction. If it is set, then extrusion will be performed in the opposite direction to the nearest faces of this body.
3) checkIntersection – if the pSolid1 and pSolid2 bodies are specified, then this flag specifies the combination of pSolid1 and pSolid2 bodies with intersection checking;

Output data:

1) Return value is a rt_Success in case of successful construction or the result code of the MbResultType operation, explaining the error that occurred.
2) result – built solid body.
3) params – modified extrusion parameters that are used to build elements of an array of operations up to the surface.

The call of the utility function for constructing the extrusion body is shown in Example 4.1. The constructed body is shown in fig. 7. To build the generatrix of the body as an array of contours, a separate function is used, CreateSketch (this approach was already used in the Lesson 3 “Composite curves and splines in two-dimensional space”, example 2.5).

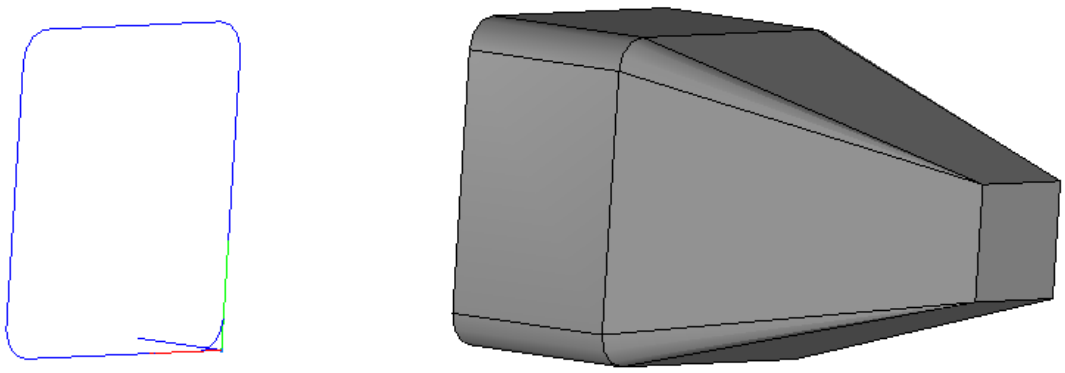

Fig. 7. Body extrusion (example 4.1). The generator (shown on the left) is given in the form of a square with rounded corners. Extrusion in the forward direction is doubled to a depth than in the opposite direction. In the forward direction, the slope angle is set.

