Open CASCADE Technology 7.5.0

Visualization

November 3, 2020

Contents

1 Introduction

Visualization in Open CASCADE Technology is based on the separation of:

- on the one hand – the data which stores the geometry and topology of the entities you want to display and select, and
- on the other hand – its **presentation** (what you see when an object is displayed in a scene) and **selection** (possibility to choose the whole object or its sub-parts interactively to apply application-defined operations to the selected entities).

Presentations are managed through the **Presentation** component, and selection through the **Selection** component.

Application Interactive Services (AIS) provides the means to create links between an application GUI viewer and the packages, which are used to manage selection and presentation, which makes management of these functionalities in 3D more intuitive and consequently, more transparent.

AIS uses the notion of the *Interactive Object*, a displayable and selectable entity, which represents an element from the application data. As a result, in 3D, you, the user, have no need to be familiar with any functions underlying AIS unless you want to create your own interactive objects or selection filters.

If, however, you require types of interactive objects and filters other than those provided, you will need to know the mechanics of presentable and selectable objects, specifically how to implement their virtual functions. To do this requires familiarity with such fundamental concepts as the Sensitive Primitive and the Presentable Object.

The the following packages are used to display 3D objects:

- *AIS*;
- *StdPrs*;
- *Prs3d*;
- *PrsMgr*;
- *V3d*;
- *Graphic3d*.

The packages used to display 3D objects are also applicable for visualization of 2D objects.

The figure below presents a schematic overview of the relations between the key concepts and packages in visualization. Naturally, "Geometry & Topology" is just an example of application data that can be handled by *AIS*, and application-specific interactive objects can deal with any kind of data.

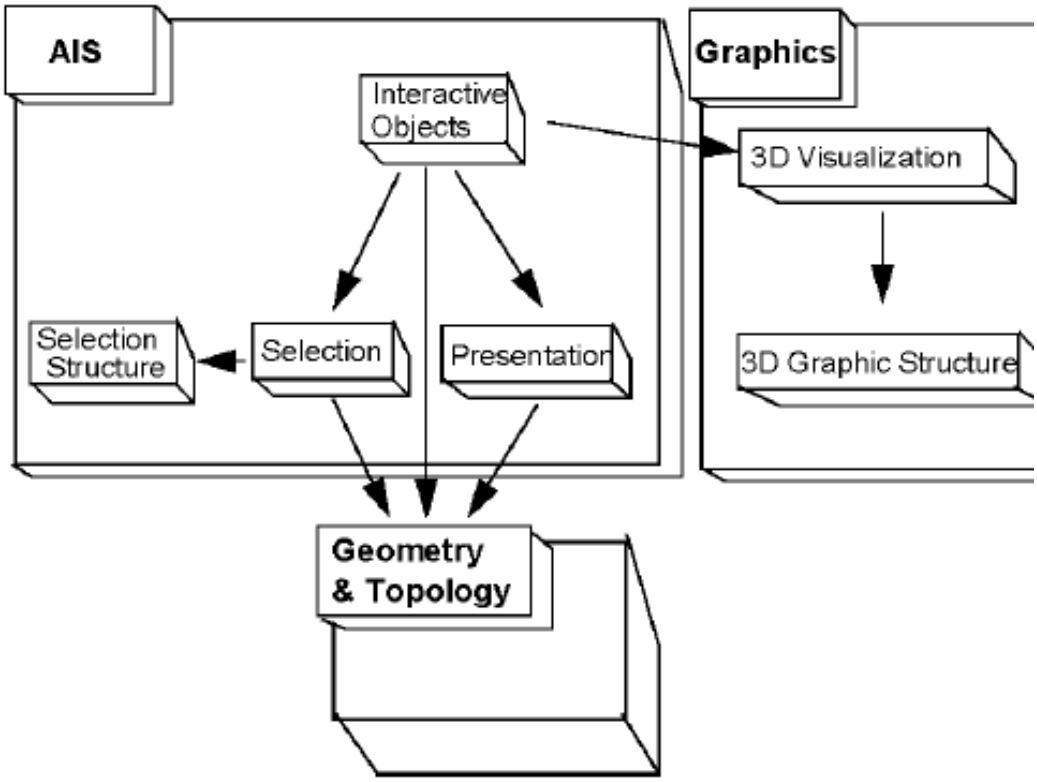


Figure 1: Key concepts and packages in visualization

To answer different needs of CASCADE users, this User's Guide offers the following three paths in reading it.

- If the 3D services proposed in AIS meet your requirements, you need only read chapter 3 AIS: Application Interactive Services.
- If you need more detail, for example, a selection filter on another type of entity – you should read chapter 2 Fundamental Concepts, chapter 3 AIS: Application Interactive Services, and 4 3D Presentations. You may want to begin with the chapter presenting AIS.

2 Fundamental Concepts

2.1 Presentation

In Open CASCADE Technology, presentation services are separated from the data, which they represent, which is generated by applicative algorithms. This division allows you to modify a geometric or topological algorithm and its resulting objects without modifying the visualization services.

2.1.1 Structure of the Presentation

Displaying an object on the screen involves three kinds of entities:

- a presentable object, the *AIS_InteractiveObject*
- a viewer
- an interactive context, the *AIS_InteractiveContext*.

The presentable object

The purpose of a presentable object is to provide the graphical representation of an object in the form of *Graphic3d_Structure*. On the first display request, it creates this structure by calling the appropriate algorithm and retaining this framework for further display.

Standard presentation algorithms are provided in the *StdPrs* and *Prs3d* packages. You can, however, write specific presentation algorithms of your own, provided that they create presentations made of structures from the *Graphic3d* packages. You can also create several presentations of a single presentable object: one for each visualization mode supported by your application.

Each object to be presented individually must be presentable or associated with a presentable object.

The viewer

The viewer allows interactively manipulating views of the object. When you zoom, translate or rotate a view, the viewer operates on the graphic structure created by the presentable object and not on the data model of the application. Creating Graphic3d structures in your presentation algorithms allows you to use the 3D viewers provided in Open CASCADE Technology for 3D visualization.

The Interactive Context

The interactive context controls the entire presentation process from a common high-level API. When the application requests the display of an object, the interactive context requests the graphic structure from the presentable object and sends it to the viewer for displaying.

2.1.2 Presentation packages

Presentation involves at least the *AIS, PrsMgr, StdPrs* and *V3d* packages. Additional packages, such as *Prs3d* and *Graphic3d* may be used if you need to implement your own presentation algorithms.

- Standard Interactive Objects
  - *AIS* package provides classes to implement interactive objects (presentable and selectable entities).
  - *PrsDim* package provides presentable objects for drawing dimensions and relations.
  - *MeshVS* package provides presentable object MeshVS_Mesh for working with mesh data.
- Standard presentation builders
  - *Prs3d* package provides ready-to-use standard presentation algorithms for simple geometries like arrow, cylinder, sphere. It also defines *Prs3d_Drawer* class controlling the attributes of the presentation to be created in terms of color, line type, thickness, etc.

  - *StdPrs* package provides ready-to-use standard presentation algorithms for B-Rep shapes. It provides generic presentation algorithms such as shading, wireframe, isolines and hidden line removal.
  - *DsgPrs* package provides tools for display of dimensions, relations and XYZ trihedrons.

- Selection services
  - *Select3D*, *SelectBasics* and *SelectMgr* implement selection (picking) services.
  - *StdSelect* package provide selection builders for B-Rep shapes.

- Viewer management *V3d* package provides the services supported by the 3D viewer.

- Low-level interfaces
  - *PrsMgr* package defines basic interfaces and tools for presentable object. It contains all classes needed to implement the presentation process: abstract classes *PrsMgr_Presentation* and *PrsMgr_PresentableObject* and concrete class *PrsMgr_PresentationManager*.
  - *Graphic3d* package provides low-level graphic structures. It also defines an interface of Graphic3d_GraphicDriver providing a connection with low-level graphics APIs like OpenGL.

2.1.3 A Basic Example: How to display a 3D object

```
Handle(V3d_Viewer) theViewer;
Handle(AIS_InteractiveContext) aContext = new AIS_InteractiveContext (theViewer);

BRepPrimAPI_MakeWedge aWedgeMaker (theWedgeDX, theWedgeDY, theWedgeDZ, theWedgeLtx);
TopoDS_Solid aShape = aWedgeMaker.Solid();
Handle(AIS_Shape) aShapePrs = new AIS_Shape (aShape); // creation of the presentable object
aContext->Display (aShapePrs, AIS_Shaded, 0, true);   // display the presentable object and redraw 3d
       viewer
```

The shape is created using the *BRepPrimAPI_MakeWedge* command. An *AIS_Shape* is then created from the shape. When calling the *Display* command, the interactive context calls the Compute method of the presentable object to calculate the presentation data and transfer it to the viewer. See figure below.

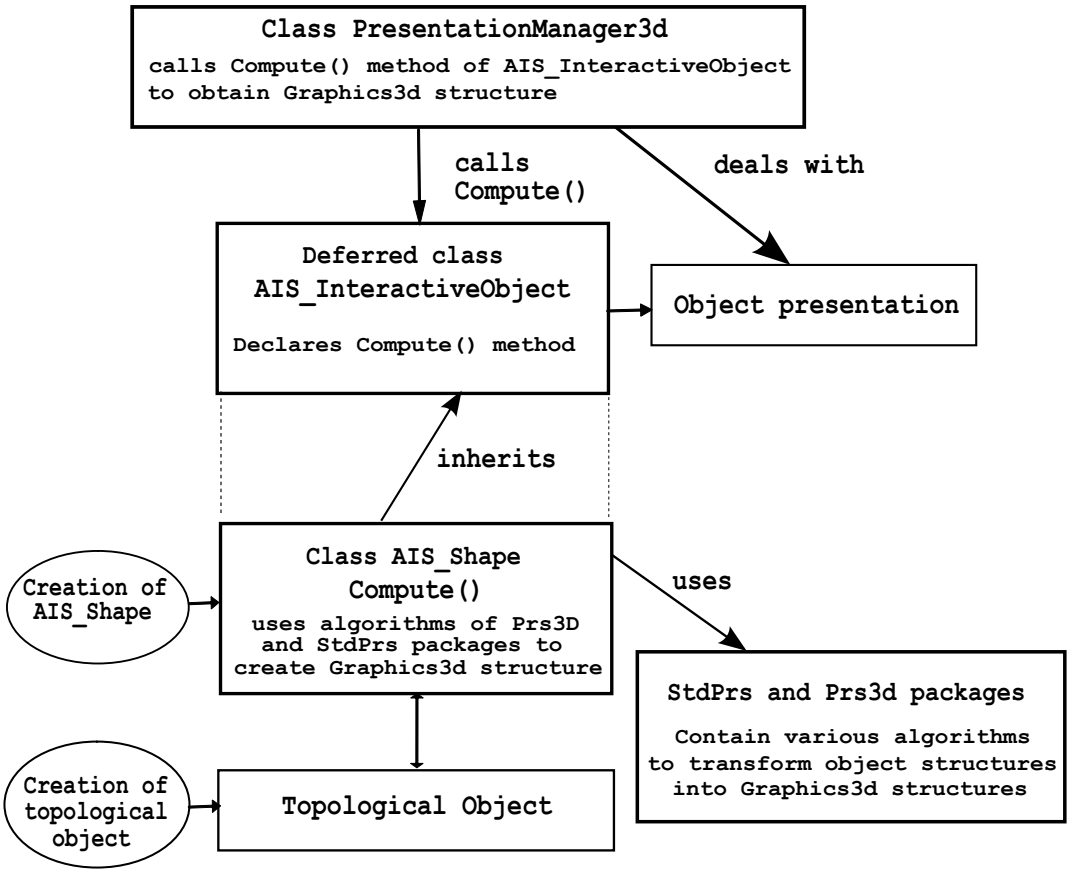


Figure 2: Processes involved in displaying a presentable shape

2.2 Selection

Standard OCCT selection algorithm is represented by 2 parts: dynamic and static. Dynamic selection causes objects to be automatically highlighted as the mouse cursor moves over them. Static selection allows to pick particular object (or objects) for further processing.

There are 3 different selection types:

- **Point selection** – allows picking and highlighting a single object (or its part) located under the mouse cursor;
- **Rectangle selection** – allows picking objects or parts located under the rectangle defined by the start and end mouse cursor positions;
- **Polyline selection** – allows picking objects or parts located under a user-defined non-self-intersecting polyline.

For OCCT selection algorithm, all selectable objects are represented as a set of sensitive zones, called **sensitive entities**. When the mouse cursor moves in the view, the sensitive entities of each object are analyzed for collision.

2.2.1 Terms and notions

This section introduces basic terms and notions used throughout the algorithm description.

Sensitive entity

Sensitive entities in the same way as entity owners are links between objects and the selection mechanism.

The purpose of entities is to define what parts of the object will be selectable in particular. Thus, any object that is meant to be selectable must be split into sensitive entities (one or several). For instance, to apply face selection to an object it is necessary to explode it into faces and use them for creation of a sensitive entity set.

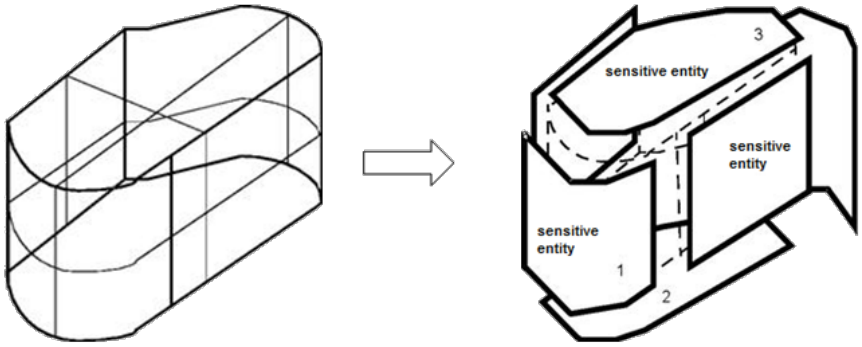


Figure 3: Example of a shape divided into sensitive entities

Depending on the user's needs, sensitive entities may be atomic (point or edge) or complex. Complex entities contain many sub-elements that can be handled by detection mechanism in a similar way (for example, a polyline stored as a set of line segments or a triangulation).

Entities are used as internal units of the selection algorithm and do not contain any topological data, hence they have a link to an upper-level interface that maintains topology-specific methods.

Entity owner

Each *Select3D_SensitiveEntity* stores a reference to its owner *SelectMgr_EntityOwner*, which is a class connecting the entity and the corresponding selectable object (*SelectMgr_SelectableObject*). Besides, owners can store any additional information, for example, the topological shape of the sensitive entity, highlight colors and methods, or if the entity is selected or not.

Selection

To simplify the handling of different selection modes of an object, sensitive entities linked to their owners are organized into sets, called **selections** (*SelectMgr_Selection*). Each selection contains entities created for a certain mode along with the sensitivity and update states.

Selectable object

Selectable object (*SelectMgr_SelectableObject* or more precisely *AIS_InteractiveObject*) stores information about all created selection modes and sensitive entities.

All successors of a selectable object must implement the method that splits its presentation into sensitive entities according to the given mode. The computed entities are arranged in one selection and added to the list of all selections of this object. No selection will be removed from the list until the object is deleted permanently.

For all standard OCCT interactive objects, zero mode is supposed to select the whole object (but it may be redefined in the custom object). For example, the *AIS_Shape* object determine the following modes (see AIS_Shape::↩ SelectionMode()):

- 0 – selection of the entire object (AIS_Shape);
- 1 – selection of the vertices (TopAbs_VERTEX);
- 2 – selection of the edges (TopAbs_EDGE);
- 3 – selection of the wires (TopAbs_WIRE);
- 4 – selection of the faces (TopAbs_FACE);

- 5 – selection of the shells (TopAbs_SHELL);
- 6 – selection of the constituent solids (TopAbs_SOLID).

