Table of Contents

1 Introduction

What is sgCore?

sgCore is a solid modeling library created by the Geometros company. This library is a kernel of the parametrical CAD system SolidGraph.
sgCore was developed on C++ only, exports about 30 classes and realizes many algorythms for working with 2D and 3D objects.

What are the main features of sgCore?

The sgCore library has all the necessary structures and functions to build the full-featured CAD system on its basis.

Here is the list of the sgCore library main features :

- creating 2D primitives in the 3D space - points, circles, arcs, splines, contours.
- the basic algorythms for working with 2D objects are: finding the plane where the object is lying, self-intersecting control, finding points of objects intersection, checking the multiplicity of closed objects.
- creating equidistants for 2D objects with various shifts and with a feature of rounding angles.
- creating 3D primitives - spheres, boxs, cones, cylinders, toruses, ellipsoids, spherical bands.
- supports both 3D solids and surfaces
- creating groups of objects
- feature to appoint the user's block of data to any object (***with further saving of this block in a file***)
- calculating of the UV texture coordinates for each polygon vertex using cubic, spherical and cylindric methods of textures mapping.
- 3D polygonal objects triangulation
- flat closed areas with holes triangulation
- boolean operations with objects - intersection, joining, subtraction
- finding intersection lines of 3D objects
- finding the clips of 3D objects by arbitrary planes
- constructing solids and surfaces of revolution
- constructing solids and surfaces of extrusion an arbitrary flat area with holes along a curve
- constructing spiral solids and surfaces.
- constructing solids and surfaces from their clips
- constructing flat faces with holes on boundary contours
- creating solids by surfaces sewing
- constructing Coons surfaces from three or four boundary contours

- constructing surfaces from the ordered array of points
- Undo-Redo
- loading AutoCAD SHX fonts
- creating text objects
- creating dimensional objects (distant, radial, diametral, angle)
- ***saving objects in the user-defined formats***
- ***loading objects from a file***
- ***DXF import/export***

(***the bold*** items are available in the chargeable version only)

2 Work with sgCore

Working with sgCore.

To use the sgCore library in your project properly you should do the following:

1) Include the library - sgCore.lib

2) Include the header file - sgCore.h

3) Initialize the library by calling the sgInitKernel() function

4) In order to prevent memory leakage you should finish working with the library by calling the sgFreeKernel() function

3 Functions

Functions

The following functions are defined in the sgCore library:

sgInitKernel
sgFreeKernel
sgGetLastError
sgGetVersion

3.1 sgInitKernel

bool sgInitKernel()

Description:

sgCore initialization. You should call this function at the beginning of your work and only once. If you don't call this function sgCore may work incorrectly. You must call the sgFreeKernel() function to finish working with sgCore

Arguments:

No arguments.

Returned value:

If the library was successfully initialized the function returns true. Otherwise - false.

Example (the shortest sgCore library using program)

```
int main(int argc, char* argv[])
{
   sgInitKernel();
   sgFreeKernel();
   return 0;
}
```

See also:
sgFreeKernel()

3.2 sgFreeKernel

void sgFreeKernel()

Description:

Finishes the work with the library correctly. You must call this function only once.
You must begin the work with sgCore by calling the sgInitKernel() function

Arguments:

No arguments.

Returned value:

No values.

Example (the shortest sgCore library using program)

```
int main(int argc, char* argv[])
{
   sgInitKernel();
   sgFreeKernel();
    return 0;
}
```

See also:
sgInitKernel()

3.3 sgGetVersion

---

void sgGetVersion(int& major, int& minor, int& release, int& build)

Description:
Returns all the components of the current library version.

Arguments:
major - leading number of the kernel version
minor - fine number of the kernel version
release - number of the kernel release which is debugged and ready-to-use
build - number of the project build

Returned value:
The function arguments has the values of the library version values.

3.4 sgGetLastError

SG_ERROR sgGetLastError()

Description:
Returns the last error code.

Arguments:
No arguments.

Returned value:
The returned errors and their codes are defined in sgErrors.h

4 Geometries

Geometries.

The *Geometry* concept is one of the core concepts of the sgCore library.

The objects the user models with the help of the library have two basic points - a mathematical description of the object and its polygonal (for 3D objects) or segment (for 2D objects) presentation. The mathematical description of the object is called *geometry* in the sgCore library.

Geometry is an independent concept. Geometry describes a mathematical object with high accuracy and there is one-to-one relationship between them.
The geometry accuracy is limited only by the accuracy of the number presentation in the processor.

Examples of geometries:

- circle geometry. To describe a circle in the three-dimensional space you should set 7 numbers - a circle radius, three coordinates of the center and three coordinates of the normal vector. In fact, the SG_CIRCLE structure realizes this presentation.

- sphere geometry. To describe a sphere in the three-dimensional space you should set 4 numbers - a sphere radius and three coordinates of the center. The position of any object in space can be unambiguously described by an affine transformations matrix. This way of setting the 3D objects position is used in sgCore library. That is why you only need to know the sphere radius to describe the sphere geometry. If you want to transform the mathematical "sphere" description into a polygonal model you shoud also know the number of meridians and parallels.
  The SG_SPHERE structure realizes the sphere geometry described above.

Besides the fact that a geometry stores an accurate mathematical description of an object, it has one more advantage. It allows you to *quickly* draw a 2D object segment presentaion without creating the object itself. Thus, there is a Draw function for arcs and circles which draws this or that geometry segment by segment. The segment presentation of a spline is stored in the array of points and with its help you can draw a spline geometry.

See also
Objects

4.1 SG_POINT

SG_POINT structure

The SG_POINT structure is a point presentation in the three-dimensional space. It has the following fields

double x; - X point coordinate
double y; - Y point coordinate
double z; - Z point coordinate

Defined in sgDefs.h

4.2 SG_VECTOR

SG_VECTOR definition

It is the SG_POINT structure overriding. For you to understand the functions arguments meanings.

Defined in sgDefs.h

See also:
SG_POINT

4.3 SG_LINE

SG_LINE structure

The SG_LINE structure is a line segment presentation in the three-dimensional space.

Has the following fields
SG_POINT p1; - first line segment point
SG_POINT p2; - second line segment point

Defined in sgDefs.h

See also:
SG_POINT

4.4 SG_CIRCLE

SG_CIRCLE structure

The SG_CIRCLE structure is a circle representaion in three-dimesional space.

Has the following fields

double radius; - circle radius
SG_VECTOR normal; - normal to a circle surface
SG_POINT center; - circle center

Has the following methods:

---

bool FromCenterRadiusNormal(const SG_POINT& cen, double rad, const SG_VECTOR& nor)

Description:
Fills all the fields of the SG_CIRCLE structure with the arguments.

Arguments:
cen - circle center,
rad - circle radius,
nor - normal to a surface

Returned value:
In case of a zero-radius or if normal to a surface length is zero returns false, otherwise - true

---

bool FromThreePoints(const SG_POINT& p1, const SG_POINT& p2, const SG_POINT& p3)

Description:
Creates a circle in the space by three points.
Arguments:
p1 - first circle point,
p2 - second circle point,

p3 - third circle point,

Returned value:

In case the points coinsides or lies on the same line returns false, otherwise - true

---

bool Draw(SG_DRAW_LINE_FUNC line_func) const

Description:

For each line of the segment circle presentation it calls the line_func function. Enables you to quickly construct a circle without creating a sgCCircle object.

Arguments:

line_func - pointer to the function called for each line of a segment circle presentation

Returned value:

In case of a zero argument returns false, otherwise - true.

Defined in sg2D.h

See also:
SG_POINT SG_VECTOR sgCCircle SG_DRAW_LINE_FUNC

4.5 SG_ARC

SG_ARC structure

The SG_ARC structure is an arc presentation in the three-dimensional space.

Has the following fields

double	radius;	- arc radius
SG_VECTOR	normal;	- normal to the arc surface
SG_POINT	center;	- arc center
SG_POINT	begin;	- arc start point
SG_POINT	end;	- arc end point
double	begin_angle;	- arc beginning angle (in radians)
double	angle;	- arc opening angle (in radians)

Has the following methods (it is recommended to create arcs using one of the five

arcs creating methods):

---

bool FromThreePoints(const SG_POINT& begP, const SG_POINT& endP, const SG_POINT& midP, bool invert)

Description:

Creates an arc in space by three points.

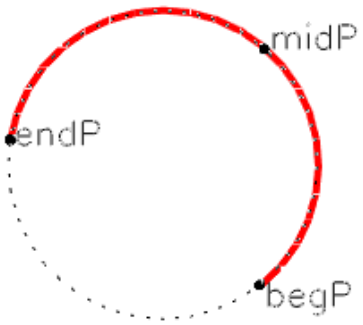
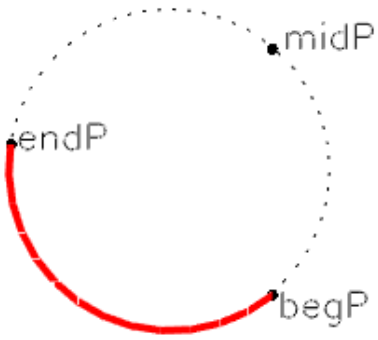
Arguments:

begP - arc start point
endP - arc end point
midP - arc point
invert - if false an arc with midP is created; if true an arc is "inverted", i.e. an arch complementing the one with midP to a circle will be created

Returned value:

If the function fails returns false, otherwise - true.

Illustration:

	invert = false
	invert = true

---

bool FromCenterBeginEnd(const SG_POINT& cenP, const SG_POINT& begP,const SG_POINT& endP, bool invert)

Description:

Creates an arc in the space by three points.

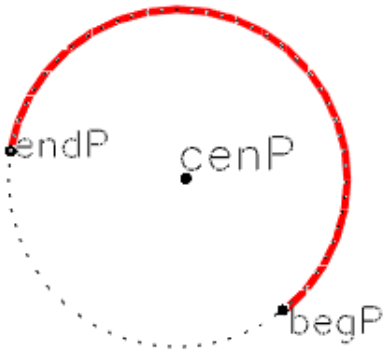
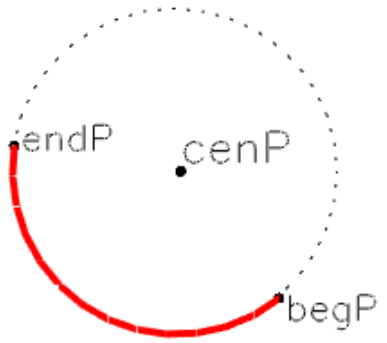
Arguments:

cenP - arc center
begP - arc start point
endP - arc end point
invert - if false an arc from begP to endP is created; if true the arc is "inverted", i.e. an arc from endP to begP is created

Returned value:

If the function fails returns false, otherwise - true.

Illustration:

	invert = false
	invert = true

bool FromBeginEndNormalRadius(const SG_POINT& begP, const SG_POINT& endP, const SG_VECTOR& nrmlV, double rad, bool invert)

Description:

Creates an arc in the space by start and end points, normal and radius.

Arguments:

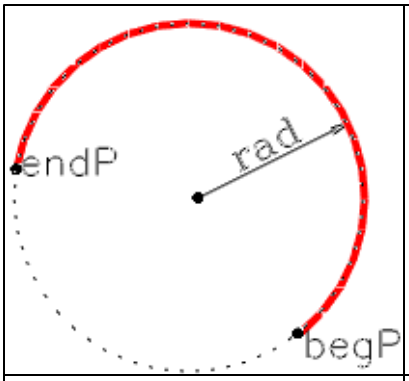
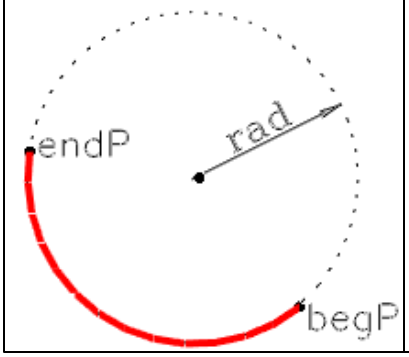
begP - arc start point
endP - arc end point
nrmlV - normal to an arc surface
rad - arc radius

invert - if false an arc from begP to endP is created; if true the arc is "inverted", i.e. an arc from endP to begP is created

Returned value:

If the function fails returns false, otherwise - true.

Illustration:

	invert = false
	invert = true

---

bool FromCenterBeginNormalAngle(const SG_POINT& cenP, const SG_POINT& begP, const SG_VECTOR& nrmlV, double ang)

Description:

Creates an arc in the space by the center, start point, normal and opening angle.

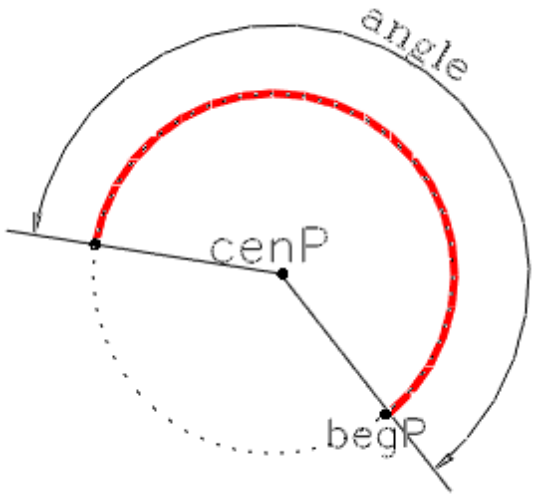
Arguments:

cenP - arc center
begP - arc start point
nrmlV - normal to an arch surface
ang - arc opening angle

Returned value:

If the function fails returns false, otherwise - true.

Illustration:



---

bool FromBeginEndNormalAngle(const SG_POINT& begP, const SG_POINT& endP,const SG_VECTOR& nrmlV, double ang)

Description:

Creates an arc in the space by the start and end points, normal and opening angle.

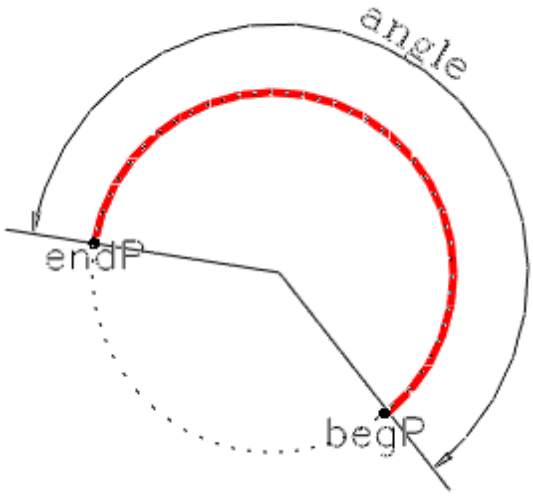
Arguments:

cenP - arc center
begP - arc start point
nrmlV - normal to an arch surface
ang - arc opening angle

Returned value:

If the function fails returns false, otherwise - true.

Illustration:



bool Draw(SG_DRAW_LINE_FUNC line_func) const

Description:
For each line of an arc segment presentation calls the line_func function. Enables you to quickly construct an arc without creating an sgCArc object.
Arguments:
line_func - pointer to a function called for each line of an arc segment presentation.
Returned value:
If the argument value is zero returns false, otherwise - true.

Defined in sg2D.h

See also:
SG_POINT SG_VECTOR sgCArc SG_DRAW_LINE_FUNC

4.6 SG_SPLINE

SG_SPLINE class

The SG_SPLINE structure is a presentation of the Bezier spline in the three-dimensional space.

Has the following fields

```
SG_POINT*   m_points;   -  array of spline points approximated by line
segments
int         m_points_count; - number of approximation points
SG_POINT*   m_knots;   -   array of spline knots
int         m_knots_count; - number of spline knots
```

Has the following methods:

---

static SG_SPLINE* Create();

Description:

The virtual class constructor. The SG_SPLINE::SG_SPLINE() constructor is closed, so that memory allocation for a spline and its deletion can take place with the means of the sgCore library. That's why you can create a spline only using the Create() static function. You must delete the spline after using it by calling the Delete(SG_SPLINE*) static function.

Arguments:

No arguments.

Returned value:

Returns the pointer to an empty spline which you must fill using the following functions AddKnot(), DeleteKnot(), MoveKnot().

---

static void Delete(SG_SPLINE* spl);

Description:

Deletes the spline previously created using the Create() function.

Arguments:

spl - a spline to be deleted

Returned value:

No values.

---

bool AddKnot(const SG_POINT& pnt, int nmbr);

Description:

Adds a knot to the spline

Arguments:
pnt - a new knot
nmbr - knot number.

Returned value:
If the knot is successfully added the function returns true, otherwise false

---

bool MoveKnot(int nmbr, const SG_POINT& pnt);

Description:
Moves a knot in the spline

Arguments:
pnt - new knot location
nmbr - knot number.

Returned value:
If the knot is successfully moved the function returns true, otherwise false

---

bool DeleteKnot(int nmbr);

Description:
Deletes a knot from the spline

Arguments:
nmbr - knot number.

Returned value:
If the knot is successfully removed the function returns true, otherwise false

---

bool IsClosed();

Description:
Checks whether the spline is closed

Arguments:
No arguments.

Returned value:
If the spline is closed the function returns true, otherwise false

---

bool Close();

Description:
Closes the spline.

Arguments:
No arguments.

Returned value:
If the spline has been successfully closed the function returns true, otherwise false

---

bool UnClose(int nmbr);

Description:
Uncloses the closed spline by a specified knot.

Arguments:
nmbr - number of the knot to unclose the spline by.

Returned value:
If the spline has been successfully unclosed the function returns true, otherwise false

---

const SG_POINT* GetPoints() const;

Description:
Returns the pointer to the array of the spline appoximation points.

Arguments:
No arguments.

Returned value:
The function Returns the pointer to the array of the spline appoximation points.

---

int GetPointsCount() const;

Description:
Returns a number of the spline approximation points.

Arguments:
No arguments.

Returned value:
The function returns a number of the spline approximation points.

---

const SG_POINT* GetKnots() const;

Description:
Returns the pointer to the array of the spline knots.

Arguments:
No arguments.

Returned value:
The function Returns the pointer to the array of the spline knots.

---

const SG_POINT* GetKnotsCount() const;

Description:
Returns the number of spline knots.

Arguments:
No arguments.

Returned value:
The function returns the number of spline knots.

Defined in sg2D.h

See also:
SG_POINT Example

4.7 SG_BOX

SG_BOX structure

The SG_BOX structure is a box presentation.

Has the following fields

```
double  SizeX;  - box length on the X axis
double  SizeY;  - box length on the Y axis
double  SizeZ;  - box length on the Z axis
```

Defined in sg3D.h

4.8 SG_SPHERE

SG_SPHERE structure

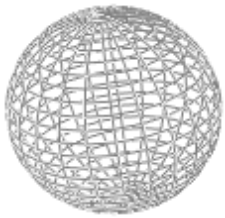
The SG_SPHERE structure is a presentation of a polygonal sphere.

Has the following fields

```
double  Raduis;           - sphere radius
short   MeridiansCount    - the number of sphere meridians
short   ParallelsCount;   - the number of sphere parallels
```

Illustration:

	MeridiansCount = 3 ParallelsCount = 5
	MeridiansCount = 10 ParallelsCount = 5

	MeridiansCount = 24 ParallelsCount = 24

Defined in sg3D.h

4.9 SG_CYLINDER

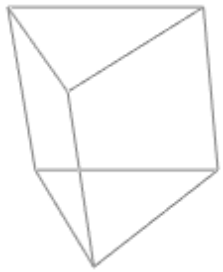
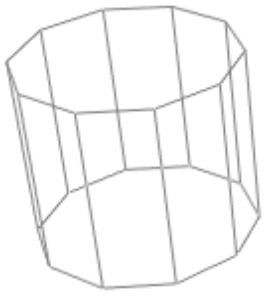
SG_CYLINDER structure

The SG_CYLINDER structure is a presentation of a polygonal cylinder.

Has the following fields

```
double  Raduis;          - cylinder radius
double  Height;          - cylinder height
short   MeridiansCount   - the number of cylinder meridians
```

Illustration:

	MeridiansCount = 3
	MeridiansCount = 10

	MeridiansCount = 24

Defined in sg3D.h

4.10 SG_CONE

SG_CONE structure

The SG_CONE structure is a presentation of a polygonal cone.

Has the following fields

```
double  Raduis1;          - radius of the first cone base
double  Raduis2;          - radius of the second cone base
double  Height;           - cone height
short   MeridiansCount    - the number of cone meridians
```

Illustration:

	MeridiansCount = 3
	MeridiansCount = 10
	MeridiansCount = 24

Defined in sg3D.h

4.11 SG_TORUS

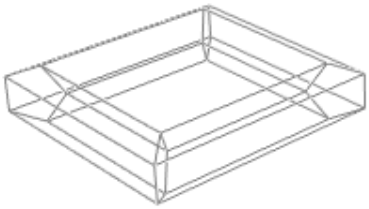
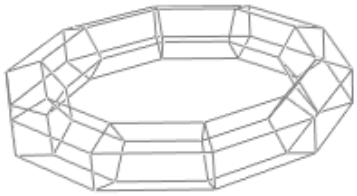
SG_TORUS structure

The SG_TORUS structure is a presentation of a polygonal torus.

Has the following fields

```
double  Raduis1;          - torus radius
double  Raduis2;          - torus thickness
short   MeridiansCount1   - the number of torus meridians
short   MeridiansCount2   - the number of torus cut meridians
```

Illustration:

	MeridiansCount1 = 4 MeridiansCount2 = 5
	MeridiansCount1 = 10 MeridiansCount2 = 5
	MeridiansCount1 = 4 MeridiansCount2 = 5

Defined in sg3D.h

4.12 SG_ELLIPSOID

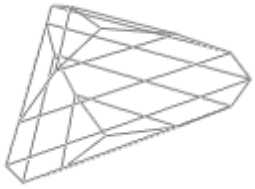
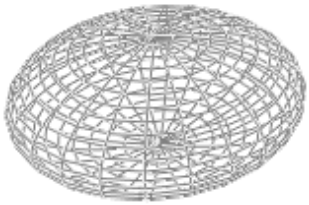
SG_ELLIPSOID structure

The SG_ELLIPSOID structure is a presentation of a polygonal ellipsoid.

Has the following fields

```
double  Raduis1;          - ellipsoid radius on the X axis
double  Raduis2;          - ellipsoid radius on the Y axis
double  Raduis3;          - ellipsoid radius on the Z axis
short   MeridiansCount    - the number of ellipsoid meridians
short   ParallelsCount;   - the number of ellipsoid parallels
```

Illustration:

	MeridiansCount = 3 ParallelsCount = 5
	MeridiansCount = 10 ParallelsCount = 5
	MeridiansCount = 24 ParallelsCount = 24

Defined in sg3D.h

4.13 SG_SPHERIC_BAND

SG_SPHERIC_BAND structure

The SG_SPHERIC_BAND structure is a presentation of a spherical band.

Has the following fields

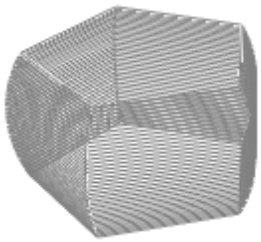
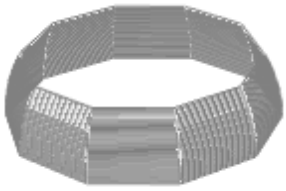
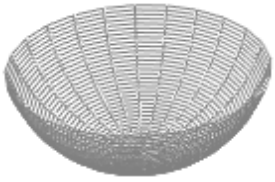
```
double  Radius; -  spherical band radius
double  BeginCoef;  - initial spherical band cut coefficient
double  EndCoef      - final spherical band cut coefficient
short   MeridiansCount - the number of spherical band meridians
```

Explanation:

BeginCoef must be from -1 to 1. If BeginCoef=-1 the spherical band won't be

cut from below
If BeginCoef=1 the spherical band won't be cut from above.
If BeginCoef=0 the spherical band will be cut by an equator of the corresponding sphere.
EndCoef must be from -1 to 1. If EndCoef=-1 the spherical band won't be cut from below
If EndCoef=1 the spherical band won't be cut from above
If EndCoef=0 the spherical band will be cut by an equator of the corresponding sphere.
BeginCoef must be less than EndCoef.

Illustration:

	BeginCoef = -0.5 EndCoef = 0.5 MeridiansCount = 5
	BeginCoef = 0 EndCoef = 0.5 MeridiansCount = 10
	BeginCoef = -1.0 EndCoef = 0 MeridiansCount = 24

Defined in sg3D.h

4.14 Additional types

Additional types are the secondary types that are used in sgCore. The objects of additional types are used as arguments or returned values of the main classes and functions of the sgCore library.

4.14.1 SG_DRAW_LINE_FUNC

Definition.

```
typedef void(*SG_DRAW_LINE_FUNC)(SG_POINT*, SG_POINT*);
```

The pointer to a function called for each line segment set by two points. Used as an argument in geometries drawing functions wihout creating objects.

Defined in sgDefs.h

5 Objects

Objects

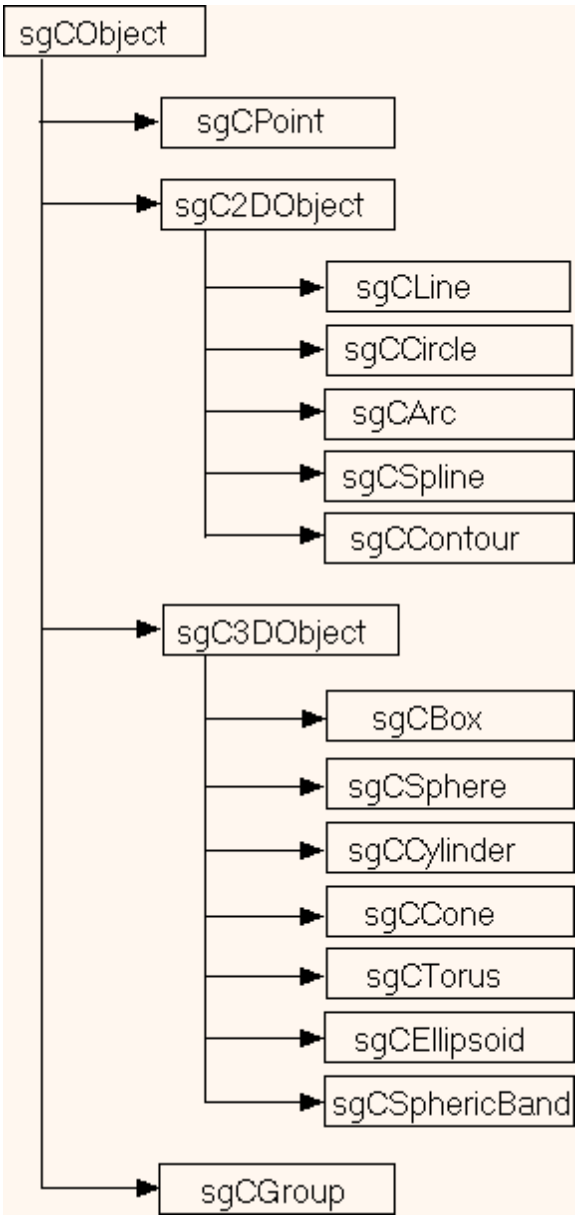
The *Object* concept alongside with Geometry is the main concept of sgCore. When *geometry* is an accurate mathematical description *object* is an approximation for reflecting the possibility on a computer screen. All the operations in sgCore are made with objects not with geometries.

A two-dimensional object (in 3D) is just a number of line segments in sgCore. A three-dimensional object is a number of triangles.

Unlike the *geometry* concept which is all-sufficient the *object* concept is not all-sufficient and directly depends on its basic geometry. We can always get an object with the given accuracy by geometry. That is why each sgCore object stores its own geometry.

The objects are divided into the following classes by a type of the realized geometry:

Objects classes hierarchy



5.1 sgCObject

sgCObject

sgCObject is a base class for all geometrical objects of sgCore. Each object has its attributes (a color number in the palette, lines thickness, lines types, a way to display the object), the status (selected/not selected), name, pointer to the parent object. Each object stores its geometry (system or specified by a user). If the object has a user-defined geometry a corresponding identifier to this geometry is stored in the

object (the GUID string 39 characters long).

One of the main concepts is the *system object type*.

System object type is an internal library category and each geometrical object belongs to one of the following types:

- point
- line segment
- circle
- arc
- spline
- contour
- text
- size
- group
- 3D object

The library user is unable to change the object type.

See also:
Objects hierarchy sgCObject methods

5.1.1 Methods

---

sgCObject methods

Each object of the class inheritated from sgCObject has the following methods:

Clone
GetType
SetUserGeometry
GetUserGeometry
GetUserGeometryID
SetUserDynamicData
GetUserDynamicData
GetAttribute
SetAttribute
GetName
SetName
GetParent
GetGabarits
Select
IsSelect
IsAttachedToScene
InitTempMatrix

DestroyTempMatrix
GetTempMatrix
ApplyTempMatrix
DeleteObject

5.1.1.1 Clone

---

sgCObject* sgCObject::Clone()

Description:
Creates the copy of the object.

Arguments:
No arguments.

Returned value:
The function returns the pointer to a newly created object. If it fails NULL is returned.

Example:

```
sgCLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgObject*  line2 = line1->Clone();
assert(line2->GetType()==SG_OT_LINE);
```

See also:
Objects hierarchy sgCObject methods

5.1.1.2 GetType

SG_OBJECT_TYPE sgCObject::GetType() const

Description:
Returns the system type of the object.

Arguments:
No arguments.

Returned value:

The function returns the system type of the object. *System object type* is an

internal library category and each geometrical object belongs to one of the following types:

- point - the return value is SG_OT_POINT
- line segment - the return value is SG_OT_LINE
- circle - the return value is SG_OT_CICLE
- arc - the return value is SG_OT_ARC
- spline - the return value is SG_OT_SPLINE
- contour - the return value is SG_OT_CONTOUR
- text - the return value is SG_OT_TEXT
- size - the return value is SG_OT_DIM
- group - the return value is SG_OT_POINT
- 3D object - the return value is SG_OT_3D

If an error occurs the function returns SG_OT_BAD_OBJECT

The library user is unable to change the object type.

Define object type

Example:

```
sgLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
assert(line2->GetType()==SG_OT_LINE);
```

See also:
Objects hierarchy sgCObject methods

5.1.1.3 SetUserGeometry

---

bool sgCObject::SetUserGeometry(const char* user_geometry_ID,
const unsigned short user_geometry_size,
const void* user_geometry_data)

Description:

Sets a user-defined geometry to the object.

Arguments:

user_geometry_ID - the identifier of the user-defined geometry, 39 characters string (GUID)
user_geometry_size - size of the user-defined geometry,
user_geometry_data - pointer to the beginning of the block of data containing the user-defined geometry

Returned value:

If succeed the function returns true, otherwise false.

Example:

```
sgCPoint*     pnt = sgCreatePoint(0.0, 0.0, 0.0);
typedef struct
{
    int         first_field;
    char     second_field;
    double  third_field;
}  USER_DATA;

USER_DATA user_data;
user_data.first_field = -1000;
user_data.second_field = 'c';
user_data.third_field = 3.14159265;

pnt->SetUserGeometry("{BE18FFEE-77FB-40f8-B7D1-570D316F696A}",sizeof
(USER_DATA), &user_data);
```

See also:
GetUserGeometry GetUserGeometryID

5.1.1.4 GetUserGeometry

---

const void* sgCObject::GetUserGeometry(unsigned short& user_geometry_size) const

Description:

Returns the pointer to the beginning of the block of data containing the user-defined geometry and its size.

Arguments:

user_geometry_size - return size of the user-defined geometry.

Returned value:

The function returns the pointer to the user-defined geometry. If the user geometry hasn't been defined returns NULL.

See also:
SetUserGeometry GetUserGeometryID

5.1.1.5 GetUserGeometryID

---

const char* sgCObject::GetUserGeometryID() const

Description:

Returns a string containing the user geometry identifier (GUID).

Arguments:

No arguments.

Returned value:

The function returns the user geometry identifier.

See also:
SetUserGeometry GetUserGeometry

5.1.1.6 SetUserDynamicData

---

bool sgCObject::SetUserDynamicData(const SG_USER_DYNAMIC_DATA* u_d_d)

Description:

Assigns a user-defined block of data to an object.

Arguments:

u_d_d - an index to the user-defined block of data. The object of the user data block should have the same class type that the inherited one from SG_USER_DYNAMIC_DATA – the memory release function must be defined.

Returned value:

If succeed the function returns true, otherwise false.

Note:

The user-defined block of data isn't saved in the same file with the object. The user must save it manually.

Example:

```
class MyData : public SG_USER_DYNAMIC_DATA
{
public:
```

```
        int* m_data;
public:
        MyData()
        {
                m_data=new int[10];
        }
        ~MyData()
        {
                delete[] m_data;
        };
        virtual void Finalize()
        {
                if (m_data)
                {
                        delete[] m_data;
                        m_data = NULL;
                        delete this;
                }
        }
    };

sgCPoint* pnt = sgCreatePoint(i-15, j-15, pZ);
sgGetScene()->AttachObject(pnt);

MyData*  usDat = new MyData();
pnt->SetUserDynamicData(usDat);
MyData*  uuu = reinterpret_cast<MyData*>(pnt->GetUserDymanicData());
uuu->m_data[5] = 999;
ASSERT(usDat->m_data[5]==999);
```

See also:
GetUserDynamicData SG_USER_DYNAMIC_DATA

5.1.1.7 GetUserDynamicData

SG_USER_DYNAMIC_DATA* sgCObject::GetUserDynamicData () const

Description:

Returns the pointer to the user block of data specified by the SetUserDynamicData function

Arguments:

No arguments.

Returned value:

Returns the pointer to the user block of data specified by SetUserDynamicData function. If the user data block wasn't assigned, it

returns NULL.

See the example from the SetUserDynamicData description

See also:
SetUserDynamicData SG_USER_DYNAMIC_DATA

5.1.1.8 GetAttribute

---

unsigned short sgCObject::GetAttribute(SG_OBJECT_ATTR_ID attributeId) const

Description:

Returns the value of the corresponding object attribute.

Arguments:

attributeId - object attribute identifier. Can have the following values:
SG_OA_COLOR - a number of the color from the user-defined palette
SG_OA_LINE_TYPE - object lines type - specified by the user as well
SG_OA_LINE_THICKNESS - object lines thickness (wireframe lines thickness for 3D objects)
SG_OA_LAYER - object layer
SG_OA_DRAW_STATE - object draw type. Can posesses the values specified by the following flags:

- SG_DS_FRAME - wireframe model of a 3D object
- SG_DS_HIDE - object is hidden
- SG_DS_GABARITE - gabarit box of the object
- SG_DS_FULL - full drawing of 3D objects.

Returned value:

The value of the corresponding object attribute.

Example:

```
sgCBox*  box    = sgCreateBox(3.0, 5.0, 10.0);
box->SetAttribute(SG_OA_DRAW_STATE, SG_DS_FRAME | SG_DS_GABARITE);
box->SetAttribute(SG_OA_COLOR,100);
assert(box->GetATTRIBUTE(SG_OA_COLOR)==100);
```

See also:
Objects hierarchy sgCObject methods SetAttribute

5.1.1.9 SetAttribute

---

void sgCObject::SetAttribute(SG_OBJECT_ATTR_ID attributeId, unsigned short attributeValue)

Description:

Returns the value of the corresponding object attribute.

Arguments:

attributeId - object attribute identifier. Can have the following values:
SG_OA_COLOR - the number of the color from the user-defined palette
SG_OA_LINE_TYPE - object lines type - specified by the user as well
SG_OA_LINE_THICKNESS - object lines thickness (wireframe lines thickness for 3D objects)
SG_OA_LAYER - object layer
SG_OA_DRAW_STATE - object draw type. Can posesses the values specified by the following flags:

- SG_DS_FRAME - wireframe model of a 3D object
- SG_DS_HIDE - object is hidden
- SG_DS_GABARITE - gabarit box of the object
- SG_DS_FULL - full drawing of 3D objects.
- 12 user flags are described

```
#define SG_DS_USER_1   0x8000
#define SG_DS_USER_2   0x4000
#define SG_DS_USER_3   0x2000
#define SG_DS_USER_4   0x1000
#define SG_DS_USER_5   0x0800
#define SG_DS_USER_6   0x0400
#define SG_DS_USER_7   0x0020
#define SG_DS_USER_8   0x0010
#define SG_DS_USER_9   0x0008
#define SG_DS_USER_10  0x0004
#define SG_DS_USER_11  0x0002
#define SG_DS_USER_12  0x0001
```

attributeValue - value of the corresponding attribute

Returned value:

No values.

Example:

```
sgCBox*  box    = sgCreateBox(3.0, 5.0, 10.0);
box->SetAttribute(SG_OA_DRAW_STATE, SG_DS_FRAME | SG_DS_GABARITE);
box->SetAttribute(SG_OA_COLOR,100);
assert(box->GetATTRIBUTE(SG_OA_COLOR)==100);
```

See also:
Objects hierarchy sgCObject methods GetAttribute

5.1.1.10 GetName

---

const char* sgCObject::GetName() const

Description:
Returns the object name.

Arguments:
No arguments.

Returned value:
The function returns the object name - the string has the length specified in SG_OBJ_NAME_MAX_LEN (in the current library version this constant value is 64).

See also:
Objects hierarchy sgCObject methods SetName

5.1.1.11 SetName

---

bool sgCObject::SetName(const char* object_name)

Description:
Assigns a value to the object.

Arguments:
object_name - new object name (the string with a maximum length specified in SG_OBJ_NAME_MAX_LEN (in the current library version this constant value is 64)).

Returned value:
If the function succeed it returns true, otherwise false.

See also:
Objects hierarchy sgCObject methods GetName

5.1.1.12 GetParent

---

const sgCObject* sgCObject::GetParent() const

Description:
Returns the pointer to a parent object.

Arguments:
No arguments.

Returned value:
Returns the pointer to a contour or a group containing the object. If the object is not contained in the group or contour NULL is returned.

See also:
sgCGroup sgCContour

5.1.1.13 GetGabarits

---

void sgCObject::GetGabarits(SG_POINT& p_min, SG_POINT& p_max)

Description:
Returns the coordinates of the gabarit box of the object.

Arguments:
p_min - takes the value of the gabarit box vertex with minimal coordinates.
p_max - takes the value of the gabarit box vertex with maximum coordinates.

Returned value:
No values.

See also:
Objects hierarchy sgCObject methods

5.1.1.14 Select

---

void sgCObject::Select(bool sel)

Description:

Sets the selection flag of the object to true or false depending on the argument value.

Arguments:

sel - if true the object will be selected, otherwise - deselected.

Returned value:

No values.

Note:

Being selected the object will be added to the list of the scene selected objects. Read more about it at sgCScene::GetSelectedObjectsList

See also:
Objects hierarchy sgCObject methods sgCObject::IsSelect

5.1.1.15 IsSelect

---

bool sgCObject::IsSelect() const

Description:

Returns the object selection flag status.

Arguments:

No arguments.

Returned value:

Returns the object selection flag status.

Example:

```
sgLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
line1->Select();
assert(line1->IsSelect()==true);
```

See also:
Objects hierarchy sgCObject methods sgCObject::Select

5.1.1.16 IsAttachedToScene

---

bool sgCObject::IsAttachedToScene() const

Description:

Returns the flag whether the object is attached to the scene.

Arguments:

No arguments.

Returned value:

Returns the flag whether the object is attached to the scene. Read more about working with scenes at sgCScene

See also:
Objects hierarchy sgCObject methods sgCScene::AttachObject sgCScene::DetachObject

5.1.1.17 InitTempMatrix

---

sgCMatrix* sgCObject::InitTempMatrix()

Description:

Initializes the matrix of affine transformations.

Arguments:

No arguments.

Returned value:

Returns the pointer to the temporary affine transformations matrix.

See also:
sgCMatrix sgCObject::DestroyTempMatrix sgCObject::GetTempMatrix sgCObject::ApplyTempMatrix

5.1.1.18 DestroyTempMatrix

---

bool sgCObject::DestroyTempMatrix()

Description:

Deletes the temporary matrix of affine transformations.

Arguments:

No arguments.

Returned value:

If the matrix was deleted successfully the function returns true, otherwise false.

Example:

```
sgLine*      line = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
line->InitTempMatrix();
line->DestroyTempMatrix();
```

See also:
sgCMatrix sgCObject::InitTempMatrix sgCObject::GetTempMatrix sgCObject::ApplyTempMatrix

5.1.1.19 GetTempMatrix

---

sgCMatrix* sgCObject::GetTempMatrix()

Description:

Returns the pointer to the object affine transformations matrix.

Arguments:

No arguments.

Returned value:

Returns the pointer to the object affine transformations matrix

See also:
sgCMatrix sgCObject::InitTempMatrix sgCObject::DestroyTempMatrix sgCObject::ApplyTempMatrix

5.1.1.20 ApplyTempMatrix

---

bool sgCObject::ApplyTempMatrix()

Description:

Applies the affine transformations matrix to the object.

Arguments:

No arguments.

Returned value:

Applies to the object the affine transformations matrix created using the InitTempMatrix function.

Example:

```
sgBox*      box = sgCreateBox(10.0, 20.0, 30.0);
SG_VECTOR trVec = {2.0, 0.0, -5,0};
box->InitTempMatrix()->Translate(trVec);
box->ApplyTempMatrix();
box->DestroyTempMatrix();
```

See also:
sgCMatrix sgCObject::InitTempMatrix sgCObject::DestroyTempMatrix sgCObject::GetTempMatrix sgCObject::ApplyTempMatrix

5.1.1.21 DeleteObject

---

static void sgCObject::DeleteObject(sgCObject* obj)

Description:

Deletes the object.

Arguments:

obj - the object to be deleted.

Returned value:

No values.

Note:

You MUST delete the object when the work with it is over and if the object wasn't added to the scene.
If the object was added to the scene you CAN'T delete it.
If the object was added to the scene and then deleted, and these actions were saved in the Undo-Redo history, the object CAN'T be deleted.
If the object was added to the scene and then deleted, and these actions

were not saved in the Undo-Redo history, the object MUST be deleted.

See also:
Objects hierarchy sgCObject methods sgCScene::AttachObject sgCScene::DetachObject
sgCScene::StartUndoGroup sgCScene::EndUndoGroup

5.1.2 sgCPoint

---

sgCPoint

sgCPoint is the representative of the POINT concept in the three-dimensional space. Any point can be set by three coordinates.

The sgCPoint class methods:

Create
GetGeometry

5.1.2.1 Create

---

static sgCPoint* sgCPoint::Create(double pX, double pY, double pZ)

Description:
Creates the object of the POINT class.

Arguments:
pX - X coordinate of the created point,
pY - Y coordinate of the created point,
pZ - Z coordinate of the created point,

Returned value:
Returns the pointer to the created object.

Following shortening is set:

```
#define    sgCreatePoint    sgCPoint::Create
```

Example:

```
sgCPoint*      pnt = sgCreatePoint(0.0, 1.0, 5.0);
```

See also:
Objects hierarchy sgCObject methods SG_POINT

5.1.2.2 GetGeometry

---

const SG_POINT* sgCPoint::GetGeometry()

Description:

Returns the pointer to the POINT object geometry.

Arguments:

No arguments.