Example 4.1. Constructing extrusion solid (Fig. 7).

```
#include <vector>
#include "alg_curve_fillet.h"

using namespace std;
using namespace c3d;
using namespace TestVariables;

// Auxiliary function for constructing a generator in the form of a square with
// fillets.
void CreateSketch(RPArray<MbContour>& _arrContours)
{
  // Creation of an array of square points, to which further fillets will be added.
  // Array size - 8 points to account for points of four rounding segments.
  SArray<MbCartPoint> arrPnts( 8 );
  arrPnts.Add( MbCartPoint(  0, 0 ) );
  arrPnts.Add( MbCartPoint( 50,  0 ) );
  arrPnts.Add( MbCartPoint( 50, 50 ) );
  arrPnts.Add( MbCartPoint(  0, 50 ) );

  // Construction of a single broken line of the outer contour by points
  MbPolyline* pPolyline = new MbPolyline(arrPnts, true);
  MbContour* pContourPolyline = NULL;

  // Specifying fillets using the function FilletPolyContour
  ::FilletPolyContour(pPolyline, 5, false, arrPnts[4], pContourPolyline);

  // Setting indices of points at which the fillet will be specified, taking into
  // account the addition of a new point during fillet using
  // the FilletTwoSegments function
  ptrdiff_t idxSideRight1 = 0;
  ptrdiff_t idxSideRight2 = 2;
  ptrdiff_t idxSideRight3 = 4;

  // Adding the fillets
  pContourPolyline->FilletTwoSegments(idxSideRight1, 5);
  pContourPolyline->FilletTwoSegments(idxSideRight2, 5);
  pContourPolyline->FilletTwoSegments(idxSideRight3, 5);

  _arrContours.push_back(pContourPolyline);
}


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

  // Creating a generating curve of an extrusion solid
  RPArray<MbContour> arrContours;
  CreateSketch(arrContours);

  // Display the generatrix (in the XY plane of the global CS)
  for (int i = 0; i<arrContours.size(); i++)
    viewManager->AddObject( Style(1, LIGHTRED), arrContours[i], &pl);

  // CONSTRUCTION OF THE EXTRUSION SOLID
  // The generator is located on the XY plane of the global SC.
  // Important note: a plane object must be created dynamically,
```

```
  // since it continues to be used in a solid object after exiting this function.
  MbPlane* pPlaneXY = new MbPlane( MbCartPoint3D(0, 0, 0),
                                   MbCartPoint3D(1, 0, 0),
                                   MbCartPoint3D(0, 1, 0) );
  // An object that stores the parameters of the generator
  MbSweptData sweptData(*pPlaneXY, arrContours);
  // Guide vector for extrusion operation
  MbVector3D dir(0, 0, -1);

  // The parameters of the extrusion operation that define the properties of the body
  // to build in the forward and backward directions (extrusion depth and slope)
  const double HEIGHT_FORWARD = 60.0, HEIGHT_BACKWARD = 30.0;
  const double ANGLE_FORWARD_DEGREE = 15.0;
  ExtrusionValues extrusionParams(HEIGHT_FORWARD, HEIGHT_BACKWARD);
  // Specify slope for extrusion in the forward direction
  extrusionParams.side1.rake = ANGLE_FORWARD_DEGREE * M_PI/180.0;

  // Name makers of the elements of a model of a solid body and generatrix
  MbSNameMaker operNames( 1, MbSNameMaker::i_SideNone, 0);
  PArray<MbSNameMaker> cNames(0, 1, false);

  // Calling an utility function to build extrusion solids
  MbSolid* pSolid = NULL;
  MbResultType res = ::ExtrusionSolid( sweptData, dir, NULL, NULL, false,
                                       extrusionParams, operNames, cNames, pSolid);

  // Display of the constructed body
  if (res == rt_Success)
  {
    // The movement of the body along the Y axis, so that when displayed
