



Open CASCADE Technology
6.9.0

Visualization

May 8, 2015

Contents

1	Introduction	1
2	Fundamental Concepts	3
2.1	Presentation	3
2.1.1	Structure of the Presentation	3
2.1.2	Presentation packages	3
2.1.3	A Basic Example: How to display a 3D object	4
2.2	Selection	6
2.2.1	The Sensitive Primitive	8
2.2.2	Dynamic Selection	8
2.2.3	Selection Packages	9
2.2.4	How to use dynamic selection	12
3	Application Interactive Services	16
3.1	Introduction	16
3.2	Interactive objects	17
3.2.1	Presentations	17
3.2.2	Hidden Line Removal	17
3.2.3	Presentation modes	18
3.2.4	Selection	19
3.2.5	Graphic attributes	20
3.2.6	Complementary Services	21
3.3	Interactive Context	23
3.3.1	Rules	23
3.3.2	Groups of functions	23
3.3.3	Management of the Interactive Context	24
3.4	Local Context	24
3.4.1	Rules and Conventions	24
3.4.2	Management of Local Context	25
3.4.3	Presentation in a Neutral Point	25
3.4.4	Presentation in the Local Context	26
3.4.5	Filters	27
3.4.6	Selection in the Local Context	28
3.4.7	Recommendations	30
3.5	Standard Interactive Object Classes	33
3.5.1	Datum	33
3.5.2	Object	34
3.5.3	Relations	35
3.5.4	Dimensions	35

3.5.5	MeshVS_Mesh	36
3.6	Dynamic Selection	37
3.6.1	How to go from the objects to 2D boxes	37
3.6.2	Implementation in an interactive/selectable object	38
4	3D Presentations	40
4.1	Glossary of 3D terms	40
4.2	Graphic primitives	40
4.2.1	Structure hierarchies	41
4.2.2	Graphic primitives	41
4.2.3	Primitive arrays	42
4.2.4	Text primitive	44
4.2.5	Materials	45
4.2.6	Textures	46
4.2.7	Shaders	46
4.3	Graphic attributes	46
4.3.1	Aspect package overview	46
4.4	3D view facilities	47
4.4.1	Overview	47
4.4.2	A programming example	47
4.4.3	Define viewing parameters	48
4.4.4	Orthographic Projection	48
4.4.5	Perspective Projection	49
4.4.6	Stereographic Projection	49
4.4.7	View frustum culling	49
4.4.8	Underlay and overlay layers management	50
4.4.9	View background styles	51
4.4.10	Dumping a 3D scene into an image file	52
4.4.11	Printing a 3D scene	53
4.4.12	Vector image export	54
4.4.13	Ray tracing support	54
4.4.14	Display priorities	55
4.4.15	Z-layer support	55
4.4.16	Clipping planes	56
4.4.17	Automatic back face culling	57
4.5	Examples: creating a 3D scene	57
4.5.1	Create attributes	57
4.5.2	Create a 3D Viewer (a Windows example)	58
4.5.3	Create a 3D view (a Windows example)	58
4.5.4	Create an interactive context	59

4.5.5	Create your own interactive object	59
4.5.6	Create primitives in the interactive object	59
5	Mesh Visualization Services	61

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.

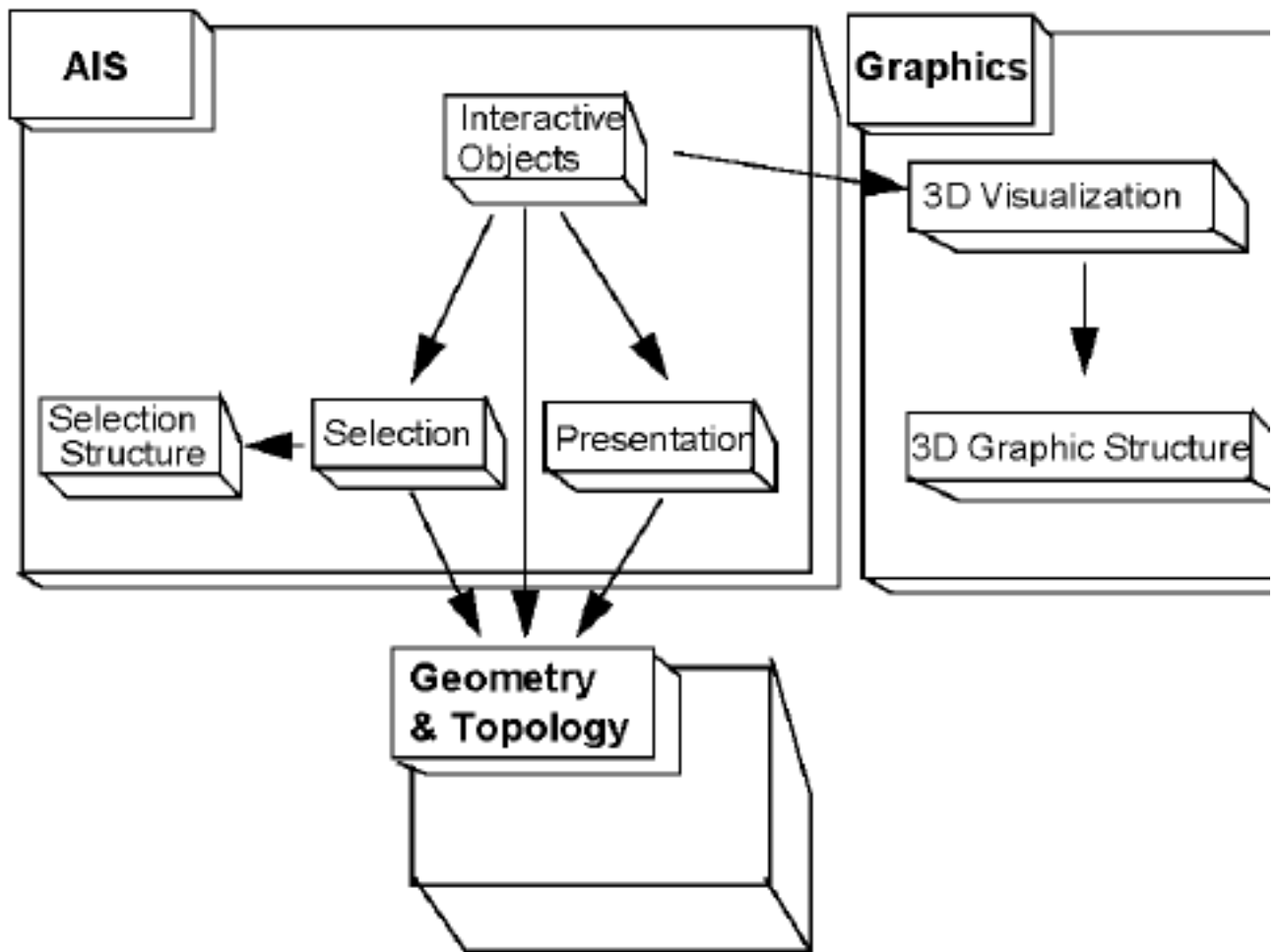


Figure 1: Key concepts and packages in visualization

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 visualisation.

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.

- *AIS* package provides all classes to implement interactive objects (presentable and selectable entities).
- *PrsMgr* package provides low level services and is only to be used when you do not want to use the services provided by *AIS*. It contains all classes needed to implement the presentation process: abstract classes *Presentation* and *PresentableObject* and concrete class *PresentationManager3d*.
- *StdPrs* package provides ready-to-use standard presentation algorithms for specific geometries: points, curves and shapes of the geometry and topology toolkits.

- *Prs3d* package provides generic presentation algorithms such as wireframe, shading and hidden line removal associated with a *Drawer* class, which controls the attributes of the presentation to be created in terms of color, line type, thickness, etc.
- *V3d* package provides the services supported by the 3D viewer.
- *Graphic3d* package provides resources to create 3D graphic structures.
- *Visual3d* package contains classes implementing commands for 3D viewer.
- *DsgPrs* package provides tools for display of dimensions, relations and XYZ trihedrons.