Returned value:

Returns the pointer to the POINT object geometry. Read more about the point geometry at SG_POINT

Example:

```
sgCPoint*      pnt = sgCreatePoint(10.0, 15.0, 40.0);
assert(pnt->GetGeometry()->x==10.0);
assert(pnt->GetGeometry()->y==15.0);
assert(pnt->GetGeometry()->z==40.0);
```

See also:
Objects hierarchy sgCObject methods SG_POINT sgCPoint::Create

5.1.3 sgC2DObject

---

sgC2DObject

The sgC2DObject class is a base class for all the objects which can be approximated by an array of line segments. Strictly speaking objects of this class can not be flat (in spite of the class name - sgC**2D**Object), for example, splines or contours. This class is intended to differ these objects from the ones which can be approximated by a triangles array (sgC3DObject).
Such algorythms of building complex 3D objects accept the objects of this class.

All the objects inherited from sgC2DObject support the following methods:

IsClosed
IsPlane
IsLinear
IsSelfIntersecting
GetEquidistantContour
GetOrient
ChangeOrient
GetPointFromCoefficient

IsObjectsOnOnePlane
IsObjectsIntersecting
IsFirstObjectInsideSecondObject

5.1.3.1 Methods

---

sgC2DObject methods

Any object of the class inherited from sgC2DObject supports the following methods:

IsClosed
IsPlane
IsLinear
IsSelfIntersecting
GetEquidistantContour
GetOrient
ChangeOrient
GetPointFromCoefficient
IsObjectsOnOnePlane
IsObjectsIntersecting
IsFirstObjectInsideSecondObject

5.1.3.1.1 IsClosed

---

virtual bool sgC2DObject::IsClosed() const

Description:
This function is pure virtual. It checks whether the object is closed. It is useful for splines and contours.

Arguments:
No arguments.

Returned value:
true if the object is closed, fasle otherwise

5.1.3.1.2 IsPlane

---

virtual bool sgC2DObject::IsPlane(SG_VECTOR*,double*) const

Description:

This function is pure virtual. It checks whether the object is flat or not. It is useful for splines and contours. If the object lies in the same plane the arguments (if not NULL) has the values of the normal to the plane and the free coefficient in the equation of plane.

Arguments:

No arguments.

Returned value:

true if the object is flat, fasle otherwise

5.1.3.1.3 IsLinear

virtual bool sgC2DObject::IsLinear() const

Description:

This function is pure virtual. It checks whether all the points of the object lie on the same object line. It is useful for splines and contours.

Arguments:

No arguments.

Returned value:

true if all the points of the object lie on the same line, fasle otherwise

5.1.3.1.4 IsSelfIntersecting

virtual bool sgC2DObject::IsSelfIntersecting() const

Description:

This function is pure virtual. It checks whether the object is self-intersecting. It is useful for splines and contours.

Arguments:

No arguments.

Returned value:
true if the object is self-intersecting, fasle otherwise

5.1.3.1.5 GetEquidistantContour

---

sgCContour* sgC2DObject::GetEquidistantContour(double h1, double h2, bool toRound)

Description:
Builds an equidistant curve to a flat nonlinear object.

Arguments:
h1 - equidistant curve shift from the start object point
h2 - equidistant curve shift from the end object point
toRound - flag for rounding sharp end points of the equidistant to be build (ignored in the current library version).

Returned value:
Returns the pointer to the created contour or NULL otherwise

5.1.3.1.6 GetOrient

SG_2D_OBJECT_ORIENT sgC2DObject::GetOrient(const SG_VECTOR& planeNormal) const

Description:
Returns the object circumvention direction depending the normal orientation.

Arguments:
planeNormal - normal to the object plane.

Returned value:
Returns the SG_2D_OBJECT_ORIENT list.

See also:
ChaneOrient GetPointFromCoefficient

5.1.3.1.7 ChangeOrient

bool sgC2DObject::ChangeOrient()

Description:

Reverses the object circumvention direction. The GetOrient function (with the same argument) returns an opposite value.

Arguments:

No arguments.

Returned value:

fasle if the GetOrient function returns the error orientation code and true otherwise.

See also:
GetOrient GetPointFromCoefficient

5.1.3.1.8 GetPointFromCoefficient

SG_POINT sgC2DObject::GetPointFromCoefficient(double coeff) const

Description:

Calculates the object point depending on the coefficient. Any 2D object has the starting and the end points (for example, start and end point for an arc; the same point for a circle). These points are the same only for the closed objects (the IsClosed() function will return true).
Let's consider that the start point is equal to 0, the end point - to 1. So, we can set a correspondence between any point of the object and a number from 0 to 1. This relation will be the one-to-one correspondence.

This function returns an object point depending on the coefficient. Mind, that after calling the ChangeOrient function the circumvention direction will change to the opposite and the start point will be correspond to 1, the end point - to 0. Thus the points corresponding coefficients will be changed according to the formula:

new=1-old.

Arguments:

coeff - coeffitient from 0 to 1 to find a point on the object.

Returned value:

Returns an object point by its argument.

See also:
GetOrient ChangeOrient SG_POINT

5.1.3.1.9 IsObjectsOnOnePlane

---

static bool sgC2DObject::IsObjectsOnOnePlane(const sgC2DObject& obj1, const sgC2DObject& obj2) const

Description:

This static function checks whether two flat nonlinear objects lie on the same plane.

Arguments:

obj1 - the first object,
obj2 - the second object.

Returned value:

true if the objects lie on the same plane and fasle otherwise

See also:
IsPlane IsLinear

5.1.3.1.10 IsObjectsIntersecting

---

static bool sgC2DObject::IsObjectsIntersecting(const sgC2DObject& obj1, const sgC2DObject& obj2) const

Description:

This static function checks whether two flat nonlinear objects liyng on the same plane are intersecting.

Arguments:

obj1 - the first object,
obj2 - the second object.

Returned value:

true if the objects intersect and fasle otherwise.

See also:
IsPlane IsLinear IsObjectsOnOnePlane

5.1.3.1.11 IsFirstObjectInsideSecondObject

---

static bool sgC2DObject::IsFirstObjectsInsideSecondObject (const sgC2DObject& obj1, const sgC2DObject& obj2) const

Description:

This static function checks whether the flat object lies inside the other closed flat object. The objects must lie on the same plane.

Arguments:

obj1 - the first object,
obj2 - the second object.

Returned value:

true if obj1 lies inside the obj2 and fasle in the following cases:
the objects are not flat, the second object is not closed, the objects don't lie on the same plane or the objects intersect.

See also:
IsPlane IsLinear IsObjectsOneOnePlane IsObjectsIntersecting

5.1.3.2 sgCLine

---

sgCLine

sgCLine is the representative of the LINE SEGMENT concept in the three-dimensional space. Any line segment can be set by three coordinates for each end point.

The sgCLine class methods:

5.1.3.2.1 Create

---

static sgCLine* sgCLine::Create(double pX1, double pY1, double pZ1,double pX2, double pY2, double pZ2)

Description:

Creates an object of the LINE SEGMENT class.

Arguments:

pX1 - X coordinate of the first point,
pY1 - Y coordinate of the first point,
pZ1 - Z coordinate of the first point,
pX2 - X coordinate of the second point,
pY2 - Y coordinate of the second point,
pZ2 - Z coordinate of the second point.

Returned value:

Returns the pointer to the created object. If the points are the same, the function returns NULL.

Following shortening is set:

```
#define  sgCreateLine    sgCLine::Create
```

Example:

```
sgCLine*      ln = sgCreateLine(0.0, 1.0, 6.0, 10.0, 34.0, 1.0);
```

See also:

Objects hierarchy sgCObject methods SG_LINE

5.1.3.2.2 GetGeometry

---

const SG_LINE* sgCLine::GetGeometry()

Description:

Returns the pointer to the LINE SEGMENT object geometry.

Arguments:

No arguments.

Returned value:

Returns the pointer to the LINE SEGMENT object geometry. Read more about the line geometry at SG_LINE

Example:

```
sgCLine*    ln = sgCreateLine(10.0, 15.0, 80.0, 0.0, 8.0, 6.0);
assert(ln->GetGeometry()->p1.x==10.0);
assert(ln->GetGeometry()->p1.y==15.0);
assert(ln->GetGeometry()->p2.z==6.0);
```

See also:
Objects hierarchy sgCObject methods SG_LINE sgCLine::Create

5.1.3.2.3 IsClosed

---

bool sgCLine::IsClosed() const

Description:

Always returns false.

Arguments:

No arguments.

Returned value:

Always returns false.

See also:
SG_LINE sgCLine::Create

5.1.3.2.4 IsPlane

bool sgCLine::IsPlane(SG_VECTOR*,double*) const

Description:
Always returns false.

Arguments:
Not used.

Returned value:
Always returns false.

Note:
A line segment is actually a flat object but it doesn't lie on the unlimited number of planes and it is impossible to deconceptine the plane equation. Let's consider that a line segment is not a flat object.

See also:
SG_LINE sgCLine::Create

5.1.3.2.5 IsLinear

bool sgCLine::IsLinear() const

Description:
Always returns true.

Arguments:
No arguments.

Returned value:
Always returns true.

See also:
SG_LINE sgCLine::Create

5.1.3.2.6 IsSelfIntersecting

bool sgCLine::IsSelfIntersecting() const

Description:
Always returns false.

Arguments:
No arguments.

Returned value:
Always returns fasle

See also:
SG_LINE sgCLine::Create

5.1.3.3 sgCCircle

sgCCircle

sgCCircle is the representative of the CIRCLE concept in the three-dimensional space. Any circle can be set by its geometry.

The sgCCircle class methods:

Create
GetGeometry
IsClosed
IsPlane
IsLinear
IsSelfIntersecting

5.1.3.3.1 Create

static sgCCircle* sgCCircle::Create(const SG_CIRCLE& cirGeom)

Description:

Creates an object of the CIRCLE class.

Arguments:

cirGeom - the circle geometry. Read more about the circle geometry at SG_CIRCLE

Returned value:

Returns the pointer to the created object. If the points are the same, the function returns NULL.

Following shortening is set:

```
#define  sgCreateCircle    sgCCircle::Create
```

Example:

```
SG_POINT    cirCenter = {1.0, 0.0, 1.0};
SG_VECTOR   cirNorm = {0.0, 0.0, 1.0};
SG_CIRCLE   cirGeom;
cirGeom.FromCenterRadiusNormal(cirCenter, 5.0, cirNorm);
sgCCircle*   crcl = sgCreateCircle(cirGeom);
```

See also:
Objects hierarchy sgCObject methods SG_CIRCLE SG_POINT SG_VECTOR

5.1.3.3.2 GetGeometry

---

const SG_CIRCLE* sgCCircle::GetGeometry() const

Description:

Returns the pointer to the CIRCLE object geometry.

Arguments:

No arguments.

Returned value:

Returns the pointer to the CIRCLE object geometry. Read more about the circle geometry at SG_CIRCLE

Example:

```
SG_POINT    cirCenter = {1.0, 0.0, 1.0};
SG_VECTOR   cirNorm = {0.0, 0.0, 1.0};
SG_CIRCLE   cirGeom;
cirGeom.FromCenterRadiusNormal(cirCenter, 5.0, cirNorm);
sgCCircle*   crcl = sgCreateCircle(cirGeom);
```

```
assert(crcl->GetGeometry()->radius==5.0);
```

See also:
Objects hierarchy sgCObject methods SG_CIRCLE SG_POINT SG_VECTOR

5.1.3.3.3 GetPointsCount

---

int sgCCircle::GetPointsCount() const

Description:

Returns the number of points in the circle segment presentation. The array of the points itself is returned by the GetPoints function

Arguments:

No arguments.

Returned value:

Returns the number of points in the circle segment presentation.

See also:
sgCCircle::Create sgCCircle::GetGeometry sgCCircle::GetPoints

5.1.3.3.4 GetPoints

---

const SG_POINT* sgCCircle::GetPoints() const

Description:

Returns the pointer to the array of the circle segment presentation. The number of the points in this array is returned by the GetPointsCount function

Arguments:

No arguments.

Returned value:

Returns the pointer to the array of the circle segment presentation.

See also:
sgCCircle::Create sgCCircle::GetGeometry sgCCircle::GetPointsCount

5.1.3.3.5 IsClosed

bool sgCCircle::IsClosed() const

Description:
Always returns true.

Arguments:
No arguments.

Returned value:
Always returns true.

See also:
SG_CIRCLE sgCCircle::Create

5.1.3.3.6 IsPlane

bool sgCCircle::IsPlane(SG_VECTOR* planeNorm, double* planeD) const

Description:
Always returns true.

Arguments:
planeNorm - if not NULL, it has the value of the normal to the circle plane, ignored otherwise.

planeD - if not NULL, it has the value of the free coefficient in the plane equation, ignored otherwise.

Returned value:
Always returns true. The arguments have the values of the normal to the circle plane and of the free coefficient in the plane equation (if the arguments are not NULL).

See also:
SG_CIRCLE sgCCircle::Create

5.1.3.3.7 IsLinear

bool sgCCircle::IsLinear() const

Description:
Always returns false.

Arguments:
No arguments.

Returned value:
Always returns false

See also:
SG_CIRCLE sgCCircle::Create

5.1.3.3.8 IsSelfIntersecting

bool sgCCircle::IsSelfIntersecting() const

Description:
Always returns false.

Arguments:
No arguments.

Returned value:
Always returns false

See also:
SG_CIRCLE sgCCircle::Create

5.1.3.4 sgCArc

sgCArc

sgCArc is the representative of the ARC concept in the three-dimensional space. Any arc can be set by its geometry.

The sgCArc class methods:

5.1.3.4.1 Create

static sgCArc* sgCArc::Create(const SG_ARC& arcGeom)

Description:

Creates an object of the ARC class.

Arguments:

arcGeom - the arc geometry. Read more about the arc geometry at SG_ARC

Returned value:

Returns the pointer to the created object. Otherwise NULL.

Following shortening is set:

```
#define  sgCreateArc    sgCArc::Create
```

Example:

```
SG_POINT    arcP1 = {0.0, 0.0, 0.0};
SG_POINT    arcP2 = {1.0, 0.0, 0.0};
SG_POINT    arcP3 = {0.0, 0.0, 1.0};
SG_ARC      arcGeom;
arcGeom.FromThreePoints(arcP1,arcP2,arcP3,false);
sgCArc*     arc = sgCreateArc(arcGeom);
```

See also:

Objects hierarchy sgCObject methods SG_ARC SG_POINT

5.1.3.4.2 GetGeometry

const SG_ARC* sgCArc::GetGeometry() const

Description:

Returns the pointer to the ARC object geometry.

Arguments:

No arguments.

Returned value:

Returns the pointer to the ARC object geometry. Read more about the ARC geometry at SG_ARC

See also:
Objects hierarchy sgCObject methods SG_ARC

5.1.3.4.3 GetPointsCount

---

int sgCArc::GetPointsCount() const

Description:

Returns the number of points in the arc segment presentation. The array of the points itself is returned by the GetPoints function

Arguments:

No arguments.

Returned value:

Returns the number of points in the arc segment presentation.

See also:
sgCArc::Create sgCArc::GetGeometry sgCArc::GetPoints

5.1.3.4.4 GetPoints

---

const SG_POINT* sgCArc::GetPoints() const

Description:

Returns the pointer to the array of the arc segment presentation. The number of the points in this array is returned by the GetPointsCount function

Arguments:

No arguments.

Returned value:
Returns the pointer to the array of the arc segment presentation.

See also:
sgCArc::Create sgCArc::GetGeometry sgCArc::GetPointsCount

5.1.3.4.5 IsClosed

---

bool sgCArc::IsClosed() const

Description:
Always returns false.

Arguments:
No arguments.

Returned value:
Always returns false

See also:
SG_ARC sgCArc::Create

5.1.3.4.6 IsPlane

---

bool sgCArc::IsPlane(SG_VECTOR* planeNorm, double* planeD) const

Description:
Always returns true.

Arguments:
planeNorm - if not NULL, it has the value of the normal to the arc plane, ignored otherwise.
planeD - if not NULL, it has the value of the free coefficient in the plane equation, ignored otherwise.

Returned value:

Always returns true. The arguments have the values of the normal to the arc plane and of the free coefficient in the plane equation (if the arguments are not NULL).

See also:
SG_ARC sgCArc::Create

5.1.3.4.7 IsLinear

---

bool sgCArc::IsLinear() const

Description:
Always returns false.

Arguments:
No arguments.

Returned value:
Always returns false

See also:
SG_ARC sgCArc::Create

5.1.3.4.8 IsSelfIntersecting

---

bool sgCArc::IsSelfIntersecting() const

Description:
Always returns false.

Arguments:
No arguments.

Returned value:
Always returns false

See also:
SG_ARC sgCArc::Create

5.1.3.5 sgCSpline

sgCSpline

sgCSpline is the representative of the SPLINE concept in the three-dimensional space. Any spline can be set by its geometry.

The sgCSpline class methods:

Create
GetGeometry
IsClosed
IsPlane
IsLinear
IsSelfIntersecting

5.1.3.5.1 Create

static sgCSpline* sgCSpline::Create(const SG_SPLINE& splGeo)

Description:
Creates an object of the SPLINE class.

Arguments:
splGeom - the spline geometry. Read more about the spline geometry at SG_SPLINE

Returned value:
Returns the pointer to the created object. Otherwise NULL.

Following shortening is set:

```
#define  sgCreateSpline    sgCSpline::Create
```

Note:
If the geometry isn't used after creating the spline you must delete it using the Delete function

Example:

```
SG_POINT tmpPnt;

SG_SPLINE* splGeo = SG_SPLINE::Create();

tmpPnt.x = 1.0; tmpPnt.y = -3.0; tmpPnt.z = 0.0;
splGeo->AddKnot(tmpPnt,0);
tmpPnt.x = 3.0; tmpPnt.y = -2.0; tmpPnt.z = 0.0;
splGeo->AddKnot(tmpPnt,1);
tmpPnt.x = 2.0; tmpPnt.y = -1.0; tmpPnt.z = 0.0;
splGeo->AddKnot(tmpPnt,2);
tmpPnt.x = 3.0; tmpPnt.y = 1.0; tmpPnt.z = 0.0;
splGeo->AddKnot(tmpPnt,3);
tmpPnt.x = 2.0; tmpPnt.y = 4.0; tmpPnt.z = 0.0;
splGeo->AddKnot(tmpPnt,4);
tmpPnt.x =4.0; tmpPnt.y = 5.0; tmpPnt.z = 0.0;
splGeo->AddKnot(tmpPnt,5);

sgCSpline* spl_obj = sgCreateSpline(*splGeo);

SG_SPLINE::Delete(splGeo);
```

See also:
Objects hierarchy sgCObject methods SG_SPLINE SG_POINT

5.1.3.5.2 GetGeometry

---

const SG_SPLINE* sgCSpline::GetGeometry() const

Description:

Returns the pointer to the SPLINE object geometry.

Arguments:

No arguments.

Returned value:

Returns the pointer to the SPLINE object geometry. Read more about the SPLINE geometry at SG_SPLINE

See also:
Objects hierarchy sgCObject methods SG_SPLINE

5.1.3.5.3 IsClosed

bool sgCSpline::IsClosed() const

Description:

Returns true if the spline is closed and false otherwise.

Arguments:

No arguments.

Returned value:

Returns true if the spline is closed and false otherwise

See also:
SG_SPLINE sgCSpline::Create

5.1.3.5.4 IsPlane

bool sgCSpline::IsPlane(SG_VECTOR* planeNorm, double* planeD) const

Description:

Returns true if the spline is flat and false otherwise.

Arguments:

planeNorm - if not NULL and the spline is flat, it has the value of the normal to the spline plane, ignored otherwise.
planeD - if not NULL and the spline is flat, it has the value of the free coefficient in the plane equation, ignored otherwise.

Returned value:

Returns true if the spline is flat and false otherwise. The arguments have the values of the normal to the spline plane and of the free coefficient in the plane equation (if the arguments are not NULL).

See also:
SG_SPLINE sgCSpline::Create

5.1.3.5.5 IsLinear

bool sgCSpline::IsLinear() const

Description:
Returns true if the spline lies on the same line and false otherwise.

Arguments:
No arguments.

Returned value:
Returns true if the spline lies on the same line and false otherwise

See also:
SG_SPLINE sgCSpline::Create

5.1.3.5.6 IsSelfIntersecting

bool sgCSpline::IsSelfIntersecting() const

Description:
Returns true if the spline is self-intersecting and false otherwise.

Arguments:
No arguments.

Returned value:
Returns true if the spline is self-intersecting and false otherwise

See also:
SG_SPLINE sgCSpline::Create

5.1.3.6 sgCContour

sgCContour

sgCContour is the representative of the CONTOUR concept in the three-

dimensional space. In the sgCore library CONTOUR is a number of lines, arcs and other contours located in
A contour can be closed and not closed, with self-intersections and without. The contour can be build up from arcs, lines and other contours and also can be taken to component parts. The GetParent function will return the pointer to this contour for each object which is a contour component part.

Any spline can be set by its geometry.

The sgCContour class methods:

CreateContour
BreakContour
GetChildrenList
IsClosed
IsPlane
IsLinear
IsSelfIntersecting

5.1.3.6.1 CreateContour

---

static sgCContour* sgCContour::CreateContour(sgCObject** objcts, int cnt)

Description:

Creates a contour by an array of line segments, arcs or other contours. *The objects must topologically followeach other, i.e. the beginning of the next object must coincide with the end of the previous one.*

Arguments:

objcts - array of the pointer to line segments, arcs or contours.
cnt - the number of objects in this array.

Returned value:

Returns the pointer to the created contour. Otherwise NULL.

Example (the contour of lines and another contour):

```
sgCObject*   objcts[4];
objcts[0] = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
objcts[1] = sgCreateLine(2.0, 3.0, 5.0, 10.0, 0.0, 3.0);
```

```
objcts[2] = sgCreateLine(10.0, 0.0, 3.0, 6.0, 13.0, 15.0);
objcts[0] = sgCContour::CreateContour(&objcts[0],3);
objcts[1] = sgCreateLine(6.0, 13.0, 15.0, 20.0, 30.0, 75.0);
objcts[2] = sgCreateLine(20.0, 30.0, 75.0, 10.0, 6.0, -3.0);
objcts[3] = sgCreateLine(10.0, 6.0, -3.0, 0.0, 0.0, 0.0);
sgCContour* resCont = sgCContour::CreateContour(&objcts[0],4);
```

See also:
Objects hierarchy sgCObject methods

5.1.3.6.2 BreakContour

---

bool sgCContour::BreakContour(sgCObject** objcts)

Description:

Takes the contour to component parts.

Arguments:

objcts - buffer to put the contour component parts. *You should allocate the memory for an array of objects pointers beforehand. The number of the elements in this array must be equal to the contour child objects (returned by GetChildrenList->GetCount()).*

Returned value:

Returns false if the function fails, otherwise true. The function argument is filled with the pointers to child objects.

Example:

```
/*Build contour*/
sgCObject*   objcts[4];
objcts[0] = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
objcts[1] = sgCreateLine(2.0, 3.0, 5.0, 10.0, 0.0, 3.0);
objcts[2] = sgCreateLine(10.0, 0.0, 3.0, 6.0, 13.0, 15.0);
objcts[0] = sgCContour::CreateContour(&objcts[0],3);
objcts[1] = sgCreateLine(6.0, 13.0, 15.0, 20.0, 30.0, 75.0);
objcts[2] = sgCreateLine(20.0, 30.0, 75.0, 10.0, 6.0, -3.0);
objcts[3] = sgCreateLine(10.0, 6.0, -3.0, 0.0, 0.0, 0.0);
sgCContour* resCont = sgCContour::CreateContour(&objcts[0],4);

   /*Break contour*/
const int ChildsCount = resCont->GetChildrenList()->GetCount();

  sgCObject**  allChilds = (sgCObject**)malloc(ChildsCount*sizeof
(sgCObject*));
  if (!resCont->BreakContour(allChilds))
  {
    assert(0);
  }
  const int sz = resCont->GetChildrenList()->GetCount();
  assert(sz==0);
```

```
sgDeleteObject(resCont);
for (int i=0;i<ChCnt;i++)
{
     sgGetScene()->AttachObject(allChilds[i]);
}
free(allChilds);
```

See also:
CreateContour

5.1.3.6.3 GetChildrenList

IObjectsList* sgCContour::GetChildrenList() const

Description:

Returns the pointer to the list of the child objects.

Arguments:

No arguments.

Returned value:

Returns the pointer to the list of the contour child objects.
More - IObjectsList

See also:
CreateContour BreakContour

5.1.3.6.4 IsClosed

---

bool sgCContour::IsClosed() const

Description:

Returns true if the contour is closed and false otherwise.

Arguments:

No arguments.

Returned value:

Returns true if the contour is closed and false otherwise

5.1.3.6.5 IsPlane

bool sgCContour::IsPlane(SG_VECTOR* planeNorm, double* planeD) const

Description:

Returns true if the contour is flat and false otherwise.

Arguments:

planeNorm - if not NULL and the contour is flat, has the value of the normal to the contour plane, ignored otherwise.
planeD - if not NULL and the contour is flat, has the value of the free coefficient in the plane equation, ignored otherwise.

Returned value:

Returns true if the contour is flat and false otherwise. The arguments have the values of the normal to the contour plane and of the free coefficient in the plane equation (if the arguments are not NULL).

5.1.3.6.6 IsLinear

bool sgCContour::IsLinear() const

Description:

Returns true if the contour lies on the same line and false otherwise.

Arguments:

No arguments.

Returned value:

Returns true if the contour lies on the same line and false otherwise

5.1.3.6.7 IsSelfIntersecting

---

bool sgCContour::IsSelfIntersecting() const

Description:
Returns true if the contour is self-intersecting and false otherwise.

Arguments:
No arguments.

Returned value:
Returns true if the contour is self-intersecting and false otherwise

5.1.3.7 Additional types

Additional types are the secondary types that are used in sgCore. The objects of additional types are used as arguments or returned values of the main classes and functions of the sgCore library.

5.1.3.7.1 SG_2D_OBJECT_ORIENT

Orientation type of the sgC2DObject class object depending on the normal

```
typedef enum
{
    OO_ERROR=0,            // error
    OO_CLOCKWISE,          // clockwise
    OO_ANTICLOCKWISE    // counterclockwise
} SG_2D_OBJECT_ORIENT;
```

5.1.4 sgC3DObject

---

sgC3DObject

The sgC3DObject class is a base class for all the objects which can be approximated by an array of triangles.

All the objects inherited from sgC3DObject support the following methods:
Get3DObjectType
AutoTriangulate

Triangulate
GetTriangles
GetWorldMatrixData
SetMaterial
GetMaterial
GetVolume
GetSquare

5.1.4.1 Methods

---

sgC3DObject methods

Any object of the class inherited from sgC3DObject support the following methods:

Get3DObjectType
AutoTriangulate
Triangulate
GetTriangles
GetWorldMatrixData
SetMaterial
GetMaterial
GetVolume
GetSquare

5.1.4.1.1 Get3DObjectType

SG_3DOBJECT_TYPE sgC3DObject::Get3DObjectType() const

Description:
Returns the 3D object type

Arguments:
No arguments.

Returned value:
3D object type. Can take values -
SG_UNKNOWN_3D (error code),
SG_BODY (solid) - if the object surface is limited by a closed volume ,
SG_SURFACE (surface)

See also:
Objects hierarchy sgCObject methods

5.1.4.1.2 AutoTriangulate

---

static void sgC3DObject::AutoTriangulate(bool isTriang, SG_TRIANGULATION_TYPE TrianType);

Description:

Sets the global flag and the automatic triangulation type at 3D objects creation.

Arguments:

isTriang - automatic triangulation flag of 3D objects. If the value is true, any 3D object will be triangulated after its creation. If the flag value is false, then the objects will not be triangulated. You need to call the sgC3DObject::Triangulate method for their triangulation.

TrianType - object triangulation type with the enabled automatic triangulation type. If you want to learn more about the triangulation types, see the SG_TRIANGULATION_TYPE description. If isTriang is false, then TrianType is ignored.

Returned value:

No values.

Important:

If you need to create a complex object in a few steps, it's recommended that you turn off the automatic triangulation flag – it will improve the algorithm speed.

To triangulate the final object, use sgC3DObject::Triangulate or turn on the automatic triangulation flag on the final step of the object creation.

See also:
Triangulate SG_TRIANGULATION_TYPE

5.1.4.1.3 Triangulate

bool sgC3DObject::Triangulate(SG_TRIANGULATION_TYPE TrianType);

Description:

Triangulates a 3D object if it hasn't been triangulated yet. If the GetTriangles function returns a value that is not equal to NULL the object has already been triangulated.

Arguments:

TrianType – object triangulation type. To learn more about the triangulation types, see the SG_TRIANGULATION_TYPE description.

Returned value:

If succeed the function returns true, otherwise false.

See also:

AutoTriangulate SG_TRIANGULATION_TYPE GetTriangles

5.1.4.1.4 GetTriangles

const SG_ALL_TRIANGLES* sgC3DObject::GetTriangles() const

Description:

Returns the pointer to the structure of a 3D object triangles description.

Arguments:

No arguments.

Returned value:

Returns the pointer to the structure of a 3D object triangles description.

The description of this structure:

```
typedef struct
{
        int             nTr;
        SG_POINT*       allVertex;
        SG_VECTOR*      allNormals;
        double*         allUV;
} SG_ALL_TRIANGLES;
```

nTr - the number of triangles describing a 3D object

allVertex - array of point triples each is a triangle vertex
allNormals - array of normals in the corresponding points of the allVertex array
allUV - array of the U and V texture coordinates in every triangle vertex.

If the object is not triangulated, NULL is returned. To triangulate the object, you need to use the sgC3DObject::Triangulate function

See also:
SG_POINT Triangulate SG_TRIANGULATION_TYPE

5.1.4.1.5 GetWorldMatrixData

---

const double* sgC3DObject::GetWorldMatrixData() const

Description:
Returns the pointer to the beginning of the 4x4 matrix of the object position and deformation in the three-dimensional space.

Arguments:
No arguments.

Returned value:
Returns the pointer to the beginning of the 4x4 matrix of the object position and deformation in the three-dimensional space. The matrix can be created using the ApplyTempMatrix function.

See also:
sgCMatrix InitTempMatrix GetTempMatrix ApplyTempMatrix

5.1.4.1.6 SetMaterial

---

void sgC3DObject::SetMaterial(const SG_MATERIAL& newMat) const

Description:
Sets a new material to the object.

Arguments:

newMat - material for the object - is an SG_MATERIAL structure object.

The description of this structure:

```
typedef  struct
{
        int                         MaterialIndex;
        double                      TextureScaleU;
        double                      TextureScaleV;
        double                      TextureShiftU;
        double                      TextureShiftV;
        double                      TextureAngle;
        bool                        TextureSmooth;
        bool                        TextureMult;
        SG_MIX_COLOR_TYPE     MixColorType;
        SG_UV_TYPE            TextureUVType;
} SG_MATERIAL;
```

MaterialIndex - the number of the material in the library defined by the user.
TextureScaleU - texture coefficient of stretch along the U coordinate. Is considered when calculating texture coordinates for each triangle.
TextureScaleV - texture coefficient of stretch along the V coordinate. Is considered when calculating texture coordinates for each triangle.
TextureShiftU - texture shift along the U coordinate. Is considered when calculating texture coordinates for each triangle.
TextureShiftV - texture shift along the V coordinate. Is considered when calculating texture coordinates for each triangle.
TextureAngle - texture rotation angle when overlapping on the object. Is considered when calculating texture coordinates for each triangle.
TextureSmooth - whether to smooth the texture when displaying. This flag is necessary only for saving the object properties from the application using sgCore.
TextureMult - whether to multiply the texture when overlapping on the object. This flag is necessary only for saving the object properties from the application using sgCore.
MixColorType - color mixing type of the material and of the object. Can have the following values: SG_MODULATE_MIX_TYPE (modulate the colors), SG_BLEND_MIX_TYPE (blend the colors) or SG_REPLACE_MIX_TYPE (replace the object color with the color of the material or the texture). This structure field is necessary only for saving the object properties from the application using sgCore.
TextureUVType - type of overlapping texture on the object. Can have the following values: SG_CUBE_UV_TYPE (texture overlapping along the axes), SG_SPHERIC_UV_TYPE (spherical type of the texture overlapping) or SG_CYLINDER_UV_TYPE (cylindrical type of the texture overlapping). This structure field is necessary to calculate texture coordinates for each object triangle.
This function changes the texture coordinates array for each object triangle -

allUV (read the SG_ALL_TRIANGLES structure description at GetTriangles ())

Returned value:
No values.

See also:
GetTriangles GetMaterial

5.1.4.1.7 GetMaterial

---

const SG_MATERIAL* sgC3DObject::GetMaterail()

Description:
Returns the pointer to the structure of the object material description if a material was set on the object.

Arguments:
No arguments.

Returned value:
Returns the pointer to the structure of the object material description or NULL if a material was not set on the object.
The material structure description - SetMaterial()

See also:
SetMaterial

5.1.4.1.8 GetVolume

---

double sgC3DObject::GetVolume()

Description:
Returns a volume of the 3D object polygonal presentation.

Arguments:
No arguments.

Returned value:
Returns a volume of the 3D object polygonal presentation. 0.0 is returned in case of a surface.

See also:

GetSquare

5.1.4.1.9 GetSquare

---

double sgC3DObject::GetSquare()

Description:

Returns a surface square of the 3D object polygonal presentation.

Arguments:

No arguments.

Returned value:

Returns a surface square of the 3D object polygonal presentation.

See also:
GetVolume

5.1.4.2 sgCBox

---

sgCBox

sgCBox is the representative of the BOX concept. Any box can be set by three dimensions.

The sgCBox class methods:

Create
GetGeometry

5.1.4.2.1 Create

---

static sgCBox* sgCBox::Create(double sizeX, double sizeY, double sizeZ)

Description:

Creates an object of the BOX class with a vertex in the coordinate origin and the sides directed along the X, Y and Z axes.

Arguments:

sizeX - box size on the X axis,
sizeY - box size on the Y axis,
sizeZ - box size on the Z axis.

Returned value:

Returns the pointer to the created object. If one of the sizes is negative or equal to zero, the function returns NULL.

Following shortening is set:

```
#define    sgCreateBox  sgCBox::Create
```

Example:

```
sgCBox*     bx = sgCreateBox(10.0, 20.0, 3.0);
```

See also:
Objects hierarchy sgCObject methods SG_BOX

5.1.4.2.2 GetGeometry

---

void sgCBox::GetGeometry(SG_BOX& box_geom)

Description:

Fills the box geometry structure.

Arguments:

box_geom - return box geometry.

Returned value:

Fills the argument with box geometry. Read more about box geometry at SG_BOX.

Example:

```
sgCBox*      bx = sgCreateBox(10.0, 15.0, 40.0);
SG_BOX     bx_geo;
bx->GetGeometry(bx_geo);
assert(bx_geo.SizeX==10.0)
assert(bx_geo.SizeY==15.0);
assert(bx_geo.SizeZ==40.0);
```

See also:
Objects hierarchy sgCObject methods SG_BOX sgCBox::Create

5.1.4.3 sgCSphere

sgCSphere

sgCSphere is the representative of the SPHERE concept. Any sphere can be set by a radius, the number of meridians and parallels.

The sgCSphere class methods:
Create
GetGeometry

5.1.4.3.1 Create

static sgCSphere* sgCSphere::Create(double rad, short merid, short parall)

Description:
Creates an object of the SPHERE class with a center in the coordinate origin.

Arguments:
rad - sphere radius,
merid - the number of meridians,
parall - the number of parallels.
More about the arguments meanings - SG_SPHERE

Returned value:
Returns the pointer to the created object. If one of the arguments is negative or equal to zero, the function returns NULL.

Following shortening is set:

```
#define    sgCreateSphere    sgCSphere::Create
```

Example:

```
sgCSphere*      sph = sgCreateSphere(10.0, 24, 24);
```

See also:
Objects hierarchy sgCObject methods SG_SPHERE

5.1.4.3.2 GetGeometry

void sgCSphere::GetGeometry(SG_SPHERE& sph_geom)

Description:

Fills the sphere geometry structure.

Arguments:

sph_geom - return sphere geometry.

Returned value:

Fills the argument with sphere geometry. Read more about sphere geometry at SG_SPHERE.

Example:

```
sgCSphere*      sph = sgCreateSphere(10.0, 24, 25);
SG_SPHERE     sph_geo;
sph->GetGeometry(sph_geo);
assert(sph_geo.Radius==10.0)
assert(sph_geo.MeridiansCount==24);
assert(sph_geo.ParallelsCount==25);
```

See also:
Objects hierarchy sgCObject methods SG_SPHERE sgCSphere::Create

5.1.4.4 sgCCylinder

sgCCylinder

sgCCylinder is the representative of the CYLINDER concept. Any cylinder can be set by the base radius, the height and the number of meridians.

The sgCCylinder class methods:

Create
GetGeometry

5.1.4.4.1 Create

static sgCCylinder* sgCCylinder::Create(double rad, double heig,

short merid)

Description:

Creates an object of the CYLINDER class with the base center in the coordinate origin and directed along the Z axis.

Arguments:

rad - cylinder radius,
heig - cylinder height,
merid - the number of meridians.
More about the arguments meanings - SG_CYLINDER

Returned value:

Returns the pointer to the created object. If one of the arguments is negative or equal to zero, the function returns NULL.

Following shortening is set:

```
#define   sgCreateCylinder    sgCCylinder::Create
```

Example:

```
sgCCylinder*     cyl = sgCreateCylinder(5.0, 10.0, 36);
```

See also:
Objects hierarchy sgCObject methods SG_CYLINDER

5.1.4.4.2 GetGeometry

void sgCCylinder::GetGeometry(SG_CYLINDER& cyl_geom)

Description:

Fills the cylinder geometry structure.

Arguments:

cyl_geom - the return cylinder geometry.

Returned value:

Fills the argument with cylinder geometry. Read more about cylinder geometry at SG_CYLINDER

.

Example:

```
sgCCylinder*   cyl = sgCreateCylinder(10.0, 15.0, 24);
SG_CYLINDER     cyl_geo;
cyl->GetGeometry(cyl_geo);
assert(cyl_geo.Radius==10.0)
```

```
assert(cyl_geo.Height==15.0);
assert(cyl_geo.MeridiansCount==24);
```

See also:
Objects hierarchy sgCObject methods SG_CYLINDER sgCCylinder::Create

5.1.4.5 sgCCone

---

sgCCone

sgCCone is the representative of the CONE concept. Any cone can be set by two base radii, the height and the number of meridians.

The sgCCone class methods:
- Create
- GetGeometry

5.1.4.5.1 Create

---

static sgCCone* sgCCone::Create(double rad_1, double rad_2, double heig, short merid)

Description:

Creates an object of the CONE class with the first base center in the coordinate origin and directed along the Z axis.

Arguments:

rad_1 - the first base radius,
rad_2 - the second base radius,
heig - cone height,
merid - the number of meridians.
More about the arguments meanings - SG_CONE

Returned value:

Returns the pointer to the created object. If one of the arguments is negative or equal to zero, the function returns NULL.

Following shortening is set:

```
#define   sgCreateCone   sgCone::Create
```

Example:

```
sgCCone*   con = sgCreateCone(2.0, 1.0, 5.0, 24);
```

See also:
Objects hierarchy sgCObject methods SG_CONE

5.1.4.5.2 GetGeometry

---

void sgCCone::GetGeometry(SG_CONE& cone_geom)

Description:

Fills the cone geometry structure.

Arguments:

cone_geom - return cone geometry.

Returned value:

Fills the argument with cone geometry. Read more about cone geometry at SG_CONE

Example:

```
sgCCone*      cn = sgCreateCone(10.0, 15.0, 40.0 ,24);
SG_CONE     cn_geo;
cn->GetGeometry(cn_geo);
assert(cn_geo.Radius1==10.0)
assert(cn_geo.Radius2==15.0);
assert(cn_geo.Height==40.0);
```

See also:
Objects hierarchy sgCObject methods SG_CONE sgCCone::Create

5.1.4.6 sgCTorus

---

sgCTorus

sgCTorus is the representative of the TORUS concept. Any torus can be set by the radius, the thickness, the number of meridians and the number of the cut meridians.

The sgCTorus class methods:

Create
GetGeometry

5.1.4.6.1 Create

static sgCTorus* sgCTorus::Create(double r1, double r2, short m1, short m2)

Description:

Creates an object of the TORUS class with the center in the coordinate origin and the normal along the Z axis.

Arguments:

r1 - torus radius,
r2 - torus thickness,
m1 - the number of meridians on the torus circle,
m2 - the number of meridians on the torus cut
More about the arguments meanings - SG_TORUS

Returned value:

Returns the pointer to the created object. If one of the arguments is negative or equal to zero, the function returns NULL.

Following shortening is set:

```
#define    sgCreateTorus   sgCTorus::Create
```

Example:

```
sgCTorus*      tor = sgCreateTorus(10.0, 1.0, 24, 24);
```

See also:
Objects hierarchy sgCObject methods SG_TORUS

5.1.4.6.2 GetGeometry

void sgCTorus::GetGeometry(SG_TORUS& torus_geom)

Description:

Fills the torus geometry structure.

Arguments:

torus_geom - return torus geometry.

Returned value:

Fills the argument with torus geometry. Read more about torus geometry at SG_TORUS

Example:

```
sgCTorus*     tor = sgCreateTorus(10.0, 15.0, 36, 36);
SG_TORUS     tor_geo;
tor->GetGeometry(tor_geo);
assert(tor_geo.Radius1==10.0)
assert(tor_geo.Radius2==15.0);
assert(tor_geo.MeridiansCount1==36);
```

See also:
Objects hierarchy sgCObject methods SG_TORUS sgCTorus::Create

5.1.4.7 sgCEllipsoid

---

sgCEllipsoid

sgCEllipsoid is the representative of the ELLIPSOID concept. Any ellispoid can be set by three radii, the number of meridians and parallels.

The sgCEllipsoid class methods:
Create
GetGeometry

5.1.4.7.1 Create

---

static sgCEllipsoid* sgCEllipsoid::Create(double radius1, double radius2, double radius3, short merid_cnt, short parall_cnt)

Description:

Creates an object of the ELLIPSOID class with the center in the coordinate origin.

Arguments:

radius1 - ellipsoid radius on the X axis,
radius2 - ellipsoid radius on the Y axis,
radius3 - ellipsoid radius on the Z axis,
merid_cnt - the number of meridians,
parall_cnt - the number of parallels.
More about the coordinates meanings - SG_ELLIPSOID

Returned value:

Returns the pointer to the created object. If one of the arguments is negative

or equal to zero, the function returns NULL.

Following shortening is set:

```
#define  sgCreateEllipsoid   sgEllipsoid::Create
```

Example:

```
sgCEllipsoid*     ell = sgCreateEllipsoid(10.0, 20.0, 5.0, 24, 36);
```

See also:
Objects hierarchy sgCObject methods SG_ELLIPSOID

5.1.4.7.2 GetGeometry

---

void sgCEllipsoid::GetGeometry(SG_ELLIPSOID& ell_geom)

Description:

Fills the ellipsoid geometry structure.

Arguments:

ell_geom - return ellipsoid geometry.

Returned value:

Fills the argument with ellipsoid geometry. Read more about ellipsoid geometry at SG_ELLIPSOID

Example:

```
sgCEllipsoid*      ell = sgCreateEllipsoid(10.0, 15.0, 40.0, 24, 24);
SG_ELLIPSOID    ell_geo;
ell->GetGeometry(ell_geo);
assert(ell_geo.Radius1==10.0)
assert(ell_geo.Radius2==15.0);
assert(ell_geo.Radius3==40.0);
```

See also:
Objects hierarchy sgCObject methods SG_ELLIPSOID sgCEllipsoid::Create

5.1.4.8 sgCSphericBand

---

sgCSphericBand

sgCSphericBand is the representative of the SPHERICAL BAND concept. Any spherical band can be set by the radius of the generating sphere, the number of meridians and two coefficients of the cuts.