    // it does not overlap the generator
    pSolid->Move( MbVector3D(0, 100, 0) );
    viewManager->AddObject(Style(1, LIGHTGRAY), pSolid);
  }

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

In Example 4.1, in the extrusionParams extrusion parameters structure, the extrusion depth and the forward slope were specified. The ExtrusionValues class inherits from the SweptValuesAndSides class “Extrusion or Revolution Parameters”. Among these parameters there are surfaces that bound the execution of the construction operation in the forward and in the opposite direction. These surfaces can be set implicitly by passing the bounding solids in the form of the optional input parameters pSolid1 and pSolid2 to the function ::ExtrusionSolid. You can also specify such surfaces explicitly in the ExtrusionValues object. This method is demonstrated in Example 4.2 for constructing an extrusion solid bounded in a straight direction by a cylindrical surface (Fig. 8). The generator of this solid coincides with the generator of Example 4.1.

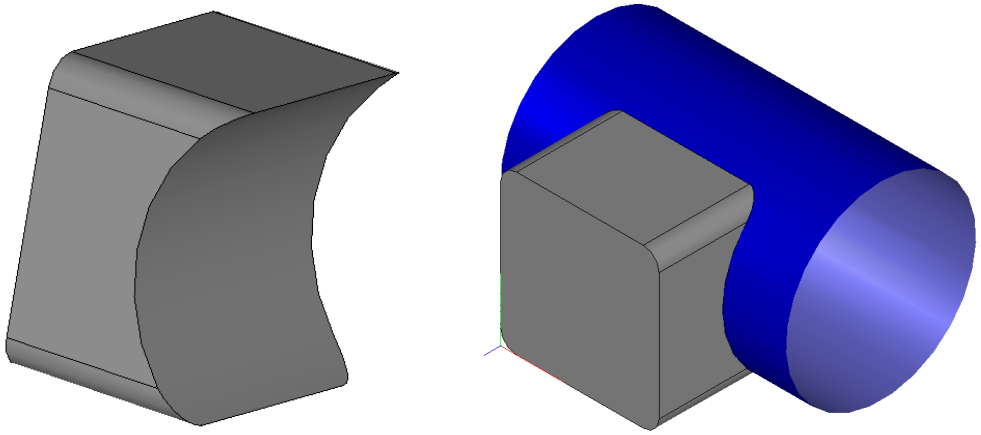

Fig. 8. Extrusion solid (Example 4.2), bounded by a cylindrical surface. The result of the construction is shown when the bounding surface is turned off (left) and on (right).

Example 4.2. Constructing the extrusion body before intersection with the surface (Fig. 8).

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

  // Calling an auxiliary function for constructing a generator (from Example 4.1)
  RPArray<MbContour> arrContours;
  CreateSketch(arrContours);

  // Construction of the bounding cylindrical surface
  MbSurface* pCylSurf = NULL;
  SArray<MbCartPoint3D> arrPnts(3);
  arrPnts.Add( MbCartPoint3D( -20, 25, -50 ) );
  arrPnts.Add( MbCartPoint3D(  70, 25, -50 ) );
  arrPnts.Add( MbCartPoint3D(   0, 55, -50) );
  ::ElementarySurface( arrPnts[0], arrPnts[1], arrPnts[2],
                       st_CylinderSurface, pCylSurf );

  // Cylindrical surface displaying
  if ( pCylSurf )
    viewManager->AddObject(Style(1, LIGHTBLUE), pCylSurf );

  // Forming the parameters of the generatrix for the operation of building
  // the extrusion solid
  MbPlane* pPlaneXY = new MbPlane( MbCartPoint3D(0, 0, 0), MbCartPoint3D(1, 0, 0),
                                   MbCartPoint3D(0, 1, 0));
  MbSweptData sweptData(*pPlaneXY, arrContours);

  // Guide vector for extrusion operation
  MbVector3D dir(0, 0, -1);

  // Extrusion operation parameters
  ExtrusionValues extrusionParam;
  // Specifying the surface to which extrusion will be performed
  // in the forward direction
  extrusionParam.SetSurface1( pCylSurf );
  // Specify the method of extrusion in the forward direction –
  // to the intersection with the surface
```

```
  extrusionParam.side1.way = sw_surface;

  // Objects for naming elements of a model of a solid body and generatrix
  MbSNameMaker operNames( 1, MbSNameMaker::i_SideNone, 0);
  PArray<MbSNameMaker> cNames(0, 1, false);

  // Calling the utility function to construct the extrusion body
  MbSolid* pSolid = NULL;
  MbResultType res = ::ExtrusionSolid( sweptData, dir, NULL, NULL, false,
                                       extrusionParam, operNames, cNames, pSolid );

  // Display of the constructed body
  if (res == rt_Success)
    viewManager->AddObject(Style(1, LIGHTGRAY), pSolid);

  // Reducing reference counters for dynamic kernel objects
  ::DeleteItem( pSolid );
  ::DeleteItem( pCylSurf );
}
```