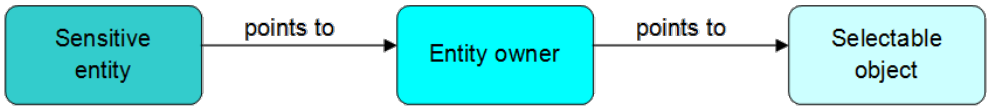


Figure 4: Hierarchy of references from sensitive entity to selectable object

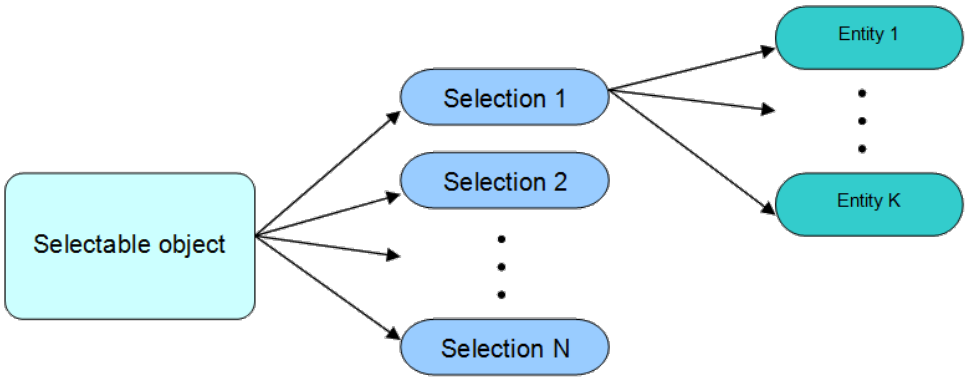


Figure 5: The principle of entities organization within the selectable object

Viewer selector

For each OCCT viewer there is a **Viewer selector** class *SelectMgr_ViewerSelector3d*. It provides a high-level API for the whole selection algorithm and encapsulates the processing of objects and sensitive entities for each mouse pick. The viewer selector maintains activation and deactivation of selection modes, launches the algorithm, which detects candidate entities to be picked, and stores its results, as well as implements an interface for keeping selection structures up-to-date.

Selection manager

Selection manager *SelectMgr_SelectionManager* is a high-level API to manipulate selection of all displayed objects. It handles all viewer selectors, activates and deactivates selection modes for the objects in all or particular selectors, manages computation and update of selections for each object. Moreover, it keeps selection structures updated taking into account applied changes.

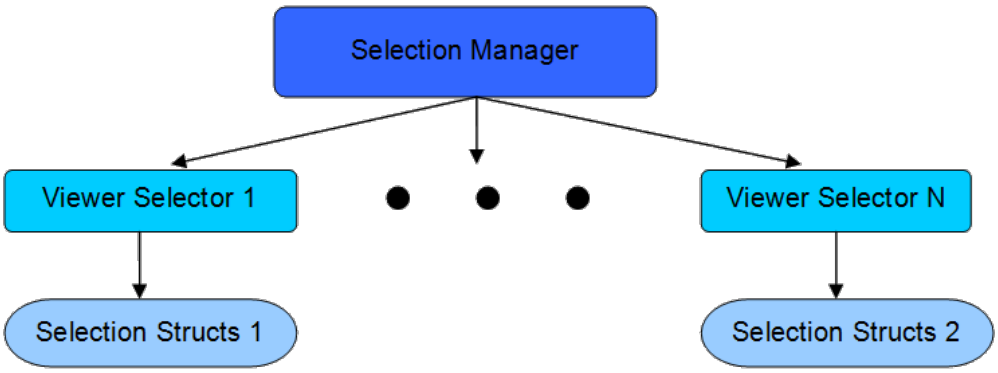


Figure 6: The relations chain between viewer selector and selection manager

2.2.2 Algorithm

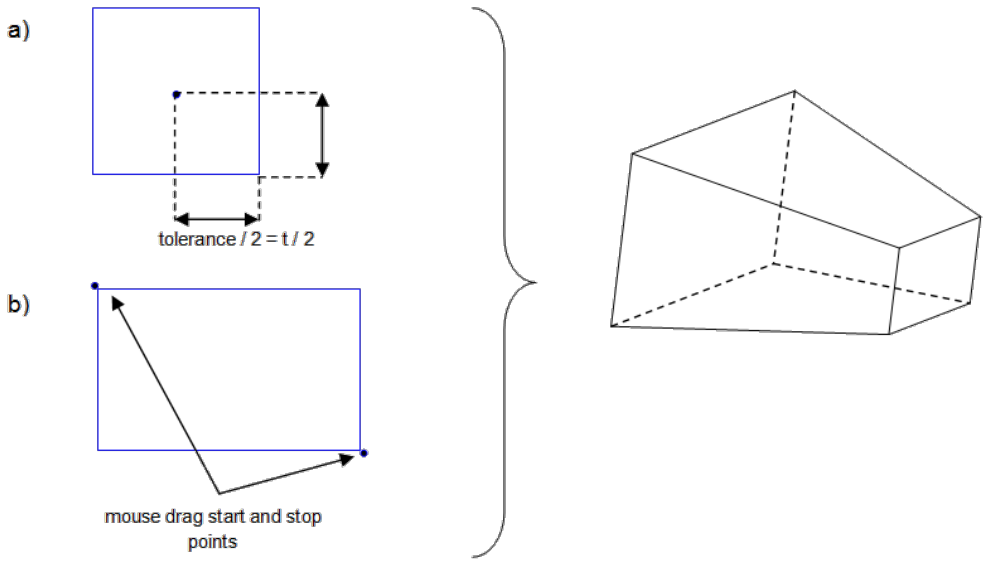
All three types of OCCT selection are implemented as a single concept, based on the search for overlap between frustum and sensitive entity through 3-level BVH tree traversal.

Selection Frustum

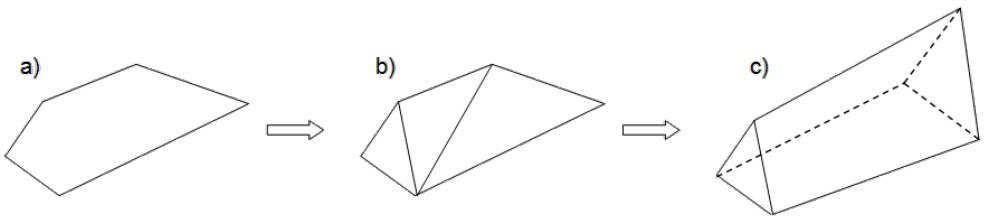
The first step of each run of selection algorithm is to build the selection frustum according to the currently activated selection type.

For the point or the rectangular selection the base of the frustum is a rectangle built in conformity with the pixel tolerance or the dimensions of a user-defined area, respectively. For the polyline selection, the polygon defined by the constructed line is triangulated and each triangle is used as the base for its own frustum. Thus, this type of selection uses a set of triangular frustums for overlap detection.

The frustum length is limited by near and far view volume planes and each plane is built parallel to the corresponding view volume plane.



The image above shows the rectangular frustum: a) after mouse move or click, b) after applying the rectangular selection.



In the image above triangular frustum is set: a) by a user-defined polyline, b) by triangulation of the polygon based on the given polyline, c) by a triangular frustum based on one of the triangles.

BVH trees

To maintain selection mechanism at the viewer level, a speedup structure composed of 3 BVH trees is used.

The first level tree is constructed of axis-aligned bounding boxes of each selectable object. Hence, the root of this tree contains the combination of all selectable boundaries even if they have no currently activated selections. Objects are added during the display of *AIS_InteractiveObject* and will be removed from this tree only when the object is destroyed. The 1st level BVH tree is build on demand simultaneously with the first run of the selection algorithm.

The second level BVH tree consists of all sensitive entities of one selectable object. The 2nd level trees are built automatically when the default mode is activated and rebuilt whenever a new selection mode is calculated for the first time.

The third level BVH tree is used for complex sensitive entities that contain many elements: for example, triangulations, wires with many segments, point sets, etc. It is built on demand for sensitive entities with more than 800K sub-elements (defined by *StdSelect_BRepSelectionTool::PreBuildBVH()*).

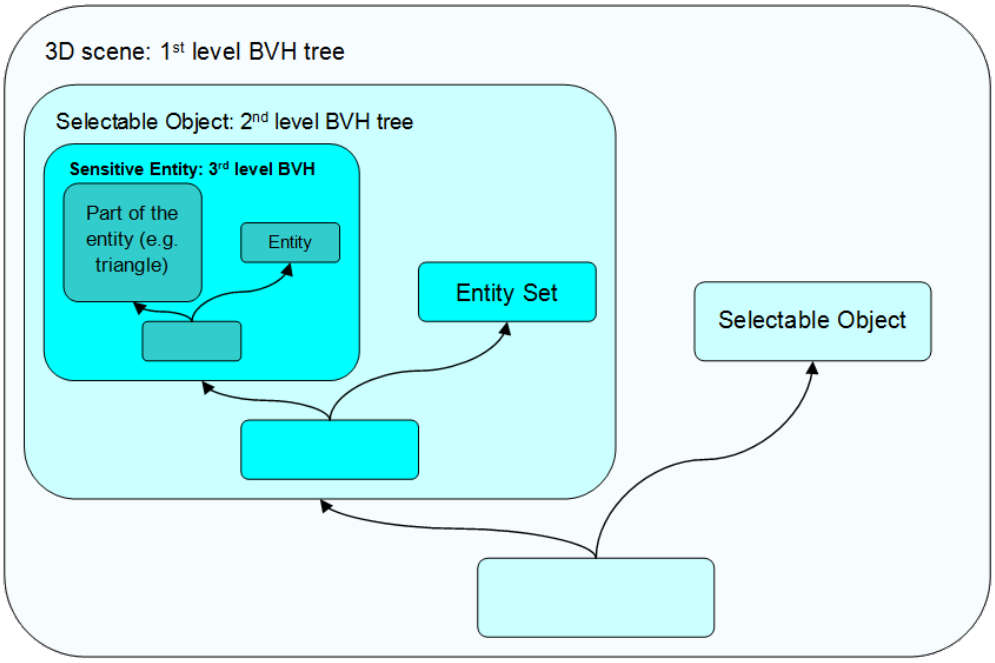


Figure 7: Selection BVH tree hierarchy: from the biggest object-level (first) to the smallest complex entity level (third)

Stages of the algorithm

The algorithm includes pre-processing and three main stages.

Pre-processing

Implies calculation of the selection frustum and its main characteristics.

First stage – traverse of the first level BVH tree

After successful building of the selection frustum, the algorithm starts traversal of the object-level BVH tree. The nodes containing axis-aligned bounding boxes are tested for overlap with the selection frustum following the terms of *separating axis theorem (SAT)*. When the traversal goes down to the leaf node, it means that a candidate object with possibly overlapping sensitive entities has been found. If no such objects have been detected, the algorithm stops and it is assumed that no object needs to be selected. Otherwise it passes to the next stage to process the entities of the found selectable object.

Second stage – traversal of the second level BVH tree

At this stage it is necessary to determine if there are candidates among all sensitive entities of one object.

First of all, at this stage the algorithm checks if there is any transformation applied for the current object. If it has its own location, then the correspondingly transformed frustum will be used for further calculations. At the next step the nodes of the second level BVH tree of the given object are visited to search for overlapping leaves. If no such leafs have been found, the algorithm returns to the second stage. Otherwise it starts processing the found entities by performing the following checks:

- activation check - the entity may be inactive at the moment as it belongs to deactivated selection;
- tolerance check - current selection frustum may be too large for further checks as it is always built with the maximum tolerance among all activated entities; thus, at this step the frustum may be scaled.

After these checks the algorithm passes to the last stage.

Third stage – overlap or inclusion test of a particular sensitive entity

If the entity is atomic, a simple SAT test is performed. In case of a complex entity, the third level BVH tree is traversed. The quantitative characteristics (like depth, distance to the center of geometry) of matched sensitive entities is analyzed and clipping planes are applied (if they have been set). The result of detection is stored and the algorithm returns to the second stage.

2.2.3 Packages and classes

Selection is implemented as a combination of various algorithms divided among several packages – *SelectBasics*, *Select3D*, *SelectMgr* and *StdSelect*.

SelectBasics

SelectBasics package contains basic classes and interfaces for selection. The most notable are:

- *SelectBasics_PickResult* – the structure for storing quantitative results of detection procedure, for example, depth and distance to the center of geometry;
- *SelectBasics_SelectingVolumeManager* – the interface for interaction with the current selection frustum.

Each custom sensitive entity must inherit at least *SelectBasics_SensitiveEntity*.

Select3D

Select3D package provides a definition of standard sensitive entities, such as:

- box;
- circle;

- curve;
- face;
- group;
- point;
- segment;
- triangle;
- triangulation;
- wire.

Each basic sensitive entity inherits *Select3D_SensitiveEntity*. The package also contains two auxiliary classes, *Select3D_SensitivePoly* and *Select3D_SensitiveSet*.

Select3D_SensitiveEntity – the base definition of a sensitive entity.

Select3D_SensitiveSet – a base class for all complex sensitive entities that require the third level BVH usage. It implements traverse of the tree and defines an interface for the methods that check sub-entities.

Select3D_SensitivePoly – describes an arbitrary point set and implements basic functions for selection. It is important to know that this class does not perform any internal data checks. Hence, custom implementations of sensitive entity inherited from *Select3D_SensitivePoly* must satisfy the terms of Separating Axis Theorem to use standard OCCT overlap detection methods.

SelectMgr

SelectMgr package is used to maintain the whole selection process. For this purpose, the package provides the following services:

- activation and deactivation of selection modes for all selectable objects;
- interfaces to compute selection mode of the object;
- definition of selection filter classes;
- keeping selection BVH data up-to-date.

A brief description of the main classes:

- *SelectMgr_BaseFrustum*, *SelectMgr_Frustum*, *SelectMgr_RectangularFrustum*, *SelectMgr_Triangular↩ Frustum* and *SelectMgr_TriangularFrustumSet* – interfaces and implementations of selecting frustums. These classes implement different SAT tests for overlap and inclusion detection. They also contain methods to measure characteristics of detected entities (depth, distance to center of geometry).
- *SelectMgr_SensitiveEntity*, *SelectMgr_Selection* and *SelectMgr_SensitiveEntitySet* – store and handle sensitive entities. *SelectMgr_SensitiveEntitySet* implements a primitive set for the second level BVH tree.
- *SelectMgr_SelectableObject* and *SelectMgr_SelectableObjectSet* – describe selectable objects. They also manage storage, calculation and removal of selections. *SelectMgr_SelectableObjectSet* implements a primitive set for the first level BVH tree.
- *SelectMgr_ViewerSelector* – encapsulates all logics of the selection algorithm and implements the third level BVH tree traverse.
- *SelectMgr_SelectionManager* – manages activation/deactivation, calculation and update of selections of every selectable object, and keeps BVH data up-to-date.

StdSelect

StdSelect package contains the implementation of some *SelectMgr* classes and tools for creation of selection structures. For example,

- *StdSelect_BRepOwner* – defines an entity owner with a link to its topological shape and methods for highlighting;
- *StdSelect_BRepSelectionTool* – contains algorithms for splitting standard AIS shapes into sensitive primitives;
- *StdSelect_FaceFilter*, *StdSelect_EdgeFilter* – implementation of selection filters.

2.2.4 Examples of usage

The first code snippet illustrates the implementation of *SelectMgr_SelectableObject::ComputeSelection()* method in a custom interactive object. The method is used for computation of user-defined selection modes. Let us assume it is required to make a box selectable in two modes – the whole shape (mode 0) and each of its edges (mode 1). To select the whole box, the application can create a sensitive primitive for each face of the interactive object. In this case, all primitives share the same owner – the box itself. To select box's edge, the application must create one sensitive primitive per edge. Here all sensitive entities cannot share the owner since different geometric primitives must be highlighted as the result of selection procedure.

```
void InteractiveBox::ComputeSelection (const Handle(SelectMgr_Selection)& theSel,
                                       const Standard_Integer theMode)
{
  switch (theMode)
  {
    case 0:   // creation of face sensitives for selection of the whole box
    {
      Handle(SelectMgr_EntityOwner) anOwner = new SelectMgr_EntityOwner (this, 5);
      for (Standard_Integer aFaceIter = 1; aFaceIter <= myNbFaces; ++aFaceIter)
      {
        Select3D_TypeOfSensitivity aSensType = myIsInterior;
        theSel->Add (new Select3D_SensitiveFace (anOwner, myFaces[aFaceIter]->PointArray(), aSensType));
      }
      break;
    }
    case 1: // creation of edge sensitives for selection of box edges only
    {
      for (Standard_Integer anEdgeIter = 1; anEdgeIter <= 12; ++anEdgeIter)
      {
        // 1 owner per edge, where 6 is a priority of the sensitive
        Handle(MySelection_EdgeOwner) anOwner = new MySelection_EdgeOwner (this, anEdgeIter, 6);
        theSel->Add (new Select3D_SensitiveSegment (anOwner, myFirstPnt[anEdgeIter]), myLastPnt[anEdgeIter]
      ));
      }
      break;
    }
  }
}
```

The algorithms for creating selection structures store sensitive primitives in *SelectMgr_Selection* instance. Each *SelectMgr_Selection* sequence in the list of selections of the object must correspond to a particular selection mode. To describe the decomposition of the object into selectable primitives, a set of ready-made sensitive entities is supplied in *Select3D* package. Custom sensitive primitives can be defined through inheritance from *Select3D_SensitiveEntity*. To make custom interactive objects selectable or customize selection modes of existing objects, the entity owners must be defined. They must inherit *SelectMgr_EntityOwner* interface.