The sgCSphericBand class methods:

Create
GetGeometry

5.1.4.8.1 Create

---

static sgCSphericBand* sgCSphericBand::Create(double radius, double beg_koef, double end_koef, short merid_cnt)

Description:

Creates an object of the SPHERICAL BAND class with the generating sphere center in the coordinate origin and the normal along the Z axis.

Arguments:

radius - spherical band radius,
beg_koef - start clipping coefficient (from -1 to 1),
end_koef - end clipping coefficient (from -1 to 1),
merid_cnt - number of meridians.
More about the coordinates meanings - SG_SPHERIC_BAND

Returned value:

Returns the pointer to the created object. If the radius is negative or the number of meridians is negative, or the clipping coefficients are smaller than -1 or more than 1, the function returns NULL.

Following shortening is set:

```
#define   sgCreateSphericBand   sgSphericBand::Create
```

Example:

```
sgCSphericBand*     spB = sgCreateSphericBand(5.0, -0.5,  0.5, 24);
```

See also:
Objects hierarchy sgCObject methods SG_SPHERIC_BAND

5.1.4.8.2 GetGeometry

---

void sgCSphericBand::GetGeometry(SG_SPHERIC_BAND& spb_geom)

Description:

Fills the spherical band geometry structure.

Arguments:

spb_geom - return spherical band geometry.

Returned value:

Fills the argument with spherical band geometry. Read more about spherical band geometry at SG_SPHERIC_BAND

Example:

```
sgCSphericBand*   spb = sgCreateSphericBand(10.0, -0.5, 0.5, 36);
SG_BOX     spb_geo;
spb->GetGeometry(spb_geo);
assert(spb_geo.Radius==10.0)
assert(spb_geo.BeginCoef==-0.5);
assert(spb_geo.EndCoef==0.5);
```

See also:
Objects hierarchy sgCObject methods SG_SPHERIC_BAND sgCSphericBand::Create

5.1.4.9 Additional types

Additional types are the secondary types that are used in sgCore. The objects of additional types are used as arguments or returned values of the main classes and functions of the sgCore library.

5.1.4.9.1 SG_TRIANGULATION_TYPE

Object triangulation methods.

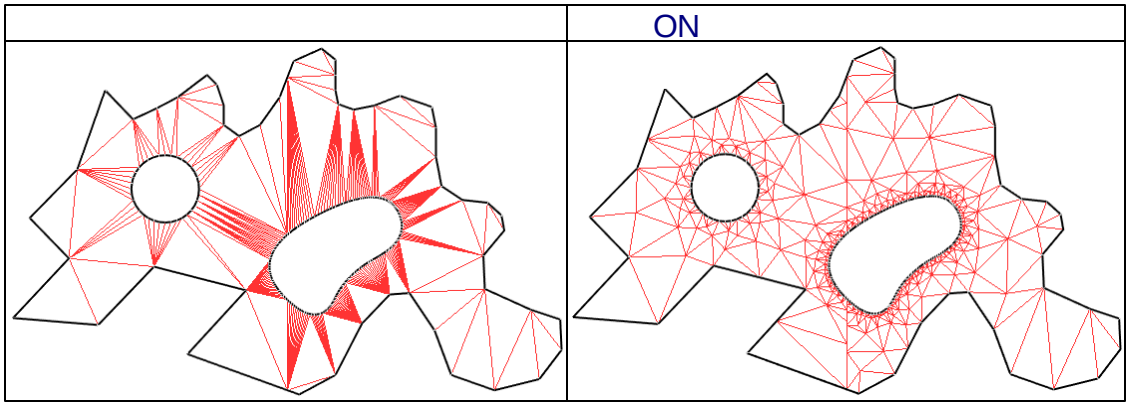
```
typedef enum
{
    SG_VERTEX_TRIANGULATION,
    SG_DELAUNAY_TRIANGULATION
} SG_TRIANGULATION_TYPE;
```

SG_VERTEX_TRIANGULATION – triangulation which doesn't create any new vertexes, i.e. all the vertexes of triangles are the vertexes of initial contours. This type has one disadvantage – it can cause the creation of stretched triangles.

SG_DELAUNAY_TRIANGULATION - Delaunay triangulation.

Let's compare the triangulation methods on the following examples:

SG_VERTEX_TRIANGULATION	SG_DELAUNAY_TRIANGULATI



Definition in sg3D.h

5.1.4.9.2 sgCBRepPiece

sgCBRepPiece

sgCBRepPiece class is the representation of the “BRep piece” term of the sgCore library. To learn more about the surface model view and BReps see the sgCBRep class description.

5.1.4.9.2.1 Methods

sgCBRepPiece methods

The following methods in the sgCBRepPiece class are defined in the current version of sgCore:

GetVertexes
GetVertexesCount
GetEdges
GetEdgesCount

void sgCBRepPiece::GetLocalGabarits(SG_POINT& p_min, SG_POINT& p_max)

Description:

Returns the angles of the bounding parallelepiped of the BRep piece.

Arguments:

p_min - takes the value of the corner in the bounding parallelepiped with the minimal coordinates.
p_max - takes the value of the corner in the bounding parallelepiped with the maximal coordinates.

Returned value:

No values.

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece

---

const SG_POINT* sgCBRepPiece::GetVertexes() const

Description:

Returns the index to the vertex massive of a BRep piece. Total number of the vertexes is returned by the GetVertexesCount() function

Arguments:

No arguments.

Returned value:

Returns the index to the vertex massive of a BRep piece. To learn more about BRep, see the description of the sgCBRep class

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece GetVertexesCount

---

unsigned int sgCBRepPiece::GetVertexesCount() const

Description:

Returns the number of vertexes in a BRep piece. The vertex array is returned by the GetVertexes function

Arguments:

No arguments.

Returned value:

Returns the number of vertexes in a BRep piece.

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece GetVertexes

---

const SG_EDGE* sgCBRepPiece::GetEdges() const

Description:

Returns the poiner to the structure array of the BRep piece edges description. The total number of the edges eturned by the GetEdgesCount() function

Arguments:

No arguments.

Returned value:

Returns the index to the structure array of the BRep piece edges description. One edge is described by the following structure:

```
typedef struct
{
  unsigned short begin_vertex_index;
  unsigned short end_vertex_index;
  unsigned short edge_type;
}   SG_EDGE;
```

begin_vertex_index - index of the first edge point in the vertexes array of the BRep piece returned by the GetVertexes function

end_vertex_index - – index of the second edge point in the vertexes array of the BRep piece returned by the GetVertexes function

edge_type - flag of the edge type. It can have the following values: SG_EDGE_1_LEVEL, SG_EDGE_2_LEVEL or SG_EDGE_3_LEVEL. . Each flag describes to which level of the object detalization the edge belongs

To learn more about BRep, see the sgCBRep class description

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece GetEdgesCount

unsigned int sgCBRepPiece::GetEdgesCount() const

Description:

Returns the number of edges in the BRep piece. The edge array itself is returned by the GetEdges function

Arguments:

No arguments.

Returned value:

Returns the number of edges in the BRep piece

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece GetEdges

void sgCBRepPiece::GetTrianglesRange(int& min_numb, int& max_numb) const

Description:

Returns the minimal and maximal number of the triangle of the BRep piece from the array of all triangles in the 3D object returned by the sgC3DObject:: GetTriangles() function. All triangles of the BRep piece are located in the array between these numbers

Arguments:

min_numb - the returned minimal number of the triangle.
max_numb - the returned maximal number of the triangle

Returned value:

No values.

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece GetVertexesCount

5.1.4.9.3 SgCBRep

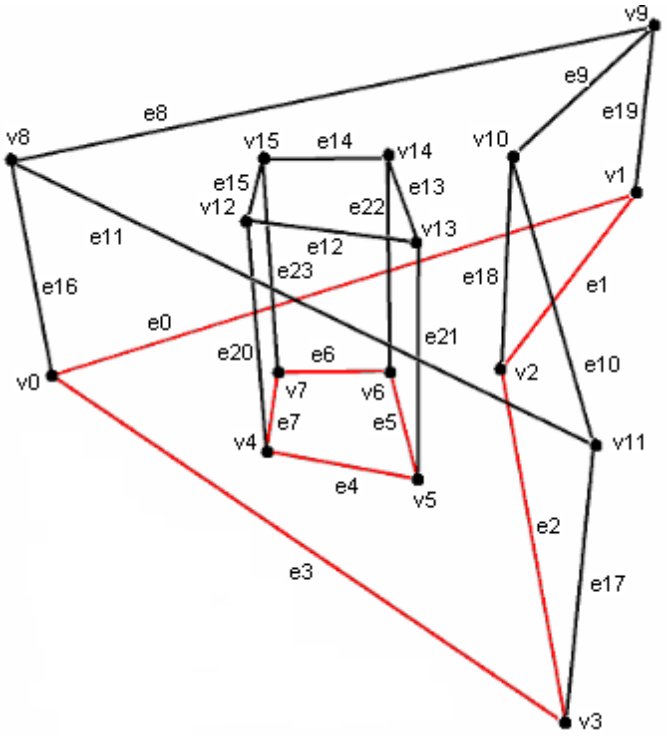
sgCBRep

In the sgCore library all 3D solids are stored in the form of the polygonal surface representation - boundary representation – BRep. This representation defines a solid implicitly by describing its boundary surface. Each surface is approximated by a set of **faces**. The fragmentation is performed in such a way that each face has a compact mathematical representation.

Face boundaries are represented by **edges.** Edges are selected like faces in such a way that each edge has a compact mathematical description. A part of the curve that forms an edge is ended by **vertexes**. It's necessary to introduce such a term as **cycle (**the description of each face contour) because every face can be a non-convex polygon with any number of holes.

A surface model which has only flat faces is called a **polygonal model** – this type of description is used in sgCore. So, every face is a polygon which consists of the vertex coordinates sequence. An object consists of the set of faces.

SgCore also introduces such a term as BRep piece. A BRep piece is a piece of simply connected boundary body surface.
This term has been introduced to accelerate the work of some algorithms (such as Boolean operations). Each BRepPiece has a limitary dimensional parallelogram. It was designed to avoid passes of each BRep pieces face in the cycles. All BRep pieces connected together form a full BRep.
Let's look at the BRep description on the solid example created by extruding a flat non-convex contour with one hole. This BRep is divided into 4 pieces – 2 bases, a side face and a hole surface.



Below you can see the table of the vertex and edge descriptions for all BRep pieces.
The vertex coordinate recurrence can be avoided by grouping the vertex coordinates in a separate array. In this case vertex coordinate indexes in the array (not the vertex coordinates themselves) get associated with faces and edges. In this case (see the picture) we have:

BRep Piece number and description	Vertexes					Edges			
	Name	Number	X	Y	Z	Name	Number	Begin	End
0 Bottom base	v0	0	-2.0	0.0	0.0	e0	0	0	1
	v1	1	1.0	2.0	0.0	e1	1	1	2
	v2	2	0.0	0.0	0.0	e2	2	2	3
	v3	3	0.0	-2.0	0.0	e3	3	3	0
	v4	4	-1.0	-0.6	0.0	e4	4	4	5
	v5	5	-0.4	-0.8	0.0	e5	5	5	6
	v6	6	-0.5	0.0	0.0	e6	6	6	7
	v7	7	-1.0	0.0	0.0	e7	7	7	4
	v0	0	-2.0	0.0	0.0	e0	0	0	2
	v8	1	-2.0	0.0	1.0	e16	1	0	1
	v1	2	1.0	2.0	0.0	e8	2	1	3

1 **Side face**	v9 v2 v10 v3 v11	3 4 5 6 7	1.0 0.0 0.0 0.0 0.0	2.0 0.0 0.0 -2.0 -2.0	1.0 0.0 1.0 0.0 1.0	e1 e19 e9 e2 e18 e10 e3 e17 e11	3 4 5 6 7 8 9 10 11	2 2 3 4 4 5 6 6 7	4 3 5 6 5 7 0 7 1
2 **Top base**	v8 v9 v10 v11 v12 v13 v14 v15	0 1 2 3 4 5 6 7	-2.0 1.0 0.0 0.0 -1.0 -0.4 -0.5 -1.0	0.0 2.0 0.0 -2.0 -0.6 -0.8 0.0 0.0	1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0	e8 e9 e10 e11 e12 e13 e14 e15	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	1 2 3 0 5 6 7 4
3 **Hole surface**	v4 v12 v5 v13 v6 v14 v7 v15	0 1 2 3 4 5 6 7	-1.0 -1.0 -0.4 -0.4 -0.5 -0.5 -1.0 -1.0	-0.6 -0.6 -0.8 -0.8 0.0 0.0 0.0 0.0	0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0	e4 e20 e12 e5 e21 e13 e6 e22 e14 e7 e23 e15	0 1 2 3 4 5 6 7 8 9 10 11	0 0 1 2 2 3 4 4 5 6 6 7	2 1 3 4 3 5 6 5 7 0 7 1

A face is represented by a set of edges. An index of the initial face edge is set and the cycle structure determines each next face edge. In this way the outer contours of the faces and their holes can be set.

5.1.4.9.3.1 Methods

sgCBRep methods

The following methods in the sgCBRep class are defined in the current version of sgCore:

GetPiecesCount
GetPiece

unsigned int sgCBRep::GetPiecesCount() const

Description:

Returns a number of pieces forming BRep.

Arguments:

No arguments.

Returned value:

Returns a number of pieces forming BRep. To learn more about BRep, see the sgCBRep class description

See also:
sgCBRep sgCBRep::GetPiece sgCBRepPiece

sgCBRepPiece* sgCBRep::GetPiece(unsigned int nmbr) const

Description:

Returns a BRep piece according to its number. The total number of pieces is returned by the GetPiecesCount() function

Arguments:

nmbr - number of BRep piece.

Returned value:

Returns the pointer to the sgCBRepPiece class object according to its number. To learn more about BRep, see the sgCBRep class description

See also:
sgCBRep sgCBRep::GetPiecesCount sgCBRepPiece

5.1.5 sgCGroup

sgCGroup

sgCGroup is the representative of the OBJECTS GROUP concept. The group itself

is not a geometrical concept but it's just a group of other geometrical objects (including other groups). Like to any other object you can set a user geometry to the sgCGroup class object and thus you can parametrize arbitrary complex objects. For each object being a component part of the group the GetParent function will return the pointer to this group.

The sgCGroup class methods:
GetChildrenList
CreateGroup
BreakGroup

5.1.5.1 CreateGroup

---

static sgCGroup* sgCGroup::CreateGroup(sgCObject** objcts, int cnt)

Description:
Creates a group of objects from the array.

Arguments:
objcts - array of pointers to the objects.
cnt - the number of objects in this array.

Returned value:
Returns the pointer to the created group. If the function fails returns NULL.

Example:

```
sgCObject*   objcts[4];
objcts[0] = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
objcts[1] = sgCreateLine(2.0, 3.0, 5.0, 10.0, 0.0, 3.0);
objcts[2] = sgCreateLine(10.0, 0.0, 3.0, 6.0, 13.0, 15.0);
objcts[0] = sgCGroup::CreateGroup(&objcts[0],3);
objcts[1] = sgCreateBox(6.0, 13.0, 15.0);
objcts[2] = sgCreateSphere(20.0, 24, 24);
objcts[3] = sgCreateLine(10.0, 6.0, -3.0, 0.0, 0.0, 0.0);
sgCGroup* resGroup = sgCGroup::CreateGroup(&objcts[0],4);
```

See also:
GetChildrenList BreakGroup

5.1.5.2 BreakGroup

---

bool sgCGroup::BreakGroup(sgCObject** objcts)

Description:

Takes the group to component parts.

Arguments:

objcts - buffer to put the group component parts. *You should allocate the memory for an array of objects pointers beforehand. The number of the elements in this array must be equal to the group child objects (returned by GetChildrenList->GetCount()).*

Returned value:

Returns false if the function fails, otherwise true. The function argument is filled with the pointers to child objects.

Example:

```
/*Build group*/
sgCObject*   objcts[4];
objcts[0] = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
objcts[1] = sgCreateLine(2.0, 3.0, 5.0, 10.0, 0.0, 3.0);
objcts[2] = sgCreateLine(10.0, 0.0, 3.0, 6.0, 13.0, 15.0);
objcts[0] = sgCGroup::CreateGroup(&objcts[0],3);
objcts[1] = sgCreateBox(6.0, 13.0, 15.0);
objcts[2] = sgCreateSphere(20.0, 24, 24);
objcts[3] = sgCreateLine(10.0, 6.0, -3.0, 0.0, 0.0, 0.0);
sgCGroup* resGroup = sgCGroup::CreateGroup(&objcts[0],4);

   /*Break contour*/
const int ChildsCount = resGroup->GetChildrenList()->GetCount();

  sgCObject**  allChilds = (sgCObject**)malloc(ChildsCount*sizeof
(sgCObject*));
  if (!resGroup->BreakGroup(allChilds))
  {
    assert(0);
  }
  const int sz = resGroup->GetChildrenList()->GetCount();
  assert(sz==0);
  sgDeleteObject(resGroup);
  for (int i=0;i<ChCnt;i++)
  {
       sgGetScene()->AttachObject(allChilds[i]);
  }
  free(allChilds);
```

See also:
CreateGroup

5.1.5.3 GetChildrenList

IObjectsList* sgCGroup::GetChildrenList() const

Description:

Returns the pointer to the list of the child objects.

Arguments:

No arguments.

Returned value:

Returns the pointer to the list of the group child objects.
More - IObjectsList

See also:
CreateGroup BreakGroup

5.2 Additional types

Additional types are the secondary types that are used in sgCore. The objects of additional types are used as arguments or returned values of the main classes and functions of the sgCore library.

5.2.1 SG_OBJECT_TYPE

Object type identifier.

Definition (in sgCObject.h):

```
typedef enum
{
    SG_OT_BAD_OBJECT,
    SG_OT_POINT,
    SG_OT_LINE,
    SG_OT_CIRCLE,
    SG_OT_ARC,
    SG_OT_SPLINE,
    SG_OT_TEXT,
    SG_OT_CONTOUR,
    SG_OT_DIM,
    SG_OT_3D,
    SG_OT_GROUP
} SG_OBJECT_TYPE;
```

Definition in sgCObject.h

5.2.2 IObjectsList

An *object list* is an ordered list of the pointers to the objects of classes inherited from sgCObject.

```
class IObjectsList
{
public:
virtual int          GetCount()    const = 0; // the number of the pointer in the list
virtual sgCObject*   GetHead()     const = 0; // the pointer to the object at the beginning of the list
virtual sgCObject*   GetNext(sgCObject*) const = 0; // the pointer to the object following the current one in the list
virtual sgCObject*   GetTail()       const = 0; // the pointer to the object at the end of the list
virtual sgCObject*   GetPrev(sgCObject*) const = 0; // the pointer to the object prior to the current one in the list
};
```

Definition (in sgCObject.h)

5.2.3 SG_USER_DYNAMIC_DATA

SG_USER_DYNAMIC_DATA - is a basic structure for a user-defined data block. A data block of any size and content can be assigned to any object of the class inherited from sgCObject. The only thing you should realize is a memory release function - Finalize().
This function is called when deleting an object.

```
struct  SG_USER_DYNAMIC_DATA
{
  virtual void Finalize()   =0;
};
```

To learn more about SG_USER_DYNAMIC_DATA, see the description of the SetUserDynamicData and GetUserDynamicData functions

6 sgCMatrix

---

sgCMatrix

sgCMatrix is a class of the 4x4 matrix mathematical concept. This class includes all necessary functions to work with matrices and affine transformations presentation in the three-dimensional space.

The class is described in sgMatrix.h

The sgCMatrix class methods:

Constructor and destructor:

- sgCMatrix()
- sgCMatrix(const double*)
- sgCMatrix(sgCMatrix&)
- ~sgCMatrix()

Methods:

- operator=
- SetMatrix
- GetData
- GetTransparentData
- Identity
- Transparent
- Inverse
- Multiply
- Translate
- Rotate
- VectorToZAxe
- ApplyMatrixToVector
- ApplyMatrixToPoint

6.1 Constructor and destructor

---

sgCMatrix::sgCMatrix()

Description:

Constructor. Creates a 4x4 unit matrix.

Arguments:

No arguments.

Returned value:

No value.

Example:

```
sgCMatrix*     matr = new sgCMatrix;
```

---

sgCMatrix::sgCMatrix(const double* matr_data)

Description:

Constructor. Creates a matrix by the array of sixteen numbers of the double type.

Arguments:

matr_data - pointer to the beginning of the array of sixteen numbers of the double type. If the argument is NULL, a unit matrix is created (which is equal to the default constructor).

Returned value:

No values.

Example:

```
double* matr_data = new double[16];
memset(matr_data, 0, sizeof(double*16));
sgCMatrix*     matr = new sgCMatrix(matr_data);
delete[]  matr_data;
```

---

sgCMatrix::sgCMatrix(sgCMatrix& matr_B)

Description:Description:

Constructor. Creates a matrix by the existing object of the sgCMatrix class.

Arguments:

matr_B - sgCMatrix class object. The created matrix is a copy of mart_B.

Returned value:

No values.

Example:

```
sgCMatrix     matr1;
```

```
sgCMatrix     matr2(matr1);
```

---

sgCMatrix::~sgCMatrix()

Description:
Destructor.

Arguments:
No arguments.

Returned value:
No values.

6.2 operator=

---

sgCMatrix& sgCMatrix::operator=(cosnt sgCmatrix& matr_B)

Description:
Sets the sgCMatrix class object equaled to matr_B.

Arguments:
matr_B - the matrix the object class is set equaled to.

Returned value:
A pointer to the current matrix.

Example:

```
sgCMatrix matr1;
sgCMatrix matr2;
matr1 = matr2;
```

6.3 SetMatrix

---

bool sgCMatrix::SetMatrix(const sgCMatrix* matr_B)

Description:
Copies the sgCMatrix object class to another sgCMatrix class object.

Arguments:

matr_B - matrix the class object is set equaled to. If the argument is NULL, the function returns false.

Returned value:

If the argument is NULL, the function returns false, otherwise true.

Example:

```
sgCMatrix matr1;
sgCMatrix matr2;
matr1.SetMatrix(&matr2);
```

6.4 GetData

---

const double* sgCMatrix::GetData()

Description:

Returns the pointer to the array of the sixteen numbers of the double type.

Arguments:

No arguments.

Returned value:

Returns the pointer to the data which are the representatives of a 4x4 matrix. In other words the pointer returns to the block of memory storing the array of the sixteen numbers of the double type.

Example:

```
sgCMatrix matr1;
const double* matr_data = matr1.GetData();
assert(matr_data[0]==1.0);
assert(matr_data[1]==0.0);
```

6.5 GetTransparentData

---

const double* sgCMatrix::GetTransparentData()

Description:

Returns the pointer to the array of the sixteen numbers of the double type.

Arguments:

No arguments.

Returned value:

Returns the pointer to the data which are the representatives of a transposed

4x4 matrix. The pointer returns to the temporary buffer, the matrix itself is not transposed.

Example:

```
sgCMatrix matr1;
const double* matr_data = matr1.GetTransparentData();
assert(matr_data[0]==1.0);
assert(matr_data[1]==0.0);
```

6.6 Identity

void sgCMatrix::Identity()

Description:
Makes the matrix a unit matrix.

Arguments:
No arguments.

Returned value:
No values.

6.7 Transparent

void sgCMatrix::Transparent()

Description:
Transposes the matrix.

Arguments:
No arguments.

Returned value:
No values.

6.8 Inverse

bool sgCMatrix::Inverse()

Description:
Makes the matrix inverse.

Arguments:
No arguments.

Returned value:
true - if there is an inverse matrix
false - if there is no inverse matrix

6.9 Multiply

---

void sgCMatrix::Multiply(const sgCMatrix& matr_B)

Description:
Multiplies the current matrix by the mart_B matrix.

Arguments:
matr_B - matrix we multiply the current one by.

Returned value:
No values.

6.10 Translate

---

void sgCMatrix::Translate(const SG_VECTOR& transVector)

Description:
Multiplies the current matrix by the matrix of affine transformation of the shift in the three-dimensional space.

Arguments:
transVector - shift vector in the three-dimensional space.

Returned value:
No values.

See also:
SG_VECTOR

6.11 Rotate

---

void sgCMatrix::Rotate(const SG_POINT& axePoint, const SG_VECTOR& axeDir, double alpha_radians)

Description:
Multiplies the current matrix by the matrix of affine transformations of the shift in the three-dimensional space.

Arguments:
axePoint - point on the rotation axis
axeDir - direction vector on the rotation axis
alpha_radians - rotation anlge about the rotation axis (in radians). Following the axeDir vector the positive angle direction is clockwise.

Returned value:
No values.

See also:
SG_POINT SG_VECTOR

6.12 VectorToZAxe

void sgCMatrix::VectorToZAxe(const SG_VECTOR& vect)

Description:
Creates an affine transformation matrix which transforms the specified vector to the (0, 0, 1) vector.

Arguments:
vect - vector to be transformed.

Returned value:
No values.

See also:
SG_VECTOR

6.13 ApplyMatrixToVector

void sgCMatrix::ApplyMatrixToVector(SG_POINT& vectBegin, SG_VECTOR& vectDir) const

Description:
Applies the current affine transformations matrix to the vector terminating at the specified point.

Arguments:

vectBegin - tail of the vector we apply the affine transformation to,
vectDir - direction of the vector we apply the affine transformation to.

Returned value:

No values.

See also:
SG_VECTOR

6.14 ApplyMatrixToPoint

---

void sgCMatrix::ApplyMatrixToPoint(SG_POINT& pnt)

Description:

Applies the affine transformation matrix to the point.

Arguments:

pnt - point we applies the affine transformation to

Returned value:

No values.

See also:
SG_POINT

7 sgSpaceMath

---

sgSpaceMath

sgSpaceMath is a namespace which provides you with some tools of the Euclidean geometry.

The class is described in sgSpaceMath.h

The sgSpaceMath namespace functions:

IsPointsOnOneLine
PointsDistance
NormalVector
VectorsAdd
VectorsSub

7.1 IsPointsOnOneLine

---

bool sgSpaceMath::IsPointsOnOneLine(const SG_POINT& p1, const SG_POINT& p2, const SG_POINT& p3)

Description:
Checks whether the point in the three-dimensional space lie on the same line.

Arguments:
p1 - first point,
p2 - second point,
p3 - third point

Returned value:
Returns true if the points lie on the same line and false otherwise.

See also:
SG_POINT

7.2 PointsDistance

---

double sgSpaceMath::PointsDistance(const SG_POINT& p1, const SG_POINT& p2)

Description:
Calculates the distance beetween two points.

Arguments:
p1 - first point,
p2 - second point.

Returned value:

Returns the distance beetween two points.

See also:
SG_POINT

7.3 NormalVector

---

bool sgSpaceMath::NormalVector(SG_VECTOR& vect)

Description:
Normalizes the vector.

Arguments:
vect - normalizable vector.

Returned value:
If the vector is zero the function returns false, otherwise true

See also:
SG_VECTOR

7.4 VectorsAdd

SG_VECTOR sgSpaceMath::VectorsAdd(const SG_VECTOR& v1,const SG_VECTOR& v2)

Description:
Sums up two vectors.

Arguments:
v1 - first vector,
v2 - second vector.

Returned value:
Returns the sum of the arguments.

See also:
SG_VECTOR

7.5 VectorsSub

SG_VECTOR sgSpaceMath::VectorsSub(const SG_VECTOR& v1,const SG_VECTOR& v2)

Description:
Calculates the difference between two vectors.

Arguments:
v1 - first vector,
v2 - second vector.

Returned value:
Returns the difference between the arguments

See also:
SG_VECTOR

7.6 VectorsScalarMult

double sgSpaceMath::VectorsScalarMult(const SG_VECTOR& v1,const SG_VECTOR& v2)

Description:
Calculates the scalar multiplication of two vectors.

Arguments:
v1 - first vector,
v2 - second vector.

Returned value:
Returns the scalar multiplication of the arguments

See also:
SG_VECTOR

7.7 VectorsVectorMult

SG_VECTOR sgSpaceMath::VectorsVectorMult(const SG_VECTOR& v1,const SG_VECTOR& v2)

Description:
Calculates the vector multiplication of two vectors.

Arguments:
v1 - first vector,
v2 - second vector.

Returned value:
Returns the vector multiplication of the arguments

See also:
SG_VECTOR

7.8 ProjectPointToLineAndGetDist

---

double sgSpaceMath::ProjectPointToLineAndGetDist(const SG_POINT& lineP, const SG_VECTOR& lineDir, const SG_POINT& pnt, SG_POINT& resPnt)

Description:
Finds a projection of the point on the line in the three-dimensional space and returns the distance from the point to the line.

Arguments:
lineP - point on the line,
lineDir - line direction vector,
pnt - projected point,
resPnt - the result of projecting the pnt point on the line.

Returned value:
Returns the distance from the point to the line and the projection of the point on the line.

See also:
SG_POINT SG_VECTOR

7.9 IntersectPlaneAndLine

SG_PLANE_AND_LINE sgSpaceMath::IntersectPlaneAndLine (const SG_VECTOR& planeNorm, const double planeD, const

SG_POINT& lineP, const SG_VECTOR& lineDir, SG_POINT& resP)

Description:

Calculates the positional relationship of the line and the plane. If they are intersecting the function returns the point of intersection.

Arguments:

planeNorm - normal to the plane,
planeD - free coefficient in the plane equation,
lineP - point on the line,
lineDir - direction vector of the line,
resP - the found intersection point of the line and the plane

Returned value:

One of the following values is returned:

SG_LINE_PARALLEL - if the line is parallel to the plane and lie on it, the resP value won't be changed,
SG_LINE_ON_PLANE - if the line lies on the plane, the resP value won't be changed,
SG_EXIST_INTERSECT_PONT - if the line and the plane are intersecting, resP will be the point of intersection.

See also:
SG_POINT SG_VECTOR Finding plane by three points Finding plane by normal and point

7.10 IsSegmentsIntersecting

---

bool sgSpaceMath::IsSegmentsIntersecting(const SG_LINE& ln1, bool as_line1, const SG_LINE& ln2, bool as_line2, SG_POINT& resP)

Description:

Checks whether two lines or line segments are intersecting in the three-dimensional space.

Arguments:

ln1 - first line segment or two points on the line if as_line1==true,
as_line1 - if true, the first argument is considered as two points on the line,
ln2 - second line segment or two points on the line if as_line2==true,
as_line2 - if true, the second argument is considered as two points on the line,
resP - the result of intersecting line segments and lines if they are

intersecting.

Returned value:

Returns true if the line segments or lines are intersecting (in this case resP is the result of they intersection) and false otherwise.

See also:
SG_LINE SG_POINT

7.11 PlaneFromPoints

---

bool sgSpacMath::PlaneFromPoints(const SG_POINT& p1, const SG_POINT& p2, const SG_POINT& p3, SG_VECTOR& resPlaneNorm, double& resPlaneD)

Description:

Calculates a normal and a free coefficient in the plane equation by three points.

Arguments:

p1 - first point,
p2 - second point,
p3 - third point,
resPlaneNorm - normal to the resultant plane,
resPlaneD - free coefficient in the resultant plane equation

Returned value:

Returns false if the points are coinsided or lie on the same line, otherwise - true.

See also:
SG_POINT SG_VECTOR

7.12 PlaneFromNormalAndPoint

---

void sgSpaceMath::PlaneFromNormalAndPoint(const SG_POINT& planePnt, const SG_VECTOR& planeNorm, double& resPlaneD)

Description:

Calculates a free coefficient in the plane equation by the point on the plane and the normal to this plane.

Arguments:

planePnt - point on the plane,
planeNorm - normal to the plane,
resPlaneD - free coefficient in the resultant plane equation

Returned value:

No values. resPlaneD takes the value of the free equation coefficient.

See also:
SG_POINT SG_VECTOR

7.13 GetPlanePointDistance

---

double sgSpaceMath::GetPlanePointDistance(const SG_POINT& pnt, const SG_VECTOR& planeNorm, const double planeD)

Description:

Calculates a distance from a point to a plane.

Arguments:

pnt - point,
planeNorm - normal to the plane,
planeD - free coefficient in the plane equation

Returned value:

Returns a distance from a point to a plane.

See also:
SG_POINT SG_VECTOR

8 sgCScene

---

sgCScene

The *Scene* concept is actually the objects container in the sgCore library. There are two object lists in a scene - the list of all scene objects and the list of the selected objects.
Besides storing objects in a scene you can save a scene to a file (and load the saved scene as well) and keep the history of working with a scene (Undo-Redo). You can both add objects to the scene and delete them from the scene. *If you added an object to the scene you can't call the DeleteObject method for this*

object.

In the current sgCore library version an object of the sgCScene class is actually a global object and is created in the sgInitKernel function. To access this global object you can use the static function - GetScene.

The sgCScene class methods:

GetScene
GetObjectsList
GetSelectedObjectsList
AttachObject
DetachObject
StartUndoGroup
EndUndoGroup
IsUndoStackEmpty
IsRedoStackEmpty
Undo
Redo
Clear
GetGabarits

8.1 GetScene

---

static sgCScene* sgCScene::GetScene()

Description:

Returns the pointer to a global object of the sgCScene class.

Arguments:

No arguments.

Returned value:

Returns the pointer to a global object of the sgCScene class. You work with the scene through this pointer.

Following shortening is set:

```
#define    sgGetScene    sgCScene::GetScene
```

Example:

```
sgLine*      line = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgGetScene()->AttachObject(line);
```

See also:
AttachObject DetachObject

8.2 GetObjectsList

IObjectsList* sgCScene::GetObjectsList() const

Description:

Returns the pointer to the scene objects list which was created after opening a file or by using the AttachObject function.

Arguments:

No arguments.

Returned value:

Returns the pointer to the objects list. More about the objects list - IObjectsList

Example:

```
sgLine*      line = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgGetScene()->AttachObject(line);
assert(sgGetScene()->GetObjectsList()->GetTail()->GetType()==SG_OT_LINE)
```

See also:
AttachObject DetachObject

8.3 GetSelectedObjectsList

IObjectsList* sgCScene::GetSelectedObjectsList() const

Description:

Returns the pointer to the scene selected objects list which was created by using the AttachObject and Select functions.

Arguments:

No arguments.

Returned value:

Returns the pointer to the selected objects list. More about the objects list - IObjectsList

Example:

```
sgLine*      line = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgGetScene()->AttachObject(line);
```

```
line->Select(true);
assert(sgGetScene()->GetSlectedObjectsList()->GetTail()->GetType()
==SG_OT_LINE)
```

See also:
AttachObject DetachObject Select

8.4 AttachObject

---

bool sgCScene::AttachObject(sgCObject* obj)

Description:
Attaches a new object to the scene.

Arguments:
obj - attachable to the scene object.

Returned value:
Returns false if the object has been already attached to the scene or if the funtion fails, otherwise - true

Note:
If the object was attached to the scene you can't use the DeleteObject function

See also:
DetachObject

8.5 DetachObject

---

void sgCScene::DetachObject(sgCObject* obj)

Description:
Detaches the object from the scene.

Arguments:
obj - detachable object.

Returned value:
No values.

Note:
If the object was detached from the scene outside the command brackets StartUndoGroup - EndUndoGroup or it doesn't attached to the scene, you must call the DeleteObject function for this object

See also:
AttachObject

8.6 StartUndoGroup

---

bool sgCScene::StartUndoGroup()

Description:
Begins saving commands into the scene history. All the commands to work with the scene and its objects will be saved to the undo group before you call the EndUndoGroup function, i.e. this group will be undo using the Undo function.

Arguments:
No arguments.

Returned value:
Returns false if the function fails and true otherwise.

Example:

```
sgLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgLine*      line2 = line1->Clone();
sgGetScene()->StartUndoGroup();
sgGetScene()->AttachObject(line1);
sgGetScene()->AttachObject(line2);
sgGetScene()->EndUndoGroup();
sgGetScene()->Undo();
```

See also:
GetScene EndUndoGroup IsUndoStackEmpty IsRedoStackEmpty Undo Redo

8.7 EndUndoGroup

bool sgCScene::EndUndoGroup()

Description:

Finishes saving commands into the scene history which was started by the StartUndoGroup function.
All the commands to work with the scene and its objects will be saved to the undo group between calling the StartUndoGroup and EndUndoGroup functions, i.e. this group will be undone using the Undo function.

Arguments:

No arguments.

Returned value:

Returns false if the function fails and true otherwise.

Example:

```
sgLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgLine*      line2 = line1->Clone();
sgGetScene()->StartUndoGroup();
sgGetScene()->AttachObject(line1);
sgGetScene()->AttachObject(line2);
sgGetScene()->EndUndoGroup();
sgGetScene()->Undo();
```

See also:
GetScene StartUndoGroup IsUndoStackEmpty IsRedoStackEmpty Undo Redo

8.8 IsUndoStackEmpty

bool sgCScene::IsUndoStackEmpty() const

Description:

Checks whether the Undo history is empty.

Arguments:

No arguments.

Returned value:

Returns false if there are any commands in the Undo history and true if the

history is empty.

Example:

```
sgLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgLine*      line2 = line1->Clone();
sgGetScene()->StartUndoGroup();
sgGetScene()->AttachObject(line1);
sgGetScene()->AttachObject(line2);
sgGetScene()->EndUndoGroup();
asset(sgGetScene()->IsUndoStackEmpty()==false);
asset(sgGetScene()->IsRedoStackEmpty()==true);
sgGetScene()->Undo();
asset(sgGetScene()->IsUndoStackEmpty()==true);
asset(sgGetScene()->IsRedoStackEmpty()==false);
```

See also:
GetScene StartUndoGroup IsUndoStackEmpty IsRedoStackEmpty Undo Redo

8.9 IsRedoStackEmpty

bool sgCScene::IsRedoStackEmpty() const

Description:
Checks whether the Redo history is empty.

Arguments:
No arguments.

Returned value:
Returns false if there are any commands in the Redo history and true if the history is empty.

Example:

```
sgLine*      line1 = sgCreateLine(0.0, 0.0, 0.0, 2.0, 3.0, 5.0);
sgLine*      line2 = line1->Clone();
sgGetScene()->StartUndoGroup();
sgGetScene()->AttachObject(line1);
sgGetScene()->AttachObject(line2);
sgGetScene()->EndUndoGroup();
asset(sgGetScene()->IsUndoStackEmpty()==false);
asset(sgGetScene()->IsRedoStackEmpty()==true);
sgGetScene()->Undo();
asset(sgGetScene()->IsUndoStackEmpty()==true);
asset(sgGetScene()->IsRedoStackEmpty()==false);
```

See also:
GetScene StartUndoGroup IsUndoStackEmpty IsRedoStackEmpty Undo Redo

8.10 Undo

---

bool sgCScene::Undo()

Description:
You can backtrack your recent commands created with the help of the command brackets StartUndoGroup - EndUndoGroup. *Do not call this function inside these command brackets.*

Arguments:
No arguments.

Returned value:
Returns false if the function fails and true otherwise.

See also:
GetScene StartUndoGroup EndUndoGroup IsUndoStackEmpty IsRedoStackEmpty Redo

8.11 Redo

---

bool sgCScene::Redo()

Description:
You can undo the backtracking of your recent commands created with the help of the command brackets StartUndoGroup - EndUndoGroup. *Do not call this function inside these command brackets.*

Arguments:
No arguments.

Returned value:
Returns false if the function fails and true otherwise.

See also:
GetScene StartUndoGroup EndUndoGroup IsUndoStackEmpty IsRedoStackEmpty Undo

8.12 Clear

---

void sgCScene::Clear()

Description:
Deletes all the scene objects. Clears the Undo-Redo history.

Arguments:
No arguments.

Returned value:
No values.

8.13 GetGabarits

---

void sgCScene::GetGabarits(SG_POINT& p_min, SG_POINT& p_max)

Description:
Returns the coordinates of the gabarit box of the scene.

Arguments:
p_min - takes the value of the gabarit box vertex with minimal coordinates.
p_max - takes the value of the gabarit box vertex with maximum coordinates.

Returned value:
No values.

9 Algorithms

---

sgCore library algorithms

Many mathematical algorithms to work with 3D and 2D objects are realized in the sgCore library.
The basic and the most important of them can create new objects from the existing ones.

In modern computer-aided design systems the main tools to create complex 3D objects are Boolean operations and operations creating solids and surfaces based on 2D objects.

The sgCore library offers the realization of most of these algorithms to developers.

The following algorithm groups are realized in the library:

Boolean operations
Kinematic operations
Creating 3D objects based on 2D objects

Read more about each algorithm group in the proper section.

9.1 sgBoolean

Boolean operations

Boolean operations is the means of creating new objects from existing ones by performing set-theoretic operations.

In the sgCore library all the functions realizing Boolean operations are collected in the same namespace - sgBoolean described in sgAlgs.h

You can construct many objects using Boolean operations. That's why in the sgCore library all the functions related to the Boolean algorythms group return pointers to the objects of the sgCGroup type.

The following algorythms belong to the Boolean algorythms group:

3D objects intersection
Joining 3D objects
Subtracting 3D objects
3D objects surfaces intersection
Object surface and plane intersection

9.1.1 Intersection

sgCGroup* sgBoolean::Intersection(const sgC3DObject& aOb, const sgC3DObject& bOb)

Description:

Creates a group of objects which are an intersection of two objects, i.e. all the objects with points lying both on the first and on the second object.

Arguments:

aOb - first object,
bOb - second object.

Returned value:

Returns the pointer to a group of objects which are an intersection of the arguments. If the objects are not intersecting or in case of an internal error the function returns NULL.

Note:

An internal error may occur in case you call this function for the surfaces or solids with self-intersections (these solids may be constructed, for example, by a number of kinematic operations).
The function for correcting self-intersections of solids is planned in the next library version.

See also:
sgCGroup EXAMPLE

9.1.2 Union

---

sgCGroup* sgBoolean::Union(const sgC3DObject& aOb, const sgC3DObject& bOb)

Description:

Creates a group of objects by joining two objects, i.e. all the objects with points lying on the first or on the second object.

Arguments:

aOb - first object,
bOb - second object.

Returned value:

Returns the pointer to a group of objects constructed by joining the arguments. If the objects are not intersecting or in case of an internal error the function returns NULL.

Note:

An internal error may occur in case you call this function for the surfaces or solids with self-intersections (these solids may be constructed, for example, by a number of kinematic operations).
The function for correcting self-intersections of solids is planned in the next library version.

See also:
sgCGroup ПРИМЕР

9.1.3 Sub

---

sgCGroup* sgBoolean::Sub(const sgC3DObject& aOb, const sgC3DObject& bOb)

Description:

Creates a group of objects by subtracting the second object from the first one, i.e. the objects with points lying on the first object but not on the second one.

Arguments:

aOb - first object,
bOb - second object.

Returned value:

Returns the pointer to a group of objects constructed by subtracting the second object from the first one. If the objects are not intersecting or in case of an internal error the function returns NULL.

Note:

An internal error may occur in case you call this function for the surfaces or solids with self-intersections (these solids may be constructed, for example, by a number of kinematic operations).
The function for correcting self-intersections of solids is planned in the next library version.

See also:
sgCGroup EXAMPLE

9.1.4 IntersectionContour

---

sgCGroup* sgBoolean::IntersectionContour(const sgC3DObject& aOb, const sgC3DObject& bOb)

Description:

Creates a group of line segments which are the intersection of two 3D objects surfaces, i.e. the line segments with the points lying both on the surface of the first and of the second object.