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

```
Void Standard_Real dx = ...; //Parameters
Void Standard_Real dy = ...; //to build a wedge
Void Standard_Real dz = ...;
Void Standard_Real ltx = ...;

Handle(V3d_Viewer)aViewer = ...;
Handle(AIS_InteractiveContext)aContext;
aContext = new AIS_InteractiveContext(aViewer);

BRepPrimAPI_MakeWedge w(dx, dy, dz, ltx);
TopoDS_Solid & = w.Solid();
Handle(AIS_Shape) anAis = new AIS_Shape(S);
//creation of the presentable object
aContext -> Display(anAis);
//Display the presentable object in the 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.

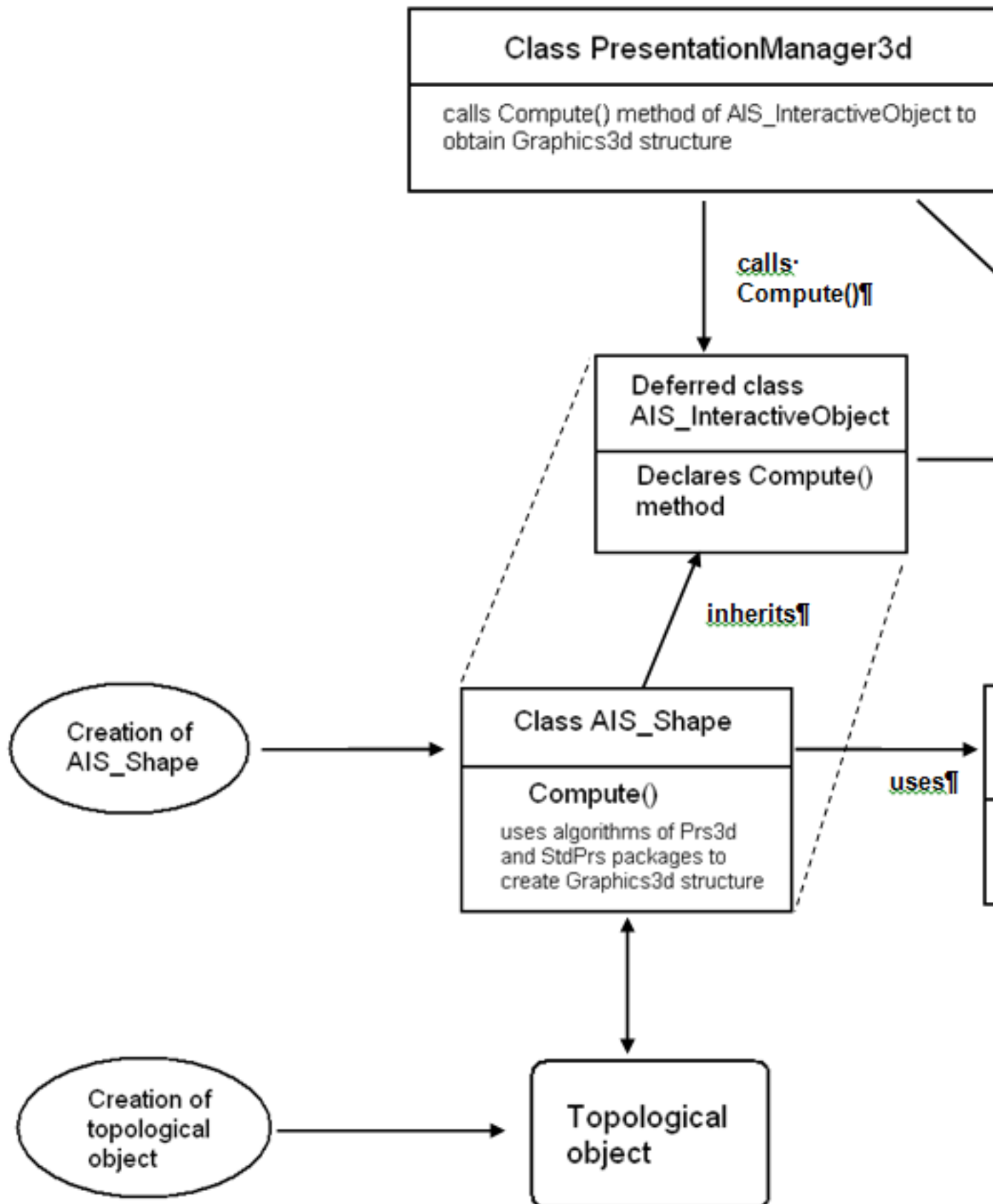


Figure 2: Processes involved in displaying a presentable shape

2.2 Selection

Objects that may be selected graphically, are displayed as sets of sensitive primitives, which provide sensitive zones in 2D graphic space. These zones are sorted according to their position on the screen when starting the selection process.

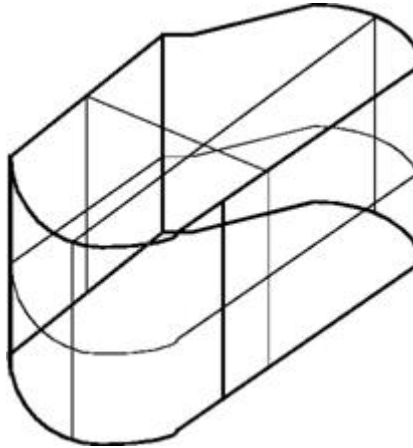


Figure 3: A model

The position of the mouse is also associated with a sensitive zone. When moving within the window where objects are displayed, the areas touched by the zone of the mouse are analyzed. The owners of these areas are then highlighted or signaled by other means such as the name of the object highlighted in a list. That way, you are informed of the identity of the detected element.

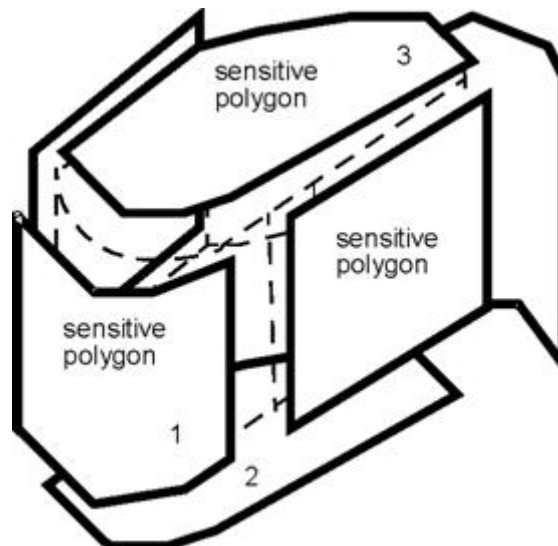


Figure 4: Modeling faces with sensitive primitives

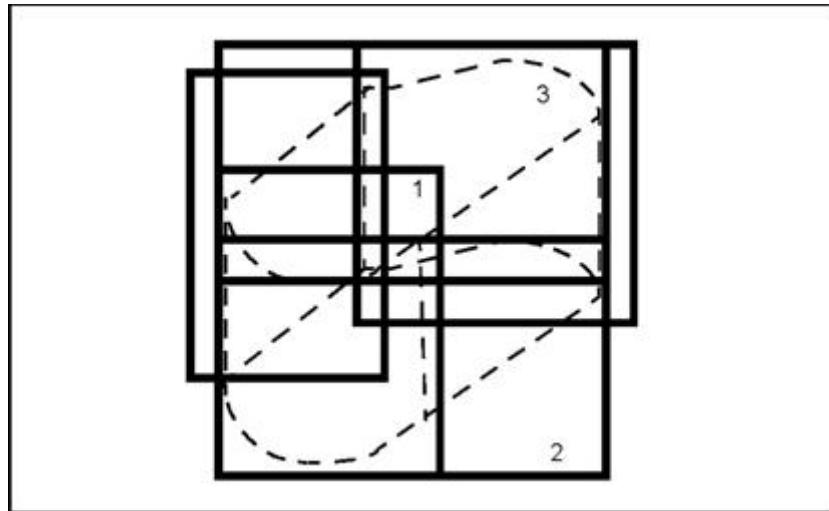


Figure 5: In a dynamic selection, each sensitive polygon is represented by its bounding rectangle

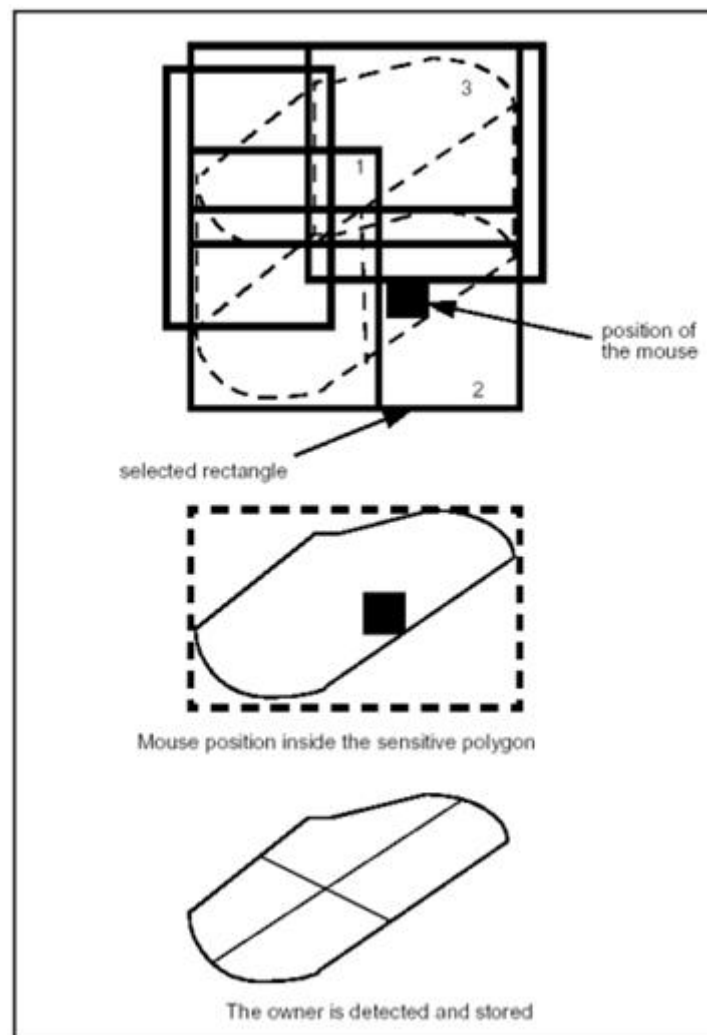


Figure 6: Reference to the sensitive primitive, then to the owner

2.2.1 The Sensitive Primitive

The sensitive primitive along with the entity owner allows defining what can be made selectable, and providing the link between the applicative object and the sensitive zones defined by the 2D bounding boxes. To be dynamically selectable, an object has to be represented either as a sensitive primitive or a set of them, e.g. 2D boxes that will be included in a sorting algorithm.

The use of 2D boxes allows a pre-selection of the detected entities. After pre-selection, the algorithm checks which sensitive primitives are actually detected. When detected, the primitives provide their owners' identity.

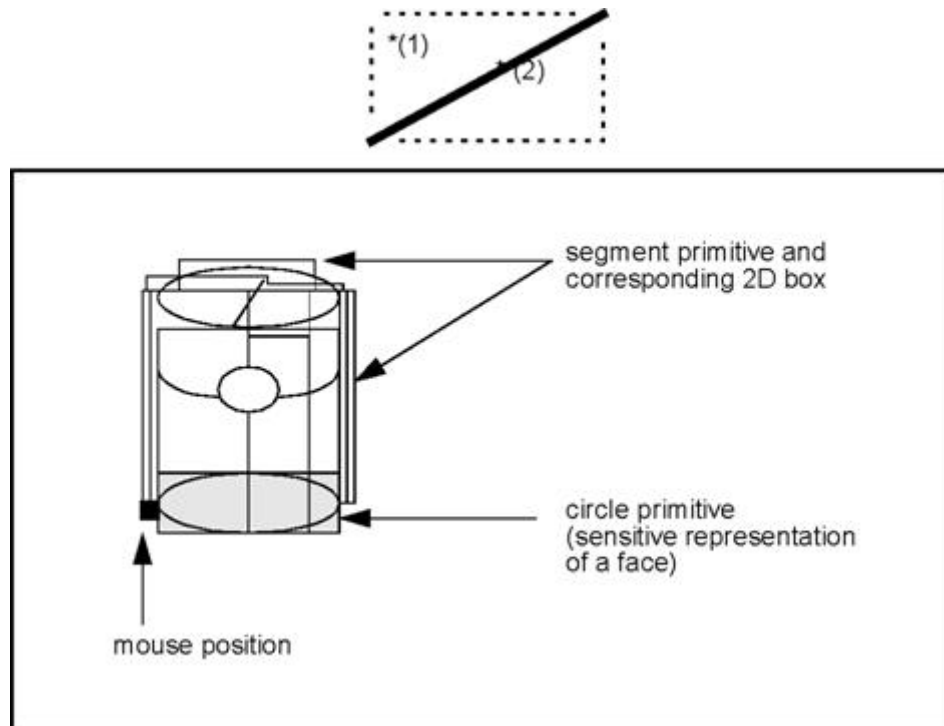


Figure 7: Example of sensitive primitives

In the example, the sensitive line segment proposes a bounding box to the selector. During selection, positions 1 and 2 of the mouse detect the box but after sorting, only position 2 retains the line segment as selected by the algorithm.

When the Box associated with the position of the mouse intersects the Box of a sensitive primitive, the owner of the sensitive primitive is called and its presentation is highlighted.

The notion of sensitive primitive is important for the developer when defining his own classes of sensitive primitives for the chosen selection modes. The classes must contain *Areas* and *Matches* functions.

The former provides the list of 2D sensitive boxes representing the sensitive primitive at pre-selection and the latter determines if the detection of the primitive by the 2D boxes is valid.

2.2.2 Dynamic Selection

Dynamic selection causes objects in a view to be automatically highlighted as the mouse cursor moves over them. This allows the user to be certain that the picked object is the correct one. Dynamic Selection is based on the following two concepts:

- a Selectable Object (*AIS_InteractiveObject*)
- an Interactive Context

Selectable Object

A selectable object presents a given number of selection modes which can be redefined, and which will be activated or deactivated in the selection manager's selectors.

Note that the selection mode of a selectable object, can refer to the selection mode of the object itself or to the selection mode of its part.

For each selection mode, a *SelectMgr_Selection* object class is included in the selectable object. (Each selection mode establishes a priority of selection for each class of selectable object defined.)

The notion of **Selection** is comparable to the notion of **Display**. Just as a display contains a set of graphic primitives that allow display of the entity in a specific display mode, a **Selection** contains a set of sensitive primitives, which allow detection of the entities they are associated with.

Interactive Context

The interactive context is used to manage both selectable objects and selection processes.

Selection modes may be activated or de-activated for given selectable objects. Information is then provided about the status of activated/de-activated selection modes for a given object in a given selector.

See also AIS: Application Interactive Services.

Let us consider, for example, a 3D selectable shape object, which corresponds to a topological shape.

For this class, seven selection modes can be defined:

- mode 0 - selection of the shape itself
- mode 1 - selection of vertices
- mode 2 - selection of edges
- mode 3 - selection of wires
- mode 4 - selection of faces
- mode 5 - selection of shells
- mode 6 - selection of solids
- mode 7 - selection of compsolids
- mode 8 - selection of compounds

Selection 2 includes the sensitive primitives that model all the edges of the shape. Each of these primitives contains a reference to the edge it represents.

The selections may be calculated before any activation and are graph independent as long as they are not activated in a given selector. Activation of selection mode 3 in a selector associated with a view V leads to the projection of the 3D sensitive primitives contained in the selection; then the 2D areas which represent the 2D bounding boxes of these primitives are provided to the sorting process of the selector containing all the detectable areas.

To deactivate selection mode 3 remove all those 2D areas.

2.2.3 Selection Packages

Selection of 3D data structures is provided using various algorithms. The following selection packages exist : *SelectBasics*, *SelectMgr*, *Select3D* and *StdSelect*.

Basic Selection

SelectBasics package contains the basic classes of the selection:

- the main definition of a sensitive primitive: *SensitiveEntity*, which is a selectable entity in a view;
- the definition of a sensitive primitive owner: *EntityOwner* this entity relates the primitive to the application entity which is to be selected in the view.
- the algorithm used for sorting sensitive boxes: *SortAlgo*

EntityOwner is used to establish a link from *SensitiveEntity* to application-level objects. For example, *SelectMgr_EntityOwner* (see below) class holds a pointer to corresponding *SelectableObject*.

Standard Selections

Select3D package provides definition of all 3D standard sensitive primitives such as point, curve and face. All these classes inherit from 3D *SensitiveEntry* from *SelectBasics* with an additional method, which allows recovery of the bounding boxes in the 2D graphic selection space, if required. This package also includes the 3D-2D projector.

StdSelect package provides standard uses of the classes described above and main tools used to prevent the developer from redefining the selection objects. In particular, *StdSelect* includes standard modes for selection of topological shapes, definition of several filter standard *Selection2d.ap* classes and 3D viewer selectors.

Note that each new Interactive Object must have all its selection modes defined.

Selection Management

SelectMgr package is used to manage the whole dynamic selection process.

It provides low level services and classes *SelectMgr_SelectionManager* and *SelectMgr_ViewerSelector*. They can be used when you do not want to use the services provided by AIS.

There are also implementations of *ViewerSelector* interface for 3D selection in *StdSelect* package: *ViewerSelector3d*.

SelectMgr manages the process of dynamic selection through the following services:

- Activating and deactivating selection modes for Interactive Objects.
- Adding and removing viewer selectors.
- Definitions of abstract filter classes.

The principle of graphic selection consists in representing the objects which you want to select by a bounding box in the selection view. The object is selected when you use the mouse to designate the zone produced by the object.

To realize this, the application creates a selection structure which is independent of the point of view. This structure is made up of sensitive primitives which have one owner object associated to each of them. The role of the sensitive primitive is to reply to the requests of the selection algorithm whereas the owner's purpose is to make the link between the sensitive primitive and the object to be selected. Each selection structure corresponds to a selection mode which defines the elements that can be selected.

Example: Selection of a Geometric Model

For example, to select a complete geometric model, the application can create a sensitive primitive for each face of the interactive object representing the geometric model. In this case, all the primitives share the same owner. On the other hand, to select an edge in a model, the application must create one sensitive primitive per edge.

```
void InteractiveBox::ComputeSelection (const Handle(SelectMgr_Selection)& theSel,
                                     const Standard_Integer theMode)
{
    switch (theMode)
    {
        case 0: // locating the whole box by making its faces sensitive
        {
```

```

Handle(SelectMgr_EntityOwner) anOwner = new SelectMgr_EntityOwner (this, 5);
for (Standard_Integer anIt = 1; anIt <= aFacesNb; anIt++)
{
    theSel->Add (new Select3D_SensitiveFace (anOwner,[array of the vertices] face I);
    break;
}
case 1: // locating the edges
{
    for (Standard_Integer anIt = 1; anIt <= 12; anIt++)
    {
        // 1 owner per edge
        Handle(mypk_EdgeOwner) anOwner = new mypk_EdgeOwner (this, anIt, 6); // 6->priority
        theSel->Add (new Select3D_SensitiveSegment (anOwner, firstpt (anIt), lastpt (anIt));
    }
}
}
}

```

The algorithms for creating selection structures store the sensitive primitives in a *SelectMgr_Selection* object. To do this, a set of ready-made sensitive primitives is supplied in the *Select3D* package. New sensitive primitives can be defined through inheritance from *SensitiveEntity*. For the application to make its own objects selectable, it must define owner classes inheriting *SelectMgr_EntityOwner*.

Selection structures for any interactive object are generated in *ComputeSelection()* method. In the example below there are different modes of selection on the topological shape contained within the interactive object, selection of the shape itself, the vertices, the edges, the wires, the faces.

```

void MyPack_MyClass::ComputeSelection(
    const Handle(SelectMgr_Selection)& theaSelection,
    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* object provides a high level service which will make the topological shape *myShape* selectable when the *AIS_InteractiveContext* is asked to display your object.

Note:

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 *Prs3d_Presentation* object, that results in large memory overhead for thousands of owners.
- Drawing selected owners one by one is not efficient from the OpenGL usage viewpoint.

That is why a different method has been introduced. On the basis of *SelectMgr_EntityOwner::IsAutoHighlight()* return value *AIS_LocalContext* object either uses the traditional way of highlighting (*IsAutoHighlight()* returned true) or groups such owners according to their Selectable Objects and finally calls *SelectMgr_SelectableObject::HighlightSelected()* or *ClearSelected()*, passing a group of owners as an argument.

Hence, an application can derive its own interactive object and redefine *HighlightSelected()*, *ClearSelected()* and *HighlightOwnerWithColor()* virtual methods to take advantage of such OpenGL technique as arrays of primitives. In any case, these methods should at least have empty implementation.

The *AIS_LocalContext::UpdateSelected (const Handle(AIS_InteractiveObject)&, Standard_Boolean)* method can be used for efficient redrawing a selection presentation for a given interactive object from an application code.

Additionally, the *SelectMgr_SelectableObject::ClearSelections()* method now accepts an optional Boolean argument. This parameter defines whether all object selections should be flagged for further update or not. This improved method can be used to re-compute an object selection (without redisplaying the object completely) when some selection mode is activated not for the first time.

2.2.4 How to use dynamic selection

Several operations must be performed prior to using dynamic selection:

1. Implement specific sensitive primitives if those defined in *Select3D* are not sufficient. These primitives must inherit from *SensitiveEntity* from *SelectBasics* or from a suitable *Select3D* sensitive entity class when a projection from 3D to 2D is necessary.
2. Define all the owner types, which will be used, and the classes of selectable objects, i.e. the number of possible selection modes for these objects and the calculation of the decomposition of the object into sensitive primitives of all the primitives describing this mode. It is possible to define only one default selection mode for a selectable object if this object is to be selectable in a unique way.
3. Install the process, which provides the user with the identity of the owner of the detected entities in the selection loop.