Selection structures for any interactive object are created in *SelectMgr_SelectableObject::ComputeSelection()* method. The example below shows how computation of different selection modes of the topological shape can be done using standard OCCT mechanisms, implemented in *StdSelect_BRepSelectionTool*.

```
void MyInteractiveObject::ComputeSelection (const Handle(SelectMgr_Selection)& theSelection,
                                            const Standard_Integer theMode)
{
  switch (theMode)
  {
    case 0: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_SHAPE);  break;
    case 1: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_VERTEX); break;
    case 2: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_EDGE);   break;
```

```
    case 3: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_WIRE);   break;
    case 4: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_FACE);   break;
  }
}
```

The *StdSelect_BRepSelectionTool* class provides a high level API for computing sensitive entities of the given type (for example, face, vertex, edge, wire and others) using topological data from the given *TopoDS_Shape*.

The traditional way of highlighting selected entity owners adopted by Open CASCADE Technology assumes that each entity owner highlights itself on its own. This approach has two drawbacks:

- each entity owner has to maintain its own *Graphic3d_Structure* object, that results in a considerable memory overhead;
- drawing selected owners one by one is not efficient from the visualization point of view.

Therefore, to overcome these limitations, OCCT has an alternative way to implement the highlighting of a selected presentation. Using this approach, the interactive object itself will be responsible for the highlighting, not the entity owner.

On the basis of *SelectMgr_EntityOwner::IsAutoHilight()* return value, *AIS_InteractiveContext* object either uses the traditional way of highlighting (in case if *IsAutoHilight()* returns TRUE) or groups such owners according to their selectable objects and finally calls *SelectMgr_SelectableObject::HilightSelected()* or *SelectMgr_SelectableObject::ClearSelected()*, passing a group of owners as an argument.

Hence, an application can derive its own interactive object and redefine virtual methods *HilightSelected()*, *ClearSelected()* and *HilightOwnerWithColor()* from *SelectMgr_SelectableObject*. *SelectMgr_SelectableObject::GetHilightPresentation* and *SelectMgr_SelectableObject::GetSelectPresentation* methods can be used to optimize filling of selection and highlight presentations according to the user's needs.

After all the necessary sensitive entities are computed and packed in *SelectMgr_Selection* instance with the corresponding owners in a redefinition of *SelectMgr_SelectableObject::ComputeSelection()* method, it is necessary to register the prepared selection in *SelectMgr_SelectionManager* through the following steps:

- if there was no *AIS_InteractiveContext* opened, create an interactive context and display the selectable object in it;
- load the selectable object to the selection manager of the interactive context using *AIS_InteractiveContext::Load()* method. If the selection mode passed as a parameter to this method is not equal to -1, *ComputeSelection()* for this selection mode will be called;
- activate or deactivate the defined selection mode using *AIS_InteractiveContext::Activate()* or *AIS_InteractiveContext::Deactivate()* methods.

After these steps, the selection manager of the created interactive context will contain the given object and its selection entities, and they will be involved in the detection procedure.

The code snippet below illustrates the above steps. It also contains the code to start the detection procedure and parse the results of selection.

```
// Suppose there is an instance of class InteractiveBox from the previous sample.
// It contains an implementation of method InteractiveBox::ComputeSelection() for selection
// modes 0 (whole box must be selected) and 1 (edge of the box must be selectable)
Handle(InteractiveBox) theBox;
Handle(AIS_InteractiveContext) theContext;
// To prevent automatic activation of the default selection mode
theContext->SetAutoActivateSelection (false);
theContext->Display (theBox, false);

// Load a box to the selection manager without computation of any selection mode
theContext->Load (theBox, -1, true);
// Activate edge selection
theContext->Activate (theBox, 1);

// Run the detection mechanism for activated entities in the current mouse coordinates and in the current
       view.
// Detected owners will be highlighted with context highlight color
theContext->MoveTo (aXMousePos, aYMousePos, myView, false);
```

```
// Select the detected owners
theContext->Select();
// Iterate through the selected owners
for (theContext->InitSelected(); theContext->MoreSelected() && !aHasSelected; theContext->NextSelected())
{
  Handle(AIS_InteractiveObject) anIO = theContext->SelectedInteractive();
}

// deactivate all selection modes for aBox1
theContext->Deactivate (aBox1);
```

It is also important to know, that there are 2 types of detection implemented for rectangular selection in OCCT:

- **inclusive** detection. In this case the sensitive primitive is considered detected only when all its points are included in the area defined by the selection rectangle;
- **overlap** detection. In this case the sensitive primitive is considered detected when it is partially overlapped by the selection rectangle.

The standard OCCT selection mechanism uses inclusion detection by default. To change this, use the following code:

```
// Assume there is a created interactive context
const Handle(AIS_InteractiveContext) theContext;
// Retrieve the current viewer selector
const Handle(StdSelect_ViewerSelector3d)& aMainSelector = theContext->MainSelector();
// Set the flag to allow overlap detection
aMainSelector->AllowOverlapDetection (true);
```

3 Application Interactive Services

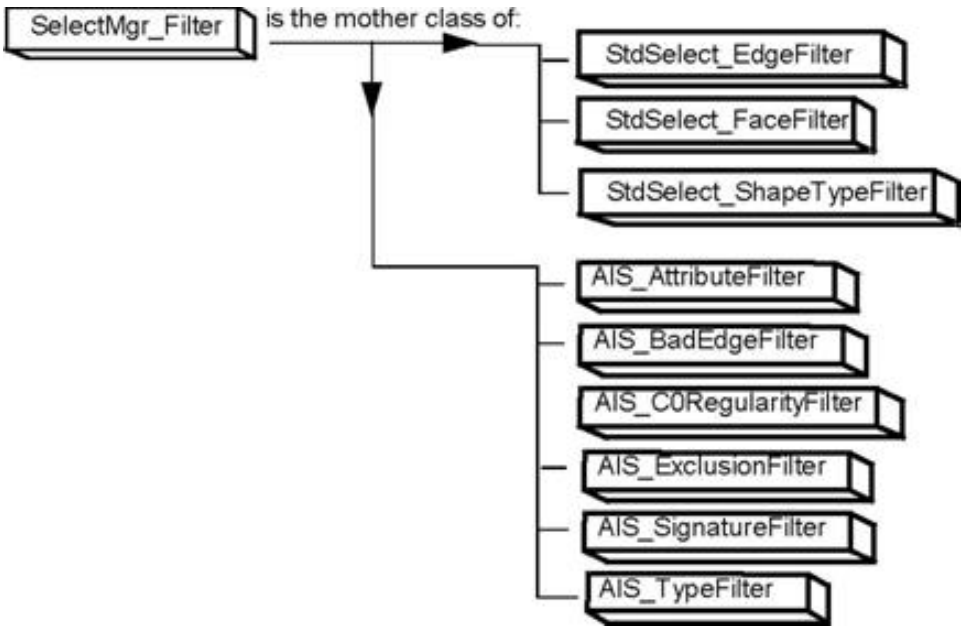
3.1 Introduction

Application Interactive Services allow managing presentations and dynamic selection in a viewer in a simple and transparent manner. The central entity for management of visualization and selections is the **Interactive Context** (*AIS_InteractiveContext*). It is connected to the main viewer (*V3d_Viewer*).

Interactive context by default starts at **Neutral Point** with each selectable object picked as a whole, but the user might activate **Local Selection** for specific objects to make selectable parts of the objects. Local/global selection is managed by a list of selection modes activated for each displayed object with 0 (default selection mode) usually meaning Global (entire object) selection.

Interactive Objects (*AIS_InteractiveObject*) are the entities, which are visualized and selected. You can use classes of standard interactive objects for which all necessary functions have already been programmed, or you can implement your own classes of interactive objects, by respecting a certain number of rules and conventions described below.

An Interactive Object is a "virtual" entity, which can be presented and selected. An Interactive Object can have a certain number of specific graphic attributes, such as visualization mode, color and material. When an Interactive Object is visualized, the required graphic attributes are taken from its own **Drawer** (*Prs3d_Drawer*) if it has the required custom attributes or otherwise from the context drawer.



It can be necessary to filter the entities to be selected. Consequently there are **Filter** entities (*SelectMgr_Filter*), which allow refining the dynamic detection context. Some of these filters can be used only within at the Neutral Point, others only within Local Selection. It is possible to program custom filters and load them into the interactive context.

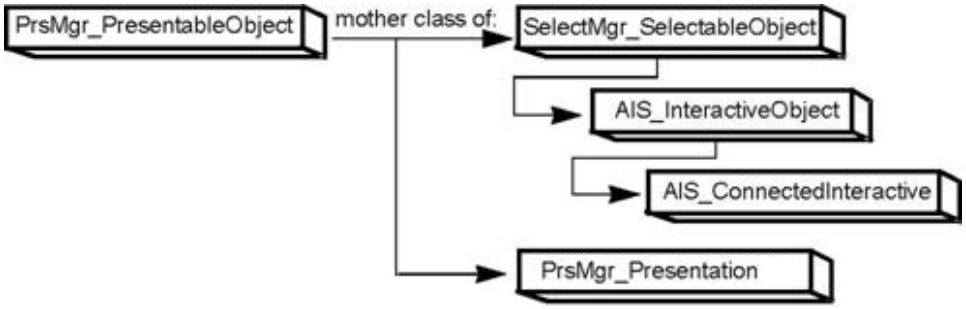
3.2 Interactive objects

Entities which are visualized and selected in the AIS viewer are objects (*AIS_InteractiveObject*). They connect the underlying reference geometry of a model to its graphic representation in *AIS*. You can use the predefined OCCT classes of standard interactive objects, for which all necessary functions have already been programmed, or, if you are an advanced user, you can implement your own classes of interactive objects.

3.2.1 Presentations

An interactive object can have as many presentations as its creator wants to give it. 3D presentations are managed by **Presentation Manager** (*PrsMgr_PresentationManager*). As this is transparent in AIS, the user does not have to worry about it.

A presentation is identified by an index (*Display Mode*) and by the reference to the Presentation Manager, which it depends on. By convention, the default mode of representation for the Interactive Object has index 0.



Calculation of different presentations of an interactive object is done by the *Compute* functions inheriting from *PrsMgr_PresentableObject::Compute* functions. They are automatically called by *PresentationManager* at a visualization or an update request.

If you are creating your own type of interactive object, you must implement the Compute function in one of the following ways:

For 3D:

```
void PackageName_ClassName::Compute (const Handle(PrsMgr_PresentationManager3d)& thePresentationManager,
                                     const Handle(Prs3d_Presentation)& thePresentation,
                                     const Standard_Integer theMode);
```

For hidden line removal (HLR) mode in 3D:

```
void PackageName_ClassName::Compute (const Handle(Prs3d_Projector)& theProjector,
                                     const Handle(Prs3d_Presentation)& thePresentation);
```

3.2.2 Hidden Line Removal

The view can have two states: the normal mode or the computed mode (Hidden Line Removal mode). When the latter is active, the view looks for all presentations displayed in the normal mode, which have been signaled as accepting HLR mode. An internal mechanism allows calling the interactive object's own *Compute*, that is projector function.

By convention, the Interactive Object accepts or rejects the representation of HLR mode. It is possible to make this declaration in one of two ways:

- Initially by using one of the values of the enumeration *PrsMgr_TypeOfPresentation3d*:
  - *PrsMgr_TOP_AllView*,
  - *PrsMgr_TOP_ProjectorDependant*
- Later by using the function *PrsMgr_PresentableObject::SetTypeOfPresentation*

AIS_Shape class is an example of an interactive object that supports HLR representation. The type of the HLR algorithm is stored in *Prs3d_Drawer* of the shape. It is a value of the *Prs3d_TypeOfHLR* enumeration and can be set to:

- *Prs3d_TOH_PolyAlgo* for a polygonal algorithm based on the shape's triangulation;

- *Prs3d_TOH_Algo* for an exact algorithm that works with the shape's real geometry;
- *Prs3d_TOH_NotSet* if the type of algorithm is not set for the given interactive object instance.

The type of the HLR algorithm used for *AIS_Shape* can be changed by calling the *AIS_Shape::SetTypeOfHLR()* method. The current HLR algorithm type can be obtained using *AIS_Shape::TypeOfHLR()* method is to be used.

These methods get the value from the drawer of *AIS_Shape*. If the HLR algorithm type in the *Prs3d_Drawer* is set to *Prs3d_TOH_NotSet*, the *Prs3d_Drawer* gets the value from the default drawer of *AIS_InteractiveContext*. So it is possible to change the default HLR algorithm used by all newly displayed interactive objects. The value of the HLR algorithm type stored in the context drawer can be *Prs3d_TOH_Algo* or *Prs3d_TOH_PolyAlgo*. The polygonal algorithm is the default one.

3.2.3 Presentation modes

There are four types of interactive objects in AIS:

- the "construction element" or Datum,
- the Relation (dimensions and constraints)
- the Object
- the None type (when the object is of an unknown type).

Inside these categories, additional characterization is available by means of a signature (an index). By default, the interactive object has a NONE type and a signature of 0 (equivalent to NONE). If you want to give a particular type and signature to your interactive object, you must redefine two virtual functions:

- *AIS_InteractiveObject::Type*
- *AIS_InteractiveObject::Signature*.

Note that some signatures are already used by "standard" objects provided in AIS (see the List of Standard Interactive Object Classes).

The interactive context can have a default mode of representation for the set of interactive objects. This mode may not be accepted by a given class of objects. Consequently, to get information about this class it is necessary to use virtual function *AIS_InteractiveObject::AcceptDisplayMode*.

Display Mode

The functions *AIS_InteractiveContext::SetDisplayMode* and *AIS_InteractiveContext::UnsetDisplayMode* allow setting a custom display mode for an objects, which can be different from that proposed by the interactive context.

Highlight Mode

At dynamic detection, the presentation echoed by the Interactive Context, is by default the presentation already on the screen.

The functions *AIS_InteractiveObject::SetHilightMode* and *AIS_InteractiveObject::UnsetHilightMode* allow specifying the display mode used for highlighting (so called highlight mode), which is valid independently from the active representation of the object. It makes no difference whether this choice is temporary or definitive.

Note that the same presentation (and consequently the same highlight mode) is used for highlighting *detected* objects and for highlighting *selected* objects, the latter being drawn with a special *selection color* (refer to the section related to *Interactive Context* services).

For example, you want to systematically highlight the wireframe presentation of a shape - non regarding if it is visualized in wireframe presentation or with shading. Thus, you set the highlight mode to *0* in the constructor of the interactive object. Do not forget to implement this representation mode in the *Compute* functions.

Infinite Status

If you do not want an object to be affected by a *FitAll* view, you must declare it infinite; you can cancel its "infinite" status using *AIS_InteractiveObject::SetInfiniteState* and *AIS_InteractiveObject::IsInfinite* functions.