Arguments:

aOb - first object,
bOb - second object.

Returned value:

Returns the pointer to a group of line segments which are the intersection of the arguments surfaces. If the objects surfaces are not intersecting or in case of an internal error the function returns NULL.

Note:

An internal error may occur in case you call this function for the surfaces or solids with self-intersections (these solids may be constructed, for example, by a number of kinematic operations).
The function for correcting self-intersections of solids is planned in the next library version.

See also:
sgCGroup EXAMPLE

9.1.5 Section

---

sgCGroup* sgBoolean::Section(const sgC3DObject& obj, const SG_VECTOR& planeNormal, double planeD)

Description:

Creates a group of line segments which are the intersection of a 3D object surface and a plane, i.e. the line segments with all their points lying both on the surface of the first object and on the plane.

Arguments:

obj - 3D object,
planeNormal - normal to the plane,
planeD - free coefficient in the plane equation

Returned value:

Returns the pointer to a group of line segments which are the intersection of an object surface and a plane. If the object surface and the plane are not intersecting or in case of an internal error the function returns NULL.

Note:

An internal error may occur in case you call this function for the surfaces or solids with self-intersections (these solids may be constructed, for example, by a number of kinematic operations).
The function for correcting self-intersections of solids is planned in the next library version.

See also:
sgCGroup EXAMPLE

9.2 sgKinematic

Kinematic operations

Kinematic operations is the means of creating new objects by extuding a 2D object along a path in the three-dimensional space. Another 2D object can also be a similar path.
Let's call an extruded object a *clip*, and a path or an object the clip is moving along a *profile*.

In the sgCore library all the functions realizing kinematic operations are collected in the same namespace - sgKinematic described in sgAlgs.h

In most kinematic operations a clip as well as an array of closed holes (if the clip is closed) can be arguments. In this case the final object will have holes with the same profile as the clip.

If the clip isn't closed the created solid will always be a surface. But if the clip is closed you can create both a surface and a solid.

The following algorythms belong to the kinematic algorythms group:

Rotation
Extrusion
Spiral
Pipe-like

9.2.1 Rotation

---

sgCObject* sgKinematic::Rotation(const sgC2DObject& rotObj, const SG_POINT& axePnt1, const SG_POINT& axePnt2, double angl_degree, bool isClose, short meridians_count=24)

Description:

Creates a revolved solid or a surface by rotating the specified contour about an axis.

Arguments:

rotObj - rotated contour. Must be flat and not self-intersecting.
axePnt1 - first point on the rotation axis. Must lie on the rotated contour plane.
axePnt2 - second point on the rotation axis. Must lie on the rotated contour plane. If the rotated contour is linear the rotation axis line must not coincide with the contour line
angle_degree - the axis rotation angle (in degrees). Must not be equal to zero. The clockwise angle direction from axePnt1 to axePnt2 is positive. If the angle is less than -360 degrees, it is nevertheless -360, if more than 360 - is equal to 360.
isClose - whether to close to a solid. The argument is ignored if the rotated contour is not closed.
meridians_count - the number of meridians

Returned value:

Returns the pointer to a newly created object. If the function fails (the arguments don't meet the requirment) NULL is returned.

Note:

The programmer should check whether the rotation axis and the rotated contour are intersecting. The object will be created even if the rotation axis

and the rotated contour are intersecting but it will be self-intersecting, and in this case errors may occur when performing Boolean operations with this object. The function for correcting self-intersections of solids is planned in the next library version.

See also:
EXAMPLE of revolved surface EXAMPLE of revolved solid

9.2.2 Extrude

---

sgCObject* sgKinematic::Extrude(const sgC2DObject& outContour, const sgC2DObject** holes, int holes_count, const SG_VECTOR& extrDir, bool isClose)

Description:

Creates a solid or a surface of extrusion by extruding an object along a vector.

Arguments:

outContour - outer contour. Must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function).
holes - array of holes. The holes must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function). The holes must lie on the same plane as the outer contour (checked by the IsObjectsOnOnePlane function) and strictly inside it (checked by the IsFirstObjectInsideSeconfObject function). The holes must not be intersecting (checked by the IsObjectsIntersecting function) neither can they be inside each other (checked by the IsFirstObjectInsideSeconfObject function). If the argument is NULL the object is created withour holes.
holes_count - the number of holes in the holes array.
extrDir - extrusion vector. Mustn't be zero
isClose - whether to close to a solid. The argument is ignored if the outer contour isn't closed.

Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

See also:
EXAMPLE of surface of extrusion EXAMPLE of solid of extrusion

9.2.3 Spiral

---

sgCObject* sgKinematic::Spiral(const sgC2DObject& outContour, const sgC2DObject** holes, int holes_count, const SG_POINT& axePnt1, const SG_POINT& axePnt2, double screw_step, double screw_height, short meridians_count, bool isClose)

Description:

Creates a spiral solid or a surface of the specified clip with the specified number of holes and spiral parameters.

Arguments:

outContour - outer contour. Must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function).
holes - array of holes. The holes must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function). The holes must lie on the same plane as the outer contour (checked by the IsObjectsOnOnePlane function) and strictly inside it (checked by the IsFirstObjectInsideSeconfObject function). The holes must not be intersecting (checked by the IsObjectsIntersecting function) neither can they be inside each other (checked by the IsFirstObjectInsideSeconfObject function). If the argument is NULL the object is created withour holes.
holes_count - the number of holes in the holes array.

axePnt1 - first point on the spiral axis. Must lie on the outer contour plane.
axePnt2 - second point on the spiral axis. Must lie on the outer contour plane. If the outer contour is linear the rotation axis line must not coincide with the contour line
screw_step - spiral step length.
screw_height - spiral length
meridians_count - the number of meridians in a circle of one spiral step
isClose - whether to close to a solid. The argument is ignored if the outer contour isn't closed.

Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

Note:

The programmer should check whether the rotation axis and the outer contour are intersecting. The object will be created even if the axis and the contour are intersecting but it will be self-intersecting, and in this case errors may occur when performing Boolean operations with this object. The function for correcting self-intersections of solids is planned in the next library version.

See also:
EXAMPLE of surface EXAMPLE of solid

9.2.4 Pipe

---

sgCObject* sgKinematic::Pipe(const sgC2DObject& outContour, const sgC2DObject** holes, int holes_count, const sgC2DObject& guideContour, const SG_POINT& point_in_outContour_plane, double angle_around_point_in_outContour_plane, bool& isClose)

Description:

Creates a solid or a surface of extrusion by extruding an object along another object.

Arguments:

outContour - outer contour. Must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function).
holes - array of holes. The holes must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function). The holes must lie on the same plane as the outer contour (checked by the IsObjectsOnOnePlane function) and strictly inside it (checked by the IsFirstObjectInsideSeconfObject function). The holes must not be intersecting (checked by the IsObjectsIntersecting function) neither can they be inside each other (checked by the IsFirstObjectInsideSeconfObject function). If the argument is NULL the object is created withour holes.
holes_count - the number of holes in the holes array.

guideContour - object along which the outer contour and holes are extruded.
point_in_outContour_plane - a point in the outer contour plane which is moving along the pipe profile
angle_around_point_in_outContourPlane - angle to turn the outer contour to

and the holes around the point moving along the profile
isClose - whether to close to a solid. The argument is ignored if the outer contour isn't closed.

Returned value:
Returns the pointer to a newly created object. If the function fails NULL is returned.

Note:
It is highly possible that as a result of the operation a self-intersecting object will be created, and in this case errors may occur when performing Boolean operations with this object. The function for correcting self-intersections of solids is planned in the next library version.

See also:
EXAMPLE of surface EXAMPLE of solid

9.3 sgSurfaces

Constructing 3D objects from 2D objects

The sgCore library realizes a number of function to create 3D surfaces and solids from the existing 2D objects. All the functions realizing such operations are collected in the same namespace - sgSurfaces described in sgAlgs.h

The following algorythms belong to this group:
Constructing flat face with holes
Coons surfaces
Constructing surface by specified points array
Sewing surfaces trying to create solid
Constructing ruled surface or solid from two clips
Constructing smooth surface or solid from clips

9.3.1 Face

---

sgCObject* sgSurfaces::Face(const sgC2DObject& outContour, const sgC2DObject** holes, int holes_count)

Description:

Creates a flat face from an outer contour and an array of holes. It can be also considered as the realization of Delone triangulation of a flat area with holes.

Arguments:

outContour - outer contour. Must be flat (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function).
holes - array of holes. The holes must be flat (checked by the IsPlane function), (checked by the IsPlane function), closed (checked by the IsClosed function) and not self-intersecting (checked by the IsSelfIntersecting function). The holes must lie on the same plane as the outer contour (checked by the IsObjectsOnOnePlane function) and strictly inside it (checked by the IsFirstObjectInsideSeconfObject function). The holes must not be intersecting (checked by the IsObjectsIntersecting function) neither can they be inside each other (checked by the IsFirstObjectInsideSeconfObject function). If the argument is NULL the face is created withour holes.
holes_count - the number of holes in the holes array.

Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

See also:
sgC2DObject EXAMPLE

9.3.2 Coons

---

sgCObject* sgSurfaces::Coons(const sgC2DObject& firstSide, const sgC2DObject& secondSide, const sgC2DObject& thirdSide, const sgC2DObject* fourthSide)

Description:

Creates a Coons surface from three or four boudary contours. *The contours must be not closed and not self-intersecting. The contours end points must coincide with the end points of the neighboring contours, i.e. form a closed area.*

Arguments:

firstSide - first boundary contour.
secondSide - second boundary contour. An end point must coincide with an end point of the previous argument - firstSide.
thirdSide - second boundary contour. An end point must coincide with the end point of the previous contour (secondSide) which doesn't coincide with the end point of the firstSide. If the next argument (fourthSude) is NULL, the

second end point of thirdSide must coincide with a free end point of the first contour.

fourthSide - second boundary contour. May be NULL. One end point must coincide with the end point of the previous contour (thirdSide), and another one - with the free end point of the first contour.

Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

See also:
sgC2DObject EXAMPLE WITH THREE CONTOURS EXAMPLE WITH FOUR CONTOURS

9.3.3 Mesh

---

sgCObject* sgSurfaces::Mesh(short dimens_1, short dimens_2, const SG_POINT* pnts)

Description:

Creates a mesh from an array of the points.

Arguments:

dimens_1 - first dimension of the pnts array,
dimens_2 - second dimension of the pnts array,
pnts - array of the points the mesh is constructed from.

Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

See also:
sgC2DObject ПРИМЕР

9.3.4 SewSurfaces

---

sgCObject* sgSurfaces::SewSurfaces(const sgC3DObject** surfaces, int surf_count)

Description:

Sews a number of surfaces into a single surface trying to create a solid if the surfaces limit the closed area. The surfaces must side with each other. If they

don't a group of objects is created from the arguments.

Arguments:

surfaces - array of the pointers to sewn surfaces,
surf_count - the number of sewn surfaces in the surfaces array.

Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

9.3.5 LinearSurfaceFromSections

---

sgCObject* sgSurfaces::LinearSurfaceFromSections(const sgC2DObject& firstSide, const sgC2DObject& secondSide, double firstParam, bool isClose)

Description:

Creates a surface or a solid by moving the end points of a line segment along two different curves.

Arguments:

firstSide - path curve for the first end point of the ruling line segment,
secondSide - path curve for the second end point of the ruling line segment,
firstParam - parameter of the point on the first path curve connecting with the start point on the second path curve; this parameter defines the spiral degree of the new object and takes the value from 0 to 1. The point on the object can be found using the GetPointFromCoefficient() function. This parameter is ignored if the firstSide object is not closed .
isClose - whether to close to a solid in case firstSide and secondSide are flat and closed curves without self-intersections.

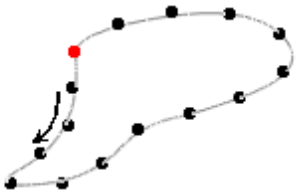
Returned value:

Returns the pointer to a newly created object. If the function fails NULL is returned.

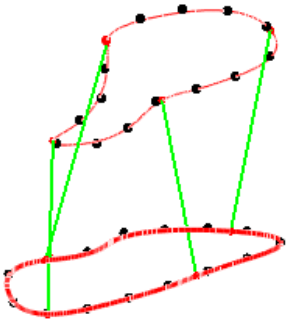
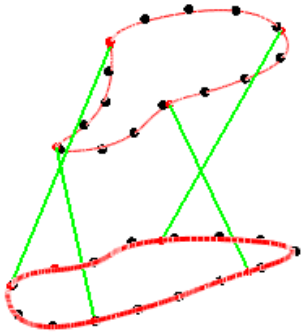
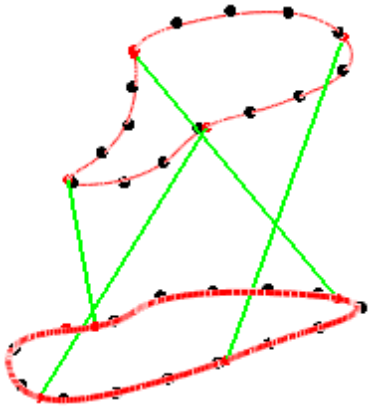
Explanation:

While constructing objects using this function special attention should be paid to the start parameter of the first curve and to orientation (can be changed using ChangeOrient()) of both the curves. Let's illustrate it with examples.

We have two closed curves which can be used as path curves to create a surface:

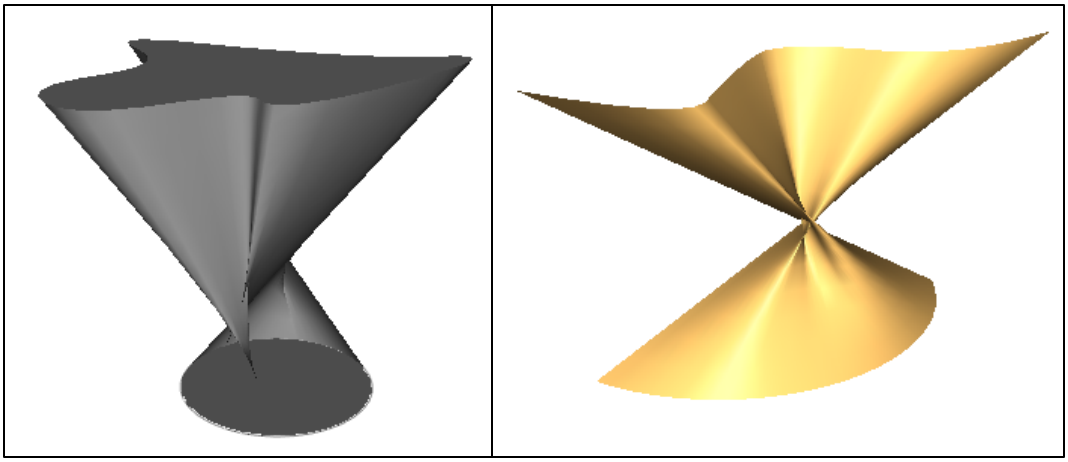


The start points of both curves are indicated red and objects orientation as arrows.
Below you can see different cases of various spirals of obtained surfaces with different initial parameters of the first curve

firsParam = 0	fistParam = 0.12	firstParam = 0.75
		

It's obvious that the objects will have different spiral degrees.

Special attention should be paid to the objects orientation. A new object is created by moving the end point of a line segment along two path curves. But along each curve each end point moves in the curve direction. You can change the object direction round using the ChangeOrient() function.
The wrong orientation may cause such results:



In case of unclosed objects only objects orientation is considered.

See also:
GetPointFromCoeficient() ChangeOrient()

9.3.6 SplineSurfaceFromSections

---

sgCObject* sgSurfaces::SplineSurfaceFromSections(const sgC2DObject** sections, const double* params, int sections_count, bool isClose)

Description:

Creates a smoothed surface or a solid from three or more clips. To draw an object from two clips you can use the LinearSurfaceFromSections() function.

Arguments:

sections - array of clips,
params - array of the parameters of the base line points on which a new object is to be created; this parameter defines the spiral degree of the new object and takes the value from 0 to 1. The point on the object can be found using the GetPointFromCoefficient() function. This parameter is ignored if the corresponding object from the section array is not closed.
isClose - whether to close to a solid in case firstSide and secondSide are flat and closed curves without self-intersections.

Returned value:

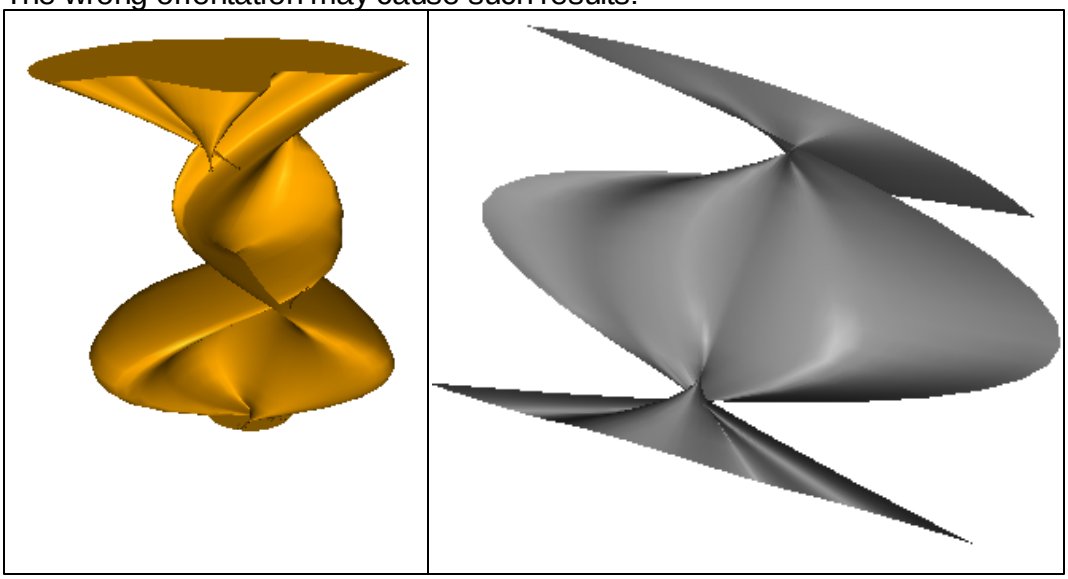
Returns the pointer to a newly created object. If the function fails NULL is returned.

Explanation:

See more about curve points parameters values in the description of the

LinearSurfaceFromSections() function.

Special attention should be paid to the objects orientation. A new object is created by moving the end point of a line segment along two path curves. But along each curve each end point moves in the curve direction. You can change the object direction round using the ChangeOrient() function.
The wrong orientation may cause such results:



In case of unclosed objects only objects orientation is considered.

See also:
GetPointFromCoeficient() ChangeOrient()

10 sgCFont

sgCFont

sgCFont is the representative of the font concept. A font can be loaded from a file with .SHX extension - it's an AutoCAD font format. After loading the font you can get a structure describing the font. After finishing work with the font you have to unload it.

Defined in sgTD.h

The sgCFont class methods:
- LoadFont
- UnloadFont
- GetFontData

10.1 LoadFont

---

static sgCFont* sgCFont::LoadFont(const char* path, char* comment, short commentLength)

Description:

Creates a "font" class object by an .SHX file.

Arguments:

path - full path to the font file
comment - buffer to copy a comment from an .SHX file to
commentLength - length of the buffer

Returned value:

Returns the pointer to the loaded font. If the function fails NULL is returned.

See also:
UnloadFont GetFontData

10.2 UnloadFont

---

static bool sgCFont::UnloadFont(sgCFont* fnt)

Description:

Unloads the font from the memory and destroys the font class object. To avoid memory leakage you must call this function when finishes the work with the font.

Arguments:

fnt - pointer to the sgCFont class object you want to delete.

Returned value:

Returns true, if the font was successfully unloaded and false otherwise.

See also:
LoadFont GetFontData

10.3 GetFontData

---

const SG_FONT_DATA* const sgCFont::GetFontData() const

Description:

Returns the pointer to the font description structure.

Arguments:

No arguments.

Returned value:

Returns the pointer to the font description structure - SG_FONT_DATA.
This structure has the following description:

```
typedef struct
{
        char            name[15];
        unsigned short  table_begin;
        unsigned short  table_end;
        unsigned short  table_size;
        unsigned char   posit_size;
        unsigned char   negat_size;
        unsigned char   state;
        double          proportion;
        unsigned char   symbols_table[1];
} SG_FONT_DATA;
```

name - font name (name of the file the font was loaded from)
table_begin - code of the first font symbol
table_end - code of the last font symbol
table_size - the number of the font characters
posit_size - size above the zero level
negat_size - size under the zero level
state - whether the vertical font direction is allowed
proportion - average symbol length to the symbol heightratio
symbols_table - beginning of the symbols code table

The style of symbols in the loaded font can be defined by drawing (the Draw function) the text or by creating (the Create function) a text object

See also:
LoadFont UnloadFont

11 sgFontManager

sgFontManager

sgFontManager is a namespace of the function which adds font to a scene and removes the loaded ones from a scene. The pointer to a scene is returned by the sgCScene::GetScene() function. The current font concept is defined in a scene.

Defined in sgTD.h

The sgFontManager namespace functions:

- AttachFont
- GetFontsCount
- GetFont
- SetCurrentFont
- GetCurrentFont

11.1 AttachFont

static bool sgFontManager::AttachFont(sgCFont* fnt)

Description:

Attaches a font to a scene. The attachable font gets its personal order number according to the number it was attached to a scene. You can work with this font by

Arguments:

fnt - font you want to attach.

Returned value:

Returns false if the font has been already attached to the scene, otherwise true

See also:
LoadFont

11.2 GetFontsCount

---

static unsigned int sgFontManager::GetFontsCount()

Description:
Returns the number of the attached to the scene fonts.

Arguments:
No arguments.

Returned value:
Returns the number of the attached to the scene fonts.

See also:
AttachFont

11.3 GetFont

---

static const sgCFont* sgFontManager::GetFont(unsigned int nmbr)

Description:
Returns the attached to the scene font by its number.

Arguments:
nmbr - font number.

Returned value:
Returns the pointer to the attached to the scene font by its order number.

See also:
AttachFont

11.4 SetCurrentFont

---

static bool sgFontManager::SetCurrentFont(unsigned int nmbr)

Description:

Sets the current font by the number.

Arguments:

nmbr - number of the font you want to do the current.

Returned value:

Returns false if there is no font with this number or in case of failure, otherwise true. You can get the font itself using the GetFont function

See also:
AttachFont

11.5 GetCurrentFont

---

static unsigned int sgFontManager::GetCurrentFont()

Description:

Returns the number of the current font.

Arguments:

No arguments.

Returned value:

Returns the number of the current font. You can get the font itself using the GetFont function

See also:
AttachFont

12 sgCText

sgCText

sgCText is the representative of the TEXT concept. A font can be loaded from a file with .SHX extension - it's an AutoCAD font format. After loading the font you can get a structure describing the font. After finishing work with the font you have to unload it.

The sgCText class methods:

12.1 Create

static sgCText* sgCText::Create(const sgCFont* fnt, const SG_TEXT_STYLE& stl,const char* string)

Description:

Creates an object of the TEXT class with the specified font and style. The text object is created in the XOY plane so that the coordinates origin coincides with the bottom left corner of the first text character.

Arguments:

fnt - pointer to the font (read more about fonts - sgCFont),
stl - text style structure.

This structure is described in sgTD.h and has the following fields:

```
typedef struct
{
    unsigned char  state;
    double         height;
    double         proportions;
    double         angle;
```

```
        double          horiz_space_proportion;
        double          vert_space_proportion;
    } SG_TEXT_STYLE;
```

state - text vertical status. If this field value is SG_TEXT_VERTICAL the text direction is vertical, otherwise horizontal.
height - character height
proportions - width to height ratio
angle - characters slope angle (in degrees)
horiz_space_proportion - horizontal character interval (in %)
vert_space_proportion - vertical character interval (in %)

string - text string itself

Returned value:

Returns the pointer to the created object. If the function fails NULL is returned.

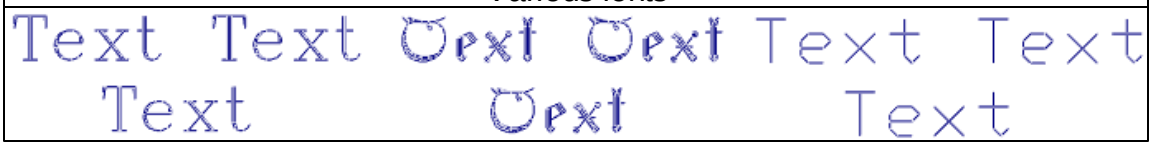
Following shortening is defined:

```
#define    sgCreateText  sgCText::Create
```

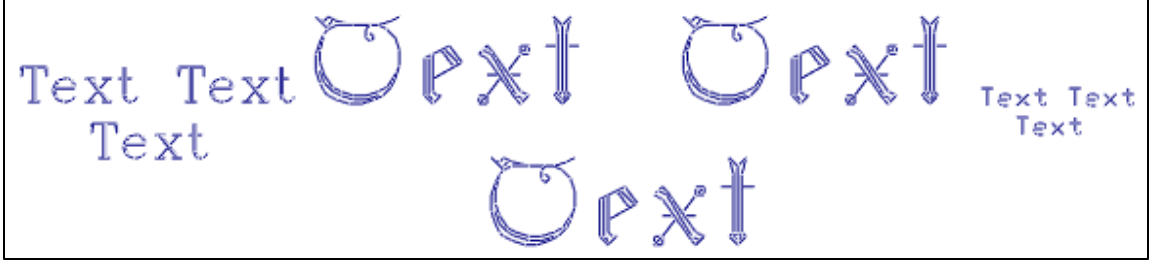
Explanation:

Below you can see illustrations of text objects with various parameters:

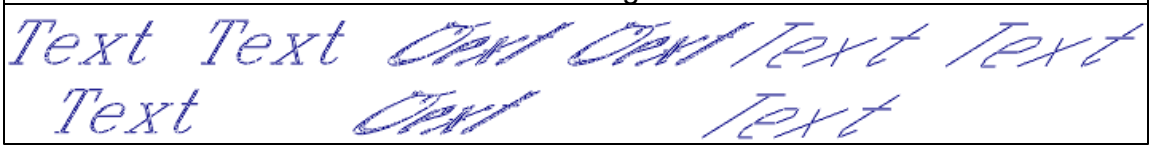
Various fonts



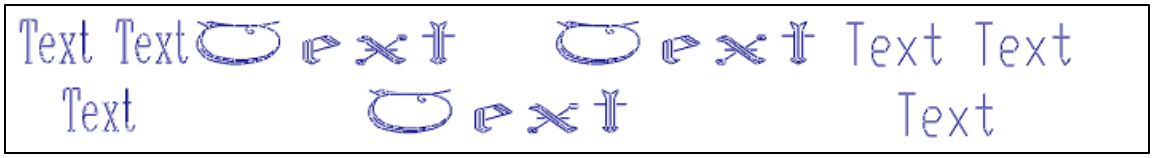
Various dimensions



Various angles



Various ratio



See also:
Objects hierarchy sgCObject methods

12.2 Draw

---

static bool sgCText::Draw(const sgCFont* fnt, const SG_TEXT_STYLE& stl, sgCMatrix* mtrx, const char* string, SG_DRAW_LINE_FUNC line_func)

Description:

Draws the text without creating a text object. By default the string is drawn on the XOY plane so that the coordinates origin coincides with the bottom left corner of the first text character. To draw the text in the arbitrary position you have to set an affine transformation matrix.

Arguments:

fnt - pointer to the font (read more about fonts - sgCFont),
stl - text style structure (read more about text style structure - here)
mtrx - affine transformation matrix transforming the string from the XOY plane into the user-defined position. If the argument is NULL the string is drawn on the XOY plane so that the coordinates origin coincides with the bottom left corner of the first text character..
string - text string itself
line_func - pointer to the function to be called for each line segment of the text style

Returned value:

Returns false if the function fails, otherwise true.

See also:
Create SG_DRAW_LINE_FUNC

12.3 GetLines

---

const SG_LINE* sgCText::GetLines() const

Description:

Returns the pointer to the array of the line segments of the text object segment presentation (the SG_LINE description). The number of line segments in this array is returned by the GetLinesCount() function

Arguments:

No arguments.

Returned value:

Returns the pointer to the array of the line segments of the text object segment presentation.

See also:
sgCText::Create sgCText::GetLinesCount SG_LINE

12.4 GetLinesCount

---

int sgCText::GetLinesCount() const

Description:

Returns the number of line segments in the text object segment presentation. The array itself is returned by the GetLines function

Arguments:

No arguments.

Returned value:

Returns the number of line segments in the text object segment presentation.

See also:
sgCText::Create sgCText::GetLines

12.5 GetStyle

SG_TEXT_STYLE sgCText::GetStyle() const

Description:
Returns a style of a text object.

Arguments:
No arguments.

Returned value:
Returns a style of a text object (more about the style structure here)

See also:
sgCText::Create sgCText::Draw

12.6 GetText

---

const char* sgCText::GetText()

Description:
Returns a text of a text object.

Arguments:
No arguments.

Returned value:
Returns a text of a text object.

See also:
sgCText::Create sgCText::Draw

12.7 GetFont

unsigned int sgCText::GetFont()

Description:

Returns the number of the text object font. The font itself is returned by the sgFontManager::GetFont() function

Arguments:

No arguments.

Returned value:

Returns the number of the text object font.

See also:
sgCText::Create sgCText::Draw sgFontManager::GetFont()

12.8 GetWorldMatrix

bool sgCText::GetWorldMatrix(sgCMatrix& mtrx) const

Description:

Returns the matrix of the text object position. If the matrix is unit the text object is located on the XOY plane so that the coordinates origin coincides with the bottom left corner of the first text character.

Arguments:

mtrx - returned matrix of the text object position.

Returned value:

Returns false if the function fails, otherwise true.

See also:
sgCText::Create sgCText::Draw sgCMatrix

13 sgCDimensions

sgCDimensions

sgCDimensions is the presentation of the DIMENSIONAL OBJECT concept. You can use dimensional objects to construct drawings and to specify a distance between points, angles, radii and diameters.

- distant
- angular
- radial
- diametral

The sgCore library supports all the four dimensional objects types. And when creating the object (or drawing without creating) you can specify a few parameters.

The sgCDimensions class methods:

Create
Draw
GetLines
GetLinesCount
GetFormedPoints
GetType
GetStyle
GetText
GetFont

13.1 Create

static sgCDimensions* sgCDimensions::Create (SG_DIMENSION_TYPE dimType, const SG_POINT* formed_points, const sgCFont* fnt, const SG_DIMENSION_STYLE& stl,const char* string)

Description:

Creates an object of the DIMENSIONS class of the specified type and with the specified font and style.

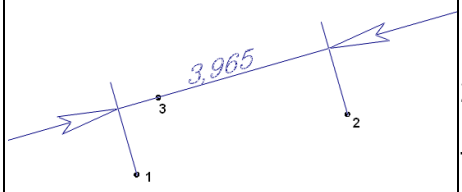
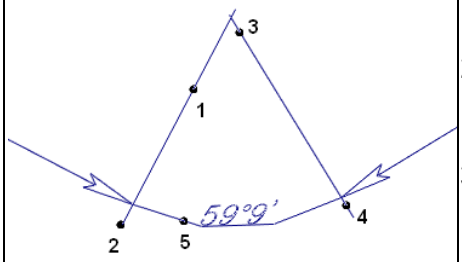
Arguments:

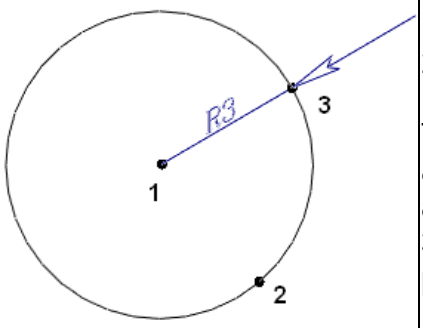
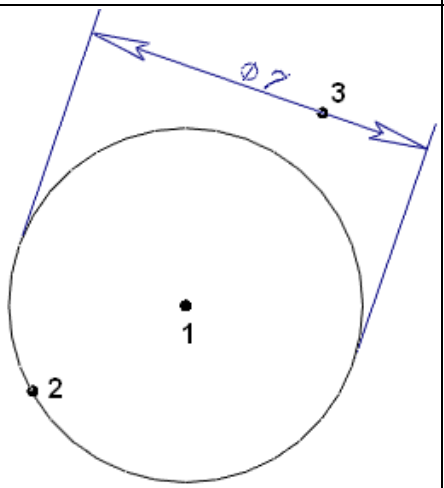
dimType - type of a dimensional object.

Can take the following values:

- SG_DT_LINEAR - linear dimension
- SG_DT_ANGLE - angle dimension
- SG_DT_RAD - radial dimension
- SG_DT_DIAM - diametral dimension

formed_points - array of the distinguished point the dimension is built on. These points has the following meanings for various dimensions types:
The dimension itself is calculated as the distance between the first and the second points.

Dimension type	Distinguished points	Explanation	Necessary number
Distant (linear)		1. The first point is where the start dimension point is. 2. The second point is where the end dimension point is. The dimension itself is calculated as the distance between the first and the second points. 3. The third point is an arbitrary point on the dimensional line.	3
Angle		1. The first point is the first one lying on the first side of the measured angle. 2. The second point is the second one lying on the first side of the measured angle. 3. The third point is the first one lying on the second side of the measured angle. 4. The forth point is the second one lying on the second side of the measured angle. The dimension itself is calculated as the angle between two these line segments. 5. The fifth point is an arbitrary point on the dimensional line.	5

Radial		1. The first point is the center of the measured object. 2. The second point is a point on the object. The dimension itself is calculated as the distance between the first and the second points. 3. The third point is an arbitrary point on the dimensional line.	3
Diametral		1. The first point is the center of the measured object. 2. The second point is a point on the object. The dimension itself is calculated as the double distance between the first and the second points. 3. The third point is an arbitrary point on the dimensional line.	3

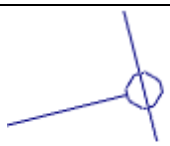
The number of points for each object type given in the table above is a requirment for creating an object.

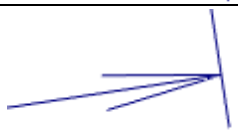
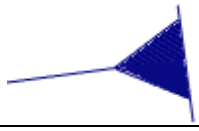
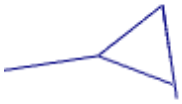
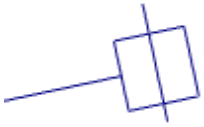
fnt - pointer to a font (read more about fonts - sgCFont) to write the text with
stl - dimension style structure.
This structure is described in sgTD.h and has the following fields:

```
typedef struct
{
    bool            dimension_line;
    bool            first_side_line;
    bool            second_side_line;
    double          lug_size;

    bool            automatic_arrows;

    bool            out_first_arrow;
    unsigned char   first_arrow_style;

    bool            out_second_arrow;
    unsigned char   second_arrow_style;

    double          arrows_size;

    SG_TEXT_ALIGN   text_align;
```

```
    SG_TEXT_STYLE     text_style;
    bool              invert;
    SG_DIMENSION_BEHAVIOUR   behaviour_type;
    unsigned short  precision;
} SG_DIMENSION_STYLE;
```

dimension_line - sign showing that the dimensional line is present (above the text line)
first_side_line - sign showing that the first extension line is present
second_side_line - sign showing that the second extension line is present
lug_size - lug size of extension lines above arrows
automatic_arrows - sing of arrows position automatic calculation - inside or outside the dimension. If this structure field value is true, the arrows position is calculated automatically depending of the text length and arrows size. In this case the out_first_arrow and out_second_arrow fields are ignored.
out_first_arrow - whether the first arrow is outside the dimension (if the automatic_arrows field value is true the field is ignored)
out_second_arrow - whether the second arrow is outside the dimension (if the automatic_arrows field value is true the field is ignored)
first_arrow_style and second_arrow_style - arrows style.
The following styles are supported:

Field value	Arrow type
0	
1	
2	
3	

4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

arrows_size - arrows size
text_align - text string position type on the dimensional line. Can have the following values:
SG_TA_CENTER - center text position,
SG_TA_LEFT - left text position,
SG_TA_RIGHT - right text position.
text_style - text style (read more about text style structure here)
invert - is necessary for angular dimensions (ignored in other cases). If the field value is true, the angle is added to 360 degrees
behaviour_type - is necessary for distant dimensions (ignored in other cases) . Specifies the way of building the dimension by the third point (see the values table - formed_points). Has the following values:
SG_DBT_VERTICAL - dimensional line is construced with the vertical segment line projection between first two points
SG_DBT_HORIZONTAL - dimensional line is construced with the horizontal segment line projection between the first two points
SG_DBT_PARALLEL - dimensional line is constructed parallel to the segment line between the first two points
SG_DBT_SLANT - dimensional line is constructed parallel to the segment line between the first two points and the third point is its start (slanting dimensional line)
SG_DBT_OPTIMAL - optimum dimension is constructed between two points considering the third one
precision - dimensional string display accuracy (characters number after the point)
string - text string to be written on the dimensional line. Can be NULL - in this case the dimension value is written on the dimensional line with the accuracy specified in the corresponding field of the stl structure

Returned value:
Returns the pointer to the created object. If the function fails NULL is returned.

See also:
Objects hierarchy sgCObject methods

13.2 Draw

static bool sgCText::Draw(SG_DIMENSION_TYPE dimType, const SG_POINT* formed_points, const sgCFont* fnt, const SG_DIMENSION_STYLE& stl, const char* string, SG_DRAW_LINE_FUNC line_func)

Description:

Draws dimensions without creating a dimensional object.

Arguments:

dimType - dimensional object type (more about dimensional types here)
formed_points - array of forming point the dimension is constructed from (more about forming points here)
fnt - pointer to the font (more about fonts - sgCFont) to write the text with
stl - dimenstion style structure (more about dimension style here)
string - text string to be written on a dimensional line. Can be NULL - in this case the dimension value is written on the dimensional line with accuracy specified in the corresponding field of the stl structure
line_func - pointer to the function which is called for each line segment of the dimension drawing

Returned value:

Returns false if the function fails, otherwise true.

See also:
Create SG_DRAW_LINE_FUNC

13.3 GetLines

---

const SG_LINE* sgCDimensions::GetLines() const

Description:

Returns the pointer to the array of the line segments of the dimensional object segment presentation (the SG_LINE description). The number of line segments in this array is returned by the GetLinesCount() function

Arguments:

No arguments.

Returned value:

Returns the pointer to the array of the line segments of the dimensional object segment presentation.

See also:
sgCDimensions::Create sgCDimensions::GetLinesCount SG_LINE

13.4 GetLinesCount

int sgCDimensions::GetLinesCount() const

Description:

Returns the number of line segments in the dimensional object segment presentation. The array itself is returned by the GetLines function

Arguments:

No arguments.

Returned value:

Returns the number of line segments in the dimensional object segment presentation.

See also:
sgCDimensions::Create sgCDimensions::GetLines

13.5 GetFormedPoints

const SG_POINT* sgCDimensions::GetFormedPoints() const

Description:

Returns the pointer to the array of the dimensional object forming points.

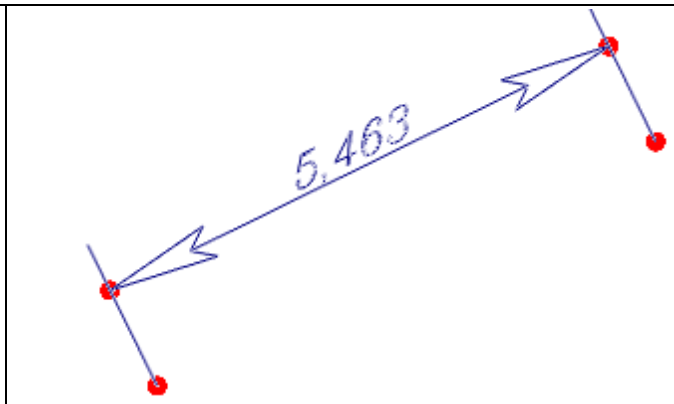
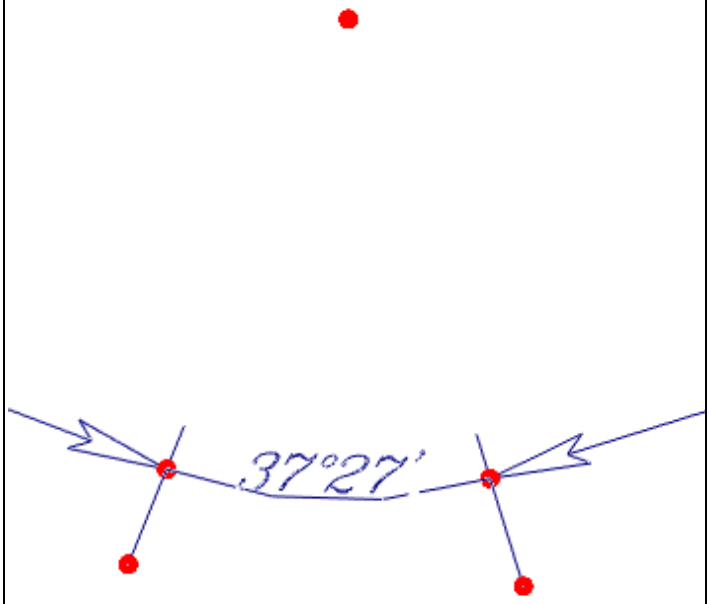
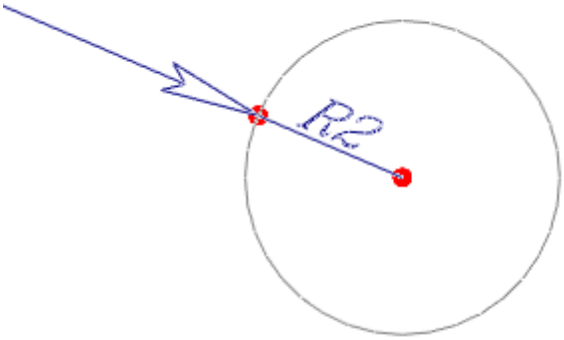
Arguments:

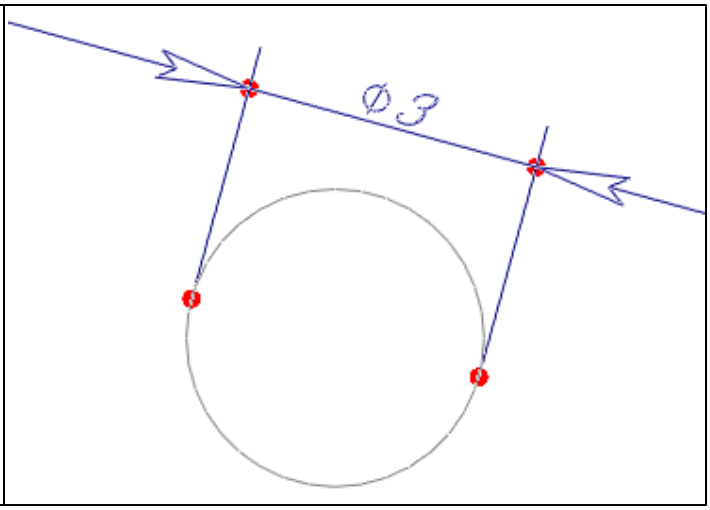
No arguments.