By selecting the type of generator and parameters of ExtrusionValues, you can build a large number of different solids. The generator does not have to be a single plane contour and does not need to be closed. A two-dimensional contour for constructing the extrusion body can be located both on a plane and curved surface. If a set of non-intersecting two-dimensional contours is located on one surface, then when constructing an extrusion body, the considered method determines external contours and internal contours. Contouring can be multiple.

4.1 Tasks

1) Perform examples 4.1 and 4.2. Using the function from task 3.1.3, display the edges and vertices of the constructed bodies.
2) The ExtrusionValues::thickness1 parameter defines the wall thickness outward from the generator curve, and ExtrusionValues::thickness2 - inside. In Example 4.1, in the extrusion parameters, remove the slope angle in the forward direction and try to build a body from this example, setting a wall thickness for it equal to 10 (first outward, then inward, and then simultaneously outward and inward of the generator).
3) The bool parameter ExtrusionValues::shellClosed allows you to build a closed (by default, true) or open (false) shell. In the original example 4.1 (in its original form, without specifying the wall thickness) try to build a solid body with an open shell, indicating the value of the parameter:

```
extrusionParams.shellClosed = false;
```

4) Construct the extrusion body in the form of a rectangular parallelepiped bounded on both sides (in the forward and in the opposite extrusion direction) with spherical surfaces of different radii.
5) According to fig. 9 build generators consisting of three (Fig. 9, left) and four (Fig. 9, right) non-intersecting contours and perform an extrusion operation for them.

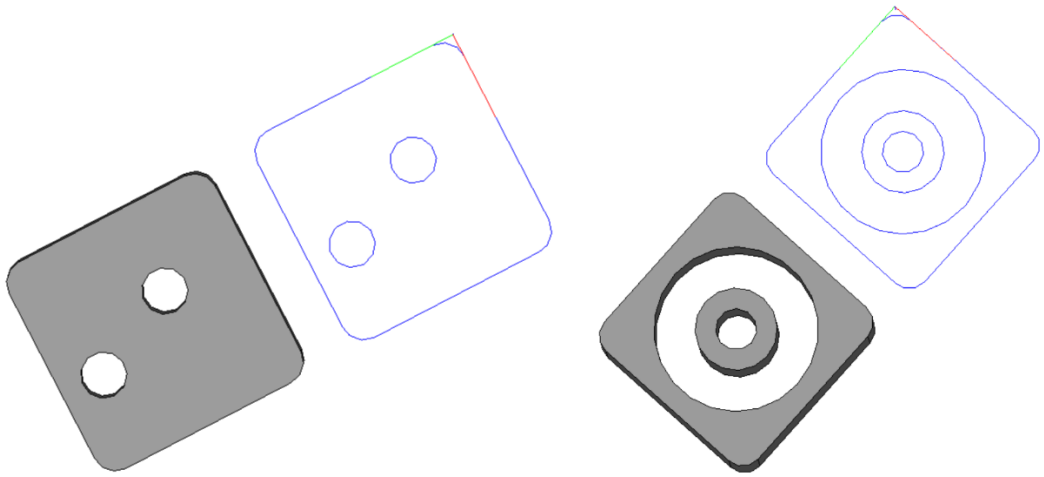

Fig. 9. Examples of generators and constructed extrusion bodies for task 4.1.4.

6) Build an open curvilinear contour using a NURBS object constructed from a set of points (see Lesson 3, “Composite curves and splines in two-dimensional space”) and apply an extrusion operation to it, specifying a certain wall thickness. To build an extrusion solid along an open loop, the value of the shellClosed parameter must be false. An example of a built extrusion solid along an open contour is shown in Fig. 10. If the function ::ExtrusionSolid returns an error when trying to build a solid, then try to reduce the wall thickness (so that no solid intersections occur in the vicinity of the spline's inflection points).

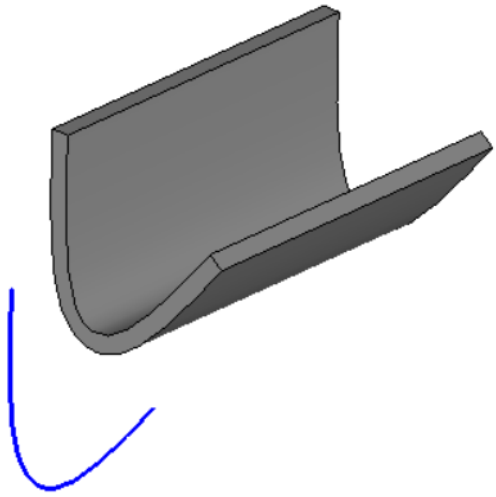

Fig. 10. An example of a built body for task 4.1.6.

5. Construction of revolution solids

Along with extrusion solids, revolution solids are one of the most frequently encountered solids in geometric modeling. To build the solids of revolution in C3D, the utility function ::RevolutionSolid (header file action_solid.h) is used, which has the following prototype:

```
MbResultType RevolutionSolid(
                const MbSweptData& sweptData,
                const MbAxis3D& axis,
                RevolutionValues& params,
                const MbSNameMaker& names,
                PArray<MbSNameMaker>& contoursNames,
```

```
                MbSolid*& result );
```

Function input parameters (all are required):

1) sweptData – generating curve data;
2) axis – axis of revolution;
3) params – revolution operation parameters;
4) names – operation name maker;
5) contoursNames – name maker of the segments of the generating contour;

Output data:

1) Return value is a rt_Success in case of successful construction or the result code of the MbResultType operation, explaining the error that occurred.
2) result – built solid body.

The utility functions for constructing solids of revolution and extrusion have similar prototypes. They use the same MbSweptData class to describe the generator curve. The revolution axis is set by the axis parameter. The revolution angle for the construction of the body is specified in the RevolutionValues constructor and specified in radians.

The RevolutionValues class is intended for storing parameters of the revolution operation. This class (like the ExtrusionValues class) is inherited from the SweptValuesAndSides class. The parameters of the shell closure, the wall thickness of the thin-walled solid of revolution and the bounding surfaces can be set in the same way as was done above for the extrusion bodies.

Example 5.1 demonstrates the construction of a body of revolution (Fig. 11) using the generator of Example 4.1. The axis of revolution is parallel to the Y axis of the global SC and is obtained by transferring the Y axis along the X axis to a given vector. In the text of Example 5.1, after creating the revParams object, there are several commented lines that demonstrate the setting of various parameters of the revolution operation that affect the shape of the resulting body.

Example 5.1. The construction of the body of revolution (Fig. 9).

```
void MakeUserCommand0()
{
  // A factor to convert angular values from degrees to radians
  const double DEG_TO_RAD = M_PI / 180.0;

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

  // Calling a function to build a generator (from Example 4.1)
  RPArray<MbContour> arrContours;
  CreateSketch(arrContours);

  // Generator displaying (in the XY plane of the global CS)
  for (int i = 0; i<arrContours.size(); i++)
    viewManager->AddObject(Style(1, RGB(0, 0, 255)), arrContours[i], &pl);

  // Preparation of parameters for calling the function of constructing
  // a body of revolution.
  // sweptData - object that stores information about the generatrix.
  MbPlane* pPlaneXY = new MbPlane( MbCartPoint3D(0, 0, 0), MbCartPoint3D(1, 0, 0),
                                   MbCartPoint3D(0, 1, 0) );
  MbSweptData sweptData(*pPlaneXY, arrContours);

  // Parameter object for the operation of constructing a body of revolution.
  // The first two parameters of the constructor set the revolution angles in the
forward
  // and reverse direction (normal to the generator or opposite to this normal).
```

```
  // In this example, the revolution is 120 degrees in the forward direction.
  // The third parameter sets the shape of the solid topology.
  // For s = 0, a body is constructed with the topology of the sphere;
  // for s = 1, with a topology of the torus.
  // If the generator is an open curve, and the axis of revolution lies in the plane of
  // the curve, then when s = 0, the generator will be automatically extended to the
  // axis of revolution when built. In this example, there are no differences between
the
  // s values because the generator has the form of a closed contour.
  RevolutionValues revParms( 120*DEG_TO_RAD, 0, 0);

  // The next line defines the construction of a body with an open shell ("empty").
  // revParms.shellClosed = false;
  // Body wall thickness setting
  // revParms.thickness1 = 8;

  // Name makers for the operation of building a body of revolution and
  // for the contours of the generatrix
  MbSNameMaker operNames( 1, MbSNameMaker::i_SideNone, 0);
  PArray<MbSNameMaker> cNames(0, 1, false);

  // Revolution axis for body construction:
  // Y axis of the global SC is shifted by -50 units along the X axis.
  MbAxis3D axis(pl.GetAxisY());
  axis.Move( MbVector3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(-50, 0, 0)) );

  // Calling a utility function for constructing a solid body of revolution
  MbSolid* pSolid = NULL;
  MbResultType res = ::RevolutionSolid( sweptData, axis, revParms,
                                        operNames, cNames, pSolid );

  // Display of the constructed body
  if (res == rt_Success)
    viewManager->AddObject(Style(1, LIGHTGRAY), pSolid);

  // Reducing reference counters for dynamic kernel objects
  ::DeleteItem( pSolid );
}
```