Let us take for example the class called *IShape* representing an interactive object:

```
myPk_IShape::myPk_IShape (const TopoDS_Shape& theShape, PrsMgr_TypeOfPresentation theType)
: AIS_InteractiveObject (theType), myShape (theShape) { SetHilightMode (0); }

Standard_Boolean myPk_IShape::AcceptDisplayMode (const Standard_Integer theMode) const
{
  return theMode == 0 || theMode == 1;
}

void myPk_IShape::Compute (const Handle(PrsMgr_PresentationManager3d)& thePrsMgr,
                           const Handle(Prs3d_Presentation)& thePrs,
                           const Standard_Integer theMode)
{
  switch (theMode)
  {
    // algo for calculation of wireframe presentation
    case 0: StdPrs_WFDeflectionShape::Add (thePrs, myShape, myDrawer); return;
    // algo for calculation of shading presentation
    case 1: StdPrs_ShadedShape::Add (thePrs, myShape, myDrawer); return;
  }
}

void myPk_IShape::Compute (const Handle(Prs3d_Projector)& theProjector,
                           const Handle(Prs3d_Presentation)& thePrs)
{
  // Hidden line mode calculation algorithm
  StdPrs_HLRPolyShape::Add (thePrs, myShape, myDrawer, theProjector);
}
```

3.2.4 Selection

An interactive object can have an indefinite number of selection modes, each representing a "decomposition" into sensitive primitives. Each primitive has an **Owner** (*SelectMgr_EntityOwner*) which allows identifying the exact interactive object or shape which has been detected (see Selection chapter).

The set of sensitive primitives, which correspond to a given mode, is stocked in a **Selection** (*SelectMgr_Selection*).

Each selection mode is identified by an index. By convention, the default selection mode that allows us to grasp the interactive object in its entirety is mode *0*. However, it can be modified in the custom interactive objects using method *SelectMgr_SelectableObject::setGlobalSelMode()*.

The calculation of selection primitives (or sensitive entities) is done in a virtual function *ComputeSelection*. It should be implemented for each type of interactive object that is assumed to have different selection modes using the function *AIS_InteractiveObject::ComputeSelection*. A detailed explanation of the mechanism and the manner of implementing this function has been given in Selection chapter.

There are some examples of selection mode calculation for the most widely used interactive object in OCCT – *AIS_Shape* (selection by vertex, by edges, etc). To create new classes of interactive objects with the same selection behavior as *AIS_Shape* – such as vertices and edges – you must redefine the virtual function *AIS_InteractiveObject::AcceptShapeDecomposition*.

3.2.5 Graphic attributes

Graphic attributes manager, or *Prs3d_Drawer*, stores graphic attributes for specific interactive objects and for interactive objects controlled by interactive context.

Initially, all drawer attributes are filled out with the predefined values which will define the default 3D object appearance. When an interactive object is visualized, the required graphic attributes are first taken from its own drawer if one exists, or from the context drawer if no specific drawer for that type of object exists.

Keep in mind the following points concerning graphic attributes:

- Each interactive object can have its own visualization attributes.

- By default, the interactive object takes the graphic attributes of the context in which it is visualized (visualization mode, deflection values for the calculation of presentations, number of isoparameters, color, type of line, material, etc.)
- In the *AIS_InteractiveObject* abstract class, standard attributes including color, line thickness, material, and transparency have been privileged. Consequently, there is a certain number of virtual functions, which allow acting on these attributes. Each new class of interactive object can redefine these functions and change the behavior of the class.

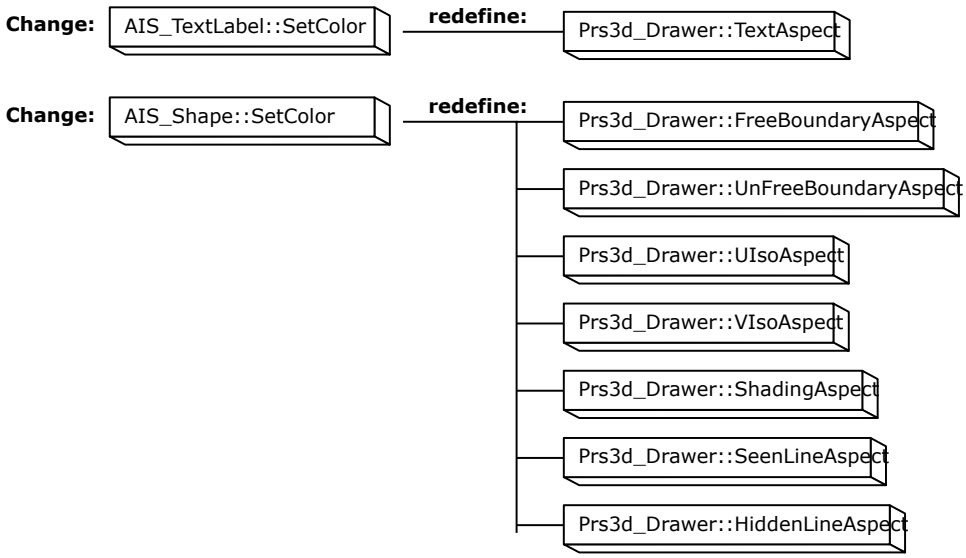


Figure 8: Redefinition of virtual functions for changes in AIS_Shape and AIS_TextLabel.

The following virtual functions provide settings for color, width, material and transparency:

- *AIS_InteractiveObject::UnsetColor*
- *AIS_InteractiveObject::SetWidth*
- *AIS_InteractiveObject::UnsetWidth*
- *AIS_InteractiveObject::SetMaterial*
- *AIS_InteractiveObject::UnsetMaterial*
- *AIS_InteractiveObject::SetTransparency*
- *AIS_InteractiveObject::UnsetTransparency*

These methods can be used as a shortcut assigning properties in common way, but result might be not available. Some interactive objects might not implement these methods at all or implement only a sub-set of them. Direct modification of *Prs3d_Drawer* properties returned by *AIS_InteractiveObject::Attributes* can be used for more precise and predictable configuration.

It is important to know which functions may imply the recalculation of presentations of the object. If the presentation mode of an interactive object is to be updated, a flag from *PrsMgr_PresentableObject* indicates this. The mode can be updated using the functions *Display* and *Redisplay* in *AIS_InteractiveContext*.

3.2.6 Complementary Services

When you use complementary services for interactive objects, pay special attention to the cases mentioned below.

Change the location of an interactive object

The following functions allow "moving" the representation and selection of Interactive Objects in a view without recalculation (and modification of the original shape).

- *AIS_InteractiveContext::SetLocation*
- *AIS_InteractiveContext::ResetLocation*
- *AIS_InteractiveContext::HasLocation*
- *AIS_InteractiveContext::Location*

Connect an interactive object to an applicative entity

Each Interactive Object has functions that allow attributing it an *GetOwner* in form of a *Transient*.

- *AIS_InteractiveObject::SetOwner*
- *AIS_InteractiveObject::HasOwner*
- *AIS_InteractiveObject::GetOwner*

An interactive object can therefore be associated or not with an applicative entity, without affecting its behavior.

NOTE: Don't be confused by owners of another kind - *SelectMgr_EntityOwner* used for identifying selectable parts of the object or object itself.

Resolving coincident topology

Due to the fact that the accuracy of three-dimensional graphics coordinates has a finite resolution the elements of topological objects can coincide producing the effect of "popping" some elements one over another.

To the problem when the elements of two or more Interactive Objects are coincident you can apply the polygon offset. It is a sort of graphics computational offset, or depth buffer offset, that allows you to arrange elements (by modifying their depth value) without changing their coordinates. The graphical elements that accept this kind of offsets are solid polygons or displayed as boundary lines and points. The polygons could be displayed as lines or points by setting the appropriate interior style.

The methods *AIS_InteractiveObject::SetPolygonOffsets* and *AIS_InteractiveContext::SetPolygonOffsets* allow setting up the polygon offsets.

3.2.7 Object hierarchy

Each *PrsMgr_PresentableObject* has a list of objects called *myChildren*. Any transformation of *PrsMgr_↩ PresentableObject* is also applied to its children. This hierarchy does not propagate to *Graphic3d* level and below.

PrsMgr_PresentableObject sends its combined (according to the hierarchy) transformation down to *Graphic3d_↩ Structure*. The materials of structures are not affected by the hierarchy.

Object hierarchy can be controlled by the following API calls:

- *PrsMgr_PresentableObject::AddChild*;
- *PrsMgr_PresentableObject::RemoveChild*.

3.2.8 Instancing

The conception of instancing operates the object hierarchy as follows:

- Instances are represented by separated *AIS* objects.

- Instances do not compute any presentations.

Classes *AIS_ConnectedInteractive* and *AIS_MultipleConnectedInteractive* are used to implement this conception.

AIS_ConnectedInteractive is an object instance, which reuses the geometry of the connected object but has its own transformation and visibility flag. This connection is propagated down to *OpenGl* level, namely to *OpenGl_Structure*. *OpenGl_Structure* can be connected only to a single other structure.

AIS_ConnectedInteractive can be referenced to any *AIS_InteractiveObject* in general. When it is referenced to another *AIS_ConnectedInteractive*, it just copies the reference.

AIS_MultipleConnectedInteractive represents an assembly, which does not have its own presentation. The assemblies are able to participate in the object hierarchy and are intended to handle a grouped set of instanced objects. It behaves as a single object in terms of selection. It applies high level transformation to all sub-elements since it is located above in the hierarchy.

All *AIS_MultipleConnectedInteractive* are able to have child assemblies. Deep copy of object instances tree is performed if one assembly is attached to another.

Note that *AIS_ConnectedInteractive* cannot reference *AIS_MultipleConnectedInteractive*. *AIS_Connected↩ Interactive* copies sensitive entities of the origin object for selection, unlike *AIS_MultipleConnectedInteractive* that re-uses the entities of the origin object.

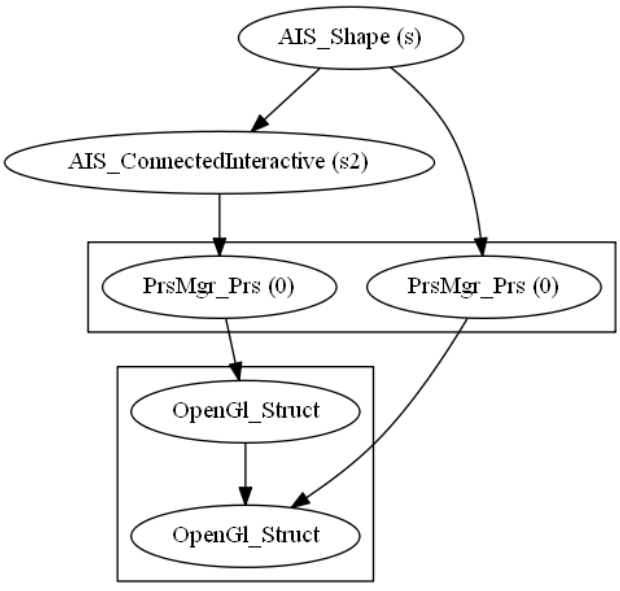
Instances can be controlled by the following DRAW commands:

- *vconnect* : Creates and displays *AIS_MultipleConnectedInteractive* object from input objects and location.
- *vconnectto* : Makes an instance of object with the given position.
- *vdisconnect* : Disconnects all objects from an assembly or disconnects an object by name or number.
- *vaddconnected* : Adds an object to the assembly.
- *vlistconnected* : Lists objects in the assembly.

Have a look at the examples below:

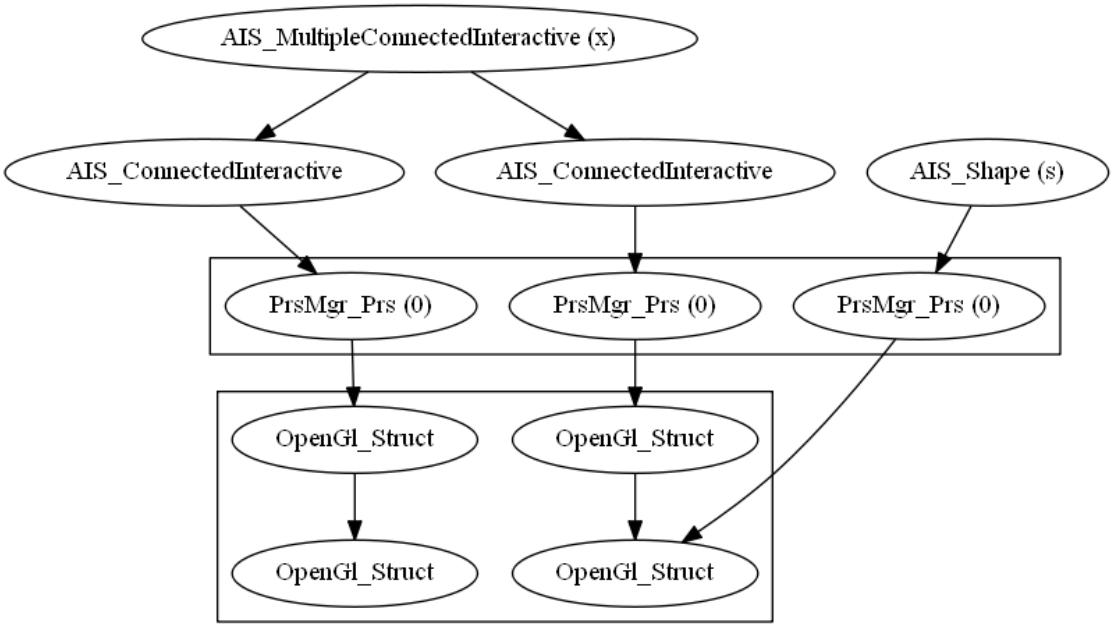
```
pload MODELING VISUALIZATION
vinit
psphere s 1
vdisplay s
vconnectto s2 3 0 0 s  # make instance
vfit
```

See how proxy *OpenGl_Structure* is used to represent instance:



The original object does not have to be displayed in order to make instance. Also selection handles transformations of instances correctly:

```
pload MODELING VISUALIZATION
vinit
psphere s 1
psphere p 0.5
vdisplay s              # p is not displayed
vsetloc s -2 0 0
vconnect x 3 0 0 s p    # make assembly
vfit
```



Here is the example of a more complex hierarchy involving sub-assemblies:

```
pload MODELING VISUALIZATION
vinit
box b 1 1 1
psphere s 0.5
vdisplay b s
vsetlocation s 0 2.5 0
box d 0.5 0.5 3
box d2 0.5 3 0.5
vdisplay d d2

vconnectto b1 -2 0 0 b
vconnect z 2 0 0 b s
vconnect z2 4 0 0 d d2
vconnect z3 6 0 0 z z2
vfit
```

3.3 Interactive Context

3.3.1 Rules

The Interactive Context allows managing in a transparent way the graphic and **selectable** behavior of interactive objects in one or more viewers. Most functions which allow modifying the attributes of interactive objects, and which were presented in the preceding chapter, will be looked at again here.

There is one essential rule to follow: the modification of an interactive object, which is already known by the Context, must be done using Context functions. You can only directly call the functions available for an interactive object if it has not been loaded into an Interactive Context.

```
Handle(AIS_Shape) aShapePrs = new AIS_Shape (theShape);
myIntContext->Display (aShapePrs, AIS_Shaded, 0, false, aShapePrs->AcceptShapeDecomposition());
myIntContext->SetColor(aShapePrs, Quantity_NOC_RED);
```

You can also write

```
Handle(AIS_Shape) aShapePrs = new AIS_Shape (theShape);
aShapePrs->SetColor (Quantity_NOC_RED);
aShapePrs->SetDisplayMode (AIS_Shaded);
myIntContext->Display (aShapePrs);
```

3.3.2 Groups of functions

Neutral Point and **Local Selection** constitute the two operating modes or states of the **Interactive Context**, which is the central entity which pilots visualizations and selections. The **Neutral Point**, which is the default mode, allows easily visualizing and selecting interactive objects, which have been loaded into the context. Activating **Local Selection** for specific Objects allows selecting of their sub-parts.

3.3.3 Management of the Interactive Context

An interactive object can have a certain number of specific graphic attributes, such as visualization mode, color, and material. Correspondingly, the interactive context has a set of graphic attributes, the *Drawer*, which is valid by default for the objects it controls. When an interactive object is visualized, the required graphic attributes are first taken from the object's own *Drawer* if it exists, or from the context drawer if otherwise.

The following adjustable settings allow personalizing the behavior of presentations and selections:

- Default Drawer, containing all the color and line attributes which can be used by interactive objects, which do not have their own attributes.
- Default Visualization Mode for interactive objects. By default: *mode 0*;
- Highlight color of entities detected by mouse movement. By default: *Quantity_NOC_CYAN1*;
- Pre-selection color. By default: *Quantity_NOC_GREEN*;
- Selection color (when you click on a detected object). By default: *Quantity_NOC_GRAY80*;

All of these settings can be modified by functions proper to the AIS_InteractiveContext. When you change a graphic attribute pertaining to the Context (visualization mode, for example), all interactive objects, which do not have the corresponding appropriate attribute, are updated.