Returned value:

Forming points for different dimensional object types are marked red on the pictures below:

Dimensional object type	Forming points

Distant (linear)	
Angular	
Radial	

Diametral	

See also:
sgCDimensions::Create sgCDimensions::GetLinesCount

13.6 GetType

SG_DIMENSION_TYPE sgCDimensions::GetType() const

Description:
Returns the type of a dimensional object.

Arguments:
No arguments.

Returned value:
Returns the type of a dimensional object (read more about dimension types here).

See also:
sgCDimensions::Create sgCDimensions::Draw

13.7 GetStyle

SG_DIMENSION_STYLE sgCDimensions::GetStyle() const

Description:
Returns the style of a dimensional object.

Arguments:
No arguments.

Returned value:
Returns the style of a dimensional object (read more about style structure here).

See also:
sgCDimensions::Create sgCDimensions::Draw

13.8 GetText

---

const char* sgCDimensions::GetText()

Description:
Returns the text of a dimensional object.

Arguments:
No arguments.

Returned value:
Returns the text of a dimensional object.

See also:
sgCDimensions::Create sgCDimensions::Draw

13.9 GetFont

---

unsigned int sgCDimensions::GetFont()

Description:

Returns the number of the dimensional object font. The font itself is returned by the sgFontManager::GetFont() function

Arguments:

No arguments.

Returned value:

Returns the number of the dimensional object font.

See also:
sgCDimensions::Create sgCDimensions::Draw sgFontManager::GetFont()

14 sgFileManager

sgFileManager

sgFileManager is a namespace of the functions to save and to load the whole scene and separate objects to files. Besides, it is responsible for either import and export to other formats.

You can save your scene to a file and then load it from the file in the following two ways:

- the first way considers the geometry repetitions, links to font files and to text objects strings - that is why the file itself will be of the minimum size (as far as possible). The structure of this file is the following:
  1. A standard header.
  2. A user-defined block of data of an arbitrary size.
  3. The scene itself saved in the optimal format.

  The following functions are responsible for scene saving and loading using the current way:
  Save
  GetFileHeader
  GetUserData
  Open

- The objects saving and loading in the second way is totally the responsibility of the programmer, i.e. he himself develops the file structure. The sgCore library presents two functions - converting an object to a memory block and creating an

object from a memory block. I.e. it is for the programmer to decide where to save an object in the file and how to load an object from the file.

The following functions are responsible for scene saving and loading using the current way:
ObjectToBitArray
BitArrayToObject

The sgFileManager namespace is defined in sgFileManager.h

The sgFileManager namespace functions:
Save
GetFileHeader
GetUserData
Open
ExportDXF
ImportDXF
ObjectToBitArray
BitArrayToObject

14.1 Save

---

bool sgFileManager::Save(const sgCScene* scen, const char* file_name,
const void* userData, unsigned long userDataSize)

Description:
Saves a scene to the file of the specified structure with user-defined block of data (read more how to save the scene here).

Arguments:
scen - scene to be saved
file_name - full path to a file the scene will be saved to
userData - user-defined block of data which is saved right after the header in the file (can be NULL)
userDataSize - user-defined block of data size (ignored if userData is NULL)

Returned value:

Returns true if the scene was successfully saved, otherwise - false

See also:
GetFileHeader GetUserData Open

14.2 GetFileHeader

---

bool sgFileManager::GetFileHeader(const char* file_name, SG_FILE_HEADER& file_header)

Description:
Returns the file header structure (read more how to save the scene here).
Arguments:
file_name - full path to a file the scene will be saved to
file_header - returned file header structure.
This structure has the following description

```
typedef struct
{
        char            signature[5];
        int             major_ver;
        int             minor_ver;
        unsigned long   userBlockSize;
} SG_FILE_HEADER;
```

signature - standard file signature (defined through SG_FILE_SIGNATURE)
major_ver - major kernel version the file was created by (more about the version - sgGetVerson)
minor_ver - minor kernel version the file was created by (more about the version - sgGetVerson)
userBlockSize - size of user-defined block of data which is saved right after the header in the file. You can get the user-defined block of data itself using the GetUserData function.

Returned value:
Returns true if the specified file is a file with sgCore objects, otherwise false

See also:
GetUserData

14.3 GetUserData

bool sgFileManager::GetUserData(const char* file_name, void* usetData)

Description:

Returns the user-defined block of data from the file with the sgCore saved scene (read more how to save the scene here).

Arguments:

file_name - full path to a file with the scene
userData - user-defined block of data read from the file. The user must allocate the memory for this block of data (you can find out the size of the block using the GetFileHeader() function). This block should be released also by the user.

Returned value:

Returns true if the specified file is a file with sgCore objects and the user block of data was read, otherwise false

See also:
GetFileHeader

14.4 Open

bool sgFileManager::Open(const sgCScene* scen, const char* file_name)

Description:

Loads objects from a file and attaches them to the specified scene (read more how to save the scene here). In this case the user-defined block of data is ignored (you can get it using the GetFileHeader() and GetUserData() functions)

Arguments:

scen - scene the loaded from the file object are attached to (the result of calling the sgCScene::GetScene() function)
file_name - full path to a file the scene will be saved to

Returned value:

Returns true in case of successful loading, otherwise - false

See also:
GetFileHeader GetUserData Save

14.5 ExportDXF

---

bool sgFileManager::ExportDXF(const sgCScene* scen, const char* file_name)

Description:

Exports scene objects to the .DXF format file.

Arguments:

scen - scene to be exported (the result of calling the sgCScene::GetScene() function)
file_name - full path to a .DXF file

Returned value:

Returns true if the scene was successfully saved, otherwise - false

See also:
ImportDXF

14.6 ImportDXF

bool sgFileManager::ImportDXF(const sgCScene* scen, const char* file_name)

Description:

Imports scene objects from the .DXF format file and attaches them to the specified scene.

Arguments:

scen - scene to attach the imported from the file objects to (the result of calling the sgCScene::GetScene() function)
file_name - full path to the .DXF file

Returned value:

Returns true if the scene was successfully loaded, otherwise - false

See also:
ExportDXF

14.7 ExportSTL

bool sgFileManager::ExportSTL(const sgCScene* scen, const char* file_name)

Description:

Exports scene objects to the .STL format file.

Arguments:

scen - scene to be exported (the result of calling the sgCScene::GetScene() function)
file_name - full path to a .STL file

Returned value:

Returns true if the scene was successfully saved, otherwise - false

See also:
ImportSTL

14.8 ImportSTL

---

bool sgFileManager::ImportSTL(const sgCScene* scen, const char* file_name)

Description:

Imports scene objects from the .STL format file and attaches them to the specified scene.

Arguments:

scen - scene to attach the imported from the file objects to (the result of calling the sgCScene::GetScene() function)
file_name - full path to the .STL file

Returned value:

Returns true if the scene was successfully loaded, otherwise - false

See also:
ExportSTL

14.9 ObjectToBitArray

---

const void* sgFileManager::ObjectToBitArray(const sgCObject* obj, unsigned long& arrSize)

Description:

Converts the object to a binary array in order to realize the second way of saving objects (read more how to save the scene here).

Arguments:

obj - object to be converted to a binary array
arrSize - size of the obtained binary array

Returned value:

Returns the pointer to the beginning of the binary array (the size of this array is returned by arrSize). If this array is supposed to be used after the object is destroyed you should copy this array to your block of memory, as after the object is destroyed the pointer returned by the current function becomes

invalid.
If the convertion fails NULL is returned.

See also:
BitArrayToObject

14.10 BitArrayToObject

sgCObject* sgFileManager::BitArrayToObject(const void* bitArray, unsigned long arrSize)

Description:
Creates an object by a binary array in order to realize the second way of saving objects (read more how to save the scene here).

Arguments:
bitArray - binary array on which the object is created
arrSize - size of the binary array

Returned value:
Returns the pointer to the created object. If the function fails NULL is returned.

See also:
ObjectToBitArray

14.11 ObjectFromTriangles

sgCObject* sgFileManager::ObjectFromTriangles(const SG_POINT* vertexes, long vert_count,
const SG_INDEX_TRIANGLE* triangles, long tr_count,
float smooth_angle_in_radians=30.0*3.14159265/180.0,
bool solids_checking=false)

Description:

Creating objects by an array of indexed triangles.

Arguments:

vertexes - the array of all vertices in triangles

triangles - the array of structures describing triangles -

SG_INDEX_TRIANGLE (Definition in sg3D.h):

```
typedef struct
{
  long   ver_indexes[3];
} SG_INDEX_TRIANGLE;
```

this structure is a triple of indexes of vertices in a triangle from the array vertexes

tr_count - number of triangles

smooth_angle_in_radians - the maximal angle (in radians) between the faces of the object, less than which the normal lines of triangles will be automatically smoothed. If the angle between the faces is larger than smooth_angle_in_radians, normal lines will not be smoothed. By default, this argument equals 30 degrees

solids_checking - checking the array of triangles for a closure - that is, whether a solid can be created. If you are sure that the triangles form a solid or a surface, it is better to set this argument to false (the default value) because this check may considerably slow down the function. If triangles are positioned so that a vertex of one triangle belongs to an edge of another triangle and if this argument is set to true, the triangles will be automatically split in such a way that no vertex of one triangle will belong to an edge of another triangle

Returned value:

Returns the pointer to the created object. NULL is returned in case of a failure. If the triangles were connected, an object of the sgC3DObject type will be created. If the triangles were split into unconnected groups in space, an object of the sgCGroup type will be created

Example (a group out of a cube and a triangle):

```
const SG_POINT vert[] = {
    {0.00000, 0.00000, 0.000000},
    {0.00000, 1.50000, 1.500000},
    {0.00000, 0.00000, 2.000000},
    {0.00000, 2.00000, 0.000000},
    {0.00000, 0.50000, 0.500000},
    {0.00000, 2.00000, 2.000000},
    {2.00000, 0.00000, 0.000000},
    {2.00000, 0.00000, 2.000000},
    {2.00000, 2.00000, 0.000000},
    {2.00000, 2.00000, 2.000000},

    {10.0000, 10.0000, 0.0000},
```

```
        {10.0000, 11.0000, 0.0000},
        {11.0000, 11.0000, 0.0000},
        {11.0000, 10.0000, 0.0000}
};

const SG_INDEX_TRIANGLE  indexes[] = {
        {1, 0, 2},
        {3, 4, 5},
        {1, 2, 5},
        {0, 4, 3},
        {2, 0, 6},
        {2, 6, 7},
        {0, 3, 8},
        {0, 8, 6},
        {2, 7, 9},
        {2, 9, 5},
        {5, 9, 8},
        {5, 8, 3},
        {6, 8, 9},
        {6, 9, 7},

        {10, 11, 12},
        {10,12,13}
};

sgGetScene()->AttachObject(sgFileManager::ObjectFromTriangles(vert,14,
indexes,15));
```

15 Memory manager

Memory manager of the sgCore library.

The internal *memory manager* causes dynamic memory allocation and release in the sgCore library. It allows to detect memory leakage in case of the incorrect library use.

Inside the sgFreeKernel() function call the check for the user's correct work with library objects takes place besides the internal library structures release. In case of memory leakage you'll get the message about the number of unreleased objects and the volume of unreleased memory.

For example, this code:

```
sgInitKernel();
sgCSphere* sph = sgCreateSphere(10,36,36);
sgFreeKernel();
```

will display the following message:

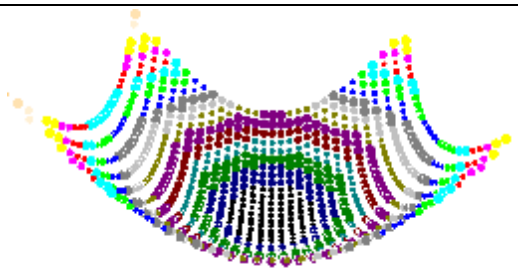
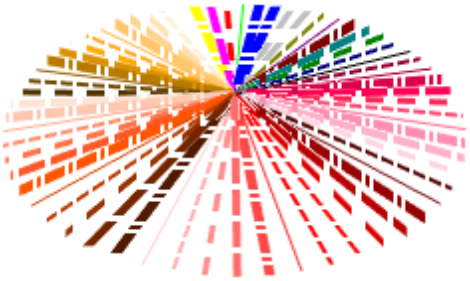
If you get such a message you should find the location of the incorrect work with sgCore functions and classes.

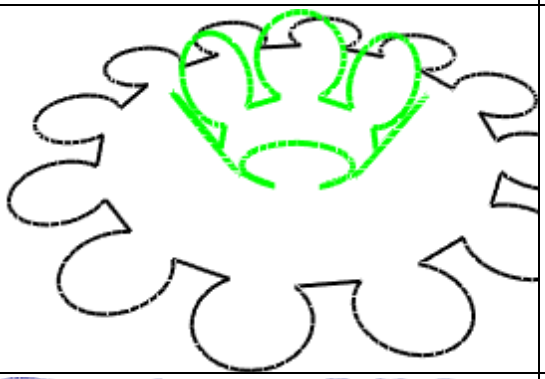
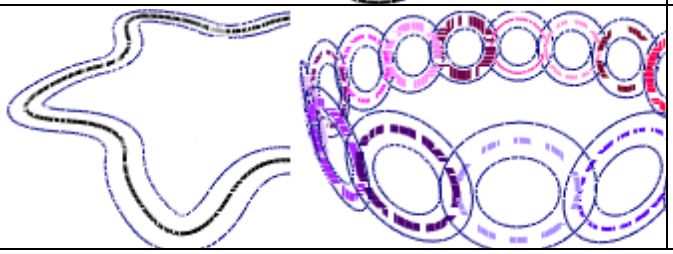
16 Examples

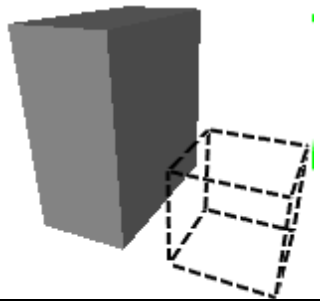
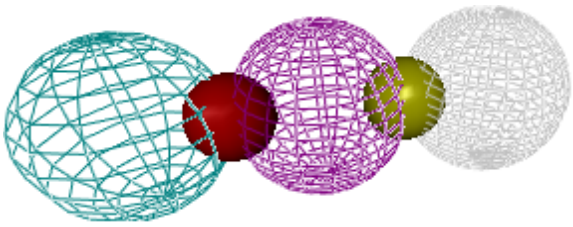
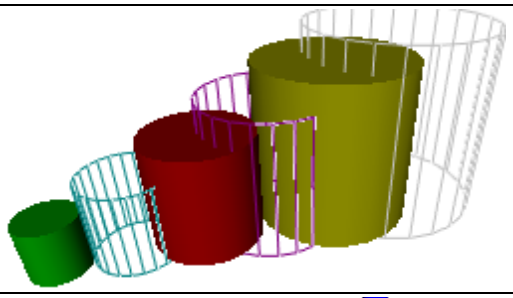
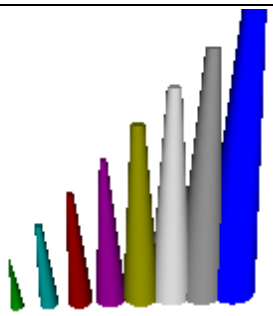
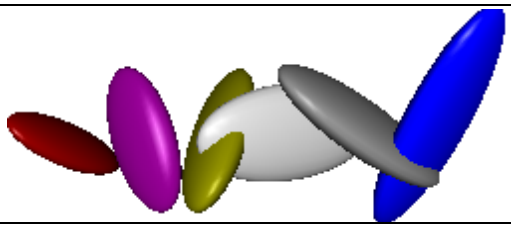
Examples of working with the sgCore library.

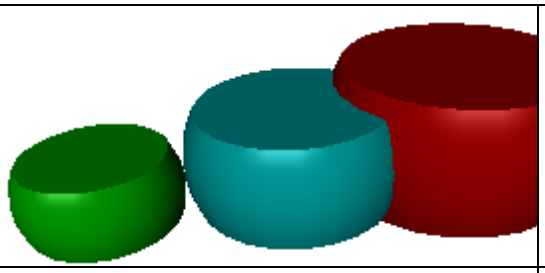
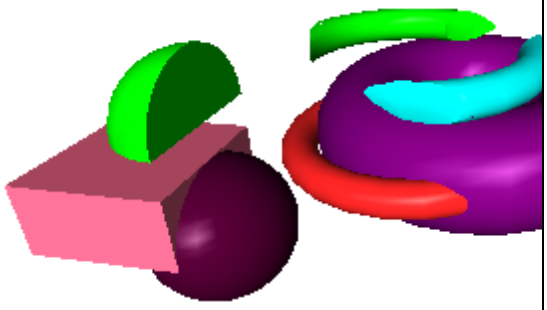
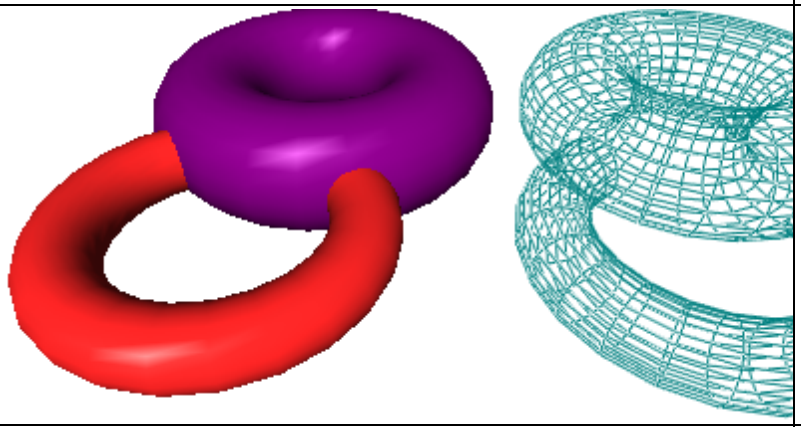
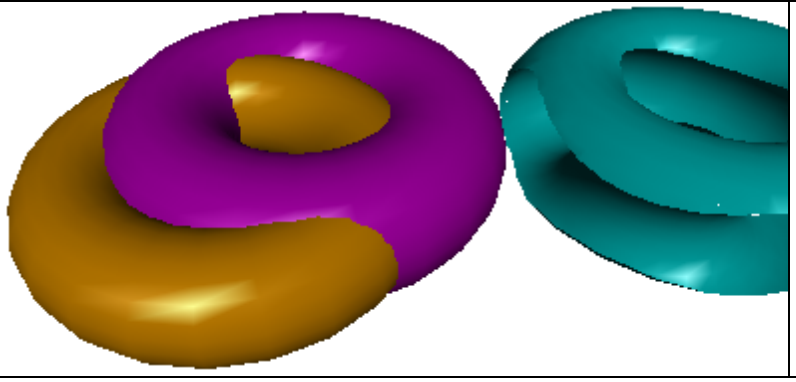
In this section the examples realized in the demo application (Demo.exe) are analyzed.
Here you can find only objects creation methods. One of the examples of drawing objects is given in the source code of the Demo.exe application (the Painter class).

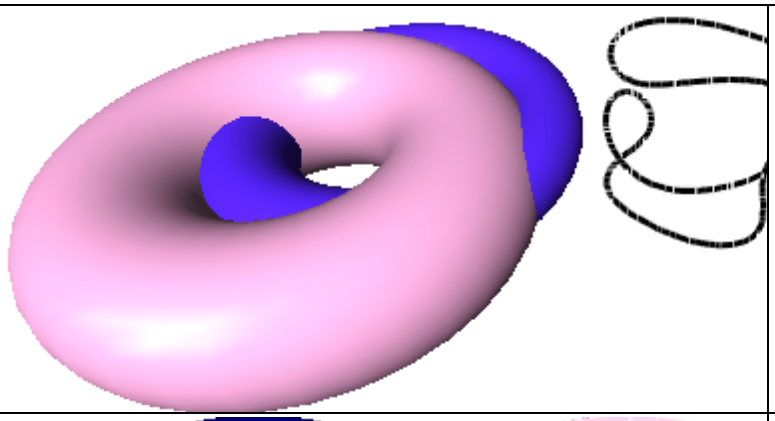
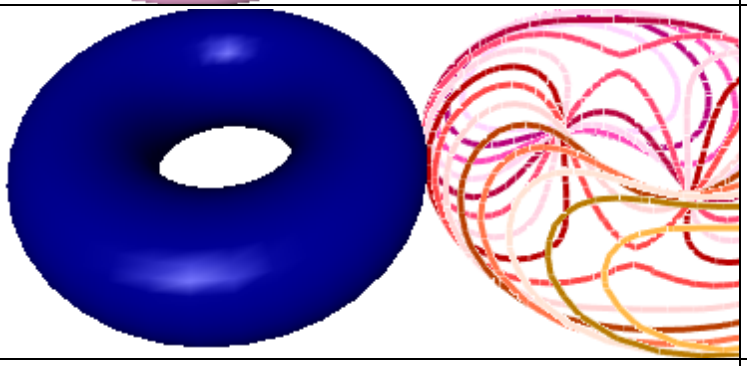
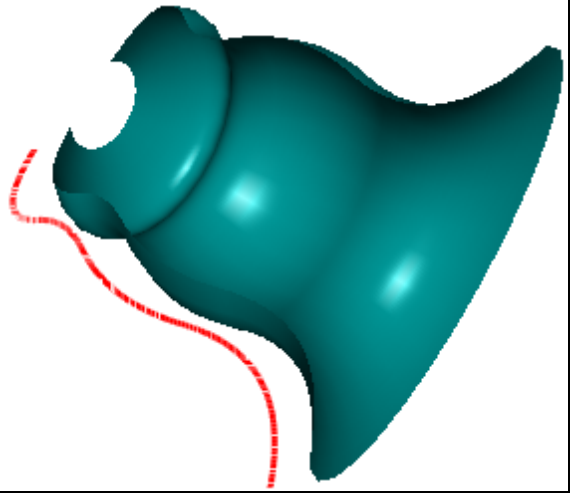
Clicking an image of the necessary example you will open the detailed description how to create it (C++ program):

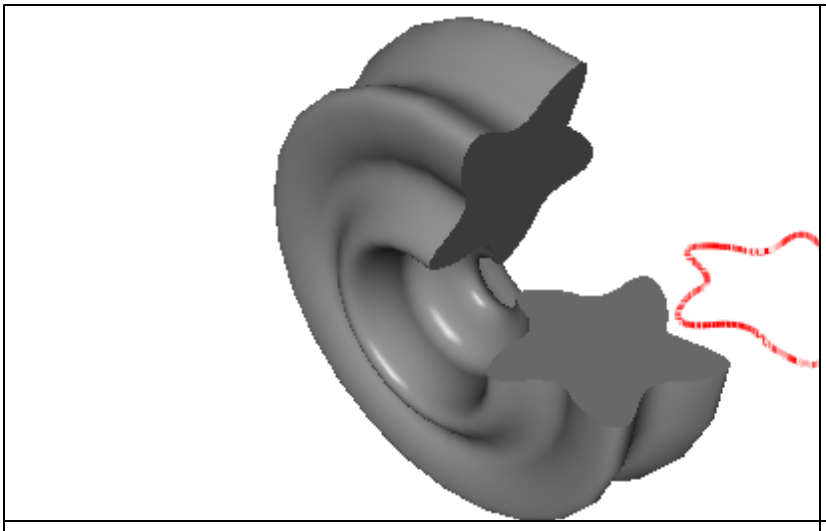
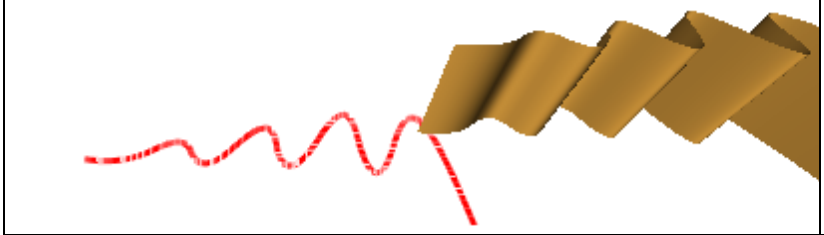
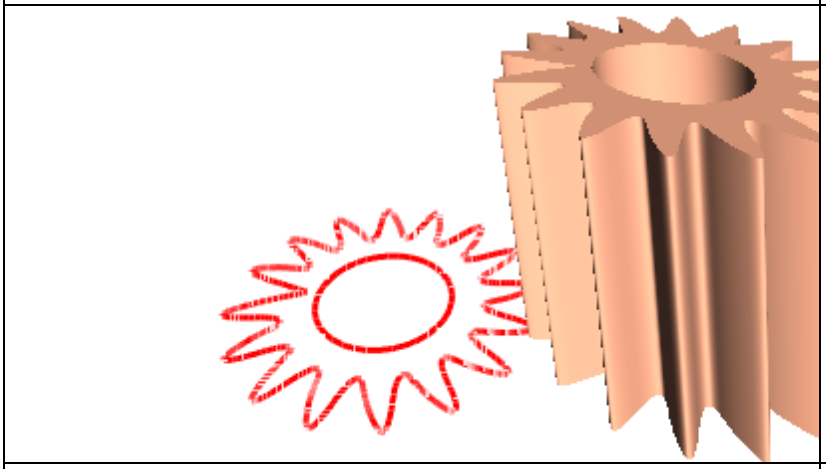
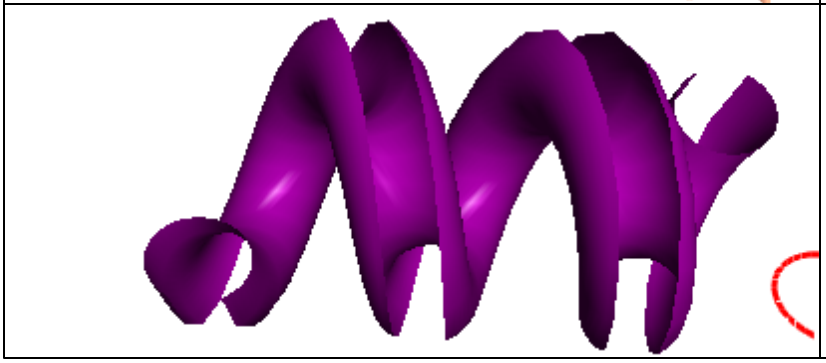
	Points
	Line segments

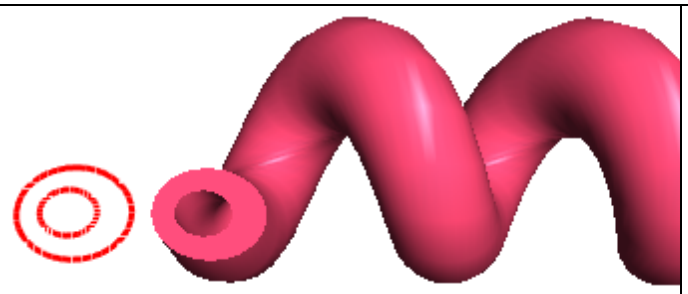
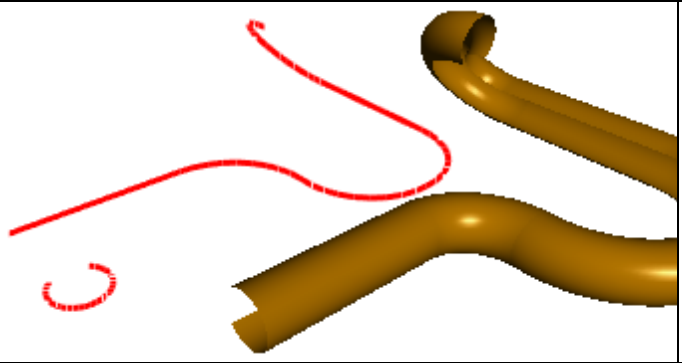
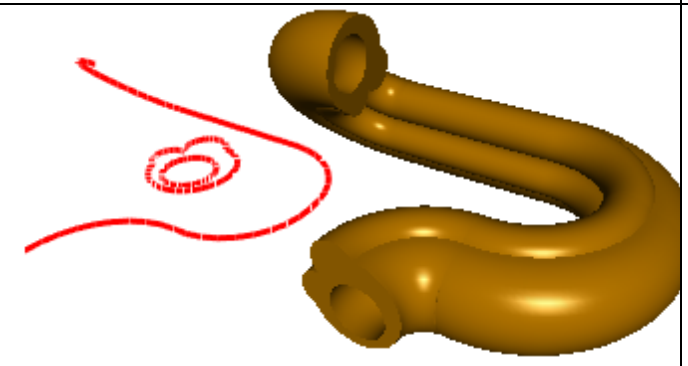
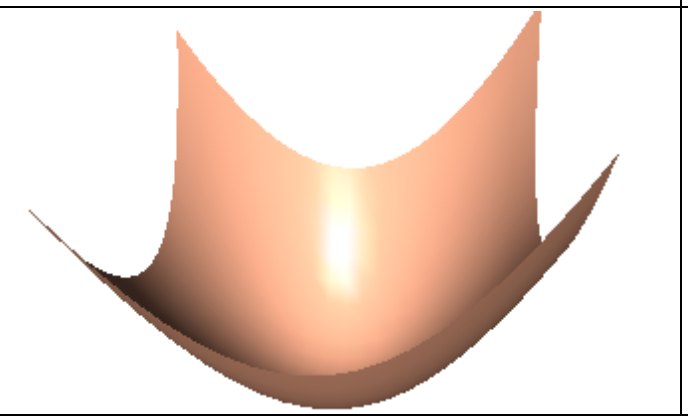
	Circles
	Arcs
	Splines
	Contours
	Equidistant line

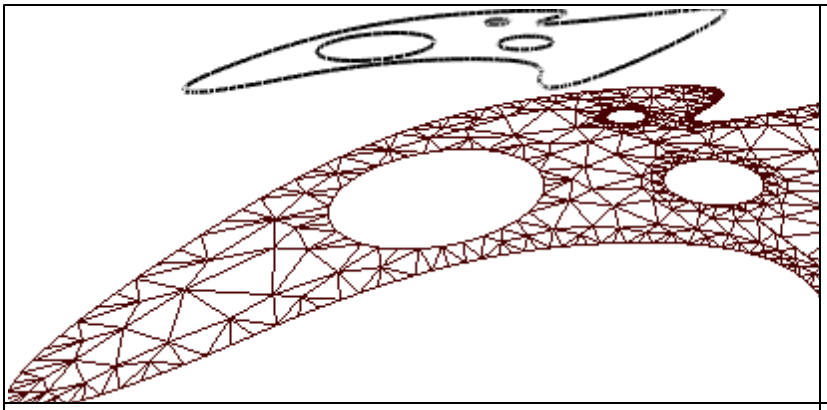
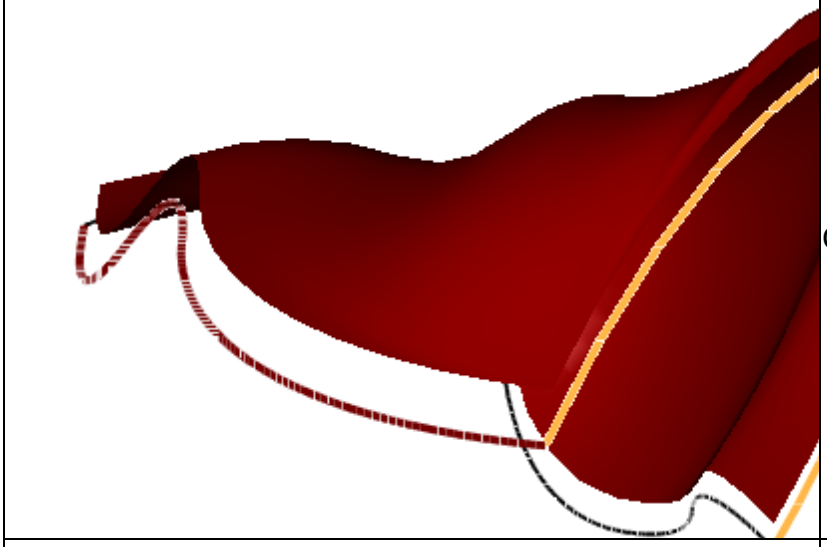
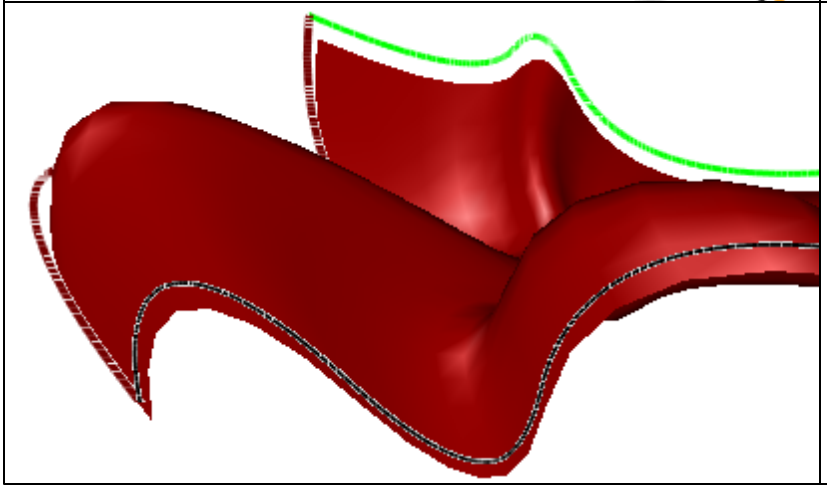
	Boxes
	Spheres
	Cylinder
	Cones
	Toruses
	Ellipsoids

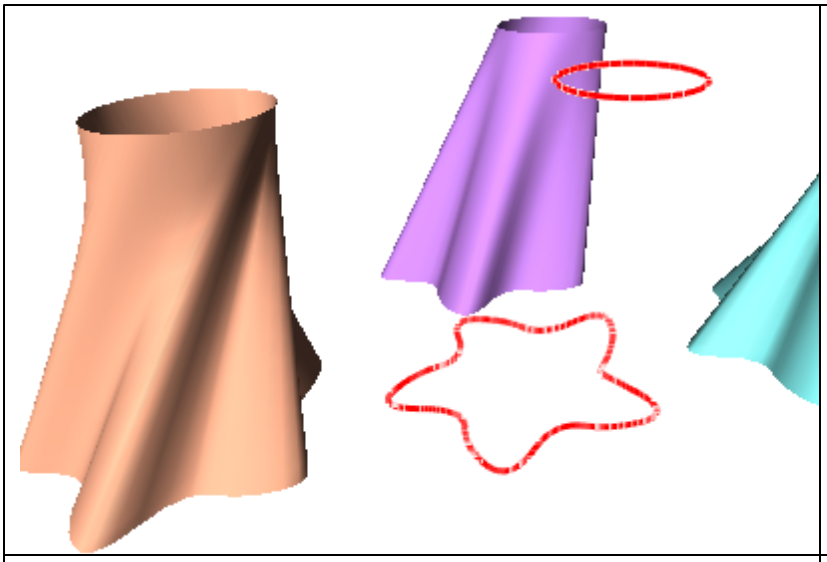
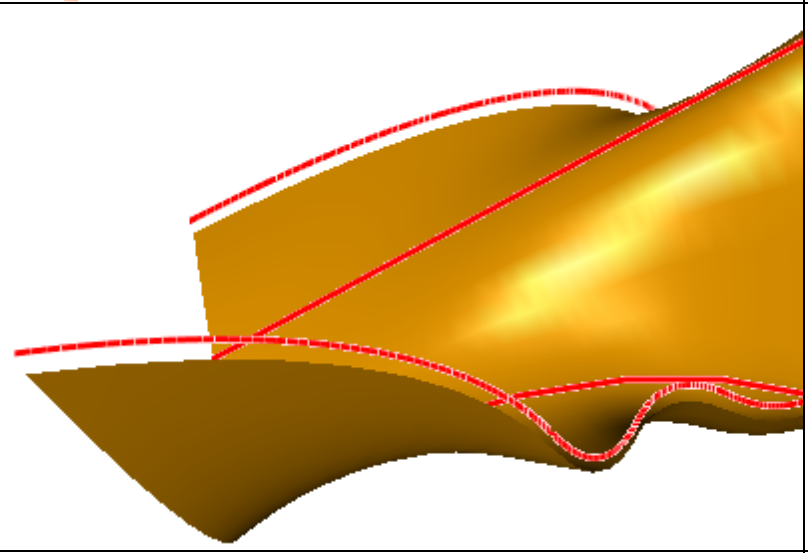
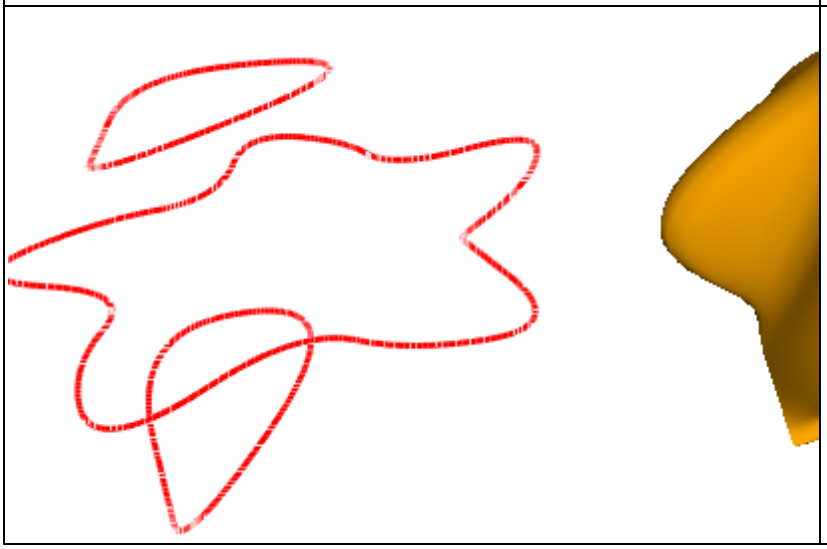
	Spherical band
	Boolean intersection
	Boolean uniting
	Boolean subtraction

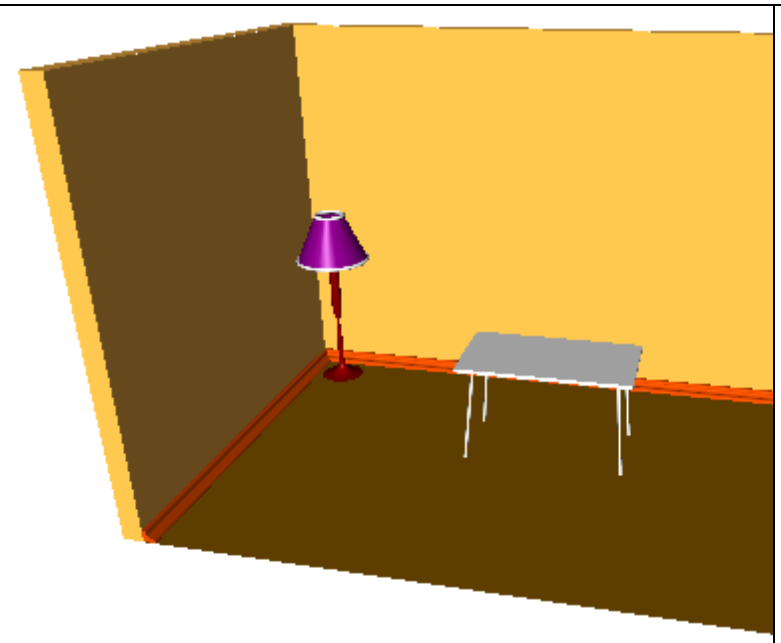
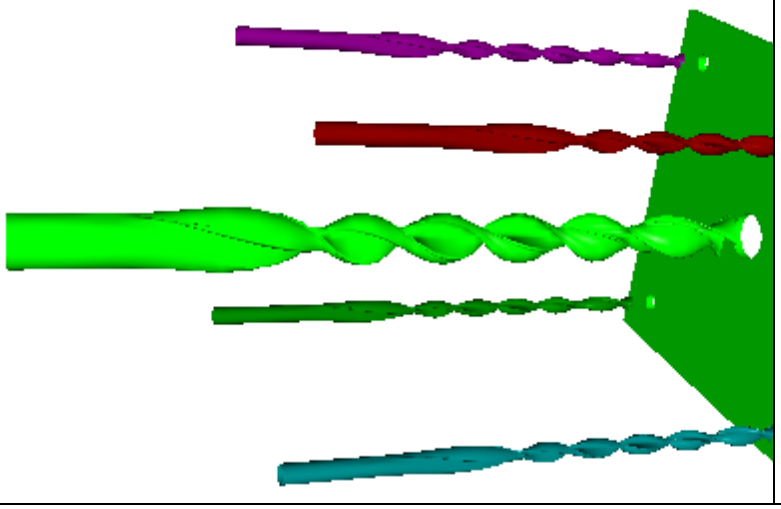
	Intersection contours
	Clips by planes
	Revolved surface

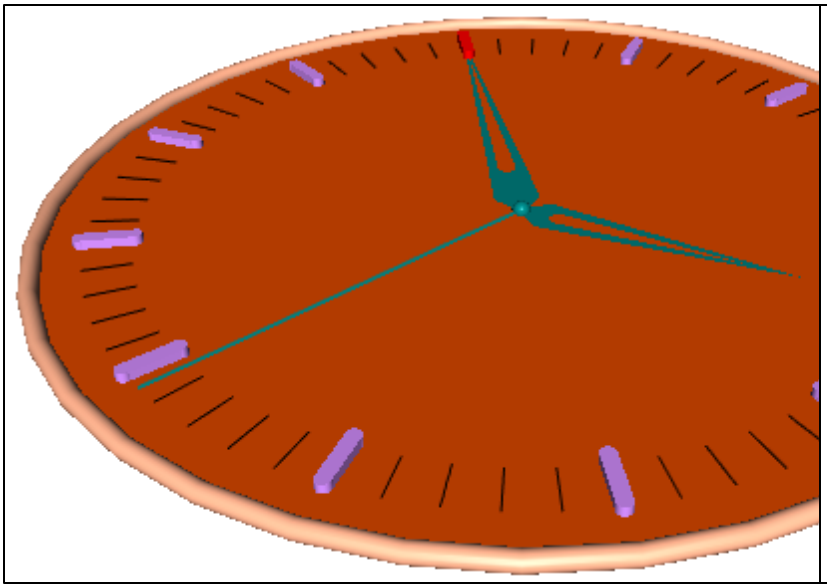
	Revolved solids
	Surfaces of extrusion
	Solids of extrusion
	Spiral surfaces

	Spiral solids
	Pipe-like surfaces
	Pipe-like solids
	Meshes

	Flat faces with holes
	Coons surfaces by three boundary contours
	Coons surfaces by four boundary contours

	Ruled surfaces by two clips
	Spline surfaces by clips
	Spline solids by clips

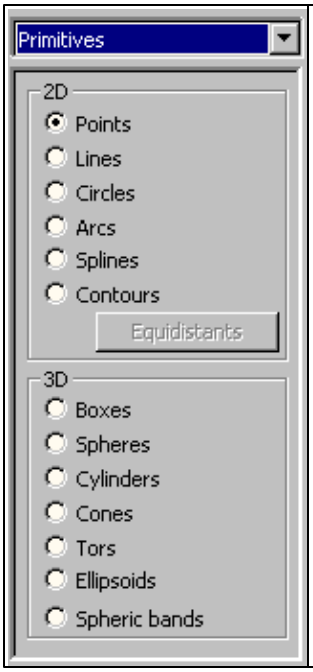
	Room
	Drills



Clock

16.1 Primitives

Primitives.



Points
Line segments
Circles
Arcs
Splines
Contours
Equidistant lines
Boxes
Spheres
Cylinders
Cones
Toruses
Ellipsoids
Spherical bands

16.1.1 Points

Points.