When all these steps have been carried out, follow the procedure below:

1. Create an interactive context.
2. Create the selectable objects and calculate their various possible selections.
3. Load these selectable objects in the interactive context. The objects may be common to all the selectors, i.e. they will be seen by all the selectors in the selection manager, or local to one selector or more.
4. Activate or deactivate the objects' selection modes in the selector(s). When activating a selection mode in a selector for a given object, the manager sends the order to make the sensitive primitives in this selector selectable. If the primitives are to be projected from 3D to 2D, the selector calls the specific method used to carry out this projection.

At this stage, the selection of selectable entities in the selectors is available. The selection loop informs constantly the selectors with the position of the mouse and questions them about the detected entities.

Let us suppose that you create an application that displays houses in a viewer of the *V3d* package and you want to select houses or parts of these houses (windows, doors, etc.) in the graphic window. You define a selectable object called *House* and propose four possible selection modes for this object:

1. selection of the house itself;
2. selection of the rooms
3. selection of the walls
4. selection of the doors.

You have to write the method, which calculates the four selections above, i.e. the sensitive primitives which are activated when the mode is. You must define the class *Owner* specific to your application. This class will contain the reference to the house element it represents: wall, door or room. It inherits from *EntityOwner* from *SelectMgr*. For example, let us consider a house with the following representation:

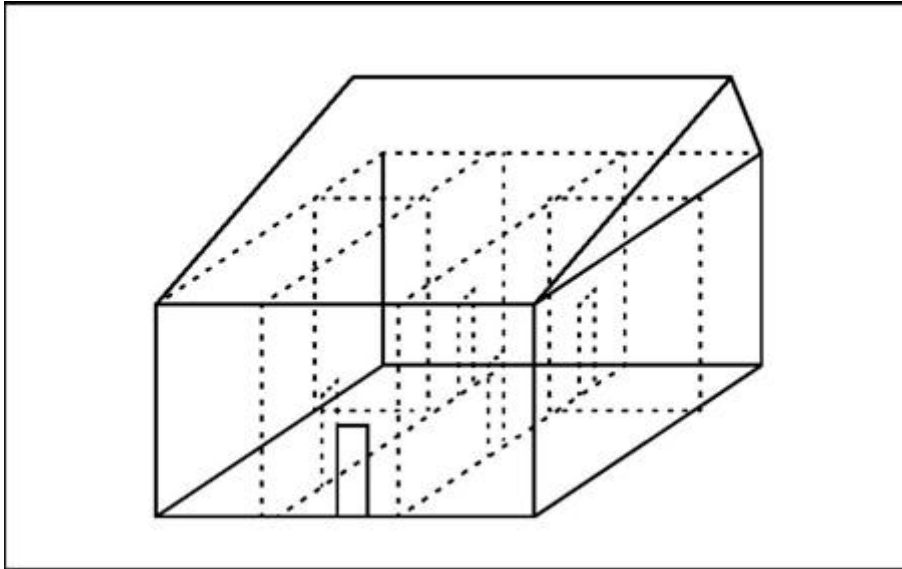


Figure 8: Selection of rooms in a house

To build the selection, which corresponds to the mode "selection of the rooms" (selection 2 in the list of selection modes), use the following procedure:

```

Void House::ComputeSelection
(
    Const Handle(SelectMgr_Selection)& Sel,
    const Standard_Integer mode {
        switch(mode) {
        case 0: //Selection of the rooms
        {
            for(Standard_Integer i = 1; i <= myNbRooms; i++)
            {
                //for every room, create an instance of the owner, the given room and its name.
                Handle(RoomOwner) aRoomOwner = new RoomOwner (Room(i), NameRoom(i));
                //Room() returns a room and NameRoom() returns its name.
                Handle(Select3d_SensitiveBox) aSensitiveBox;
                aSensitiveBox = new Select3d_SensitiveBox
                (aRoomOwner, Xmin, Ymin, Zmin, Xmax, Ymax, Zmax);
                Sel -> Add(aSensitiveBox);
            }
            break;
        case 1: ... //Selection of the doors
        } //Switch
    } // ComputeSelection

```

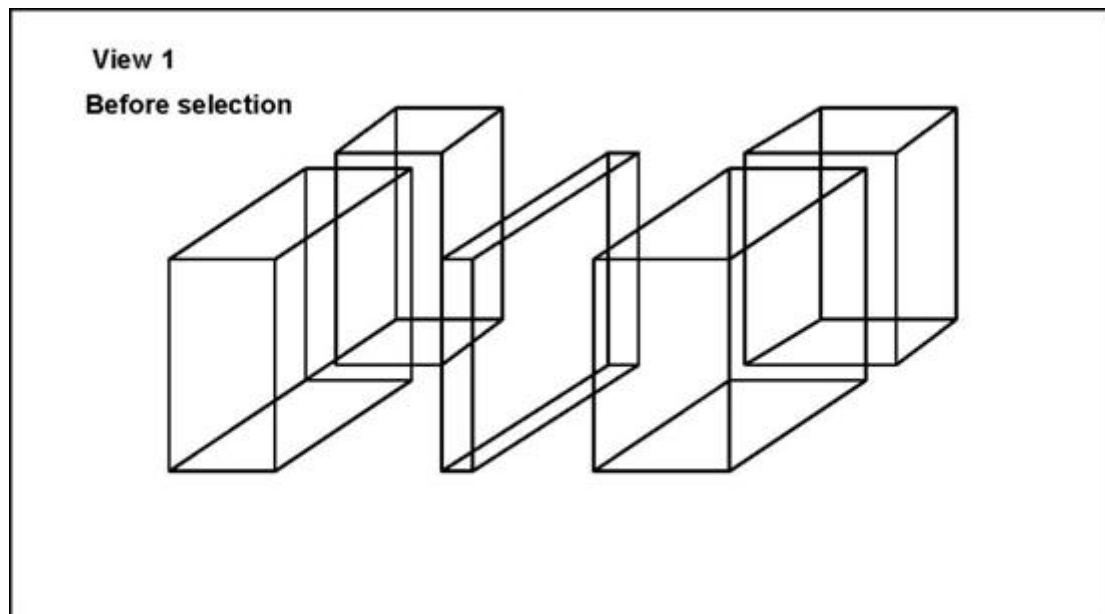


Figure 9: Activated sensitive boxes corresponding to selection mode 0 (selection of rooms)

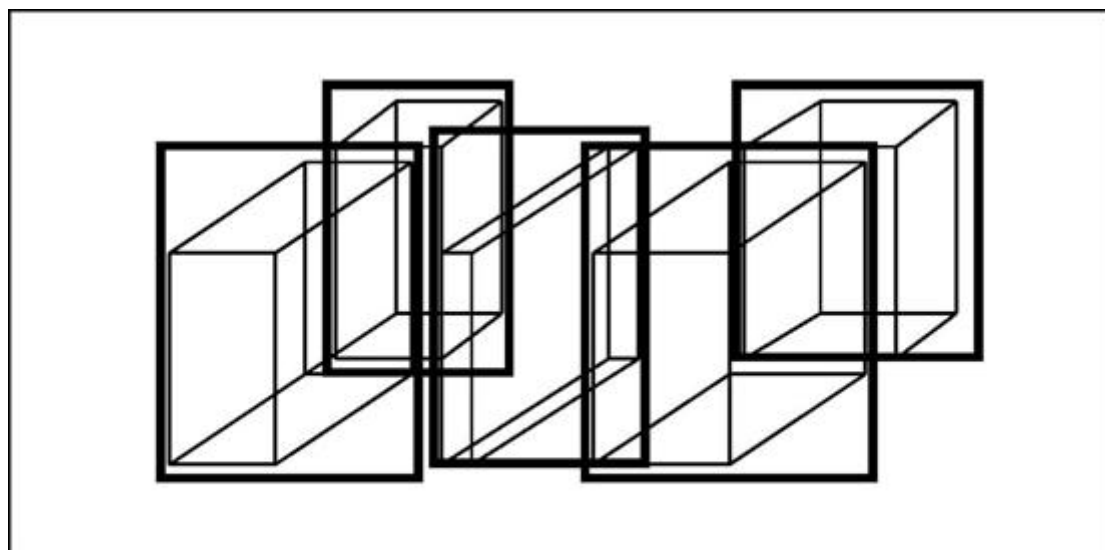


Figure 10. Activated sensitive rectangles in the selector during dynamic selection in view 1

Figure 10: Activated sensitive rectangles in the selector during dynamic selection in view 1

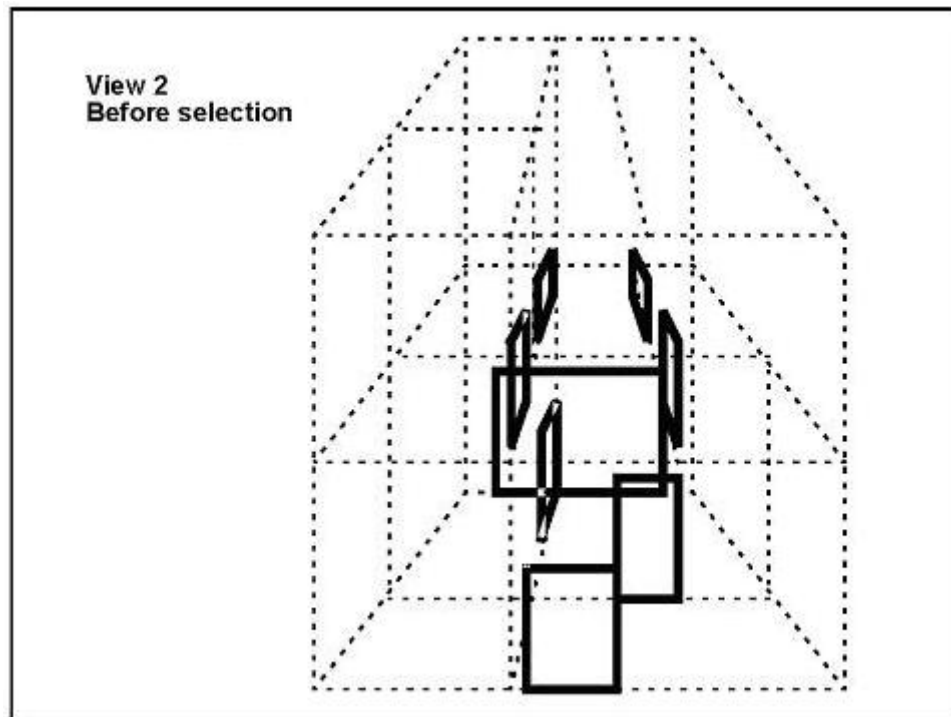


Figure 11: Activated sensitive polygons corresponding to selection mode 1 (selection of doors)

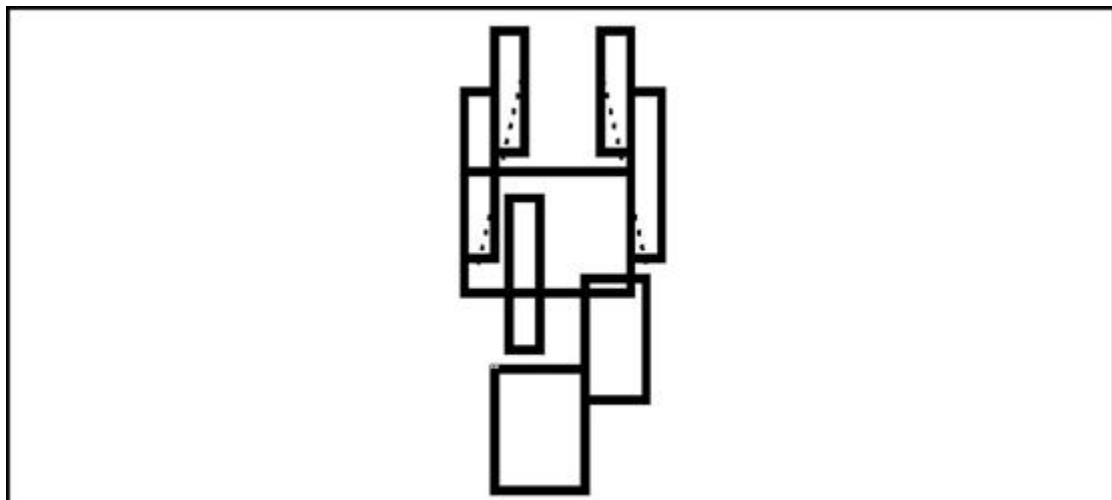


Figure 12: Sensitive rectangles in the selector during dynamic selection in view 2

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.

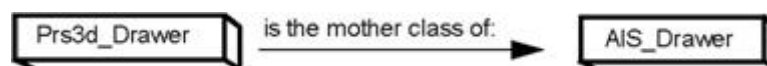
The central entity for management of visualization and selections is the **Interactive Context**. It is connected to the main viewer (and if need be, the trash bin viewer). It has two operating modes: the Neutral Point and the local visualization and selection context.

The neutral point, which is the default mode, allows easily visualizing and selecting interactive objects loaded into the context.

Local Contexts can be opened to prepare and use a temporary selection environment without disturbing the neutral point. It is possible to choose the interactive objects, which you want to act on, the selection modes, which you want to activate, and the temporary visualizations, which you will execute.

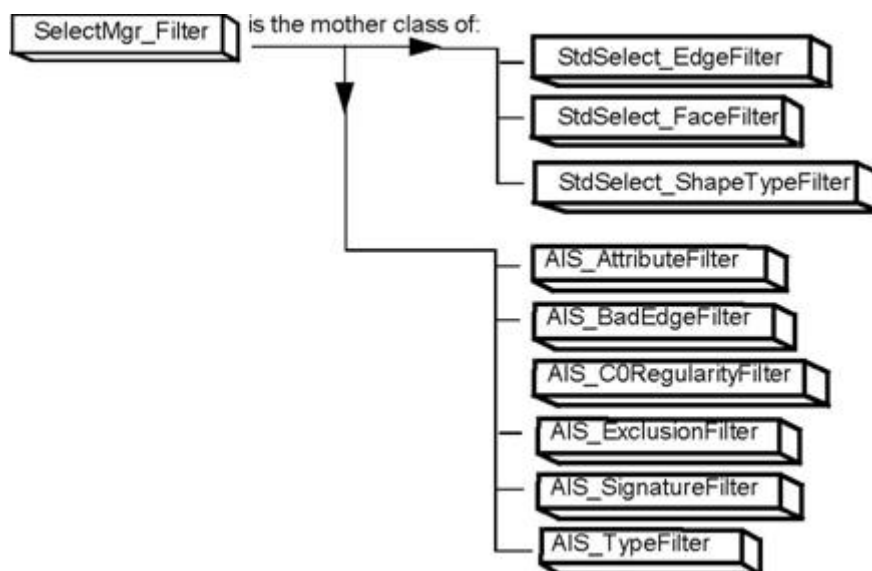
When the operation is finished, you close the current local context and return to the state in which you were before opening it (neutral point or previous local context).

Interactive Objects 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** 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, which allow refining the dynamic detection context. Some of these filters can be used at the Neutral Point, others only in an open local context. 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. 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.

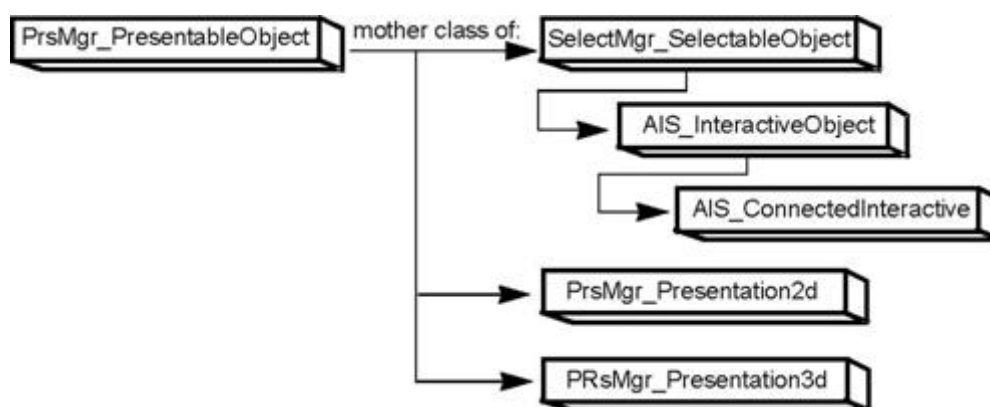
3.2.1 Presentations

An interactive object can have as many presentations as its creator wants to give it.

3D presentations are managed by `PresentationManager3D`. As this is transparent in AIS, the user does not have to worry about it.

A presentation is identified by an index 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)& aPresentationManager,
 const Handle(Prs3d_Presentation)& aPresentation,
 const Standard_Integer aMode = 0);
  
```

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

```

void PackageName_ClassName::Compute
(const Handle(Prs3d_Projector)& aProjector,
 const Handle(Prs3d_Presentation)& aPresentation);
  
```

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 signalled 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_TypeOfPresentation*:
 - *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. It supports two types of the HLR algorithm:

- the polygonal algorithm based on the shape's triangulation;
- the exact algorithm that works with the shape's real geometry.

The type of the HLR algorithm is stored in *AIS_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;
- *Prs3d_TOH_Algo* for an exact algorithm;
- *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 *AIS_Drawer* is set to *Prs3d_TOH_NotSet*, the *AIS_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::SetHighlightMode* and *AIS_InteractiveObject::UnSetHighlightMode* 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& SH, PrsMgr_TypeOfPresentation aType):
  AIS_InteractiveObject(aType), myShape(SH), myDrwr(new AIS_Drawer()) {SetHighlightMode(0);}
void myPk_IShape::Compute
  (const Handle(PrsMgr_PresentationManager3d) & PM,
   const Handle(Prs3d_Presentation)& P,
   const Standard_Integer TheMode)
{
  switch (TheMode) {
    case 0:
      StdPrs_WFDeflectionShape::Add (P,myShape,myDrwr); //algo for calculation of wireframe presentation
      break;
    case 1:
      StdPrs_ShadedShape::Add (P,myShape,myDrwr); //algo for calculation of shading presentation.
      break;
  }
}
void myPk_IsShape::Compute
  (const Handle(Prs3d_Projector)& Prj,
   const Handle(Prs3d_Presentation) P)
{
  StdPrs_HLRPolyShape::Add(P, myShape, myDrwr);
  //Hidden line mode calculation algorithm
}
```

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 entity which has been detected (see *Dynamic 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.

The calculation of Selection primitives (or sensitive primitives) is done by the intermediary of a virtual function, *ComputeSelection*. This should be implemented for each type of interactive object on which you want to make different type selections using the function *AIS_ConnectedInteractive::ComputeSelection*.

A detailed explanation of the mechanism and the manner of implementing this function has been given in *Dynamic Selection* chapter.

Moreover, just as the most frequently manipulated entity is *TopoDS_Shape*, the most used Interactive Object is *AIS_Shape*. You will see below activation functions for standard selection modes are proposed in the Interactive context (selection by vertex, by edges etc). To create new classes of interactive object with the same behavior as *AIS_Shape* - such as vertices and edges - you must redefine the virtual function *AIS_ConnectedInteractive::AcceptShapeDecomposition*.

You can change the default selection mode index of an Interactive Object using the following functions:

- *AIS_InteractiveObject::HasSelectionMode* checks if there is a selection mode;
- *AIS_InteractiveObject::SelectionMode* check the current selection mode;
- *AIS_InteractiveContext::SetSelectionMode* sets a selection mode;
- *AIS_InteractiveContext::UnsetSelectionMode* unsets a selection mode.

These functions can be useful if you decide that the 0 mode used by default will not do. In the same way, you can temporarily change the priority of certain interactive objects for selection of 0 mode to facilitate detecting them graphically using the following functions:

- *AIS_InteractiveObject::HasSelectionPriority* checks if there is a selection priority setting for the owner;
- *AIS_InteractiveObject::SelectionPriority* checks the current priority;
- *AIS_InteractiveObject::SetSelectionPriority* sets a priority;
- *AIS_InteractiveObject::UnsetSelectionPriority* unsets the priority.

3.2.5 Graphic attributes

Graphic attributes manager, or *AIS 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.
- The set of graphic attributes of an interactive object is stocked in an *AIS_Drawer*, which is only a *Prs3d_Drawer* with the possibility of a link to another drawer
- 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 13: Figure 13. Redefinition of virtual functions for changes in *AIS_Point*

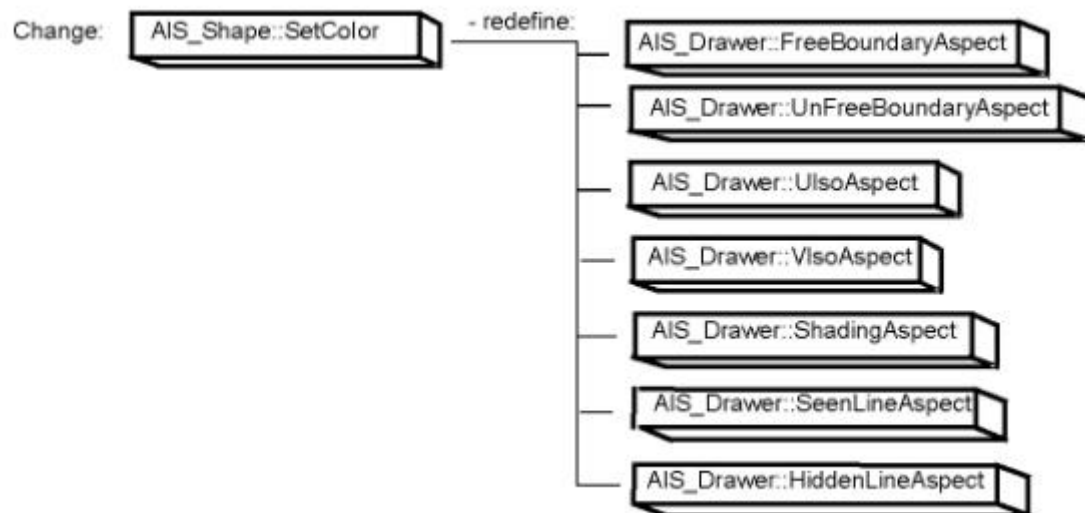


Figure 14: Figure 14. Redefinition of virtual functions for changes in AIS_Shape.

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

- *AIS_InteractiveObject::UnsetColor*
- *AIS_InteractiveObject::SetWidth*
- *AIS_InteractiveObject::UnsetWidth*
- *AIS_InteractiveObject::SetMaterial (const Graphic3d_NameOfPhysicalMaterial & aName)*
- *AIS_InteractiveObject::SetMaterial (const Graphic3d_MaterialAspect & aMat)*
- *AIS_InteractiveObject::UnsetMaterial*
- *AIS_InteractiveObject::SetTransparency*
- *AIS_InteractiveObject::UnsetTransparency*

For other types of attribute, it is appropriate to change the Drawer of the object directly using:

- *AIS_InteractiveObject::SetAttributes*
- *AIS_InteractiveObject::UnsetAttributes*

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 temporarily "moving" the representation and selection of Interactive Objects in a view without recalculation.

- *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 *Owner* in form of a *Transient*.

- *AIS_InteractiveObject::SetOwner*
- *AIS_InteractiveObject::HasOwner*
- *AIS_InteractiveObject::Owner*

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

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 method *AIS_InteractiveObject::SetPolygonOffsets* (*const Standard_Integer aMode*, *const Standard_Real aFactor*, *const Standard_Real aUnits*) allows setting up the polygon offsets.

The parameter *aMode* can contain various combinations of *Aspect_PolygonOffsetMode* enumeration elements:

- *Aspect_POM_None*
- *Aspect_POM_Off*
- *Aspect_POM_Fill*
- *Aspect_POM_Line*
- *Aspect_POM_Point*
- *Aspect_POM_All*

The combination of these elements defines the polygon display modes that will use the given offsets. You can switch off the polygon offsets by passing *Aspect_POM_Off*. Passing *Aspect_POM_None* allows changing the *aFactor* and *aUnits* values without changing the mode. If *aMode* is different from *Aspect_POM_Off*, the *aFactor* and *aUnits* arguments are used by the graphics renderer to calculate the depth offset value:

```
offset = aFactor * m + aUnits * r
```

where *m* is the maximum depth slope for the currently displayed polygons, *r* is the minimum depth resolution (implementation-specific).

Negative offset values move polygons closer to the viewer while positive values shift polygons away.

Warning

This method has a side effect - it creates its own shading aspect if not yet created, so it is better to set up the object shading aspect first.

You can use the following functions to obtain the current settings for polygon offsets:

```
void AIS_InteractiveObject::PolygonOffsets
(Standard_Integer &aMode,
 Standard_Real &aFactor,
 Standard_Real &aUnits)
Standard_Boolean AIS_InteractiveObject::HasPolygonOffsets()
```

The same operation could be performed for the interactive object known by the *AIS_InteractiveContext* with the following methods:

```
void AIS_InteractiveContext::SetPolygonOffsets
(const Handle(AIS_InteractiveObject) &anObj,
 const Standard_Integer aMode,
 const Standard_Real aFactor,
 const Standard_Real aUnits)
void AIS_InteractiveContext::PolygonOffsets
(const Handle(AIS_InteractiveObject) &anObj,
 Standard_Integer &aMode,
 Standard_Real &aFactor,
 Standard_Real &aUnits)
Standard_Boolean AIS_InteractiveContext::HasPolygonOffsets
(const Handle(AIS_InteractiveObject) &anObj)
```

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) TheAISShape = new AIS_Shape (ashape);
myIntContext->Display(TheAISShape);
myIntContext->SetDisplayMode(TheAISShape ,1);
myIntContext->SetColor(TheAISShape,Quantity_NOC_RED);
```

You can also write

```
Handle (AIS_Shape) TheAISShape = new AIS_Shape (ashape);
TheAISShape->SetColor(Quantity_NOC_RED);
TheAISShape->SetDisplayMode(1);
myIntContext->Display(TheAISShape);
```

3.3.2 Groups of functions

Neutral Point and **Local Context** 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. Opening **Local contexts** allows preparing and using a temporary selection environment without disturbing the neutral point.

A set of functions allows choosing the interactive objects which you want to act on, the selection modes which you want to activate, and the temporary visualizations which you will execute. When the operation is finished, you close the current local context and return to the state in which you were before opening it (neutral point or previous local context).

The Interactive Context is composed of many functions, which can be conveniently grouped according to the theme:

- management proper to the context;
- management in the local context;

- presentations and selection in open/closed context;
- selection strictly speaking.

Some functions can only be used in open Local Context; others in closed local context; others do not have the same behavior in one state as in the other.

3.3.3 Management of the Interactive Context

The **Interactive Context** is made up of a **Principal Viewer** and, optionally, a trash bin or **Collector Viewer**.

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 one exists, or from the context drawer for the others.

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*;
- Sub-Intensity color. By default: *Quantity_NOC_GRAY40*.

All of these settings can be modified by functions proper to the Context.

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: *obj1* and *obj2*:

```
TheCtx->Display(obj1,Standard_False); // False = no viewer update
TheCtx->Display(obj2,Standard_True); // True = viewer update
TheCtx->SetDisplayMode(obj1,3,Standard_False);
TheCtx->SetDisplayMode(2);
// obj2 is visualised in mode 2 (if it accepts this mode)
// obj1 stays visualised in its mode 3.
```

PresentationManager3D and *Selector3D*, which manage the presentation and selection of present interactive objects, are associated to the main Viewer. The same is true of the optional Collector.

3.4 Local Context

3.4.1 Rules and Conventions

- Opening a local context allows preparing an environment for temporary presentations and selections, which will disappear once the local context is closed.
- It is possible to open several local contexts, but only the last one will be active.
- When you close a local context, the previous one, which is still on the stack, is activated again. If none is left, you return to Neutral Point.
- Each local context has an index created when the context opens. You should close the local context, which you have opened.

The interactive object, which is used the most by applications, is *AIS_Shape*. Consequently, standard functions are available which allow you to easily prepare selection operations on the constituent elements of shapes (selection of vertices, edges, faces etc) in an open local context. The selection modes specific to "Shape" type objects are called **Standard Activation Mode**. These modes are only taken into account in open local context and only act on interactive objects which have redefined the virtual function *AcceptShapeDecomposition()* so that it returns *TRUE*.

- Objects, which are temporarily in a local context, are not recognized by other local contexts a priori. Only objects visualized in Neutral Point are recognized by all local contexts.
- The state of a temporary interactive object in a local context can only be modified while another local context is open.

Warning

The specific modes of selection only concern the interactive objects, which are present in the Main Viewer. In the Collector, you can only locate interactive objects, which answer positively to the positioned filters when a local context is open, however, they are never decomposed in standard mode.

3.4.2 Management of Local Context

The local context can be opened using method *AIS_InteractiveContext::OpenLocalContext*. The following options are available:

- *UseDisplayedObjects*: allows loading the interactive objects visualized at Neutral Point in the opened local context. If *FALSE*, the local context is empty after being opened. If *TRUE*, the objects at Neutral Point are modified by their default selection mode.
- *AllowShapeDecomposition*: *AIS_Shape* allows or prevents decomposition in standard shape location mode of objects at Neutral Point, which are type-privileged (see *Selection* chapter). This Flag is only taken into account when *UseDisplayedObjects* is *TRUE*.
- *AcceptEraseOfObjects*: authorizes other local contexts to erase the interactive objects present in this context. This option is rarely used. The last option has no current use.

This function returns the index of the created local context. It should be kept and used when the context is closed.

To load objects visualized at Neutral Point into a local context or remove them from it use methods

```
AIS_InteractiveContext::UseDisplayedObjects
AIS_InteractiveContext::NotUseDisplayedObjects
```

Closing Local Contexts is done by:

```
AIS_InteractiveContext::CloseLocalContext
AIS_InteractiveContext::CloseAllContexts
```

Warning When the index is not specified in the first function, the current Context is closed. This option can be dangerous, as other Interactive Functions can open local contexts without necessarily warning the user. For greater security, you have to close the context with the index given on opening.

To get the index of the current context, use function *AIS_InteractiveContext::IndexOfCurrentLocal*. It allows closing all open local contexts at one go. In this case, you find yourself directly at Neutral Point.

When you close a local context, all temporary interactive objects are deleted, all selection modes concerning the context are canceled, and all content filters are emptied.

3.4.3 Presentation in a Neutral Point

You must distinguish between the **Neutral Point** and the **Open Local Context** states. Although the majority of visualization functions can be used in both situations, their behavior is different.

Neutral Point should be used to visualize the interactive objects, which represent and select an applicative entity. Visualization and Erasing orders are straightforward:

```

AIS_InteractiveContext::Display
(const Handle(AIS_InteractiveObject)& anIobj,
 const Standard_Boolean updateviewer=Standard_True);

AIS_InteractiveContext::Display
(const Handle(AIS_InteractiveObject)& anIobj,
 const Standard_Integer amode,
 const Standard_Integer aSelectionMode,
 const Standard_Boolean updateviewer = Standard_True,
 const Standard_Boolean allowdecomposition = Standard_True);

AIS_InteractiveContext::Erase
AIS_InteractiveContext::EraseMode
AIS_InteractiveContext::ClearPrs
AIS_InteractiveContext::Redisplay
AIS_InteractiveContext::Remove
AIS_InteractiveContext::EraseAll
AIS_InteractiveContext::Highlight
AIS_InteractiveContext::HighlightWithColor

```

Bear in mind the following points:

- It is recommended to display and erase interactive objects when no local context is opened, and open a local context for local selection only.
- The first *Display* function among the two ones available in *InteractiveContext* visualizes the object in its default mode (set with help of *SetDisplayMode()* method of *InteractiveObject* prior to *Display()* call), or in the default context mode, if applicable. If it has neither, the function displays it in 0 presentation mode. The object's default selection mode is automatically activated (0 mode by convention).
- Activating the displayed object by default can be turned off with help of *SetAutoActivateSelection()* method. This might be efficient if you are not interested in selection immediately after displaying an object.
- The second *Display* function should only be used in Neutral Point to visualize a supplementary mode for the object, which you can erase by *EraseMode (...)*. You activate the selection mode. This is passed as an argument. By convention, if you do not want to activate a selection mode, you must set the *SelectionMode* argument to -1. This function is especially interesting in open local context, as we will see below.
- In Neutral Point, it is not advisable to activate other selection modes than the default selection one. It is preferable to open a local context in order to activate particular selection modes.
- When you call *Erase(Interactive object)* function, the *PutInCollector* argument, which is *FALSE* by default, allows you to visualize the object directly in the Collector and makes it selectable (by activation of 0 mode). You can nonetheless block its passage through the Collector by changing the value of this option. In this case, the object is present in the Interactive Context, but is not seen anywhere.
- *Erase()* with *putInCollector = Standard_True* might be slow as it recomputes the object presentation in the Collector. Set *putInCollector* to *Standard_False* if you simply want to hide the object's presentation temporarily.
- Visualization attributes and graphic behavior can be modified through a set of functions similar to those for the interactive object (color, thickness of line, material, transparency, locations, etc.) The context then manages immediate and deferred updates.
- Call *Remove()* method of *InteractiveContext* as soon as the interactive object is no longer needed and you want to destroy it.. Otherwise, references to *InteractiveObject* are kept by *InteractiveContext*, and the *Object* is not destroyed, which results in memory leaks. In general, if the presentation of an interactive object can be computed quickly, it is recommended to *Remove()* it instead of using *Erase()* method.

3.4.4 Presentation in the Local Context

In open local context, the *Display* functions presented above can be as well.

WARNING

The function *AIS_InteractiveObject::Display* automatically activates the object's default selection mode. When you only want to visualize an Interactive Object in open Context, you must call the function *AIS_InteractiveContext::Display*.

You can activate or deactivate specific selection modes in the local open context in several different ways: Use the Display functions with the appropriate modes.

```
AIS_InteractiveContext::ActivateStandardMode
//can be used only if a Local Context is opened.
AIS_InteractiveContext::DeactivateStandardMode
AIS_InteractiveContext::ActivatedStandardModes
AIS_InteractiveContext::SetShapeDecomposition
```

This activates the corresponding selection mode for all objects in Local Context, which accept decomposition into sub-shapes. Every new Object which has been loaded into the interactive context and which meets the decomposition criteria is automatically activated according to these modes.

WARNING

If you have opened a local context by loading an object with the default options (*AllowShapeDecomposition = Standard_True*), all objects of the "Shape" type are also activated with the same modes. You can change the state of these "Standard" objects by using *SetShapeDecomposition(Status)*.

Load an interactive object by the function *AIS_InteractiveContext::Load*.

This function allows loading an Interactive Object whether it is visualized or not with a given selection mode, and/or with the necessary decomposition option. If *AllowDecomp=TRUE* and obviously, if the interactive object is of the "Shape" type, these "standard" selection modes will be automatically activated as a function of the modes present in the Local Context.

Use *AIS_InteractiveContext::Activate* and *AIS_InteractiveContext::Deactivate* to directly activate/deactivate selection modes on an object.

3.4.5 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 in local context 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 the Local context 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 *IsOk()*:

```
class MyFilter : public SelectMgr_Filter { };
virtual Standard_Boolean MyFilter::IsOk
    (const Handle(SelectMgr_EntityOwner)& anObj) const = 0;
```

In *SelectMgr*, there are also Composition filters (AND Filters, OR Filters), which allow combining several filters. In *InteractiveContext*, all filters that you add are stocked 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.

As there are specific behaviors on shapes, each new *Filter* class must, if necessary, redefine *AIS_LocalContext::ActsOn* function, which informs the Local Context if it acts on specific types of sub-shapes. By default, this function answers *FALSE*.

WARNING

Only type filters are activated in Neutral Point to make it possible to identify a specific type of visualized object. For filters to come into play, one or more object selection modes must be activated.

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 local context.

Example

```
myContext->OpenLocalContext(Standard_False);
// no object in neutral point is loaded

myContext->ActivateStandardMode(TopAbs_Face);
//activates decomposition of shapes into faces.
Handle (AIS_Shape) myAIShape = new AIS_Shape ( ATopoShape);

myContext->Display(myAIShape,1,-1,Standard_True,Standard_True);

//shading visualization mode, no specific mode, authorization for decomposition into sub-shapes. At this
//stage, myAIShape is decomposed into faces...

Handle(StdSelect_FaceFilter) Fil1= new
    StdSelect_FaceFilter(StdSelect_Rev);
Handle(StdSelect_FaceFilter) Fil2= new
    StdSelect_FaceFilter(StdSelect_Plane);

myContext->AddFilter(Fil1);
myContext->AddFilter(Fil2);

//only faces of revolution or planar faces will be selected

myContext->MoveTo( xpix,ypix,Vue);
// detects the mouse position
```

3.4.6 Selection in the Local Context

Dynamic detection and selection are put into effect in a straightforward way. There are only a few conventions and functions to be familiar with. The functions are the same in neutral point and in open local context:

- *AIS_InteractiveContext::MoveTo* - passes mouse position to Interactive Context selectors
- *AIS_InteractiveContext::Select* - stocks what has been detected on 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, whether you are in neutral point or Local 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::SetAutomaticHighlight
AIS_InteractiveContext::AutomaticHighlight
```

If there is no open local context, the objects selected are called **current objects**. If there is a local context, they are called **selected objects**. Iterators allow entities to be recovered in either case. A set of functions allows manipulating the objects, which have been placed in these different lists.

WARNING

When a Local Context is open, you can select entities other than interactive objects (vertices, edges etc.) from decompositions in standard modes, or from activation in specific modes on specific interactive objects. Only interactive objects are stocked in the list of selected objects.

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

- *AIS_InteractiveContext::HasDetected* informs if something has been detected;
- *AIS_InteractiveContext::HasDetectedShape* informs if it is a shape;
- *AIS_InteractiveContext::DetectedShape* gets the shape if the detected entity is an object;
- *AIS_InteractiveContext::DetectedInteractive* gets the interactive object if the detected entity is an object.

After using the *Select* and *ShiftSelect* functions in Neutral Point, you can explore the list of selections, referred to as current objects in this context. The following functions can be used:

- *AIS_InteractiveContext::InitCurrent* initiates a scan of this list;
- *AIS_InteractiveContext::MoreCurrent* extends the scan;
- *AIS_InteractiveContext::NextCurrent* resumes the scan;
- *AIS_InteractiveContext::Current* gets the name of the current object detected in the scan;
- *AIS_InteractiveContext::FirstCurrentObject* gets the first current interactive object;
- *AIS_InteractiveContext::HighlightCurrents* highlights current objects;
- *AIS_InteractiveContext::UnhighlightCurrents* removes highlight from current objects;
- *AIS_InteractiveContext::ClearCurrents* empties the list of current objects in order to update it;
- *AIS_InteractiveContext::IsCurrent* finds the current object.

In the Local Context, you can explore the list of selected objects available. The following functions can be used:

- *AIS_InteractiveContext::InitSelected* initiates the list of objects;
- *AIS_InteractiveContext::MoreSelected* extends the list of objects;
- *AIS_InteractiveContext::NextSelected* resumes a scan;
- *AIS_InteractiveContext::SelectedShape* gets the name of the selected object;
- *AIS_InteractiveContext::HasSelectedShape* checks if the selected shape is obtained;
- *AIS_InteractiveContext::Interactive* gets the picked interactive object;
- *AIS_InteractiveContext::HasApplicative* checks if the applicative object has an owner from Interactive attributed to it;
- *AIS_InteractiveContext::Applicative* gets the owner of the detected applicative entity;
- *AIS_InteractiveContext::IsSelected* gets the name of the selected object.

Example

```

myAISCtx->InitSelected();
while (myAISCtx->MoreSelected())
{
    if (myAISCtx->HasSelectedShape)
    {
        TopoDS_Shape  ashape = myAISCtx->SelectedShape();
        // to be able to use the picked shape
    }

    else
    {
        Handle_AIS_InteractiveObject  anyobj = myAISCtx->Interactive();
        // to be able to use the picked interactive object
    }
}
myAISCtx->NextSelected();
}

```

You have to ask whether you have selected a shape or an interactive object before you can recover the entity in the Local Context or in the iteration loop. If you have selected a Shape from *TopoDS* on decomposition in standard mode, the *Interactive()* function returns the interactive object, which provided the selected shape. Other functions allow you to manipulate the content of Selected or Current Objects:

- *AIS_InteractiveContext::EraseSelected* erases the selected objects;
- *AIS_InteractiveContext::DisplaySelected* displays them;
- *AIS_InteractiveContext::SetSelected* puts the objects in the list of selections;
- *AIS_InteractiveContext::SetSelectedCurrent* takes the list of selected objects from a local context and puts it into the list of current objects in Neutral Point;
- *AIS_InteractiveContext::AddOrRemoveSelected* adds or removes an object from the list of selected entities;
- *AIS_InteractiveContext::HighlightSelected* highlights the selected object;
- *AIS_InteractiveContext::UnhighlightSelected* removes highlighting from the selected object;
- *AIS_InteractiveContext::ClearSelected* empties the list of selected objects.

You can highlight and remove highlighting from a current object, and empty the list of current objects using the following functions:

```

AIS_InteractiveContext::HighlightCurrents
AIS_InteractiveContext::UnhighlightCurrents
AIS_InteractiveContext::ClearCurrents

```

When you are in an open Local Context, you may need to keep "temporary" interactive objects. This is possible using the following functions:

- *AIS_InteractiveContext::KeepTemporary* transfers the characteristics of the interactive object seen in its local context (visualization mode, etc.) to the neutral point. When the local context is closed, the object does not disappear.
- *AIS_InteractiveContext::SetSelectedCurrent* allows the selected object to become the current object when you close the local context.

You can also want to use function *AIS_InteractiveContext::ClearLocalContext* to modify in a general way the state of the local context before continuing a selection (emptying objects, removing filters, standard activation modes).

3.4.7 Recommendations

The possibilities of use for local contexts are numerous depending on the type of operation that you want to perform:

- working on all visualized interactive objects,

- working on only a few objects,
- working on a single object.

When you want to work on one type of entity, you should open a local context with the option *UseDisplayedObjects* set to *FALSE*. Some functions which allow you to recover the visualized interactive objects, which have a given Type, and Signature from the "Neutral Point" are:

```
AIS_InteractiveContext::DisplayedObjects (AIS_ListOfInteractive& aListOfIO) const;
AIS_InteractiveContext::DisplayedObjects (const AIS_KindOfInteractive WhichKind, const Standard_Integer
WhichSignature;
AIS_ListOfInteractive& aListOfIO) const;
```

At this stage, you only have to load the functions *Load*, *Activate*, and so on.

When you open a Local Context with default options, you must keep the following points in mind:

- The Interactive Objects visualized at Neutral Point are activated with their default selection mode. You must deactivate those, which you do not want to use.
- The Shape Type Interactive Objects are automatically decomposed into sub-shapes when standard activation modes are launched.
- The "temporary" Interactive Objects present in the Local Contexts are not automatically taken into account. You have to load them manually if you want to use them.

The stages could be the following:

1. Open a Local Context with the right options;
2. Load/Visualize the required complementary objects with the desired activation modes.
3. Activate Standard modes if necessary
4. Create its filters and add them to the Local Context
5. Detect/Select/recover the desired entities
6. Close the Local Context with the adequate index.

It is useful to create an **interactive editor**, to which you pass the Interactive Context. This allow setting up different contexts of selection/presentation according to the operation, which you want to perform.

Let us assume that you have visualized several types of interactive objects: *AIS_Points*, *AIS_Axes*, *AIS_Trihedrons*, and *AIS_Shapes*.

For your applicative function, you need an axis to create a revolved object. You could obtain this axis by identifying:

- an axis which is already visualized,
- 2 points,
- a rectilinear edge on the shapes which are present,
- a cylindrical face on the shapes (You will take the axis of this face)

```
myIHMEditor::myIHMEditor
(const Handle(AIS_InteractiveContext)& Ctx,
...) :
myCtx (Ctx),
...
{
}

myIHMEditor::PrepareContext ()
{
myIndex =myCtx->OpenLocalContext ();
```

```

//the filters

Handle(AIS_SignatureFilter) F1 = new AIS_SignatureFilter(AIS_KOI_Datum,AIS_SD_Point);
//filter on the points

Handle(AIS_SignatureFilter) F2 = new AIS_SignatureFilter(AIS_KOI_Datum,AIS_SD_Axis);
//filters on the axes.

Handle(StdSelect_FaceFilter) F3 = new StdSelect_FaceFilter(AIS_Cylinder);
//cylindrical face filters
//...
// activation of standard modes on the shapes..
myCtx->ActivateStandardMode(TopAbs_FACE);
myCtx->ActivateStandardMode(TopAbs_VERTEX);
myCTX->Add(F1);
myCTX->Add(F2);
myCTX->Add(F3);

// at this point, you can call the selection/detection function
}

void myIHMEditor::MoveTo(xpix,ypix,Vue)

{ myCTX->MoveTo(xpix,ypix,vue);
// the highlight of what is detected is automatic.
}
Standard_Boolean myIHMEditor::Select()
{
// returns true if you should continue the selection
myCTX->Select();
myCTX->InitSelected();
if(myCTX->MoreSelected())
{
if(myCTX->HasSelectedShape())
{ const TopoDS_Shape& sh = myCTX->SelectedShape();
if( vertex){
if(myFirstV...)
{
//if it is the first vertex, you stock it, then you deactivate the faces and only keep the
filter on the points:
mypoint1 = ...;
myCtx->RemoveFilters();
myCTX->DeactivateStandardMode(TopAbs_FACE);
myCtx->Add(F1);
// the filter on the AIS_Points
myFirstV = Standard_False;
return Standard_True;
}
else
{
mypoint2 =...;
// construction of the axis return Standard_False;
}
}
else
{
//it is a cylindrical face : you recover the axis; visualize it; and stock it.
return Standard_False;
}
}
// it is not a shape but is no doubt a point.
else
{
Handle(AIS_InteractiveObject)
SelObj = myCTX->SelectedInteractive();
if(SelObj->Type()==AIS_KOI_Datum)
{
if(SelObj->Signature()==1)
{
if (firstPoint)
{
mypoint1 =...
return Standard_True;
}
else
{
mypoint2 = ...;
//construction of the axis, visualization, stocking
return Standard_False;
}
}
}
else
{
// you have selected an axis; stock the axis
return Standard_False;
}
}
}
}

```

```

    }
    }
    }
}

void myIHMEditor::Terminate()
{
myCtx->CloseLocalContext(myIndex);
...
}

```

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 0 : selection of a trihedron;
- mode 1 : selection of the origin of the trihedron;
- mode 2 : selection of the axes;
- mode 3 : selection of the planes XOY, YOZ, XOZ.

when you activate one of modes: 1 2 3 4, 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 unit of length is millimeter, and the default value for the representation of axes is 100. If you modify these dimensions, you must temporarily recover the object **Drawer**. From it, take the *Aspects* in which the values for length are stored (*PlaneAspect* for the plane, *FirstAxisAspect* for trihedra), and change these values inside these Aspects. Finally, recalculate the presentation.

3.5.2 Object

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

AIS_Shape has three visualization modes :

- mode 0 : Line (default mode)
- mode 1 : Shading (depending on the type of shape)
- mode 2 : Bounding Box

And at maximum seven selection modes, depending on the shape complexity:

- mode 0 : selection of the *AIS_Shape*;
- mode 1 : selection of the vertices;
- mode 2 : selection of the edges;
- mode 3 : selection of the wires;
- mode 4 : selection of the faces;
- mode 5 : selection of the shells;
- mode 6 : selection of the constituent solids.
- *AIS_Triangulation* is a simple interactive object for displaying triangular mesh contained in *Poly_Triangulation* container.
- *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_ConnectedShape* is an object connected to interactive objects having a shape; this class has the same decompositions as *AIS_Shape*. Furthermore, it allows a presentation of hidden parts, which are calculated automatically from the shape of its 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 calculations of presentation)
- *AIS_MultipleConnectedShape* is an interactive Object connected to a list of interactive objects having a Shape (*AIS_Shape*, *AIS_ConnectedShape*, *AIS_MultipleConnectedShape*). The presentation of hidden parts is calculated automatically.
- *AIS_TexturedShape* is an Interactive Object that supports texture mapping. It is constructed as a usual *AIS_Shape*, but has additional methods that allow to map a texture on it.
- *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_Color (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_Color (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.

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 *AIS*:

- *AIS_ConcentricRelation*
- *AIS_FixRelation*
- *AIS_IdenticRelation*
- *AIS_ParallelRelation*
- *AIS_PerpendicularRelation*
- *AIS_Relation*
- *AIS_SymmetricRelation*
- *AIS_TangentRelation*

The list of relations is not exhaustive.

3.5.4 Dimensions

- *AIS_AngleDimension*
- *AIS_Chamf3dDimension*
- *AIS_DiameterDimension*
- *AIS_DimensionOwner*
- *AIS_LengthDimension*
- *AIS_OffsetDimension*
- *AIS_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) &Builder, Standard_Boolean TreatAsHilighter)
MeshVS_Mesh::RemoveBuilder (const Standard_Integer Index)
MeshVS_Mesh::RemoveBuilderById (const Standard_Integer Id)
```

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_HilighPrs
```

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_OD*
- *MeshVS_SMF_Link*
- *MeshVS_SMF_Face*
- *MeshVS_SMF_Volume*
- *MeshVS_SMF_Element* - groups *OD*, *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 (StlMesh_Mesh) aSTLMesh = RWStl::ReadFile (aFileName);
Handle (XSDRAWSTLVRML_DataSource) aDataSource = new XSDRAWSTLVRML_DataSource (aSTLMesh);

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

// use default presentation builder
Handle (MeshVS_MeshPrsBuilder) aBuilder = new MeshVS_MeshPrsBuilder (aMesh);
aMesh->AddBuilder (aBuilder, Standard_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
(aMesh, MeshVS_DMF_NodalColorDataPrs | MeshVS_DMF_OCCMask);
aBuilder->UseTexture (Standard_True);

// prepare color map
Aspect_SequenceOfColor aColorMap;
aColorMap.Append ((Quantity_NameOfColor) Quantity_NOC_RED);
aColorMap.Append ((Quantity_NameOfColor) 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, Standard_True);
```

3.6 Dynamic Selection

The idea of dynamic selection is to represent the entities, which you want to select by a bounding box in the actual 2D space of the selection view. The set of these zones is ordered by a powerful sorting algorithm. To then find the applicative entities actually detected at this position, all you have to do is read which rectangles are touched at mouse position (X,Y) of the view, and judiciously reject some of the entities which have provided these rectangles.

3.6.1 How to go from the objects to 2D boxes

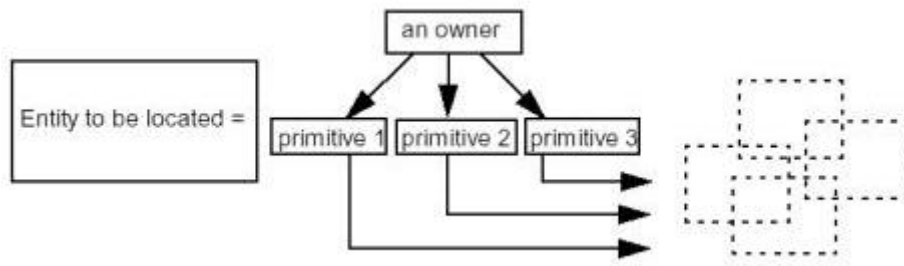
An intermediary stage consists in representing what you can make selectable by means of sensitive primitives and owners, entities of a high enough level to be known by the selector mechanisms.

The sensitive primitive is capable of:

- giving a 2D bounding box to the selector.
- answering the rejection criteria positively or negatively by a "Matches" function.
- being projected from 3D in the 2D space of the view if need be.
- returning the owner which it will represent in terms of selection.

A set of standard sensitive primitives exists in Select3D packages for 3D primitives.

The owner is the entity, which makes it possible to link the sensitive primitives and the objects that you really wanted to detect. It stocks the diverse information, which makes it possible to find objects. An owner has a priority (5 by default), which you can change to make one entity more selectable than another.



3.6.2 Implementation in an interactive/selectable object

Define the number of selection modes possible, i.e. what you want to identify by activating each of the selection modes.

For example: for an interactive object representing a topological shape:

- mode 0: selection of the interactive object itself;
- mode 1: selection of the vertices;
- mode 2: selection of the edges;
- mode 3: selection of the wires;
- mode 4: selection of the detectable faces.

For each selection mode of an interactive object, "model" is the set of entities, which you want to locate by these primitives and these owners.

There is an "owner" root class, *SelectMgr_EntityOwner*, containing a reference to a selectable object, which has created it. If you want to stock its information, you have to create classes derived from this root class. Example: for shapes, there is the *StdSelect_BRepOwner* class, which can save a *TopoDS* shape as a field as well as the Interactive Object.

The set of sensitive primitives which has been calculated for a given mode is stocked in *SelectMgr_Selection*.

For an Interactive object, the modeling is done in the *ComputeSelection* virtual function.

Let us consider an example of an interactive object representing a box.

We are interested in two location modes:

- mode 0: location of the whole box.
- mode 1: location of the edges on the box.

For the first mode, all sensitive primitives will have the same owner, which will represent the interactive object. In the second case, we have to create an owner for each edge, and this owner will have to contain the index for the edge, which it represents. You will create a class of owner, which derives from *SelectMgr_EntityOwner*.

The *ComputeSelection* function for the interactive box can have the following form:

```

void InteractiveBox::ComputeSelection
(const Handle Owner = new SelectMgr_EntityOwner(this,5);
            for(Standard_Integer I=1;I<=Nbfaces;I++)
            {
                //Array is a TColgp_Array1OfPnt: which represents the array of vertices. Sensitivity is
                Select3D_TypeOfSensitivity value
                Sel->Add(new Select3D_SensitiveFace(Owner,Array,Sensitivity));
            }
            break;
        }
    }
}

```

```

    }
    case 1:
    // locates the edges {
    for(Standard_Integer i=1;i<=12;i++)
    {
        // 1 owner per edge...
        Handle(mykp_EdgeOwner) Ownr = new mykp_EdgeOwner(this,i,6);
        //6->priority
        Sel->Add(new Select3D_SensitiveSegment (Ownr,firstpt(i),lastpt(i)));
    }
    break;
    }
}
}

```

Selectable objects are loaded in the selection manager, which has one or more selectors; in general, we suggest assigning one selector per viewer. All you have to do afterwards is to activate or deactivate the different selection modes for selectable objects. The *SelectionManager* looks after the call to the *ComputeSelection* functions for different objects.

NOTE: This procedure is completely hidden if you use the AIS Interactive Context

Example

```

//We have several " interactive boxes " box1, box2, box3;
Handle(SelectMgr_SelectionManager) SM = new SelectMgr_SelectionManager();
Handle(StdSelect_ViewerSelector3d) VS = new StdSelect_ViewerSelector3d();
SM->Add(VS);
SM->Load(box1); SM->Load(box2); SM->Load(box3);
// box load.
SM->Activate(box1,0,VS);
// activates mode 0 of box 1 in the selector VS
SM->Activate(box1,1,VS);
M->Activate(box3,1,VS);
VS->Pick(xpix,ypix,vue3d)
// detection of primitives by mouse position.
Handle(EntityOwner) POwnr = VS->OnePicked();
// picking of the "best" owner detected
for(VS->Init();VS->More();VS->Next())
{
    VS->Picked();
    // picking of all owners detected
}
SM->Deactivate(box1);
// deactivate all active modes of box1

```

4 3D Presentations

4.1 Glossary of 3D terms

- **Anti-aliasing** This mode attempts to improve the screen resolution by drawing lines and curves in a mixture of colors so that to the human eye the line or curve is smooth. The quality of the result is linked to the quality of the algorithm used by the workstation hardware.
- **Depth-cueing** Reduces the color intensity for the portion of an object further away from the eye to give the impression of depth. This is used for wireframe objects. Shaded objects do not require this.
- **Group** - a set of primitives and attributes on those primitives. Primitives and attributes may be added to a group but cannot be removed from a group, except by erasing them globally. A group can have a pick identity.
- **Light** There are five kinds of light source - ambient, headlight, directional, positional and spot. The light is only activated in a shading context in a view.
- **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. 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). It has no default attributes for the primitive lines, faces, markers, and text. Attributes may be set in a structure but they are overridden by the attributes in each group. Each structure has a display priority associated with it, which rules the order in which it is redrawn in a 3D viewer. If the visualization mode is incompatible with the view it is not displayed in that view, e.g. a shading-only object is not visualized in a wireframe view.
- **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. It 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 and attributes, such as polylines, planar polygons with or without holes, 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.

- **Polygons**

- Have one closed boundary,
- Have at least three vertices,
- Are planar and have a normal,
- Have interior attributes - style, color, front and back material, texture and reflection ratio,
- Have a boundary with the following attributes - type, width scale factor, color. The boundary is only drawn when the interior style is hollow.

- **Polygons with holes**

- Have multiple closed boundaries, each one with at least three vertices,
- Are planar and have a normal,
- Have interior attributes - style, color, front and back material,
- Have a boundary with the following attributes - type, width scale factor, color. The boundary is only drawn when the interior style is hollow.

- **Polylines**

- 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

Primitive arrays are a more efficient approach to describe and display the primitives from the aspects of memory usage and graphical performance. The key feature of the primitive arrays is that the primitive data is not duplicated. For example, two polygons could share the same vertices, so it is more efficient to keep the vertices in a single array and specify the polygon vertices with indices of this array. In addition to such kind of memory savings, the OpenGL graphics driver provides the Vertex Buffer Objects (VBO). VBO is a sort of video memory storage that can be allocated to hold the primitive arrays, thus making the display operations more efficient and releasing the RAM memory.

The Vertex Buffer Objects are enabled by default, but VBOs availability depends on the implementation of OpenGL. If the VBOs are unavailable or there is not enough video memory to store the primitive arrays, the RAM memory will be used to store the arrays.

The Vertex Buffer Objects can be disabled at the application level. You can use the method *Graphic3d_GraphicDriver::EnableVBO* (*const Standard_Boolean status*) to enable/disable VBOs:

The following example shows how to disable the VBO support:

```
// get the graphic driver
Handle (Graphic3d_GraphicDriver) aDriver =
    myAISContext->CurrentViewer()->Driver();

// disable VBO support
aDriver->EnableVBO (Standard_False);
```

Note that the use of Vertex Buffer Objects requires the application level primitive data provided by the *Graphic3d_ArrayOfPrimitives* to be transferred to the video memory. *TKOpenGL* transfers the data and releases the *Graphic3d_ArrayOfPrimitives* internal pointers to the primitive data. Thus it might be necessary to pay attention to such kind of behaviour, as the pointers could be modified (nullified) by the *TKOpenGL*.

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

- *Graphic3d_ArrayOfPoints*,
- *Graphic3d_ArrayOfPolygons*,
- *Graphic3d_ArrayOfPolylines*,
- *Graphic3d_ArrayOfQuadrangles*,
- *Graphic3d_ArrayOfQuadrangleStrips*,
- *Graphic3d_ArrayOfSegments*,
- *Graphic3d_ArrayOfTriangleFans*,
- *Graphic3d_ArrayOfTriangles*,
- *Graphic3d_ArrayOfTriangleStrips*.

The *Graphic3d_ArrayOfPrimitives* is a base class for these primitive arrays.

Method *Graphic3d_ArrayOfPrimitives::AddVertex* allows adding There is a set of similar methods to add vertices to the primitive array.

These methods take vertex coordinates as an argument and allow you to define the color, the normal and the texture coordinates assigned to the vertex. The return value is the actual number of vertices in the array.

You can also modify the values assigned to the vertex or query these values by the vertex index:

- *void Graphic3d_ArrayOfPrimitives::SetVertex*
- *void Graphic3d_ArrayOfPrimitives::SetVertexColor*
- *void Graphic3d_ArrayOfPrimitives::SetVertexNormal*

- `void Graphic3d_ArrayOfPrimitives::SetVertexTexel`
- `gp_Pnt Graphic3d_ArrayOfPrimitives::Vertices`
- `gp_Dir Graphic3d_ArrayOfPrimitives::VertexNormal`
- `gp_Pnt3d Graphic3d_ArrayOfPrimitives::VertexTexel`
- `Quantity_Color Graphic3d_ArrayOfPrimitives::VertexColor`
- `void Graphic3d_ArrayOfPrimitives::Vertices`
- `void Graphic3d_ArrayOfPrimitives::VertexNormal`
- `void Graphic3d_ArrayOfPrimitives::VertexTexel`
- `void Graphic3d_ArrayOfPrimitives::VertexColor`

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

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

// 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 = Prs3d_Root::CurrentGroup (aPrs);
aGroup->BeginPrimitives ();
aGroup->AddPrimitiveArray (anArray);
aGroup->EndPrimitives ();
```

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
Standard_Boolean IsNormals = Standard_False;
Standard_Boolean IsColors = Standard_False;
Standard_Boolean IsTextureCrds = Standard_False;
Handle (Graphic3d_ArrayOfTriangles) anArray =
    new Graphic3d_ArrayOfTriangles (aVerticesMaxCount,
                                    aEdgesMaxCount,
                                    IsNormals,
                                    IsColors,
                                    IsTextureCrds);

// 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->AddEdge (1); // first triangle
anArray->AddEdge (2);
anArray->AddEdge (3);
anArray->AddEdge (1); // second triangle
anArray->AddEdge (2);
anArray->AddEdge (4);

// add the array to the structure
Handle (Graphic3d_Group) aGroup = Prs3d_Root::CurrentGroup (aPrs);
aGroup->BeginPrimitives ();
aGroup->AddPrimitiveArray (anArray);
aGroup->EndPrimitives ();
```

If the primitive array presents primitives built from sequential sets of vertices, for example polygons, then you can specify the bounds, or the number of vertices for each primitive. You can use the method `Graphic3d_ArrayOfPrimitives::AddBound` to define the bounds and the color for each bound. This method returns the actual number of bounds.

It is also possible to set the color and query the number of edges in the bound and bound color.

```
Standard_Integer Graphic3d_ArrayOfPrimitives::Bound
Quantity_Color Graphic3d_ArrayOfPrimitives::BoundColor
void Graphic3d_ArrayOfPrimitives::BoundColor
```

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

```
// create an array
Standard_Boolean IsNormals      = Standard_False;
Standard_Boolean IsVertexColors = Standard_False;
Standard_Boolean IsFaceColors   = Standard_False;
Standard_Boolean IsTextureCrds  = Standard_False;
Handle (Graphic3d_ArrayOfPolygons) anArray =
    new Graphic3d_ArrayOfPolygons (aVerticesMaxCount,
                                   aBoundsMaxCount,
                                   aEdgesMaxCount,
                                   IsNormals,
                                   IsVertexColors,
                                   IsFaceColors,
                                   IsTextureCrds);

// add bounds to the array, first polygon
anArray->AddBound (3);
anArray->AddVertex (-1.0, 0.0, 0.0);
anArray->AddVertex ( 1.0, 0.0, 0.0);
anArray->AddVertex ( 0.0, 1.0, 0.0);

// add bounds to the array, second polygon
anArray->AddBound (4);
anArray->AddVertex (-1.0, 0.0, 0.0);
anArray->AddVertex ( 1.0, 0.0, 0.0);
anArray->AddVertex ( 1.0,-1.0, 0.0);
anArray->AddVertex (-1.0,-1.0, 0.0);

// add the array to the structure
Handle (Graphic3d_Group) aGroup = Prs3d_Root::CurrentGroup (aPrs);
aGroup->BeginPrimitives ();
aGroup->AddPrimitiveArray (anArray);
aGroup->EndPrimitives ();
```

There are also several helper methods. You can get the type of the primitive array:

```
Graphic3d_TypeOfPrimitiveArray
Graphic3d_ArrayOfPrimitives::Type
Standard_CString Graphic3d_ArrayOfPrimitives::StringType
```

and check if the primitive array provides normals, vertex colors and vertex texels (texture coordinates):

```
Standard_Boolean Graphic3d_ArrayOfPrimitives::HasVertexNormals
Standard_Boolean Graphic3d_ArrayOfPrimitives::HasVertexColors
Standard_Boolean Graphic3d_ArrayOfPrimitives::HasVertexTexels
```

or get the number of vertices, edges and bounds:

```
Standard_Integer Graphic3d_ArrayOfPrimitives::VertexNumber
Standard_Integer Graphic3d_ArrayOfPrimitives::EdgeNumber
Standard_Integer Graphic3d_ArrayOfPrimitives::BoundNumber
```

4.2.4 Text primitive

The OpenGL graphics driver uses advanced text rendering powered by FTGL library. This library provides vector text rendering, as a result the text can be rotated and zoomed without quality loss. *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::Text
(
    const Standard_CString AText,
    const Graphic3d_Vertex& APoint,
    const Standard_Real AHeight,
    const Quantity_PlaneAngle AAngle,
    const Graphic3d_TextPath ATp,
    const Graphic3d_HorizontalTextAlignment AHta,
    const Graphic3d_VerticalTextAlignment AVta,
    const Standard_Boolean EvalMinMax),
```

AText parameter is the text string, *APoint* is the three-dimensional position of the text, *AHeight* is the text height, *AAngle* is the orientation of the text (at the moment, this parameter has no effect, but you can specify the text orientation through the *Graphic3d_AspectText3d* attributes).

ATp parameter defines the text path, *AHta* is the horizontal alignment of the text, *AVta* is the vertical alignment of the text.

You can pass *Standard_False* as *EvalMinMax* 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.

```
void Graphic3d_Group::Text
(
    const Standard_CString AText,
    const Graphic3d_Vertex& APoint,
    const Standard_Real AHeight,
    const Standard_Boolean EvalMinMax)
void Graphic3d_Group::Text
(
    const TCcollection_ExtendedString &AText,
    const Graphic3d_Vertex& APoint,
    const Standard_Real AHeight,
    const Quantity_PlaneAngle AAngle,
    const Graphic3d_TextPath ATp,
    const Graphic3d_HorizontalTextAlignment AHta,
    const Graphic3d_VerticalTextAlignment AVta,
    const Standard_Boolean EvalMinMax)
void Graphic3d_Group::Text
(
    const TCcollection_ExtendedString &AText,
    const Graphic3d_Vertex& APoint,
    const Standard_Real AHeight,
    const Standard_Boolean EvalMinMax)
```

See the example:

```
// get the group
Handle (Graphic3d_Group) aGroup = Prs3d_Root::CurrentGroup (aPrs);

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

// add a text primitive to the structure
Graphic3d_Vertex aPoint (1, 1, 1);
aGroup->Text (Standard_CString ("Text"), aPoint, 16.0);
```

4.2.5 Materials

A *Graphic3d_MaterialAspect* is defined by:

- Transparency;
- Diffuse reflection - a component of the object color;
- Ambient reflection;
- Specular reflection - a component of the color of the light source;
- Refraction index.

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.

4.2.6 Textures

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

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

4.2.7 Shaders

OCCT visualization core supports GLSL shaders. Currently OCCT supports only vertex and fragment GLSL shader. Shaders can be assigned to a generic presentation by its drawer attributes (Graphic3d aspects). To enable custom shader for a specific AISShape in your application, the following API functions are 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 AISShape
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, Depth-cueing, 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 (note that only Z-clipping planes can work with the Phigs interface),
- 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*, *Phigs* and *math*.

```
//Create a default display connection
Handle(Aspect_DisplayConnection) aDisplayConnection = new Aspect_DisplayConnection();

//Create a Graphic Driver from the default Aspect_DisplayConnection
Handle(OpenGL_GraphicDriver) GD = new OpenGL_GraphicDriver (aDisplayConnection);

//Create a Viewer to this Driver
Handle(V3d_Viewer) VM = new V3d_Viewer(GD, 400.,
    // Space size
    V3d_Xpos,
    // Default projection
    Quantity_NOC_DARKVIOLET,
    // Default background
    V3d_ZBUFFER,
    // Type of visualization
    V3d_GOURAUD,
    // Shading model
    V3d_WAIT;
    // Update mode
// Create a structure in this Viewer
Handle(Graphic3d_Structure) S = new Graphic3d_Structure(VM->Viewer());

// Type of structure
S->SetVisual (Graphic3d_TOS_SHADING);

// Create a group of primitives in this structure
Handle(Graphic3d_Group) G = new Graphic3d_Group(S);

// Fill this group with one polygon of size 100
Graphic3d_Array1OfVertex Points(0,3);
Points(0).SetCoord(-100./2.,-100./2.,-100./2.);
Points(1).SetCoord(-100./2., 100./2.,-100./2.);
Points(2).SetCoord( 100./2., 100./2.,-100./2.);
Points(3).SetCoord( 100./2.,-100./2.,-100./2.);
Normal.SetCoord(0.,0.,1.);
G->Polygon(Points,Normal);

// Create Ambient and Infinite Lights in this Viewer
Handle(V3d_AmbientLight) L1 = new V3d_AmbientLight
    (VM,Quantity_NOC_GRAY50);
Handle(V3d_DirectionalLight) L2 = new V3d_DirectionalLight
    (VM,V3d_XnegYnegZneg,Quantity_NOC_WHITE);
```

```
// Create a 3D quality Window with the same DisplayConnection
Handle(Xw_Window) W = new Xw_Window(aDisplayConnection, "Test V3d", 0.5, 0.5, 0.5, 0.5) ;

// Map this Window to this screen
W->Map() ;

// Create a Perspective View in this Viewer
Handle(V3d_View) aView = new V3d_View(VM);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Perspective);
// Associate this View with the Window
aView ->SetWindow(W);
// Display ALL structures in this View
VM->Viewer()->Display();
// Finally update the Visualization in this View
aView->Update();
```

As an alternative to manual setting of perspective parameters the `V3d_View::ZfitAll()` and `V3d_View::FitAll()` functions can be used:

```
// Display shape in Viewer VM
Handle(AIS_InteractiveContext) aContext = new AIS_InteractiveContext (VM);
aContext->Display(shape);
// Create a Perspective View in Viewer VM
Handle(V3d_View) V = new V3d_View (VM);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Perspective);
// Change Z-min and Z-max planes of projection volume to match the displayed objects
V->ZFitAll();
// Fit view to object size
V->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, 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)), 30.0);
aView->Camera()->Transform (aTrsf);
```

4.4.4 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 (VM);
aView->Camera()->SetProjectionType (Graphic3d_Camera::Projection_Orthographic);
// update the Visualization in this View
aView->Update();
```

4.4.5 Perspective Projection

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

The following code configures the camera for perspective rendering:

```
// Create a perspective View in this Viewer
Handle(V3d_View) aView = new V3d_View(VM);
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:

- *IODType_Absolute* : Intraocular distance is defined as an absolute value.
- *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.

To enable stereo projection, 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(VM);
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();
```

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::CalculateBoundingBox()* 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 *OpenGL_BVHClipPrimitiveSet* and *OpenGL_BVHTreeSelector* 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 Underlay and overlay layers management

In addition to interactive 3d graphics displayed in the view you can display underlying and overlying graphics: text, color scales and drawings.

All *V3d* view graphical objects in the overlay are managed by the default layer manager (*V3d_LayerMgr*). The *V3d* view has a basic layer manager capable of displaying the color scale, but you can redefine this class to provide your own overlay and underlay graphics.

The method *V3d_View::SetLayerMgr(const Handle (V3d_LayerMgr)& aMgr)* allows assigning a custom layer manager to the *V3d* view.

There are three virtual methods to prepare graphics in the manager for further drawing: setting up layer dimensions and drawing static graphics. These methods can be redefined:

```
void V3d_LayerMgr::Begin ()
void V3d_LayerMgr::Redraw ()
void V3d_LayerMgr::End ()
```

The layer manager controls layers (*Visual3d_Layer*) and layer items (*Visual3d_LayerItem*). Both the overlay and underlay layers can be created by the layer manager.

The layer entity is presented by the *Visual3d_Layer* class. This entity provides drawing services in the layer, for example:

```
void Visual3d_Layer::DrawText
void Visual3d_Layer::DrawRectangle
void Visual3d_Layer::SetColor
void Visual3d_Layer::SetViewport
```

The following example demonstrates how to draw overlay graphics by the *V3d_LayerMgr*.

```
// redefined method of V3d_LayerMgr
void MyLayerMgr::Redraw ()
{
    Quantity_Color aRed (Quantity_NOC_RED);
    myOverlayLayer->SetColor (aRed);
    myOverlayLayer->DrawRectangle (0, 0, 100, 100);
}
```

The layer contains layer items that will be displayed on view redraw. Such items are *Visual3d_LayerItem* entities. To manipulate *Visual3d_LayerItem* entities assigned to the layer's internal list you can use the following methods:

```
void Visual3d_Layer::AddLayerItem (const Handle (Visual3d_LayerItem)& Item)
void Visual3d_Layer::RemoveLayerItem (const Handle (Visual3d_LayerItem)& Item)
void Visual3d_Layer::RemoveAllLayerItems ()
const Visual3d_NListOfLayerItem& Visual3d_Layer::GetLayerItemList ()
```

The layer's items are rendered when the method *void Visual3d_Layer::RenderLayerItems()* is called by the graphical driver.

The *Visual3d_LayerItem* has virtual methods that are used to render the item:

```
void Visual3d_LayerItem::RedrawLayerPrs ()
void Visual3d_LayerItem::ComputeLayerPrs ()
```

The item presentation can be computed before drawing by the *ComputeLayerPrs* method to save time on redraw. It also has an additional flag that is used to tell that the presentation should be recomputed:

```
void Visual3d_LayerItem::SetNeedToRecompute (const Standard_Boolean NeedToRecompute)
Standard_Boolean Visual3d_LayerItem::IsNeedToRecompute
```

An example of *Visual3d_LayerItem* is *V3d_ColorScaleLayerItem* that represents the color scale entity as the layer's item. The *V3d_ColorScaleLayerItem* sends render requests to the color scale entity represented by it. As this entity (*V3d_ColorScale*) is assigned to the *V3d_LayerMgr* it uses its overlay layer's services for drawing:

Example

```
// tell V3d_ColorScale to draw itself
void V3d_ColorScaleLayerItem::RedrawLayerPrs ()
{
    Visual3d_LayerItem::RedrawLayerPrs ()
    if (!MyColorScale.IsNull ())
        MyColorScale->DrawScale ();
}

// V3d_ColorScale has a reference to a LayerMgr
void V3d_ColorScale::DrawScale ()
{
    // calls V3d_ColorScale::PaintRect, V3d_ColorScale::PaintText, etc.
}

// PaintRect method uses overlay layer of LayerMgr to draw a rectangle
void V3d_ColorScale::PaintRect
    (const Standard_Integer X, const Standard_Integer Y,
     const Standard_Integer W, const Standard_Integer H,
     const Quantity_Color aColor,
     const Standard_Boolean aFilled)
{
    const Handle (Visual3d_Layer)& theLayer = myLayerMgr->Overlay ();
    ...
    theLayer->SetColor (aColor);
    theLayer->DrawRectangle (X, Y, W, H);
    ...
}
```

4.4.9 View background styles

There are three types of background styles available for *V3d_view*: solid color, gradient color and image.

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

```
void V3d_View::SetBackgroundColor
    (const Quantity_TypeOfColor Type,
     const Quantity_Parameter V1,
     const Quantity_Parameter V2,
     const Quantity_Parameter V3)
```

This method allows you to specify the background color in RGB (red, green, blue) or HLS (hue, lightness, saturation) color spaces, so the appropriate values of the *Type* parameter are *Quantity_TOC_RGB* and *Quantity_TOC_HLS*.

Note that the color value parameters *V1*, *V2*, *V3* should be in the range between 0.0-1.0.

```
void V3d_View::SetBackgroundColor(const Quantity_Color &Color)
void V3d_View::SetBackgroundColor(const Quantity_NameOfColor Name)
```

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

```
void V3d_View::SetBgGradientColors
    (const Quantity_Color& Color1,
     const Quantity_Color& Color2,
     const Aspect_GradientFillMethod FillStyle,
     const Standard_Boolean update)

void V3d_View::SetBgGradientColors
    (const Quantity_NameOfColor Color1,
     const Quantity_NameOfColor Color2,
     const Aspect_GradientFillMethod FillStyle,
     const Standard_Boolean update)
```

The *Color1* and *Color2* parameters define the boundary colors of interpolation, the *FillStyle* parameter defines the direction of interpolation. You can pass *Standard_True* as the last parameter to update the view.

The fill style can be also set with the method *void V3d_View::SetBgGradientStyle(const Aspect_GradientFillMethod AMethod, const Standard_Boolean update)*.

To get the current background color you can use the following methods:

```

void V3d_View::BackgroundColor
(const Quantity_TypeOfColor Type,
 Quantity_Parameter &V1,
 Quantity_Parameter &V2,
 Quantity_Parameter &V3)
Quantity_Color V3d_View::BackgroundColor()
void V3d_View::GradientBackgroundColors(Quantity_Color& Color1, Quantity_Color& Color2)
Aspect_GradientBackground GradientBackground()

```

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

```

void V3d_View::SetBackgroundImage
(const Standard_CString FileName,
 const Aspect_FillMethod FillStyle,
 const Standard_Boolean update)
void V3d_View::SetBgImageStyle
(const Aspect_FillMethod FillStyle,
 const Standard_Boolean update)

```

The *FileName* parameter defines the image file name and the path to it, the *FillStyle* 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.

4.4.10 Dumping a 3D scene into an image file

The 3D scene displayed in the view can be dumped in high resolution into an image file. The high resolution (8192x8192 on some implementations) is achieved using the Frame Buffer Objects (FBO) provided by the graphic driver. Frame Buffer Objects enable off-screen rendering into a virtual view to produce images in the background mode (without displaying any graphics on the screen).

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.

```

Standard_Boolean V3d_View::Dump
(const Standard_CString theFile,
 const Aspect_FormatOfSheetPaper theFormat,
 const Image_TypeOfImage theBufferType)

```

Makes the dimensions of the output image compatible to a certain format of printing paper passed by *theFormat* argument.

These methods dump the 3D scene into an image file passed by its name and path as *theFile*.

The raster image data handling algorithm is based on the *Image_Pixmap* class. The supported extensions are ".png", ".bmp", ".pnm", ".tga".

The value passed as *theBufferType* argument defines the type of the buffer for an output image (*RGB*, *RGBA*, *floating-point*, *RGBF*, *RGBAF*). Both methods return *Standard_True* if the scene has been successfully dumped.

There is also class *Image_AlienPixmap* providing import / export from / to external image files in formats supported by **FreeImage** library.

Note that dumping the image for a paper format with large dimensions is a memory consuming operation, it might be necessary to take care of preparing enough free memory to perform this operation.


```
Handle_Image_Pixmap V3d_View::ToPixmap
(const Standard_Integer theWidth,
 const Standard_Integer theHeight,
 const Image_TypeOfImage theBufferType,
 const Standard_Boolean theForceCentered)
```

Dumps the displayed 3d scene into a pixmap with a width and height passed as *theWidth* and *theHeight* arguments.

The value passed as *theBufferType* argument defines the type of the buffer for a pixmap (*RGB*, *RGBA*, *floating-point*, *RGBF*, *RGBAF*). The last parameter allows centering the 3D scene on dumping.

All these methods assume that you have created a view and displayed a 3d scene in it. However, the window used for such a view could be virtual, so you can dump the 3d scene in the background mode without displaying it on the screen. To use such an opportunity you can perform the following steps:

- Create display connection;
- Initialize graphic driver;
- Create a window;
- Set up the window as virtual, *Aspect_Window::SetVirtual()* ;
- Create a view and an interactive context;
- Assign the virtual window to the view;
- Display a 3D scene;
- Use one of the functions described above to dump the 3D scene.

The following example demonstrates this procedure for *WNT_Window* :

```
// create a dummy display connection
Handle(Aspect_DisplayConnection) aDisplayConnection;

// create a graphic driver
Handle (Graphic3d_GraphicDriver) aDriver = Graphic3d::InitGraphicDriver (aDisplayConnection);

// create a window
Standard_Integer aDefWidth = 800;
Standard_Integer aDefHeight = 600;
Handle (WNT_WClass) aWClass = new WNT_WClass ("Virtual Class",DefWindowProc,
                                             CS_VREDRAW | CS_HREDRAW, 0, 0,
                                             ::LoadCursor (NULL, IDC_ARROW));
Handle (WNT_Window) aWindow = new WNT_Window ("VirtualWnd", aWClass,
                                             WS_OVERLAPPEDWINDOW, 0, 0,
                                             aDefWidth, aDefHeight);

// set up the window as virtual
aWindow->SetVirtual (Standard_True);

// create a view and an interactive context
Handle (V3d_Viewer) aViewer = new V3d_Viewer (aDriver,
                                             Standard_ExtString ("Virtual"));
Handle (AIS_InteractiveContext) aContext = new AIS_InteractiveContext (aViewer);
Handle (V3d_View) aView = aViewer->CreateView ();

// assign the virtual window to the view
aView->SetWindow (aWindow);

// display a 3D scene
Handle (AIS_Shape) aBox = new AIS_Shape (BRepPrimAPI_MakeBox (5, 5, 5));
aContext->Display (aBox);
aView->FitAll();

// dump the 3D scene into an image file
aView->Dump ("3dscene.png");
```

4.4.11 Printing a 3D scene

The contents of a view can be printed out. Moreover, the OpenGL graphic driver used by the v3d view supports printing in high resolution. The print method uses the OpenGL frame buffer (Frame Buffer Object) for rendering the view contents and advanced print algorithms that allow printing in, theoretically, any resolution.

The method `void V3d_View::Print(const Aspect_Handle hPrnDC, const Standard_Boolean showDialog, const Standard_Boolean showBackground, const Standard_CString filename, const Aspect_PrintAlgo printAlgorithm)` prints the view contents:

`hPrnDC` is the printer device handle. You can pass your own printer handle or `NULL` to select the printer by the default dialog. In that case you can use the default dialog or pass `Standard_False` as the `showDialog` argument to select the default printer automatically.

You can define the filename for the printer driver if you want to print out the result into a file. If you do not want to print the background, you can pass `Standard_False` as the `showBackground` argument. The `printAlgorithm` argument allows choosing between two print algorithms that define how the 3d scene is mapped to the print area when the maximum dimensions of the frame buffer are smaller than the dimensions of the print area by choosing `Aspect_PA_STRETCH` or `Aspect_PA_TILE`.

The first value defines the stretch algorithm: the scene is drawn with the maximum possible frame buffer dimensions and then is stretched to the whole printing area. The second value defines `TileSplit` algorithm: covering the whole printing area by rendering multiple parts of the viewer.

Note that at the moment the printing is implemented only for Windows.

4.4.12 Vector image export

The 3D content of a view can be exported to the vector image file format. The vector image export is powered by the `GL2PS` library. You can export 3D scenes into a file format supported by the `GL2PS` library: PostScript (PS), Encapsulated PostScript (EPS), Portable Document Format (PDF), Scalable Vector Graphics (SVG), LaTeX file format and Portable LaTeX Graphics (PGF).

The method `void Visual3d_View::Export(const Standard_CString FileName, const Graphic3d_ExportFormat Format, const Graphic3d_SortType aSortType, const Standard_Real Precision, const Standard_Address ProgressBarFunc, const Standard_Address ProgressObject)` of `Visual3d_View` class allows exporting a 3D scene:

The `FileName` defines the output image file name and the `Format` argument defines the output file format:

- `Graphic3d_EF_PostScript` (PS),
- `Graphic3d_EF_EhnPostScript` (EPS),
- `Graphic3d_EF_TEX` (TEX),
- `Graphic3d_EF_PDF` (PDF),
- `Graphic3d_EF_SVG` (SVG),
- `Graphic3d_EF_PGF` (PGF).

The `aSortType` parameter defines `GL2PS` sorting algorithm for the primitives. The `Precision`, `ProgressBarFunc` and `ProgressObject` parameters are implemented for future uses and at the moment have no effect.

The `Export` method supports only basic 3d graphics and has several limitations:

- Rendering large scenes could be slow and can lead to large output files;
- Transparency is only supported for PDF and SVG output;
- Textures and some effects are not supported by the `GL2PS` library.

4.4.13 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 = Standard_True;
// enable specular reflections.
aParams.IsReflectionEnabled = Standard_True;
// enable adaptive anti-aliasing
aParams.IsAntialiasingEnabled = Standard_True;
// enable light propagation through transparent media.
aParams.IsTransparentShadowEnabled = Standard_True;
// update the view
aView->Update();
```

4.4.14 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.

4.4.15 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) aContext = ...
Handle(AIS_InteractiveObject) anInterObj = ...
Standard_Integer anId = 3;
aViewer->AddZLayer (anId);
aContext->SetZLayer (anInterObj, 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.

The corresponding method *SetZLayerOption (...)* is available in *Graphic3d_GraphicDriver* interface. You can get the options using getter from *Visual3d_ViewManager* and *V3d_Viewer*. It returns *Graphic3d_ZLayerSettings* cached in *Visual3d_ViewManager* for a given *LayerId*.

Example:

```
// change z-layer settings
Graphic3d_ZLayerSettings aSettings = aViewer->ZLayerSettings (anId);
aSettings.EnableSetting (Graphic3d_ZLayerDepthTest);
aSettings.EnableSetting (Graphic3d_ZLayerDepthWrite);
aSettings.EnableSetting (Graphic3d_ZLayerDepthClear);
aSettings.EnableSetting (Graphic3d_ZLayerDepthOffset);
aViewer->SetZLayerSettings (anId, aSettings);
```

4.4.16 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::InquirePlaneLimit()*:

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

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
const Handle(Graphic3d_ClipPlane)& aClipPlane = new Graphic3d_ClipPlane();
// change equation of the clipping plane
Standard_Real aCoeffA = ...
Standard_Real aCoeffB = ...
Standard_Real aCoeffC = ...
Standard_Real 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();

```

4.4.17 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_ToolShadedShape::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_MATRABLU);
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 (1.0);
aFirebrickMarker->SetType (Aspect_TOM_BALL);
// or this
// it is a preferred way (supports full-color images on modern hardware).
aFirebrickMarker->SetMarkerImage (theImage)

```

Create facet attributes.

```

Handle(Graphic3d_AspectFillArea3d) aFaceAspect = new Graphic3d_AspectFillArea3d();
Graphic3d_MaterialAspect aBrassMaterial (Graphic3d_NOM_BRASS);
Graphic3d_MaterialAspect aGoldMaterial (Graphic3d_NOM_GOLD);
aFaceAspect->SetInteriorStyle (Aspect_IS_SOLID);
aFaceAspect->SetInteriorColor (aMyColor);
aFaceAspect->SetDistinguishOn ();
aFaceAspect->SetFrontMaterial (aGoldMaterial);
aFaceAspect->SetBackMaterial (aBrassMaterial);
aFaceAspect->SetEdgeOn();

```

Create text attributes.

```

Handle(Graphic3d_AspectText3d) aTextAspect = new Graphic3d_AspectText3d (aForest, Graphic3d_NOF_ASCII_MONO,
1.0, 0.0);

```

4.5.2 Create a 3D Viewer (a Windows example)

```

// create a default connection
Handle(Aspect_DisplayConnection) aDisplayConnection;
// create a graphic driver from default connection
Handle(OpenGL_GraphicDriver) aGraphicDriver = new OpenGL_GraphicDriver (GetDisplayConnection());
// create a viewer
TCollection_ExtendedString aName ("3DV");
myViewer = new V3d_Viewer (aGraphicDriver,aName.ToExtString(), "");
// 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()*.

```

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);
```

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 *Prs3d_Presentation* 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(Prs3d_Presentation)& thePrs,
 const Standard_Integer theMode)
(
    //...
)

void
myPresentableObject::Compute (const Handle(Prs3d_Projector)& ,
                             const Handle(Prs3d_Presentation)& thePrs)
(
    //...
)
```

4.5.6 Create primitives in the interactive object

Get the group used in *Prs3d_Presentation*.

```
Handle(Graphic3d_Group) aGroup = Prs3d_Root::CurrentGroup (thePrs);
```

Update the group attributes.

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

Create two triangles in *aGroup*.

```
Standard_Integer aNbTria = 2;
Handle(Graphic3d_ArrayOfTriangles) aTriangles = new Graphic3d_ArrayOfTriangles (3 * aNbTria, 0,
    Standard_True);
Standard_Integer anIndex;
for (anIndex = 1; anIndex <= aNbTria; nt++)
{
    aTriangles->AddVertex (anIndex * 5., 0., 0., 1., 1., 1.);
    aTriangles->AddVertex (anIndex * 5 + 5, 0., 0., 1., 1., 1.);
    aTriangles->AddVertex (anIndex * 5 + 2.5, 5., 0., 1., 1., 1.);
}
aGroup->BeginPrimitives();
aGroup->AddPrimitiveArray (aTriangles);
aGroup->EndPrimitives();
```

The methods *BeginPrimitives()* and *EndPrimitives()* are used when creating a set of various primitives in the same group. 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->BeginPrimitives();
aGroup->AddPrimitiveArray(aPolylines);
aGroup->EndPrimitives();
```

Create text and markers in group *aGroup*.

```
static char* texte[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->BeginPrimitives();
aGroup->AddPrimitiveArray (aPtsArr);
aGroup->EndPrimitives();

Graphic3d_Vertex aMarker (0.0, 0.0, 0.0);
for (i=0; i <= 2; i++)
{
    aMarker.SetCoord (-(Standard_Real )i * 4 + 30,
                     (Standard_Real )i * 4,
                     -(Standard_Real )i * 4);
    aGroup->Text (texte[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 *SetDataSource()* 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).