Let us examine the case of two interactive objects: *theObj1* and *theObj2*:

```
theCtx->Display (theObj1, false);
theCtx->Display (theObj2, true);  // TRUE for viewer update
theCtx->SetDisplayMode (theObj1, 3, false);
theCtx->SetDisplayMode (2, true);
// theObj2 is visualized in mode 2 (if it accepts this mode)
// theObj1 stays visualized in its mode 3
```

PrsMgr_PresentationManager and *SelectMgr_ViewerSelector3d*, which manage the presentation and selection of present interactive objects, are associated to the main Viewer.

WARNING! Do NOT use integer values (like in sample above) in real code - use appropriate enumerations instead! Each presentable object has independent list of supported display and selection modes; for instance, *AIS_Display↩ Mode* enumeration is applicable only to *AIS_Shape* presentations.

3.4 Local Selection

3.4.1 Selection Modes

The Local Selection is defined by index (Selection Mode). The Selection Modes implemented by a specific interactive object and their meaning should be checked within the documentation of this class. See, for example, *MeshVS_SelectionModeFlags* for *MeshVS_Mesh* object.

AIS_Shape is the most used interactive object. It provides API to manage selection operations on the constituent elements of shapes (selection of vertices, edges, faces, etc.). The Selection Mode for a specific shape type (*TopAbs_ShapeEnum*) is returned by method *AIS_Shape::SelectionMode()*.

The method *AIS_InteractiveContext::Display()* without a Selection Mode argument activates the default Selection Mode of the object. The methods *AIS_InteractiveContext::Activate()* and *AIS_InteractiveContext::Deactivate()* activate and deactivate a specific Selection Mode.

More than one Selection Mode can be activated at the same time (but default 0 mode for selecting entire object is exclusive - it cannot be combined with others). The list of active modes can be retrieved using function *AIS_InteractiveContext::ActivatedModes*.

3.4.2 Filters

To define an environment of dynamic detection, you can use standard filter classes or create your own. A filter questions the owner of the sensitive primitive to determine if it has the desired qualities. If it answers positively, it is kept. If not, it is rejected.

The root class of objects is *SelectMgr_Filter*. The principle behind it is straightforward: a filter tests to see whether the owners (*SelectMgr_EntityOwner*) detected in mouse position by selector answer *OK*. If so, it is kept, otherwise it is rejected. You can create a custom class of filter objects by implementing the deferred function *SelectMgr_Filter::IsOk()*.

In *SelectMgr*, there are also Composition filters (AND Filters, OR Filters), which allow combining several filters. In Interactive Context, all filters that you add are stored in an OR filter (which answers *OK* if at least one filter answers *OK*).

There are Standard filters, which have already been implemented in several packages:

- *StdSelect_EdgeFilter* – for edges, such as lines and circles;
- *StdSelect_FaceFilter* – for faces, such as planes, cylinders and spheres;
- *StdSelect_ShapeTypeFilter* – for shape types, such as compounds, solids, shells and wires;
- *AIS_TypeFilter* – for types of interactive objects;
- *AIS_SignatureFilter* – for types and signatures of interactive objects;
- *AIS_AttributeFilter* – for attributes of Interactive Objects, such as color and width.

There are several functions to manipulate filters:

- *AIS_InteractiveContext::AddFilter* adds a filter passed as an argument.
- *AIS_InteractiveContext::RemoveFilter* removes a filter passed as an argument.
- *AIS_InteractiveContext::RemoveFilters* removes all present filters.
- *AIS_InteractiveContext::Filters* gets the list of filters active in a context.

Example

```
// shading visualization mode, no specific mode, authorization for decomposition into sub-shapes
const TopoDS_Shape theShape;
Handle(AIS_Shape) aShapePrs = new AIS_Shape (theShape);
myContext->Display (aShapePrs, AIS_Shaded, -1, true, true);

// activates decomposition of shapes into faces
const int aSubShapeSelMode = AIS_Shape::SelectionMode (TopAbs_Face);
myContext->Activate (aShapePrs, aSubShapeSelMode);

Handle(StdSelect_FaceFilter) aFil1 = new StdSelect_FaceFilter (StdSelect_Revol);
Handle(StdSelect_FaceFilter) aFil2 = new StdSelect_FaceFilter (StdSelect_Plane);
myContext->AddFilter (aFil1);
myContext->AddFilter (aFil2);

// only faces of revolution or planar faces will be selected
myContext->MoveTo (thePixelX, thePixelY, myView, true);
```

3.4.3 Selection

Dynamic detection and selection are put into effect in a straightforward way. There are only a few conventions and functions to be familiar with:

- *AIS_InteractiveContext::MoveTo* – passes mouse position to Interactive Context selectors.
- *AIS_InteractiveContext::Select* – stores what has been detected at the last *MoveTo*. Replaces the previously selected object. Empties the stack if nothing has been detected at the last move.
- *AIS_InteractiveContext::ShiftSelect* – if the object detected at the last move was not already selected, it is added to the list of the selected objects. If not, it is withdrawn. Nothing happens if you click on an empty area.
- *AIS_InteractiveContext::Select* – selects everything found in the surrounding area.
- *AIS_InteractiveContext::ShiftSelect* – selects what was not previously in the list of selected, deselects those already present.

Highlighting of detected and selected entities is automatically managed by the Interactive Context. The Highlight colors are those dealt with above. You can nonetheless disconnect this automatic mode if you want to manage this part yourself:

```
AIS_InteractiveContext::SetAutomaticHilight
AIS_InteractiveContext::AutomaticHilight
```

You can question the Interactive context by moving the mouse. The following functions can be used:

- *AIS_InteractiveContext::HasDetected* – checks if there is a detected entity;
- *AIS_InteractiveContext::DetectedOwner* – returns the (currently highlighted) detected entity.

After using the *Select* and *ShiftSelect* functions, you can explore the list of selections. The following functions can be used:

- *AIS_InteractiveContext::InitSelected* – initializes an iterator;
- *AIS_InteractiveContext::MoreSelected* – checks if the iterator is valid;
- *AIS_InteractiveContext::NextSelected* – moves the iterator to the next position;
- *AIS_InteractiveContext::SelectedOwner* – returns an entity at the current iterator position.

The owner object *SelectMgr_EntityOwner* is a key object identifying the selectable entity in the viewer (returned by methods *AIS_InteractiveContext::DetectedOwner* and *AIS_InteractiveContext::SelectedOwner*). The Interactive Object itself can be retrieved by method *SelectMgr_EntityOwner::Selectable*, while identifying a sub-part depends on the type of Interactive Object. In case of *AIS_Shape*, the (sub)shape is returned by method *StdSelect_BRep↩ Owner::Shape*.

Example

```
for (myAISCtx->InitSelected(); myAISCtx->MoreSelected(); myAISCtx->NextSelected())
{
  Handle(SelectMgr_EntityOwner) anOwner = myAISCtx->SelectedOwner();
  Handle(AIS_InteractiveObject) anObj = Handle(AIS_InteractiveObject)::DownCast (anOwner->Selectable());
  if (Handle(StdSelect_BRepOwner) aBRepOwner = Handle(StdSelect_BRepOwner)::DownCast (anOwner))
  {
    // to be able to use the picked shape
    TopoDS_Shape aShape = aBRepOwner->Shape();
  }
}
```

3.5 Standard Interactive Object Classes

Interactive Objects are selectable and viewable objects connecting graphic representation and the underlying reference geometry.

They are divided into four types:

- the **Datum** – a construction geometric element;
- the **Relation** – a constraint on the interactive shape and the corresponding reference geometry;
- the **Object** – a topological shape or connection between shapes;
- **None** – a token, that instead of eliminating the object, tells the application to look further until it finds an acceptable object definition in its generation.

Inside these categories, there is a possibility of additional characterization by means of a signature. The signature provides an index to the further characterization. By default, the **Interactive Object** has a *None* type and a signature of 0 (equivalent to *None*). If you want to give a particular type and signature to your interactive object, you must redefine the two virtual methods: *Type* and *Signature*.

3.5.1 Datum

The **Datum** groups together the construction elements such as lines, circles, points, trihedrons, plane trihedrons, planes and axes.

AIS_Point, AIS_Axis, AIS_Line, AIS_Circle, AIS_Plane and *AIS_Trihedron* have four selection modes:

- mode AIS_TrihedronSelectionMode_EntireObject : selection of a trihedron;
- mode AIS_TrihedronSelectionMode_Origin : selection of the origin of the trihedron;
- mode AIS_TrihedronSelectionMode_Axes : selection of the axes;
- mode AIS_TrihedronSelectionMode_MainPlanes : selection of the planes XOY, YOZ, XOZ.

when you activate one of modes, you pick AIS objects of type:

- *AIS_Point*;
- *AIS_Axis* (and information on the type of axis);
- *AIS_Plane* (and information on the type of plane).

AIS_PlaneTrihedron offers three selection modes:

- mode 0 : selection of the whole trihedron;
- mode 1 : selection of the origin of the trihedron;
- mode 2 : selection of the axes – same remarks as for the Trihedron.

For the presentation of planes and trihedra, the default length unit is millimeter and the default value for the representation of axes is 10. To modify these dimensions, you must temporarily recover the object **Drawer**. From it, take the *DatumAspect()* and change the value *FirstAxisLength*. Finally, recalculate the presentation.

3.5.2 Object

The **Object** type includes topological shapes, and connections between shapes.

AIS_Shape has two visualization modes:

- mode AIS_WireFrame : Line (default mode)
- mode AIS_Shaded : Shading (depending on the type of shape)

AIS_ConnectedInteractive is an Interactive Object connecting to another interactive object reference, and located elsewhere in the viewer makes it possible not to calculate presentation and selection, but to deduce them from your object reference. *AIS_MultipleConnectedInteractive* is an object connected to a list of interactive objects (which can also be Connected objects; it does not require memory-hungry presentation calculations).

MeshVS_Mesh is an Interactive Object that represents meshes, it has a data source that provides geometrical information (nodes, elements) and can be built up from the source data with a custom presentation builder.

The class *AIS_ColoredShape* allows using custom colors and line widths for *TopoDS_Shape* objects and their sub-shapes.

```
AIS_ColoredShape aColoredShape = new AIS_ColoredShape (theShape);

// setup color of entire shape
aColoredShape->SetColor (Quantity_NOC_RED);

// setup line width of entire shape
aColoredShape->SetWidth (1.0);

// set transparency value
aColoredShape->SetTransparency (0.5);

// customize color of specified sub-shape
aColoredShape->SetCustomColor (theSubShape, Quantity_NOC_BLUE1);

// customize line width of specified sub-shape
aColoredShape->SetCustomWidth (theSubShape, 0.25);
```

The presentation class *AIS_PointCloud* can be used for efficient drawing of large arbitrary sets of colored points. It uses *Graphic3d_ArrayOfPoints* to pass point data into OpenGl graphic driver to draw a set points as an array of "point sprites". The point data is packed into vertex buffer object for performance.

- The type of point marker used to draw points can be specified as a presentation aspect.
- The presentation provides selection by a bounding box of the visualized set of points. It supports two display / highlighting modes: points or bounding box.

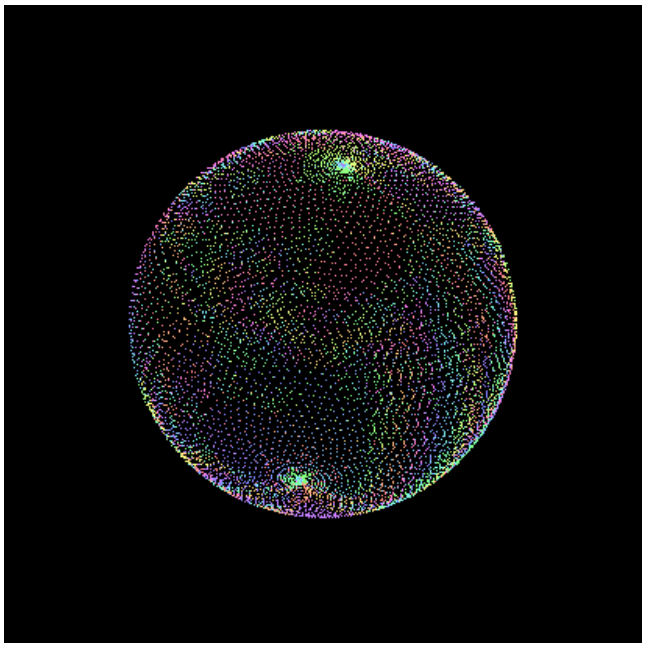


Figure 9: A random colored cloud of points

Example:

```
Handle(Graphic3d_ArrayOfPoints) aPoints = new Graphic3d_ArrayOfPoints (2000, Standard_True);
aPoints->AddVertex (gp_Pnt(-40.0, -40.0, -40.0), Quantity_Color (Quantity_NOC_BLUE1));
aPoints->AddVertex (gp_Pnt (40.0,  40.0,  40.0), Quantity_Color (Quantity_NOC_BLUE2));

Handle(AIS_PointCloud) aPntCloud = new AIS_PointCloud();
aPntCloud->SetPoints (aPoints);
```

The draw command *vpointcloud* builds a cloud of points from shape triangulation. This command can also draw a sphere surface or a volume with a large amount of points (more than one million).

3.5.3 Relations

The **Relation** is made up of constraints on one or more interactive shapes and the corresponding reference geometry. For example, you might want to constrain two edges in a parallel relation. This constraint is considered as an object in its own right, and is shown as a sensitive primitive. This takes the graphic form of a perpendicular arrow marked with the || symbol and lying between the two edges.

The following relations are provided by *PrsDim*:

- *PrsDim_ConcentricRelation*
- *PrsDim_FixRelation*
- *PrsDim_IdenticRelation*
- *PrsDim_ParallelRelation*
- *PrsDim_PerpendicularRelation*
- *PrsDim_Relation*
- *PrsDim_SymmetricRelation*
- *PrsDim_TangentRelation*

The list of relations is not exhaustive.

3.5.4 Dimensions

- *PrsDim_AngleDimension*
- *PrsDim_Chamf3dDimension*
- *PrsDim_DiameterDimension*
- *PrsDim_DimensionOwner*
- *PrsDim_LengthDimension*
- *PrsDim_OffsetDimension*
- *PrsDim_RadiusDimension*

3.5.5 MeshVS_Mesh

MeshVS_Mesh is an Interactive Object that represents meshes. This object differs from the *AIS_Shape* as its geometrical data is supported by the data source *MeshVS_DataSource* that describes nodes and elements of the object. As a result, you can provide your own data source.

However, the *DataSource* does not provide any information on attributes, for example nodal colors, but you can apply them in a special way – by choosing the appropriate presentation builder.

The presentations of *MeshVS_Mesh* are built with the presentation builders *MeshVS_PrsBuilder*. You can choose between the builders to represent the object in a different way. Moreover, you can redefine the base builder class and provide your own presentation builder.

You can add/remove builders using the following methods:

```
MeshVS_Mesh::AddBuilder (const Handle(MeshVS_PrsBuilder)& theBuilder, Standard_Boolean
      theToTreatAsHilighter);
MeshVS_Mesh::RemoveBuilder (const Standard_Integer theIndex);
MeshVS_Mesh::RemoveBuilderById (const Standard_Integer theId);
```

There is a set of reserved display and highlighting mode flags for *MeshVS_Mesh*. Mode value is a number of bits that allows selecting additional display parameters and combining the following mode flags, which allow displaying mesh in wireframe, shading and shrink modes:

```
MeshVS_DMF_WireFrame
MeshVS_DMF_Shading
MeshVS_DMF_Shrink
```

It is also possible to display deformed mesh in wireframe, shading or shrink modes using:

```
MeshVS_DMF_DeformedPrsWireFrame
MeshVS_DMF_DeformedPrsShading
MeshVS_DMF_DeformedPrsShrink
```

The following methods represent different kinds of data:

```
MeshVS_DMF_VectorDataPrs
MeshVS_DMF_NodalColorDataPrs
MeshVS_DMF_ElementalColorDataPrs
MeshVS_DMF_TextDataPrs
MeshVS_DMF_EntitiesWithData
```

The following methods provide selection and highlighting:

```
MeshVS_DMF_SelectionPrs
MeshVS_DMF_HilightPrs
```

MeshVS_DMF_User is a user-defined mode.