Let's create an array of the points lying on a paraboloid.

The intervals of changing the coordinates will be:
X lies in the interval of [-15, 15]
Y lies in the interval of [-15, 15]
The step of changing X and Y coordinates will be 1.
The paraboloid equation will be of the following type:
Z = 0.05*X*X+0.03*Y*Y;

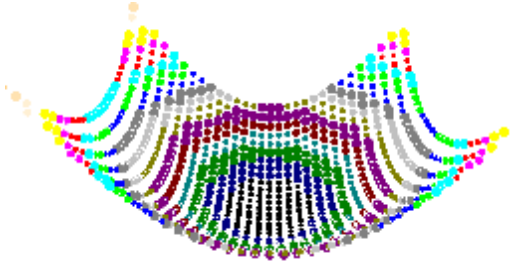
Let's calculate the color and thickness of the point depending on the Z coordinate on the following formulas:
point color number from palette = residue of division Z by 100
point thickness = residue of division Z by 3 + 1;

So, the function of creating the necessary points array will look like this:

```
for (int i=-15;i<15;i++)
{
     for (int j=-15;j<15;j++)
     {
          double pZ = 0.05*i*i+0.03*j*j;
          sgCPoint* pnt = sgCreatePoint(i, j, pZ);
          sgGetScene()->AttachObject(pnt);
          pnt->SetAttribute(SG_OA_COLOR,((int)pZ)%100);
          pnt->SetAttribute(SG_OA_LINE_THICKNESS, ((int)pZ)%3+1);
     }
}
```

See also:
sgCPoint sgCPoint::Create sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.2 Line segments

Line segments.

Let's create an array of the line segments with a vertex in the coordinates origin and the second vertex lying on a circle.

Let the variation interval of the angle moving along a circle be from 0 to Pi.
Let the variation angle step be equal to 0.1 radians, and the circle radius with the second line segment vertices be equal to 5.

Let's calculate the color, thickness and styles of the line segments on the following formulas:

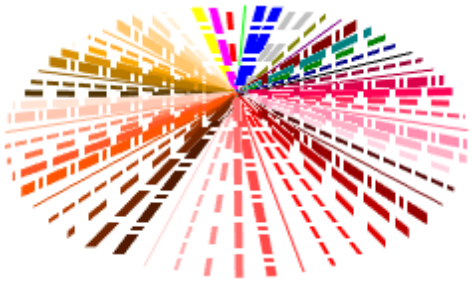
*line segment color number from palette = residue of division integer part (10*angle) by 100;*
*line segment thickness = residue of division integer part (10*angle) by 5;*
*line segment style number = residue of division integer part (10*angle) by 5;*

So, the function of creating the necessary line segments array will look like this:

```
for (double i=0.0;i<2.0*3.14159265;i+=0.1)
{
     sgCLine* ln = sgCreateLine(0.0, 0.0, 0.0, 5.0*cos(i), 5.0*sin(i), 0.0);
     sgGetScene()->AttachObject(ln);
     ln->SetAttribute(SG_OA_COLOR,((int)(10*i))%200);
     ln->SetAttribute(SG_OA_LINE_THICKNESS, ((int)(10*i))%5);
     ln->SetAttribute(SG_OA_LINE_TYPE, ((int)(10*i))%5);
}
```

See also:
sgCLine sgCLine::Create sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.3 Circles

Circles.

Let's create an array of circles with centers lying on the circle and normals directed to the centre of this circle.

Let the variation interval of the angle moving along the circle be from 0 to Pi. Let the variation angle step be equal to 0.4 radians, the circle radius with the centers be equal to 5, and the radii of the circles themselves be the same and equal to 1.

Let's calculate the color, thickness and styles of the circles on the following formulas:

*circle color number from palette = residue of division integer part (10*angle) by 200+50;*
*circle thickness = residue of division integer part (10*angle) by 5;*
*circle style number = residue of division integer part (10*angle) by 5;*

So, the function of creating the necessary line segments array will look like this:

```
for (double i=0.0;i<2.0*3.14159265;i+=0.4)
  {
    SG_POINT   crCen = {5.0*cos(i),5.0*sin(i),0.0};
    SG_VECTOR  crNor;
    crNor.x = crCen.x - 0.0;
    crNor.y = crCen.y - 0.0;
    crNor.z = 0.0;
    sgSpaceMath::NormalVector(crNor);
    SG_CIRCLE  crGeo;
    crGeo.FromCenterRadiusNormal(crCen, 1, crNor);
    sgCCircle* cr = sgCreateCircle(crGeo);
    sgGetScene()->AttachObject(cr);
    cr->SetAttribute(SG_OA_COLOR,((int)(10*i))%200+50);
    cr->SetAttribute(SG_OA_LINE_THICKNESS, ((int)(10*i))%5);
    cr->SetAttribute(SG_OA_LINE_TYPE, ((int)(10*i))%5);
  }
```

See also:
sgCCircle sgCCircle::Create SG_CIRCLE sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:

16.1.4 Arcs

Arcs.

Let's create an array of the arcs with centers lying on a circle.

Let the variation interval of the angle moving along the circle be from 0 to Pi.
Let the variation angle step be equal to 0.8 radians.

We'll draw arcs from three points - two end points and the one lying on the arc.
The end points will lie on a circle with the Z center coordinate equal to 0 and radius equal to 5, and the arc middle point will lie on a circle with the Z center coordinate equal to 3 and radius equal to 7.

Let's calculate the color, thickness and styles of the arcs on the following formulas:

*arc color number from palette = residue of division integer part (10*angle) by 200+50;*
*arc thickness = residue of division integer part (10*angle) by 5;*
*arc style number = residue of division integer part (10*angle) by 5;*

So, the function of creating the necessary line segments array will look like this:

```
for (double i=0.0;i<2.0*3.14159265;i+=0.8)
{
  SG_POINT   arP1 = {5.0*cos(i),5.0*sin(i),0.0};
  SG_POINT   arP2 = {5.0*cos(i+0.4),5.0*sin(i+0.4),0.0};
  SG_POINT   arP3 = {7.0*cos(i+0.2),7.0*sin(i+0.2),3.0};
  SG_ARC     arGeo;
  if (arGeo.FromTreePoints(arP1, arP2, arP3,false))
  {
    sgCArc* ar = sgCreateArc(arGeo);
    if (ar)
    {
      sgGetScene()->AttachObject(ar);
      ar->SetAttribute(SG_OA_COLOR,((int)(10*i))%200+50);
      ar->SetAttribute(SG_OA_LINE_THICKNESS, ((int)(10*i))%5);
      ar->SetAttribute(SG_OA_LINE_TYPE, ((int)(10*i))%5);
    }
  }
```

```
}
```

See also:
sgCArc sgCArc::Create SG_ARC sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:

16.1.5 Splines

Splines.

Let's create two splines.

The first spline won't be flat and closed. Its knots will lye on circles with different radii which Z coordinates will be 1 and -1 one after another.

The color of the first spline will be number 10 from the palette, line thickness will be 2:

```
SG_SPLINE* splGeo_1 = SG_SPLINE::Create();
SG_POINT tmpPnt;

int fl=0;
for (double i=0.0;i<2.0*3.14159265;i+=0.4)
{
  tmpPnt.x = ((double)(fl%3))*cos(i);
  tmpPnt.y = ((double)(fl%3))*sin(i);
  tmpPnt.z = (fl%2)?1.0:-1.0;
  splGeo_1->AddKnot(tmpPnt,fl);
  fl++;
}

sgCSpline* spl1_obj = sgCreateSpline(*splGeo_1);
spl1_obj->SetAttribute(SG_OA_COLOR,10);
spl1_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);
sgGetScene()->AttachObject(spl1_obj);
SG_SPLINE::Delete(splGeo_1);
```

The second spline will be flat and closed. Its knots will lye on circles with 2, 3 and 4 radii one after another.

The color of the second spline will be number 0 from the palette, line thickness will be 1:

```
SG_SPLINE* splGeo_2 = SG_SPLINE::Create();
fl=0;
for (double i=0.0;i<2.0*3.14159265;i+=0.4)
{
  tmpPnt.x = ((double)(fl%3+2))*cos(i);
  tmpPnt.y = ((double)(fl%3+2))*sin(i);
  tmpPnt.z = 0.0;
  splGeo_2->AddKnot(tmpPnt,fl);
  fl++;
}
splGeo_2->Close();

sgCSpline* spl2_obj = sgCreateSpline(*splGeo_2);
spl2_obj->SetAttribute(SG_OA_COLOR,0);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 1);
sgGetScene()->AttachObject(spl2_obj);

SG_SPLINE::Delete(splGeo_2);
```

See also:
sgCSpline sgCSpline::Create SG_SPLINE sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:

16.1.6 Contours

Contours.

Let's create two contours.

The first one will be flat and closed. It will consist of an array of alternating arcs and line segments lying on a circle.

The color of the first contour will be number 0 from the palette, line thickness will be 1:

```
  std::vector<sgCObject*> objts;

  for (double i=0.0;i<2.0*3.14159265;i+=2.0*3.14159265/12.0)
  {
  SG_POINT   arP1 = {5.0*cos(i),5.0*sin(i),0.0};
  SG_POINT   arP2 = {5.0*cos(i+2.0*3.14159265/24.0),5.0*sin
(i+2.0*3.14159265/24.0),0.0};
  SG_POINT   arP3 = {7.0*cos(i+2.0*3.14159265/48.0),7.0*sin
(i+2.0*3.14159265/48.0),0.0};
  SG_ARC     arGeo;
  if (arGeo.FromTreePoints(arP1, arP2, arP3,false))
  {
         sgCArc* ar = sgCreateArc(arGeo);
         if (ar)
         {
                objts.push_back(ar);
         }
  }
  objts.push_back(sgCreateLine(5.0*cos(i+2.0*3.14159265/24.0),5.0*sin
(i+2.0*3.14159265/24.0),0.0,
                5.0*cos(i+2.0*3.14159265/12.0),5.0*sin
(i+2.0*3.14159265/12.0),0.0));
  }

  sgCContour* cnt1 = sgCContour::CreateContour(&objts[0],objts.size());
  objts.clear();
  sgGetScene()->AttachObject(cnt1);
  cnt1->SetAttribute(SG_OA_COLOR,0);
  cnt1->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

The second contour will be closed but not flat. It will also consist of an array of alternating arcs and line segments, but the arcs will be turned in space - the Z coordinate of the arc middle point will be 2.

The color of the second contour will be number 10 from the palette, line thickness will be 2:

```
      for (double i=0.0;i<2.0*3.14159265;i+=2.0*3.14159265/6.0)
        {
        SG_POINT   arP1 = {2.0*cos(i),2.0*sin(i),0.0};
        SG_POINT   arP2 = {2.0*cos(i+2.0*3.14159265/12.0),2.0*sin
    (i+2.0*3.14159265/12.0),0.0};
        SG_POINT   arP3 = {3.0*cos(i+2.0*3.14159265/24.0),3.0*sin
    (i+2.0*3.14159265/24.0),2.0};
        SG_ARC     arGeo;
        if (arGeo.FromTreePoints(arP1, arP2, arP3,false))
        {
               sgCArc* ar = sgCreateArc(arGeo);
               if (ar)
               {
                      objts.push_back(ar);
```

```
            }
        }
        objts.push_back(sgCreateLine(2.0*cos(i+2.0*3.14159265/12.0),2.0*sin
    (i+2.0*3.14159265/12.0),0.0,    2.0*cos(i+2.0*3.14159265/6.0),2.0*sin
    (i+2.0*3.14159265/6.0),0.0));
        }

        sgCContour* cnt2 = sgCContour::CreateContour(&objts[0],objts.size());
        objts.clear();
        sgGetScene()->AttachObject(cnt2);
        cnt2->SetAttribute(SG_OA_COLOR,10);
        cnt2->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

See also:
sgCContour sgCContour::CreateContour sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:

16.1.7 Equidistant lines

Equidistant lines.

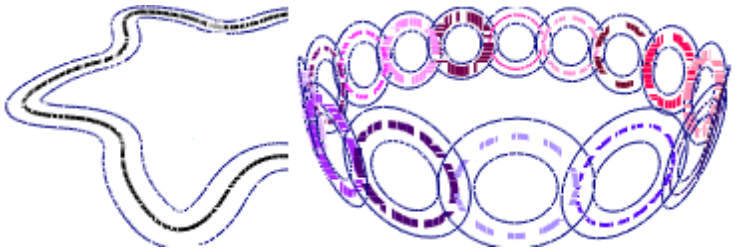
Let's create equidistant lines both with positive and negative shift to all flat objects located in the current scene. For this, we'll check the type of each object in the scene using the cycle statement. If the object is nonlinear and flat we'll build to it equidistant lines with a shift 0.3 and -0.3 from each side:

```
    sgCObject* curObj = sgGetScene()->GetObjectsList()->GetHead();
    std::vector<sgC2DObject*>   good_objcts;
    int i=0;
    while (curObj)
    {
      SG_OBJECT_TYPE ot = curObj->GetType();
      if (ot==SG_OT_CIRCLE ||ot==SG_OT_ARC || ot==SG_OT_SPLINE ||
  ot==SG_OT_CONTOUR)
      {
        sgC2DObject* o2D = reinterpret_cast<sgC2DObject*>(curObj);
```

```
        if (!o2D->IsLinear() && o2D->IsPlane(NULL,NULL))
          good_objcts.push_back(o2D);
      }
      curObj = sgGetScene()->GetObjectsList()->GetNext(curObj);
    }

    size_t sz = good_objcts.size();
    for (size_t i=0;i<sz;i++)
    {
      sgCContour* eq1 = good_objcts[i]->GetEquidistantContour(0.3, 0.3, false);
      if (eq1)
        sgGetScene()->AttachObject(eq1);
      sgCContour* eq2 = good_objcts[i]->GetEquidistantContour(-0.3, -0.3,
  false);
      if (eq2)
        sgGetScene()->AttachObject(eq2);
    }
```

See also:
sgC2DObject sgC2DObject::GetEquidistantContour
sgCObject::GetType sgC2DObject::IsLinear sgC2DObject::IsPlane
sgGetScene sgCScene::GetObjectsList sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.8 Boxes

Boxes.

Let's create two boxes.

The first one with side lengths:
1 on the X axis,
2 on the Y axis,
2.1 on the Z axis,

The vertex of the first box will be in the coordinates origin and will have the color number 8 from the palette:

```
sgCBox* bx1 = sgCreateBox(1,2,2.1);
sgGetScene()->AttachObject(bx1);
bx1->SetAttribute(SG_OA_COLOR,8);
```

The second box with side lengths:

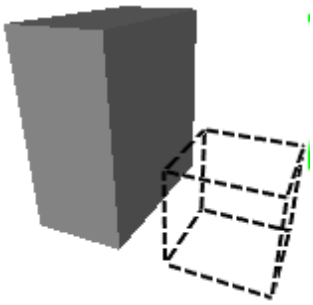
1 on the X axis,
1 on the Y axis,
1 on the Z axis,

The color of the second box will be number 0, line thickness and type will be 1, the means of drawing - a wireframe.
Let's move the vertex of the second box to the (1.5, 0, 0) point

```
sgCBox* bx2 = sgCreateBox(1,1,1);
SG_VECTOR transV = {1.5,0,0};
bx2->InitTempMatrix()->Translate(transV);
bx2->ApplyTempMatrix();
bx2->DestroyTempMatrix();
sgGetScene()->AttachObject(bx2);
bx2->SetAttribute(SG_OA_COLOR,0);
bx2->SetAttribute(SG_OA_LINE_THICKNESS,1);
bx2->SetAttribute(SG_OA_LINE_TYPE,1);
bx2->SetAttribute(SG_OA_DRAW_STATE,SG_DS_FRAME);
```

See also:
sgCBox sgCBox::Create
sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::
DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.9 Spheres

Spheres.

Let's create an array of the spheres lying on the same line but having different parameters:

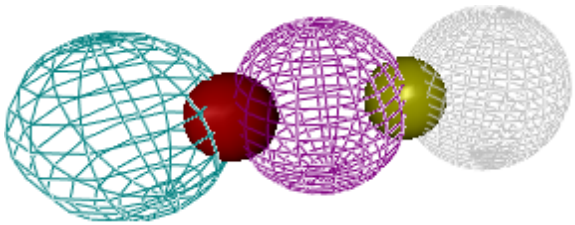
```
for (int i=3;i<8;i++)
{
    sgCSphere* sp1 = sgCreateSphere(i%2+1,4*i,4*i);
    SG_VECTOR transV1 = {2*i,2,0};
    sp1->InitTempMatrix()->Translate(transV1);
    sp1->ApplyTempMatrix();
    sp1->DestroyTempMatrix();
```

```
        sgGetScene()->AttachObject(sp1);
        sp1->SetAttribute(SG_OA_COLOR,i);
        if (i%2)
          sp1->SetAttribute(SG_OA_DRAW_STATE,SG_DS_FRAME);
    }
```

See also:

sgCSphere sgCSphere::Create sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.10 Cylinders

Cylinders.

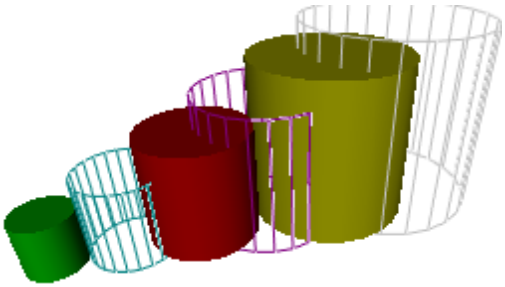
Let's create an array of the cylinders lying on the same line but having different parameters:

```
for (int i=2;i<8;i++)
{
  sgCCylinder* cy1 = sgCreateCylinder(i,2*i,24);
  SG_VECTOR transV1 = {5*i,10,10};
  cy1->InitTempMatrix()->Translate(transV1);
  cy1->ApplyTempMatrix();
  cy1->DestroyTempMatrix();

  sgGetScene()->AttachObject(cy1);
  cy1->SetAttribute(SG_OA_COLOR,i);

  if (i%2)
    cy1->SetAttribute(SG_OA_DRAW_STATE,SG_DS_FRAME);

}
```

See also:

sgCCylinder sgCCylinder::Create sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.11 Cones

Cones.

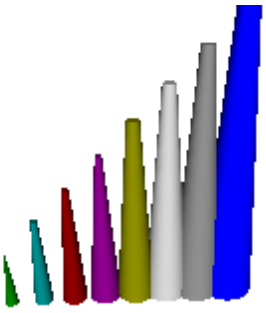
Let's create an array of the cones lying on the same line but having different parameters:

```
for (int i=2;i<10;i++)
{
  sgCCone* co1 = sgCreateCone(i,i/3,10*i,36);
  SG_VECTOR transV1 = {10*i,10,10};
  co1->InitTempMatrix()->Translate(transV1);
  co1->ApplyTempMatrix();
  co1->DestroyTempMatrix();

  sgGetScene()->AttachObject(co1);
  co1->SetAttribute(SG_OA_COLOR,i);
}
```

See also:

sgCCone sgCCone::Create sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.12 Toruses

Toruses.

Let's create an array of the toruses lying on the same line but having different parameters and randomly turned in space:

```
for (int i=2;i<10;i++)
{
  sgCTorus* tor1 = sgCreateTorus(i,i/6+2,24,24);
  SG_POINT rotCen = {0.0,0.0,0.0};
  SG_VECTOR rotDir = {rand(),rand(),rand()};
  tor1->InitTempMatrix()->Rotate(rotCen,rotDir, rand()*360);
  SG_VECTOR transV1 = {10*i,10,10};
  tor1->GetTempMatrix()->Translate(transV1);
  tor1->ApplyTempMatrix();
  tor1->DestroyTempMatrix();

  sgGetScene()->AttachObject(tor1);
  tor1->SetAttribute(SG_OA_COLOR,i);
}
```

See also:

sgCTorus sgCTorus::Create sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:

16.1.13 Ellipsoids

Ellipsoids.

Let's create an array of the ellipsoids lying on the same line but having different parameters and randomly turned in space:

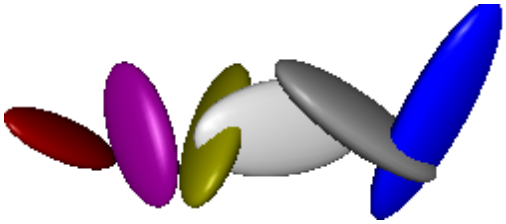
```
for (int i=3;i<10;i++)
{
  sgCEllipsoid* ell1 = sgCreateEllipsoid(i,i/6+2,2*i, 24,24);
  SG_POINT rotCen = {0.0,0.0,0.0};
  SG_VECTOR rotDir = {rand(),rand(),rand()};
  ell1->InitTempMatrix()->Rotate(rotCen,rotDir, rand()*360);
```

```
    SG_VECTOR transV1 = {10*i,10,10};
    ell1->GetTempMatrix()->Translate(transV1);
    ell1->ApplyTempMatrix();
    ell1->DestroyTempMatrix();

    sgGetScene()->AttachObject(ell1);
    ell1->SetAttribute(SG_OA_COLOR,i);
  }
```

See also:

sgCEllipsoid sgCEllipsoid::Create sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.1.14 Spherical bands

Spherical bands.

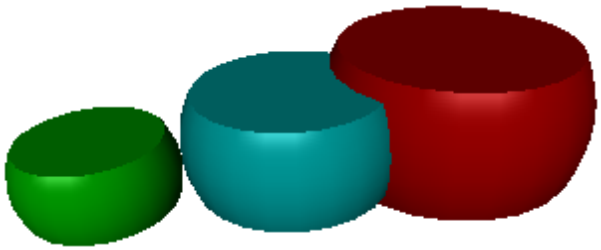
Let's create three spherical bands with different radii of the ruling spheres but with the same clipping coefficient:

```
  for (int i=2;i<5;i++)
  {
    sgCSphericBand* sb1 = sgCreateSphericBand(i,-0.4, 0.5, 24);
    SG_VECTOR transV1 = {5*i,10,10};
    sb1->InitTempMatrix()->Translate(transV1);
    sb1->ApplyTempMatrix();
    sb1->DestroyTempMatrix();

    sgGetScene()->AttachObject(sb1);
    sb1->SetAttribute(SG_OA_COLOR,i);
  }
```

See also:

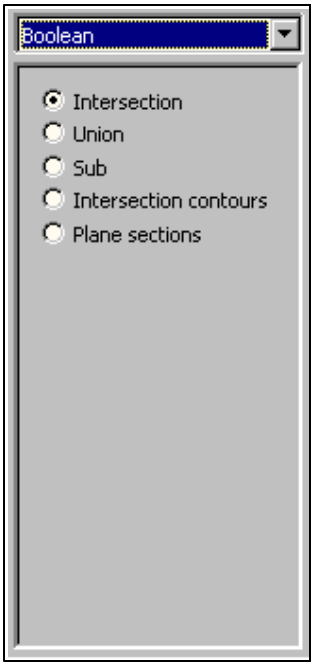
sgCSphericBand sgCSphericBand::Create sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.2 Boolean operations

Boolean operations.



Intersection
Union
Subtraction
Intersection contour
Clipping by plane

16.2.1 Intersection

Boolean intersection.

Let's create a number of objects being an intersection of 3D primitives.

Two first objects will be built by intersection of two toruses.

As sgCore is a solid modeling library the resulting Boolean operations objects are in no way connected with each other. They are grouped to return from the function.

For the further work with each object separately you should take this group to component parts.

Let's give its own color to each object obtained as a result of toruses intersection, and raise it to 1.5 for a better view.

The code of creating toruses and constructing objects from their intersection looks as follows:

```
sgCTorus* tor1 = sgCreateTorus(2,1 ,24,24);
sgCTorus* tor2 = sgCreateTorus(2,0.3 ,24,24);
SG_VECTOR transV1 = {1,1,0};
tor2->InitTempMatrix()->Translate(transV1);
tor2->ApplyTempMatrix();
tor2->DestroyTempMatrix();
sgGetScene()->AttachObject(tor1);
tor1->SetAttribute(SG_OA_COLOR,5);
sgGetScene()->AttachObject(tor2);
tor2->SetAttribute(SG_OA_COLOR,45);

SG_VECTOR transV2 = {0,0,1.5};

sgCGroup* bool1 = sgBoolean::Intersection(*tor1, *tor2);

int ChCnt = bool1->GetChildrenList()->GetCount();

sgCObject**  allChilds = (sgCObject**)malloc(ChCnt*sizeof(sgCObject*));
if (!bool1->BreakGroup(allChilds))
{
  assert(0);
}
sgCObject::DeleteObject(bool1);
for (int i=0;i<ChCnt;i++)
{
  allChilds[i]->InitTempMatrix()->Translate(transV2);
  allChilds[i]->ApplyTempMatrix();
  allChilds[i]->DestroyTempMatrix();
  sgGetScene()->AttachObject(allChilds[i]);
  allChilds[i]->SetAttribute(SG_OA_COLOR,10+i);
}

free(allChilds);
```

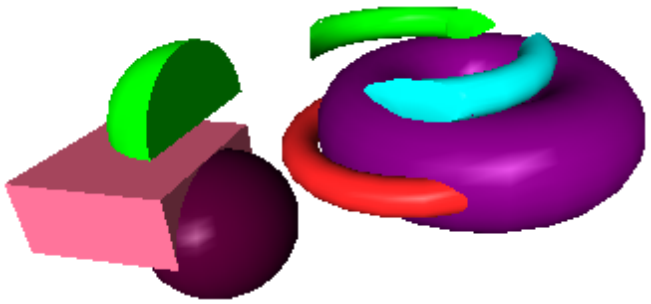
Even if only one solid is constructed as a result of objects intersection the function returns a group of objects (with one child object) all the same. It is illustrated on an example of a sphere and a box intersection:

```
sgCBox* bx1 = sgCreateBox(2,2 ,1);
sgCSphere* sp1 = sgCreateSphere(1 ,24,24);
SG_VECTOR transV4 = {3,3,0};
bx1->InitTempMatrix()->Translate(transV4);
bx1->ApplyTempMatrix();
bx1->DestroyTempMatrix();
sgGetScene()->AttachObject(bx1);
bx1->SetAttribute(SG_OA_COLOR,55);
SG_VECTOR transV5 = {3,4,0};
sp1->InitTempMatrix()->Translate(transV5);
sp1->ApplyTempMatrix();
sp1->DestroyTempMatrix();
```

```
sgGetScene()->AttachObject(sp1);
sp1->SetAttribute(SG_OA_COLOR,75);

sgCGroup* bool2 = sgBoolean::Intersection(*sp1, *bx1);

ChCnt = bool2->GetChildrenList()->GetCount();

allChilds = (sgCObject**)malloc(ChCnt*sizeof(sgCObject*));
if (!bool2->BreakGroup(allChilds))
{
  assert(0);
}
sgCObject::DeleteObject(bool2);
for (int i=0;i<ChCnt;i++)
{
  allChilds[i]->InitTempMatrix()->Translate(transV2);
  allChilds[i]->ApplyTempMatrix();
  allChilds[i]->DestroyTempMatrix();
  sgGetScene()->AttachObject(allChilds[i]);
  allChilds[i]->SetAttribute(SG_OA_COLOR,10+i);
}

free(allChilds);
```

See also:
sgBoolean sgBoolean::Intersection
sgCGroup sgCGroup::BreakGroup sgCGroup::GetChildrenList
sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::
DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute
sgCObject::DeleteObject

Illustration:



16.2.2 Union

Boolean union.

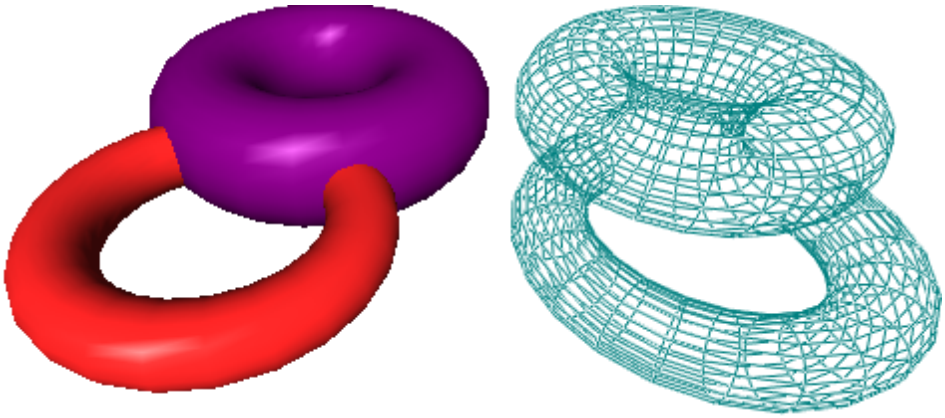
Let's create a group being a two toruses joining:

```
sgCTorus* tor1 = sgCreateTorus(2,1 ,24,24);
sgCTorus* tor2 = sgCreateTorus(2,0.5 ,24,24);
SG_VECTOR transV1 = {1,3.5,0};
tor2->InitTempMatrix()->Translate(transV1);
```

```
tor2->ApplyTempMatrix();
tor2->DestroyTempMatrix();
sgGetScene()->AttachObject(tor1);
tor1->SetAttribute(SG_OA_COLOR,5);
sgGetScene()->AttachObject(tor2);
tor2->SetAttribute(SG_OA_COLOR,45);

SG_VECTOR transV2 = {-6.5,0,0};

sgCGroup* bool1 = sgBoolean::Union(*tor1, *tor2);

bool1->InitTempMatrix()->Translate(transV2);
bool1->ApplyTempMatrix();
bool1->DestroyTempMatrix();
sgGetScene()->AttachObject(bool1);
bool1->SetAttribute(SG_OA_COLOR,3);
bool1->SetAttribute(SG_OA_DRAW_STATE,SG_DS_FRAME);
```

See also:
sgBoolean sgBoolean::Union
sgCGroup sgCGroup::BreakGroup sgCGroup::GetChildrenList
sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::
DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.2.3 Subtraction

Boolean subtraction.

Let's create a group by subtracting a torus from another one.

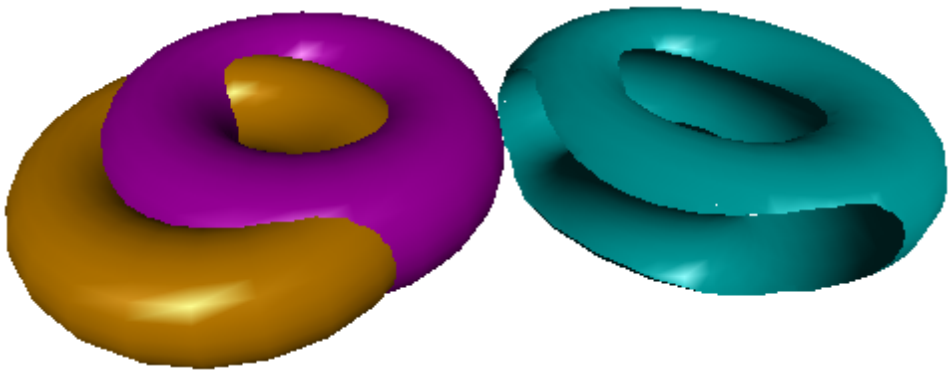
As sgCore is a solid modeling library the resulting Boolean operations objects are in no way connected with each other. They are grouped to return from the function. For the further work with each object separately you should take this group component parts.

The code of creating toruses and constructing objects from their subtraction looks as follows:

```
sgCTorus* tor1 = sgCreateTorus(2,1 ,24,24);
sgCTorus* tor2 = sgCreateTorus(2,0.8 ,24,24);
SG_VECTOR transV1 = {1,1,0};
tor2->InitTempMatrix()->Translate(transV1);
tor2->ApplyTempMatrix();
tor2->DestroyTempMatrix();
sgGetScene()->AttachObject(tor1);
tor1->SetAttribute(SG_OA_COLOR,5);
sgGetScene()->AttachObject(tor2);
tor2->SetAttribute(SG_OA_COLOR,25);

SG_VECTOR transV2 = {-6.0,0,0};

sgCGroup* bool1 = sgBoolean::Sub(*tor1, *tor2);

bool1->InitTempMatrix()->Translate(transV2);
bool1->ApplyTempMatrix();
bool1->DestroyTempMatrix();
sgGetScene()->AttachObject(bool1);
bool1->SetAttribute(SG_OA_COLOR,3);
```

See also:
sgBoolean sgBoolean::Sub
sgCGroup sgCGroup::BreakGroup sgCGroup::GetChildrenList
sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::
DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.2.4 Intersections contours

Two 3D objects surfaces intersection contours.

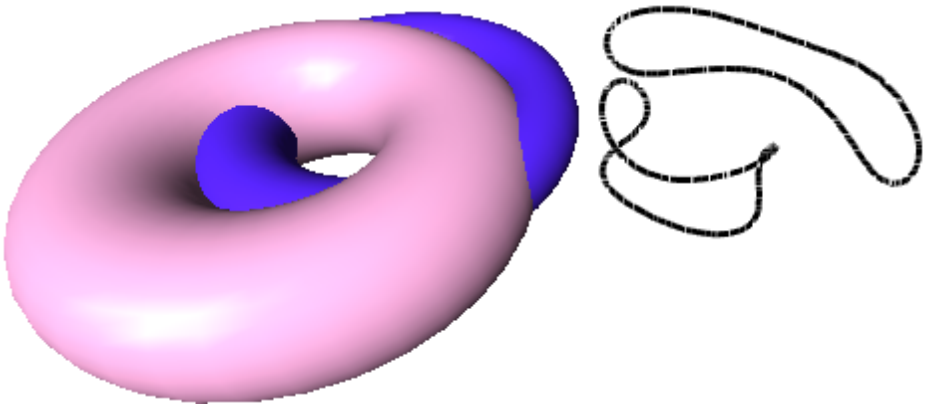
Let's create a set of groups of line segments by intersecting the surfaces of two toruses.

The code of creating toruses and constructing a group of line segments from their intersection looks as follows:

```
sgCTorus* tor1 = sgCreateTorus(2,1 ,36,36);
sgCTorus* tor2 = sgCreateTorus(2,0.6 ,36,36);
SG_VECTOR transV1 = {1,1,0};
tor2->InitTempMatrix()->Translate(transV1);
tor2->ApplyTempMatrix();
tor2->DestroyTempMatrix();
sgGetScene()->AttachObject(tor1);
tor1->SetAttribute(SG_OA_COLOR,65);
sgGetScene()->AttachObject(tor2);
tor2->SetAttribute(SG_OA_COLOR,105);

SG_VECTOR transV2 = {5,0,0.0};

sgCGroup* bool1 = sgBoolean::IntersectionContour(*tor1, *tor2);

bool1->InitTempMatrix()->Translate(transV2);
bool1->ApplyTempMatrix();
bool1->DestroyTempMatrix();
sgGetScene()->AttachObject(bool1);
bool1->SetAttribute(SG_OA_COLOR,0);
bool1->SetAttribute(SG_OA_LINE_THICKNESS,2);
```

See also:
sgBoolean sgBoolean::IntersectionContour
sgCGroup sgCGroup::BreakGroup sgCGroup::GetChildrenList
sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::
DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.2.5 Clips by plane

3D object surface and plane intersection contours.

Let's create a set of groups of line segments by intersecting the surface of a torus

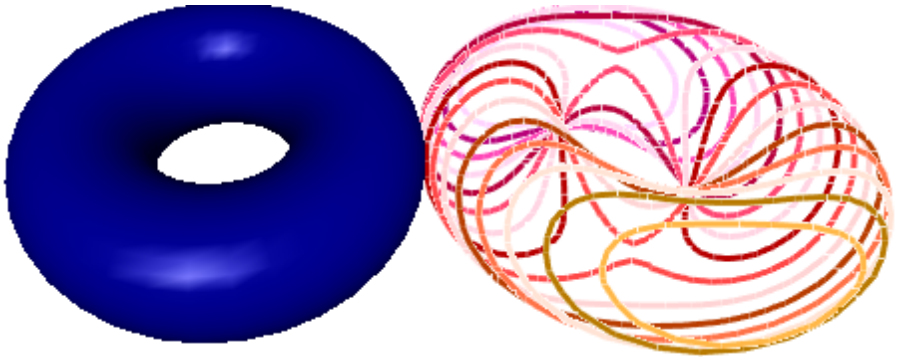
and a set of surfaces.

The code of creating a torus and drawing clips:

```
sgCTorus* tor1 = sgCreateTorus(2,1 ,36,36);
sgGetScene()->AttachObject(tor1);

SG_VECTOR transV2 = {-6,0,0};
for (int i=-30;i<30;i+=4)
{
  SG_VECTOR plN;
  plN.x =0.0; plN.y = 1.0; plN.z = 1.0;

  sgCGroup* bool1 = sgBoolean::Section(*tor1, plN, 0.1*i );

  if (bool1)
  {
    bool1->InitTempMatrix()->Translate(transV2);
    bool1->ApplyTempMatrix();
    bool1->DestroyTempMatrix();
    sgGetScene()->AttachObject(bool1);
    bool1->SetAttribute(SG_OA_COLOR,i+50);
    bool1->SetAttribute(SG_OA_LINE_THICKNESS,2);
  }
}
```

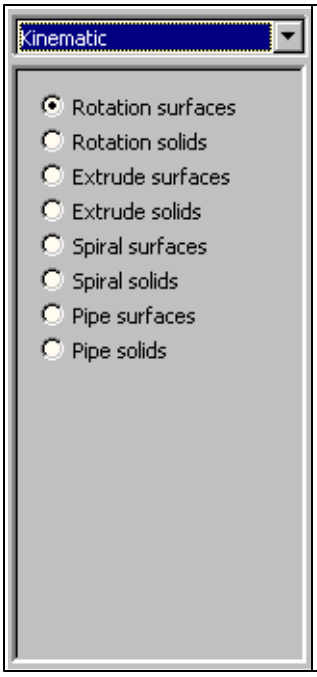
See also:
sgBoolean sgBoolean::Section
sgCGroup sgCGroup::BreakGroup sgCGroup::GetChildrenList
sgCObject::InitTempMatrix sgCMatrix::Translate sgCObject::ApplyTempMatrix SgCObject::DestroyTempMatrix sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3 Kinematic

Kinematic objects.



16.3.1 Surfaces of revolution

Surfaces of revolution.

Let's create a surface of revolution. An unclosed spline will be the rotated contour.

This code creates a spline:

```
SG_POINT tmpPnt;

SG_SPLINE* spl2 = SG_SPLINE::Create();

tmpPnt.x = 1.0; tmpPnt.y = -3.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,0);
tmpPnt.x = 3.0; tmpPnt.y = -2.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,1);
tmpPnt.x = 2.0; tmpPnt.y = -1.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,2);
tmpPnt.x = 3.0; tmpPnt.y = 1.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,3);
tmpPnt.x = 2.0; tmpPnt.y = 4.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,4);
tmpPnt.x =4.0; tmpPnt.y = 5.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,5);

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,12);
```

```
   spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

   SG_SPLINE::Delete(spl2);
```

Let's rotate the contour around the axis lying on the (0, 0, 0) and (0, -2, 0) points. The rotation angle will be 250 degrees:

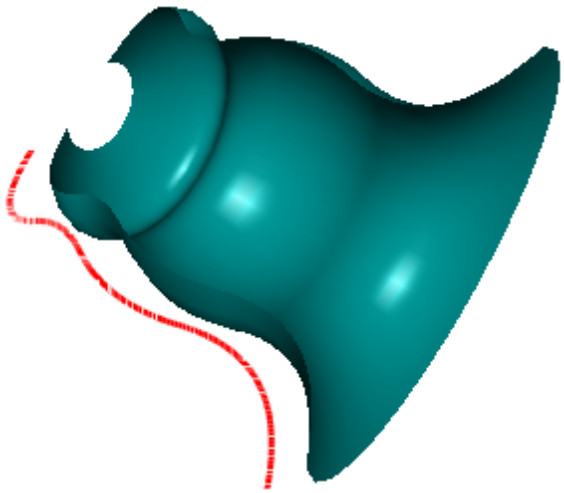
```
   SG_POINT p1 = {0,-2,0};
   SG_POINT p2 = {0,0,0};

   sgC3DObject* rO = (sgC3DObject*)sgKinematic::Rotation(*spl2_obj,p1,p2,250,
false);

   sgGetScene()->AttachObject(rO);
   rO->SetAttribute(SG_OA_COLOR,3);
```

Then we'll move the obtained surface:

```
   SG_VECTOR transV1 = {-1,0,0};
   rO->InitTempMatrix()->Translate(transV1);
   rO->ApplyTempMatrix();
   rO->DestroyTempMatrix();
```

See also:
sgKinematic::Rotation
sgCSpline sgCSpline::Create SG_SPLINE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.2 Solids of revolution

Solids of revolution.

Let's create a solid of revolution. A closed spline will be the rotated contour.

This code creates a spline:

```
SG_POINT tmpPnt;
SG_SPLINE* spl2 = SG_SPLINE::Create();
int fl=0;
for (double i=0.0;i<2.0*3.14159265;i+=0.4)
{
       tmpPnt.x = ((double)(fl%3+2))*cos(i);
       tmpPnt.y = ((double)(fl%3+2))*sin(i);
       tmpPnt.z = 0.0;
       spl2->AddKnot(tmpPnt,fl);
       fl++;
}
spl2->Close();

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,12);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS,2);

SG_SPLINE::Delete(spl2);
```

Let's rotate the contour around the axis lying on the (10, 10, 0) and (20, 15, 0) points. The rotation angle will be 290 degrees:

```
SG_POINT p1 = {10,10,0};
SG_POINT p2 = {20,15,0};

sgC3DObject* rO = (sgC3DObject*)sgKinematic::Rotation(*spl2_obj,p1,p2,290,
true);

sgGetScene()->AttachObject(rO);
rO->SetAttribute(SG_OA_COLOR,8);
```

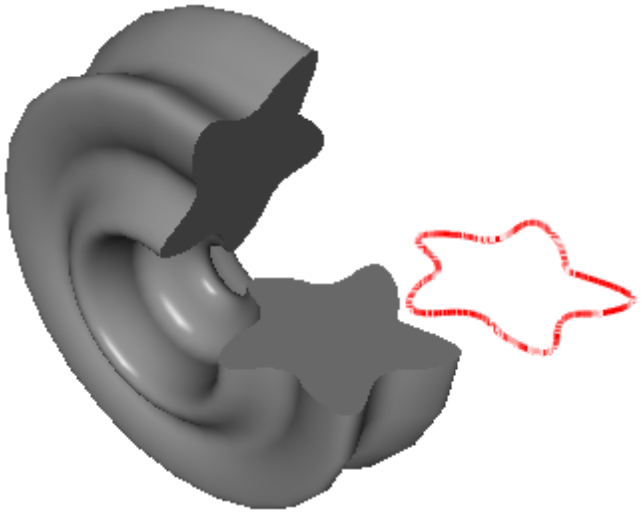
Then we'll move the obtained solid:

```
SG_VECTOR transV1 = {-6,0,0};
rO->InitTempMatrix()->Translate(transV1);
rO->ApplyTempMatrix();
rO->DestroyTempMatrix();
```

See also:
sgKinematic::Rotation

sgCSpline sgCSpline::Create SG_SPLINE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.3 Surfaces of extrusion

Surfaces of extrusion.

Let's create a surface of extrusion. An unclosed spline will be the extruded contour.