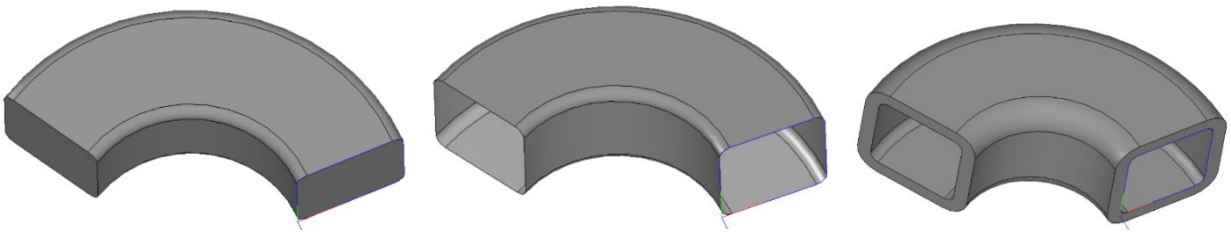

Fig. 11. The bodies of revolution obtained using Example 5.1. (On the left - the operation parameters are set only in the RevolutionValues constructor; in the center – revParms.shellClosed = false; on the right - shellClosed = true, thickness1 = 8.)

The body of revolution can be obtained by performing revolution to a given bounding surface. The bounding surface can be specified for one or two directions of revolution. Example 5.2 shows the construction of a body of revolution (Fig. 12) using a cylindrical bounding surface for revolution in the forward direction. The bounding surface and how to use it is set using the RevolutionValues class.

Example 5.2. Building a revolution solid to the bounding surface (Fig. 12).

```
void MakeUserCommand0()
{
  // A factor to convert angular values from degrees to radians
  const double DEG_TO_RAD = M_PI / 180.0;

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

  // Calling a function to build a generator (from Example 4.1)
  RPArray<MbContour> arrContours;
  CreateSketch(arrContours);

  // Construction of the bounding cylindrical surface
  MbSurface* pCylSurf = NULL;
  SArray<MbCartPoint3D> arrPnts(3);
  arrPnts.Add(MbCartPoint3D(-50, -25, -75));
  arrPnts.Add(MbCartPoint3D(-50, 75, -75));
  arrPnts.Add(MbCartPoint3D(-20, -25, -75));
  ::ElementarySurface( arrPnts[0], arrPnts[1], arrPnts[2],
                       st_CylinderSurface, pCylSurf );

  // Display bounding surface
  if ( pCylSurf )
    viewManager->AddObject( Style(1, LIGHTBLUE), pCylSurf );

  // Preparation of parameters for calling the function of constructing
  // a body of revolution.
  // sweptData - object that stores information about the generatrix.
  MbPlane* pPlaneXY = new MbPlane( MbCartPoint3D(0, 0, 0), MbCartPoint3D(1, 0, 0),
                                   MbCartPoint3D(0, 1, 0) );
  MbSweptData sweptData(*pPlaneXY, arrContours);

  // Parameter object for the operation of constructing a body of revolution.
  RevolutionValues revParms;
  // Specify the surface that will be rotated in the forward direction.
  revParms.SetSurface1( pCylSurf );
  // Indication of the method of performing the rotation operation
  // in the forward direction - before intersection with the surface.
  revParms.side1.way = sw_surface;

  // Name makers for the operation of building a body of revolution and for
  // the contours of the generatrix
  MbSNameMaker operNames( 1, MbSNameMaker::i_SideNone, 0);
  PArray<MbSNameMaker> cNames(0, 1, false);

  // Rotation axis for body construction:
  // Y axis of the world IC is shifted by -50 units along the X axis.
  MbAxis3D axis(pl.GetAxisY());
  axis.Move( MbVector3D(MbCartPoint3D(0, 0, 0), MbCartPoint3D(-50, 0, 0)) );

  // Calling a utility function for constructing a solid body of revolution
  MbSolid* pSolid = NULL;
  MbResultType res = ::RevolutionSolid( sweptData, axis, revParms,
                                        operNames, cNames, pSolid );

  // Display of the constructed body
  if (res == rt_Success)
    viewManager->AddObject(Style(1, LIGHTGRAY), pSolid);
```

```
  // Reducing reference counters for dynamic kernel objects
  ::DeleteItem( pCylSurf );
  ::DeleteItem( pSolid );
}
```

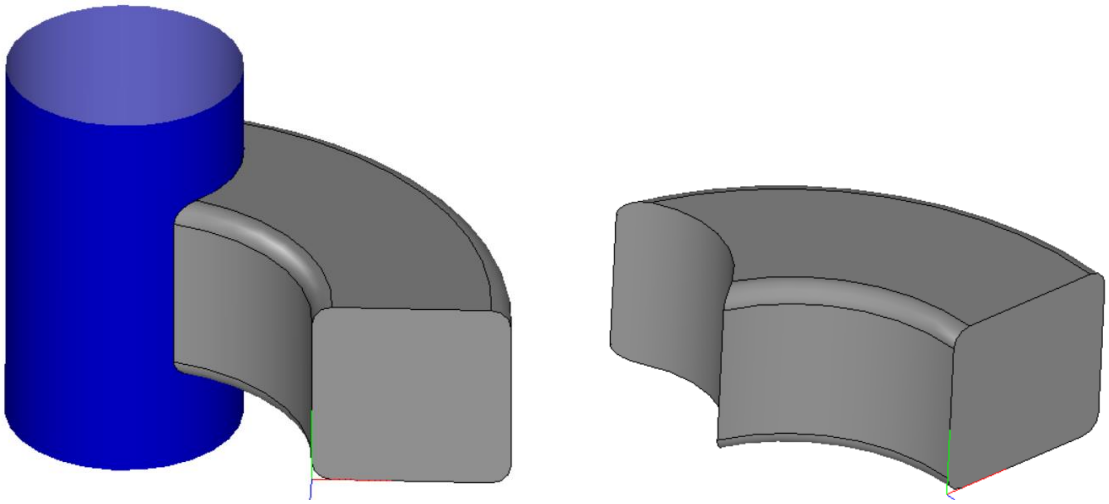

Fig. 12. Revolution solid (Example 5.2), bounded by a cylindrical surface. The result of the construction is shown with the bounding surface displayed (left) and off (right).

5.1 Tasks

1) Using the generators from task 4.1.5, construct the solids of revolution shown in Fig. 13. The operation for constructing revolution solids, as well as the construction operation for extrusion solids, automatically classifies nonintersecting contours into internal and external. Select the values of the RevolutionValues parameters so that the constructed solids look like solids in fig. 13.

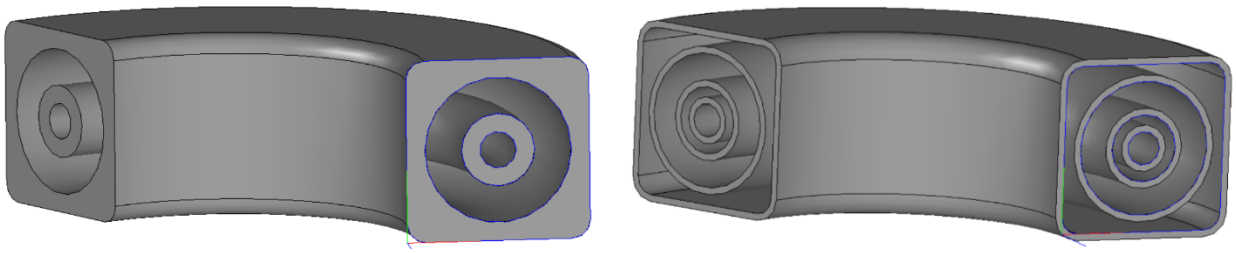

Fig. 13. Samples of revolution solids for the task 5.1.1.

2) Build a thin-walled solid of revolution using the open-curved generatrix from task 4.1.6. To build a revolution solid with a generator in the form of an open contour, the value of the shellClosed parameter must be true.

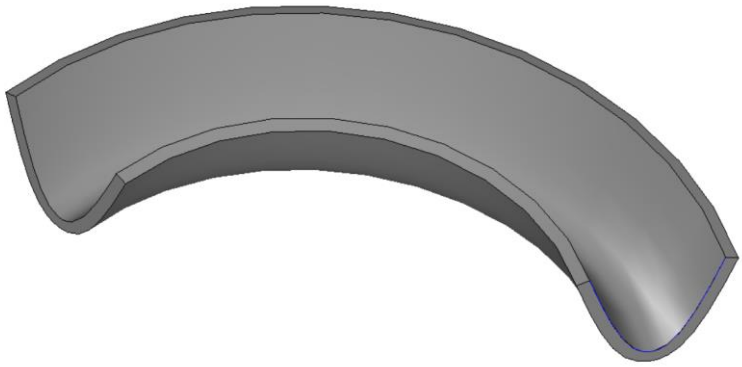

Fig. 14. Sample revolution solid for task 5.1.2.

3) When constructing a revolution solid obtained in task 5.1.2, for one of the directions of rotation, specify the bounding spherical surface.

6. Conclusion

The lesson addressed a number of issues related to the construction of models of solids using C3D Modeler. The main class for representing solids is the MbSolid "Solid" class. This class is inherited from the MbItem abstract class "Geometric model object".

A characteristic feature of the objects of the geometric model is that the build log is stored in these objects. This log is implemented using the MbTransactions class (this is one of the MbItem parent classes). The log stores information about kernel operations that must be performed to build a solid. These operations are represented as objects of the building classes inherited from the abstract class MbCreator. You can see a significant complication of the programmatic representation of solids compared to geometric objects like curves and surfaces, considered in previous lessons.

For the construction of solids, utility functions are usually used that implement various construction operations and perform construction error checking. The lesson discusses three functions of this type: ::ElementarySolid for the construction of elementary solids, ::ExtrusionSolid for extrusion solids and ::RevolutionSolid for solids of revolution. The parameters of construction operations are set using the parameters of these functions, as well as using objects of classes specific to a particular operation. The lesson demonstrates the influence of some of the construction parameters on the appearance of the resultant solid (for example, the construction of solid and "empty" bodies, the construction of thin-walled solids, the construction of solids bounded by given surfaces).