These values will be used by the presentation builder. There is also a set of selection modes flags that can be grouped in a combination of bits:

- *MeshVS_SMF_0D*
- *MeshVS_SMF_Link*
- *MeshVS_SMF_Face*
- *MeshVS_SMF_Volume*
- *MeshVS_SMF_Element* – groups *0D, Link, Face* and *Volume* as a bit mask;
- *MeshVS_SMF_Node*
- *MeshVS_SMF_All* – groups *Element* and *Node* as a bit mask;
- *MeshVS_SMF_Mesh*
- *MeshVS_SMF_Group*

Such an object, for example, can be used for displaying the object and stored in the STL file format:

```
// read the data and create a data source
Handle(Poly_Triangulation) aSTLMesh = RWStl::ReadFile (aFileName);
Handle(XSDRAWSTLVRML_DataSource) aDataSource = new XSDRAWSTLVRML_DataSource (aSTLMesh);

// create mesh
Handle(MeshVS_Mesh) aMeshPrs = new MeshVS();
aMeshPrs->SetDataSource (aDataSource);

// use default presentation builder
Handle(MeshVS_MeshPrsBuilder) aBuilder = new MeshVS_MeshPrsBuilder (aMeshPrs);
aMeshPrs->AddBuilder (aBuilder, true);
```

MeshVS_NodalColorPrsBuilder allows representing a mesh with a color scaled texture mapped on it. To do this you should define a color map for the color scale, pass this map to the presentation builder, and define an appropriate value in the range of 0.0 - 1.0 for every node. The following example demonstrates how you can do this (check if the view has been set up to display textures):

```
// assign nodal builder to the mesh
Handle(MeshVS_NodalColorPrsBuilder) aBuilder = new MeshVS_NodalColorPrsBuilder (theMeshPrs,
      MeshVS_DMF_NodalColorDataPrs | MeshVS_DMF_OCCMask);
aBuilder->UseTexture (true);

// prepare color map
Aspect_SequenceOfColor aColorMap;
aColorMap.Append (Quantity_NOC_RED);
aColorMap.Append (Quantity_NOC_BLUE1);

// assign color scale map  values (0..1) to nodes
TColStd_DataMapOfIntegerReal aScaleMap;
...
// iterate through the  nodes and add an node id and an appropriate value to the map
aScaleMap.Bind (anId, aValue);

// pass color map and color scale values to the builder
aBuilder->SetColorMap (aColorMap);
aBuilder->SetInvalidColor (Quantity_NOC_BLACK);
aBuilder->SetTextureCoords (aScaleMap);
aMesh->AddBuilder (aBuilder, true);
```

3.6 Dynamic Selection

The dynamic selection represents the topological shape, which you want to select, by decomposition of *sensitive primitives* – the sub-parts of the shape that will be detected and highlighted. The sets of these primitives are handled by the powerful three-level BVH tree selection algorithm.

For more details on the algorithm and examples of usage, refer to Selection chapter.

4 3D Presentations

4.1 Glossary of 3D terms

- **Group** – a set of primitives and attributes on those primitives. Primitives and attributes may be added to a group but cannot be removed from it, unless erased globally. A group can have a pick identity.
- **Light** There are five kinds of light source – ambient, headlight, directional, positional and spot.
- **Primitive** – a drawable element. It has a definition in 3D space. Primitives can either be lines, faces, text, or markers. Once displayed markers and text remain the same size. Lines and faces can be modified e.g. zoomed. Attributes are set within the group. Primitives must be stored in a group.
- **Structure** – manages a set of groups. The groups are mutually exclusive. A structure can be edited, adding or removing groups. A structure can reference other structures to form a hierarchy. It has a default (identity) transformation and other transformations may be applied to it (rotation, translation, scale, etc). Each structure has a display priority associated with it, which rules the order in which it is redrawn in a 3D viewer.
- **View** – is defined by a view orientation, a view mapping, and a context view.
- **Viewer** – manages a set of views.
- **View orientation** – defines the manner in which the observer looks at the scene in terms of View Reference Coordinates.
- **View mapping** – defines the transformation from View Reference Coordinates to the Normalized Projection Coordinates. This follows the Phigs scheme.
- **Z-Buffering** – a form of hidden surface removal in shading mode only. This is always active for a view in the shading mode and cannot be suppressed.

4.2 Graphic primitives

The *Graphic3d* package is used to create 3D graphic objects in a 3D viewer. These objects called **structures** are made up of groups of primitives, such as line segments, triangles, text and markers, and attributes, such as color, transparency, reflection, line type, line width, and text font. A group is the smallest editable element of a structure. A transformation can be applied to a structure. Structures can be connected to form a tree of structures, composed by transformations. Structures are globally manipulated by the viewer.

Graphic structures can be:

- Displayed,
- Highlighted,
- Erased,
- Transformed,
- Connected to form a tree hierarchy of structures, created by transformations.

There are classes for:

- Visual attributes for lines, faces, markers, text, materials,
- Vectors and vertices,
- Graphic objects, groups, and structures.

4.2.1 Structure hierarchies

The root is the top of a structure hierarchy or structure network. The attributes of a parent structure are passed to its descendants. The attributes of the descendant structures do not affect the parent. Recursive structure networks are not supported.

4.2.2 Graphic primitives

- **Markers**
  - Have one or more vertices,
  - Have a type, a scale factor, and a color,
  - Have a size, shape, and orientation independent of transformations.
- **Triangulation**
  - Has at least three vertices,
  - Has nodal normals defined for shading,
  - Has interior attributes – style, color, front and back material, texture and reflection ratio.
- **Polylines** or **Segments**
  - Have two or more vertices,
  - Have the following attributes – type, width scale factor, color.
- **Text**
  - Has geometric and non-geometric attributes,
  - Geometric attributes – character height, character up vector, text path, horizontal and vertical alignment, orientation, three-dimensional position, zoomable flag
  - Non-geometric attributes – text font, character spacing, character expansion factor, color.

4.2.3 Primitive arrays

The different types of primitives could be presented with the following primitive arrays:

- *Graphic3d_ArrayOfPoints,*
- *Graphic3d_ArrayOfPolylines,*
- *Graphic3d_ArrayOfSegments,*
- *Graphic3d_ArrayOfTriangleFans,*
- *Graphic3d_ArrayOfTriangles,*
- *Graphic3d_ArrayOfTriangleStrips.*

The *Graphic3d_ArrayOfPrimitives* is a base class for these primitive arrays. Method set *Graphic3d_ArrayOf↩ Primitives::AddVertex* allows adding vertices to the primitive array with their attributes (color, normal, texture coordinates). You can also modify the values assigned to the vertex or query these values by the vertex index.

The following example shows how to define an array of points:

```
// create an array
Handle(Graphic3d_ArrayOfPoints) anArray = new Graphic3d_ArrayOfPoints (theVerticiesMaxCount);

// add vertices to the array
anArray->AddVertex (10.0, 10.0, 10.0);
anArray->AddVertex (0.0,  10.0, 10.0);

// add the array to the structure
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();
aGroup->AddPrimitiveArray (anArray);
aGroup->SetGroupPrimitivesAspect (myDrawer->PointAspect()->Aspect());
```

If the primitives share the same vertices (polygons, triangles, etc.) then you can define them as indices of the vertices array. The method *Graphic3d_ArrayOfPrimitives::AddEdge* allows defining the primitives by indices. This method adds an "edge" in the range *[1, VertexNumber()]* in the array. It is also possible to query the vertex defined by an edge using method *Graphic3d_ArrayOfPrimitives::Edge*.

The following example shows how to define an array of triangles:

```
// create an array
Handle(Graphic3d_ArrayOfTriangles) anArray = new Graphic3d_ArrayOfTriangles (theVerticesMaxCount,
      theEdgesMaxCount, Graphic3d_ArrayFlags_None);
// add vertices to the array
anArray->AddVertex (-1.0, 0.0, 0.0); // vertex 1
anArray->AddVertex ( 1.0, 0.0, 0.0); // vertex 2
anArray->AddVertex ( 0.0, 1.0, 0.0); // vertex 3
anArray->AddVertex ( 0.0,-1.0, 0.0); // vertex 4

// add edges to the array
anArray->AddEdges (1, 2, 3); // first triangle
anArray->AddEdges (1, 2, 4); // second triangle

// add the array to the structure
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();
aGroup->AddPrimitiveArray (anArray);
aGroup->SetGroupPrimitivesAspect (myDrawer->ShadingAspect()->Aspect());
```

4.2.4 Text primitive

TKOpenGl toolkit renders text labels using texture fonts. *Graphic3d* text primitives have the following features:

- fixed size (non-zoomable) or zoomable,
- can be rotated to any angle in the view plane,
- support unicode charset.

The text attributes for the group could be defined with the *Graphic3d_AspectText3d* attributes group. To add any text to the graphic structure you can use the following methods:

```
void Graphic3d_Group::AddText (const Handle(Graphic3d_Text)& theTextParams,
                               const Standard_Boolean theToEvalMinMax);
```

You can pass FALSE as *theToEvalMinMax* if you do not want the Graphic3d structure boundaries to be affected by the text position.

Note that the text orientation angle can be defined by *Graphic3d_AspectText3d* attributes.

See the example:

```
// get the group
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();

// change the text aspect
Handle(Graphic3d_AspectText3d) aTextAspect = new Graphic3d_AspectText3d();
aTextAspect->SetTextZoomable (true);
aTextAspect->SetTextAngle (45.0);
aGroup->SetPrimitivesAspect (aTextAspect);

// add a text primitive to the structure
Handle(Graphic3d_Text) aText = new Graphic3d_Text (16.0f);
aText->SetText ("Text");
aText->SetPosition (gp_Pnt (1, 1, 1));
aGroup->AddText (aText);
```

4.2.5 Materials

A *Graphic3d_MaterialAspect* defines the following Common material properties:

- Transparency;

- Diffuse reflection – a component of the object color;
- Ambient reflection;
- Specular reflection – a component of the color of the light source.

The following items are required to determine the three colors of reflection:

- Color;
- Coefficient of diffuse reflection;
- Coefficient of ambient reflection;
- Coefficient of specular reflection.

Common material properties are used within Gouraud and Phong shading models (Graphic3d_TOSM_FACET, Graphic3d_TOSM_VERTEX and Graphic3d_TOSM_FRAGMENT). Within PBR shading model (Graphic3d_TOSM_PBR and Graphic3d_TOSM_PBR_FACET), material properties are defined by the following *Graphic3d_PBRMaterial* properties (Graphic3d_MaterialAspect::PBRMaterial()):

- Albedo (main color);
- Metallic factor;
- Roughness factor;
- Transparency;
- Index of refraction.

4.2.6 Textures

A *texture* is defined by a name. Three types of texture are available:

- 1D;
- 2D;
- Environment mapping.

4.2.7 Custom shaders

OCCT visualization core supports GLSL shaders. Custom shaders can be assigned to a generic presentation by its drawer attributes (Graphic3d aspects). To enable custom shader for a specific AIS_Shape in your application, the following API functions can be used:

```
// Create shader program
Handle(Graphic3d_ShaderProgram) aProgram = new Graphic3d_ShaderProgram();

// Attach vertex shader
aProgram->AttachShader (Graphic3d_ShaderObject::CreateFromFile (Graphic3d_TOS_VERTEX, "<Path to VS>"));

// Attach fragment shader
aProgram->AttachShader (Graphic3d_ShaderObject::CreateFromFile (Graphic3d_TOS_FRAGMENT, "<Path to FS>"));

// Set values for custom uniform variables (if they are)
aProgram->PushVariable ("MyColor", Graphic3d_Vec3 (0.0f, 1.0f, 0.0f));

// Set aspect property for specific AIS_Shape
theAISShape->Attributes()->ShadingAspect()->Aspect()->SetShaderProgram (aProgram);
```

4.3 Graphic attributes

4.3.1 Aspect package overview

The *Aspect* package provides classes for the graphic elements in the viewer:

- Groups of graphic attributes;
- Edges, lines, background;
- Window;
- Driver;
- Enumerations for many of the above.

4.4 3D view facilities

4.4.1 Overview

The *V3d* package provides the resources to define a 3D viewer and the views attached to this viewer (orthographic, perspective). This package provides the commands to manipulate the graphic scene of any 3D object visualized in a view on screen.

A set of high-level commands allows the separate manipulation of parameters and the result of a projection (Rotations, Zoom, Panning, etc.) as well as the visualization attributes (Mode, Lighting, Clipping, etc.) in any particular view.

The *V3d* package is basically a set of tools directed by commands from the viewer front-end. This tool set contains methods for creating and editing classes of the viewer such as:

- Default parameters of the viewer,
- Views (orthographic, perspective),
- Lighting (positional, directional, ambient, spot, headlight),
- Clipping planes,
- Instantiated sequences of views, planes, light sources, graphic structures, and picks,
- Various package methods.

4.4.2 A programming example

This sample TEST program for the *V3d* Package uses primary packages *Xw* and *Graphic3d* and secondary packages *Visual3d, Aspect, Quantity* and *math*.

```
// create a default display connection
Handle(Aspect_DisplayConnection) aDispConnection = new Aspect_DisplayConnection();
// create a Graphic Driver
Handle(OpenGl_GraphicDriver) aGraphicDriver = new OpenGl_GraphicDriver (aDispConnection);
// create a Viewer to this Driver
Handle(V3d_Viewer) aViewer = new V3d_Viewer (aGraphicDriver);
aViewer->SetDefaultBackgroundColor (Quantity_NOC_DARKVIOLET);
// Create a structure in this Viewer
Handle(Graphic3d_Structure) aStruct = new Graphic3d_Structure (aViewer->StructureManager());
aStruct->SetVisual (Graphic3d_TOS_SHADING); // Type of structure

// Create a group of primitives  in this structure
Handle(Graphic3d_Group) aPrsGroup = aStruct->NewGroup();

// Fill this group with one quad of size 100
Handle(Graphic3d_ArrayOfTriangleStrips) aTriangles = new Graphic3d_ArrayOfTriangleStrips (4);
aTriangles->AddVertex (-100./2., -100./2., 0.0);
aTriangles->AddVertex (-100./2.,  100./2., 0.0);
aTriangles->AddVertex ( 100./2., -100./2., 0.0);
```

```
aTriangles->AddVertex ( 100./2.,  100./2., 0.0);

Handle(Graphic3d_AspectFillArea3d) anAspects = new Graphic3d_AspectFillArea3d (Aspect_IS_SOLID,
      Quantity_NOC_RED,
                                                                                Quantity_NOC_RED,
      Aspect_TOL_SOLID, 1.0f,

      Graphic3d_NameOfMaterial_Gold, Graphic3d_NameOfMaterial_Gold);
aPrsGroup->SetGroupPrimitivesAspect (anAspects);
aPrsGroup->AddPrimitiveArray (aTriangles);

// Create Ambient and Infinite Lights in this Viewer
Handle(V3d_AmbientLight)     aLight1 = new V3d_AmbientLight (Quantity_NOC_GRAY50);
Handle(V3d_DirectionalLight) aLight2 = new V3d_DirectionalLight (V3d_Zneg, Quantity_NOC_WHITE, true);
aViewer->AddLight (aLight1);
aViewer->AddLight (aLight2);
aViewer->SetLightOn();

// Create a 3D quality  Window with the same DisplayConnection
Handle(Xw_Window) aWindow = new Xw_Window (aDispConnection, "Test V3d", 100, 100, 500, 500);
aWindow->Map(); // Map this Window to this screen

// Create a Perspective  View in this Viewer
Handle(V3d_View) aView = new V3d_View (aViewer);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Perspective);
// Associate this View with the Window
aView->SetWindow (aWindow);
// Display presentation in this View
aStruct->Display();
// Finally update the Visualization in this View
aView->Update();
// Fit view to object size
aView->FitAll();
```

4.4.3 Define viewing parameters

View projection and orientation in OCCT *V3d_View* are driven by camera. The camera calculates and supplies projection and view orientation matrices for rendering by OpenGL. The allows to the user to control all projection parameters. The camera is defined by the following properties:

- **Eye** – defines the observer (camera) position. Make sure the Eye point never gets between the Front and Back clipping planes.
- **Center** – defines the origin of View Reference Coordinates (where camera is aimed at).
- **Direction** – defines the direction of camera view (from the Eye to the Center).
- **Distance** – defines the distance between the Eye and the Center.
- **Front** Plane – defines the position of the front clipping plane in View Reference Coordinates system.
- **Back** Plane – defines the position of the back clipping plane in View Reference Coordinates system.
- **ZNear** – defines the distance between the Eye and the Front plane.
- **ZFar** – defines the distance between the Eye and the Back plane.

Most common view manipulations (panning, zooming, rotation) are implemented as convenience methods of *V3d_View* class or by *AIS_ViewController* tool. However *Graphic3d_Camera* class can also be used directly by application developers. Example:

```
// rotate camera by X axis on 30.0 degrees
gp_Trsf aTrsf;
aTrsf.SetRotation (gp_Ax1 (gp_Pnt (0.0, 0.0, 0.0), gp_Dir (1.0, 0.0, 0.0)), M_PI / 4.0);
aView->Camera()->Transform (aTrsf);
```

4.4.4 Orthographic Projection

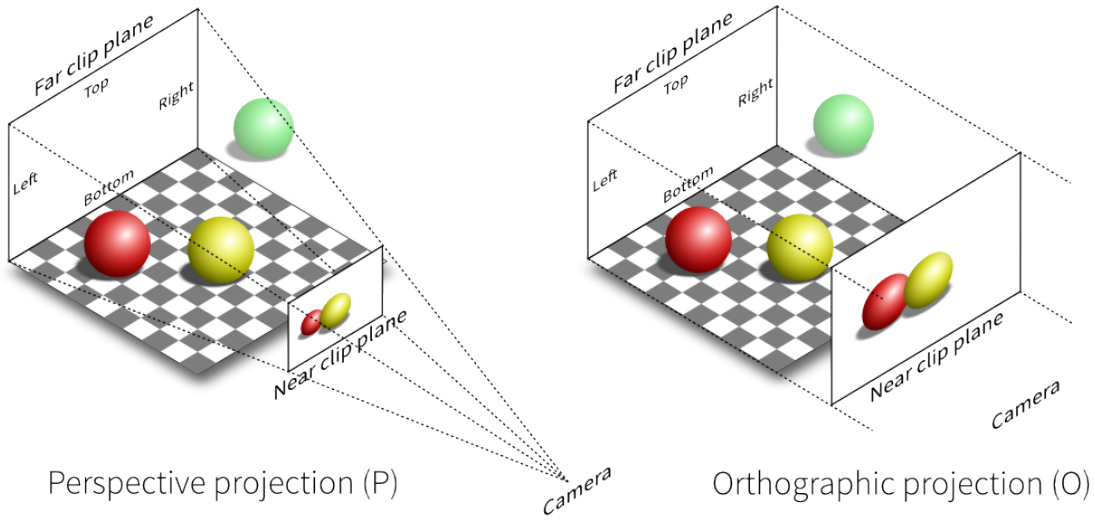


Figure 10: Perspective and orthographic projection

The following code configures the camera for orthographic rendering:

```
// Create an orthographic View in this Viewer
Handle(V3d_View) aView = new V3d_View (theViewer);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Orthographic);
aView->Update(); // update the Visualization in this View
```

4.4.5 Perspective Projection

Field of view (FOVy) – defines the field of camera view by y axis in degrees (45° is default).

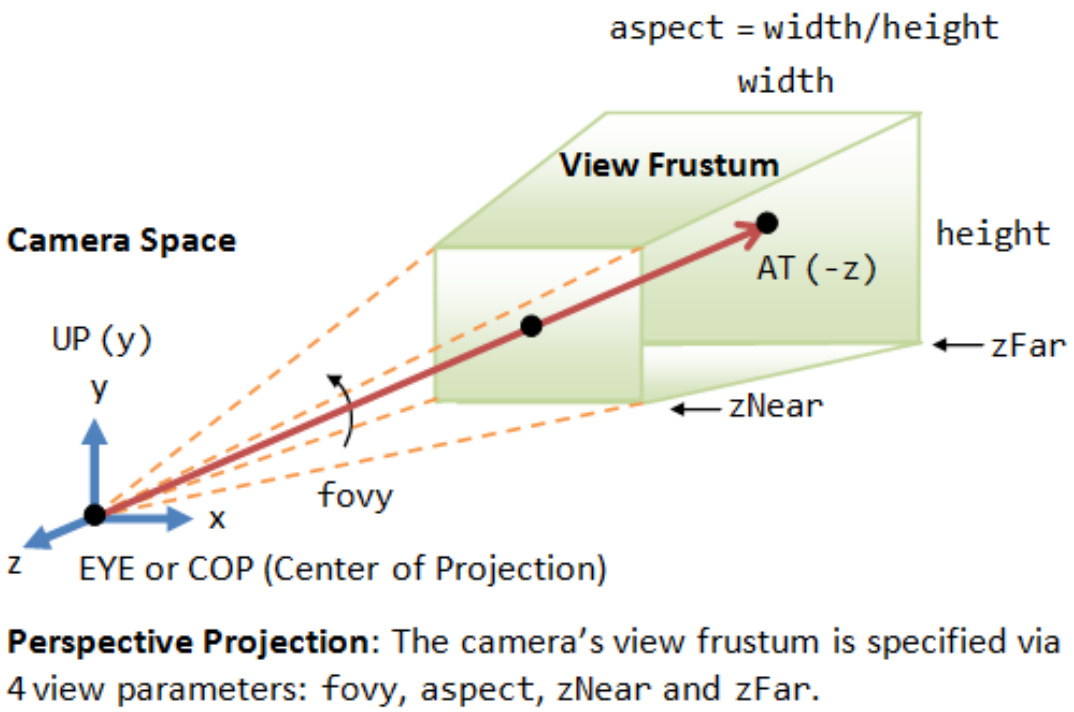


Figure 11: Perspective frustum

The following code configures the camera for perspective rendering:

```
// Create a perspective View in this Viewer
Handle(V3d_View) aView = new V3d_View (theViewer);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Perspective);
aView->Update();
```

4.4.6 Stereographic Projection

IOD – defines the intraocular distance (in world space units).

There are two types of IOD:

- *Graphic3d_Camera::IODType_Absolute* : Intraocular distance is defined as an absolute value.
- *Graphic3d_Camera::IODType_Relative* : Intraocular distance is defined relative to the camera focal length (as its coefficient).

Field of view (FOV) – defines the field of camera view by y axis in degrees (45° is default).

ZFocus – defines the distance to the point of stereographic focus.

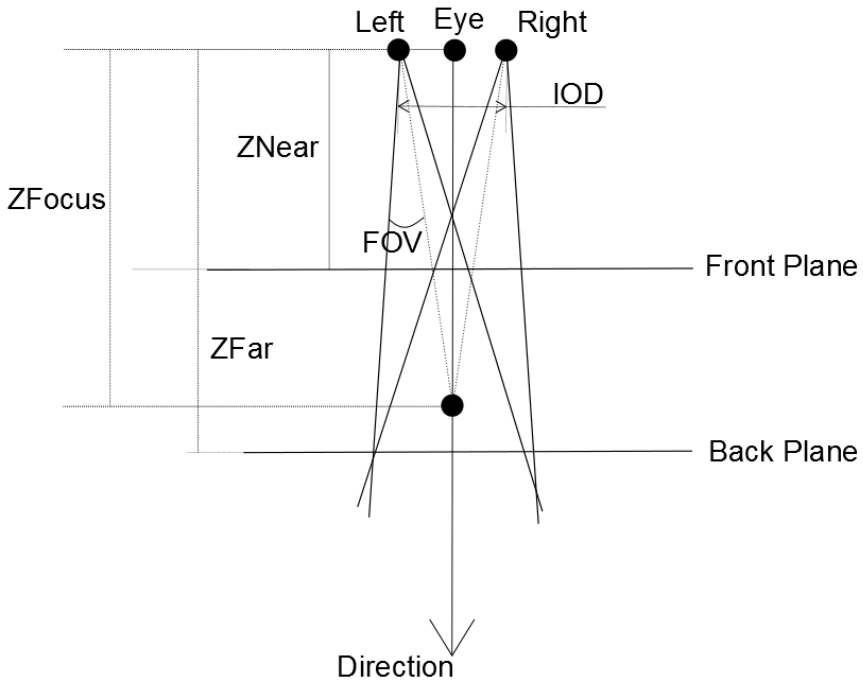


Figure 12: Stereographic projection

To enable stereo projection for active (shutter) 3D glasses, your workstation should meet the following requirements:

- The graphic card should support quad buffering.
- You need active 3D glasses (LCD shutter glasses).
- The graphic driver needs to be configured to impose quad buffering for newly created OpenGl contexts; the viewer and the view should be created after that.

In stereographic projection mode the camera prepares two projection matrices to display different stereo-pictures for the left and for the right eye. In a non-stereo camera this effect is not visible because only the same projection is used for both eyes.

To enable quad buffering support you should provide the following settings to the graphic driver *OpenGl_Caps*:

```
Handle(OpenGl_GraphicDriver) aDriver = new OpenGl_GraphicDriver();
OpenGl_Caps& aCaps = aDriver->ChangeOptions();
aCaps.contextStereo = Standard_True;
```

The following code configures the camera for stereographic rendering:

```
// Create a Stereographic View in this Viewer
Handle(V3d_View) aView = new V3d_View (theViewer);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Stereo);
// Change stereo parameters
aView->Camera()->SetIOD (IODType_Absolute, 5.0);
// Finally update the Visualization in this View
aView->Update();
```

Other 3D displays are also supported, including row-interlaced with passive glasses and anaglyph glasses - see *Graphic3d_StereoMode* enumeration. Example to activate another stereoscopic display:

```
Handle(V3d_View) theView;
theView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Stereo);
theView->ChangeRenderingParams().StereoParams = Graphic3d_StereoMode_RowInterlaced;
```

Supporting of VR/AR headsets in application is more involving. Class *Aspect_XRSession* defines a basic interface for working with extended reality.

4.4.7 View frustum culling

The algorithm of frustum culling on CPU-side is activated by default for 3D viewer. This algorithm allows skipping the presentation outside camera at the rendering stage, providing better performance. The following features support this method:

- *Graphic3d_Structure::CalculateBoundBox()* is used to calculate axis-aligned bounding box of a presentation considering its transformation.
- *V3d_View::SetFrustumCulling* enables or disables frustum culling for the specified view.
- Classes *Graphic3d_BvhCStructureSet* and *Graphic3d_CullingTool* handle the detection of outer objects and usage of acceleration structure for frustum culling.
- *BVH_BinnedBuilder* class splits several objects with null bounding box.

4.4.8 View background styles

There are several types of background styles available for *V3d_View*: solid color, gradient color, image and environment cubemap.

To set solid color for the background you can use the following method:

```
void V3d_View::SetBackgroundColor (const Quantity_Color& theColor);
```

The gradient background style could be set up with the following method:

```
void V3d_View::SetBgGradientColors (const Quantity_Color& theColor1,
                                    const Quantity_Color& theColor2,
                                    const Aspect_GradientFillMethod theFillStyle,
                                    const Standard_Boolean theToUpdate = false);
```

The *theColor1* and *theColor2* parameters define the boundary colors of interpolation, the *theFillStyle* parameter defines the direction of interpolation.

To set the image as a background and change the background image style you can use the following method:

```
void V3d_View::SetBackgroundImage (const Standard_CString theFileName,
                                   const Aspect_FillMethod theFillStyle,
                                   const Standard_Boolean theToUpdate = false);
```

The *theFileName* parameter defines the image file name and the path to it, the *theFillStyle* parameter defines the method of filling the background with the image. The methods are:

- *Aspect_FM_NONE* – draws the image in the default position;
- *Aspect_FM_CENTERED* – draws the image at the center of the view;
- *Aspect_FM_TILED* – tiles the view with the image;
- *Aspect_FM_STRETCH* – stretches the image over the view.

4.4.9 Dumping a 3D scene into an image file

The 3D scene displayed in the view can be dumped into image file with resolution independent from window size (using offscreen buffer). The *V3d_View* has the following methods for dumping the 3D scene:

```
Standard_Boolean V3d_View::Dump (const Standard_CString theFile,
                                 const Image_TypeOfImage theBufferType);
```

Dumps the scene into an image file with the view dimensions. The raster image data handling algorithm is based on the *Image_AlienPixMap* class. The supported extensions are ".png", ".bmp", ".jpg" and others supported by **FreeImage** library. The value passed as *theBufferType* argument defines the type of the buffer for an output image (RGB, RGBA, floating-point, RGBF, RGBAF). Method returns TRUE if the scene has been successfully dumped.

```
Standard_Boolean V3d_View::ToPixMap (Image_PixMap&               theImage,
                                     const V3d_ImageDumpOptions& theParams);
```

Dumps the displayed 3d scene into a pixmap with a width and height passed through parameters structure *theParams*.

4.4.10 Ray tracing support

OCCT visualization provides rendering by real-time ray tracing technique. It is allowed to switch easily between usual rasterization and ray tracing rendering modes. The core of OCCT ray tracing is written using GLSL shaders. The ray tracing has a wide list of features:

- Hard shadows
- Refractions
- Reflection
- Transparency
- Texturing
- Support of non-polygon objects, such as lines, text, highlighting, selection.
- Performance optimization using 2-level bounding volume hierarchy (BVH).

The ray tracing algorithm is recursive (Whitted's algorithm). It uses BVH effective optimization structure. The structure prepares optimized data for a scene geometry for further displaying it in real-time. The time-consuming re-computation of the BVH is not necessary for view operations, selections, animation and even editing of the scene by transforming location of the objects. It is only necessary when the list of displayed objects or their geometry changes. To make the BVH reusable it has been added into an individual reusable OCCT package *TKMath/BVH*.

There are several ray-tracing options that user can switch on/off:

- Maximum ray tracing depth
- Shadows rendering
- Specular reflections
- Adaptive anti aliasing
- Transparency shadow effects

Example:

```
Graphic3d_RenderingParams& aParams = aView->ChangeRenderingParams();
// specifies rendering mode
aParams.Method = Graphic3d_RM_RAYTRACING;
// maximum ray-tracing depth
aParams.RaytracingDepth = 3;
// enable shadows rendering
aParams.IsShadowEnabled = true;
// enable specular reflections
aParams.IsReflectionEnabled = true;
// enable adaptive anti-aliasing
aParams.IsAntialiasingEnabled = true;
// enable light propagation through transparent media
aParams.IsTransparentShadowEnabled = true;
// update the view
aView->Update();
```

4.4.11 Display priorities

Structure display priorities control the order, in which structures are drawn. When you display a structure you specify its priority. The lower is the value, the lower is the display priority. When the display is regenerated, the structures with the lowest priority are drawn first. The structures with the same display priority are drawn in the same order as they have been displayed. OCCT supports eleven structure display priorities within [0, 10] range.

4.4.12 Z-layer support

OCCT features depth-arranging functionality called z-layer. A graphical presentation can be put into a z-layer. In general, this function can be used for implementing "bring to front" functionality in a graphical application.

Example:

```
// set z-layer to an interactive object
Handle(AIS_InteractiveContext) theContext;
Handle(AIS_InteractiveObject) theInterObj;
Standard_Integer anId = -1;
aViewer->AddZLayer (anId);
theContext->SetZLayer (theInterObj, anId);
```

For each z-layer, it is allowed to:

- Enable / disable depth test for layer.
- Enable / disable depth write for layer.
- Enable / disable depth buffer clearing.
- Enable / disable polygon offset.

You can get the options using getter from *V3d_Viewer*. It returns *Graphic3d_ZLayerSettings* for a given *LayerId*.

Example:

```
// change z-layer settings
Graphic3d_ZLayerSettings aSettings = aViewer->ZLayerSettings (anId);
aSettings.SetEnableDepthTest (true);
aSettings.SetEnableDepthWrite(true);
aSettings.SetClearDepth      (true);
aSettings.SetPolygonOffset   (Graphic3d_PolygonOffset());
aViewer->SetZLayerSettings (anId, aSettings);
```

Another application for Z-Layer feature is treating visual precision issues when displaying objects far from the World Center. The key problem with such objects is that visualization data is stored and manipulated with single precision floating-point numbers (32-bit). Single precision 32-bit floating-point numbers give only 6-9 significant decimal digits precision, while double precision 64-bit numbers give 15-17 significant decimal digits precision, which is sufficient enough for most applications.

When moving an Object far from the World Center, float number steadily eats precision. The camera Eye position adds leading decimal digits to the overall Object transformation, which discards smaller digits due to floating point

number nature. For example, the object of size 0.0000123 moved to position 1000 has result transformation 1000.↩ 0000123, which overflows single precision floating point - considering the most optimistic scenario of 9 significant digits (but it is really not this case), the result number will be 1000.00001.

This imprecision results in visual artifacts of two kinds in the 3D Viewer:

- Overall per-vertex Object distortion. This happens when each vertex position has been defined within World Coordinate system.
- The object itself is not distorted, but its position in the World is unstable and imprecise - the object jumps during camera manipulations. This happens when vertices have been defined within Local Coordinate system at the distance small enough to keep precision within single precision float, however Local Transformation applied to the Object is corrupted due to single precision float.

The first issue cannot be handled without switching the entire presentation into double precision (for each vertex position). However, visualization hardware is much faster using single precision float number rather than double precision - so this is not an option in most cases. The second issue, however, can be negated by applying special rendering tricks.

So, to apply this feature in OCCT, the application:

- Defines Local Transformation for each object to fit the presentation data into single precision float without distortion.
- Spatially splits the world into smaller areas/cells where single precision float will be sufficient. The size of such cell might vary and depends on the precision required by application (e.g. how much user is able to zoom in camera within application).
- Defines a Z-Layer for each spatial cell containing any object.
- Defines the Local Origin property of the Z-Layer according to the center of the cell.

```
Graphic3d_ZLayerSettings aSettings = aViewer->ZLayerSettings (anId);
aSettings.SetLocalOrigin (400.0, 0.0, 0.0);
```

- Assigns a presentable object to the nearest Z-Layer.

Note that Local Origin of the Layer is used only for rendering - everything outside will be still defined in the World Coordinate System, including Local Transformation of the Object and Detection results. E.g., while moving the presentation between Z-layers with different Local Origins, the Object will stay at the same place - only visualization quality will vary.

4.4.13 Clipping planes

The ability to define custom clipping planes could be very useful for some tasks. OCCT provides such an opportunity.

The *Graphic3d_ClipPlane* class provides the services for clipping planes: it holds the plane equation coefficients and provides its graphical representation. To set and get plane equation coefficients you can use the following methods:

```
Graphic3d_ClipPlane::Graphic3d_ClipPlane (const gp_Pln& thePlane)
void Graphic3d_ClipPlane::SetEquation (const gp_Pln& thePlane)
Graphic3d_ClipPlane::Graphic3d_ClipPlane (const Equation& theEquation)
void Graphic3d_ClipPlane::SetEquation (const Equation& theEquation)
gp_Pln Graphic3d_ClipPlane::ToPlane() const
```

The clipping planes can be activated with the following method:

```
void Graphic3d_ClipPlane::SetOn (const Standard_Boolean theIsOn)
```

The number of clipping planes is limited. You can check the limit value via method *Graphic3d_GraphicDriver::*↩ *InquireLimit()*;

```
// get the limit of clipping planes for the current view
Standard_Integer aMaxClipPlanes = aView->Viewer()->Driver()->InquireLimit (
      Graphic3d_TypeOfLimit_MaxNbClipPlanes);
```

Let us see for example how to create a new clipping plane with custom parameters and add it to a view or to an object:

```
// create a new clipping plane
Handle(Graphic3d_ClipPlane) aClipPlane = new Graphic3d_ClipPlane();
// change equation of the clipping plane
Standard_Real aCoeffA, aCoeffB, aCoeffC, aCoeffD = ...
aClipPlane->SetEquation (gp_Pln (aCoeffA, aCoeffB, aCoeffC, aCoeffD));
// set capping
aClipPlane->SetCapping (aCappingArg == "on");
// set the material with red color of clipping plane
Graphic3d_MaterialAspect aMat = aClipPlane->CappingMaterial();
Quantity_Color aColor (1.0, 0.0, 0.0, Quantity_TOC_RGB);
aMat.SetAmbientColor (aColor);
aMat.SetDiffuseColor (aColor);
aClipPlane->SetCappingMaterial (aMat);
// set the texture of clipping plane
Handle(Graphic3d_Texture2Dmanual) aTexture = ...
aTexture->EnableModulate();
aTexture->EnableRepeat();
aClipPlane->SetCappingTexture (aTexture);
// add the clipping plane to an interactive object
Handle(AIS_InteractiveObject) aIObj = ...
aIObj->AddClipPlane (aClipPlane);
// or to the whole view
aView->AddClipPlane (aClipPlane);
// activate the clipping plane
aClipPlane->SetOn (Standard_True);
// update the view
aView->Update();
```

4.4.14 Automatic back face culling

Back face culling reduces the rendered number of triangles (which improves the performance) and eliminates artifacts at shape boundaries. However, this option can be used only for solid objects, where the interior is actually invisible from any point of view. Automatic back-face culling mechanism is turned on by default, which is controlled by *V3d_View::SetBackFacingModel()*.

The following features are applied in *StdPrs_ToolTriangulatedShape::IsClosed()*, which is used for definition of back face culling in *ShadingAspect*:

- disable culling for free closed Shells (not inside the Solid) since reversed orientation of a free Shell is a valid case;
- enable culling for Solids packed into a compound;
- ignore Solids with incomplete triangulation.

Back face culling is turned off at TKOpenGl level in the following cases:

- clipping/capping planes are in effect;
- for translucent objects;
- with hatching presentation style.

4.5 Examples: creating a 3D scene

To create 3D graphic objects and display them in the screen, follow the procedure below:

1. Create attributes.
2. Create a 3D viewer.

3. Create a view.
4. Create an interactive context.
5. Create interactive objects.
6. Create primitives in the interactive object.
7. Display the interactive object.

4.5.1 Create attributes

Create colors.

```
Quantity_Color aBlack (Quantity_NOC_BLACK);
Quantity_Color aBlue (Quantity_NOC_MATRABLUE);
Quantity_Color aBrown (Quantity_NOC_BROWN4);
Quantity_Color aFirebrick (Quantity_NOC_FIREBRICK);
Quantity_Color aForest (Quantity_NOC_FORESTGREEN);
Quantity_Color aGray (Quantity_NOC_GRAY70);
Quantity_Color aMyColor (0.99, 0.65, 0.31, Quantity_TOC_RGB);
Quantity_Color aBeet (Quantity_NOC_BEET);
Quantity_Color aWhite (Quantity_NOC_WHITE);
```

Create line attributes.

```
Handle(Graphic3d_AspectLine3d) anAspectBrown = new Graphic3d_AspectLine3d();
Handle(Graphic3d_AspectLine3d) anAspectBlue  = new Graphic3d_AspectLine3d();
Handle(Graphic3d_AspectLine3d) anAspectWhite = new Graphic3d_AspectLine3d();
anAspectBrown->SetColor (aBrown);
anAspectBlue ->SetColor (aBlue);
anAspectWhite->SetColor (aWhite);
```

Create marker attributes.

```
Handle(Graphic3d_AspectMarker3d aFirebrickMarker = new Graphic3d_AspectMarker3d();
// marker attributes
aFirebrickMarker->SetColor (Firebrick);
aFirebrickMarker->SetScale (1.0f);
aFirebrickMarker->SetType (Aspect_TOM_BALL);
// or custom image
aFirebrickMarker->SetMarkerImage (theImage)
```

Create facet attributes.

```
Handle(Graphic3d_AspectFillArea3d) aFaceAspect = new Graphic3d_AspectFillArea3d();
Graphic3d_MaterialAspect aBrassMaterial (Graphic3d_NameOfMaterial_Brass);
Graphic3d_MaterialAspect aGoldMaterial  (Graphic3d_NameOfMaterial_Gold);
aFaceAspect->SetInteriorStyle (Aspect_IS_SOLID_WIREFRAME);
aFaceAspect->SetInteriorColor (aMyColor);
aFaceAspect->SetDistinguishOn ();
aFaceAspect->SetFrontMaterial (aGoldMaterial);
aFaceAspect->SetBackMaterial  (aBrassMaterial);
```

Create text attributes.

```
Handle(Graphic3d_AspectText3d) aTextAspect = new Graphic3d_AspectText3d (aForest, Font_NOF_MONOSPACE, 1.0,
      0.0);
```

4.5.2 Create a 3D Viewer (a Windows example)

```
// create a graphic driver
Handle(OpenGl_GraphicDriver) aGraphicDriver = new OpenGl_GraphicDriver (Handle(Aspect_DisplayConnection)())
      ;
// create a viewer
myViewer = new V3d_Viewer (aGraphicDriver);
// set parameters for V3d_Viewer
// defines default lights -
//   positional-light 0.3 0.0 0.0
//   directional-light V3d_XnegYposZpos
```

```
//   directional-light V3d_XnegYneg
//   ambient-light
a3DViewer->SetDefaultLights();
// activates all the lights defined in this viewer
a3DViewer->SetLightOn();
// set background color to black
a3DViewer->SetDefaultBackgroundColor (Quantity_NOC_BLACK);
```

4.5.3 Create a 3D view (a Windows example)

It is assumed that a valid Windows window may already be accessed via the method *GetSafeHwnd()* (as in case of MFC sample).

```
Handle(WNT_Window) aWNTWindow = new WNT_Window (GetSafeHwnd());
myView = myViewer->CreateView();
myView->SetWindow (aWNTWindow);
```

4.5.4 Create an interactive context

```
myAISContext = new AIS_InteractiveContext (myViewer);
```

You are now able to display interactive objects such as an *AIS_Shape*.

```
TopoDS_Shape aShape = BRepAPI_MakeBox (10, 20, 30).Solid();
Handle(AIS_Shape) anAISShape = new AIS_Shape (aShape);
myAISContext->Display (anAISShape, true);
```

4.5.5 Create your own interactive object

Follow the procedure below to compute the presentable object:

1. Build a presentable object inheriting from *AIS_InteractiveObject* (refer to the Chapter on Presentable Objects).
2. Reuse the *Graphic3d_Structure* provided as an argument of the compute methods.

Note that there are two compute methods: one for a standard representation, and the other for a degenerated representation, i.e. in hidden line removal and wireframe modes.

Let us look at the example of compute methods

```
void MyPresentableObject::Compute (const Handle(PrsMgr_PresentationManager3d)& thePrsManager,
                                   const Handle(Graphic3d_Structure)& thePrs,
                                   const Standard_Integer theMode)
(
  //...
)

void MyPresentableObject::Compute (const Handle(Prs3d_Projector)& theProjector,
                                   const Handle(Graphic3d_Structure)& thePrs)
(
  //...
)
```

4.5.6 Create primitives in the interactive object

Get the group used in *Graphic3d_Structure*.

```
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();
```

Update the group attributes.

```
aGroup->SetGroupPrimitivesAspect (anAspectBlue);
```

Create two triangles in *aGroup*.

```
Standard_Integer aNbTria = 2;
Handle(Graphic3d_ArrayOfTriangles) aTriangles = new Graphic3d_ArrayOfTriangles (3 * aNbTria, 0,
      Graphic3d_ArrayFlags_VertexNormal);
for (Standard_Integer aTriIter = 1; aTriIter <= aNbTria; ++aTriIter)
{
  aTriangles->AddVertex (aTriIter * 5.,      0., 0., 1., 1., 1.);
  aTriangles->AddVertex (aTriIter * 5 + 5,   0., 0., 1., 1., 1.);
  aTriangles->AddVertex (aTriIter * 5 + 2.5, 5., 0., 1., 1., 1.);
}
aGroup->AddPrimitiveArray (aTriangles);
aGroup->SetGroupPrimitivesAspect (new Graphic3d_AspectFillArea3d());
```

Use the polyline function to create a boundary box for the *thePrs* structure in group *aGroup*.

```
Standard_Real Xm, Ym, Zm, XM, YM, ZM;
thePrs->MinMaxValues (Xm, Ym, Zm, XM, YM, ZM);

Handle(Graphic3d_ArrayOfPolylines) aPolylines = new Graphic3d_ArrayOfPolylines (16, 4);
aPolylines->AddBound (4);
aPolylines->AddVertex (Xm,  Ym, Zm);
aPolylines->AddVertex (Xm,  Ym, ZM);
aPolylines->AddVertex (Xm,  YM, ZM);
aPolylines->AddVertex (Xm,  YM, Zm);
aPolylines->AddBound (4);
aPolylines->AddVertex (Xm,  Ym, Zm);
aPolylines->AddVertex (XM,  Ym, Zm);
aPolylines->AddVertex (XM,  Ym, ZM);
aPolylines->AddVertex (XM,  YM, ZM);
aPolylines->AddBound (4);
aPolylines->AddVertex (XM,  YM, Zm);
aPolylines->AddVertex (XM,  Ym, Zm);
aPolylines->AddVertex (XM,  YM, Zm);
aPolylines->AddVertex (Xm,  YM, Zm);
aPolylines->AddBound (4);
aPolylines->AddVertex (Xm,  YM, ZM);
aPolylines->AddVertex (XM,  YM, ZM);
aPolylines->AddVertex (XM,  Ym, ZM);
aPolylines->AddVertex (Xm,  Ym, ZM);

aGroup->AddPrimitiveArray(aPolylines);
aGroup->SetGroupPrimitivesAspect (new Graphic3d_AspectLine3d());
```

Create text and markers in group *aGroup*.

```
static char* THE_TEXT[3] =
{
  "Application title",
  "My company",
  "My company address."
};
Handle(Graphic3d_ArrayOfPoints) aPtsArr = new Graphic3d_ArrayOfPoints (2, 1);
aPtsArr->AddVertex (-40.0, -40.0, -40.0);
aPtsArr->AddVertex (40.0, 40.0, 40.0);
aGroup->AddPrimitiveArray (aPtsArr);
aGroup->SetGroupPrimitivesAspect (new Graphic3d_AspectText3d());

Graphic3d_Vertex aMarker (0.0, 0.0, 0.0);
for (int i = 0; i <= 2; i++)
{
  aMarker.SetCoord (-(Standard_Real )i * 4 + 30,
                     (Standard_Real )i * 4,
                    -(Standard_Real )i * 4);
  aGroup->Text (THE_TEXT[i], Marker, 20.);
}
```

5 Mesh Visualization Services

MeshVS (Mesh Visualization Service) component extends 3D visualization capabilities of Open CASCADE Technology. It provides flexible means of displaying meshes along with associated pre- and post-processor data.

From a developer's point of view, it is easy to integrate the *MeshVS* component into any mesh-related application with the following guidelines:

- Derive a data source class from the *MeshVS_DataSource* class.
- Re-implement its virtual methods, so as to give the *MeshVS* component access to the application data model. This is the most important part of the job, since visualization performance is affected by performance of data retrieval methods of your data source class.
- Create an instance of *MeshVS_Mesh* class.
- Create an instance of your data source class and pass it to a *MeshVS_Mesh* object through the *SetData↩ Source()* method.
- Create one or several objects of *MeshVS_PrsBuilder*-derived classes (standard, included in the *MeshVS* package, or your custom ones).
- Each *PrsBuilder* is responsible for drawing a *MeshVS_Mesh* presentation in a certain display mode(s) specified as a *PrsBuilder* constructor's argument. Display mode is treated by *MeshVS* classes as a combination of bit flags (two least significant bits are used to encode standard display modes: wireframe, shading and shrink).
- Pass these objects to the *MeshVS_Mesh::AddBuilder()* method. *MeshVS_Mesh* takes advantage of improved selection highlighting mechanism: it highlights its selected entities itself, with the help of so called "highlighter" object. You can set one of *PrsBuilder* objects to act as a highlighter with the help of a corresponding argument of the *AddBuilder()* method.

Visual attributes of the *MeshVS_Mesh* object (such as shading color, shrink coefficient and so on) are controlled through *MeshVS_Drawer* object. It maintains a map "Attribute ID --> attribute value" and can be easily extended with any number of custom attributes.

In all other respects, *MeshVS_Mesh* is very similar to any other class derived from *AIS_InteractiveObject* and it should be used accordingly (refer to the description of *AIS package* in the documentation).