This code creates a spline:

```
SG_POINT tmpPnt;

SG_SPLINE* spl2 = SG_SPLINE::Create();

int fl=0;
for (double i=0.0;i<10.0;i+=1.0)
{
	tmpPnt.x = i;
	tmpPnt.y = (fl%2)?-i/2.0:i/2.0;
	tmpPnt.z = 0.0;
	spl2->AddKnot(tmpPnt,fl);
	fl++;
}


sgCSpline* spl2_obj = sgCreateSpline(*spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,12);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

```
SG_SPLINE::Delete(spl2);
```

Let's extrude along the (1, -2, 3) vector:

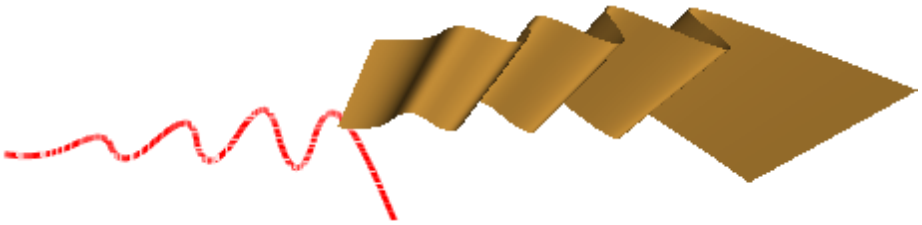
```
SG_VECTOR extVec = {1,-2,3};

sgC3DObject* exO = (sgC3DObject*)sgKinematic::Extrude((const sgC2DObject&)
(*spl2_obj),NULL,0,extVec,false);

sgGetScene()->AttachObject(exO);
exO->SetAttribute(SG_OA_COLOR,20);
```

Then we'll move the obtained surface:

```
SG_VECTOR transV1 = {8,0,0};
exO->InitTempMatrix()->Translate(transV1);
exO->ApplyTempMatrix();
exO->DestroyTempMatrix();
```

See also:
sgKinematic::Extrude
sgCSpline sgCSpline::Create SG_SPLINE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.4 Solids of extrusion

Solids of extrusion.

Let's create a solid of extrusion with a circular hole. An unclosed spline will be the extruded contour.

This code creates a spline:

```
SG_POINT tmpPnt;
SG_SPLINE* spl2 = SG_SPLINE::Create();
int fl=0;
for (double i=0.0;i<2.0*3.14159265;i+=0.13)
{
       tmpPnt.x = ((double)(fl%3+2))*cos(i);
```

```
        tmpPnt.y = ((double)(fl%3+2))*sin(i);
        tmpPnt.z = 0.0;
        spl2->AddKnot(tmpPnt,fl);
        fl++;
    }
    spl2->Close();
    sgCSpline* spl2_obj = sgCreateSpline(*spl2);
    SG_SPLINE::Delete(spl2);
    sgGetScene()->AttachObject(spl2_obj);
    spl2_obj->SetAttribute(SG_OA_COLOR,12);
    spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

The hole will have the form of the circle with 1.6 radius:

```
    SG_CIRCLE   cirGeo;
    SG_POINT    circ = {0.0, 0.0, 0.0};
    SG_VECTOR    cirNor = {0.0, 0.0, 1.0};
    cirGeo.FromCenterRadiusNormal(circ,1.6, cirNor);
    sgC2DObject*  cir = sgCreateCircle(cirGeo);
    sgGetScene()->AttachObject(cir);
    cir->SetAttribute(SG_OA_COLOR,12);
    cir->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Let's extrude along the (1, -2, 5) vector:

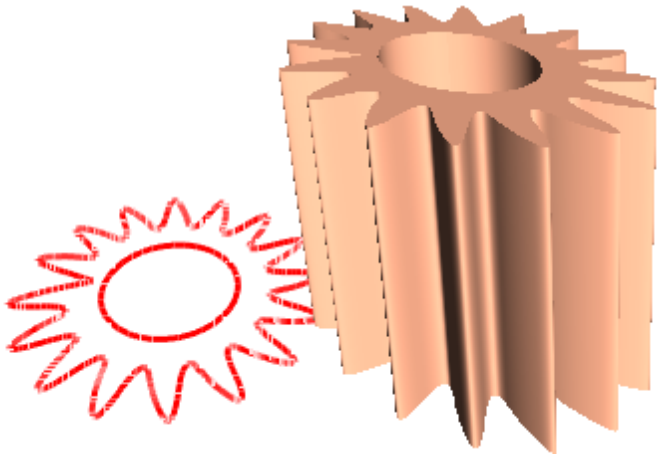
```
   SG_VECTOR extVec = {1,-2,5};

    sgC3DObject* exO = (sgC3DObject*)sgKinematic::Extrude((const sgC2DObject&)
(*spl2_obj),
          (const sgC2DObject**)(&cir),1,extVec,true);

    sgGetScene()->AttachObject(exO);
    exO->SetAttribute(SG_OA_COLOR,30);
```

Then we'll move the obtained solid:

```
    SG_VECTOR transV1 = {-7,0,0};
    exO->InitTempMatrix()->Translate(transV1);
    exO->ApplyTempMatrix();
    exO->DestroyTempMatrix();
```

See also:
sgKinematic::Extrude
sgCSpline sgCSpline::Create SG_SPLINE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.5 Spiral surfaces

Spiral surfaces.

Let's create a spiral surface with an arc-clip.

This code creates an arc:

```
SG_ARC   ArcGeo;
SG_POINT    ArP1 = {-1.0, -3.0, 0.0};
SG_POINT    ArP2 = {-1.0, -2.0, 0.0};
SG_POINT    ArP3 = {0.0, -3.5, 0.0};
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
sgC2DObject*  ar = sgCreateArc(ArcGeo);
sgGetScene()->AttachObject(ar);
ar->SetAttribute(SG_OA_COLOR,12);
ar->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

The axis of the spiral will lie on the (2, -3, 0) and (2, 3, 0) points. As the spiral clip isn't a closed object there will be no holes. Length of a spiral step will be 4, the spiral length - 10, and the number of meridians on the one spiral step circle will be 15:

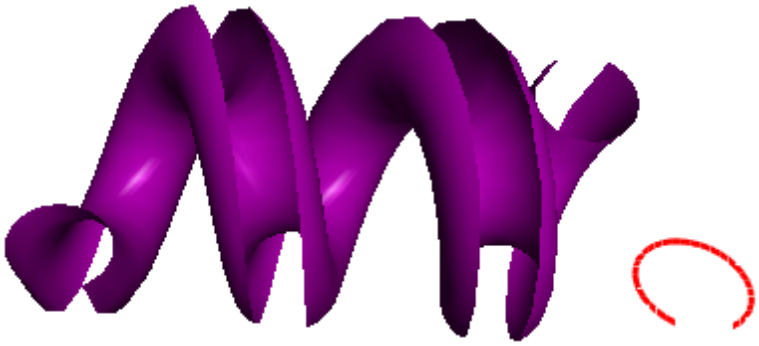
```
SG_POINT axeP1 = {2.0, -3.0, 0.0};
 SG_POINT axeP2 = {2.0, 3.0, 0.0};

  sgC3DObject* spirO = (sgC3DObject*)sgKinematic::Spiral((const sgC2DObject&)
(*ar),NULL,0,
                                                 axeP1,axeP2,4,10,15,false);

  sgGetScene()->AttachObject(spirO);
  spirO->SetAttribute(SG_OA_COLOR,85);
```

Then we'll move the obtained surface:

```
SG_VECTOR transV1 = {0,-11.5,0};
  spirO->InitTempMatrix()->Translate(transV1);
  spirO->ApplyTempMatrix();
  spirO->DestroyTempMatrix();
```

See also:
sgKinematic::Spiral
sgCArc SG_ARC
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.6 Spiral solids

Spiral solids.

Let's create a spiral solid with a circle-clip and a round hole.

This code creates the circles:

```
SG_CIRCLE   cirGeo1;
SG_POINT    cirC1 = {2.0, -2.0, 0.0};
SG_VECTOR    cirNor1 = {0.0, 0.0, 1.0};
cirGeo1.FromCenterRadiusNormal(cirC1,3.0, cirNor1);
sgC2DObject*  cir1 = sgCreateCircle(cirGeo1);
sgGetScene()->AttachObject(cir1);
cir1->SetAttribute(SG_OA_COLOR,12);
cir1->SetAttribute(SG_OA_LINE_THICKNESS, 2);

SG_CIRCLE   cirGeo2;
SG_POINT    cirC2 = {2.0, -2.3, 0.0};
SG_VECTOR    cirNor2 = {0.0, 0.0, 1.0};
cirGeo2.FromCenterRadiusNormal(cirC2,1.5, cirNor2);
sgC2DObject*  cir2 = sgCreateCircle(cirGeo2);
sgGetScene()->AttachObject(cir2);
```

```
cir2->SetAttribute(SG_OA_COLOR,12);
cir2->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

The axis of the spiral will lie on the (6, -3, 0) and (6, 3, 0) points. Length of a spiral step will be 12, the spiral length - 30, and the number of meridians on the one spiral step circle will be 16:

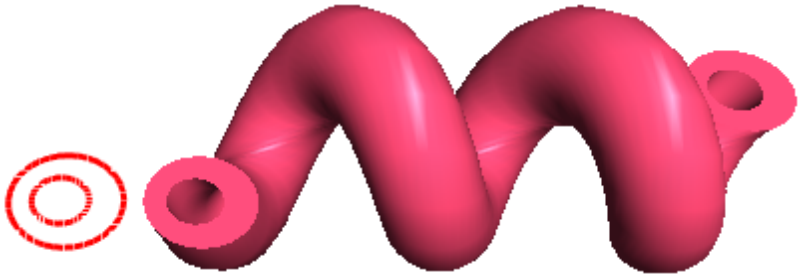
```
SG_POINT axeP1 = {6.0, -3.0, 0.0};
SG_POINT axeP2 = {6.0, 3.0, 0.0};

sgC3DObject* spirO = (sgC3DObject*)sgKinematic::Spiral((const sgC2DObject&)
(*cir1),
        (const sgC2DObject**)(&cir2),1,
        axeP1,axeP2,12,30,16,true);

sgGetScene()->AttachObject(spirO);
spirO->SetAttribute(SG_OA_COLOR,56);
```

Then we'll move the obtained solid:

```
SG_VECTOR transV1 = {0,7,0};
spirO->InitTempMatrix()->Translate(transV1);
spirO->ApplyTempMatrix();
spirO->DestroyTempMatrix();
```

See also:
sgKinematic::Spiral
sgCCircle SG_CIRCLE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.7 Pipe-like surfaces

Pipe-like surfaces.

Let's create a pipe-like surface with an arc-clip and a contour-profile consitsing of arcs and line segments.

This code creates arc-clips:

```
SG_ARC   ArcGeo;
SG_POINT    ArP1 = {1.0, -4.0, 0.0};
SG_POINT    ArP2 = {1.0, -3.6, 0.0};
SG_POINT    ArP3 = {1.2, -3.5, 0.0};
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
sgC2DObject*  ar = sgCreateArc(ArcGeo);
sgGetScene()->AttachObject(ar);
ar->SetAttribute(SG_OA_COLOR,12);
ar->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Now let's create a contour-profile consisting of 2 line segments and 4 arcs:

```
sgCObject*   objcts[6];

objcts[0] = sgCreateLine(0.0, -4.0, 0.0, 0.0, -2.0, 0.0);

ArP1.x = 0.0; ArP1.y = -2.0; ArP1.z = 0.0;
ArP2.x = 1.0; ArP2.y = -1.0; ArP2.z = 0.0;
ArP3.x = 0.4; ArP3.y = -1.2; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[1] = sgCreateArc(ArcGeo);

ArP1.x = 1.0; ArP1.y = -1.0; ArP1.z = 0.0;
ArP2.x = 2.0; ArP2.y = 0.0; ArP2.z = 0.0;
ArP3.x = 1.9; ArP3.y = -0.5; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[2] = sgCreateArc(ArcGeo);

ArP1.x = 2.0; ArP1.y = 0.0; ArP1.z = 0.0;
ArP2.x = 1.0; ArP2.y = 1.0; ArP2.z = 0.0;
ArP3.x = 1.6; ArP3.y = 0.8; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[3] = sgCreateArc(ArcGeo);

objcts[4] = sgCreateLine(1.0, 1.0, 0.0, -1.0, 1.0, 0.0);

ArP1.x = -1.0; ArP1.y = 1.0; ArP1.z = 0.0;
ArP2.x = -1.0; ArP2.y = 0.0; ArP2.z = 1.0;
ArP3.x = -1.1; ArP3.y = 1.0; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[5] = sgCreateArc(ArcGeo);

sgCContour* cnt1 = sgCContour::CreateContour(objcts,6);
sgGetScene()->AttachObject(cnt1);
cnt1->SetAttribute(SG_OA_COLOR,12);
cnt1->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Further you must define both a point in the arc-clip plane which will move along the profile, and a rotation angle around this point. Let this point be (1, -4, 0) and the rotation angle 0 degrees. Let's build a surface:

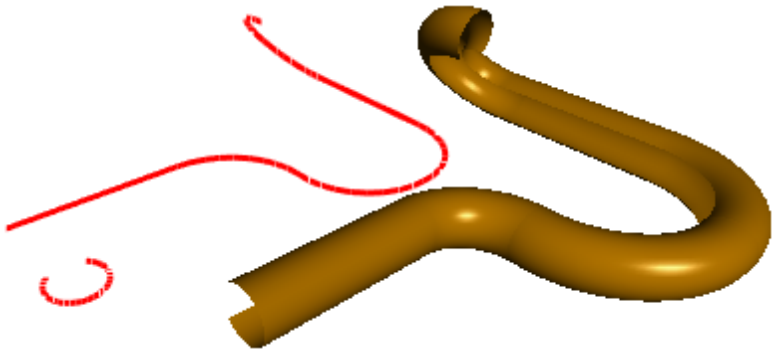
```
   SG_POINT point_in_plane = {1.0, -4.0, 0.0};

   bool  close = false;
   sgC3DObject* pipO = (sgC3DObject*)sgKinematic::Pipe((const sgC2DObject&)(*ar),
NULL,0,
          (const sgC2DObject&)(*cnt1), point_in_plane, 0.0, close);

   sgGetScene()->AttachObject(pipO);
   pipO->SetAttribute(SG_OA_COLOR,25);
```

Then we'll move the obtained surface:

```
 SG_VECTOR transV1 = {2.5,1,0};
   pipO->InitTempMatrix()->Translate(transV1);
   pipO->ApplyTempMatrix();
   pipO->DestroyTempMatrix();
```

See also:
sgKinematic::Pipe
sgCArc SG_ARC
sgCLine SG_LINE
sgCContour sgCContour::CreateContour
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.3.8 Pipe-like solids

Pipe-like solids.

Let's create a pipe-like solid with a contour-clip, round hole and contour-profile consitsing of arcs and line segments.

This code creates a contour-clip and a round hole:

```
   sgCObject*   objcts[7];
```

```
SG_POINT    ArP1;
SG_POINT    ArP2;
SG_POINT    ArP3;
SG_ARC      ArcGeo;

ArP1.x = -0.2; ArP1.y = -0.2; ArP1.z = 0.0;
ArP2.x = -0.1; ArP2.y = 0.2; ArP2.z = 0.0;
ArP3.x = -0.3; ArP3.y = 0.1; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[0] = sgCreateArc(ArcGeo);


ArP1.x = -0.1; ArP1.y = 0.2; ArP1.z = 0.0;
ArP2.x = 0.3; ArP2.y = 0.5; ArP2.z = 0.0;
ArP3.x = 0.2; ArP3.y = 0.6; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[1] = sgCreateArc(ArcGeo);

ArP1.x = 0.3; ArP1.y = 0.5; ArP1.z = 0.0;
ArP2.x = -0.2; ArP2.y = -0.2; ArP2.z = 0.0;
ArP3.x = 0.6; ArP3.y = -0.4; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[2] = sgCreateArc(ArcGeo);

sgCContour* cnt2 = sgCContour::CreateContour(objcts,3);
sgGetScene()->AttachObject(cnt2);
cnt2->SetAttribute(SG_OA_COLOR,12);
cnt2->SetAttribute(SG_OA_LINE_THICKNESS, 2);

SG_CIRCLE   cirGeo;
SG_POINT    cirC = {0.3, -0.1, 0.0};
SG_VECTOR    cirNor = {0.0, 0.0, 1.0};
cirGeo.FromCenterRadiusNormal(cirC,0.31, cirNor);
sgC2DObject*  cir = sgCreateCircle(cirGeo);
sgGetScene()->AttachObject(cir);
cir->SetAttribute(SG_OA_COLOR,12);
cir->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Now let's create a contour-profile consisting of 1 line segment and 4 arcs:

```
ArP1.x = 0.0; ArP1.y = -2.0; ArP1.z = 0.0;
ArP2.x = 1.0; ArP2.y = -1.0; ArP2.z = 0.0;
ArP3.x = 0.4; ArP3.y = -1.2; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[0] = sgCreateArc(ArcGeo);

ArP1.x = 1.0; ArP1.y = -1.0; ArP1.z = 0.0;
ArP2.x = 2.0; ArP2.y = 0.0; ArP2.z = 0.0;
ArP3.x = 1.9; ArP3.y = -0.5; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[1] = sgCreateArc(ArcGeo);
```

```
ArP1.x = 2.0; ArP1.y = 0.0; ArP1.z = 0.0;
ArP2.x = 1.0; ArP2.y = 1.0; ArP2.z = 0.0;
ArP3.x = 1.6; ArP3.y = 0.8; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[2] = sgCreateArc(ArcGeo);

objcts[3] = sgCreateLine(1.0, 1.0, 0.0, -1.0, 1.0, 0.0);

ArP1.x = -1.0; ArP1.y = 1.0; ArP1.z = 0.0;
ArP2.x = -1.0; ArP2.y = 0.0; ArP2.z = 1.0;
ArP3.x = -1.1; ArP3.y = 1.0; ArP3.z = 0.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
objcts[4] = sgCreateArc(ArcGeo);

sgCContour* cnt1 = sgCContour::CreateContour(objcts,5);
sgGetScene()->AttachObject(cnt1);
cnt1->SetAttribute(SG_OA_COLOR,12);
cnt1->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Further you must define both a point in the arc-clip plane which will move along the profile, and a rotation angle around this point. Let this point be (0, 0, 0) and the rotation angle 0 degrees. Let's build a surface:

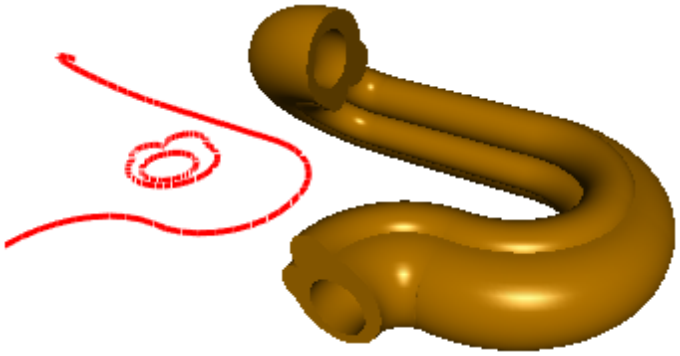
```
SG_POINT point_in_plane = {0.0,0.0,0.0};

bool  close = true;
sgC3DObject* pipO = (sgC3DObject*)sgKinematic::Pipe((const sgC2DObject&)
(*cnt2),
        (const sgC2DObject**)(&cir),1,
        (const sgC2DObject&)(*cnt1), point_in_plane, 0.0, close);

sgGetScene()->AttachObject(pipO);
pipO->SetAttribute(SG_OA_COLOR,25);
```

Then let's move the obtained surface:

```
SG_VECTOR transV1 = {3.0,1.0,0};
pipO->InitTempMatrix()->Translate(transV1);
pipO->ApplyTempMatrix();
pipO->DestroyTempMatrix();
```

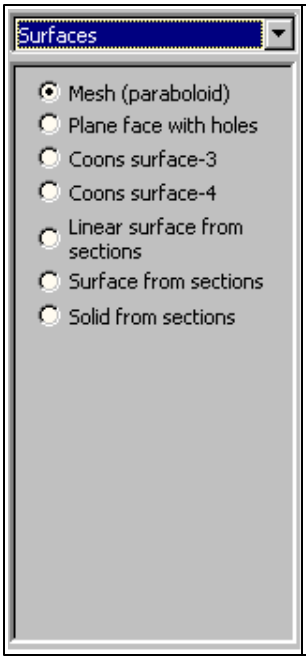
See also:
sgKinematic::Pipe
sgCArc SG_ARC
sgCLine SG_LINE
sgCContour sgCContour::CreateContour
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4 Surfaces

Surfaces.



Mesh
Flat face with holes
Coons surface by three contours
Coons surface by four contours
Ruled surface by clips
Spline surface by clips
Spline solid by clips

16.4.1 Mesh

Mesh.

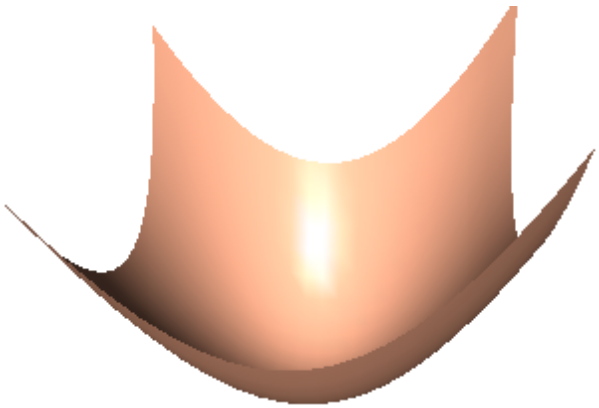
Let's create a mesh by a two-dimensional array of the points lying on a paraboloid:

```
int x_sz = 20;
int y_sz =20;
```

```
        SG_POINT*  pnts = new SG_POINT[x_sz*y_sz];

        for (int i=0;i<x_sz;i++)
               for (int j=0;j<y_sz;j++)
               {
                      pnts[i*y_sz+j].x = i-x_sz/2;
                      pnts[i*y_sz+j].y = j-y_sz/2;
                      pnts[i*y_sz+j].z = 0.1*(pnts[i*y_sz+j].x*pnts[i*y_sz+j].x+
                                                    pnts[i*y_sz+j].y*pnts[i*y_sz+j]
.y);
               }

        sgCObject* msh= sgSurfaces::Mesh(x_sz,y_sz,pnts);

        delete [] pnts;

        sgGetScene()->AttachObject(msh);
        msh->SetAttribute(SG_OA_COLOR,30);
        msh->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

See also:
sgSurfaces::Mesh
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4.2 Flat face with holes

Flat face with holes.

Let's create a flat face with a spline as an outer contour and three circles as holes.

The code of creating the spline as an outer contour:

```
        SG_POINT tmpPnt;
```

```
SG_SPLINE* spl2 = SG_SPLINE::Create();

tmpPnt.x = -1.0; tmpPnt.y = -3.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,0);
tmpPnt.x = -3.0; tmpPnt.y = 0.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,1);
tmpPnt.x = -1.0; tmpPnt.y = -1.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,2);
tmpPnt.x = 0.0; tmpPnt.y = 1.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,3);
tmpPnt.x = -1.0; tmpPnt.y = 4.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,4);
tmpPnt.x =3.0; tmpPnt.y = 1.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,5);
tmpPnt.x =2.0; tmpPnt.y = -3.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,6);
tmpPnt.x =1.0; tmpPnt.y = -1.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,7);
tmpPnt.x =1.0; tmpPnt.y = -4.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,8);
spl2->Close();

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
SG_SPLINE::Delete(spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,0);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

Holes:

```
sgC2DObject*  holes[3];

SG_CIRCLE   cirGeo;
SG_POINT    cirC = {0.8, 1.0, 0.0};
SG_VECTOR    cirNor = {0.0, 0.0, 1.0};
cirGeo.FromCenterRadiusNormal(cirC,0.8, cirNor);
holes[0] = sgCreateCircle(cirGeo);
sgGetScene()->AttachObject(holes[0]);
holes[0]->SetAttribute(SG_OA_COLOR,0);
holes[0]->SetAttribute(SG_OA_LINE_THICKNESS, 1);

cirC.x = 1.6; cirC.y = -1.0;
cirGeo.FromCenterRadiusNormal(cirC,0.2, cirNor);
holes[1] = sgCreateCircle(cirGeo);
sgGetScene()->AttachObject(holes[1]);
holes[1]->SetAttribute(SG_OA_COLOR,0);
holes[1]->SetAttribute(SG_OA_LINE_THICKNESS, 1);

cirC.x = 0.0; cirC.y = -1.0;
cirGeo.FromCenterRadiusNormal(cirC,0.4, cirNor);
holes[2] = sgCreateCircle(cirGeo);
sgGetScene()->AttachObject(holes[2]);
holes[2]->SetAttribute(SG_OA_COLOR,0);
```

```
holes[2]->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

Constructing a face with holes:

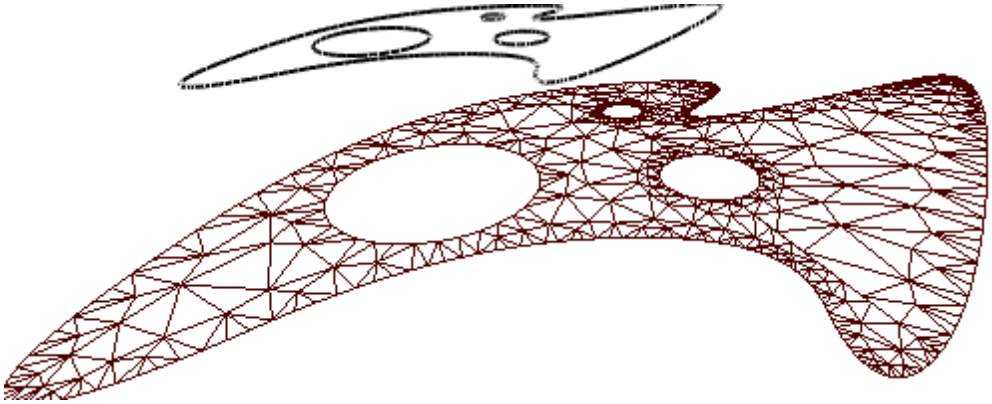
```
sgC3DObject* fcO = (sgC3DObject*)sgSurfaces::Face((const sgC2DObject&)
(*spl2_obj),
       (const sgC2DObject**)(&holes[0]),3);

sgGetScene()->AttachObject(fcO);
fcO->SetAttribute(SG_OA_COLOR,50);
```

Then let's move the obtained surface:

```
SG_VECTOR transV1 = {-5,0,0};
fcO->InitTempMatrix()->Translate(transV1);
fcO->ApplyTempMatrix();
fcO->DestroyTempMatrix();
```

See also:
sgSurcfaces::Face
sgCSpline SG_SPLINE
sgCCircle SG_CIRCLE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4.3 Coons surface by three boundary contours

Coons surface by three boundary contours.

Let's create a Coons surface by three boundary contours - two splines and one arc. The necessary requirment when creating the Coons surface is matching the end points of boundary contours.

This code creates the first spline:

```
SG_POINT tmpPnt;

SG_SPLINE* spl1 = SG_SPLINE::Create();

tmpPnt.x = -6.0; tmpPnt.y = 0.0; tmpPnt.z = -1.0;
spl1->AddKnot(tmpPnt,0);
tmpPnt.x = -5.0; tmpPnt.y = 0.0; tmpPnt.z = -2.0;
spl1->AddKnot(tmpPnt,1);
tmpPnt.x = -3.0; tmpPnt.y = 0.0; tmpPnt.z = -1.0;
spl1->AddKnot(tmpPnt,2);
tmpPnt.x = -2.0; tmpPnt.y = 0.0; tmpPnt.z = -2.0;
spl1->AddKnot(tmpPnt,3);
tmpPnt.x = -1.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl1->AddKnot(tmpPnt,4);
tmpPnt.x =2.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl1->AddKnot(tmpPnt,5);
tmpPnt.x =2.0; tmpPnt.y = 0.0; tmpPnt.z = 0.0;
spl1->AddKnot(tmpPnt,6);
tmpPnt.x =4.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl1->AddKnot(tmpPnt,7);

sgCSpline* spl1_obj = sgCreateSpline(*spl1);
SG_SPLINE::Delete(spl1);
sgGetScene()->AttachObject(spl1_obj);
spl1_obj->SetAttribute(SG_OA_COLOR,0);
spl1_obj->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

This code creates the second spline:

```
SG_SPLINE* spl2 = SG_SPLINE::Create();

tmpPnt.x = -6.0; tmpPnt.y = 0.0; tmpPnt.z = -1.0;
spl2->AddKnot(tmpPnt,0);
tmpPnt.x = -6.0; tmpPnt.y = 0.5; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,1);
tmpPnt.x = -6.0; tmpPnt.y = 2.0; tmpPnt.z = -1.0;
spl2->AddKnot(tmpPnt,2);
tmpPnt.x = -6.0; tmpPnt.y = 3.0; tmpPnt.z = 1.0;
spl2->AddKnot(tmpPnt,3);
tmpPnt.x =-4.0; tmpPnt.y = 4.0; tmpPnt.z = 1.0;
spl2->AddKnot(tmpPnt,4);

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
SG_SPLINE::Delete(spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,50);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 3);
```

This code creates the arc:

```
SG_POINT    ArP1;
SG_POINT    ArP2;
SG_POINT    ArP3;
SG_ARC      ArcGeo;
```

```
ArP1.x = -4.0; ArP1.y = 4.0; ArP1.z = 1.0;
ArP2.x = 4.0; ArP2.y = 0.0; ArP2.z = 1.0;
ArP3.x = 0.0; ArP3.y = 5.0; ArP3.z = -4.0;
ArcGeo.FromTreePoints(ArP1,ArP2,ArP3,false);
sgCArc* arcObj = sgCreateArc(ArcGeo);
```

Let's create the Coons surface itself:

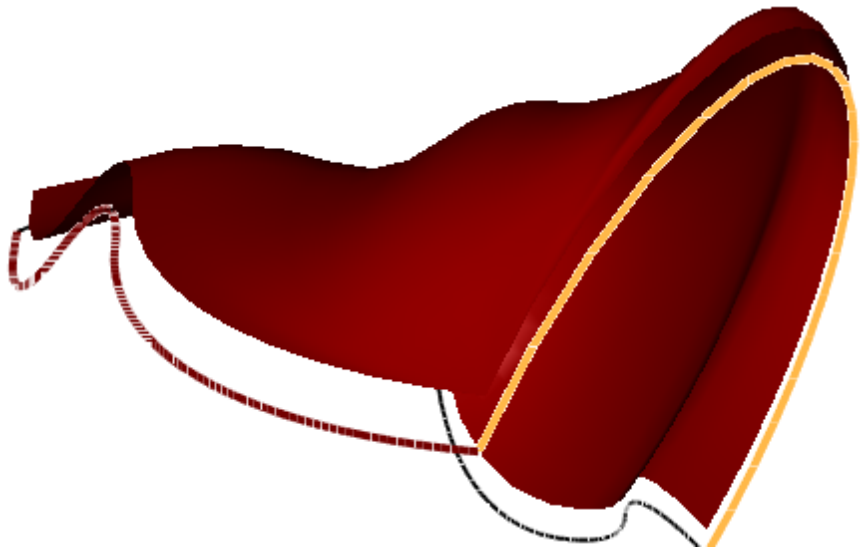
```
sgCObject* coons = sgSurfaces::Coons((const sgC2DObject&)*spl1_obj,
        (const sgC2DObject&)*spl2_obj,
        (const sgC2DObject&)*arcObj,
        NULL,36,36,4,4);

sgGetScene()->AttachObject(coons);
coons->SetAttribute(SG_OA_COLOR,4);
```

Then let's move the obtained surface:

```
SG_VECTOR transV1 = {0,0,-0.4};
coons->InitTempMatrix()->Translate(transV1);
coons->ApplyTempMatrix();
coons->DestroyTempMatrix();
```

See also:
sgSurcfaces::Coons
sgCSpline SG_SPLINE
sgCArc SG_ARC
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4.4 Coons surface by four boundary contours

Coons surface by four boundary contours.

Let's create a Coons surface by four boundary contours - four splines.
The necessary requirment when creating the Coons surface is matching the end points of boundary contours.

This code creates the first spline:

```
SG_POINT tmpPnt;

SG_SPLINE* spl1 = SG_SPLINE::Create();

tmpPnt.x = -6.0; tmpPnt.y = 0.0; tmpPnt.z = -1.0;
spl1->AddKnot(tmpPnt,0);
tmpPnt.x = -5.0; tmpPnt.y = 0.0; tmpPnt.z = 2.0;
spl1->AddKnot(tmpPnt,1);
tmpPnt.x = -3.0; tmpPnt.y = 0.0; tmpPnt.z = -2.0;
spl1->AddKnot(tmpPnt,2);
tmpPnt.x = -2.0; tmpPnt.y = 0.0; tmpPnt.z = -2.0;
spl1->AddKnot(tmpPnt,3);
tmpPnt.x = -1.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl1->AddKnot(tmpPnt,4);
tmpPnt.x =2.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl1->AddKnot(tmpPnt,5);
tmpPnt.x =2.0; tmpPnt.y = 0.0; tmpPnt.z = 0.0;
spl1->AddKnot(tmpPnt,6);
tmpPnt.x =4.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl1->AddKnot(tmpPnt,7);

sgCSpline* spl1_obj = sgCreateSpline(*spl1);
SG_SPLINE::Delete(spl1);
sgGetScene()->AttachObject(spl1_obj);
spl1_obj->SetAttribute(SG_OA_COLOR,0);
spl1_obj->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

This code creates the second spline:

```
SG_SPLINE* spl2 = SG_SPLINE::Create();

tmpPnt.x = -5.0; tmpPnt.y = 4.0; tmpPnt.z = 2.0;
spl2->AddKnot(tmpPnt,0);
tmpPnt.x = -3.0; tmpPnt.y = 4.0; tmpPnt.z = 1.0;
spl2->AddKnot(tmpPnt,1);
tmpPnt.x = -2.0; tmpPnt.y = 4.0; tmpPnt.z = 2.0;
spl2->AddKnot(tmpPnt,2);
tmpPnt.x = -1.0; tmpPnt.y = 4.0; tmpPnt.z = -1.0;
spl2->AddKnot(tmpPnt,3);
tmpPnt.x =4.0; tmpPnt.y = 4.0; tmpPnt.z = 0.0;
spl2->AddKnot(tmpPnt,4);

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
```

```
SG_SPLINE::Delete(spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,10);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

This code creates the third spline:

```
SG_SPLINE* spl3 = SG_SPLINE::Create();

tmpPnt.x = -6.0; tmpPnt.y = 0.0; tmpPnt.z = -1.0;
spl3->AddKnot(tmpPnt,0);
tmpPnt.x = -7.0; tmpPnt.y = 0.5; tmpPnt.z = 3.0;
spl3->AddKnot(tmpPnt,1);
tmpPnt.x = -4.0; tmpPnt.y = 2.0; tmpPnt.z = -1.0;
spl3->AddKnot(tmpPnt,2);
tmpPnt.x = -5.0; tmpPnt.y = 3.0; tmpPnt.z = 1.0;
spl3->AddKnot(tmpPnt,3);
tmpPnt.x =-5.0; tmpPnt.y = 4.0; tmpPnt.z = 2.0;
spl3->AddKnot(tmpPnt,4);

sgCSpline* spl3_obj = sgCreateSpline(*spl3);
SG_SPLINE::Delete(spl3);
sgGetScene()->AttachObject(spl3_obj);
spl3_obj->SetAttribute(SG_OA_COLOR,50);
spl3_obj->SetAttribute(SG_OA_LINE_THICKNESS, 3);
```

This code creates the forth spline:

```
SG_SPLINE* spl4 = SG_SPLINE::Create();

tmpPnt.x = 4.0; tmpPnt.y = 0.0; tmpPnt.z = 1.0;
spl4->AddKnot(tmpPnt,0);
tmpPnt.x = 3.0; tmpPnt.y = 1.0; tmpPnt.z = 2.0;
spl4->AddKnot(tmpPnt,1);
tmpPnt.x = 5.0; tmpPnt.y = 2.0; tmpPnt.z = -1.0;
spl4->AddKnot(tmpPnt,2);
tmpPnt.x = 3.0; tmpPnt.y = 3.0; tmpPnt.z = 0.5;
spl4->AddKnot(tmpPnt,3);
tmpPnt.x =4.0; tmpPnt.y = 4.0; tmpPnt.z = 0.0;
spl4->AddKnot(tmpPnt,4);

sgCSpline* spl4_obj = sgCreateSpline(*spl4);
SG_SPLINE::Delete(spl4);
sgGetScene()->AttachObject(spl4_obj);
spl4_obj->SetAttribute(SG_OA_COLOR,100);
spl4_obj->SetAttribute(SG_OA_LINE_THICKNESS, 4);
```

Let's create the Coons surface itself:

```
sgCObject* coons = sgSurfaces::Coons((const sgC2DObject&)*spl1_obj,
                                                                    (const
sgC2DObject&)*spl2_obj,
                                                                    (const
sgC2DObject&)*spl3_obj,
                                                                    (const
```

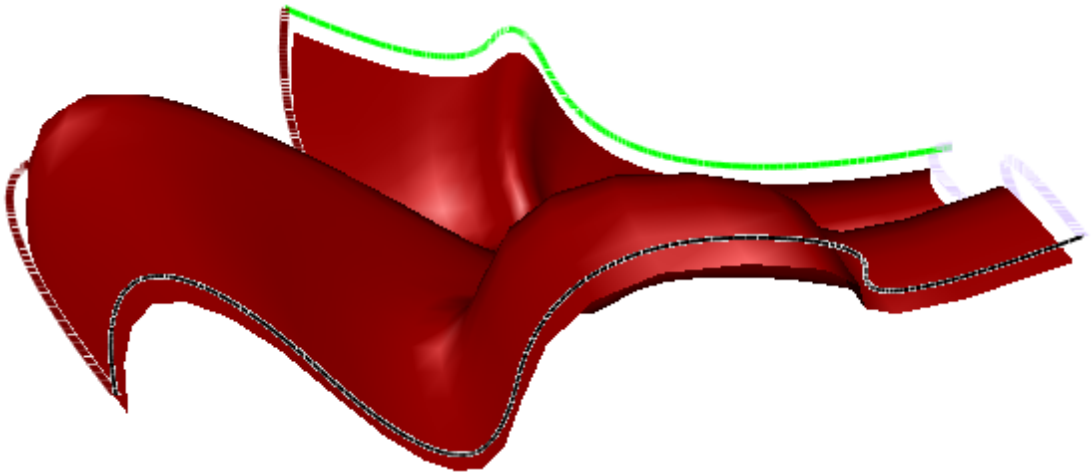
```
sgC2DObject*)spl4_obj,36,36,4,4);
        sgGetScene()->AttachObject(coons);
        coons->SetAttribute(SG_OA_COLOR,4);
```

Then let's move the obtained surface:

```
SG_VECTOR transV1 = {0,0,-0.3};
coons->InitTempMatrix()->Translate(transV1);
coons->ApplyTempMatrix();
coons->DestroyTempMatrix();
```

See also:
sgSurcfaces::Coons
sgCSpline SG_SPLINE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4.5 Ruled surface from clips

Ruled surface from clips.

Let's create three ruled surfaces from two contours - a circle and a spline with different spiral degrees.

To start with, let's draw a circle and a spline:

```
SG_POINT tmpPnt;

SG_SPLINE* spl1 = SG_SPLINE::Create();
int fl=0;
for (double i=0.0;i<2.0*3.14159265;i+=0.4)
{
        tmpPnt.x = ((double)(fl%3+2))*cos(i);
        tmpPnt.y = ((double)(fl%3+2))*sin(i);
```

```
        tmpPnt.z = 0.0;
        spl1->AddKnot(tmpPnt,fl);
        fl++;
}
spl1->Close();

sgCSpline* spl1_obj = sgCreateSpline(*spl1);
sgGetScene()->AttachObject(spl1_obj);
spl1_obj->SetAttribute(SG_OA_COLOR,12);
spl1_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

SG_SPLINE::Delete(spl1);

SG_CIRCLE cirGeo;
cirGeo.center.x = 2.0; cirGeo.center.y = -2.0; cirGeo.center.z = 8.0;
cirGeo.normal.x = 0.0; cirGeo.normal.y = 0.0; cirGeo.normal.z = 1.0;
cirGeo.radius = 1.5;
sgCCircle* cir = sgCreateCircle(cirGeo);
sgGetScene()->AttachObject(cir);
cir->SetAttribute(SG_OA_COLOR,12);
cir->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Then we'll construct the ruled surfaces and move them at once.

The first surface:

```
sgC3DObject* linO1 = (sgC3DObject*)sgSurfaces::LinearSurfaceFromSections((const
                                    sgC2DObject&)(*spl1_obj),
                                    (const sgC2DObject&)(*cir),0.5,false);

sgGetScene()->AttachObject(linO1);
linO1->SetAttribute(SG_OA_COLOR,90);

SG_VECTOR transV1 = {0,7,0};
linO1->InitTempMatrix()->Translate(transV1);
linO1->ApplyTempMatrix();
linO1->DestroyTempMatrix();
```

The second surface:

```
sgC3DObject* linO2 = (sgC3DObject*)sgSurfaces::LinearSurfaceFromSections((const
sgC2DObject&)(*spl1_obj),
        (const sgC2DObject&)(*cir),0.7,false);

sgGetScene()->AttachObject(linO2);
linO2->SetAttribute(SG_OA_COLOR,150);

transV1.x = 8.0; transV1.y = 0.0;
linO2->InitTempMatrix()->Translate(transV1);
linO2->ApplyTempMatrix();
linO2->DestroyTempMatrix();
```

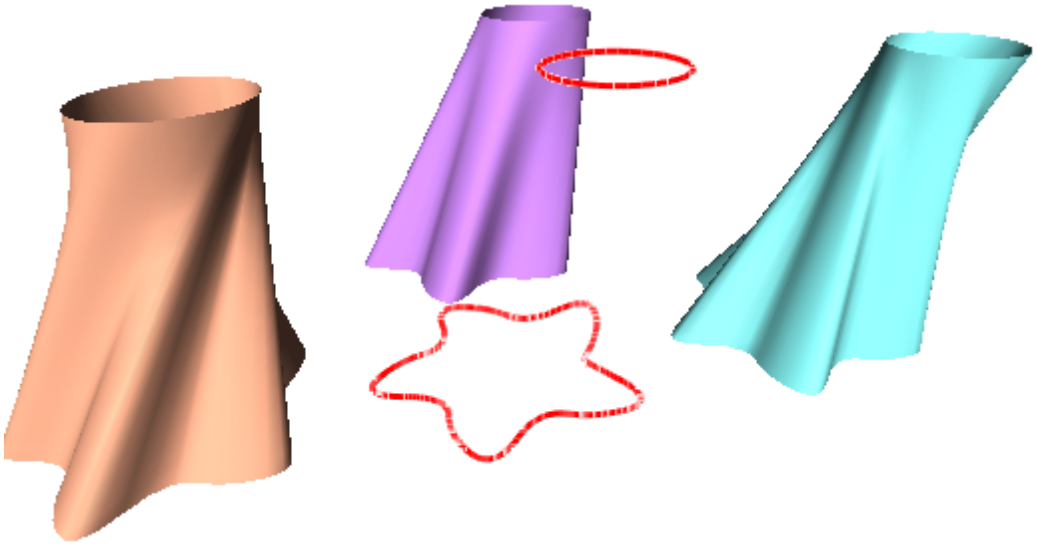
The third surface:

```
sgC3DObject* linO3 = (sgC3DObject*)sgSurfaces::LinearSurfaceFromSections((const
sgC2DObject&)(*spl1_obj),
```

```
        (const sgC2DObject&)(*cir),0.3,false);

    sgGetScene()->AttachObject(lin03);
    lin03->SetAttribute(SG_OA_COLOR,30);

    transV1.x = -8.0; transV1.y = 0.0;
    lin03->InitTempMatrix()->Translate(transV1);
    lin03->ApplyTempMatrix();
    lin03->DestroyTempMatrix();
```

See also:
sgSurcfaces::LinearSurfaceFromSections
sgCSpline SG_SPLINE
sgCCircle SG_CIRCLE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4.6 Spline surface from clips

Spline surface from clips.

Let's create a smooth surface from three clips - two splines, a line segment and an arc.

This code creates the clips:

```
    SG_POINT tmpPnt;

    SG_SPLINE* spl1 = SG_SPLINE::Create();
```

```
tmpPnt.x = 98.0; tmpPnt.y = 0.0; tmpPnt.z = -13.0;
spl1->AddKnot(tmpPnt,0);
tmpPnt.x = 85.0; tmpPnt.y = 0.0; tmpPnt.z = 19.0;
spl1->AddKnot(tmpPnt,1);
tmpPnt.x = 43.0; tmpPnt.y = 0.0; tmpPnt.z = -31.0;
spl1->AddKnot(tmpPnt,2);
tmpPnt.x = 5.0; tmpPnt.y = 0.0; tmpPnt.z = -3.0;
spl1->AddKnot(tmpPnt,3);
tmpPnt.x = -11.0; tmpPnt.y = 0.0; tmpPnt.z = -39.0;
spl1->AddKnot(tmpPnt,4);
tmpPnt.x =-48.0; tmpPnt.y = 0.0; tmpPnt.z = 23.0;
spl1->AddKnot(tmpPnt,5);
tmpPnt.x =-125.0; tmpPnt.y = 0.0; tmpPnt.z = 23.0;
spl1->AddKnot(tmpPnt,6);

sgCSpline* spl1_obj = sgCreateSpline(*spl1);
SG_SPLINE::Delete(spl1);
sgGetScene()->AttachObject(spl1_obj);
spl1_obj->SetAttribute(SG_OA_COLOR,12);
spl1_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

SG_SPLINE* spl2 = SG_SPLINE::Create();

tmpPnt.x = 96.0; tmpPnt.y = 150.0; tmpPnt.z = 8.0;
spl2->AddKnot(tmpPnt,0);
tmpPnt.x = 66.0; tmpPnt.y = 150.0; tmpPnt.z = -20.0;
spl2->AddKnot(tmpPnt,1);
tmpPnt.x = 12.0; tmpPnt.y = 150.0; tmpPnt.z = 37.0;
spl2->AddKnot(tmpPnt,2);
tmpPnt.x = -128.0; tmpPnt.y = 150.0; tmpPnt.z = -23.0;
spl2->AddKnot(tmpPnt,3);

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
SG_SPLINE::Delete(spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,12);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

sgCLine* ln_obj = sgCreateLine(100.0,100.0,50.0, -121.0,100.0,-50.0);
sgGetScene()->AttachObject(ln_obj);
ln_obj->SetAttribute(SG_OA_COLOR,12);
ln_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

SG_ARC arcG;
SG_POINT arcBeg = {98.0, 50.0 , -80.0};
SG_POINT arcEnd = {-117.0, 50.0 , -80.0};
SG_POINT arcMid = {-55.0, 50.0 , -50.0};
arcG.FromTreePoints(arcBeg,arcEnd,arcMid,false);
sgCArc*  arc_obj = sgCreateArc(arcG);
sgGetScene()->AttachObject(arc_obj);
arc_obj->SetAttribute(SG_OA_COLOR,12);
arc_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);
```

Let's create the surface itself from these clips. As the clips are unclosed only contour orientations will be considered when constructing the surface - the contours had the same orientation by its construction. Let's specify the zero array as the array of parameters:

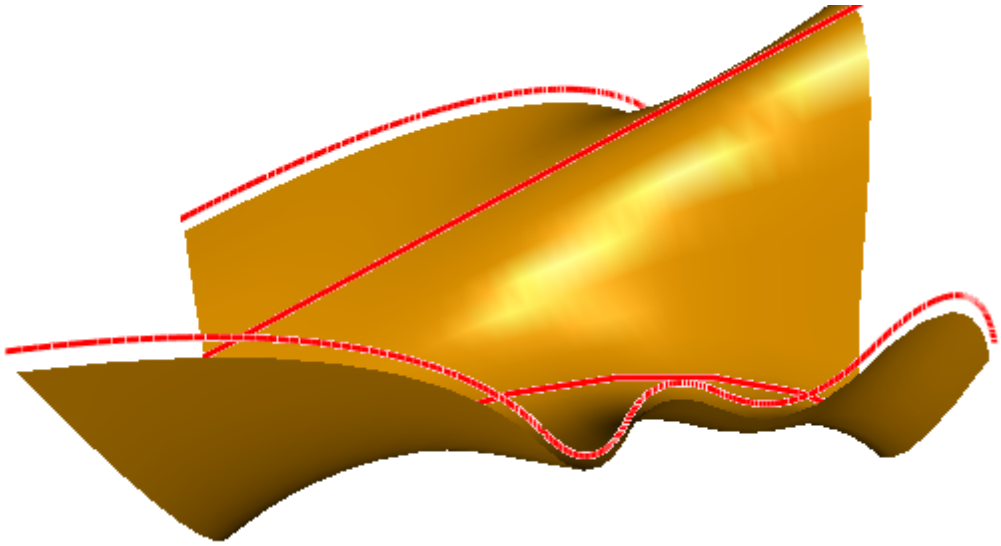
```
   sgC2DObject*  objcts[4];
   objcts[0] = spl1_obj;
   objcts[1] = arc_obj;
   objcts[2] = ln_obj;
   objcts[3] = spl2_obj;

   double param[4];
   param[0] = param[1] = param[2] = param[3] = 0.0;

   sgCObject* surf = sgSurfaces::SplineSurfaceFromSections((const sgC2DObject**)
(objcts),
          param,4,false);
   sgGetScene()->AttachObject(surf);
   surf->SetAttribute(SG_OA_COLOR,24);
```

Then let's move the obtained surface:

```
  SG_VECTOR transV1 = {0,0,-5};
   surf->InitTempMatrix()->Translate(transV1);
   surf->ApplyTempMatrix();
   surf->DestroyTempMatrix();
```

See also:
sgSurcfaces::SplineSurfaceFromSections
sgCSpline SG_SPLINE
sgCArc SG_ARC
sgCLine SG_LINE
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.4.7 Spline solid from clips

Spline solid from clips.

Let's create a smooth solid from three clips - splines. Two of the splines are identical but moved and turned one about another.

This code creates two clips:

```
SG_POINT tmpPnt;
SG_SPLINE* spl1 = SG_SPLINE::Create();
int fl=0;
for (double i=0.0;i<2.0*3.14159265;i+=0.4)
{
       tmpPnt.x = ((double)(fl%3+2))*cos(i);
       tmpPnt.y = ((double)(fl%3+2))*sin(i);
       tmpPnt.z = 0.0;
       spl1->AddKnot(tmpPnt,fl);
       fl++;
}
spl1->Close();

sgCSpline* spl1_obj = sgCreateSpline(*spl1);
SG_SPLINE::Delete(spl1);
sgGetScene()->AttachObject(spl1_obj);
spl1_obj->SetAttribute(SG_OA_COLOR,12);
spl1_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

SG_SPLINE* spl2 = SG_SPLINE::Create();

tmpPnt.x = 0.0; tmpPnt.y = -0.9; tmpPnt.z = 2.0;
```

```
spl2->AddKnot(tmpPnt,0);
tmpPnt.x = 1.4; tmpPnt.y = 0.9; tmpPnt.z = 2.0;
spl2->AddKnot(tmpPnt,1);
tmpPnt.x = -1.6; tmpPnt.y = 0.6; tmpPnt.z = 2.0;
spl2->AddKnot(tmpPnt,2);
tmpPnt.x = -1.2; tmpPnt.y = -1.6; tmpPnt.z = 2.0;
spl2->AddKnot(tmpPnt,3);
spl2->Close();

sgCSpline* spl2_obj = sgCreateSpline(*spl2);
SG_SPLINE::Delete(spl2);
sgGetScene()->AttachObject(spl2_obj);
spl2_obj->SetAttribute(SG_OA_COLOR,12);
spl2_obj->SetAttribute(SG_OA_LINE_THICKNESS, 2);

sgGetScene()->AttachObject(spl1_obj);
sgGetScene()->AttachObject(spl2_obj);
```

The third clip will be the copy of the first one but moved and turned:

```
sgC2DObject* ooo[3];
ooo[1] = spl1_obj;
ooo[0] = spl2_obj;
ooo[2] = (sgC2DObject*)spl2_obj->Clone();

sgGetScene()->AttachObject(ooo[2]);

SG_POINT axeP = {0.0, 0.0, 0.0};
SG_VECTOR axeD = {0.0, 0.0, 1.0};
axeD.z = 0.0; axeD.x = 1.0;
ooo[2]->InitTempMatrix()->Rotate(axeP,axeD, 3.14159265/4.0);
SG_VECTOR trV = {-1.0, 1.0, -3.0};
ooo[2]->GetTempMatrix()->Translate(trV);
ooo[2]->ApplyTempMatrix();
ooo[2]->DestroyTempMatrix();
```

This code creates the solid itself:

```
double ppp[3];
ppp[0] = 0.1;
ppp[1] = 0.0;
ppp[2] = 0.2;


sgCObject* lsf = sgSurfaces::SplineSurfaceFromSections((const sgC2DObject**)
(&ooo[0]),
        ppp,3,true);

sgGetScene()->AttachObject(lsf);
lsf->SetAttribute(SG_OA_COLOR,23);
```

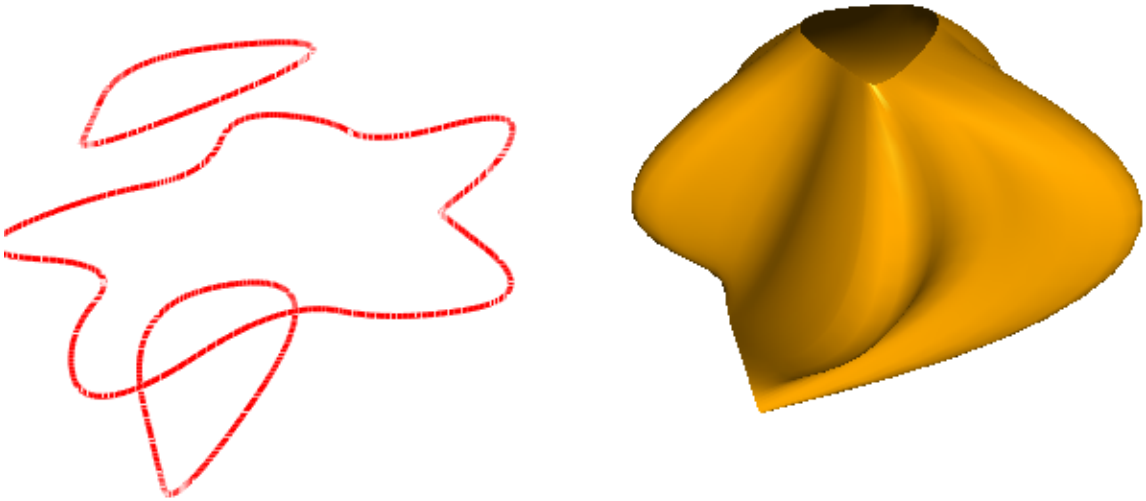
Then let's move the obtained solid:

```
SG_VECTOR transV1 = {8,0,0};
```

```
lsf->InitTempMatrix()->Translate(transV1);
lsf->ApplyTempMatrix();
lsf->DestroyTempMatrix();
```

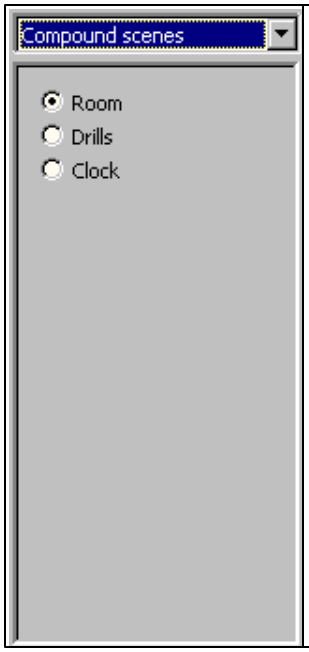
See also:
sgSurcfaces::SplineSurfaceFromSections
sgCSpline SG_SPLINE
sgCObject::Clone
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Illustration:



16.5 Composite scenes

Composite scenes.



Room
Drills
Clock

16.5.1 Room

Room.

Let's create in sequence, object by object, such a scene as a room - two walls, a floor, a plinth, a standard lamp and a table.

To start with let's create the walls and the floor.

Let's define a parameterization and set up the parameters of the length (m_room_X_size), the width (m_room_Y_size) and the height (m_room_Z_size) of the room.

Let's create three similar contours consisting of two line segments along the length and the width of the room. We shall use the first contour to create the base contour of the walls, the second one to create the floor and the third one as the plinth profile.

This code creates the contour and duplicates it:

```
 sgCObject*   objects_buffer[10];
objects_buffer[0] = sgCreateLine(0.0, 0.0, 0.0, m_room_X_size, 0.0, 0.0);
objects_buffer[1] = sgCreateLine(0.0, 0.0, 0.0, 0.0, -m_room_Y_size, 0.0);

sgCContour*  cnt1 = sgCContour::CreateContour(objects_buffer,2);
sgCContour*  cnt2 = (sgCContour*)cnt1->Clone();
sgCContour*  cnt3 = (sgCContour*)cnt1->Clone();
```

Further let's create the walls with the help of extrusion. For this, let's create the walls contour as a start:

```
   objects_buffer[0] = cnt1;
   objects_buffer[1] = sgCreateLine(0.0, -m_room_Y_size, 0.0,
                                                          -m_wall_thickness, -
m_room_Y_size, 0.0);
   objects_buffer[2] = sgCreateLine(-m_wall_thickness, -m_room_Y_size, 0.0,
                                                          -m_wall_thickness,
m_wall_thickness, 0.0);
   objects_buffer[3] = sgCreateLine(-m_wall_thickness, m_wall_thickness, 0.0,
                                                          m_room_X_size,
m_wall_thickness, 0.0);

   objects_buffer[4] = sgCreateLine(m_room_X_size, m_wall_thickness, 0.0,
                                                          m_room_X_size, 0.0,
0.0);

   objects_buffer[5] = sgCContour::CreateContour(objects_buffer,5);
```

Now let's extrude this contour to the m_room_Z_size height and then remove this contour as an unnecessary one:

```
   SG_VECTOR  extVec = {0.0, 0.0, m_room_Z_size};

   m_walls = sgKinematic::Extrude((const sgC2DObject&)(*objects_buffer[5]),NULL,0,
extVec,true);

   sgGetScene()->AttachObject(m_walls);
   m_walls->SetAttribute(SG_OA_COLOR,20);
   m_walls->SetAttribute(SG_OA_LINE_THICKNESS, 1);

   sgDeleteObject(objects_buffer[5]);
```

Let's create the floor using the operation of constructing a flat face:

```
   objects_buffer[0] = cnt2;
   objects_buffer[1] = sgCreateLine(m_room_X_size, 0.0 , 0.0,
                                                    m_room_X_size, -m_room_Y_size,
0.0);
   objects_buffer[2] = sgCreateLine(m_room_X_size, -m_room_Y_size, 0.0,
                                                    0.0 , -m_room_Y_size, 0.0);

   objects_buffer[3] = sgCContour::CreateContour(objects_buffer,3);

   m_floor = sgSurfaces::Face((const sgC2DObject&)(*objects_buffer[3]),NULL,0);

   sgGetScene()->AttachObject(m_floor);
   m_floor->SetAttribute(SG_OA_COLOR,25);
   m_floor->SetAttribute(SG_OA_LINE_THICKNESS, 1);

   sgDeleteObject(objects_buffer[3]);
```

Then we'll create the plinth. First we'll construct the plinth clip - a flat contour consisting of two line segments and three arcs. The following code creates this contour:

```
   objects_buffer[0] = sgCreateLine(0.0, 0.0, 0.0, 0.0, -0.3, 0.0);

  SG_ARC arcGeo;
  SG_POINT p1,p2,p3;
  p1.x = 0.0;    p1.y = -0.3;   p1.z = 0.0;
  p2.x = -0.104; p2.y = -0.193; p2.z = 0.0;
  p3.x = -0.099; p3.y = -0.232; p3.z = 0.0;
  arcGeo.FromTreePoints(p1, p2, p3,false);
  objects_buffer[1] = sgCreateArc(arcGeo);

  p1.x = -0.104; p1.y = -0.193; p1.z = 0.0;
  p2.x = -0.205; p2.y = -0.095; p2.z = 0.0;
  p3.x = -0.132; p3.y = -0.128; p3.z = 0.0;
  arcGeo.FromTreePoints(p1, p2, p3,false);
  objects_buffer[2] = sgCreateArc(arcGeo);

  p1.x = -0.205; p1.y = -0.095; p1.z = 0.0;
  p2.x = -0.300; p2.y = 0.000;  p2.z = 0.0;
  p3.x = -0.294; p3.y = -0.047; p3.z = 0.0;
  arcGeo.FromTreePoints(p1, p2, p3,false);
  objects_buffer[3] = sgCreateArc(arcGeo);

  objects_buffer[4] = sgCreateLine(-0.3, 0.0, 0.0, 0.0, 0.0, 0.0);

  objects_buffer[5] = sgCContour::CreateContour(objects_buffer,5);
```

Now we create the plinth from this contour using the operation of constructing a pipe-like solid of an arbitrary clip:

```
  SG_POINT  pntOnPl = {0.0, 0.0, 0.0};
   bool      clPl = true;

  m_down_plinth = sgKinematic::Pipe((const sgC2DObject&)(*objects_buffer[5]),
NULL,0,
                              (const sgC2DObject&)(*cnt3),pntOnPl, 90.0, clPl);

  SG_VECTOR transV = {0,0,0.01};
  m_down_plinth->InitTempMatrix()->Translate(transV);
  m_down_plinth->ApplyTempMatrix();
  m_down_plinth->DestroyTempMatrix();

  sgGetScene()->AttachObject(m_down_plinth);
  m_down_plinth->SetAttribute(SG_OA_COLOR,35);
  m_down_plinth->SetAttribute(SG_OA_LINE_THICKNESS, 1);

  sgDeleteObject(objects_buffer[5]);
```

```
   sgDeleteObject(cnt3);
```

Then let's create in sequence the stem of the standard lamp and its upper part. The stem will be created using the operation of rotating a spline around the axis:

```
  SG_SPLINE*  spl = SG_SPLINE::Create();
   p1.x = 0.1; p1.y = 0.0; p1.z = 4.0;
   spl->AddKnot(p1,0);
   p1.x = 0.2; p1.y = 0.0; p1.z = 2.0;
   spl->AddKnot(p1,1);
   p1.x = 0.03; p1.y = 0.0; p1.z = 1.8;
   spl->AddKnot(p1,2);
   p1.x = 0.1; p1.y = 0.0; p1.z = 1.0;
   spl->AddKnot(p1,3);
   p1.x = 0.05; p1.y = 0.0; p1.z = 0.1;
   spl->AddKnot(p1,4);
   p1.x = 0.6; p1.y = 0.0; p1.z = 0.01;
   spl->AddKnot(p1,5);

   objects_buffer[0] = sgCreateSpline(*spl);
   SG_SPLINE::Delete(spl);
   p1.x = p1.y = p1.z = 0.0;
   p2.x = p2.y = 0.0; p2.z = 10.0;
   m_stem = sgKinematic::Rotation(*(sgC2DObject*)objects_buffer[0],p1,p2,360,
false);

   SG_VECTOR transV1 = {1.0, -1.0 ,0.0};
   m_stem->InitTempMatrix()->Translate(transV1);
   m_stem->ApplyTempMatrix();
   m_stem->DestroyTempMatrix();

   sgDeleteObject(objects_buffer[0]);


   sgGetScene()->AttachObject(m_stem);
   m_stem->SetAttribute(SG_OA_COLOR,4);
   m_stem->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

And the upper part will consist of three parts - the solid of a line segment revolution and two toruses:

```
    objects_buffer[0] = sgCreateLine(1.0, 0.0, 3.7, 0.4, 0.0, 5.0);;
   m_lamp_head = sgKinematic::Rotation(*(sgC2DObject*)objects_buffer[0],p1,p2,360,
false);
   m_lamp_head->InitTempMatrix()->Translate(transV1);
   m_lamp_head->ApplyTempMatrix();
   m_lamp_head->DestroyTempMatrix();
   sgGetScene()->AttachObject(m_lamp_head);
   m_lamp_head->SetAttribute(SG_OA_COLOR,5);
   m_lamp_head->SetAttribute(SG_OA_LINE_THICKNESS, 1);
   sgDeleteObject(objects_buffer[0]);
```

```
  objects_buffer[0] = sgCreateTorus(1.0, 0.05, 24,24);
  transV1.z = 3.7;
  objects_buffer[0]->InitTempMatrix()->Translate(transV1);
  objects_buffer[0]->ApplyTempMatrix();
  objects_buffer[0]->DestroyTempMatrix();
  sgGetScene()->AttachObject(objects_buffer[0]);
  objects_buffer[0]->SetAttribute(SG_OA_COLOR,15);
  objects_buffer[0]->SetAttribute(SG_OA_LINE_THICKNESS, 1);

  objects_buffer[0] = sgCreateTorus(0.4, 0.05, 24,24);
  transV1.z = 5.0;
  objects_buffer[0]->InitTempMatrix()->Translate(transV1);
  objects_buffer[0]->ApplyTempMatrix();
  objects_buffer[0]->DestroyTempMatrix();
  sgGetScene()->AttachObject(objects_buffer[0]);
  objects_buffer[0]->SetAttribute(SG_OA_COLOR,15);
  objects_buffer[0]->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

After that let's create the legs of the table using extrusion and move them to the required distance:

```
 SG_CIRCLE tableLeg;
  tableLeg.center.x = 0.0;tableLeg.center.y = 0.0;tableLeg.center.z = 0.0;
  tableLeg.normal.x = 0.0;tableLeg.normal.y = 0.0;tableLeg.normal.z = 1.0;
  tableLeg.radius = 0.04;
  objects_buffer[0] = sgCreateCircle(tableLeg);

  extVec.x = 0.2;extVec.y = 0.2;extVec.z = 2.5;
  m_table_legs[0] = sgKinematic::Extrude((const sgC2DObject&)(*objects_buffer
[0]),NULL,0,extVec,true);
  extVec.x = -0.2;extVec.y = 0.2;extVec.z = 2.5;
  m_table_legs[1] = sgKinematic::Extrude((const sgC2DObject&)(*objects_buffer
[0]),NULL,0,extVec,true);
  extVec.x = -0.2;extVec.y = -0.2;extVec.z = 2.5;
  m_table_legs[2] = sgKinematic::Extrude((const sgC2DObject&)(*objects_buffer
[0]),NULL,0,extVec,true);
  extVec.x = 0.2;extVec.y = -0.2;extVec.z = 2.5;
  m_table_legs[3] = sgKinematic::Extrude((const sgC2DObject&)(*objects_buffer
[0]),NULL,0,extVec,true);

  sgDeleteObject(objects_buffer[0]);

  for (i=0;i<4;i++)
  {
        transV1.z = 0.0;
        switch(i)
        {
        case 0:
              transV1.x = -2.0;
              transV1.y = -1.0;
              break;
        case 1:
```

```
                transV1.x = 2.0;
                transV1.y = -1.0;
                break;
        case 2:
                transV1.x = 2.0;
                transV1.y = 1.0;
                break;
        case 3:
                transV1.x = -2.0;
                transV1.y = 1.0;
                break;
        }
        m_table_legs[i]->InitTempMatrix()->Translate(transV1);
        m_table_legs[i]->ApplyTempMatrix();
        m_table_legs[i]->DestroyTempMatrix();
        objects_buffer[i] = m_table_legs[i];
}
```

To create the upper part of the table we'll construct an ordinary box:

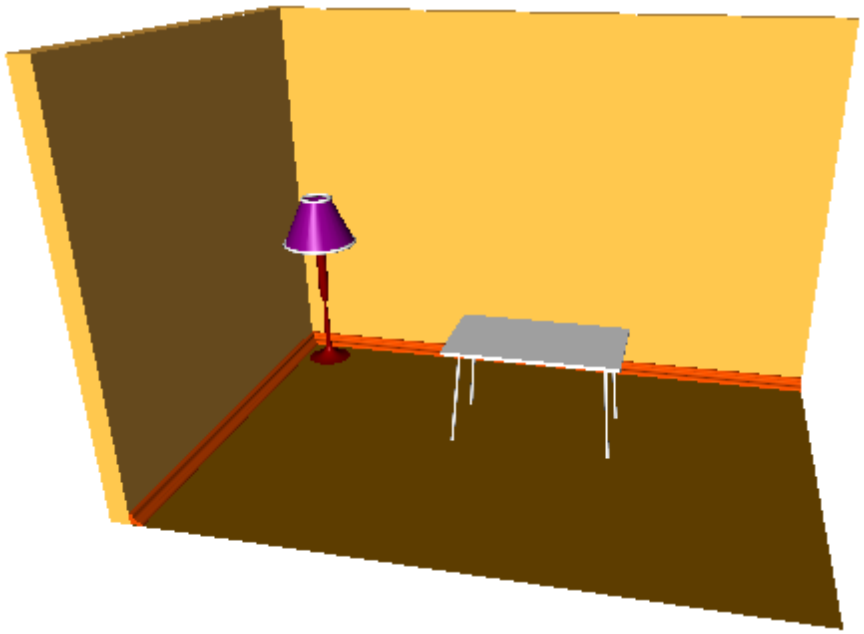
```
objects_buffer[4] = sgCreateBox(4.4,2.4,0.1);

transV1.x = -2.2;
transV1.y = -1.2;
transV1.z = 2.5;
objects_buffer[4]->InitTempMatrix()->Translate(transV1);
objects_buffer[4]->ApplyTempMatrix();
objects_buffer[4]->DestroyTempMatrix();
```

Finally, let's group the parts of the table and locate it in the room:

```
m_table = sgCGroup::CreateGroup(objects_buffer,5);

transV1.x = 8.0;
transV1.y = -4.0;
transV1.z = 0.01;
m_table->InitTempMatrix()->Translate(transV1);
m_table->ApplyTempMatrix();
m_table->DestroyTempMatrix();

sgGetScene()->AttachObject(m_table);
m_table->SetAttribute(SG_OA_COLOR,15);
m_table->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

See also:
sgKinematic::Pipe
sgKinematic::Rotation
sgKinematic::Extrude
sgSurfaces::Face
sgCArc SG_ARC
sgCLine SG_LINE
sgCContour sgCContour::CreateContour
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Result:



16.5.2 Drills

Drills.

Let's create a scene of five drills and a plate with five holes below the drills.

To start with, let's write the function creating the clip of the drill of the specified radius. The drill clip will be a contour of 8 line segments and 4 arcs. The function creating the clip will be the following:

```
sgCContour* CreateDrillSection(double rad)
{
   sgCObject*  objcts[6];

   objcts[0] = sgCreateLine(0.0, -0.99*rad, 0.0, -0.05*rad, -0.99*rad, 0.0);
   objcts[1] = sgCreateLine(-0.05*rad, -0.99*rad, 0.0, -0.05*rad, -0.97*rad, 0.0);

   SG_ARC arcGeo;
   SG_POINT arcP1;
   SG_POINT arcP2;
   SG_POINT arcP3;

   arcP1.x = -0.05*rad;  arcP1.y = -0.97*rad; arcP1.z = 0.0;
   arcP2.x = -1.0*rad;   arcP2.y = 0.0;       arcP2.z = 0.0;
   arcP3.x = -0.87*rad;  arcP3.y = -0.5*rad;  arcP3.z = 0.0;
```

```
   arcGeo.FromTreePoints(arcP1,arcP2,arcP3,false);

   objcts[2] = sgCreateArc(arcGeo);

   arcP1.x = -1.0*rad;   arcP1.y = 0.0;       arcP1.z = 0.0;
   arcP2.x = -0.05*rad;  arcP2.y = 0.0;       arcP2.z = 0.0;
   arcP3.x = -0.35*rad;  arcP3.y = -0.33*rad; arcP3.z = 0.0;

   arcGeo.FromTreePoints(arcP1,arcP2,arcP3,false);

   objcts[3] = sgCreateArc(arcGeo);

   objcts[4] = sgCreateLine(-0.05*rad, 0.0, 0.0, -0.05*rad, 0.99*rad, 0.0);
   objcts[5] = sgCreateLine(-0.05*rad, 0.99*rad, 0.0, 0.0, 0.99*rad, 0.0);

   sgCContour*  cnt1 = sgCContour::CreateContour(objcts,6);
   sgCContour*  cnt2 = (sgCContour*)cnt1->Clone();

   SG_POINT     rotAxeP = {0.0, 0.0, 0.0};
   SG_VECTOR    rotAxeDir = {0.0, 0.0, 1.0};
   cnt2->InitTempMatrix()->Rotate(rotAxeP,rotAxeDir,3.14159265);
   cnt2->ApplyTempMatrix();
   cnt2->DestroyTempMatrix();

   objcts[0] = cnt1;
   objcts[1] = cnt2;

   return sgCContour::CreateContour(objcts,2);
}
```

Then let's create five drills - four drills with radius 1 and one drill - with 2.5 - using the operation of constructing a solid from clips. The following function creates a drill:

```
sgCObject* CreateDrill(double rad, double dH)
{
   sgC2DObject* drill_1_sections[10];
   double       params[10];
   memset(params,0,sizeof(double)*10);
   drill_1_sections[0] = CreateDrillSection(rad);

   SG_POINT     rotAxeP = {0.0, 0.0, 0.0};
   SG_VECTOR    rotAxeDir = {0.0, 0.0, 1.0};
   SG_VECTOR    transVec = {0.0, 0.0, 1.0};
   for (int i=1;i<7;i++)
   {
          drill_1_sections[i] = (sgCContour*)(drill_1_sections[0]->Clone());
          drill_1_sections[i]->InitTempMatrix()->Rotate(rotAxeP,rotAxeDir,
i*5.0*3.14159265/6.0);
          transVec.z = dH*i;
          drill_1_sections[i]->GetTempMatrix()->Translate(transVec);
          drill_1_sections[i]->ApplyTempMatrix();
```

```
        drill_1_sections[i]->DestroyTempMatrix();
   }

   SG_CIRCLE cirGeo;
   cirGeo.center.x = cirGeo.center.y = 0.0;  cirGeo.center.z =  7*dH+5;
   cirGeo.normal.x = cirGeo.normal.y = 0.0;  cirGeo.normal.z = 1.0;
   cirGeo.radius = rad-0.1;

   drill_1_sections[7] = sgCreateCircle(cirGeo);
   drill_1_sections[7]->ChangeOrient();
   cirGeo.center.z = 7*dH+10;
   drill_1_sections[8] = sgCreateCircle(cirGeo);
   drill_1_sections[8]->ChangeOrient();
   cirGeo.center.z =7*dH+15;
   drill_1_sections[9] = sgCreateCircle(cirGeo);
   drill_1_sections[9]->ChangeOrient();


   sgCObject* resul = sgSurfaces::SplineSurfaceFromSections((const sgC2DObject**)
drill_1_sections,
        params,10,true);

   for (int i=0;i<10;i++)
        sgDeleteObject(drill_1_sections[i]);

   return resul;
}
```

And then we'll use this function to create the drills:

```
  m_drills[0] = CreateDrill(1.0,5.0);
   m_drills[1] = m_drills[0]->Clone();
   m_drills[2] = m_drills[0]->Clone();
   m_drills[3] = m_drills[0]->Clone();
   m_drills[4] = CreateDrill(2.5,8.0);
```

Let's create a plate in the form of a box in which later on we'll construct the holes below the drills:

```
  sgCObject*  tmpOb1;
   tmpOb1 = sgCreateBox(40.0,40.0, 1.0);
   transVec.x = -20.0;
   transVec.y = -20.0;
   transVec.z = -4.0;
   tmpOb1->InitTempMatrix()->Translate(transVec);
   tmpOb1->ApplyTempMatrix();
   tmpOb1->DestroyTempMatrix();
```

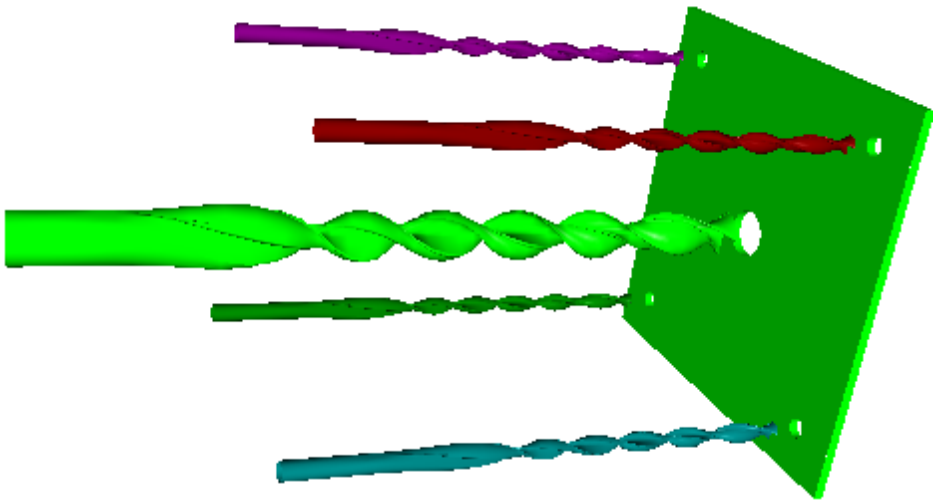
To create the holes let's use the boolean subtraction of cylinders with radii equal to the drill radii from the plate:

```
   sgCCylinder*  holes[5];
   for (int i=0;i<4;i++)
        holes[i] = sgCCylinder::Create(1.0, 20.0, 24);
   holes[4] = sgCCylinder::Create(2.5, 20.0, 24);
```

```
        for (int i=0;i<5;i++)
        {
                tmpGroup = sgBoolean::Sub((const sgC3DObject&)(*tmpOb1),
                                                                (const sgC3DObject&)
(*holes[i]));
                ASSERT(tmpGroup->GetChildrenList()->GetCount()==1);
                sgDeleteObject(tmpOb1);
                tmpGroup->BreakGroup(&tmpOb1);
                sgDeleteObject(tmpGroup);
                sgDeleteObject(holes[i]);
        }
```

See also:
sgBoolean::Sub
sgSurfaces::SplineSurfaceFromSections
sgCArc SG_ARC
sgCLine SG_LINE
sgCContour sgCContour::CreateContour
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Result:



16.5.3 Clock

Clock.

Let's create an object - a clock.

First, let's construct a face - a flat face stretched on a circle with the radius equal to 10:

```
SG_CIRCLE cirG;
cirG.center.x = cirG.center.y =  cirG.center.z = 0.0;
cirG.normal.x = cirG.normal.y =  0.0; cirG.normal.z = 1.0;
cirG.radius = 10.0;

sgCCircle* cir = sgCreateCircle(cirG);
m_face = sgSurfaces::Face(*cir,NULL,0);
sgDeleteObject(cir);
sgGetScene()->AttachObject(m_face);
m_face->SetAttribute(SG_OA_COLOR,35);
m_face->SetAttribute(SG_OA_LINE_THICKNESS, 1);
SG_VECTOR trVec = {0.0, 0.0, -0.03};
m_face->InitTempMatrix()->Translate(trVec);
m_face->ApplyTempMatrix();
m_face->DestroyTempMatrix();
```

Then comes the torus edging with the face:

```
m_face_tor = sgCTorus::Create(10.0, 0.2,36,24);
sgGetScene()->AttachObject(m_face_tor);
m_face_tor->SetAttribute(SG_OA_COLOR,30);
m_face_tor->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

Further let's create a hairline which will be located in the place of the hour figures. It will be an object of extrusion with closing to a solid. The twelve o'clock hairline will be colored differently from the rest. The following function creates the extruded contour:

```
sgCContour*  CreateHairLine()
{
       sgCObject*  objcts[4];

       SG_ARC arcGeo;
       SG_POINT arcP1;
       SG_POINT arcP2;
       SG_POINT arcP3;

       arcP1.x = -0.15;  arcP1.y = 9.0;   arcP1.z = 0.0;
       arcP2.x = 0.15;   arcP2.y = 9.0;   arcP2.z = 0.0;
       arcP3.x = 0.0;    arcP3.y = 9.2;   arcP3.z = 0.0;

       arcGeo.FromTreePoints(arcP1,arcP2,arcP3,false);

       objcts[0] = sgCreateArc(arcGeo);

       objcts[1] = sgCreateLine(0.15, 9.0, 0.0, 0.15, 8.0, 0.0);

       arcP1.x = 0.15;  arcP1.y = 8.0;   arcP1.z = 0.0;
       arcP2.x = -0.15;   arcP2.y = 8.0;   arcP2.z = 0.0;
       arcP3.x = 0.0;    arcP3.y = 7.8;   arcP3.z = 0.0;
```

```
        arcGeo.FromTreePoints(arcP1,arcP2,arcP3,false);

        objcts[2] = sgCreateArc(arcGeo);

        objcts[3] = sgCreateLine(-0.15, 8.0, 0.0, -0.15, 9.0, 0.0);

        return sgCContour::CreateContour(objcts,4);
  }
```

And now let's create a solid of extrusion from this contour:

```
  sgCContour* tmpCnt = CreateHairLine();
  SG_VECTOR extrVec = {0.0, 0.0, 0.2};
  m_hairLines3D[0] = sgKinematic::Extrude((const sgC2DObject&)(*tmpCnt),NULL,0,
extrVec,true);
  sgDeleteObject(tmpCnt);
  sgGetScene()->AttachObject(m_hairLines3D[0]);
  m_hairLines3D[0]->SetAttribute(SG_OA_COLOR,12);
  m_hairLines3D[0]->SetAttribute(SG_OA_LINE_THICKNESS, 1);
```

Let's multiply this solid of extrusion on all hour figures:

```
   SG_POINT     rotAxeP = {0.0, 0.0, 0.0};
  SG_VECTOR    rotAxeDir = {0.0, 0.0, 1.0};
  for (int i=1;i<12;i++)
  {
        m_hairLines3D[i] = (sgCContour*)(m_hairLines3D[0]->Clone());
        m_hairLines3D[i]->InitTempMatrix()->Rotate(rotAxeP,rotAxeDir,
i*3.14159265/6.0);
        m_hairLines3D[i]->ApplyTempMatrix();
        m_hairLines3D[i]->DestroyTempMatrix();
        sgGetScene()->AttachObject(m_hairLines3D[i]);
        m_hairLines3D[i]->SetAttribute(SG_OA_COLOR,90);
  }
```

Then let's create minute hairlines - just line segments:

```
   m_hairLines2D[0] = sgCreateLine(0.0, 8.0, 0.0, 0.0, 9.0, 0.0);
  sgGetScene()->AttachObject(m_hairLines2D[0]);

  for (int i=1;i<60;i++)
  {
        m_hairLines2D[i] = (sgCContour*)(m_hairLines2D[0]->Clone());
        m_hairLines2D[i]->InitTempMatrix()->Rotate(rotAxeP,rotAxeDir,
i*3.14159265/30.0);
        m_hairLines2D[i]->ApplyTempMatrix();
        m_hairLines2D[i]->DestroyTempMatrix();
        sgGetScene()->AttachObject(m_hairLines2D[i]);
        m_hairLines2D[i]->SetAttribute(SG_OA_COLOR,0);
  }
```

Let's put a sphere into the center of the clock where we will fix the hands:

```
m_clock_center = sgCreateSphere(0.2,36,36);
 sgGetScene()->AttachObject(m_clock_center);
 m_clock_center->SetAttribute(SG_OA_COLOR,3);
```

And finally we'll create three hands with various rotation angles:
The hour hand is a flat face with a hole:

```
sgCObject* CClockScene::CreateHourHand()
{
	sgCObject*  ob_buff[4];

	ob_buff[0] = sgCreateLine(0.0, -0.1, 0.0, 0.5, 0.5, 0.0);
	ob_buff[1] = sgCreateLine(0.5, 0.5, 0.0, 0.0, 6.0, 0.0);
	ob_buff[2] = sgCreateLine(0.0, 6.0, 0.0, -0.5, 0.5, 0.0);
	ob_buff[3] = sgCreateLine(-0.5, 0.5, 0.0, 0.0, -0.1, 0.0);

	sgC2DObject* cont1 = sgCContour::CreateContour(ob_buff,4);

	ob_buff[0] = sgCreateLine(0.0, 0.6, 0.0, 0.2, 0.8, 0.0);
	ob_buff[1] = sgCreateLine(0.2, 0.8, 0.0, 0.0, 5.0, 0.0);
	ob_buff[2] = sgCreateLine(0.0, 5.0, 0.0, -0.2, 0.8, 0.0);
	ob_buff[3] = sgCreateLine(-0.2, 0.8, 0.0, 0.0, 0.6, 0.0);
	sgC2DObject* cont2 = sgCContour::CreateContour(ob_buff,4);


	sgCObject* res = sgSurfaces::Face(*cont1,(const sgC2DObject**)
(&cont2),1);
	sgDeleteObject(cont1);
	sgDeleteObject(cont2);
	return res;
}
```

The minute hand is a flat face with a hole:

```
sgCObject* CClockScene::CreateMinuteHand()
{
	sgCObject*  ob_buff[4];

	ob_buff[0] = sgCreateLine(0.0, -0.1, 0.0, 0.5, 0.5, 0.0);
	ob_buff[1] = sgCreateLine(0.5, 0.5, 0.0, 0.0, 9.0, 0.0);
	ob_buff[2] = sgCreateLine(0.0, 9.0, 0.0, -0.5, 0.5, 0.0);
	ob_buff[3] = sgCreateLine(-0.5, 0.5, 0.0, 0.0, -0.1, 0.0);

	sgC2DObject* cont1 = sgCContour::CreateContour(ob_buff,4);

	ob_buff[0] = sgCreateLine(0.0, 1.6, 0.0, 0.2, 1.8, 0.0);
	ob_buff[1] = sgCreateLine(0.2, 1.8, 0.0, 0.0, 8.0, 0.0);
	ob_buff[2] = sgCreateLine(0.0, 8.0, 0.0, -0.2, 1.8, 0.0);
	ob_buff[3] = sgCreateLine(-0.2, 1.8, 0.0, 0.0, 1.6, 0.0);
	sgC2DObject* cont2 = sgCContour::CreateContour(ob_buff,4);
```

```
        sgCObject* res = sgSurfaces::Face(*cont1,(const sgC2DObject**)
(&cont2),1);
        sgDeleteObject(cont1);
        sgDeleteObject(cont2);
        return res;
}
```

The second hand is a line segment:

```
sgCObject* CClockScene::CreateSecondHand()
{
        return sgCreateLine(0.0, 0.0, 0.0, 0.0, 9.0, 0.0);
}
```

See also:
sgKinematic::Extrude
sgSurfaces::Face
sgCArc SG_ARC
sgCLine SG_LINE
sgCContour sgCContour::CreateContour
sgGetScene sgCScene::AttachObject sgCObject::SetAttribute

Result: