TGS

Open Inventor

FROM TGS

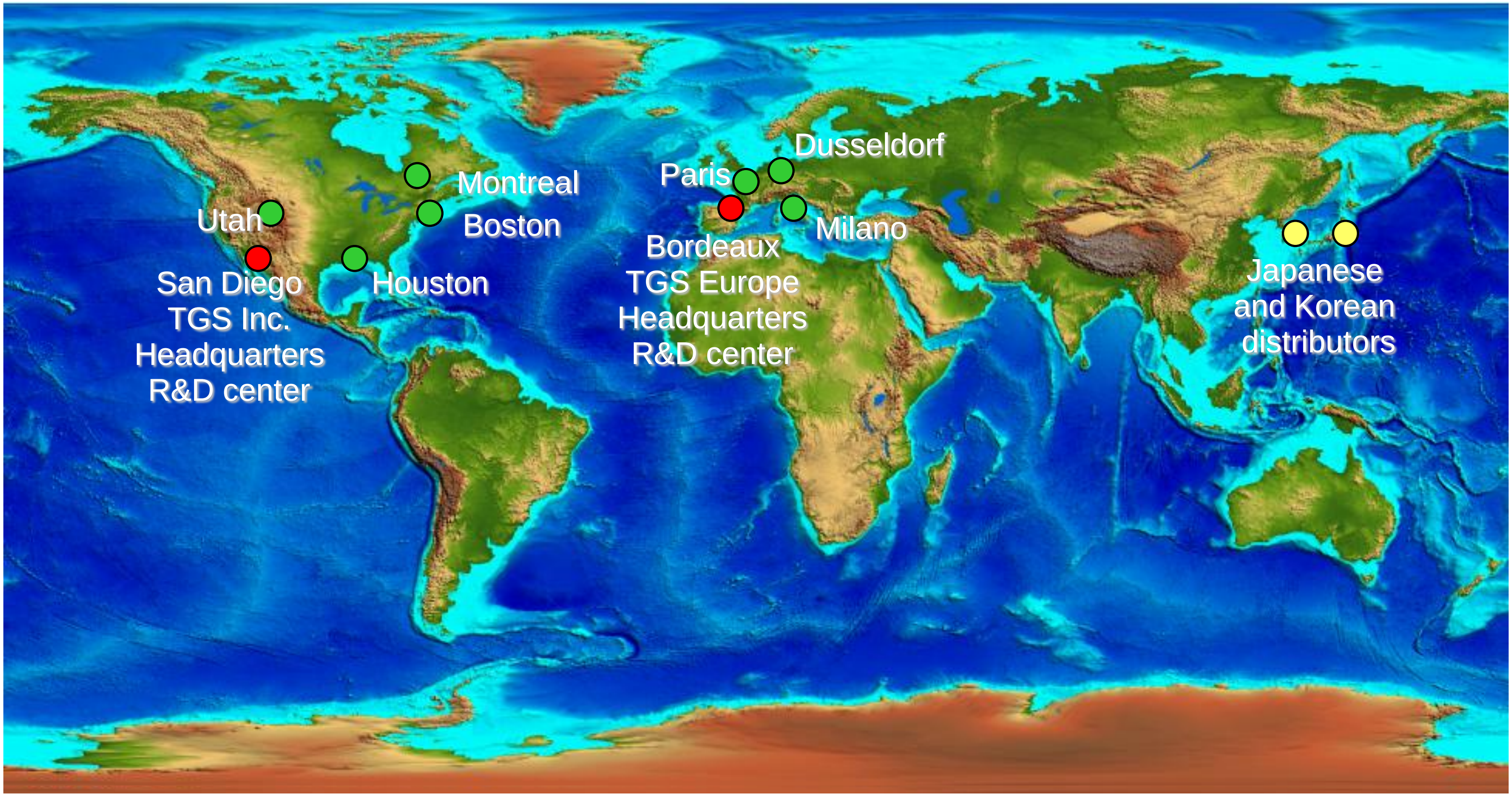
OBJECT-ORIENTED 3D GRAPHICS DEVELOPERS TOOLKIT

Trademarks

- **WebSpace** is a trademark of Silicon Graphics, Inc.
- **Open Inventor** is a trademark of Silicon Graphics, Inc.
- **OpenGL** is a registered trademark of Silicon Graphics, Inc.
- **TGS** is a source licensee of OpenGL, Open Inventor, and WebSpace from Silicon Graphics, Inc.
- **Windows** is a registered trademark of Microsoft.
- All other trademarks belong to their companies.

Training Contents

Introducing TGS



TGS products

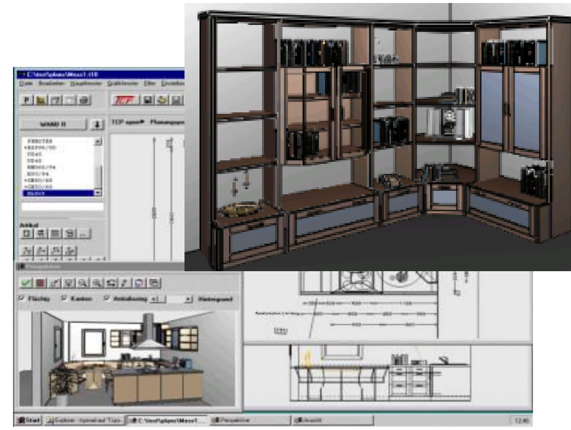
- ISO standard libraries **GKS, PHIGS**
  - Cross-platform graphics for C/FORTRAN
- **Open Inventor, DataViz (3DMasterSuite)**
  - Cross-platform graphics for C++/Java
  - Teamed with SGI to create VRML
- **Amira**
  - Interactive data visualization workbench
- 3D media authoring
  - **Amapi** intuitive modeler, **Carrara studio**…

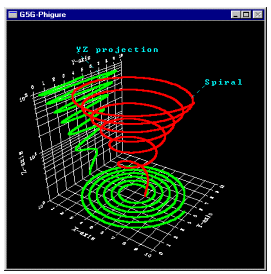
eovia
A TGS COMPANY

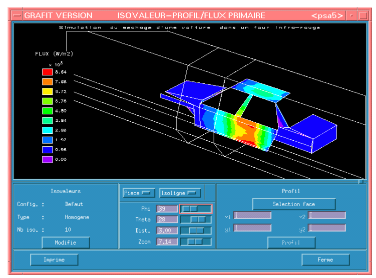
- **Professional Services**
  - Training
  - Assistance
  - Expertise
  - Custom components
  - Custom applications
  - Partnership with contractors and software vendors

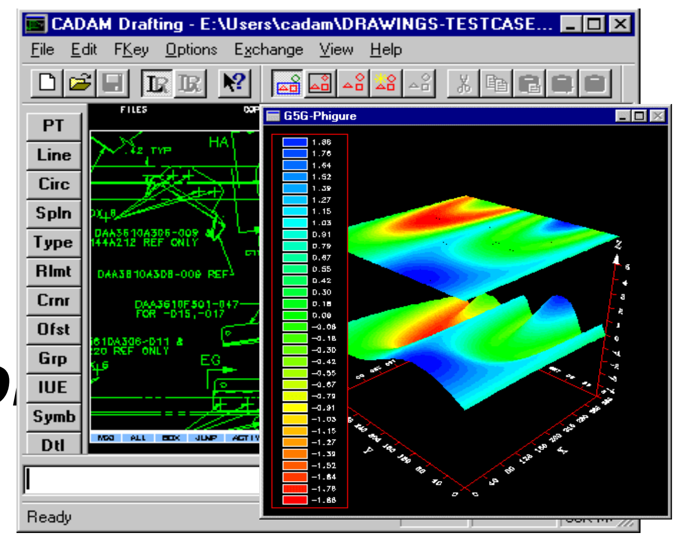
Solution for C or FORTRAN developers

- **GPHIGS, GKSBx**
  - 2D/3D portable structured graphics
- **PHIGURE** & **GPHIGS_GUI**
  - Data visualisation
  - User interface components and dev tools





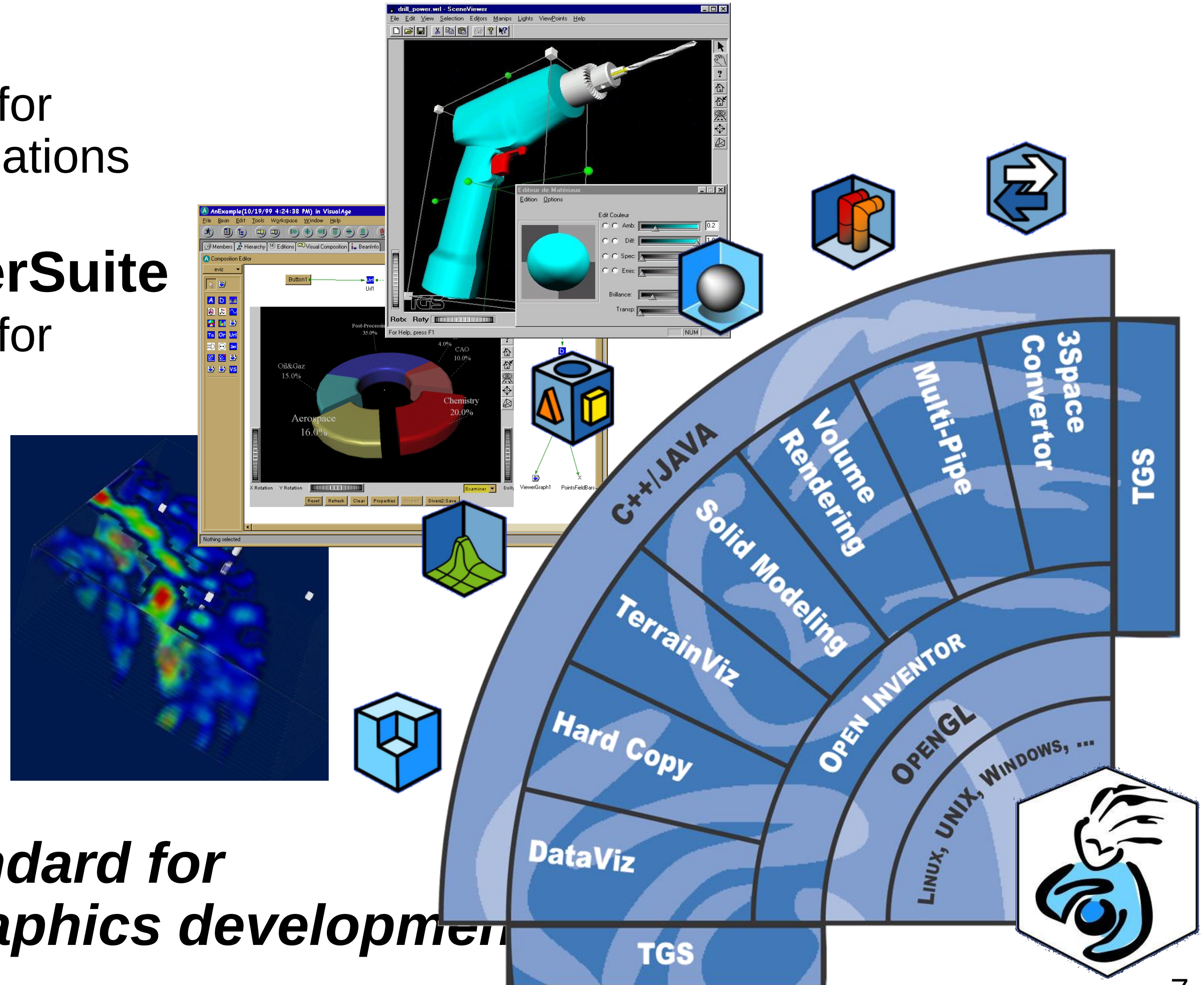




- ***Norms for long lifetime applicatio*** ***with traditional developpement***

Solution for C++ or Java developers

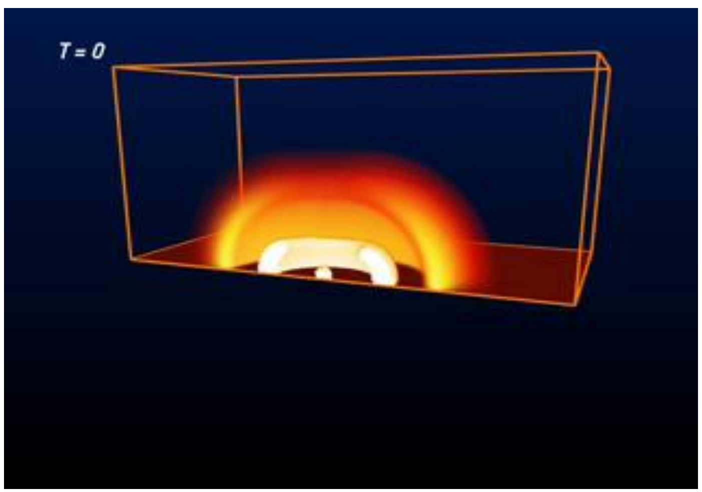
- **Open Inventor**
  - Rapid development for 3D interactive applications
- **DataViz / 3D-MasterSuite**
  - Rapid development for data visualisation
- **Volume Rendering**
- **SolidViz**
- **TerrainViz**

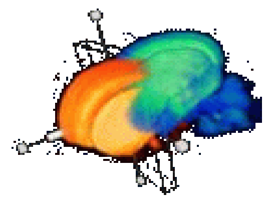


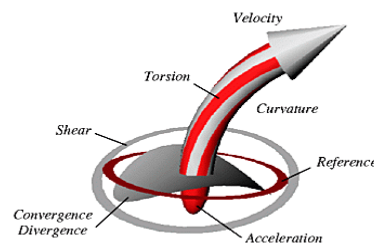
- ***The reference standard for object oriented graphics development***

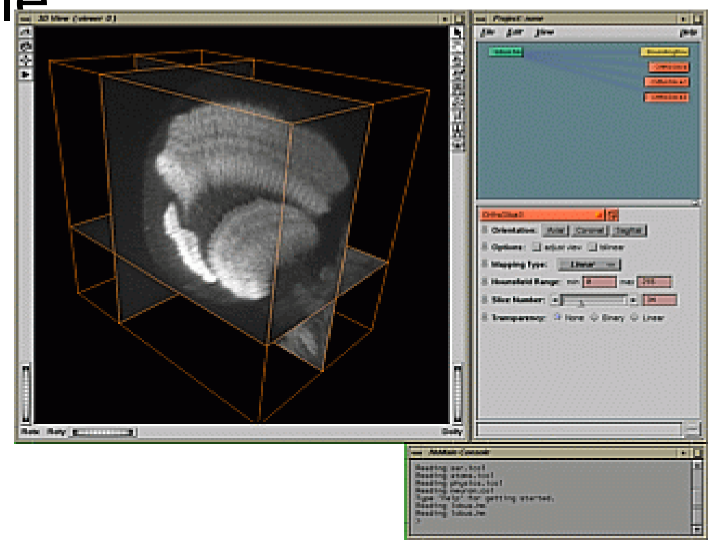
amira
Solution for non-developer

- **amira**
  - Modular interactive visualisation workbench
  - Segmentation, surface and volume reconstruction
  - Direct volume rendering...
  - Powered by Open Inventor, extensible









- ***State-of-the-art data visualization***

TGS Solutions

APPLICATION

Amira

Applications - components – plugins
ActiveX - applets - Java beans

C++ Java

API

Open Inventor
DataViz/3D-MasterSuite
Volume Rendering,
SolidViz, TerrainViz ...

SPI

OpenGL PostScript CGM HPGL GDI GIF JPEG PNG HTML VRML STEP IGES ...

HW

PC Windows Linux Unix Sun HP IBM SGI Digital 32/64bits...

Open Inventor: the standard

- Easy to learn and use
- High performance 3D applications
- High level standard API for OpenGL
- Large community of developers (news://comp.graphics.api.inventor)
- “.iv” format integrated into CAD applications
- Includes VRML 1.0 native and VRML 2.0 nodes
- Designed by Silicon Graphics

- Robust implementation on top of OpenGL
- Unique integration with Windows environment and Windows frameworks.

The effects and advantages of Open Inventor

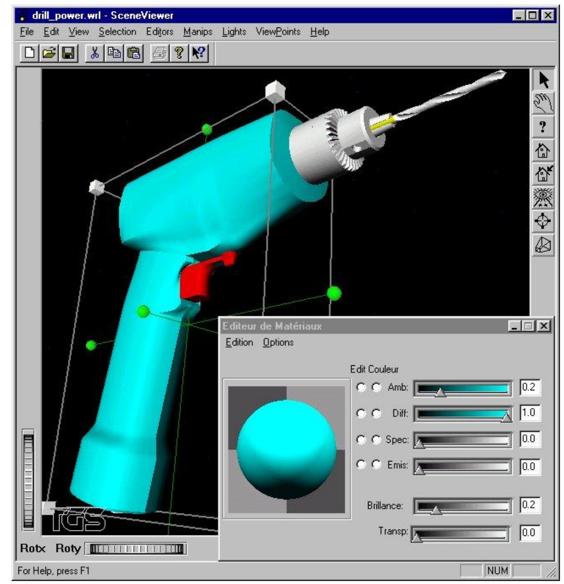
- 3D “everywhere”
  - Make 3D integration by developers trivial
  - Do this with a seamless path to acceleration
  - Deliver this in an “open” framework
- Desktop 3D communications (WebSpace)
- Desktop 3D documentation (3Space product line)
- C++ Class library
  - Rapid development, less code, fewer bugs, faster enhancements and maintanance
- Cross-platform API
- Standard data interchange format
  - For interoperability between applications
- Extensible framework
  - For customization and non-graphics integration

Open Inventor :
3D programming for humans

- Designed for ease of use
  - Flexible API for fast development
  - Dramatically simplifies OpenGL development

- Interaction, animation
  - Selection, ray picking
  - Draggers, manipulators

- Interface components
  - Viewers: fly, walk, examiner, plane
  - Editors : color, material, lighting
  - SceneViewer



- ***The open framework for rapid development with more than 450 classes***

Open Inventor background

Open Inventor 3.1 from TGS
2002

Open Inventor 3.0 from TGS
2001

Open Inventor 2.6 from TGS
2000

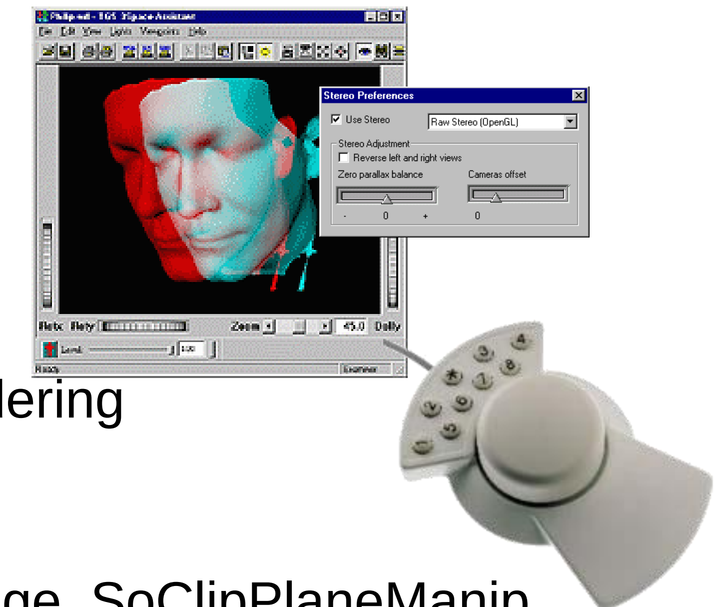
Open Inventor 2.5 from TGS
• • • 1999

Open Inventor 2.1
1994

Inventor 2.0
1992

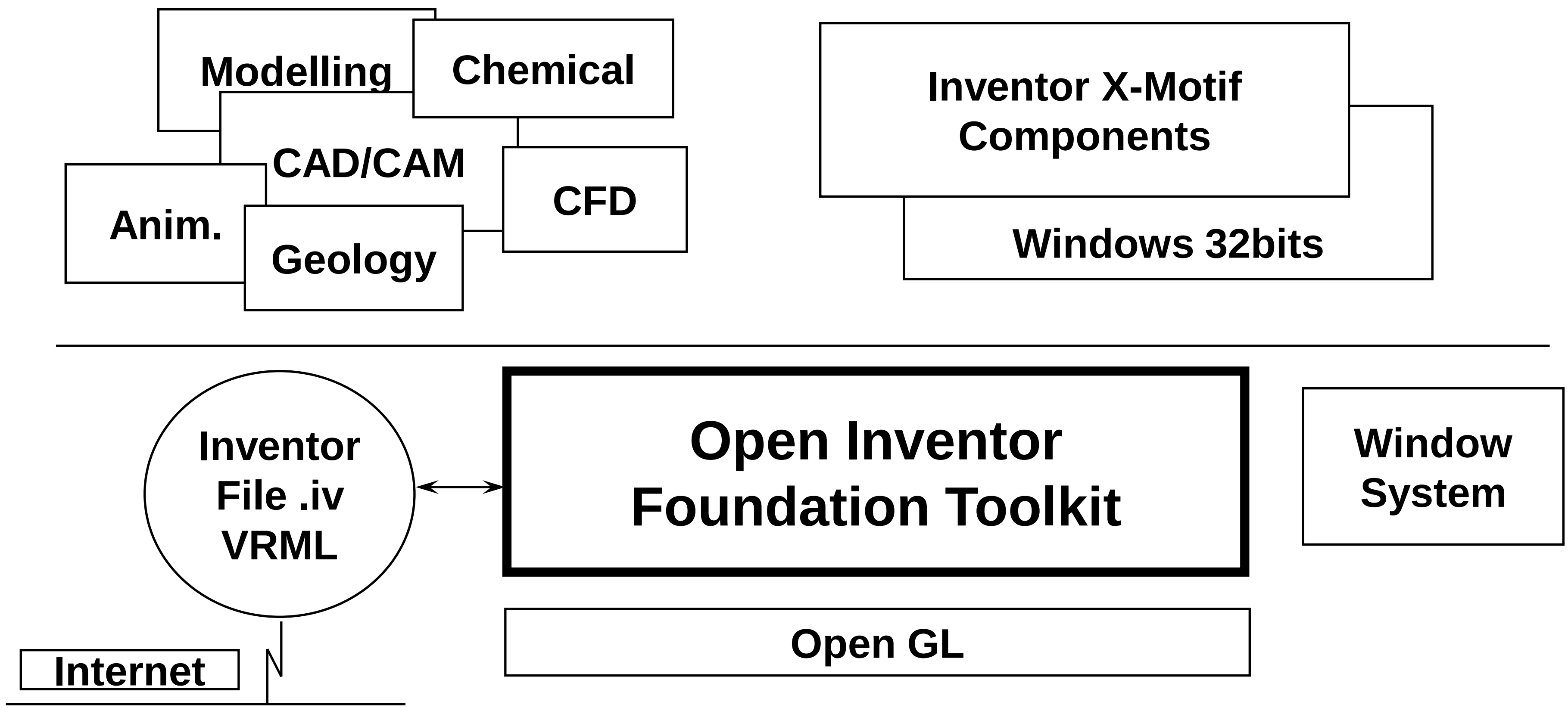
Open Inventor from 2.1 to 3.1

- VRML2/VRML 97 extensions
- SoWin & IFV user interface components for Windows
- Large Model Viewing : level of simplification, spatial optimisation, adaptive viewing, texture loading, pass by reference…
- Collision detection
- Re-written fast and robust NURBS engine
- Extended selection : box & lasso
- Extended input/output formats :
  PNG, BMP, JPEG, TIFF, HTML, VRML, ZAP…



- Enhanced stereo, multi-pipe support, remote rendering
- Double precision math classes
- 3D textures, polygon offset, vertex array
- Enhanced text, stroke fonts, SoMarkerSet, SoImage, SoClipPlaneManip, patterns…
- Multi-threading support, remote rendering…
- New extensions

Open Inventor Architecture : an extensible framework



Open Inventor for Java overview

- Open Inventor for Java(TM) (formerly 3D-MasterSuite for Java) provides an interface to the popular object oriented toolkits Open Inventor and extensions from TGS.

- Open Inventor for Java allows you to use the very powerful 3D Open Inventor features across the different supported platforms with Java applets and html browsers.

- Open Inventor for Java uses the native method of Open Inventor C++ to access the hardware resources. Thus Open Inventor for Java is able to access 3D performances of the underlying hardware through Java.

Open Inventor for Java architecture

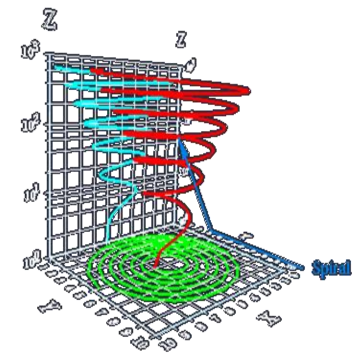
Open Inventor for Java is provided as a set of Java(TM) packages : com.tgs.inventor.*, com.tgs.dataviz.*

Open Inventor for Java contains two main parts

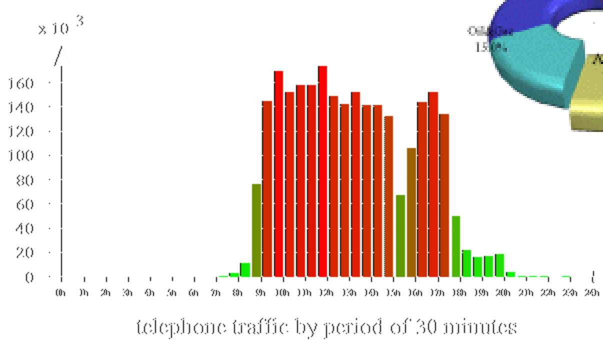
- The Java(TM) version of the *core API* is very similar to the standard C++ API.

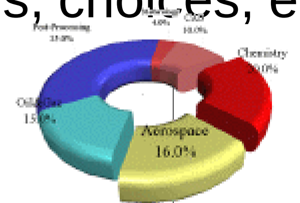
- The *component* library, known as Inventor Xt or SoXt, has been completely replaced by a pure AWT interface. This package uses some of the most powerful Java(TM) concepts to provide users with a very simple to use and efficient way to build complex applications using 3D graphics in Java(TM). From the basic Drawing Area to the high level Viewers, *com.tgs.inventor.awt* contains all the necessary tools to facilitate application developments.

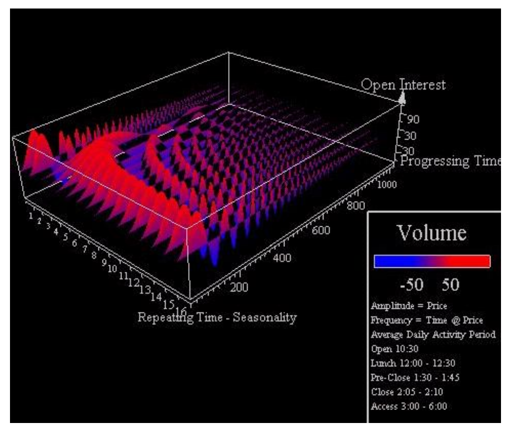
DataViz/3D-MasterSuite
Rapid visualization

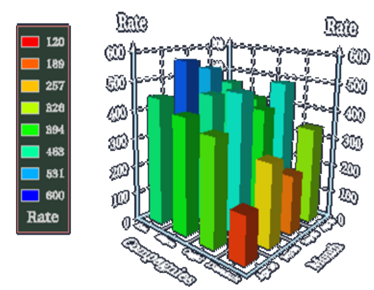


- **GraphMaster**
  - 2D/3D curves, histograms, pie charts, statistic charts...
  - Comprehensive legends & axis : linear, log, time, polar…
- **HardCopy**
  - Vector rendering for PostScript, CGM, HPGL, GDI/EMF...
- **DialogMaster** : portable user interface components (UNIX/Win32 C++)
  - sliders, triggers, choices, editable texts, labels, axis & legend editors...

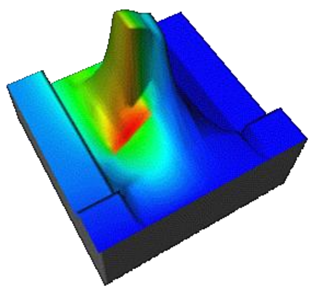


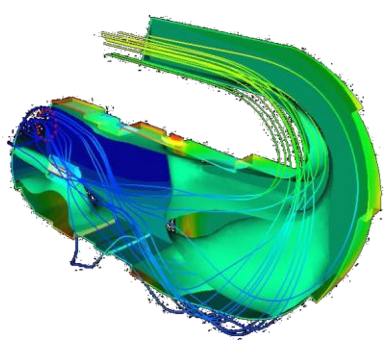


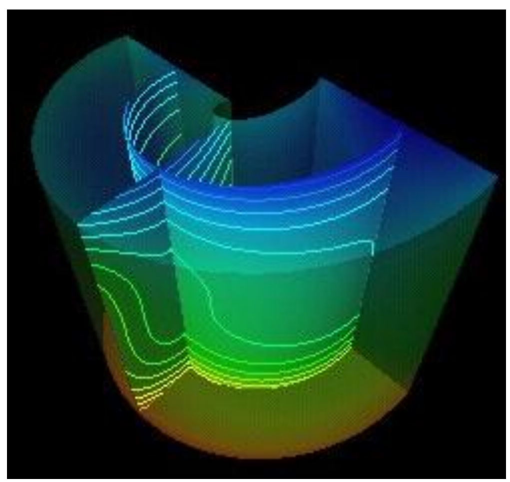


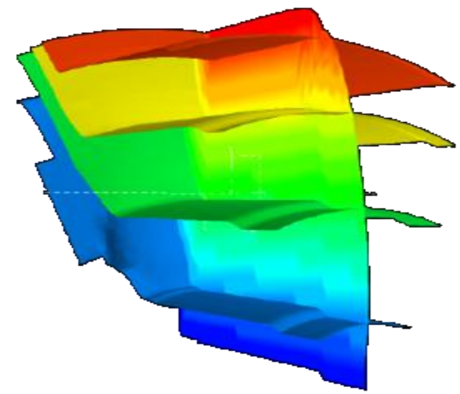


- **3DDataMaster**
  2D/3D meshes, annotated contouring, legends, cross-sections, isosurfaces, vector fields, streamlines, particle advection, probes...



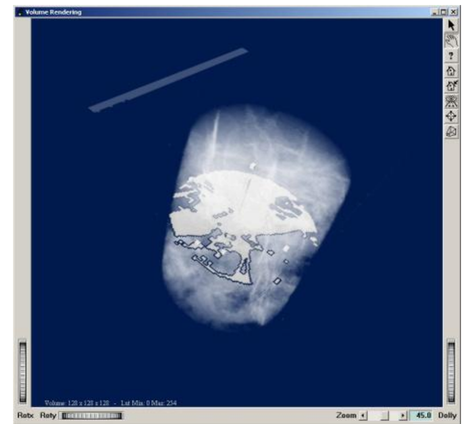


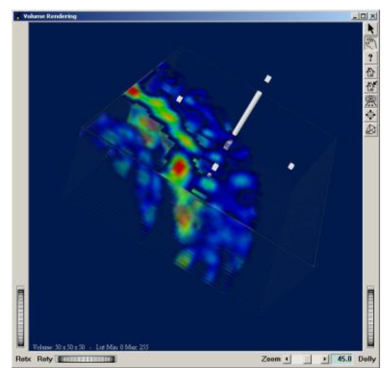


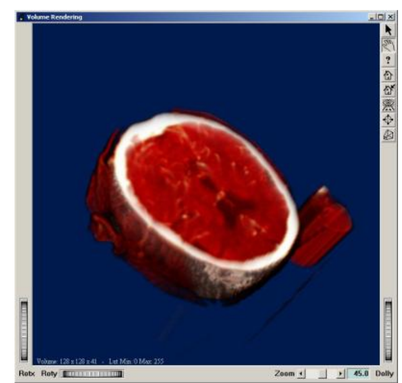


Volume Rendering for Open Inventor

- Full featured volume rendering option
  - Ortho slice, Oblique slice, Clipping
  - Voxel rendering, geometry mixable
  - Hardware acceleration support
    - 2D, 3D textures and VolumePro
  - Max intensity, sum intensity, alpha blending
  - Transfer function
    - Histogram, pre-defined color ramps
    - Real-time change with paletted texture
  - Byte and short data
  - Subsetting, Resampling, ROI
  - Data lighting
  - Cross platform
  - C++



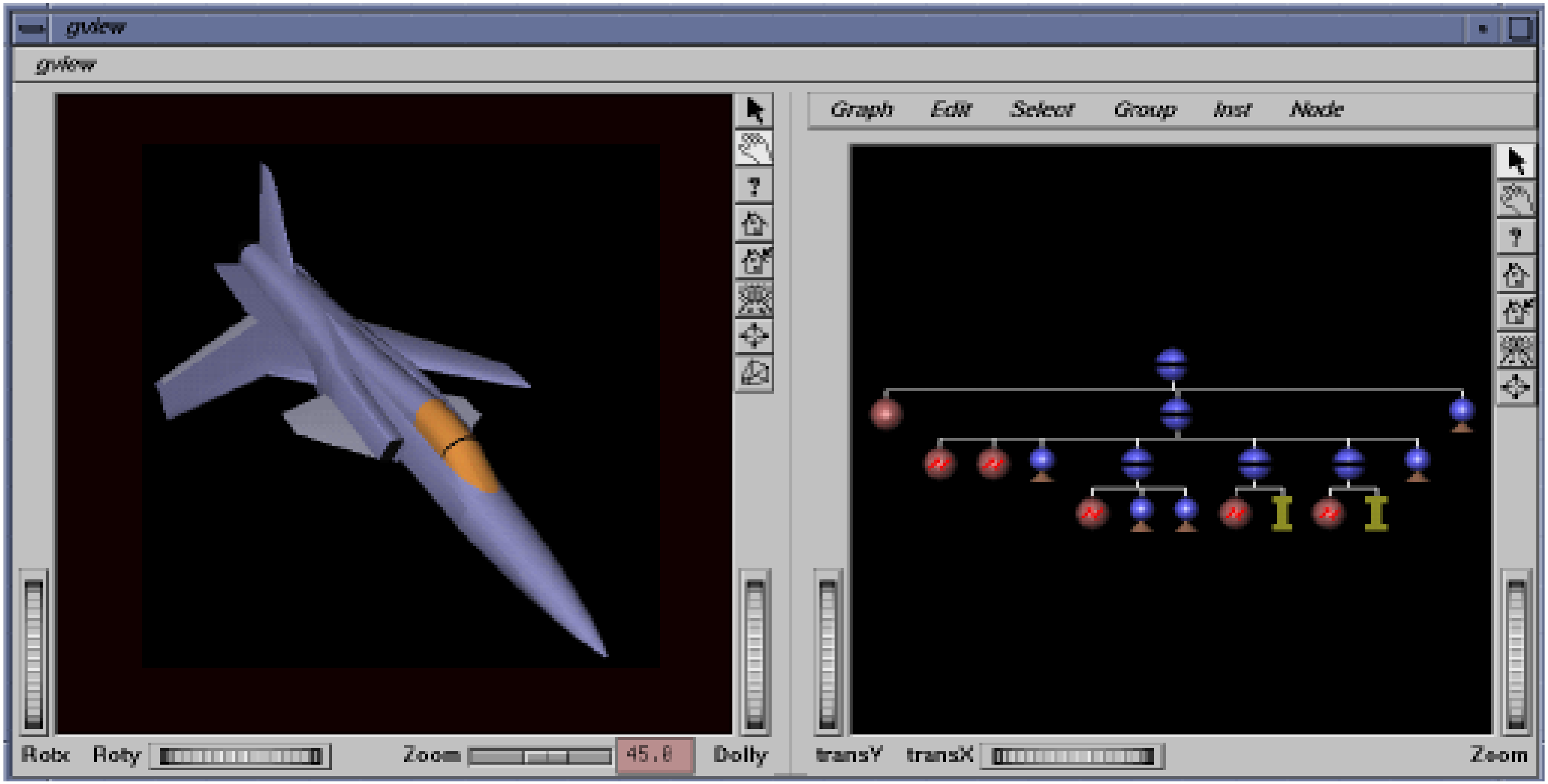




CAD/CAM converters for Open Inventor

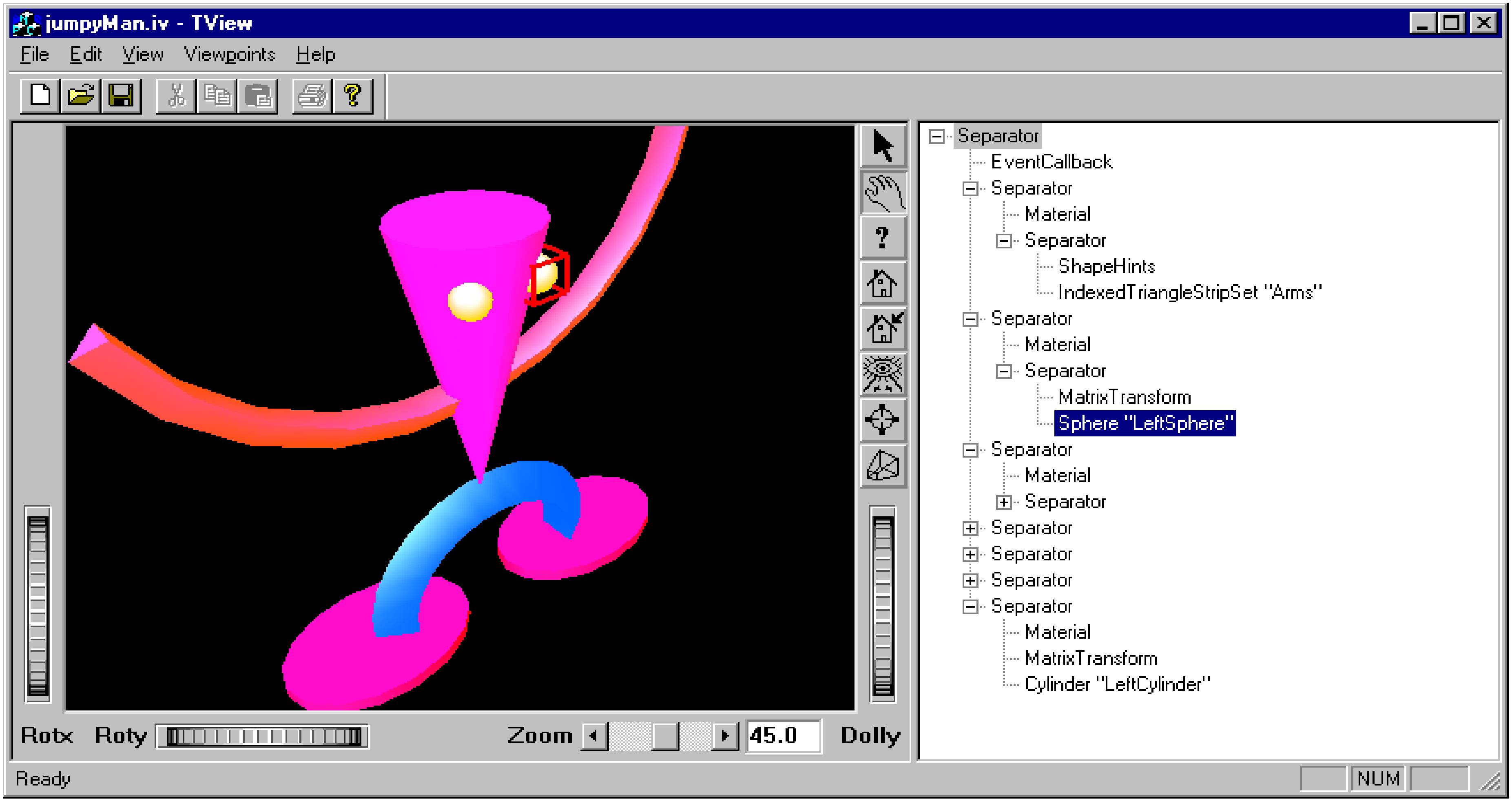
- Input data formats
    - IGES 5.1
    - VDA-FS (automotive profile)
    - STL Ascii (prototyping)
    - DXF R14
    - STEP AP 214 CC1
- Solution to convert and visualize 3D CAD/CAM models
    - User interface for Windows
    - Batch commande
    - Commande line integration
- Robust NURBS support for CAD

Open Inventor scene graph
Gview



$OIVHOME/src/demos/GView

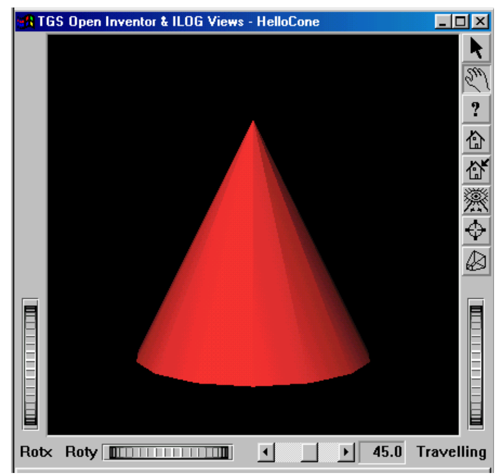
Open Inventor scene graph
TreeView



%OIVHOME%\src\examples\ÌVF\TreeView

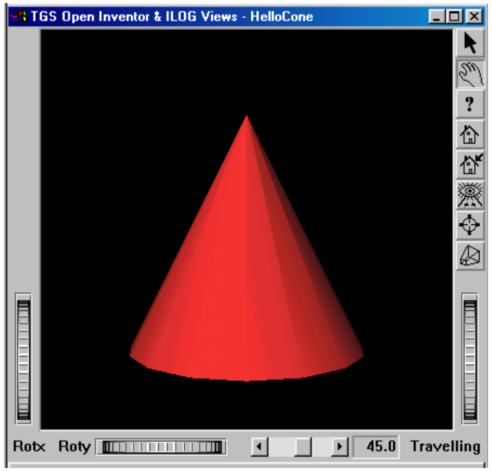
Open Inventor simple example
HelloCone.cpp

```
main() {
    // Initialise Open Inventor and Create a viewer
    Widget topLevelWindow = SoXt::init();
    SoXtExaminerViewer *examinerViewer =
      new SoXtExaminerViewer(topLevelWindow);

    // Create a simple cone
    SoCone* cone = new SoCone();

    // Put our scene in the viewer
    examinerViewer->setSceneGraph(cone);

    // Make viewer visible
    examinerViewer->show();

    // Start event loop
    SoXt::mainLoop();
}
```

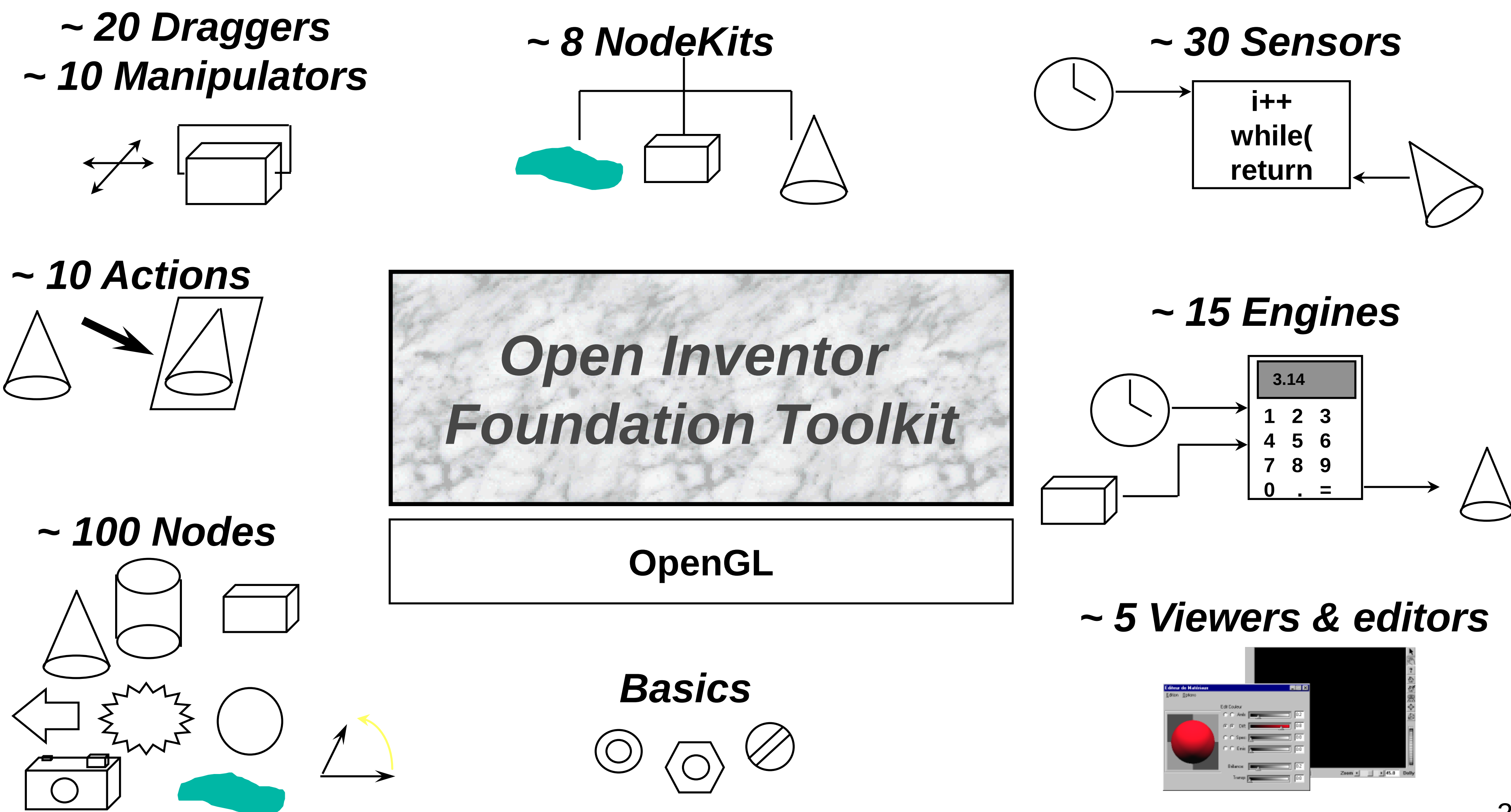


Open Inventor simple example
HelloCone.java

```
public static void main(String argv[]) {
   // Create a viewer
   SwSimpleViewer viewer = new SwSimpleViewer();

   // Create a simple cone
   SoCone cone = new SoCone();

   // Put our scene in the viewer and set it visible
   viewer.setSceneGraph(cone);

   Panel panel = new Panel(new BorderLayout()) ;
   panel.add(viewer);

   Frame f = new Frame ("HelloCone");
   f.add(panel);
   f.pack();
   f.show();
 }
```



Open Inventor classes



Some Open Inventor nodes (1/2)

- **Shape nodes**
  - Cone, Cube, Cylinder, Sphere, Text2, Text3, NurbsSurface, NurbsCurve, FaceSet, LineSet, PointSet, QuadMesh, TriangleStripSet and more...
- **Property nodes**
  - Material, Texture2, DrawStyle, Font, LightModel, Coordinate3, Complexity, Normal, Environment, PickStyle, Transform, Translation...
- **Groups**
  - Annotation, Array, File, Group, LOD, Selection, Separator, Switch, TransformSeparator, WWWAnchor, WWWInline
- **Lights**
  - Directional, Point, Spot
- **Cameras**
  - Perspective and Orthographic

Some Open Inventor nodes (2/2)

- **Events**
  - EventCallback nodes
- **Node kits**
  - AppearanceKit, CameraKit, LightKit, Draggers, SceneKit, SeparatorKit, ShapeKit, WrapperKit
- **Engines**
  - Rotor, Pendulum, Calculator… : can be embedded in iv files !
- **Manipulators**
  - Centerball, Trackball, HandleBox, Jack, DirectionalLight, PointLight and SpotLight manipulators
- **Sensors … are not nodes**
  - Alarm, Idle, OneShot, Timer, Field, Node, Path : use callback function as action

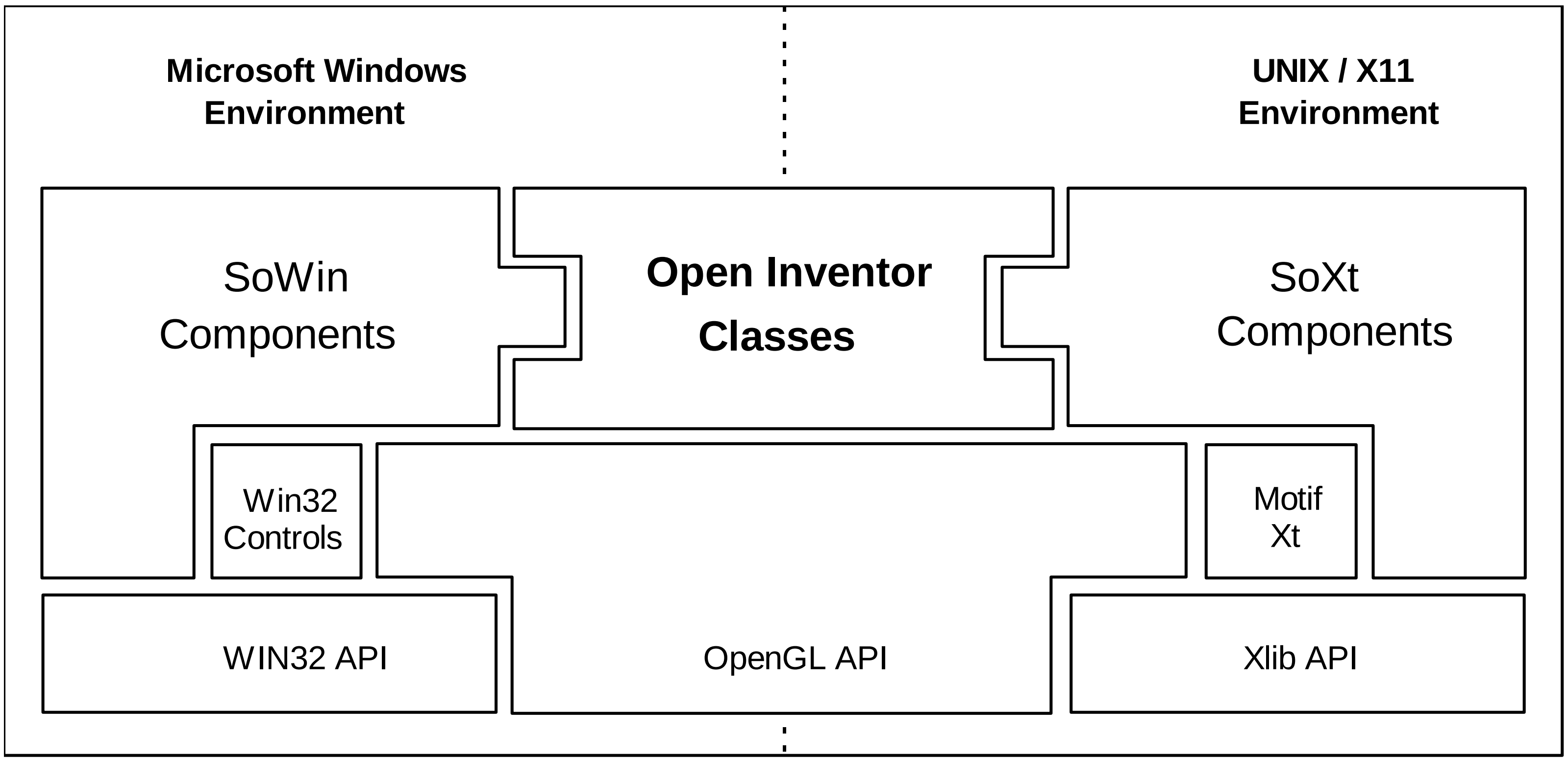
The Component Library (1/2)

- The Component Library includes :
  - Render area object (window)
  - Main loop and initialization convenience routines
  - Event translator utility
  - Editors (material, color, lights...)
  - Viewers (examiner, fly, walk, ...)
- The Open Inventor Component Library gives you access to an easy way to integrate your application in your windowing environment and also an easy and efficient way to deal with basic manipulation of your scene and your model.
- A good implementation must reflect the standard look and feel of application running on the platform, so the user does not have to adapt to a different style of interactivity.

The Component Library (2/2)

- On Unix platforms, libSoXt is based on libXt and gives a Motif look and feel.
- Using libSoXt will give your application the same look and feel on all Motif-based platforms.
- Open Inventor translates XEvents to system-independent SoEvents.
- On Windows 95 and NT platforms Open Inventor provides:
  - SoXt clone: This Component Library allows you to develop portable cross-platform applications between Unix and Windows.
    - Same programing interface
    - Windows look and feel
  - SoWin: The same functionality as SoXt but for specific Windows development. Do not use it if your application has to be ported to a platform other than Windows.

SoXt/SoWin Components Architecture

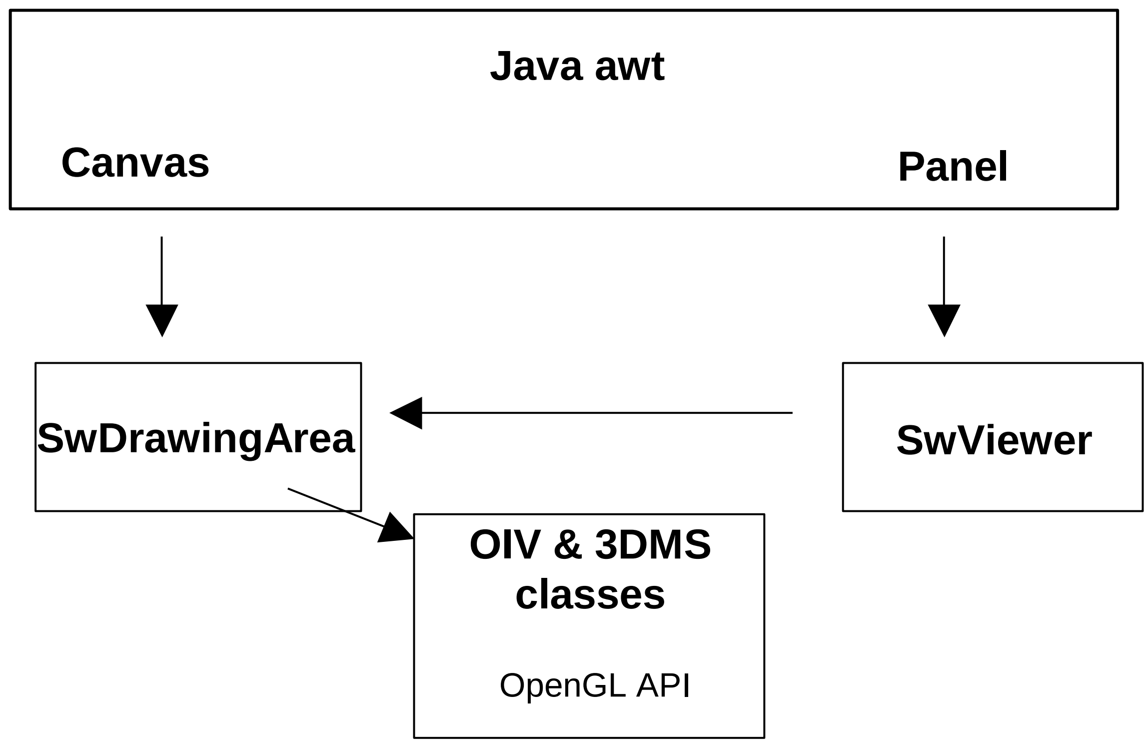


Sw : Open Inventor for Java AWT package

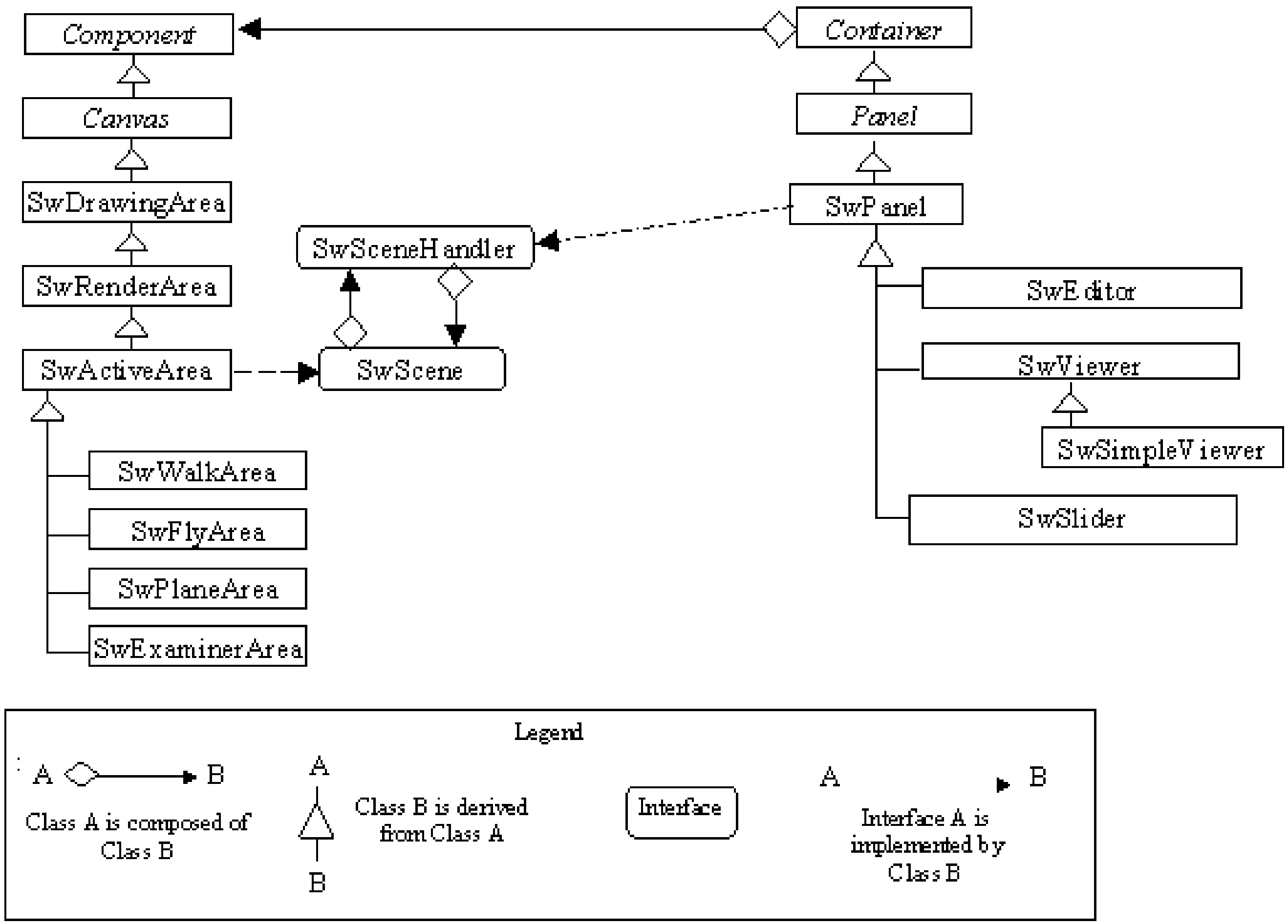
- Open Inventor for Java contains a package *com.tgs.inventor.awt* similar to C++ Open Inventor component library (SoXt/SoWin) to use the Java(TM) to handle user and window system actions.

- The main features are the following :
  - A Render Area (*SwRenderArea*) . This is the basic graphic window. It accepts events, translates them into an Inventor event and then passes it to smart objects such as manipulators that may handle the event. It performs OpenGL rendering.
  - Viewers (Example: *SwSimpleViewer*). This kind of object is a complete application with a lot of features like moving the object with the mouse, thumb wheels and slider trim at the sides, pop-up menu controlling display options, viewer icons.
  - Editors. This type of component provide some 3D related editing function, for example *SwDirectionalLightEditor* is used to edit a SoDirectionalLight node.

Sw Components Architecture



Sw Open Inventor for Java AWT class tree

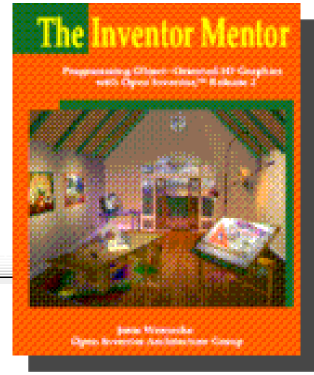


Cross-platform GUI with Open Inventor from TGS

- Open Inventor DialogMaster
  - instant portability between Window and Unix
- Open Inventor with Motif
  - To Windows :
Hummingbird Exceed (Exceed 3D extension available)
Datafocus Nutcracker
Softway's OpenNT-Interix
- Open Inventor with Windows MCF/IVF
  - To UNIX : Wind/U
- Open Inventor with Qt, Tcl/Tk, etc…
  - See example package in src/examples/techniques/Qt/
- Open Inventor for Java with AWT/Swing

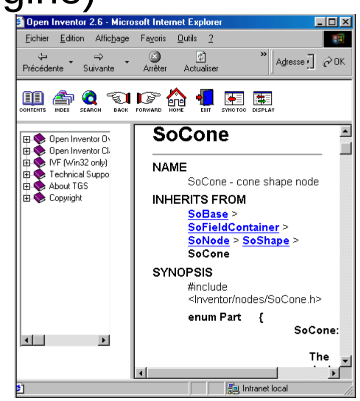
Cross-platform solution

- Hints to limit porting problems when developing an Open Inventor application:
  - Use a 100% portable implementation of Open Inventor.
  - Use a 100% compliant implementation of OpenGL if the application is dealing with direct callbacks to OpenGL
  - Use a hardware platform compatible with the application needs for performance both for CPU and graphic output purposes.
  - If Open Inventor is extended with your own sub-classes, be sure to write portable code not dependent on your development platform.
  - Use the Open Inventor Component Library SoXt for window system integration if you will need to port your application to different windowing systems.



Documentation set and starting points

- Printed books :
  - The **Inventor Mentor** : Open Inventor and 3D basics (as of version 2.0)
  - The **Inventor Toolmaker** : advanced use, how to build extensions (2.0)
  - The Open Inventor from TGS **User Guide** : the mentor update for Open Inventor 3.1, including optimization guide, VRML, large model viewing, Open Inventor for Win32, extensions. DataViz chapter is the recommended starting point for charting and scientific data visualisation users.
  - The Inventor C++ Reference - the green book (as of version 2.0)
    Warning : some compatibility breaks between 2.0 and 2.1, check “Open Inventor 2.1 porting guide” (in TGS User Guide)
- Online :
  - Check Open Inventor **installation** and **release notes**
  - Open Inventor **C++ Reference (Windows help/html** pages with search engine)
  - **Open Inventor Reference**
  - **Open Inventor from TGS User Guide** (pdf)
  - **Open Inventor for Java** online documentation



- Internet resources : www.tgs.com
  - news://comp.graphics.api.inventor
  - http://www.motifzone.com/tmd/articles/OpenInventor/OpenInventor.html
  - http://www.cs.brown.edu/~lsh/oivnet.html
  - http://www.ieighty.net/~davepamn/oiv1.html
  - http://www-ci.u-aizu.ac.jp/OpenInventor/Workshop/
  - http://www.sgi.com/Technology/Inventor/VRML/TIMSummary.html

Application skeleton

- Most applications will have nearly the same skeleton:
  - Import Java packages or include C++ headers
  - Create database
  - Create user interface
  - Define viewing environment
  - Display database
  - Deal with Interaction (interaction loop)
    - Modify database
    - Update display

Naming conventions

- Open Inventor basic types begin with **Sb**
  - SbVec3f… (*3f* stands for 3 float coordinates)
- Open Inventor other classes begin with **So**
  - SoNode…
- Open Inventor C++ user interface components begin with **SoXt** or **SoWin**
  - SoXtExaminerViewer…
- Open Inventor for Java user interface components begin with **Sw**
  - SwSimpleViewer…
- Open Inventor for classes begin with **Pb, Po** or **PoXt**
  - PbDomain, PoLinearAxis, PoXtAxisEditor…
- Methods and variables begin with a **lowercase** letter
- Each word within a basic type, a class, method, or variable name begins with an **uppercase** letter
  - getNormal()
- All enumerated type values are in **uppercase**
  - EXAMINER, PLANE, …

A first C++ example

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

void main(int , char **argv)
{
   // Initialize Inventor. Returns a main window to use. If unsuccessful, exit.
   Widget myWindow = SoXt::init(argv[0]); // pass the app name
   if (myWindow == NULL) exit(1);

   // Build a scene containing a red cone
   SoSeparator *root = new SoSeparator;
   root->ref();
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Red
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   root->addChild(myMaterial);
   root->addChild(new SoCone);

   // Create a renderArea within the main window in which to see our scene graph.
   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

   // Make myCamera see everything.
   myCamera->viewAll(root, myRenderArea->getViewportRegion());

   // Put our scene in myRenderArea
   myRenderArea->setSceneGraph(root);
   myRenderArea->show();

   SoXt::show(myWindow);  // Display main window
   SoXt::mainLoop();      // Main Inventor event loop
}
```

A first Java example

```
import java.awt.* ;
import java.awt.event.*;
import com.tgs.inventor.*;
import com.tgs.inventor.awt.* ;
import com.tgs.inventor.nodes.* ;

public class HelloCone  {
  public static void main(String argv[]) {
    // Make a scene containing a red cone
    SoSeparator root = new SoSeparator();
    SoMaterial myMaterial = new SoMaterial();
    myMaterial.diffuseColor.setValue(1,0,0); // Red
    root.addChild(new SoDirectionalLight());
    root.addChild(myMaterial);
    root.addChild(new SoCone());

    // Put the scene in myRenderArea
    SwSimpleViewer myRenderArea = new SwSimpleViewer();
    myRenderArea.setSceneGraph(root);
    Panel panel = new Panel(new BorderLayout()) ;
    panel.add(myRenderArea);
    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      };
    Frame f = new Frame ("HelloCone");
    f.addWindowListener(l);
    f.add(panel);
    f.pack();
    f.show();
  }
}
```

Open Inventor scene graph

- Before displaying anything an application must create a **scene graph** which holds the representation of what will to be displayed.
- A **scene graph** is a set of one or more nodes defining a **shape** (geometry), a **property**, or a **grouping** node. The scene becomes hierarchical by adding nodes as children of another node.
- The resulting graph must be directed acyclic.
- A **path** is a chain of nodes, each of which is a child of the previous node. It is a subgraph and is useful to identify part of the scene graph and is used, for examples by the picking action to return information. Because of shared instances of a node, a path is required to identify a specific instance of a node in the scene graph.

Open Inventor traversal

- To display, write to a file, pick or get the bounding box of (part of) the database, an **action** is applied to it.
- Applying an action results in activation of a **traversal** process.
- Traversal processes the graph nodes from top to bottom and left to right.
- The traversal process maintains a **traversal state list** which includes the current rendering attributes:
    - Current transformation
    - Current material
    - Current coordinates...
- **Property** nodes modify the traversal state list.
- Traversing a shape node causes its shape to be rendered depending on the current traversal state list attributes
- **Group** nodes can save/restore traversal state elements

SoDB: scene graph database class
SoInput: how to read a scene file

- #include <Inventor/SoDB.h>
- First methods to know:
  - static void **init**()
    - Must be called before any other calls that modify the database (called by SoXt::init()).
  - static SoSeparator ***readAll**(SoInput *in)
    - To read back all graphs from a file.

- Some useful methods of SoInput class:
  - SbBool **openFile**(const char *fileName, SbBool okIfNotFound = FALSE)
    - Opens the named file and sets the file pointer to result. FALSE is returned on error, and if okIfNotFound is FALSE (default), an error message is output.
  - void **setBuffer** (void *bufPointer, size_t bufSize)
    - Allows to read an in-memory buffer or character string.

SoDB: scene graph database class
SoInput: how to read a scene file

- import com.tgs.inventor.SoDB;
- First methods to know:
  - static SoNode **readNode**(SoInput in)
    - To read back a scene graph from a file and return a node.

- Some useful methods of SoInput class:
  - boolean **openFile**(String, boolean okIfNotFound)
    - Opens the named file and sets the file pointer to result. false is returned on error, and if okIfNotFound is false, an error message is output.
  - void **setBuffer** (byte[] buffer, int size)
    - Allows to read an in-memory buffer.

SoXt: The Xt compatibility class

- SoXt: Xt compatibility class
- Some useful methods:
  - static Widget **init**(const char *appName)
    - This call initializes Inventor (SoDB::init()) and Xt.
  - static void **mainLoop**()
    - To get and dispatch the next event from the event queue.
  - static void **show**(Widget widget)
    - To show (realize + map) the widget.
- An Inventor application may begin using **SoXt::init**(...) to initialize the Inventor database, Inventor events, and the X Toolkit.

Using rendering areas and viewers

- After initializing the toolkit (SoXt::init(...)) an Inventor application must create a window in which to render the scene graph. Using the libSoXt component library, this window can be one of the following:
  - **SoXtRenderArea**: The basic window
  - **SoXtExaminerViewer**: A viewer that uses a virtual trackball to see the data
  - **SoXtWalkViewer**: A viewer that moves the camera in a plane simulating a walk
  - **SoXtFlyViewer**: A viewer to fly through space
  - **SoXtPlaneViewer**: A viewer that moves the camara in a plane
- RenderArea has built-in event translator and all other viewers have misc features for user interface.

Using rendering areas and viewers

The application must create a window in which to render the scene graph. Using the package com.tgs.inventor.awt, this window can be one of the following:

- **SwActiveArea**: The basic window
- **SwExaminerArea**: A viewer that uses a virtual trackball to see the data
- **SwPlaneArea**: A viewer that moves the camara in a plane
- ***SwWalkArea*** *: A viewer that moves the camera in a plane simulating a walk*
- ***SwFlyArea*** *: A viewer to fly through space*
- **SwSimpleViewer** : An extension of panel that contains an area (examiner, plane …) buttons and wheels to interact with the area.

How to render on the screen?

- void **setSceneGraph**(SoNode *scene) method must be called to associate the scene with any of the rendering areas or viewers.
- void **show**() method to render the scene graph. This method realizes and maps the widget.
- A basic Inventor application skeleton using libSoXt should be:

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
...
Widget myWindow = SoXt::init(argv[0]);
SoXtRenderArea *Area=new SoXtRenderArea(myWindow);
...
Area->setSceneGraph(scene);
Area->show();
SoXt::show(myWindow);
SoXt::mainLoop();
```

How to render on the screen?

- void **setSceneGraph**(SoNode scene) method must be called to associate the scene with any of the rendering areas or viewers.
- The area or viewer must be added to a Frame or Window or Applet. void **setVisible**() method (or show) to render the scene graph.
- A basic Open Inventor for application skeleton should be:

```
import com.tgs.inventor.awt.*;
...
SwExaminerArea area=new SwExaminerArea();
...
area.setSceneGraph(scene);
…
Frame f = new Frame();
f.add(area);
f.pack();
f.setVisible(true);
```

Exercise 1

- Write an application which reads back an .iv file to create the scene graph and then use a viewer to display it on the screen.

Creating and changing a node

- To create a node use the **new** operator (nodes cannot be declared on the stack!!!)
  - SoCone *myCone = new SoCone;
  - SoLineSet *myLines = new SoLineSet;

- Nodes are composed of a set of data elements: the **fields**. These fields can be changed using the methods provided by each field:
    - myCone->bottomRadius.setValue(2.);
    - myCone->height.setValue(4.);
    - myMaterial->ambientColor.setValue(1.,0.,0.);

  - or for single-value fields :
    - myCone->bottomRadius = 2.;
    - myCone->height = 4.;

Creating and changing a node

- Nodes are composed of a set of data elements: the **fields**. These fields can be changed using the methods provided by each field:
    - myCone.bottomRadius.setValue(2);
    - myCone.height.setValue(4.);
    - myMaterial.ambientColor.setValue(1,0,0);

    - radius = myCone.bottomRadius.getValue();
    - SbColor c = myMaterial.ambientColor.getValue();

Node fields

- Node-specific values are stored in **fields** and not in members. This allows Inventor to trace scene graph changes and allows **connections** between fields.
- There are two types of fields
  - Single value fields (So**SF**Float, So**SF**Long...)
    - Use **setValue** or **=** operator to initialize the field
    - Use **getValue** or = operator to retrieve the field
  - Multiple value fields (So**MF**Float, So**MF**Long...)
    - Use **setValues** to set multiple values
    - Use **set1Value** to initialize one of the values of the field
    - Use **getValues** or **[]** operator to retrieve the field
    - Use **deleteValues** to delete some values in the field…
    - Fast multiple field creation or editing :
      - Give first the size if known with **setNum** to avoid reallocations
      - Use **setValue<u>s</u>** instead of loop with set1Value()
      - Or use **startEditing / finishEditing**

Node fields

- Node-specific values are stored in **fields** and not in members. This allows Inventor to trace scene graph changes and allows **connections** between fields.
- There are two types of fields
  - Single value fields (So**SF**Float, So**SF**Int...)
    - Use **setValue** to modify the field
    - Use **getValue** to retrieve the field
  - Multiple value fields (So**MF**Float, So**MF**Int...)
    - Use **setValues** to set multiple values
    - Use **set1Value** to initialize one of the values of the field
    - Use **getValues** or to retrieve the field
    - Use **deleteValues** to delete some values in the field…
    - Fast multiple field creation or editing :
      - Give first the size if known with **setNum** to avoid reallocations
      - Use **setValue<u>s</u>** instead of loop with set1Value()

Node referencing (1/2)

- Do not allocate an **array** of nodes
- Do not use **delete** operator to delete a node
- A node maintains the number of references to itself in the scene graph. If this count **goes down to 0**, then the node is automatically deleted.
- Each time a node is added as child of another node or referenced in a path its reference count is incremented.
- When you apply an action to a node which has a 0 reference count, because **action creates a path**, node reference is incremented to 1; at end of action the reference count is decremented to 0 and then the node is deleted!!!
- When a node is created, its reference count is zero, but because it’s not decremented to zero, the node is not deleted (!!!)

Node referencing (2/2)

- Use **ref**(), **unref**() or **unrefNoDelete**() methods to control node reference count. node->ref() when creating a node prevents it from being deleted.
- Example: A routine to create a sphere giving its radius and returning its bounding box

<u>Buggy version</u>

```
SoSphere *makeSphere(float radius, SbBox3f &box)
{
 SoSphere *sphere = new SoSphere;
 sphere->radius.setValue(radius);
 SoGetBoundingBoxAction *ba = new
SoGetBoundingBoxAction;
 ba->apply(sphere);
 box = ba->getBoundingBox();

 return sphere;
}
```

<u>Fixed version</u>

```
SoSphere *makeSphere(float radius, SbBox3f &box)
{
 SoSphere *sphere = new SoSphere;
 sphere->radius.setValue(radius);
 sphere->ref();
 SoGetBoundingBoxAction *ba = new
SoGetBoundingBoxAction;
 ba->apply(sphere);
 box = ba->getBoundingBox();
 sphere->unrefNodelete();

 return sphere;
}
```

Naming nodes

- Nodes can be named using the **setName** method (inherited from SoBase class).
- Nodes can be searched for by name using the **getByName** method.
- Node names appear in .iv files. This make them more legible and parts in the file are easily identified.

```
#include <Inventor/SoDB.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
void RemoveCube();

void main(int , char **)
{
  SoDB::init();

  // Create some objects and give them names:
  SoSeparator *root = new SoSeparator;
  root->ref();
  root->setName("Root");
  SoCube *myCube = new SoCube;
  root->addChild(myCube);
  myCube->setName("MyCube");
  SoSphere *mySphere = new SoSphere;
  root->addChild(mySphere);
  mySphere->setName("MySphere");
  RemoveCube();
}

void RemoveCube()
{
  SoSeparator *myRoot;
  myRoot = (SoSeparator *)SoNode::getByName("Root");
  SoCube *myCube;
  myCube = (SoCube *)SoNode::getByName("MyCube");
  myRoot->removeChild(myCube);
}
```

Naming nodes

- Nodes can be named using the **setName** method (inherited from SoBase class).
- Nodes can be searched for by name using the **getName** method.
- Node names appear in .iv files. This make them more legible and parts in the file are easily identified.

```
import com.tgs.inventor.*;
import com.tgs.inventor.awt.* ;
import com.tgs.inventor.nodes.* ;

class NamingNodes {
  public static void main(String argv[]) {
    // Create some objects and give them names:
    SoSeparator root = new SoSeparator();
    root.setName("Root");
    SoCube myCube = new SoCube();
    root.addChild(myCube);
    myCube.setName("MyCube");
    SoSphere mySphere = new SoSphere();
    root.addChild(mySphere);
    mySphere.setName("MySphere");
    RemoveCube();
    System.exit(0);
  }

  static void RemoveCube() {
    SoSeparator myRoot;
    myRoot = (SoSeparator)SoNode.getByName("Root");
    SoCube myCube;
    myCube = (SoCube)SoNode.getByName("MyCube");
    myRoot.removeChild(myCube);
  }
}
```

Three basic types of nodes

- **Shape nodes**
  - They represent 3D geometric objects
- **Property nodes**
  - They represent appearance and qualitative characteristics of the scene
    - transformations
    - appearance
    - metrics
- **Group nodes**
  - They are used as containers for a set of nodes

Shape node classes

- SoBase
  - SoFieldContainer
    - SoNode
      - SoShape
        - SoCone
        - SoCube
        - SoCylinder
        - SoIndexedNurbsCurve
        - SoIndexedNurbsSurface
        - SoNurbsCurve
        - SoNurbsSurface
        - SoSphere
        - SoText2
        - SoText3
        - SoVertexShape
          - SoIndexedShape
            - SoIndexedFaceSet
            - SoIndexedLineSet
            - SoIndexedTriangleStripSet
          - SoNonIndexedShape
            - SoFaceSet
            - SoLineSet
            - SoPointSet
            - SoQuadMesh
            - SoTriangleStripSet

SoCylinder

- **SoCylinder** fields:
  - SoSFBitMask **parts**: Visible parts of the cylinder
    - SoCylinder::SIDES
    - SoCylinder::TOP
    - SoCylinder::BOTTOM
    - SoCylinder::ALL (default)
  - SoSFFloat **radius**
  - SoSFFloat **height**
- SoCylinder methods:
  - void **addPart**(SoCylinder::Part part)
  - void **removePart**(SoCylinder::Part part)
    - To turn on and off parts of the cylinder

```
SoCylinder *myCylinder = new SoCylinder;
myCylinder->radius = 1.;
myCylinder->height = 3.;
myCylinder->removePart(SoCylinder::TOP);
```

SoCylinder

- SoCylinder fields:
  - SoSFBitMask **parts**: Visible parts of the cylinder
    - SoCylinder.SIDES
    - SoCylinder.TOP
    - SoCylinder.BOTTOM
    - SoCylinder.ALL (default)
  - SoSFFloat **radius**
  - SoSFFloat **height**
- SoCylinder methods:
  - void **addPart**(int part)
  - void **removePart**(int part)
    - To turn on and off parts of the cylinder

```
SoCylinder myCylinder = new SoCylinder();
myCylinder.radius.setValue(1);
myCylinder.height.setValue(3);
myCylinder.removePart(SoCylinder.TOP);
```

SoLineSet

- **SoLineSet** fields:
  - SoMFInt32 **numVertices**: list of number of vertices for each polyline
  - SoSFNode **vertexProperty** : points description
  - Reminder : since 2.1, *startIndex* field is obsolete (for all non-indexed shapes),
    as well as *SO_LINE_SET_USE_REST_OF_VERTICES* (-1) in numVertices
- If **vertexProperty** is NULL, SoLineSet uses the current coordinates given by the last **SoVertexProperty** or **SoCoordinate** property node traversed.
- Following example, draw three polylines, the first one uses points 1,2,3 of the coordinates list, the second one, 4,5,6,7 and last one 8 and 9.

```
static long numPoints[] = {3,4,2}
SoLineSet *myLineSet = new SoLineSet;
myLineSet->numVertices.setValues(0,3,numPoints);
```

SoLineSet

- SoLineSet fields:
  - SoMFInt **numVertices**: list of number of vertices for each polyline
  - SoSFNode **vertexProperty** : points description
- If **vertexProperty** is null, SoLineSet uses the current coordinates given by the last **SoVertexProperty** or **SoCoordinate** property node traversed.
- Following example, draw three polylines, the first one uses points 1,2,3 of the coordinates list, the second one, 4,5,6,7 and last one 8 and 9.

```
static int[] numPoints={3,4,2};
SoLineSet myLineSet = new SoLineSet();
myLineSet.numVertices.setValues(0,numPoints);
```

SoIndexedFaceSet

- **SoIndexedFaceSet** fields: (all from **SoIndexedShape** class)
  - SoMFInt32 **coordIndex**: list of vertex indices for each face. Each face list ends with SO_END_FACE_INDEX (-1).
  - **materialIndex**, **normalIndex** and **textureIndex** fields may be initialized to specify a color/normal/texture at each vertex/face/shape (Described later with property nodes).
  - SoSFNode **vertexProperty**
- If **vertexProperty** is NULL, SoIndexedFaceSet uses the current coordinates given by the last **SoVertexProperty** or **SoCoordinate** property node traversed.
- Following example, draw a set of two faces with 3 and 4 vertices.

```
static long vertexList[] = {2,4,3,-1, 7,3,4,5,-1};
SoIndexedFaceSet *myIndexedFaceSet = new SoIndexedFaceSet ;
myIndexedFaceSet ->coordIndex(0,9, vertexList);
```

SoIndexedFaceSet

- SoIndexedFaceSet fields: (all from **SoIndexedShape** class)
  - SoMFInt **coordIndex**: list of vertex indices for each face. Each face list ends with SO_END_FACE_INDEX (-1).
  - **materialIndex**, **normalIndex** and **textureIndex** fields may be initialized to specify a color/normal/texture at each vertex/face/shape (Described later with property nodes).
  - SoSFNode **vertexProperty**
- If **vertexProperty** is null, SoIndexedFaceSet uses the current coordinates given by the last **SoVertexProperty** or **SoCoordinate** property node traversed.
- Following example, draw a set of two faces with 3 and 4 vertices.

```
static int[] vertexList = {2,4,3,-1, 7,3,4,5,-1};
SoIndexedFaceSet fset = new SoIndexedFaceSet ();
fset.coordIndex.setValues(0, vertexList);
```

SoText2

- **SoText2** fields:
  - SoMFString **string**
  - SoSFFloat **spacing**: distance between two successive characters depending on text height. 1. for single spacing, 2. for double...
  - SoSFEnum **justification**:
    - SoText2::LEFT
    - SoText2::RIGHT
    - SoText2::CENTER

```
SoFont *myFont = new SoFont;
myFont->name.setValue("Times-Roman");
myFont->size.setValue(24.0);
SoText2 *myText = new SoText2;
myText->string = "STRING";
myText->spacing = .5;
myText->justification = CENTER;
```

- The 2D text will not be affected by any of the transformations defined in the scene graph and will always be drawn screen-aligned.
- The text font and text height (in pixels) are defined using the SoFont property node

SoText2

- SoText2 fields:
  - SoMFString **string**
  - SoSFFloat **spacing**: distance between two successive characters depending on text height. 1. for single spacing, 2. for double.
  - SoSFEnum **justification**:
    - SoText2.LEFT
    - SoText2.RIGHT
    - SoText2.CENTER

```
SoFont myFont = new SoFont();
myFont.name.setValue("Times-Roman");
myFont.size.setValue(24);
SoText2 myText = new SoText2();
myText.string.setValue("STRING");
myText.spacing.setValue(0.5f);
myText.justification.setValue(SoText2.CENTER);
```

- The 2D text will not be affected by any of the transformations defined in the scene graph and will always be drawn screen-aligned.
- The text font and text height (in pixels) are defined using the SoFont property node

SoText3

- **SoText3** fields:
  - SoMFString **string**
  - SoSFFloat **spacing** (See SoText2)
  - SoSFEnum **justification** (See SoText2)
  - SoSFBitMask **parts**
    - SoText3::FRONT (default)
    - SoText3::BACK
    - SoText3::SIDES
    - SoText3::ALL
- The 3D text will be rendered by extruding its 2D faces along a **profile** defined by an **SoProfile** property node. By default, the profile is a straight line of length 1.

Property node classes

- SoBase
  - SoFieldContainer
    - SoNode
      - SoBaseColor
      - SoColorIndex
      - SoComplexity
      - SoCoordinate3
      - SoCoordinate4
      - SoDrawStyle
      - SoEnvironment
      - SoFont
      - SoInfo
      - SoLabel
      - SoLightModel
      - SoMaterial
      - SoMaterialBinding
      - SoMaterialIndex (obsolete)
      - SoNormal
      - SoNormalBinding
      - SoPackedColor
      - SoPickStyle
      - SoProfile
        - SoLinearProfile
        - SoNurbsProfile
      - SoProfileCoordinate2
      - SoProfileCoordinate3
      - SoShapeHints
      - SoTexture2
      - SoTexture2Transform
      - SoTextureCoordinate2
      - SoTextureCoordinateBinding
      - SoTextureCoordinateFunction
        - SoTextureCoordinateDefault
        - SoTextureCoordinateEnvironment
        - SoTextureCoordinatePlane
      - SoTransformation
        - SoAntiSquish
        - SoMatrixTransform
        - SoResetTransform
        - SoRotation
          - SoPendulum
          - SoRotor
        - SoRotationXYZ
        - SoScale
        - SoSurroundScale
        - SoTransform … (SoTransformManip)
        - SoTranslation
          - SoShuttle
        - SoUnits
      - SoUnknownNode
      - SoVertexProperty

Coordinates

- **SoCoordinate3**: Node which replaces current 3D coordinates. This node sets coordinates used by subsequent shape nodes.
  - Field: SoMFVec3f **point**
- **SoCoordinate4**: Node which replaces current 3D coordinates. This node sets coordinates used by subsequent shape nodes. The first three coordinates are divided by the fourth one to define the 3D coordinates. This node is mainly used by NURBS nodes.
  - Field: SoMFVec4f **point**
- **SoNormal**: Node to define surface normal for subsequent shape nodes.
  - Field: SoMFVec3f **vector**
- **SoProfileCoordinate2(3)**: Node which replaces current 2(3)D profile coordinates. This node sets coordinates used by subsequent SoProfile nodes. Used for 3D text or NURBES
  - Field: SoMFVec2(3)f **point**

SoVertexProperty

- **SoVertexProperty**: used to efficiently specify coordinates, normals, texture coordinates, colors, transparency values, material binding and normal binding for vertex-based shapes, i.e., shapes of class **SoVertexShape**.
- SoVertexProperty **fields** :
  - SoMFVec3f **vertex**
  - SoMFVec3f **normal**
  - SoMFUInt32 **orderedRGBA**
  - SoMFVec2f **texCoord**
  - SoSFEnum **normalBinding**
  - SoSFEnum **materialBinding**

•SoVertexProperty::OVERALL
•SoVertexProperty::PER_PART
•SoVertexProperty::PER_PART_INDEXED
•SoVertexProperty::PER_FACE
•SoVertexProperty::PER_FACE_INDEXED
•SoVertexProperty::PER_VERTEX
•SoVertexProperty::PER_VERTEX_INDEXED

- Can be used as a child of a group node in a scene graph
- Can also be directly referenced as the **VertexProperty SoSFField** of a vertex-based shape, bypassing scene graph inheritance *(preferred for performance).*

SoVertexProperty

- **SoVertexProperty**: used to efficiently specify coordinates, normals, texture coordinates, colors, transparency values, material binding and normal binding for vertex-based shapes, i.e., shapes of class **SoVertexShape**.
- SoVertexProperty **fields** :
  - SoMFVec3f **vertex**
  - SoMFVec3f **normal**
  - SoMFInt **orderedRGBA**
  - SoMFVec2f **texCoord**
  - SoSFEnum **normalBinding**
  - SoSFEnum **materialBinding**

•SoVertexProperty.OVERALL
•SoVertexProperty.PER_PART
•SoVertexProperty.PER_PART_INDEXED
•SoVertexProperty.PER_FACE
•SoVertexProperty.PER_FACE_INDEXED
•SoVertexProperty.PER_VERTEX
•SoVertexProperty.PER_VERTEX_INDEXED

- Can be used as a child of a group node in a scene graph
- Can also be directly referenced as the **VertexProperty SoSFField** of a vertex-based shape, bypassing scene graph inheritance *(preferred for performance).*

SoDrawStyle

- **SoDrawStyle** property node is used to define the style of rendering
  - Fields:
    - SoSFEnum **style**: drawing mode:
      - SoDrawStyle::FILLED
      - SoDrawStyle::LINES
      - SoDrawStyle::POINTS
      - SoDrawStyle::INVISIBLE
    - SoSFFloat **pointSize**: radius of points
    - SoSFFloat **lineWidth**
    - SoSFUShort **linePattern** (0 to 0xffff)

```
SoDrawStyle *style = new SoDrawStyle ;
style->style = POINTS;
style->lineWidth = 2.;
```

SoDrawStyle

- **SoDrawStyle** property node is used to define the style of rendering
  - Fields:
    - SoSFEnum **style**: drawing mode:
      - SoDrawStyle.FILLED
      - SoDrawStyle.LINES
      - SoDrawStyle.POINTS
      - SoDrawStyle.INVISIBLE
    - SoSFFloat **pointSize**: radius of points
    - SoSFFloat **lineWidth**
    - SoSFShort **linePattern** (0 to 0xffff)

```
SoDrawStyle drawStyle = new SoDrawStyle() ;
drawStyle.style.setValue(SoDrawStyle.POINTS);
drawStyle.lineWidth.setValue(2);
```

SoMaterial

- **SoMaterial** is used to define current surface material properties. If you want to set only a diffuse color for shape nodes use **SoBaseColor** instead to get **faster rendering**.
  - Fields (*SoMFColor* for backward compatibility only, supports only unique value) :
    - *SoMFColor* **ambientColor**
    - SoMFColor **diffuseColor**
    - *SoMFColor* **specularColor**
    - *SoMFColor* **emissiveColor**
    - *SoMFFloat* **shininess** (0. to 1.)
    - *SoMFFloat* **transparency** (0. opaque to 1. full)

```
SoMaterial *gold = new SoMaterial;
gold->ambientColor.setValue(.3,.1,.1);
gold->diffuseColor.setValue(.8,.7,.2);
gold->specularColor.setValue(.4,.3,.1);
gold->shininess = .4;
```

- **Transparency** and **diffuseColor** can be used as multiple fields to be used with **SoMaterialBinding** property node.
- Note that **SoVertexProperty** provides support for changing diffuse color and transparency within a shape (with accelerated performance).

SoMaterial

- **SoMaterial** is used to define current surface material properties. If you want to set only a diffuse color for shape nodes use **SoBaseColor** instead to get **faster rendering**.
  - Fields (*SoMFColor* for backward compatibility only, supports only unique value) :
    - *SoMFColor* **ambientColor**
    - SoMFColor **diffuseColor**
    - *SoMFColor* **specularColor**
    - *SoMFColor* **emissiveColor**
    - *SoMFFloat* **shininess** (0. to 1.)
    - *SoMFFloat* **transparency** (0. opaque to 1. full)

```
SoMaterial gold = new SoMaterial();
gold.ambientColor.setValue(.3f,.1f,.1f);
gold.diffuseColor.setValue(.8f,.7f,.2f);
gold.specularColor.setValue(.4f,.3f,.1f);
gold.shininess.setValue( .4f);
```

- **Transparency** and **diffuseColor** can be used as multiple fields to be used with **SoMaterialBinding** property node.
- Note that **SoVertexProperty** provides support for changing diffuse color and transparency within a shape (with accelerated performance).

SoMaterialBinding

- **SoMaterialBinding** property node is used to specify how **multiple material** are bound to shapes (i.e. diffuse colors and transparency, see **SoMaterial**).
  The correct number of diffuse colors must be specified.
  If not enough transparency are provided, the first one is used.
  - Fields:
    - SoSFEnum **value:** This value may be meaningless depending on which shape is going to use it.
      - SoMaterialBinding::DEFAULT
      - SoMaterialBinding::NONE
      - SoMaterialBinding::OVERALL
      - SoMaterialBinding::PER_PART
      - SoMaterialBinding::PER_PART_INDEXED
      - SoMaterialBinding::PER_FACE
      - SoMaterialBinding::PER_FACE_INDEXED
      - SoMaterialBinding::PER_VERTEX
      - SoMaterialBinding::PER_VERTEX_INDEXED

Normals

- If an Open Inventor application does not provide the normals needed for rendering, does not provide normal list (**SoNormal** or **SoVertexProperty**) or does not specify enough normals for the number of vertices, faces or parts
  - then Open Inventor will automatically compute the normals using the **creaseAngle** field sets using **SoShapeHints** node.
- Automatic calculation is time consuming so it may be better to provide the SoNormal node to avoid this. Caching can be used to avoid multiple calculation if normals cannot be provided in an SoNormal node and the automatic calculation is needed.

SoNormalBinding

- **SoNormalBinding** property node is used to specify how normals from **SoNormal** node are bound to shapes.
  - Fields:
    - SoSFEnum **value:** This value may be meaningless depending on which shape is going to use it.
      - SoNormalBinding ::OVERALL
      - SoNormalBinding ::PER_PART
      - SoNormalBinding ::PER_PART_INDEXED
      - SoNormalBinding ::PER_FACE
      - SoNormalBinding ::PER_FACE_INDEXED
      - SoNormalBinding ::PER_VERTEX
      - SoNormalBinding ::PER_VERTEX_INDEXED
      - (DEFAULT and NONE are obsolete, equivalent to PER_VERTEX_INDEXED)
- You must specify the correct number of normals in the SoNormal node (no cycling).

SoLightModel

- **SoLightModel** node is used to set the lighting model for rendering.
  - Fields:
    - SOSFEnum **model**
      - SoLightModel::**BASE_COLOR**
        - Use only the diffuse color of the model. In this mode light sources are ignored.
      - SoLightModel::**PHONG** (default)
        - Use Phong lighting model. In this mode light sources are combined with material properties depending on the surface orientation to produce the correct shading.
        - Does not support indexed colors (**SoColorIndex**).

SoShapeHints

- SoShapeHints property node is used to give some hints about the geometry of shapes to be rendered. This is used by Inventor to optimize rendering performance.
  - Fields:
    - SoSFEnum **vertexOrdering**
      - SoShapeHints::UNKNOWN_ORDERING
      - SoShapeHints::CLOCKWISE
      - SoShapeHints::COUNTERCLOCKWISE
    - SoSFEnum **shapeType**: Is the shape a closed volume?
      - SoShapeHints::UNKNOWN_SHAPE_TYPE
      - SoShapeHints::SOLID
    - SoSFEnum **faceType**
      - SoShapeHints::UNKNOWN_FACE_TYPE
      - SoShapeHints::CONVEX
    - SoSFFloat **creaseAngle**: minimum angle to form a sharp crease.

SoComplexity

- SoComplexity node is used to detemine the complexity or precision of the way shapes are rendered.
  - Fields:
    - SoSFEnum **type**
      - SoComplexity::SCREEN_SPACE: Subdivision of the object relies on the size of the projection on the screen.
      - SoComplexity::OBJECT_SPACE: Subdivision of the object in faces relies on the object size itself. This is the default.
      - SoComplexity::BOUNDING_BOX: Only the bounding box of the object is rendered using current traversal state values.
    - SoSFFloat **value**: (0. - 1.) minimum to maximum complexity.
    - SoSFFloat **textureQuality**: (0. - 1.) minimum to maximum quality.

SoTransform

- SoTransform property node defines a 3D transformation consisting of a non-uniform scale about an arbitrary point, a rotation about an arbitrary point and axis, and then a translation.
- Unlike other property nodes, this node does not replace the current traversal state list value but has a cumulative effect on the current geometric transformation.
  - Fields:
    - SoSFVec3f **translation**
    - SoSFRotation **rotation**
    - SoSFVec3f **scaleFactor**
    - SoSFRotation **scaleOrientation**
    - SoSFVec3f **center**
- Methods can be used to initialize the transformation matrix without initializing each field of the SoTransform node.

“Lightweight” transformation nodes

- Instead of using an SoTransform node that initializes a complete transformation matrix, the following nodes can be used to specify a transformation:
  - **SoRotation**
    - Field: SoSFRotation **rotation**
  - **SoRotationXYZ**
    - Fields:
      - SoSFEnum **axis** (X,Y,Z)
      - SoSFFloat **angle**
  - **SoTranslation**
    - Field:
      - SoSFVec3f **translation**
  - **SoScale**
    - Field:
      - SoSFVec3f **scaleFactor**

Group node classes

- SoBase
  - SoFieldContainer
    - SoNode
      - SoGroup
        - SoArray
        - SoLevelOfDetail
        - SoMultipleCopy
        - SoPathSwitch
        - SoSeparator
          - SoAnnotation
          - SoSelection
        - SoSwitch
          - SoBlinker
        - SoTransformSeparator

Group nodes

- A group node is a container for collecting child objects.
- Basic class for all group nodes is SoGroup.
- **SoGroup** node first methods:
  - void **addChild**(SoNode *child)
  - void **insertChild**(SoNode *child, int newChildIndex)
  - void **removeChild**(int index)
  - void **removeChild**(SoNode *child)
  - void **removeAllChildren**()
  - void **replaceChild**(int index, SoNode *newChild)
  - void **replaceChild**(SoNode *old, SoNode *new)
- Order of children is very important because traversal order maintains the traversal state list and because children are going to be rendered in the same order.

Group nodes

- A group node is a container for collecting child objects.
- Basic class for all group nodes is SoGroup.
- **SoGroup** node first methods:
  - void **addChild**(SoNode child)
  - void **insertChild**(SoNode child, int newChildIndex)
  - void **removeChild**(int index)
  - void **removeChild**(SoNode child)
  - void **removeAllChildren**()
  - void **replaceChild**(int index, SoNode newChild)
  - void **replaceChild**(SoNode old, SoNode new)
- Order of children is very important because traversal order maintains the traversal state list and because children are going to be rendered in the same order.

SoSeparator

- **SoSeparator** group node inserted in a scene graph saves the traversal state list, and then restores it after the traversal has gone through all its children.
- **SoTransformSeparator** has the same effect but only on the current geometric transformation.
- SoSeparator is used to separate sub graphs in a scene graph.
- To be sure the traversal state list is set to default value before each redraw, **use a SoSeparator node as the root node** of the scene graph.
- A root node is not referenced, so its reference count remains 0. Applying an action to it (such as rendering the scene graph) will increment its reference number to 1 and then decrease it to 0 after rendering. Then the node is deleted!

```
SoSeparator *mySep = new SoSeparator;
mySep->ref();
mySep->addChild(myNode);
```

SoSeparator

- **SoSeparator** group node inserted in a scene graph saves the traversal state list, and then restores it after the traversal has gone through all its children.

- **SoTransformSeparator** has the same effect but only on the current geometric transformation.

- SoSeparator is used to separate sub graphs in a scene graph.

- To be sure the traversal state list is set to default value before each redraw, **use a SoSeparator node as the root node** of the scene graph.

SoSwitch & SoLevelOfDetail

- **SoSwitch** group node selects one of its children during traversal process according to the value of its field **whichChild.**

C++

```
SoSwitch *mySwitch = new SoSwitch ;
mySwitch->addChild(node1);
mySwitch->addChild(node2);
mySwitch->addChild(node3);
mySwitch->whichChild = 2;
```

Java

```
SoSwitch mySwitch = new SoSwitch ();
mySwitch.addChild(node1);
mySwitch.addChild(node2);
mySwitch.addChild(node3);
mySwitch.whichChild.setValue(2);
```

- **SoBlinker** node (derived from SoSwitch class) can be used to automatically cycle through the children.
- **SoLevelOfDetail** switching group node allows conditional traveral of its its children depending on area values defined in its field **screenArea.** The screen area is computed as the size of the projection of the object bounding box onto the screen in square pixel units. This area is then compared to the values in the field. If area is greater than the first value then first child is traversed, if area is between first and second value then second child... This node is very important for **display performance**.
- **SoLOD** is a distance based level of detail switching group.

Light nodes

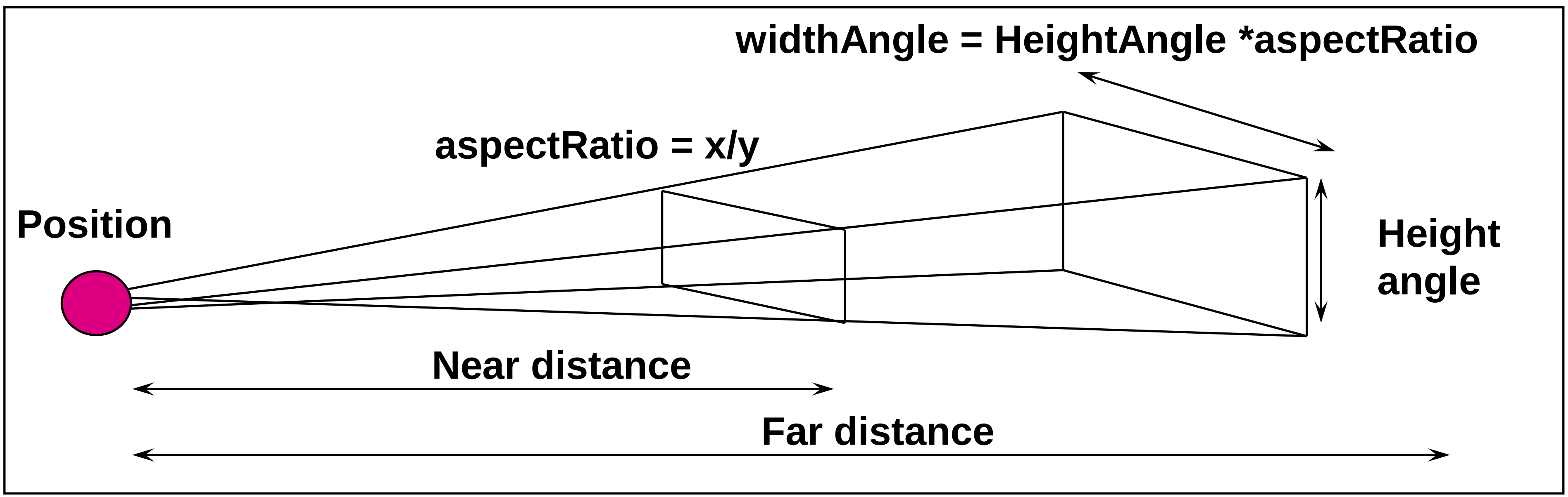
- **SoPointLight**, **SoDirectionalLight** and **SoSpotLight** may be added to a scene graph to light it.
  - Reminder: Ambient light is defined in an SoEnvironment node, and shape nodes can also have an emissive color. Also, the default lighting model is PHONG, so if a scene does not include any light source, nothing will be visible.
- **SoTransformSeparator** node may be useful to separate light location transformations from shape geometry transformations.
- Each of these light nodes has the following fields:
  - SoSFBool **on**
  - SoSFFloat **intensity** (0. to 1.)
  - SoSFColor **color**
  - SoSFVec3f **location** (Point+Spot only)
  - SoSFVec3f **direction** (Dir+Spot only)
  - SoSFFloat **dropOffRate** (Spot only)
  - SoSFFloat **cutOffAngle** (Spot only)

Camera nodes

- A scene graph may contain a camera node to define viewing.
- Only one camera node should be found when traversing a scene graph.
- **SoCamera** is the base class for all camera nodes.
- SoCamera fields:
  - SoSFEnum **viewportMapping**
  - SoSFVec3f **position**
  - SoSFRotation **orientation**
  - SoSFFloat **aspectRatio**
  - SoSFFloat **nearDistance**
  - SoSFFloat **farDistance**
  - SoSFFloat **focalDistance**

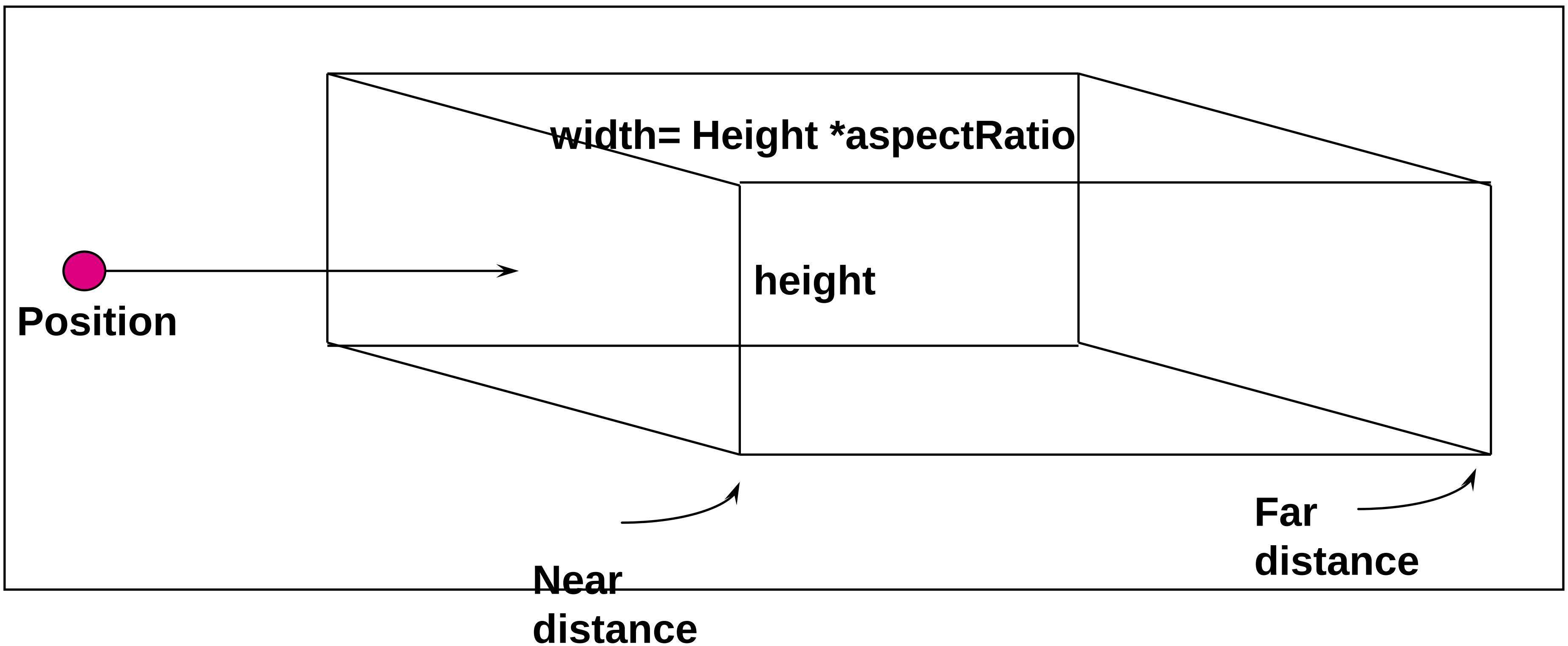
SoPerspectiveCamera

- **SoPerspectiveCamera** is derived from SoCamera class.
- This node defines a “natural” view that imitates how objects appear to a human observer.
- Added field:
  - SoSFFloat **heightAngle**



SoOrthographicCamera

- **SoOrthographicCamera** is derived from SoCamera class.
- This node defines a parallel projection.
- Added field:
  - SoSFFloat **height**



SoCamera methods

- Specifying SoCamera fields manually may be quite complex. Some useful methods may be used instead:
- void **pointAt**(const SbVec3f &targetPoint)
  - Automatically sets the **orientation** field of the SoCamera node.
- void **viewAll**(SoNode *root, const SbViewportRegion &vRegion)
- void **viewAll**(SoPath *path, const SbViewportRegion &vRegion)
  - To set automatically the **position**, **nearDistance** and **farDistance** fields of the SoCamera node to view all of the scene graph. The SbViewportRegion parameter defines the portion of the rendering area to which the 2D image will be mapped. This region can be set to the whole rendering area by calling the **getRegionViewport**() method of **SoXtRenderArea** or its derived classes.

SoCamera methods

- Specifying SoCamera fields manually may be quite complex. Some useful methods may be used instead:
- void **pointAt**(SbVec3f targetPoint)
  - Automatically sets the **orientation** field of the SoCamera node.
- void **viewAll**(SoNode root, SbViewportRegion vRegion)
- void **viewAll**(SoPath path, SbViewportRegion vRegion)
  - To set automatically the **position**, **nearDistance** and **farDistance** fields of the SoCamera node to view all of the scene graph. The SbViewportRegion parameter defines the portion of the rendering area to which the 2D image will be mapped. This region can be set to the whole rendering area by calling the **getRegionViewport**() method of **SwRenderArea** or its derived classes.

Exercise 2

2

Write an application which creates a simple scene graph describing a pen made of cone and cylinder nodes for instance, light the scene, add a camera which won't affect the light position and display the scene in a viewer.

Sensors classes

- SoSensor
  - SoDelayQueueSensor
    - SoDataSensor
      - SoFieldSensor
      - SoNodeSensor
      - SoPathSensor
    - SoIdleSensor
    - SoOneShotSensor
  - SoTimerQueueSensor
    - SoAlarmSensor
    - SoTimerSensor

Sensors

- Sensors are Inventor classes that watch for **events** and call a user-defined **callback** when events occur.
- There are two main types of sensors:
  - **Data sensors** which watch for field, node, or path changes. These sensors are placed in the **delay queue** which is called when the CPU is idle or after a time-out set by calling the **SoDB::setDelaySensorTimeout()** method
  - **Timer sensors** which respond to certain scheduling conditions. These sensors are placed in the **timer queue** which is called when an alarm or the sensor is scheduled to go off.
- **Triggering** a sensor means calling its callback and removing it from the queue
- **Scheduling** a sensor means adding it to the queue.
- **Unscheduling** a sensor means removing it from the queue.
- **Notifying** a data sensor means letting it know that the attached field, node, or path has changed. The sensor is then scheduled.

Data sensors

- **SoFieldSensor**, **SoNodeSensor,** and **SoPathSensor** are the three data sensors provided in Inventor.
- Skeleton of an application using data sensors:
  - Construct the sensor
  - Define the callback function
  - Define sensor priority (0:high (not queued) to 100:low)
  - Attach the sensor to the field, node, or path
- What happens when the attached field, node or path changes?
  - The sensor is automatically scheduled
  - When CPU is idle or after time-out, the delay queue is processed and sensors in it are triggered
- **getTriggerField**(), **getTriggerNode**(), and **getTriggerPath**() methods of SoSensor class can be used to get back information about the changed field, node, or path.

Data sensor example

- **setPriority()**, **setFunction()** and **attach()** methods are used to initialize data sensors.

```
#include <Inventor/SoDB.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/sensors/SoNodeSensor.h>

// Sensor callback function:
static void rootChangedCB(void *, SoSensor *s) {
   // We know the sensor is really a data sensor:
   SoDataSensor *mySensor = (SoDataSensor *)s;
   SoNode *changedNode = mySensor->getTriggerNode();
   SoField *changedField = mySensor->getTriggerField();

   printf("The node named '%s' changed\n",
    changedNode->getName().getString());

   if (changedField != NULL) {
      SbName fieldName;
      changedNode->getFieldName(changedField, fieldName);
      printf(" (field %s)\n", fieldName.getString());
   } else {
      printf(" (no fields changed)\n");
   }
}
```

```
void main(int , char **)
{
   SoDB::init();

   SoSeparator *root = new SoSeparator;
   root->ref();
   root->setName("Root");

   SoCube *myCube = new SoCube;
   root->addChild(myCube);
   myCube->setName("MyCube");

   SoSphere *mySphere = new SoSphere;
   root->addChild(mySphere);
   mySphere->setName("MySphere");

   SoNodeSensor *mySensor = new SoNodeSensor;

   mySensor->setPriority(0);
   mySensor->setFunction(rootChangedCB);
   mySensor->attach(root);

   // Now, make a few changes...
   myCube->width = 1.0;
   myCube->height = 2.0;
   mySphere->radius = 3.0;
   root->removeChild(mySphere);
}
```

Data sensor example

- **setPriority()**, **setFunction()** and **attach()** methods are used to initialize data sensors.

```
import com.tgs.inventor.*;
import com.tgs.inventor.nodes.*;
import com.tgs.inventor.fields.*;
import com.tgs.inventor.sensors.*;
import com.tgs.inventor.misc.callbacks.SoSensorCB;

class DataSensorExample {

  // Sensor callback function:
  static class rootChangedCB extends SoSensorCB {
    public void invoke(SoSensor s) {
      // We know the sensor is really a data sensor:
      SoDataSensor mySensor = (SoDataSensor)s;
      SoNode changedNode = mySensor.getTriggerNode();
      SoField changedField = mySensor.getTriggerField();

      System.out.println("The node named " +
                         changedNode.getName().getString() +
                         " changed");

      if (changedField != null) {
        SbName fieldName = changedNode.getFieldName(changedField);
        System.out.println("field " +
                           fieldName.getString() +
                           " changed");
      } else
        System.out.println(" (no fields changed)");
    }
  }
```

```
  public static void main(String[] argv) {
    SoSeparator root = new SoSeparator();
    root.setName("Root");

    SoCube myCube = new SoCube();
    root.addChild(myCube);
    myCube.setName("MyCube");

    SoSphere mySphere = new SoSphere();
    root.addChild(mySphere);
    mySphere.setName("MySphere");

    SoNodeSensor mySensor = new SoNodeSensor();

    mySensor.setPriority(0);
    mySensor.setFunction(new rootChangedCB());
    mySensor.attach(root);

    // Now, make a few changes...
    myCube.width.setValue(1);
    myCube.height.setValue(2);
    mySphere.radius.setValue(3);
    root.removeChild(mySphere);
  }

}
```

Other delay-queue sensors

- These sensors must be manually scheduled by applying the **schedule()** method of the SoSensor class.
- They are low-level priority sensors and may be used for low-level priority tasks.
- They will be triggered only once per scheduling.
- **SoOneShotSensor**
  - will invoke its callback only once, when the delay queue is processed
- **SoIdleSensor**
  - will invoke its callback once
  - whenever the application is idle

```
SoOneShotSensor *render;
main() {
...
  render = new SoOneShotSensor(doRenderCB, NULL);
...
}
void changeScene()
{
...
  render->schedule();
}
void doRenderCB(void *userData, SoSensor*)
{
// does rendering...
}
```

Other delay-queue sensors

- These sensors must be manually scheduled by applying the **schedule()** method of the SoSensor class.
- They are low-level priority sensors and may be used for low-level priority tasks.
- They will be triggered only once per scheduling.
- **SoOneShotSensor**
  - will invoke its callback only once, when the delay queue is processed
- **SoIdleSensor**
  - will invoke its callback once
  - whenever the application is idle

```
static SoOneShotSensor render;
static void main(String[]) {
...
  render = new SoOneShotSensor(new doRenderCB());
...
}
void changeScene()
{
...
  render.schedule();
}
static class doRenderCB extends SoSensorCB {
  public void invoke(SoSensor s) {
    // does rendering...
  }
}
```

Timer sensors

- **SoAlarmSensor** and **SoTimerSensor** are the two timer-queued sensors provided in Inventor
- Skeleton of an application using these sensors:
  - Construct the sensor
  - Define the callback function
  - Set timing parameters of the sensor
  - Schedule the sensor
- **SoAlarmSensor** will be triggered once at a specified time
  - methods:
    - void **setTime**(const SbTime &absTime)
    - void **setTimeFromNow**(const SbTime &relTime)
- **SoTimerSensor** will be triggered at regular intervals
  - methods:
    - void **setBaseTime**(const SbTime &baseTime)
    - void **setInterval**(const SbTime &interval)

Timer sensors

- **SoAlarmSensor** and **SoTimerSensor** are the two timer-queued sensors provided in Inventor
- Skeleton of an application using these sensors:
  - Construct the sensor
  - Define the callback class with invoke method
  - Set timing parameters of the sensor
  - Schedule the sensor
- **SoAlarmSensor** will be triggered once at a specified time
  - methods:
    - void **setTime**(SbTime absTime)
    - void **setTimeFromNow**(SbTime relTime)
- **SoTimerSensor** will be triggered at regular intervals
  - methods:
    - void **setBaseTime**(SbTime baseTime)
    - void **setInterval**(SbTime interval)

Timer sensors example

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoRotation.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/sensors/SoTimerSensor.h>

// This function is called either 10 times/second or once every
// second; the scheduling changes every 5 seconds (see below):
static void rotatingSensorCallback(void *data, SoSensor *)
{
   // Rotate an object...
   SoRotation *myRotation = (SoRotation *)data;
   SbRotation currentRotation = myRotation->rotation.getValue();
   currentRotation = SbRotation(
            SbVec3f(0,0,1), M_PI/90.0) * currentRotation;
   myRotation->rotation.setValue(currentRotation);
}

// This function is called once every 5 seconds, and
// reschedules the other sensor.
static void schedulingSensorCallback(void *data, SoSensor *)
{
   SoTimerSensor *rotatingSensor = (SoTimerSensor *)data;
   rotatingSensor->unschedule();
   if (rotatingSensor->getInterval() == 1.0)
      rotatingSensor->setInterval(1.0/10.0);
   else rotatingSensor->setInterval(1.0);
   rotatingSensor->schedule();
}
```

```
void main(int argc, char **argv)
{
   if (argc != 2) {
     fprintf(stderr, "Usage: %s filename.iv\n", argv[0]);
     exit(1);
   }

   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

  SoRotation *myRotation = new SoRotation;
   root->addChild(myRotation);

   SoTimerSensor *rotatingSensor =
      new SoTimerSensor(rotatingSensorCallback, myRotation);
   rotatingSensor->setInterval(1.0); // scheduled once per second
   rotatingSensor->schedule();

   SoTimerSensor *schedulingSensor =
      new SoTimerSensor(schedulingSensorCallback, rotatingSensor);
   schedulingSensor->setInterval(5.0); // once per 5 seconds
   schedulingSensor->schedule

   SoInput inputFile;
   if (inputFile.openFile(argv[1]) == FALSE) {
     fprintf(stderr, "Could not open file %s\n", argv[1]);
     exit(1);
   }
   root->addChild(SoDB::readAll(&inputFile));

   SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Two Timers");
   myViewer->show();

   SoXt::show(myWindow);  // Display main window
   SoXt::mainLoop();        // Main Inventor event loop
}
```

Timer sensors example

```
import java.awt.* ;
import java.awt.event.* ;

import com.tgs.inventor.*;
import com.tgs.inventor.sensors.* ;
import com.tgs.inventor.nodes.* ;
import com.tgs.inventor.awt.* ;
import com.tgs.inventor.misc.callbacks.SoSensorCB ;

public class TimerSensorExample {
  static private SoTimerSensor rotatingSensor;
  static private SoTimerSensor schedulingSensor;

  static class RotatingSensorCallback extends SoSensorCB {

    private SoRotation myRotation;
    public RotatingSensorCallback(SoRotation rot) { myRotation = rot; }

    public void invoke (SoSensor ted) {
      // Rotate an object...
      SbRotation currentRotation = myRotation.rotation.getValue() ;
      currentRotation.multiply(new SbRotation(new SbVec3f(0,0,1), (float)
Math.PI/90F)) ;
      myRotation.rotation.setValue(currentRotation) ;
    }
  }

  static class SchedulingSensorCallback extends SoSensorCB {
    public void invoke (SoSensor ted) {
      // This function is called once every 5 seconds, and
      // reschedules the other sensor.
      rotatingSensor.unschedule() ;
      if (rotatingSensor.getInterval().equals(new SbTime(1)))
        rotatingSensor.setInterval(new SbTime(1F/10F)) ;
      else
        rotatingSensor.setInterval(new SbTime(1)) ;
      rotatingSensor.schedule() ;
    }
  }
```

```
 public static void main (String [] argv) {
   // define the scene-graph root
   SoSeparator root = new SoSeparator();

   SoRotation myRotation = new SoRotation();
   root.addChild(myRotation);

   root.addChild(new SoCone());

   rotatingSensor = new SoTimerSensor(new
RotatingSensorCallback(myRotation),null);
   rotatingSensor.setInterval(new SbTime(1.0)); // scheduled once per second
   rotatingSensor.schedule();

   schedulingSensor = new SoTimerSensor(new SchedulingSensorCallback(),null);
   schedulingSensor.setInterval(new SbTime(5.0)); // once per 5 seconds
   schedulingSensor.schedule();

   SwSimpleViewer viewer = new SwSimpleViewer();
   viewer.setSceneGraph(root);
   viewer.viewAll();

   WindowListener l = new WindowAdapter() {
     public void windowClosing(WindowEvent e) {
       System.exit(0);
     }
   };

   Frame f = new Frame("");
   f.addWindowListener(l);
   f.add(viewer) ;
   f.pack() ;
   f.setVisible(true) ;
 }

}
```

Engine classes

- SoBase
  - SoFieldContainer
    - SoEngine
      - SoBoolOperation
      - SoCalculator
      - SoComposeMatrix
      - SoComposeRotation
      - SoComposeRotationFromTo
      - SoComposeVec2(3)(4)f
      - SoComputeBoundingBox
      - SoConcatenate
      - SoCounter
      - SoDecomposeMatrix
      - SoDecomposeRotation
      - SoDecomposeVec2(3)(4)f
      - SoElapsedTime
      - SoGate
      - SoInterpolate
        - SoInterpolateFloat
        - SoInterpolateRotation
        - SoInterpolateVec2(3)(4)f
      - SoOnOff
      - SoOneShot
      - SoSelectOne
      - SoTimerCounter
      - SoTransformVec3f
      - SoTriggerAny
      - SoUnknownEngine

Engines

- Engines are Inventor nodes which can be used to animate part of the scene or to constrain some elements of the scene to each other.
- Unlike sensors, they have built-in functions and are part of the scenegraph. They are written to .iv files and are read back.
- Engines receive input values (**SoField**) and send output values (**SoEngineOutput**) when their input values change.
- Engine input and output can be connected to fields using **connectFrom** method from SoField class. Engines can also be connected together to build an engine network. Each time an engine outputs new values, the connected fields are updated.
- **Arithmetic**, **Animation**, **Triggered** and **Array Manipulation** are the four types of engines in Inventor.

Arithmetic engines

- Arithmetic engines perform arithmetic operations on their inputs, then output the results.
- For instance, the SoBoolOperation engine performs a Boolean operation on two inputs and outputs the result.
  - Input:
    - SoMFBool **a**
    - SoMFBool **b**
    - SoMFEnum **operation** (SET, A, NOT_A, A_OR_B...)
  - Output:
    - SoMFBool **output**
    - SoMFBool **inverse**

C++

```
SoBoolOperation *myEngine = new SoBoolOperation;
myEngine->a.connectFrom(&node1->field);
myEngine->b.connectFrom(&node2->field);
myEngine->operation= SoBoolOperation::A_OR_B;
node3->field.connectFrom(&myEngine->output);
```

Java

```
SoBoolOperation myEngine = new SoBoolOperation();
myEngine.a.connectFrom(node1.field);
myEngine.b.connectFrom(node2.field);
myEngine.operation.setValue(SoBoolOperation.A_OR_B);
node3.field.connectFrom(myEngine.output);
```

Animation engines

- Animation engines can be used to animate objects in the scene graph. Each of them has a **timeIn** field which is connected to the **realTime** global field (default).
  - Reminder: **SoRotor, SoShuttle, SoPendulum, SoBlinker** nodes can be used for simple animations.
- **SoElapsedTime** functions as a stop watch.
  - Input:
    - SoSFTime **timeIn**
    - SoSFFloat **speed** (scale factor for timeout)
    - SoSFBool **on**
    - SoSFBool **pause**
  - Output:
    - SoSFTime **timeout**
- **SoOneShot** runs for a preset amount of time and stops.
- **SoTimeCounter** cycles from a minimum to a maximum count with a given frequency.

Triggered engines

- **SoCounter**, **SoOnOff** and **SoTriggerAny** engines have a trigger input field. They ouput a value only when they are triggered, i.e. when their trigger field is changed using the **touch()** or **setValue()** method. This trigger field can be seen as a start button which activates the engine.
- **SoCounter** outputs numbers from a minimum to a maximum value, increasing by a step value.
- **SoOnOff** switches on and off its output.
- **SoTriggerAny** triggers its output whenever one of its 10 input triggers is touched.
- **SoGate** engine is an engine filter which allows continuous flow of its input to its output if its **enable** field is TRUE; If enable is FALSE, the engine will output only if its trigger field is touched. Input and output fields can be any **SoType.**

Engine example

```
// Flower group
  SoSeparator *flowerGroup = new SoSeparator;
  root->addChild(flowerGroup);

  // Read the flower object from a file and add to the group
  if (!myInput.openFile("../../data/flower.iv"))
     exit (1);
  SoSeparator *flower= SoDB::readAll(&myInput);
  if (flower == NULL) exit (1);

  // Set up the flower transformations
  SoTranslation *danceTranslation = new SoTranslation;
  SoTransform *initialTransform = new SoTransform;
  flowerGroup->addChild(danceTranslation);
  initialTransform->scaleFactor.setValue(10., 10., 10.);
  initialTransform->translation.setValue(0., 0., 5.);
  flowerGroup->addChild(initialTransform);
  flowerGroup->addChild(flower);

  // Set up an engine to calculate the motion path:
  // r = 5*cos(5*theta); x = r*cos(theta); z = r*sin(theta)
  // Theta is incremented using a time counter engine,
  // and converted to radians using an expression in
  // the calculator engine.
  SoCalculator *calcXZ = new SoCalculator;
  SoTimeCounter *thetaCounter = new SoTimeCounter;

  thetaCounter->max = 360;
  thetaCounter->step = 4;
  thetaCounter->frequency = 0.075;

  calcXZ->a.connectFrom(&thetaCounter->output);
  calcXZ->expression.set1Value(0, "ta=a*M_PI/180"); // theta
  calcXZ->expression.set1Value(1, "tb=5*cos(5*ta)"); // r
  calcXZ->expression.set1Value(2, "td=tb*cos(ta)"); // x
  calcXZ->expression.set1Value(3, "te=tb*sin(ta)"); // z
  calcXZ->expression.set1Value(4, "oA=vec3f(td,0,te)");
  danceTranslation->translation.connectFrom(&calcXZ->oA);
```

Engine example

```
import java.awt.*;
import java.awt.event.*;

import com.tgs.inventor.*;
import com.tgs.inventor.awt.*;
import com.tgs.inventor.nodes.*;
import com.tgs.inventor.engines.*;

class EngineExample {
  public static void main(String[] argv) {
    // Create the scene graph
    SoSeparator root = new SoSeparator();

    // Set up the flower transformations
    SoTranslation danceTranslation = new SoTranslation();
    SoTransform initialTransform = new SoTransform();
    root.addChild(danceTranslation);
    initialTransform.scaleFactor.setValue(10,10,10);
    initialTransform.translation.setValue(0,0,5);
    root.addChild(initialTransform);
    root.addChild(new SoCylinder());

    SoTimeCounter thetaCounter = new SoTimeCounter();
    thetaCounter.max.setValue((short)180);
    thetaCounter.step.setValue((short)4);
    thetaCounter.frequency.setValue(0.075f);

    SoCalculator calcXZ = new SoCalculator();
    calcXZ.a.connectFrom(thetaCounter.output);
    calcXZ.expression.set1Value(0, "ta=a*M_PI/180"); // theta
    calcXZ.expression.set1Value(1, "tb=5*cos(5*ta)"); // r
    calcXZ.expression.set1Value(2, "td=tb*cos(ta)"); // x
    calcXZ.expression.set1Value(3, "te=tb*sin(ta)"); // z
    calcXZ.expression.set1Value(4, "oA=vec3f(td,0,te)");
    danceTranslation.translation.connectFrom(calcXZ.oA);

    SwSimpleViewer viewer = new SwSimpleViewer();
    viewer.setSceneGraph(root);
    viewer.viewAll();

    …
  }
}
```

Exercise 3

3

- Using second example scene graph, add engines nodes to animate the pen along a XY trajectory curve. A sensor is used to update a 3D text string giving pen XY coordinates and to draw the pen trajectory.
  - See Appendix 3 Tutorial3 for solution

Event handling

- Inventor translates window system-dependent events to **SoEvent** class event, using an **SoSceneManager** object. This object is included in the SoXtRenderArea object.
- For each event received the scene manager creates an instance of the **SoHandleEventAction**.
- This action goes through the scene graph and stops whenever a node which wants to deal with this event is found.
- **SoSelection** node can be inserted in the scene graph for picking.
- **SoEventCallback** node can be inserted in the scene graph. If this node is traversed by the SoHandleEventAction and it selects the current event, then the attached callback is called.
- An Inventor application can directly deal with events before translation by the scene manager, by calling **setEventCallback** method of the SoXtRenderArea object.

SoSelection node

- An **SoSelection** node may be inserted near the root of the scene graph. All children of this node may be selected.
- This node maintains a **selection list**, which is a list of paths starting with the selection node and ending with the selected object.
- This list can be updated by **select**, **deselect**, **toggle**, **deselectAll** methods or when a left mouse button event “picks” an object of the scene.
- This list can be inquired using **isSelected**, **getNumSelected**, **getList**, **getPath** methods.
- SoSelection **policy** field tells Inventor how to update the list when the left mouse button is pressed on a node. The node path may be added to the list after clearing it or not, the node may be deleted from the list if already in it.
- Selected paths in the selection list may be highlighted using the **setGLRenderAction** method of SoXtRenderArea class.

SoSelection sample

```
void mySelectionCB(void *, SoPath *selectionPath)
{
  printf(selectionPath->getTail()->getTypeId().getName().getString);
}

SoPath *pickFilterCB(void *, const SoPickedPoint *pick)
{
  // See which child of selection got picked
  SoPath *p = pick->getPath();
  SbVec3f *v  = pick->getPoint(); // can also retrieve normals, texcoords...
  int i;
  for (i = 0; i < p->getLength() - 1; i++) {
    SoNode *n = p->getNode(i);
    if (n->isOfType(SoSelection::getClassTypeId()))
      break;
  }
  // Copy 2 nodes from the path: selection and the picked child
  return p->copy(i, 2);
}

main()
{
  // ...
  SoSelection *selectionRoot = new SoSelection;
  selectionRoot->ref();
  selectionRoot-> addSelectionCallback(mySelectionCB);
  selectionRoot->setPickFilterCallback(pickFilterCB);

  // Add the scene graph (SoSelection is derived from SoSeparator)
  selectionRoot->addChild(new SoCone);
  // ...
  // To get automatic highlighting :
  viewer->setGLRenderAction(new SoBoxHighlightRenderAction());
  viewer->redrawOnSelectionChange(selectionRoot);
  // ...
}
```

SoSelection example1

```
import …
import com.tgs.inventor.actions.*;

class SelectionHighlight {
  public static void main(String[] argv) {
    new SelectionHighlight();
  }

  SelectionHighlight() {
    // Create and set up the selection node
    SoSelection selectionRoot = new SoSelection() ;

    // Add some geometry to the scene
    // a cube
    SoSeparator cubeRoot = new SoSeparator();
    SoTransform cubeTransform = new SoTransform();
    SoCube cube = new SoCube();
    cubeRoot.addChild(cubeTransform);
    cubeRoot.addChild(cube);
    cubeTransform.translation.setValue(-2, 2, 0);
    selectionRoot.addChild(cubeRoot);

    // a sphere
    SoSeparator sphereRoot = new SoSeparator();
    SoTransform sphereTransform = new SoTransform();
    SoSphere sphere = new SoSphere();
    sphereRoot.addChild(sphereTransform);
    sphereRoot.addChild(sphere);
    sphereTransform.translation.setValue(2, 2, 0);
    selectionRoot.addChild(sphereRoot);

    SwSimpleViewer viewer = new SwSimpleViewer();
    SwRenderArea area = viewer.getArea();
    area.setGLRenderAction(new SoBoxHighlightRenderAction());
    area.redrawOnSelectionChange(selectionRoot);
    viewer.setSceneGraph(selectionRoot);
    viewer.viewAll();

    …
  }
}
```

SoSelection example2

```
class SelectionPicked {
  SoCoordinate3 markerCoord;

  class myPickCB extends SoSelectionPickCB {
    public SoPath invoke(SoPickedPoint p) {
      markerCoord.point.setValue(p.getPoint());
      return null;
    }
  }

  SelectionPicked() {
    // Create and set up the selection node
    SoSelection selectionRoot = new SoSelection() ;
    selectionRoot.setPickFilterCallback(new myPickCB() );

    // a cube (PICKABLE)
    SoSeparator cubeRoot = new SoSeparator();
    SoTransform cubeTransform = new SoTransform();
    cubeTransform.translation.setValue(-2, 2, 0);
    cubeRoot.addChild(cubeTransform);
    cubeRoot.addChild(new SoCube());
    selectionRoot.addChild(cubeRoot);

    // a sphere (UNPICKABLE)
    SoSeparator sphereRoot = new SoSeparator();
    SoPickStyle pickStyle = new SoPickStyle();
    pickStyle.style.setValue(SoPickStyle.UNPICKABLE);
    sphereRoot.addChild(pickStyle);
    SoTransform sphereTransform = new SoTransform();
    sphereTransform.translation.setValue(2, 2, 0);
    sphereRoot.addChild(sphereTransform);
    sphereRoot.addChild(new SoSphere());
    selectionRoot.addChild(sphereRoot);

    // a marker at the picked point
    markerCoord = new SoCoordinate3();
    markerCoord.point.setValue(0,0,0);
    SoMaterial markerMat = new SoMaterial();
    markerMat.diffuseColor.setValue(1,0,0);
    SoMarkerSet marker = new SoMarkerSet();
    marker.markerIndex.setValue(SoMarkerSet.STAR_9_9);
    selectionRoot.addChild(markerMat);
    selectionRoot.addChild(markerCoord);
    selectionRoot.addChild(marker);
  …
  }
}
```

Exercise 4

4

- Using third example, add picking facility to turn on and off animation of the pen by picking on the main part of the pen. Then picking on end part of the pen activate or desactivate a material editor to change color of the pen and then of the trajectory curve.
  - See Appendix 4 Tutorial4 for solution

Dragger classes

SoBase

SoFieldContainer

SoNode

SoBaseKit

SoInteractionKit

SoDragger

- SoCenterBallDragger
- SoDirectionalLightDragger
- SoDragPointDragger
- SoHandleBoxDragger
- SoJackDragger
- SoPointLightDragger
- SoRotateCylindricalDragger
- SoRotateDiscDragger
- SoRotateSphericalDragger
- SoScale1Dragger
- SoScale2Dragger
- SoScale2UniformDragger
- SoScaleUniformDragger
- SoSpotLightDragger
- SoTabBoxDragger
- SoTabPlaneDragger
- SoTrackBallDragger
- SoTransformBoxDragger
- SoTranslate1Dragger
- SoTranslate2Dragger

Draggers

- Draggers are nodes in the scene graph with a built-in user interface. They insert their geometry in the scene and react to user events.
- The main use of draggers is to connect them to other node fields, and then use them as input devices equivalent to sliders for instance.

```
#include <Inventor/engines/SoCompose.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/draggers/SoTranslate1Dragger.h>

void main(int , char **argv)
{
  Widget myWindow = SoXt::init(argv[0]);
  if (myWindow == NULL) exit(1);

  SoSeparator *root = new SoSeparator;
  root->ref();

  // Create myDragger with an initial translation of (1,0,0)
  SoTranslate1Dragger *myDragger =
     new SoTranslate1Dragger;
  root->addChild(myDragger);
  myDragger->translation.setValue(1,0,0);
```

```
  // Place an SoCone above myDragger
  SoTransform *myTransform = new SoTransform;
  SoCone      *myCone = new SoCone;
  root->addChild(myTransform);
  root->addChild(myCone);
  myTransform->translation.setValue(0,3,0);

  // SoDecomposeVec3f engine extracts myDragger's x-
component
  // The result is connected to myCone's bottomRadius.
  SoDecomposeVec3f *myEngine = new SoDecomposeVec3f;
  myEngine->vector.connectFrom(&myDragger->translation);
  myCone->bottomRadius.connectFrom(&myEngine->x);

  // Display them in a viewer
  SoXtExaminerViewer *myViewer
    = new SoXtExaminerViewer(myWindow);
  myViewer->setSceneGraph(root);
  myViewer->setTitle("Dragger Edits Cone Radius");
  myViewer->viewAll();
  myViewer->show();

  SoXt::show(myWindow);
  SoXt::mainLoop();
}
```

Draggers

- Draggers are nodes in the scene graph with a built-in user interface. They insert their geometry in the scene and react to user events.
- The main use of draggers is to connect them to other node fields, and then use them as input devices equivalent to sliders for instance.

```
import java.awt.*;
import java.awt.event.*;

import com.tgs.inventor.*;
import com.tgs.inventor.awt.*;
import com.tgs.inventor.nodes.*;
import com.tgs.inventor.engines.*;
import com.tgs.inventor.draggers.*;

class DraggerExample {

  public static void main (String [] argv) {

    SoSeparator root = new SoSeparator();

    // Create myDragger with an initial translation of (1,0,0)
    SoTranslate1Dragger myDragger = new SoTranslate1Dragger();
    root.addChild(myDragger);
    myDragger.translation.setValue(1,0,0);

    // Place an SoCone above myDragger
    SoTransform myTransform = new SoTransform();
    SoCone myCone = new SoCone();
    root.addChild(myTransform);
    root.addChild(myCone);
    myTransform.translation.setValue(0,3,0);
```

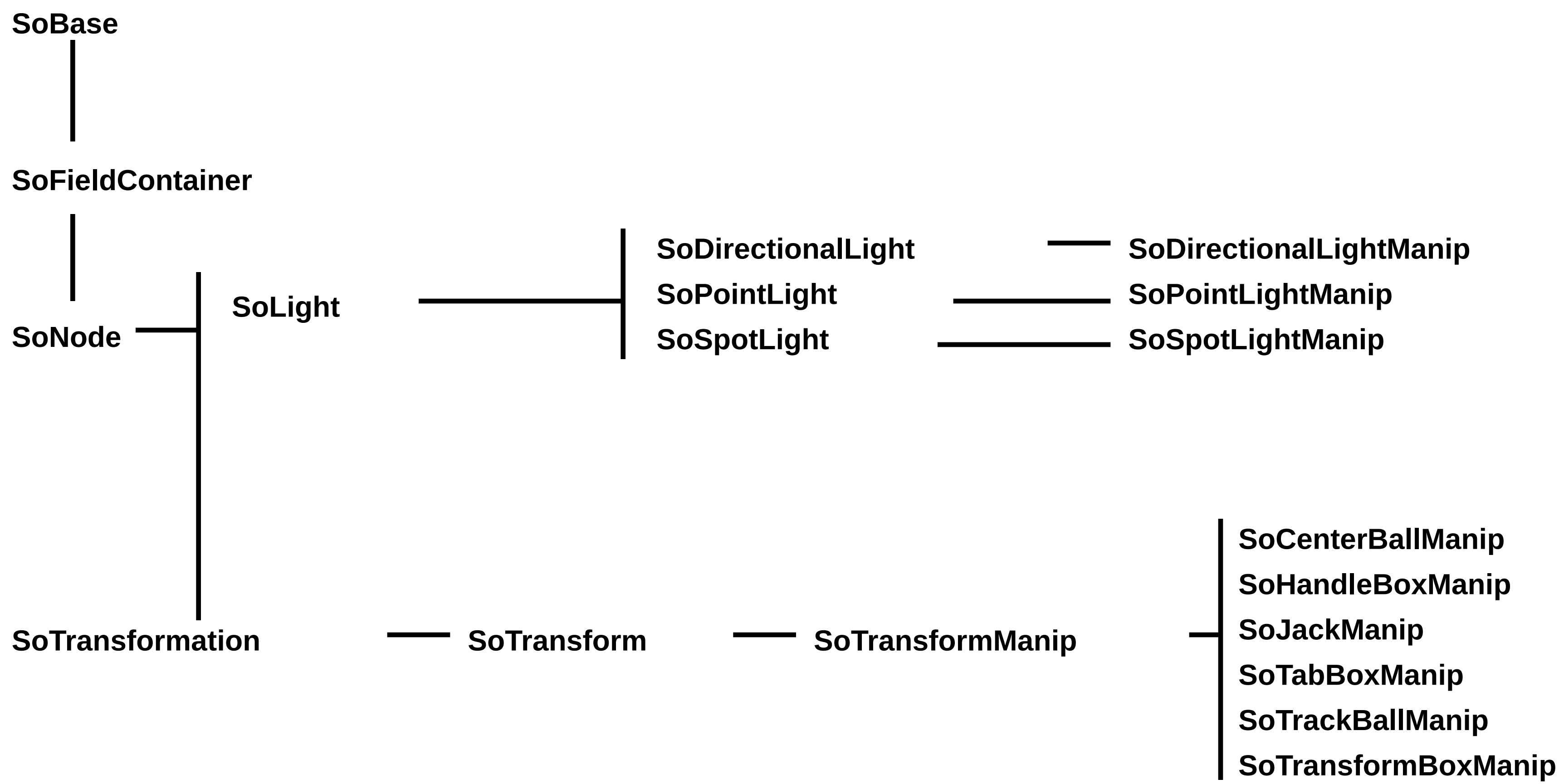
```
    // SoDecomposeVec3f engine extracts myDragger's x-component
    // The result is connected to myCone's bottomRadius.
    SoDecomposeVec3f myEngine = new SoDecomposeVec3f();
    myEngine.vector.connectFrom(myDragger.translation);
    myCone.bottomRadius.connectFrom(myEngine.x);

    SwSimpleViewer viewer = new SwSimpleViewer();
    viewer.setSceneGraph(root);
    viewer.viewAll();

    WindowListener l = new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
         System.exit(0);
       }
     };

    Frame f = new Frame("");
    f.addWindowListener(l);
    f.add(viewer) ;
    f.pack() ;
    f.setVisible(true) ;
  }
}
```

Exercise 5

5

- Using fourth example, add a dragger to change pen width.
  - See Appendix 5 Tutorial5 for solution

Manipulator classes



Manipulators

- Unlike draggers which are devices you can use to edit a field by connecting them, manipulators are **object editors**.
- Manipulators are sub-classes of other nodes. Internally they use draggers to interact and **edit themselves**.
- To use a manipulator:
  - Construct the manipulator (do not forget to reference it if you plan to use it again).
  - Use the **replaceNode** method to replace the node to edit with the corresponding manipulator.
  - Interact with the manipulator to edit the node.
  - Use the **replaceManip** method to restore the original node.

Action classes

- SoAction
  - SoCallbackAction
  - SoGLRenderAction
  - SoGetBoundingBoxAction
  - SoGetMatrixAction
  - SoHandleEventAction
  - SoPickAction
    - SoRayPickAction
  - SoSearchAction
  - SoWriteAction

Actions & misc.

- We have already seen **SoGLRenderAction** render the scene graph on the screen, and **SoHandleEventAction** dispatch events through the scene graph.
- **SoGetBoundingBoxAction** can be applied to get the bounding box of a scene graph.
- **SoWriteAction** is used to write a scene graph to a file.
- **SoOffscreenRenderer** is used to output the scene graph in EPS format. Use the **writeToPostScript**() method.

```
SbViewportRegion vp;
vp->setWindowSize(SbVec2s(300,400));
rootNode = getMyScene();
SoOffScreenRenderer renderer(vp);
renderer->render(rootNode);
renderer->writeToPostScript(stdout);
```

Java

```
SoSeparator root = getMyScene();
SbViewportRegion vp = new SbViewportRegion();
vp.setWindowSize(new SbVec2s((short)300,(short)400));

SoOffscreenRenderer renderer = new SoOffscreenRenderer(vp);
renderer.render(root);
 renderer.writeToPostScript(new FILE(...));
```

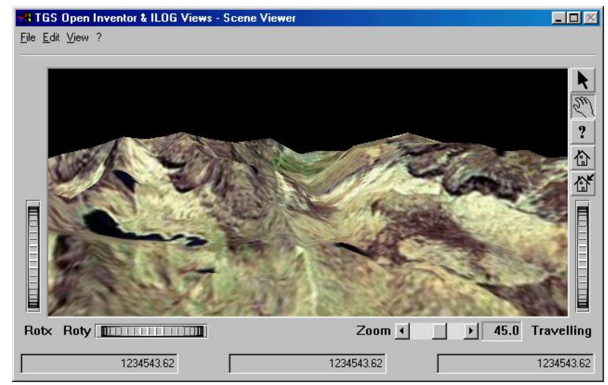
SoRayPickAction SoWriteAction sample

```
SbBool writePickedPath(SoNode *root,
                const SbViewportRegion &viewport,
                const SbVec2s &cursorPosition)
{
  SoRayPickAction myPickAction(viewport);

  // Set an 8-pixel wide region around the pixel
  myPickAction.setPoint(cursorPosition);
  myPickAction.setRadius(8.0);

  // Start a pick traversal
  myPickAction.apply(root);
  const SoPickedPoint *myPickedPoint =   myPickAction.getPickedPoint();
  if (myPickedPoint == NULL) return FALSE;

  // Write out the path to the picked object
  SoWriteAction myWriteAction;
  myWriteAction.getOutput()->openFile("output.iv");
  myWriteAction.getOutput()->setBinary(FALSE); // default
  myWriteAction.apply(myPickedPoint->getPath());
  myWriteAction.getOutput()->closeFile();

  return TRUE;
}
```

Open Inventor for VRML

- A simple VRML viewer

```
SoXtWalkViewer examinerViewer = new SoXtWalkViewer(SoXt::init();););
SoInput mySceneInput;
mySceneInput.openFile(«terrain.wrl» );
examinerViewer->setSceneGraph(SoDB::readAll(&mySceneInput));
mySceneInput.closeFile();
examinerViewer->setSceneGraph(sceneRoot);
examinerViewer->show();
SoXt::mainLoop();
```



- Converting OIV scene graph to VRML

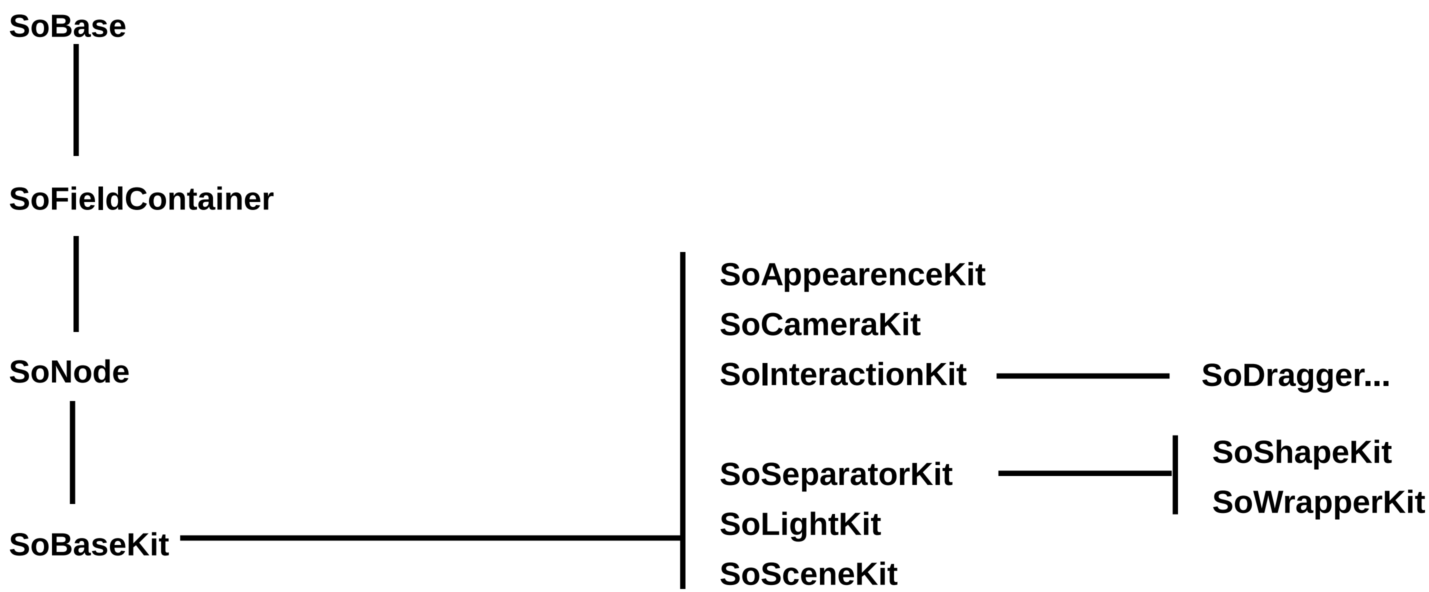
```
SoToVRML2Action *toVRML2Action = new SoToVRML2Action;
toVRML2->apply(umbrellaRoot);
SoOutput out;
out.openFile(outputfile);
out.setHeaderString(“#VRML V2.0 utf8”) ;
SoWriteAction writeAction(&out) ;
writeAction.apply((SoNode *) toVRML2->getVRML2SceneGraph();) ;
out.closeFile() ;
```

Exercise 6

6

- Using fith example, add a manipulator to change text orientation. The manipulator will be activated by picking the text, depressing “Q” letter will unactivate it. (Search action is used to retreive path to the manipulator).
  - See Appendix 6 Tutorial6 for solution

Node Kit classes



Node Kits

- **Node Kits** are collection of nodes
- Divided in parts, described in a **SoNodeKitCatalog**
- Parts include “hidden” children nodes, some are created by default.
- **SoPath** ends on node kits, cast to **SoFullPath** or **SoNodeKitPath** for more.
- **SoBaseKit** methods :
  - SoNode ***getPart**(SbName &partName, SbBool makeIfNeeded);
    // Convenience macros : **SO_GET_PART**(), **SO_CHECK_PART**()
  - SbBool **setPart**(SbName &partName, SoNode *newPart); // node
  - SbBool **set**(char *partName, char*parameters); // field
  - SbBool **set**(char *nameValuePairs); // field

```
SoShapeKit *myKit = new SoShapeKit;
myKit->setPart(“shape”, new SoText3);
myKit->set(”shape {part ALL string \”NICE\”}”);
myKit->set(“font”, “size 2”);
```

Node Kits

- **Node Kits** are collection of nodes
- Divided in parts, described in a **SoNodeKitCatalog**
- Parts include ‘‘hidden’’ children nodes, some are created by default.
- **SoPath.regular** ends on node kits, use SoPath.full or SoPath.nodekit members for more.
- **SoBaseKit** methods :
  - SoNode **getPart**(String partName, boolean makeIfNeeded);
    boolean **setPart**(String partName, SoNode newPart); // node
  - boolean **set**(String partName, String parameters); // field
  - boolean **set**(String nameValuePairs); // field

```
SoShapeKit myKit = new SoShapeKit();
myKit.setPart(‘‘shape’’, new SoText3());
myKit.set(’’shape {part ALL string \’’NICE\’’}’’);
myKit.set(‘‘font’’, ‘‘size 2’’);
```

Editor and viewer classes

- SoWinComponent
  - SoWinDirectionalLightEditor
  - SoWinMaterialEditor
  - SoWinGLWidget
    - SoWinRenderArea
      - SoWinViewer
        - SoWinFullViewer
          - SoWinConstrainedViewer
            - SoWinFlyViewer
            - SoWinWalkViewer
          - SoWinExaminerViewer
          - SoWinPlaneViewer

Editors

- Editors are independent windows which allow the user to edit interactively a node.
- Editors are attached to a specific node.
- To use an editor:
  - Construct the editor .
  - Use the **attach** method to link the editor with a node.
  - Use the **show** (**setVisible** in Java) method to display the editor.

C++

```
SoMaterialEditor *myEditor = new SoMaterialEditor;
myEditor->attach (myMaterial);
myEditor->show();
```

Java

```
SwMaterialEditor myEditor = new SwMaterialEditor();
myEditor.attach (myMaterial);
myEditor.setVisible();
```

Exercise 7

7

- Using sixth example, add picking facility to activate or desactivate a material editor by picking on the end part of the pen. The material editor will change color of the pen.

- Mixing Open Inventor for Java and Swing

- Area (SwRenderArea) and viewers (SwViewer) use awt components to render the scene graph.

- Unfortunately mxing awt and swing component need some particular attention

- See http://java.sun.com/products/jfc/tsc/articles/mixing/index.html

- Use JPopupMenu.setDefaultLightWeightPopupEnabled(boolean)

- Mixing Open Inventor for Java and swing (example)

```
import javax.swing.*;
import java.awt.* ;
import java.awt.event.*;
import com.tgs.inventor.*;
import com.tgs.inventor.awt.* ;
import com.tgs.inventor.nodes.* ;

public class HelloConeSwing  {
  public HelloConeSwing() {
    // Make a scene containing a red cone
    SoSeparator root = new SoSeparator();
    root.addChild(new SoDirectionalLight());
    SoMaterial myMaterial = new SoMaterial();
    myMaterial.diffuseColor.setValue(1,0,0); // Red
    root.addChild(myMaterial);
    root.addChild(new SoCone());

    // Put the scene in viewer
    SwSimpleViewer viewer = new SwSimpleViewer();
    viewer.setSceneGraph(root);

    JPopupMenu.setDefaultLightWeightPopupEnabled(false);
    JMenu menu = new JMenu("MyMenu");
    menu.add(new JMenuItem("opt 1"));
    menu.add(new JMenuItem("opt 2"));
```

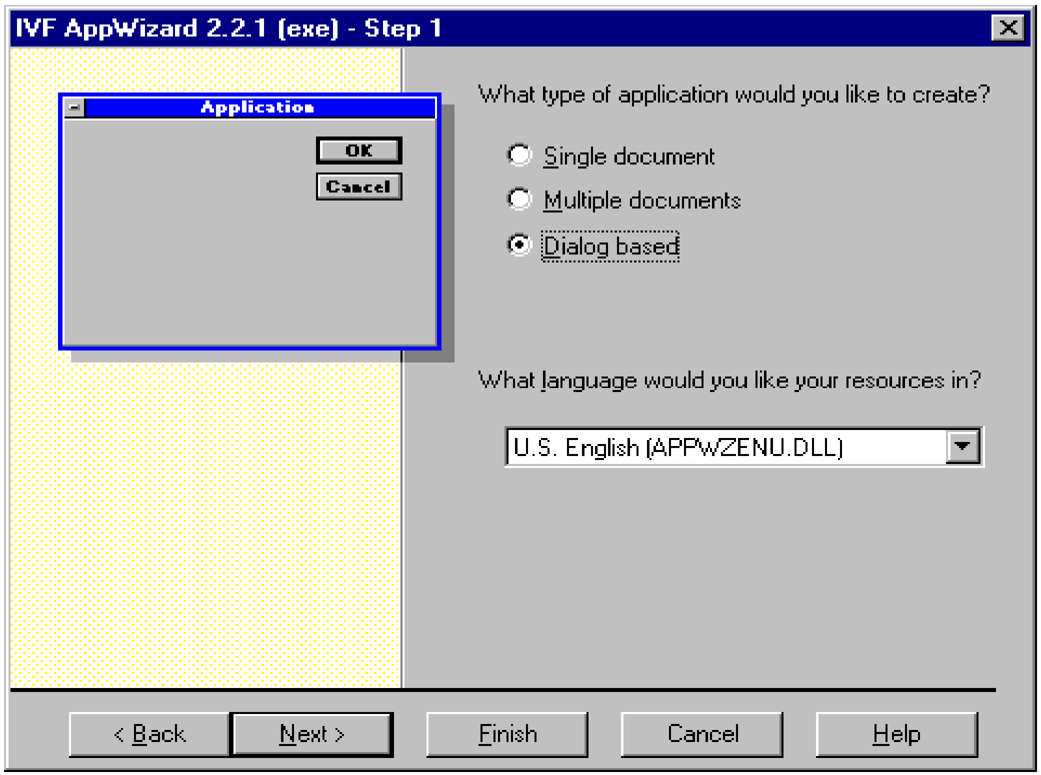
```
    JMenuBar menubar = new JMenuBar();
    menubar.add(menu);

    JFrame f = new JFrame ("HelloCone");
    f.addWindowListener(new WindowListener());
    f.getContentPane().add(viewer);
    f.setJMenuBar(menubar);
    f.pack();
    f.setVisible(true);
  }

  class WindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }

  public static void main(String argv[]) {
    new HelloConeSwing();
  }
}
```

IVF Framework

- The easiest way to integrate your Open Inventor application with Microsoft Developer Studio.

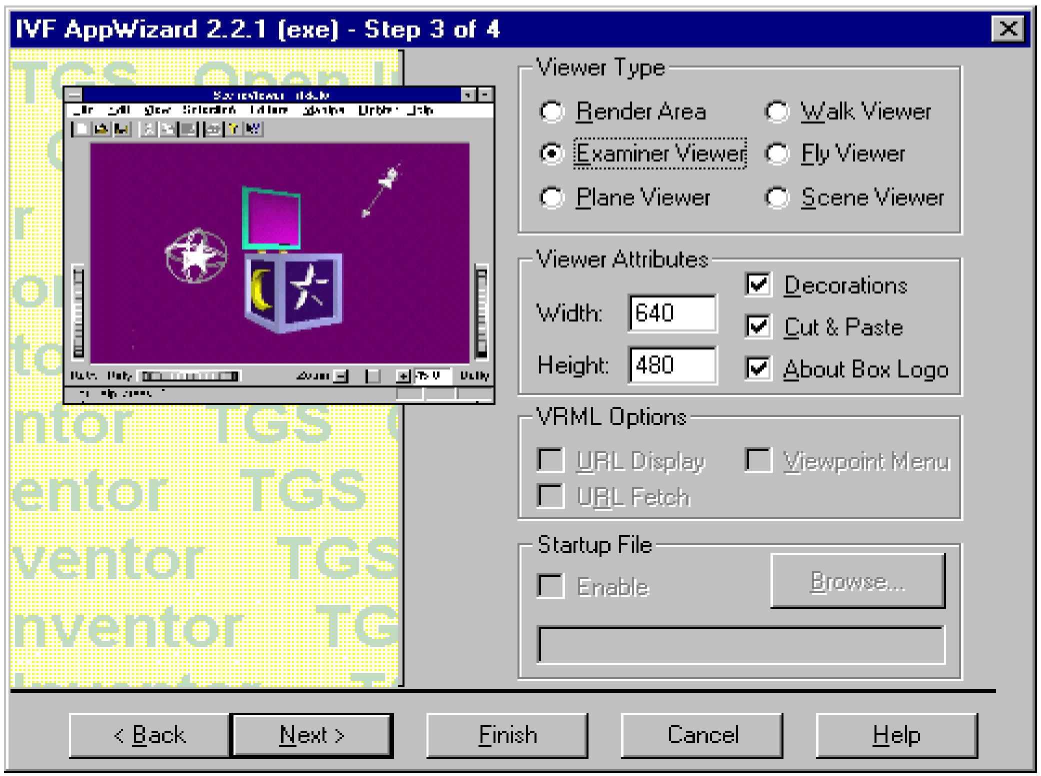
Simple Application with the IVF Wizard (1 of 4)

- Let IVF Wizard help you to write the skeleton of your application.
  - Select a dialog based application.



Simple Application with the IVF Wizard (2 of 4)

- Select the kind of viewer you want to use.
  - Select an examiner viewer.



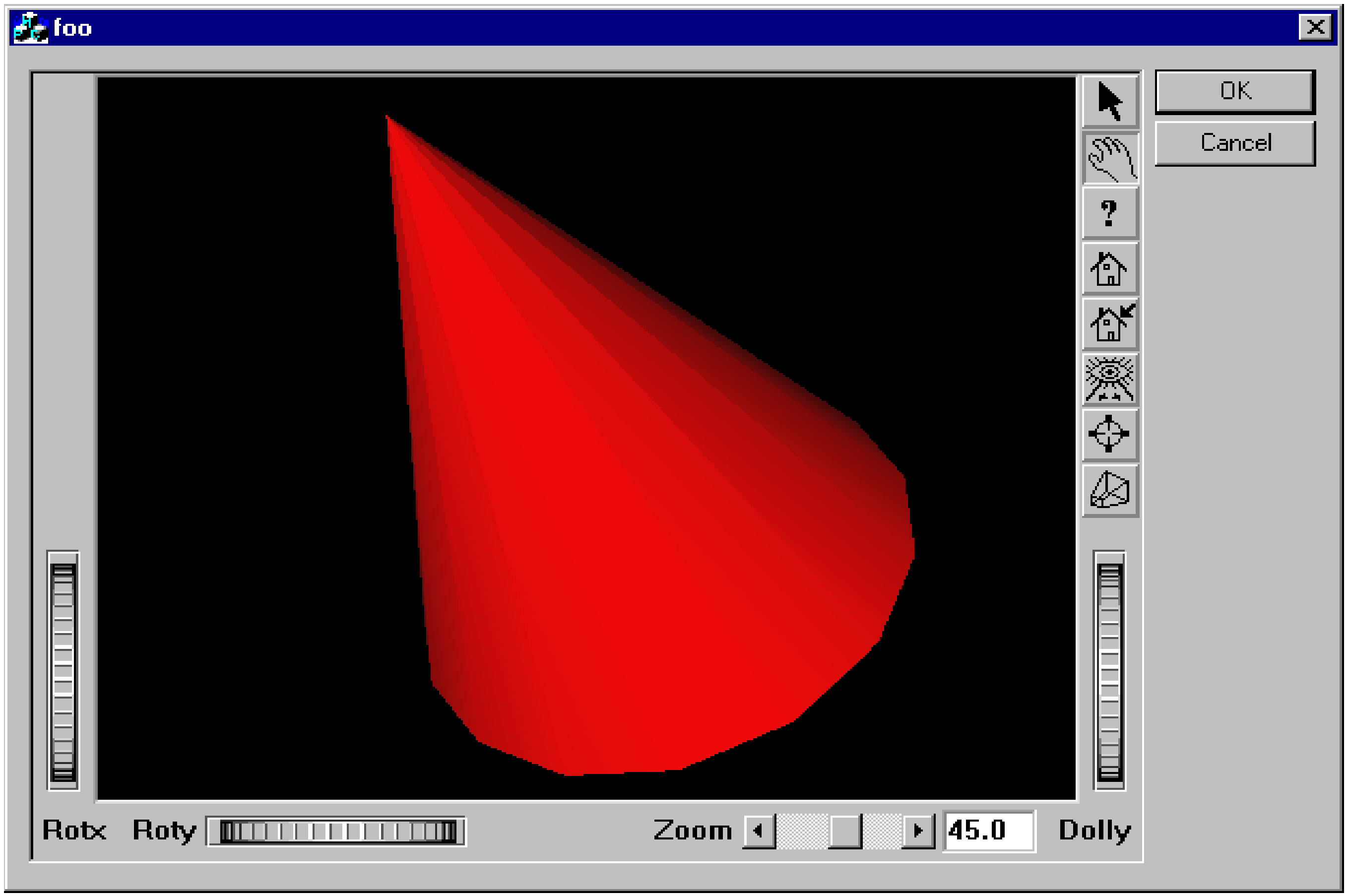
Simple Application with the IVF Wizard (3 of 4)

- Modify the code to personalize your application.
  - Write a small scene to display

```
...

#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoCone.h>

...

BOOL CFooDlg::OnInitDialog()
{
            ...

            // TODO: Add extra initialization here
            SoSeparator* root = new SoSeparator;
            root->ref();
            SoMaterial* mtl = new SoMaterial;
            mtl->diffuseColor.setValue (1, 0, 0);
            root->addChild (mtl);
            SoCone* cone = new SoCone;
            root->addChild (cone);

            IvfSetSceneGraph (root);

            ...
}
```

Simple Application with the IVF Wizard (4 of 4)

- Your application is ready to run



Using rendering areas and viewers in a dialog

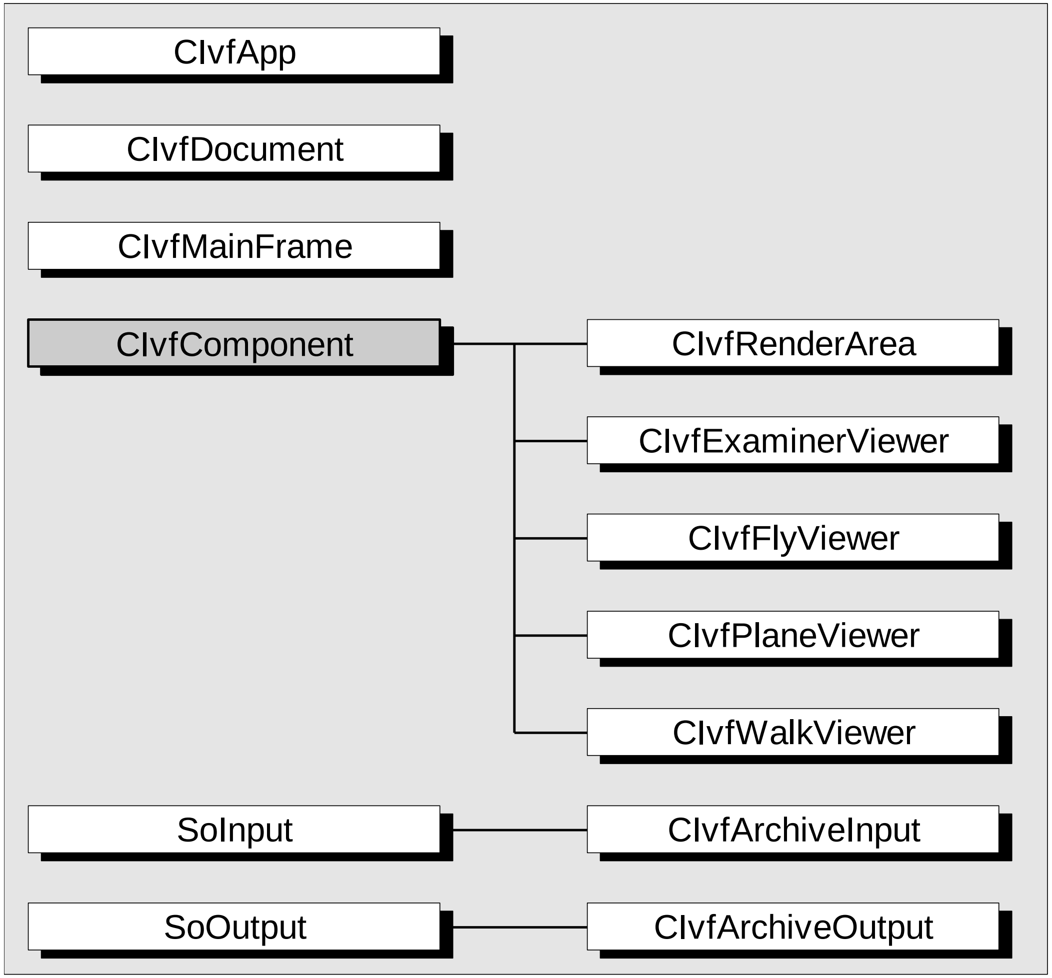
- If the dialog is the main window of the application, IVF wizard build the dialog for you.
- IVF Wizard allow you to make a choice between different kinds of viewers.
- You can add an inventor viewer in a dialog box by yourself:
  - Edit the dialog ressource, and add a *static text* control at the location you want to have your render.
  - Derived publicaly your dialog class from the IvfViewer class you want to have.
  - Add the following to the **OnInitDialog** method

```
Cdialog::OnInitDialog

static int cArgs[] = {  TRUE,      // Decoration
                        FALSE,    // Url display
                        FALSE,    // Viewpoints
                        FALSE};  // Url fetch
CIvfApp::IvfInitSoWin (this);
CWnd* pWnd = GetDlgItem (name_of_text_ressource);
IvfCreateComponent (pWnd, (void*) cArgs);
```

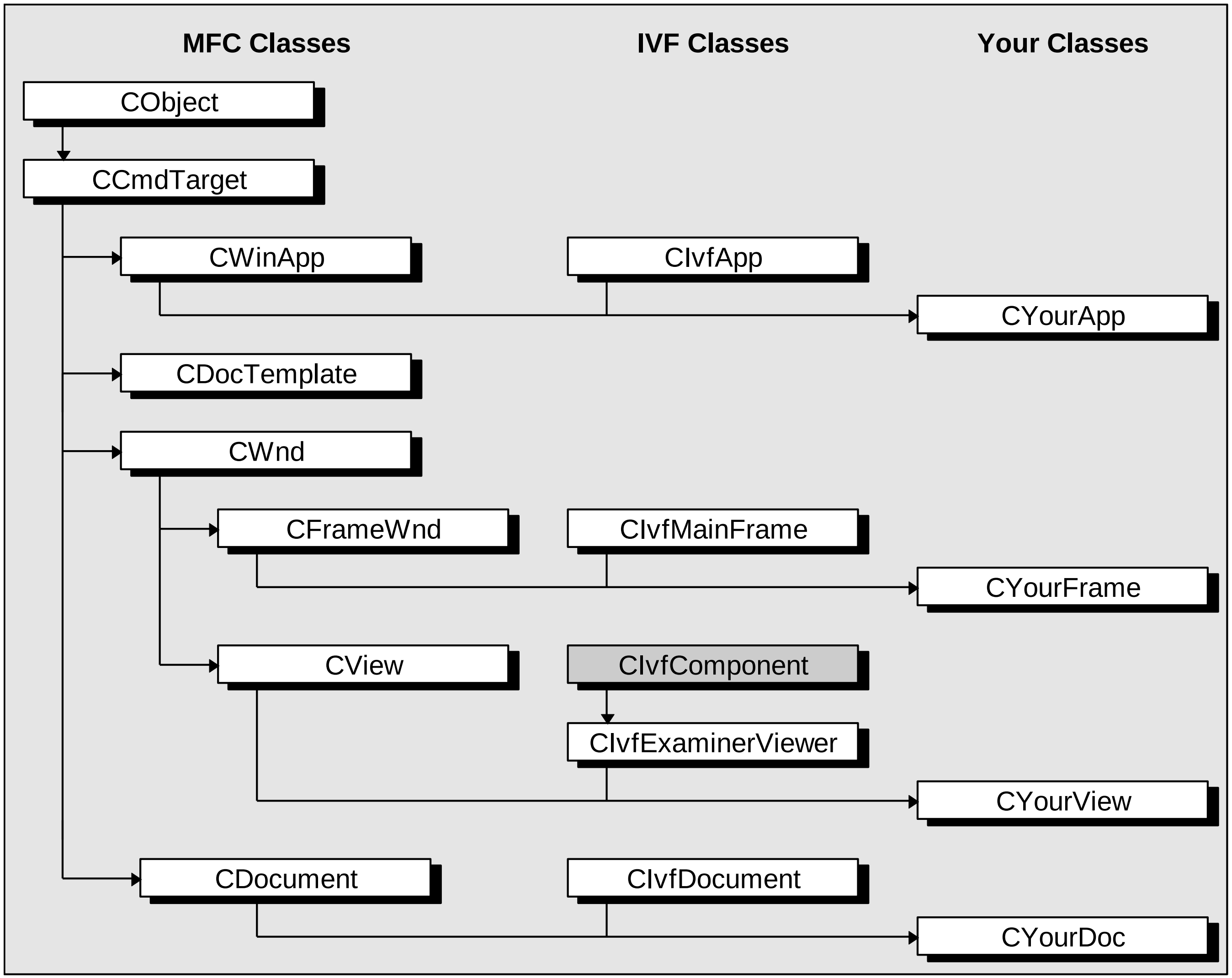
IVF framework

- The IVF library implement a framework similar to the MFC one.
- When you ask IVF wizard to build an application with multi document or single document interface, the documents and view created are derived from Ivf document and views.
- Ivf documents and views act exactly as MFC ones.
- This kind of applications allow you to use many MFC capabilities:
  - Cut and paste beetween applications,
  - Drag and drop of files on the application,
  - Recent file list saving,
  - etc...

IVF classes


CIvfApp
CIvfDocument
CIvfMainFrame
CIvfComponent
CIvfRenderArea
CIvfExaminerViewer
CIvfFlyViewer
CIvfPlaneViewer
CIvfWalkViewer
SoInput
CIvfArchiveInput
SoOutput
CIvfArchiveOutput

Example of IVF class hierarchy in a SDI application

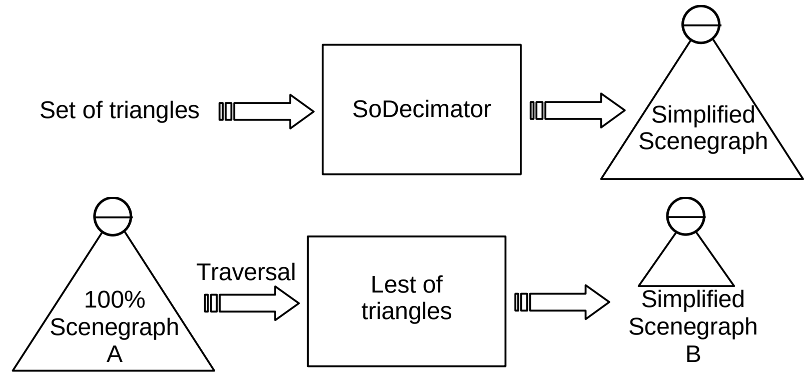


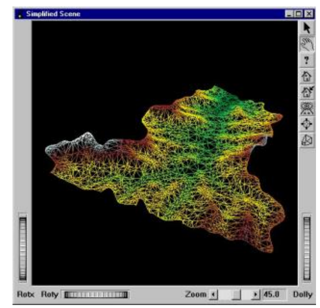
Example of Doc/View application with IVF

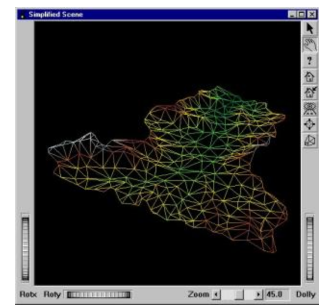
```
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSeparator.h>

BOOL CHelloConeDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

// BEGIN_IVWGEN
    IvfOnNewDocument();
// END_IVWGEN

    SoSeparator *root = new SoSeparator;
    root->ref();

    SoMaterial  *myMaterial = new SoMaterial;
    myMaterial->diffuseColor.setValue( 1., 0., 0. ); //Red
    root->addChild( myMaterial );
    root->addChild( new SoCone );

    IvfSetSceneGraph( root );
    root->unref();

    return TRUE;
}
```

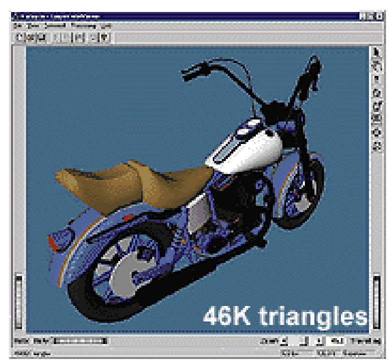
Large Models Visualisation

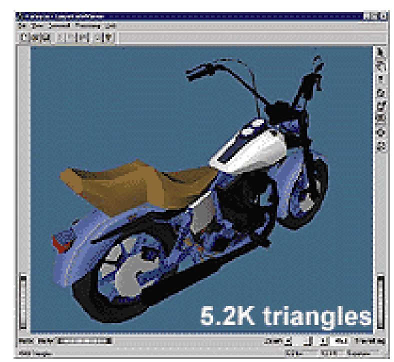
- Level of simplification
  - Automatic decimation
  - Global, shape, reorganize
- Spatial optimization
  - Octree ordering
  - Value ordering
- Adaptive viewing
  - setGoalFramePerSecond





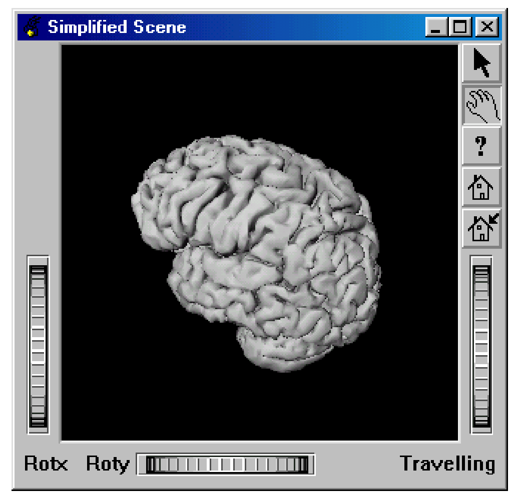






Open Inventor tip
Navigating faster in large scenes

```
SoXtExaminerViewer *myViewer = new SoXtExaminerViewer(myWindow);  // open a viewer
    SoSeparator *root = SoDB::readAll(&input);  // read a VRML or Inventor file

    SoGlobalSimplifyAction simplify(new SoDecimator); // global simplification
    static float decimation_levels[numLevels] =
          {1.0, 0.80 , 0.60, 0.40 , 0.20, 0.10, 0.01};
    simplify.setSimplificationLevels(numLevels, decimation_levels);
    simplify.apply(root);
    myViewer->setSceneGraph(root);

    // Associate callbacks to the popup menu
    myViewer >addStartCallback((SoXtViewerCB*)startViewerCB,(void*) myViewer);
    myViewer >addFinishCallback((SoXtViewerCB*)finishViewerCB,NULL);
    myViewer ->show();

    // During motion, if low resolution is selected, use simplified model
    static void startViewerCB (void *userData,SoXtViewer *viewer) {
        if(viewer->getDrawStyle(SoXtViewer::INTERACTIVE) ==
           SoXtViewer::VIEW_LOW_COMPLEXITY )
        {
            viewer->setDecimationStrategy(SoXtViewer::FIXED_PERCENTAGE);
            viewer->setFixedPercentage(0.2);
        }
    }
    // When the motion stop, switch back to full representation
    static void finishViewerCB(void *userData,SoXtViewer *viewer)
    {
        if (viewer->getDrawStyle(SoXtViewer::STILL)!=
            SoXtViewer::VIEW_LOW_COMPLEXITY )
              viewer->setDecimationStrategy(SoXtViewer::NORMAL);
    }
```



Collision detection samples

➔ SoIntersectionDetectionAction

```
SoIntersectionDetectionAction::Resp
onIntersection(void *,const SoIntersectingPrimitive *primitive1,
                      const SoIntersectingPrimitive *primitive2) {
  printf("%d %s\n",
    primitive1->path->getTail()->getTypeId().getName().getString());
  return SoIntersectionDetectionAction::NEXT_SHAPE;
}
// ...
SoIntersectionDetectionAction action;
action.addIntersectionCallback(onIntersection, NULL);
action.apply(root);
```

➔ SoCollisionManager

```
SoCollisionManager::Resp onCollision(void*,
      const SoCollidingPrimitive*, const SoCollidingPrimitive*) {
  flashObject();
  return SoCollisionManager::ABORT;
}
// ...
cm = new SoCollisionManager(objectPath, sceneRoot, transformNode);
cm->addCollisionCallback(onCollision);
```

➔ SoCollisionViewer

```
void onCollision(void *, SoXtCollisionViewer *) {
}
// ...
cv = new SoXtCollisionViewer(my_viewer);
cv->addCallback (onCollision, NULL);
```

Open Inventor DataViz and DialogMaster

- A great set of extension nodes to Open Inventor.

- Includes 4 modules :
  - GraphMaster : 2D/3D drawing and business graphics (2D/3D axis, curves, histograms, pie charts, generalized primitives, legends,…)
  - 3DDataMaster : scientific visualization (2D/3D meshes : contouring, cross sections, level surface, skeleton, skin, stream lines, probes, legends,...)
  - HardCopy/PlotMaster : vector printing (PostScript, HPGL, CGM, WMF, EMF,GDI).
  - DialogMaster : new user interface components available on Unix and Windows.

- All DatViz visualization nodes are nodekits with Inventor fields. Nodekit catalogs allow the user to set the appearence of a part of the node kit (for instance, setting the color of the arrow of an axis).

DataViz property classes

- Visualization classes depends on property classes (for their appearance, data retrieval...).

- There is two ways to define these properties :
  - Using Pbxxx classes. Pbxxx classes are not stored in the scene graph. Visualization classes have methods with a Pbxxx class as argument to refer to their properties.
  - Using property nodes. These nodes are stored in the scene graph and their use is similar to Open Inventor property nodes (SoMaterial, SoDrawStyle,…)

```
// Using Pbxxx classes

PbMiscTextAttr textAttr;
textAttr.setFontName("Courier");

PoAngularAxis *myAxis = new PoAngularAxis(.5,.0,2.5,1.,.0);
myAxis->setMiscTextAttr(&textAttr);
```

```
// Using property nodes

PoMiscTextAttr *textAttr = new PoMiscTextAttr;
textAttr->fontName = "Courier";

SoGroup *axisGroup = new SoGroup ;
PoAngularAxis *myAxis = new PoAngularAxis(.5,.0,2.5,1.,.0);

axisGroup->addChild(textAttr) ;
axisGroup->addChild(myAxis) ;
```

DataViz property classes

- Visualization classes depends on property classes (for their appearance, data retrieval...).
- Instance of property class are nodes stored in the scene graph and their use is similar to Open Inventor property nodes (SoMaterial, SoDrawStyle,…)
- Package com.tgs.dataviz.nodes

```
PoMiscTextAttr textAttr = new PoMiscTextAttr();
textAttr.fontName.setValue("Courier");

SoGroup axisGroup = new SoGroup() ;
PoAngularAxis myAxis = new PoAngularAxis();

axisGroup.addChild(textAttr) ;
axisGroup.addChild(myAxis) ;
```

DataViz property classes tree

C++ only

- **"Pb" property basic types classes**

- •**PbBase**
  - •PbDomain
  - •PbNumericDisplayFormat
  - •PbIsovaluesList
  - •**PbDataMapping**
    - •PbLinearDataMapping
    - •PbNonLinearDataMapping
    - •PbNonLinearDataMapping2
  - •PbDateFormatMapping
  - •PbMiscTextAttr

- Remark: Abstract classes appear in bold.

C++ and Java

Node property classes

•**PoNode**
- •**PoDataMapping**
  - •PoLinearDataMapping
  - •PoNonLinearDataMapping
- •PoNonLinearDataMapping2
- •PoDateFormatMapping
- •PoDomain
- •PoIsovaluesList
- •PoIsovaluesList
- •PoMiscTextAttr
- •PoNumericDisplayFormat

Remark: Abstract classes appear in bold.

DataViz *domain*

- The domain usually defines the data coordinate limits of graphics to be generated. Graph Master & 3D Data Master do not compute these limits, so this class defines them.

- In conceptual terms, a 2D domain (3D domain) is the smallest rectangle (parallelepiped) capable of containing the data for the image to be generated. The sides of this rectangle (parallelepiped) are parallel to the axis. Furthermore all Graph Master & 3D Data Master nodekits classes may be transformed according to the domain which they depend on.

- Some node fields are defined using the domain coordinates, for instance, the text height of axes or the arrow width at the end of an axis are defined as a percentage of the domain.

DataViz visualization classes

- GraphMaster and 3DDataMaster visualization classes are implemented using the Open Inventor **node kit** facility, that is they all derive from the node kit base class ***SoBaseKit***. This means that each new DataViz node is a **collection** of Open Inventor **nodes**.
- Each DataViz node kit are described by a node kit catalog. The user can access these parts to change the appearance of a node or the appearance of a part of the node kit.
- Each DataViz node kit has also **fields** to be **configured**.

C++

```
PoRectangle *myRectangle = new PoRectangle;
myRectangle->p.setValue(0.,0.);
myRectangle->q.setValue(10.,20.);
myRectangle->set("appearance.drawStyle", "style FILLED");
myRectangle->set("appearance.material", "transparency 0.9");
myRectangle->set("appearance.material", "diffuseColor 1 0
0");
```

Java

```
PoRectangle myRectangle = new PoRectangle();
myRectangle.p.setValue(0,0);
myRectangle.q.setValue(10,20);
myRectangle.set("appearance.drawStyle", "style FILLED");
myRectangle.set("appearance.material", "transparency 0.9");
myRectangle.set("appearance.material", "diffuseColor 1 0 0");
```

GraphMaster classe tree (1)

2D generalized primitives classes

- **PoBase**
  - **PoGraphMaster**
    - PoRectangle
    - PoParallelogram
    - **PoCircle**
      - PoCircleCenterRadius
      - PoCircleThreePoints
    - **PoCircleArc**
      - PoCircleArcCtrPtAngle
      - PoCircleArcCtrTwoPts
      - PoCircleArcThreePts
    - PoCircleArcCtrRadTwoAngle
    - PoArrow
    - PoCurve
    - PoLabelField
    - PoValuedMarkerField
    - PoErrorCurve
    - PoErrorPointField
    - PoBiErrorPointField
    - PoHighLowClose

Remark: Abstract classes appear in bold.

3D generalized primitives classes

- **PoBase**
  - **PoGraphMaster**
    - PoParallelogram3
    - **PoCircle3**
      - PoCircle3CenterRadius
      - PoCircle3ThreePoints
    - **PoCircleArc3**
      - PoCircleArc3CtrPtAngle
      - PoCircleArc3CtrTwoPts
      - PoCircleArc3ThreePts
    - PoArrow3
    - PoCurve3
    - PoCoordinateSystemAxis
    - PoPointsFieldBars

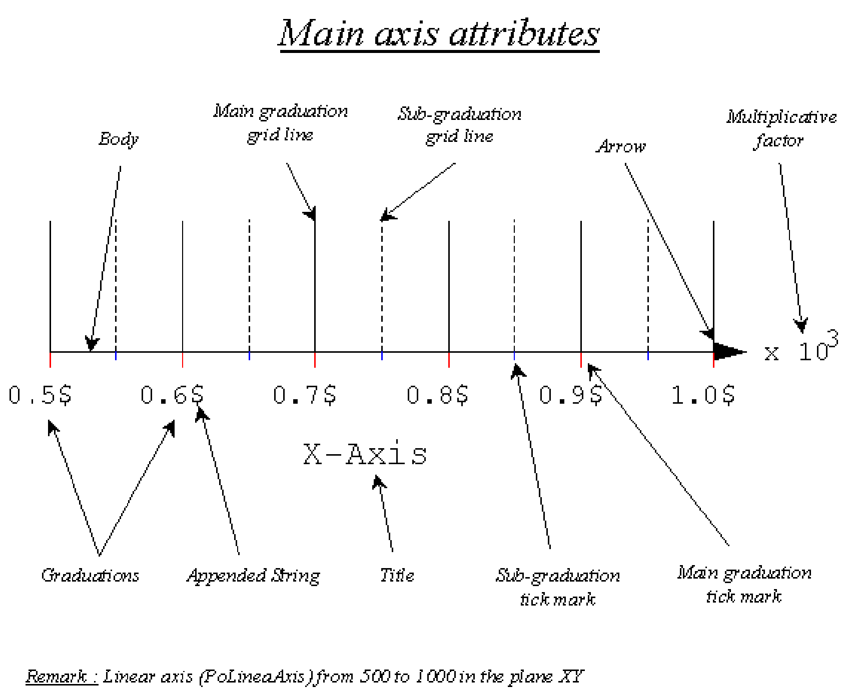
Remark: Abstract classes appear in bold.

GraphMaster classe tree (2)

Axis classes

- **PoBase**
  - **PoGraphMaster**
    - **PoBaseAxis**
      - **PoAxis**
        - **PoCartesianAxis**
          - PoLinearAxis
          - PoLogAxis
          - PoGenAxis
        - **PoAngularAxis**
        - **PoPolarAxis**
    - PoPolarLinAxis
    - PoPolarLogAxis
        - PoTimeAxis
      - PoGroup2Axis
      - PoGroup4Axis
      - PoGroup6Axis3
      - PoGroup3Axis3
      - PoAutoCubeAxis

Remark: Abstract classes appear in bold.



GraphMaster classe tree (3)

Legend classes

- **PoBase**
  - **PoGraphMaster**
    - **PoLegend**
      - PoItemLegend
      - **PoValueLegend**
        - **PoAutoValueLegend**
          - PoLinearValueLegend
    - PoNonLinearValueLegend1
    - PoNonLinearValueLegend2
    - PoNonLinearValueLegend3

Remark: Abstract classes appear in bold.

Histogram classes

- **PoBase**
  - **PoGraphMaster**
    - **PoHistogram**
      - PoSingleHistogram
    - PoMultipleHistogram

Remark: Abstract classes appear in bold.

Pie chart classes

- **PoBase**
  - **PoGraphMaster**
    - **PoPieChart**
      - PoPieChart2D
      - PoPieChart3D

Remark: Abstract classes appear in bold.

A GraphMaster example

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtPlaneViewer.h>
#include <Inventor/nodes/SoSeparator.h>
#include <graph/PoCurve.h>
#include <graph/PoLinearAxis.h>
# include <graph/PbDomain.h>
# include <nodes/PoDomain.h>

#define PI 3.1415
#define N_PT 50

int
main(int , char **argv)  {
  // Initialize Inventor and Xt
  Widget myWindow = SoXt::init(argv[0]) ;
  if (myWindow == NULL) exit(1) ;

  // Initialize DataViz
  PoBase::init() ;

  // Create the curve
  PoCurve *myCurve = new PoCurve;
  SbVec2f points[N_PT];
  double ang;
  int i;
  for (i=0, ang=0.; i < N_PT; i++, ang += 4.*PI/(double)N_PT)
    points[i].setValue(ang, 6. * sin(ang));
  myCurve->point.setValues(0,N_PT,points);
  myCurve->set("curvePointApp.material", "diffuseColor [1 0 0]");
```

```
  // Create simple automatic X and Y linear axis
  PoLinearAxis *myXAxis = new PoLinearAxis;
  myXAxis->start.setValue(SbVec3f(0.,0.,0.));
  myXAxis->end = 4.*PI;
  myXAxis->type = PoCartesianAxis::XY;
  myXAxis->set("appearance.material", "diffuseColor 0 0 0") ;
  PoLinearAxis *myYAxis = new PoLinearAxis;
  myYAxis->start.setValue(SbVec3f(0.,-6.,0.));
  myYAxis->end = 6.;
  myYAxis->type = PoCartesianAxis::YX;
  myYAxis->set("appearance.material", "diffuseColor 0 0 0") ;

  // Define domain
  PoDomain *myDom = new PoDomain;
  myDom->min = SbVec3f(0,-6,0);
  myDom->max = SbVec3f(4.*PI,6.,0);

  // Create the root of the scene graph
  SoSeparator *root = new SoSeparator;
  root->ref() ;
  root->addChild(myDom);
  root->addChild(myXAxis);
  root->addChild(myYAxis);
  root->addChild(myCurve);

  SoXtPlaneViewer *viewer = new SoXtPlaneViewer(myWindow);
  viewer->setSceneGraph(root);
  viewer->show();

  SoXt::show(myWindow);
  SoXt::mainLoop();
}
```

A GraphMaster example

```
import java.awt.* ;
import java.awt.event.*;

public class AxisExample {
  private static final int NUM_POINTS = 50 ;

  public AxisExample() {
    // Create the root of the scene graph
    SoAnnotation root = new SoAnnotation();

    // Create the curve
    PoCurve myCurve = new PoCurve();
    SbVec2f[] points = new SbVec2f[NUM_POINTS];
    double ang;
    int i;
    for(i=0, ang=0; i<NUM_POINTS; i++, ang +=
          4*Math.PI/(double)NUM_POINTS)
      points[i]= new SbVec2f((float)ang, (float)(6 * Math.sin(ang)));
    myCurve.point.setValues(0, points);
    myCurve.set("curvePointApp.material", "diffuseColor [1 0 0]");

    // Create simple automatic X and Y linear axis
    PoLinearAxis myXAxis = new PoLinearAxis();
    myXAxis.start.setValue(new SbVec3f(0, 0, 0));
    myXAxis.end.setValue((float)(4*Math.PI));
    myXAxis.type.setValue(PoCartesianAxis.XY);
    myXAxis.set("appearance.material", "diffuseColor 0 1 0") ;

    PoLinearAxis myYAxis = new PoLinearAxis();
    myYAxis.start.setValue(new SbVec3f(0, -6, 0));
    myYAxis.end.setValue(6);
    myYAxis.type.setValue(PoCartesianAxis.YX);
    myYAxis.set("appearance.material", "diffuseColor 0 1 0") ;
```

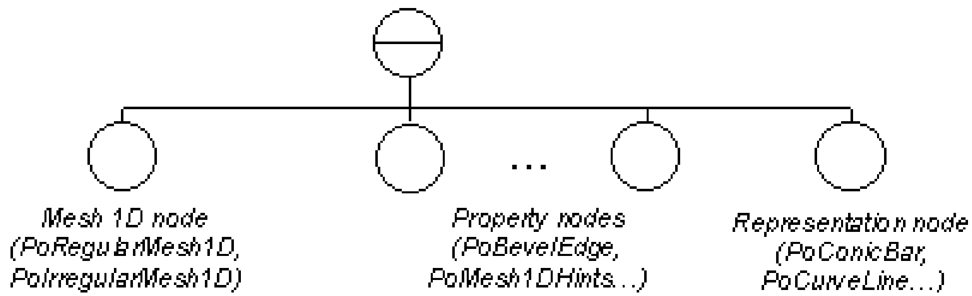
```
    // Do not forget to set the domain if you want
    // default parameters to be correct...
    PoDomain myDom = new PoDomain();
    myDom.min.setValue(new SbVec3f(0,-6,0));
    myDom.max.setValue(new SbVec3f((float)(4*Math.PI), 6, 0));

    root.addChild(myDom);
    root.addChild(myXAxis);
    root.addChild(myYAxis);
    root.addChild(myCurve);

    SwSimpleViewer viewer = new
SwSimpleViewer(SwSimpleViewer.PLANE) ;
    viewer.setSceneGraph(root) ;

    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    Frame f = new Frame("AxisExample1") ;
    f.addWindowListener(l);
    f.add(viewer) ;
    f.pack() ;
    f.setVisible(true) ;
  }

  public static void main(String[] argv) {
    new AxisExample() ;
  }

}
```

Enhanced business graphic

GraphMaster has three categories of nodes:

- **Data storage nodes**: **PoIrregularMesh1D** or **PoRegularMesh1D**
- **Property nodes**: specify the appearance of the representations nodes.
- **Representation nodes**: inherited from **PoChart**, to draw curve, histogram, tubes, ribbons,...

A classical scene graph using these nodes looks like the following:



Data storage node

one dimensional meshes store data for charting representations

- **Contains geometry : a list of coordinates defined by setGeometry method**
  - **PoIrregularMesh1D:** any list of coordinates**.**
  - **PoRegularMesh1D :** constant gap between two consecutive coord**.**

- **Contains a list of scalar data sets. A scalar set can be use as Y-coordinates or for coloration**
  - Defined by **addValuesSet(int index, float[] scalars)**
  - The scalar set used by a representation node is defined by PoChart.yValuesIndex and PoChart.colorValuesIndex

- **Contains a list of scalar string sets. Use by representation with labels such as PoPieChartRep**
  - Defined by **addStringsSet(int index, String[] strings)**
  - Select a string set with the field **stringsIndex** in **PoLabel** and **PoPieChartRep**

Property nodes (1)

- PoBevelEdge

  defines the current values to bevel edges of subsequent DataViz representations inheriting from **PoChart**.

- PoMesh1DFilter

  Filter nodes used for selecting particular points from the 1D mesh geometry (**PoIrregularMesh1D** or **PoRegularMesh1D**). Only these points are used by subsequent representations inheriting from **PoChart** in a scene graph.

  Derived classes :

  PoCoordinateListFilter: fields coord (SoMFFloat), axis (SoSFEnum)
  PoIndexListFilter: field index (SoMFInt)
  PoPeriodFilter : field period (SoSFFloat), axis (SoSFEnum)
  PoPeriodIndexFilter : field period (SoSFInt)

Property nodes (2)

PoMesh1DHints

This nodes contains a single field SoSFEnum geomInterpretation that defines the way to connect to consective points of the 1D mesh.
Example: AS_IS (polygonal), SMOOTTH, HISTO_X, …

The representations inherited from **PoChart** use these hints for their computation.

PoProfile

A profile specifies a 2D polygon which is used by some charting representations to build their geometry. For instance, for the tube curve representation **PoTube**, the current profile is used to determine the profile of the tube.

Derived classes

- PoCircularProfile **Defines a circular profile**.
- PoEllipticProfile **defines an elliptic profile**.
- PoSquareProfile **Defines a square profile**.
- PoProfileCoordinate2 **Defines a 2D polygonal profile.**

Property nodes (3)

PoLabelHints

Defines the current hints for subsequent representations inheriting from **PoChart** that display labels.

Field SoSFString addString
Defines a string to concat to the label to display.

Field SoSFBool isLabelLineVisible
visibility of a line from the label and the part to be annotated

Field SoSFEnum justification
Defines the justification used to display label.

Field SoSFEnum labelPath
Defines the path used to display label.

Business graphic nodes PoChart

All classes inherited from PoChart (< … SoBaseKit) that draw pie-chart, curve, histogram bar … according to the data in the current mesh1D node.

The base class PoChart contains the fields

SoSFEnum colorBinding

specifies how the colors are bound to the representation:

· INHERITED: The entire representation is colored with the same inherited color.
· PER_VERTEX: Each vertex of the representation is colored with a different color
· PER_PART: Each part of the representation is colored with a different color

SoSFInt(32) colorValuesIndex

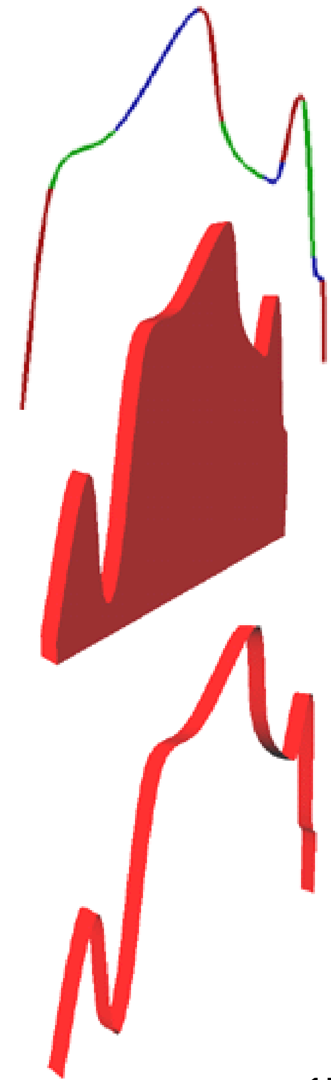
index of the set of scalar values used for coloring (if the field material is null)

SoSFNode material

Defines a list of materials used for the coloring (overrides the field colorValuesIndex)

SoSFInt yValuesIndex specifies the index of the set of values of the current 1D mesh used as the y-coordinates of each mesh node.

- SoSFInt **thicknessIndex**
  - index of the set of values used to specify the thickness.
- SoSFEnum **thicknessBinding**
  - PER_PART_THICKNESS, or PER_VERTEX_THICKNESS
- SoSFFloat **thicknessFactor**
  - multiplicative factor applied to the thickness values



SoSFEnum orientation
HORIZONTAL or VERTICAL
SoSFFloat threshold
SoSFFloat width

PoRibbon
SoSFFloat width

PoTube

- The shape of the profile is given by the current profile (PoProfile and its derived nodes).

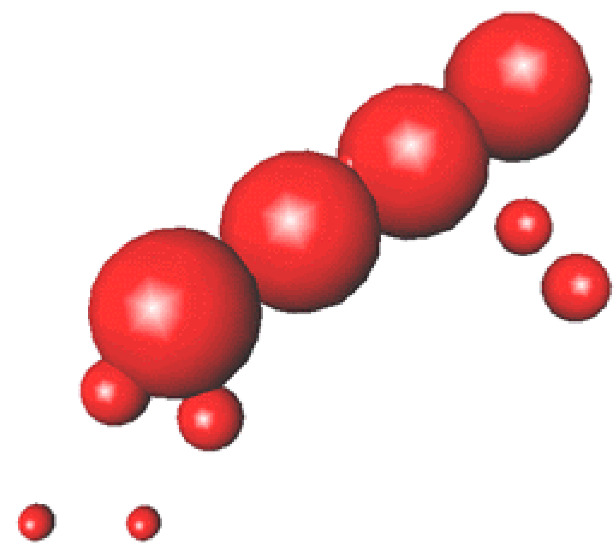
PoGenralizedScatter

- marker field representation where each marker defined by a sub-scene graph**.**

- **SoMFNode markers**

- **SoMFVec3f** scaleFactor
  SoSFInt sizeValuesIndex
  - Defines the index of the set of values used to specify the size of markers.

- **SoSFInt** zValuesIndex
  - specify a z-coordinate for markers



PoLabel

Builds a 2D label field on 1D mesh

- SoSFFloat fontSize
- SoSFEnum axis
- SoMFVec3f offset
- SoSFEnum position
  VALUE_POS,
  MIDDLE_POS,

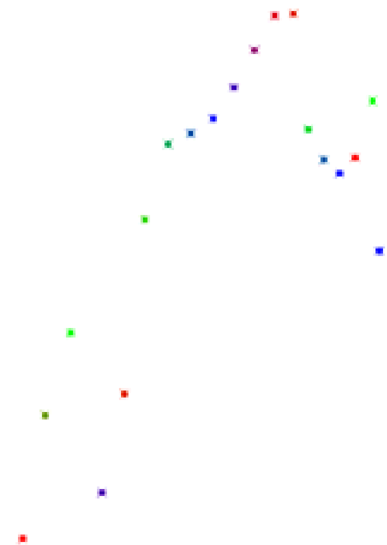
  THRESHOLD_POS
- SoSFInt stringsIndex
- SoSFFloat threshold
- SoSFBitMask valueType :
  - VALUE : The values displayed correspond to the mesh coordinate
  - NAME : The values displayed correspond to the names associated to the strings-set (see stringsIndex)

PoScatter

- Builds a 2D scatter on 1D mesh. A scatter representation is a bitmap marker field (indeed SoMarkerSet shape is used for this representation)

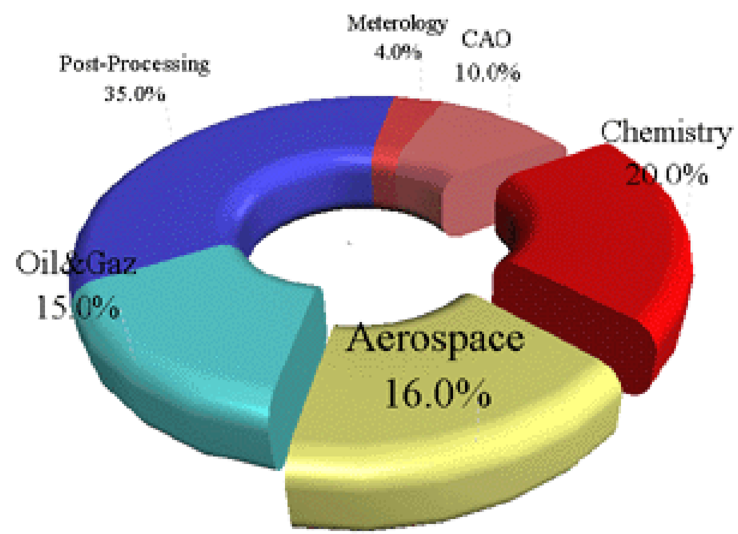
- SoMFInt markerIndex
  type of marker used (see SoMarkerSet). If the number of indices is inferior to the number of markers, they are cyclically used

- SoSFInt32 zValuesIndex
  - Defines the index of the set of values used to specify a z-coordinate for markers



PoPieChartRep

- SoSFFloat annoDistToCenter
  SoSFFloat annoFontSize
  SoSFFloat annoHeightFromSlice
  SoSFFloat height
  SoSFBool isAnnoSliceColor
  SoSFBool isNameVisible
  SoSFBool isPercentageVisible
  SoSFBool isValueVisible
  SoSFFloat radiusMax
  SoSFFloat radiusMin
  SoMFShort sliceToTranslateNumber
  SoMFFloat sliceToTranslateRadius
  SoSFInt stringsIndex



PoBar

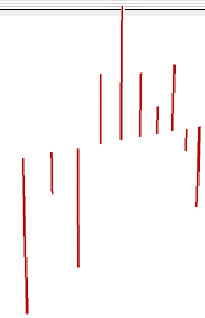
- Abstract base class for building bars on 1D mesh
  - **The abscissas of the bars are given by the geometry of the current mesh 1D, and the height are given by one of the value-set of the current mesh 1D specified by the field yValuesIndex.**
- SoSFEnum orientation
- Defines the orientation of the bars.

  HORIZONTAL or VERTICAL

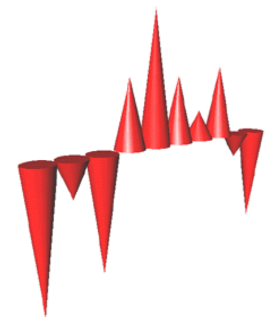
- SoSFFloat threshold

  Defines the origin of the bars

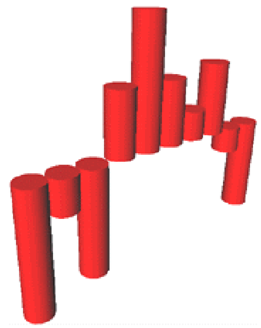
- No new field



- SoSFFloat bottomRadius
-



- SoSFFloat radius



- SoMFNode bars
  SoSFVec3f scaleFactor
-

- The shape of the profile is given by the current profile (PoProfile and its derived nodes).

Image output

- SoOffscreenRenderer is a powerful tool for generating bitmap images of Open Inventor scene. The bitmap image can be written out to a file or reused within application as asn image or texture map.

- SoOffscreenRenderer can write file with the following format
  - SGI's "rgb"
  - Encapsuled PostScript
  - TIFF
  - JPEG
  - BMP (windows)

```
SbViewportRegion myViewport = new SbViewportRegion();
  myViewport.setWindowSize(new SbVec2s((short)100,(short)100));
  // Render the scene
  SoOffscreenRenderer myRenderer = new
SoOffscreenRenderer(myViewport);
  myRenderer.setBackgroundColor(new SbColor(.6f, .7f, .9f));
  myRenderer.render(root);
  try {
     FILE myFile = new FILE("output", "w");
     myRenderer.writeToBMP(myFile);     //
myRenderer.writeToPostScript(myFile);
  } catch (Exception exc) {
  }
```

Vector output

- Plot Master provide new actions that allow you to render your scene graph or part of scene graph using vector PostScript, HPGL, CGM, (and GDI on windows)

- You cannot use these actions for rendering realistic 3D scene with smooth shading and textures. Use them to render resolution independent output when your scene is 2D or 3D with wire frame or flat shaded rendering.

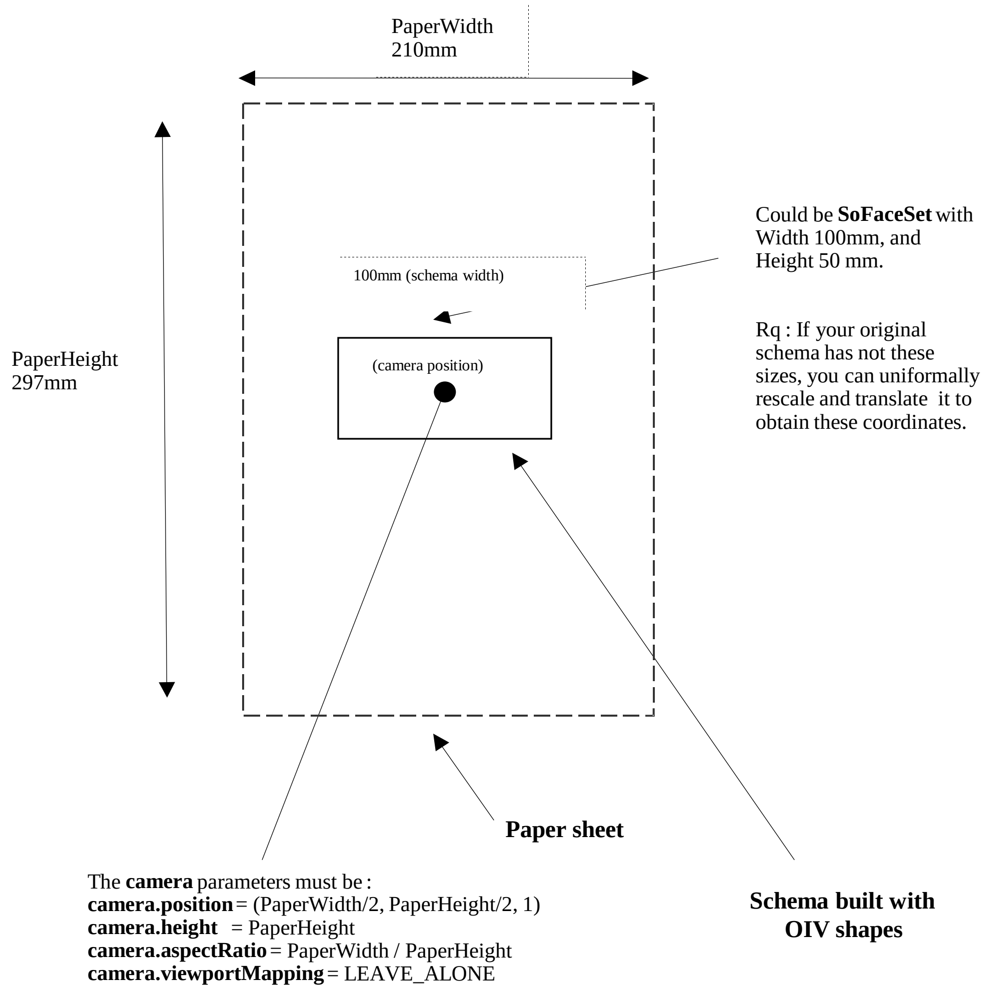
```
// Create and apply the PostScript action.
SoVectorizePSAction vecAct = new SoVectorizePSAction();
vecAct.setDrawingDimensions(80, 80);

vecAct.getOutput().openFile("VectorOutput.ps");
vecAct.apply(root);
vecAct.getOutput().closeFile();
```

Metric vector output

Printing in metric with CGM (or Postscript)

The following are some advises to realize a printing in metric with CGM (or Postscript, HPGL, GDI) :

- Work in printing coordinates
- Configure the camera to the size of the paper. For 2D schema use SoOrthographicCamera with viewport mapping set to LEAVE_ALONE (in order to prevent from deformations).



- Metric vector output (Java Example)

```
import com.tgs.inventor.nodes.*;
import com.tgs.inventor.*;

import com.tgs.dataviz.plot.*;

class PrintPSMetric {
  public static final boolean PORTRAIT = false;

  public static void main(String[] argv) {

    SoSeparator root = new SoSeparator();

    SoOrthographicCamera camera = new SoOrthographicCamera();
    root.addChild(camera) ;

    float PaperWidth;
    float PaperHeight;

    if (PORTRAIT) {
      PaperWidth = 210;
      PaperHeight = 297;
    } else {
      PaperWidth = 297;
      PaperHeight = 210;
    }
    camera.height.setValue(PaperHeight);
    camera.aspectRatio.setValue(PaperWidth/PaperHeight);
    camera.position.setValue(new SbVec3f(PaperWidth/2, PaperHeight/2, 1)) ; //
        Center of the sheet
    camera.viewportMapping.setValue(SoOrthographicCamera.LEAVE_ALONE);
```

```
    // Rectangle 100mm x 50mm center in the sheet
    SbVec3f[] rectCoords = new SbVec3f[4];
    for (int i=0; i<4; i++) rectCoords[i] = new SbVec3f();
    SbVec3f rectWidth = new SbVec3f(100,0,0);
    SbVec3f rectHeight = new SbVec3f(0,50,0);

    rectCoords[0].setValue(PaperWidth/2 -100/2, PaperHeight/2 -50/2, 0);
    rectCoords[1].setSum(rectCoords[0],rectHeight);
    rectCoords[2].setSum(rectCoords[1],rectWidth);
    rectCoords[3].setDiff(rectCoords[2],rectHeight);

    SoCoordinate3 coord3 = new SoCoordinate3();
    coord3.point.setValues(0, rectCoords) ;

    SoLightModel lModel = new SoLightModel();
    lModel.model.setValue(SoLightModel.BASE_COLOR);

    root.addChild(lModel) ;
    root.addChild(coord3) ;
    root.addChild(new SoFaceSet()) ;


    SoVectorizePSAction psAction = new SoVectorizePSAction();
    psAction.getOutput().openFile("myFile.ps") ;
    psAction.setDrawingDimensions(new SbVec2f(PaperWidth, PaperHeight)) ;
    if (PORTRAIT)
      psAction.setOrientation(SoVectorizePSAction.PORTRAIT) ;
    else
      psAction.setOrientation(SoVectorizePSAction.LANDSCAPE) ;
    psAction.apply(root) ;
    psAction.getOutput().closeFile() ;

  …
  }
}
```

3DdataMaster

3DdataMaster defines new extension nodes to OpenInventor for the developpement of scientific visualization.

3DdataMaster provides a powerful set of easy-to-use nodes to transform your data meshes into very understandable graphics visualization

2 sets of new nodes

C++

- Msuite/include/3Ddata
- Msuite/include/nodes

Java

- com.tgs.dataviz.mesh
- com.tgs. dataviz.nodes

Property / shape nodes

Msuite/include/nodes | com.tgs.dataviz.nodes

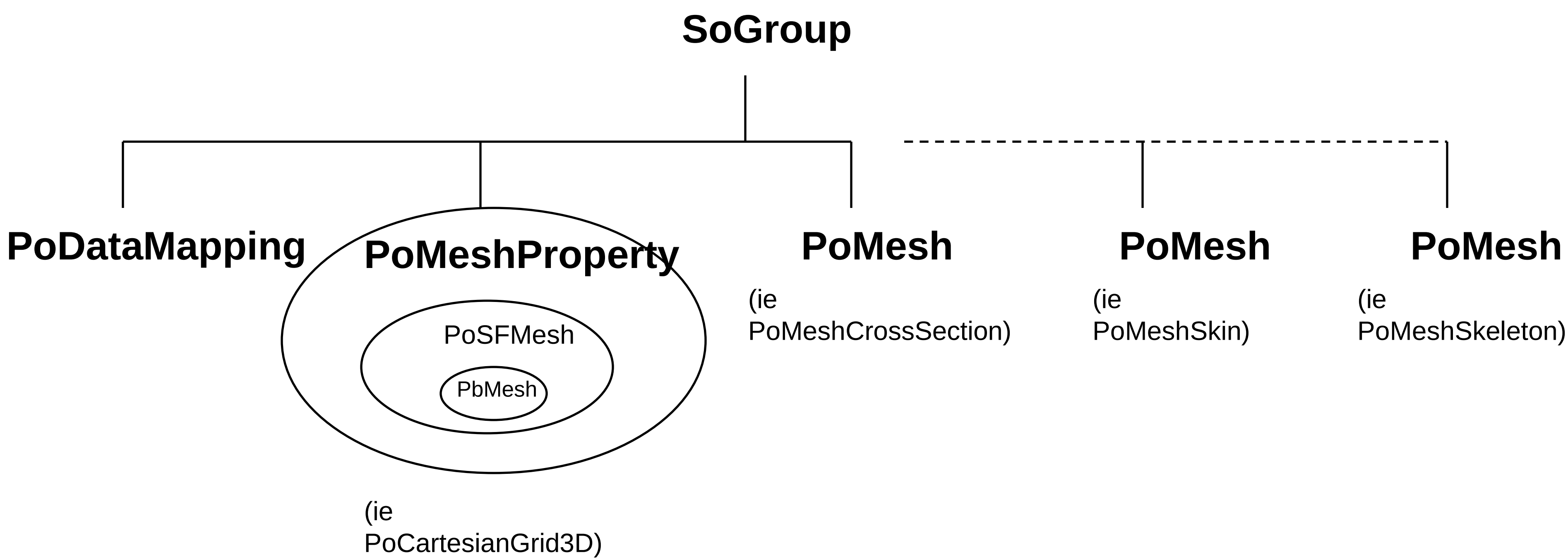
Contains new property nodes (PoMeshProperty) that defines mesh data in the scene graph

A property mesh node encapsulates a specific inventor field (PoSFMesh), itself containing a basic mesh object (PbMesh)

Msuite/include/3Ddata | com.tgs.dataviz.mesh

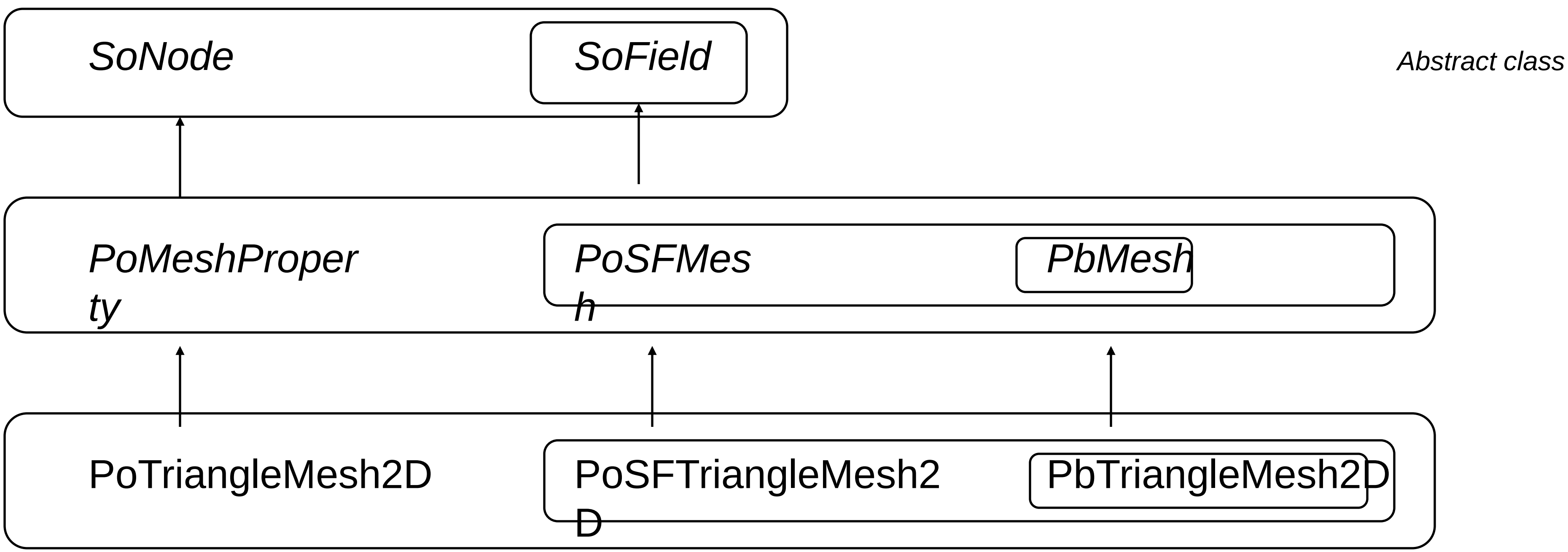
Contains new visualization-shape nodes (PoMesh) that builds a graphic representation of the inherited mesh node.

Mesh graph


SoGroup
PoDataMapping
PoMeshProperty
PoSFMesh
PbMesh
(ie
PoCartesianGrid3D)
PoMesh
(ie
PoMeshCrossSection)
PoMesh
(ie
PoMeshSkin)
PoMesh
(ie
PoMeshSkeleton)

Mesh class

All Open Inventor nodes describing a mesh are inherited from the class PoMeshProperty. They contain only one public field. Like any other field of a scene graph, it is an instance of a class inherited from SoField. These typical fields contain a single basic mesh object instance of PbMesh.



Mesh components

Any mesh object (inherited from PbMesh) defines …

- A topology
  - Regular
  - or irregular (unstructured)
  - It is defined by the choosen class and by the setGeometry method
- A geometry
  - a list of geometry nodes
  - or a list of coordinates
  - The geometry is defined by the setGeometry method. Speficic binding for each derived class of PbMesh
- A list of data sets (optional)
  - scalar data set : PbMesh.addValuesSet(int index, float val[], String setName)
  - vector data set : PbMesh.addVecsSet(int index, SbVec3f vec[], String setName)
  - string data set : PbMesh.addStringsSet(int index, String str[], String setName)
  - The size of a data set must be the number of nodes in the mesh : data are localized at node.

Some mesh methods

The PbMesh class provides a lot of useful inquire methods
As ex :

int	getNumNodes()
int	getNumCells()
float	getMaxValuesSet(int setIndex)
SbVec3f	getNodeCoord(int nodeIndex)
PbCell	getCell(int cellIndex)
SbBox3f	getBiggestCellBox()
SbBox3f	getBoundingBox()
PbArrayOfInt	getAdjacentCellsByNode(int cellIndex)
PbCell	findContainingCell(SbVec3f point, float tolerance)

Some mesh methods (2)

Some of these methods are also available in PoMeshProperty (or derived)

PbMesh	getMesh()	
void	addValuesSet(int index, float [] val, String name)	
void	addVecsSet(int index, String [] val , String name)	
void	setGeometry(…)	in derived class

they are only shortcut to the same one in PbMesh class.

PoMeshProperty.addValuesSet is equivalent to
PoMeshProperty.getMesh().addValuesSet and to
PoMeshProperty.mesh.addValuesSet

Mesh type (1)

Mesh are surface or volume
Mesh can be structured or not

PbMesh2D is the base class of 2D or 3D surfaces
- PbGrid2D is the abstract base class of structured surfaces
- PbIndexedMesh2D is the base class of unstructured surfaces

PbMesh3D is the base class of 3D volumes
- PbGrid3D is the abstract base class of structured volumes
- PbIndexedMesh3D is the base class of unstructured volumes

The visualization performances depends on the mesh topology.
- The quickest mesh to visualize are regular grid.

Mesh type (2)

"Pb" property basic types classes

- **PbMesh**
  - **PbMesh2D**
    - **PbGrid2D**
      - PbRegularCartesianGrid2D
      - PbCartesianGrid2D
        - PbParalCartesianGrid2D
      - PbPolarGrid2D
    - PbIndexedMesh2D
      - PbTriangleMesh2D
      - PbQuadrangleMesh2D
  - **PbMesh3D**
    - **PbGrid3D**
      - PbRegularCartesianGrid3D
      - PbCartesianGrid3D
        - PbParalCartesianGrid3D
    - PbIndexedMesh3D
      - PbTetrahedronMesh3D
      - PbHexahedronMesh3D

Property nodes classes

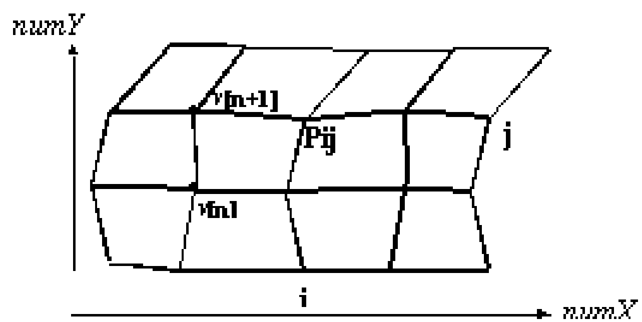
- **PoNode**
  - **PoMeshProperty**
    - PoCartesianGrid2D
    - PoParalCartesianGrid2D
  - PoRegularCartesianGrid2D
    - PoPolarGrid2D
    - PoIndexedMesh2D
    - PoTriangleMesh2D
    - PoQuadrangleMesh2D
    - PoCartesianGrid3D
    - PoParalCartesianGrid3D
  - PoRegularCartesianGrid3D
    - PoIndexedMesh3D
    - PoTetrahedronMesh3D
    - PoHexahedronMesh3D

Remark: Abstract classes appear in bold

CartesianGrid2D

PoCartesianGrid2D or PbCartesianGrid2D

This mesh represents a grid in Cartesian coordinates. It has a regular topology, but not necessarily a regular geometry. The topology of the mesh is defined by 2 integers, numX and numY. Hence the mesh is composed of (numX-1) * (numY-1) cells. The geometry is defined by an x and y coordinates array of numX * numY floats.

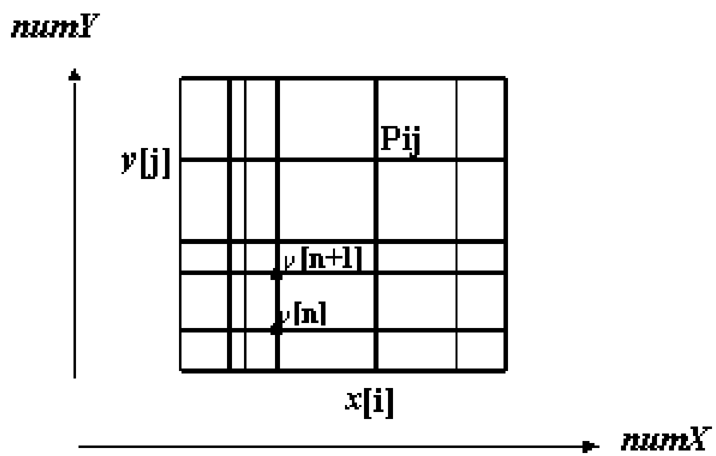


These arrays are in a y-line after y-line order: the node Pij, in the previous figure, has coordinates xP = x[i*numY+j], yP = y[i*numY+j]. A data set of this type of mesh is also defined by an array v of numX * numY floats, where v[i*numY+j] is the value of the node Pij.

Parallel CartesianGrid2D

PoParalCartesianGrid2D or PbParalCartesianGrid2D

This mesh represents a rectangular grid in Cartesian coordinates. Each cell of the mesh is a rectangle. The topology of the mesh is defined by 2 integers, *numX* and *numY*. Hence, the mesh is composed of (*numX*-1) * (*numY*-1) cells. The geometry is defined by an array *x* of *numX* abscissas of the vertical lines and by an array *y* of *numY* ordinates of the horizontal lines . These arrays must be given in a monotonically increasing or decreasing order

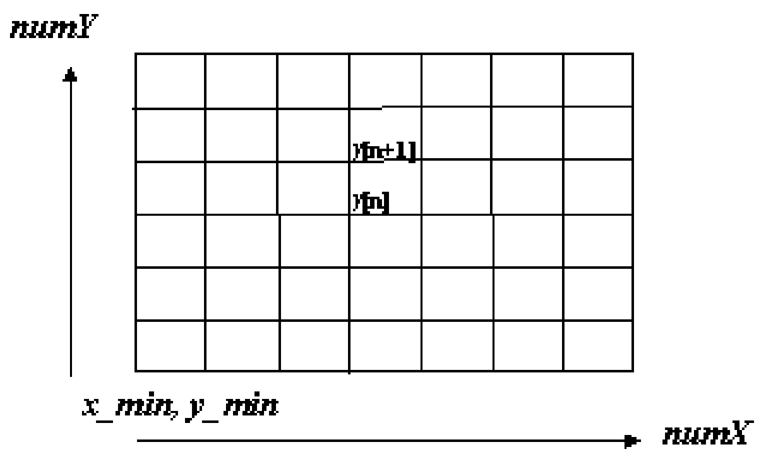


A data set of this kind of mesh is defined by an array *v* of *numX* * *numY* floats, in a y-line after y-line order. *v*[i**numY*+j] is the value of the node Pij which has coordinates *x*[i],*y*[j

Regular CartesianGrid2D

PoRegularCartesianGrid2D or PbRegularCartesianGrid2D

This mesh represents a rectangular and regular grid in Cartesian coordinates. Each cell of the mesh is a rectangle. Each cell has the same width and the same height. The topology of the mesh is defined by 2 integers, *numX* and *numY*. So, the mesh is composed of (*numX*-1) * (*numY*-1) cells. The geometry is defined by the bounding box of the mesh, i.e. by 4 float x_min,x_max,y_min,y_max.
Example: *numX*=8, *numY*=7

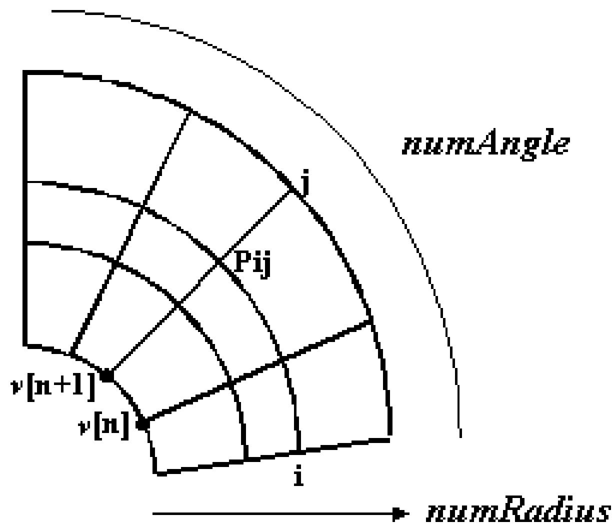


A data set of this type of ... *nY* floats, in a y-line after y-line order. *v*[i**numY*+j] is the value of the node Pij which has coordinates *x*[i],*y*[j].

Polar Grid2D

PoPolarGrid2D or PbPolarGrid2D

This mesh represents a grid in polar coordinates. Each cell of the mesh is an area between 2 arcs and 2 radius lines. The topology of the mesh is defined by *numRadius* and *numAngles*. It defines (*numRadius*-1) * (*numAngles*-1) cells. The geometry is defined by an array *radius* of *numRadius* floats and by an array *angles* of *numAngles* floats.

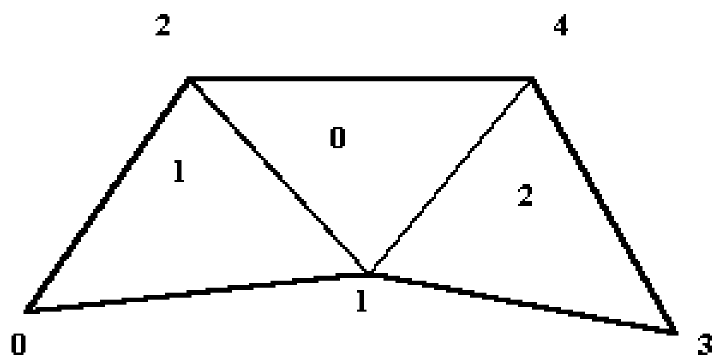


A data set of this kind of mesh is defined by an array *v* of *numRadius* * *numAngles* floats, in an arc-line after arc-line order. *v*[i**numAngles*+j] is the value of the node Pij which has polar coordinates *radius*[i], *angles*[j].

Triangle mesh

PoTriangleMesh2D or PbTriangleMesh2D

Each cell are triangle. The topology of the mesh is defined by the number of cells *numTriangles*, the number of nodes *numNodes* and the 3 node indices of each triangle *triangleIndex*. *triangleIndex* is an array of *numTriangles**3 integers, where *triangleIndex*[i*3 + j] is the j-th node of the i-th triangle (0<=j<3).



Example:

numTriangles = 3, numNodes = 5,
triangleIndex = { 1,2,4, 0,2,1, 1,4,3}

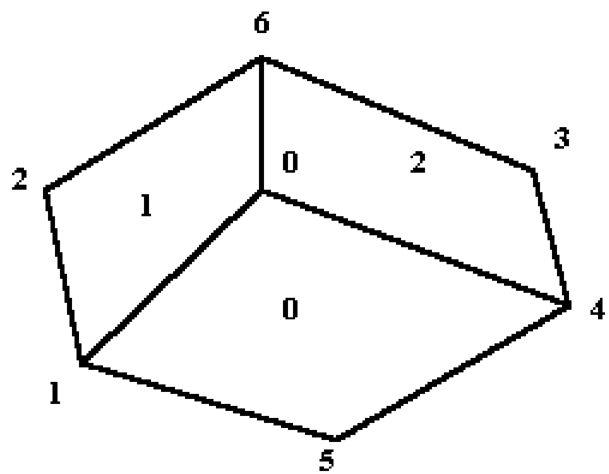
Unstructured surface mesh properties

- The mesh can be convex or not, and connected or not.
- Each cell is defined by a list of nodes indices in an array of node coordinates.
- Two adjacent cells must have 2 common node indices. If a cell's edge belongs to only one cell, this edge is considered to be part of an external or internal mesh limit. A cell can only have 1 adjacent cell along one edge or no adjacent cell at all
- The geometry of the mesh is defined by 2 or 3 arrays *xNode*, yNode, z*Node* of *numNodes* float coordinates. A data set is defined by an array *v* of *numNodes* floats. *v*[i] is the value of the node which has coordinates *xNode*[i],*yNode*[i], z*Node*[i]

Quadrangle mesh

PoQuadrangleMesh2D or PbQuadrangleMesh2D

Each cell are quadrangles.The topology of the mesh is defined by the number of cells *numQuadrangles*, the number of nodes *numNodes* and the 4 node indices of each cell *quadrangleIndex*. *quadrangleIndex* is an array of *numQuadrangles**4 integers, where *quadrangleIndex*[i*4 + j] is the j-th node of the i-th quadrangle (0<=j<4).



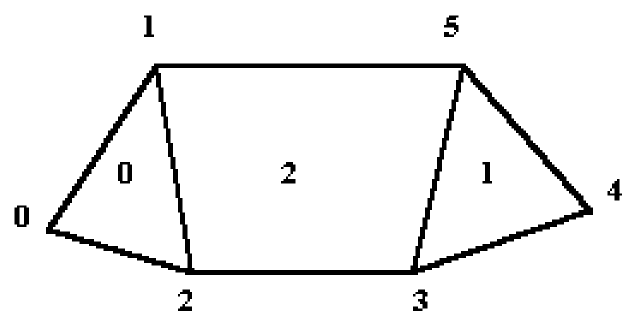
numQuadrangles = 3, numNodes = 7,

quadrangleIndex = { 0,4,5,1, 0,1,2,6, 0,6,3,4}

Indexed mesh 2D

PoIndexedMesh2D or PbIndexedMesh2D

This mesh contains triangles or quadrangles, and can also contains polygonal cells with some restrictions. The topology of the mesh is defined by the number of cells *numCells*, the number of nodes *numNodes*, the node indices list of each cell *cellIndex*, and the number of nodes of each cell *cellType*. *cellType* is an array of *numCells* integers, for example *cellType*[i] = 3 means that the i-th cell is a triangle. *cellIndex* is an array of N integers where N = *cellType*[0]+ *cellType*[1]+...+ *cellType*[*numcells*-1].



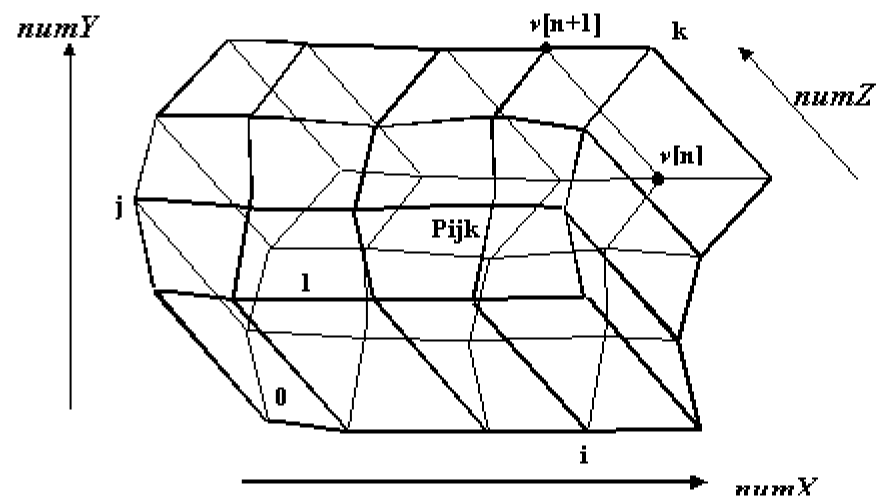
numCells = 3, numNodes = 6,
cellType = { 3,3,4},
cellIndex = { 0,1,2, 3,5,4, 2,1,5,3}

Cartesian grid 3D

PoCartesianGrid3D or PbCartesianGrid3D

Volume grid in Cartesian coordinates. It has a regular topology, but not necessarily a regular geometry. The cells are hexahedrons with opposite facets that are not necessarily parallel. The topology of the mesh is defined by 3 integers *numX*, *numY* and *numZ*. The mesh is thus composed of (*numX*-1) * (*numY*-1) * (*numZ*-1) cells.

Example: *numX* = 5, *numY* = 4, *numZ* = 2



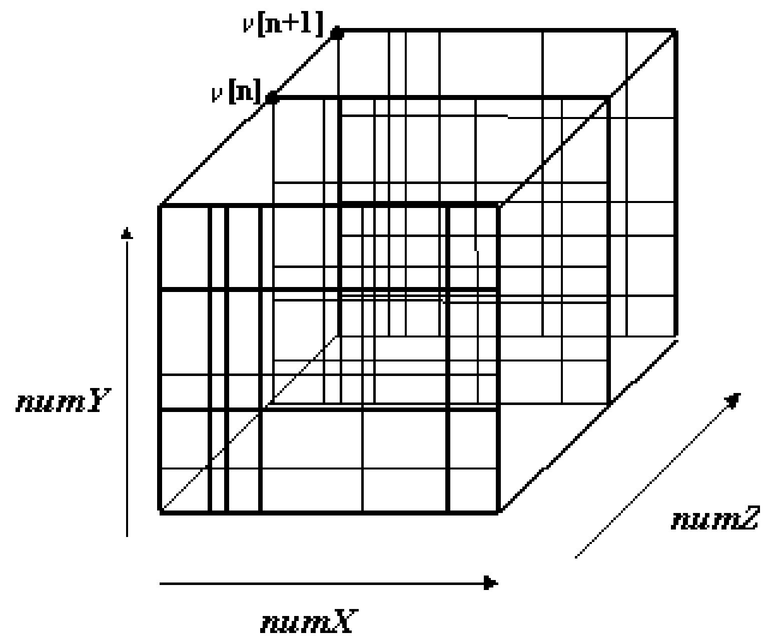
- The geometry is defined by *x*, *y* and *z* coordinates arrays of *numX* * *numY* * *numZ* floats. These arrays are in a z-lines after z-lines order: the node Pijk has X coordinates xP = *x*[i**numY***numZ*+j**numZ*+k]. *v*[i**numY***numZ* + j**numZ* + k] is the value of the node Pijk.

Parallel Cartesian grid 3D

[PoParalCartesianGrid3D]() or [PbParalCartesianGrid3D]()

Parallelepiped grid in Cartesian coordinates. Each cell of the mesh is a parallelepiped. The topology of the mesh is defined by 3 integers *numX*, *numY* and *numZ*. The mesh is thus composed of (*numX*-1) * (*numY*-1) * (*numZ*-1) cells. The geometry is defined by an array *x* of *numX* floats, an array *y* of *numY* floats and by an array *z* of *numZ* floats. *x* is the list of the *numX* abscissas of the lines perpendicular to the X-plane. Idem for *y* and *z* arrays. These arrays must be given in monotonically increasing or decreasing order.

Example: *numX* = 7, *numY* = 6, *numZ* = 3

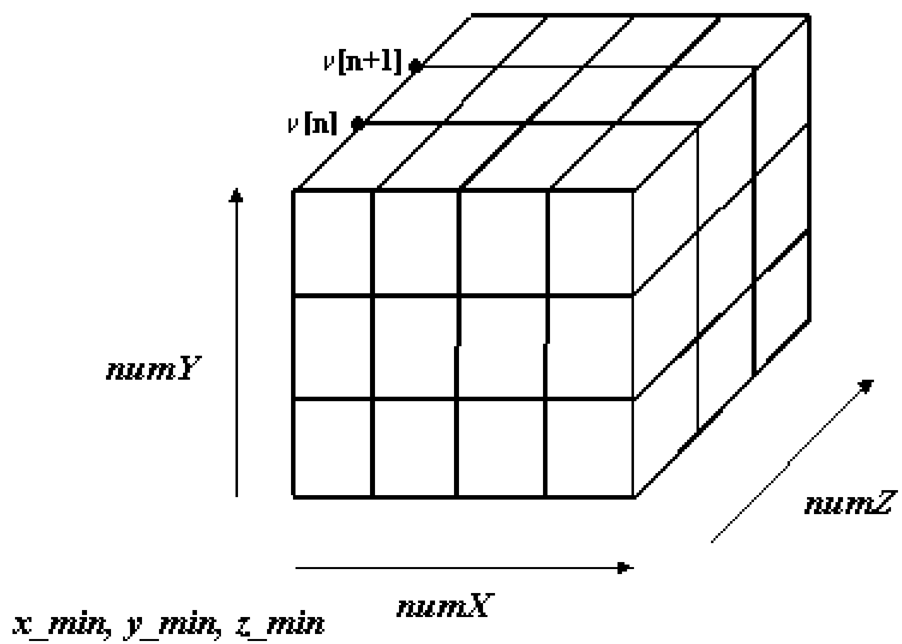


Regular Cartesian grid 3D

PoRegularCartesianGrid3D or PbRegularCartesianGrid3D

Parallelepiped and regular grid in Cartesian coordinates. Each cell of the mesh is a parallelepiped, and each one has the same size. The topology of the mesh is defined by 3 integers *numX*, *numY* and *numZ*. The mesh is thus composed of (*numX*-1) * (*numY*-1) * (*numZ*-1) cells. The geometry is defined by the bounding box of the mesh, i.e. by 6 floats x_min,x_max, y_min,y_max, z_min,z_max.

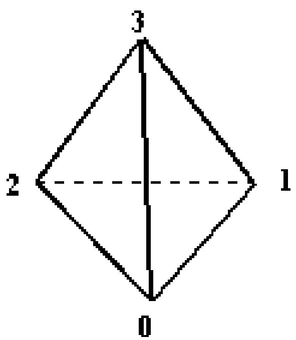
Example: *numX* = 5, *numY* = 4, *numZ* = 4



Tetrahedron mesh

PoTetrahedronMesh3D or PbTetrahedronMesh3D

Each cell are tetrahedrons defined by 4 nodes indices. Two adjacent tetrahedrons must have 3 common node indices. Each tetrahedron should be numbered as follows: The first 3 indices define a facet, and orient this facet towards the interior side of the tetrahedron.

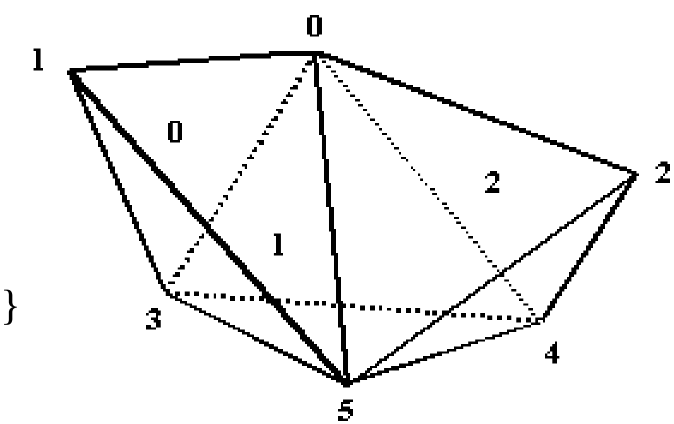


The topology of the mesh is defined by *numTetrahedrons*, *numNodes* and the 4 node indices of each tetrahedron *tetrahedronIndex*. *tetrahedronIndex* is an array of *numTetrahedrons**4 integers, where *tetrahedronIndex*[i*4 + j] is the j-th node of the i-th tetrahedron (0<=j<4).

Example

numTetrahedrons = 3, numNodes = 6,

tetrahedronIndex = { 0,5,1,3, 3,5,4,0, 0,4,2,5}



Unstructured volume mesh properties

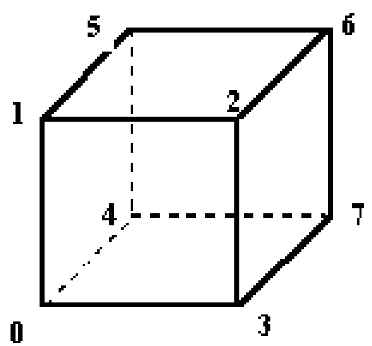
- The mesh can be convex or not, and connected or not.
- Each cell is defined by a list of nodes indices in an array of node coordinates.
- Two adjacent cells must have a common facet, ie at least 3 common node indices. If a cell's facet belongs to only one cell, this facet is considered to be part of an external or internal mesh limit. A cell can only have 1 adjacent cell along one facet or no adjacent cell at all.The facet which do not belong to the mesh limit must be referenced exactly twice in the mesh.
- The geometry of the mesh is defined by 3 arrays *xNode*, yNode, z*Node* of *numNodes* float coordinates. A data set is defined by an array *v* of *numNodes* floats. *v*[i] is the value of the node which has coordinates *xNode*[i],*yNode*[i], z*Node*[i]

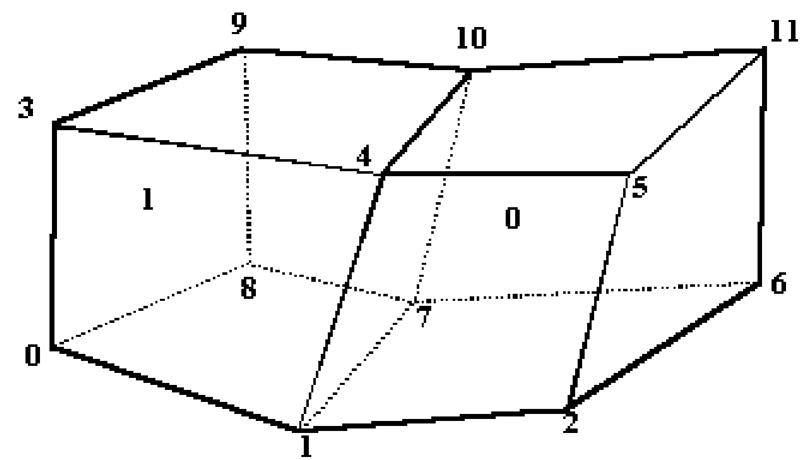
Hexahedron mesh

PoHexahedronMesh3D or PbHexahedronMesh3D

Each cell are hexahedrons defined by 8 nodes indices. Two adjacent hexahedrons must have 4 common node indices. Each hexahedron should be numbered as follows: the first 4 indices define a facet, and orient this facet towards the interior side of the hexahedron.

The topology of the mesh is defined by *numHexahedrons*, *numNodes* and the 8 node indices of each hexahedron *hexahedronIndex*. It is an array of *numHexahedrons**8 integers, where *hexahedronIndex*[i*8 + j] is the j-th node of the i-th hexahedron (0<=j<8)





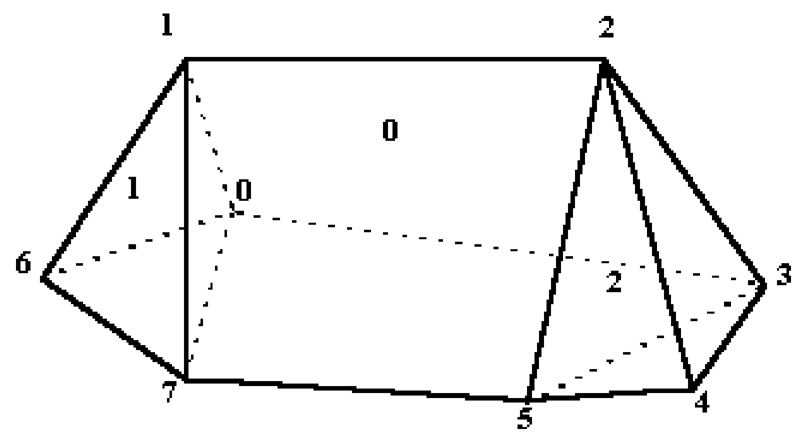
Example
numHexahedrons = 2, numNodes = 12,
hexahedronIndex = { 10,7,6,11,4,1,2,5, 4,1,0,3,10,7,8,9 }

Indexed mesh 3D

PoIndexedMesh3D or PbIndexedMesh3D

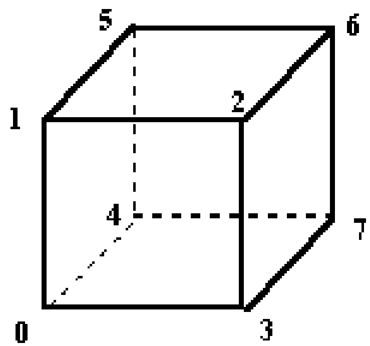
Each cell is a polyhedron which can either be a tetrahedron (4 nodes), a pyramid (5 nodes), a pentahedron (6 nodes), or a hexahedron (8 nodes). The topology of the mesh is defined by the number of cells *numCells*, the number of nodes *numNodes*, the node indices list of each cell *cellIndex*, and the number of nodes of each cell *cellType*. *cellType* is an array of *numCells* integers, for example *cellType*[i] = 4 means that the i-th cell is a tetrahedron . *cellIndex* is an array of N integers where N = *cellType*[0]+ *cellType*[1]+...+ *cellType*[*numcells*-1].

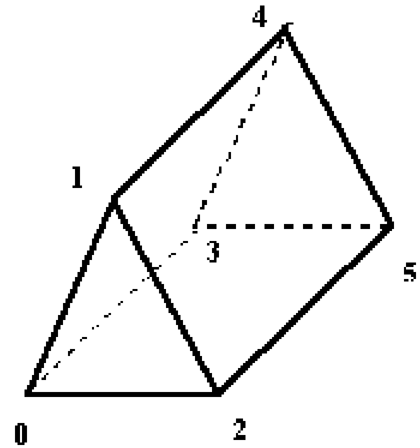
numCells = 3, numNodes = 8,
cellType = { 6,4,4},
cellIndex = { 1,7,0,2,5,3, 0,6,7,1, 2,5,3,4}

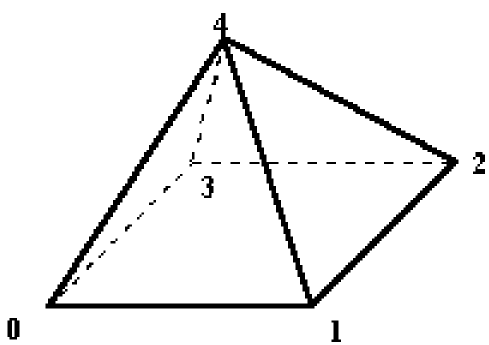


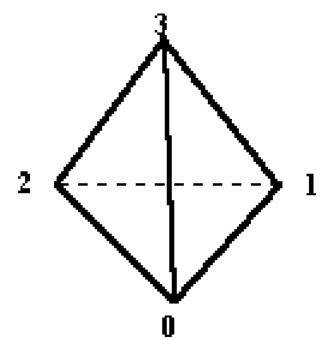
Indexed mesh 3D (2)

The cells must be numbered as follows: for each cell, the first 3 or 4 indices (depending on the cell type) define a cell's facet, and orient this facet towards the interior side of the element.









Mesh Visualization

To visualize data from a mesh, you must instantiate a class derived from **PoMesh**, depending on the type of visualization you need. For example **PoMeshSkin** allows you to visualize the skin of a volume mesh. These classes are derived from Open Inventor node kits and we call them "visualization node kits".

These visualization nodes draw a representation of the **PbMesh** object included in the current PoMeshProperty node inherited in the scene graph.

PoMesh2D nodes build a representation of surface mesh (**PbMesh2D**)
PoMesh3D nodes build a representation of volume mesh (**PbMesh3D**)

Simple example

```java
import com.tgs.inventor.* ;
import com.tgs.inventor.awt.* ;
import com.tgs.inventor.nodes.* ;
import com.tgs.dataviz.nodes.* ;
import com.tgs.dataviz.mesh.* ;

import java.awt.* ;
import java.awt.event.* ;

public class SimpleMeshViewer {

  public static SoGroup buildScene() {
    PoRegularCartesianGrid3D mesh = new
       PoRegularCartesianGrid3D();
    mesh.setGeometry(10,10,10, 5,5,5, 30,40,50);
    PoMeshSkin v_MeshSkin = new PoMeshSkin();

    SoGroup root = new SoGroup();
    root.addChild(mesh);
    root.addChild(v_MeshSkin);
    return root;
  }

public static void main(String[] argv) {

    SwSimpleViewer viewer = new
       SwSimpleViewer(SwScene.EXAMINER) ;
    viewer.setSceneGraph(buildScene()) ;
    viewer.viewAll();

    Panel panel = new Panel(new BorderLayout()) ;
    panel.add(viewer);

    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
       System.exit(0);
      }
    };

    Frame f = new Frame();
    f.addWindowListener(l);
    f.add(panel) ;
    f.pack() ;
    f.setVisible(true) ;
  }

}
```

Data mapping on mesh visualization

Since a mesh object can contain several scalar data sets, the visualization node kit can select one by using **PoMesh.valuesIndex** to color the shape in the node kit. **Warning** : default is –1 !

The field **PoMesh.coloringType** selects the type of coloring which will be applied.
4 types are available

COLOR_INHERITED (default !)

the representation of the mesh uses only 1 color inherited from the scene graph

COLOR_AVERAGE

an average of the nodes' values of an edge or facet is converted to a color applied to draw the edges or facets (for some surface mesh visualization only)

COLOR_MAPPING

each node value defines a color and the edges or facets are drawn by interpolating these colors.
(Gouraud shading)

COLOR_CONTOURING

only for visualization that draw facet. each facet is exactly sub-divided into isovalued areas. These areas are filled with the color associated to this isovalue. May slow down visualization.

Data mapping class

For coloring type COLOR_AVERAGE, COLOR_MAPPING and COLOR_CONTOURING, a **PoDataMapping** object must be inserted in the scene graph. Such object maps a floating value to a color, or maps a set of floating values to a color ramp or several color ramps. 2 classes inherited from **PoDataMapping** can be instanciated

PoLinearDataMapping

Two values, value1 and value2, are associated with color1 and color2. The color associated with a value between value1 and value2 is a linear interpolation between color1 and color2.

PoNonLinearDataMapping2

This class defines a set of colors or a set of color ramps associated with floating values. You can choose:

· LINEAR_PER_LEVEL type of mapping for a floating value **f**. If **f** is in the interval $\mathbf{f_i}$, $\mathbf{f_{i+1}}$, its associated color will be the linear interpolation between $\mathbf{c_i}$ and $\mathbf{c_{i+1}}$ colors. In this case, you must provide the same number of floating values as the number of colors.

· NON_LINEAR_PER_LEVEL type of mapping for a floating value **f**. If **f** is in the interval $\mathbf{f_i}$, $\mathbf{f_{i+1}}$, its associated color will be the $\mathbf{c_{i+1}}$th color; no interpolation is performed. If **f** is smaller than $\mathbf{f_1}$, then $\mathbf{c_1}$ is used. In this case, you must provide n+1 colors for n floating values.

Color mapping example

```
public class MeshMapping {

  public static SoGroup buildScene() {
    PoRegularCartesianGrid3D po_mesh = new PoRegularCartesianGrid3D();
    po_mesh.setGeometry(30,10,10, 5,5,5, 30,40,50);
    PbMesh pb_mesh = po_mesh.getMesh();

    float[] val = new float[pb_mesh.getNumNodes()];
    for (int i=0; i<pb_mesh.getNumNodes(); i++) val[i] = (float)Math.random();
    po_mesh.addValuesSet(0,val);                                    <----------

    PoLinearDataMapping dataMap = new PoLinearDataMapping();        <----------
    dataMap.color1.setValue(new SbColor(1,0,0));
    dataMap.color2.setValue(new SbColor(0,0,1));
    dataMap.value1.setValue(pb_mesh.getMinValuesSet(0));
    dataMap.value2.setValue(pb_mesh.getMaxValuesSet(0));

    PoMeshSkin meshSkin = new PoMeshSkin();
    meshSkin.valuesIndex.setValue(0);                               <----------
    meshSkin.coloringType.setValue(PoMesh.COLOR_MAPPING);           <----------

    SoGroup root = new SoGroup();
    root.addChild(po_mesh);
    root.addChild(dataMap);                                         <----------
    root.addChild(meshSkin);
    return root;
  }
  …
}
```

Color contouring

In order to use color contouring, you must specify the contour level values. They are defined by the class PoIsovaluesList, a property node which must be inserted in the scene graph.

A list of isovalues can be a list of any floats. However, convenience methods are available to define a regular list. In a regular list, the step size between 2 consecutive isovalues is a constant. For example, the following methods are available for creating lists:

- void **setRegularIsoList**(int numLevels, float min, float max)
- void **setRegularIsoList**(int numLevels, float firstValue, float step)
- void **setRegularIsoList**(int numVal, float[] values, int numLevels)

The following code creates a regular list of 25 isovalues, bounded by 0. and 10.:

```
PoIsovaluesList myIsoList = new PoIsovaluesList();
myIsoList.setRegularIsoList(25, 0f,10f);
```

PoIsovaluesList is also used for visualization of contouring lines on a surface mesh.

Color contouring example

```
public class MeshContouring {

  public static SoGroup buildScene() {
    PoRegularCartesianGrid3D po_mesh = new
       PoRegularCartesianGrid3D();
    po_mesh.setGeometry(10,10,10, 5,5,5, 30,40,50);
    PbMesh pb_mesh = po_mesh.getMesh();

    float[] val = new float[pb_mesh.getNumNodes()];
    for (int i=0; i<pb_mesh.getNumNodes(); i++)
       val[i] = (float)Math.random();
    po_mesh.addValuesSet(0,val);

    PoIsovaluesList isoList = new PoIsovaluesList();
    isoList.setRegularIsoList(pb_mesh.getMinValuesSet(0),
                              pb_mesh.getMaxValuesSet(0),10);

    PoLinearDataMapping dataMap = new
       PoLinearDataMapping();
    dataMap.color1.setValue(new SbColor(1,0,0));
    dataMap.color2.setValue(new SbColor(0,0,1));
    dataMap.value1.setValue(pb_mesh.getMinValuesSet(0));
    dataMap.value2.setValue(pb_mesh.getMaxValuesSet(0));

    PoMeshSkin meshSkin = new PoMeshSkin();
    meshSkin.valuesIndex.setValue(0);
    meshSkin.coloringType.setValue(PoMesh.COLOR_CONTOURING);

    SoGroup root = new SoGroup();
    root.addChild(po_mesh);
    root.addChild(dataMap);
    root.addChild(isoList);
    root.addChild(meshSkin);
    return root;
  }

  …

}
```

Mesh visualization nodes (1)

•**PoMesh**

// Surface mesh visualization

•**PoMesh2D**
- •PoMeshLimit
- •PoMeshLines
- •PoMeshFilled
- •PoMeshSides
- •PoMeshContouring
- •PoMesh2DVec

// Volume mesh visualization

•**PoMesh3D**
- •PoMeshSkin
- •PoMeshCrossSection
- •PoMeshCrossContour
- •PoMeshSkeleton
- •PoMeshLevelSurf
- •PoMesh3DVec
- •PoMesh3DVecGridCrossSection

Abstract classes appear in bold

A scalar data set can also be used as z coordinates for any surface mesh visualization. This scalar set can be selected by the field **PoMesh2D.zValuesIndex**.

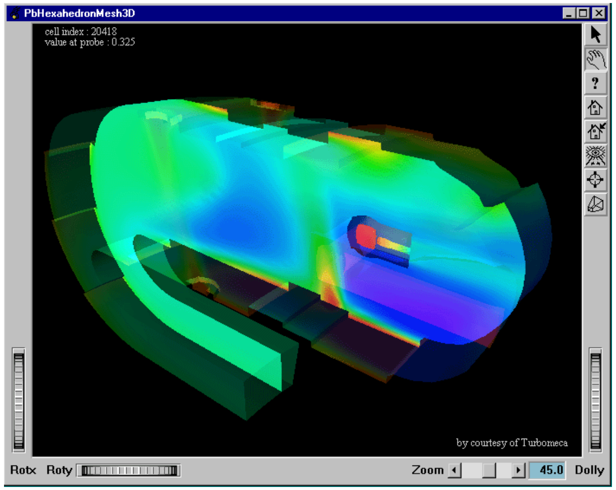
Mesh visualization nodes (2)

2D/3D mesh visualization classes

- **PoBase**
  - **Po3DdataMaster**
    - **PoMesh**
      - PoMeshProbePoint
      - **PoBaseStreamLine**
        - PoStreamLine
        - PoStreamSurface
        - **PoStreamParticleMotion**
          - PoStreamLineMotion
          - PoStreamPointMotion
          - PoStreamSphereMotion
          - PoStreamTadpoleMotion
      - **PoCellShape**
        - PoCellEdges
        - PoCellFacets
        - PoCellIndices

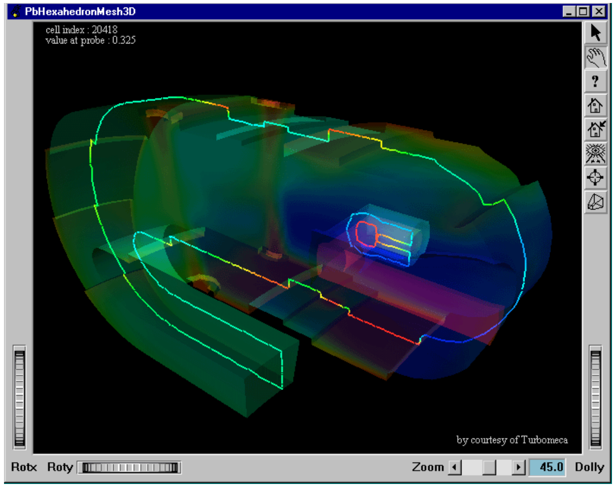
Remark: Abstract classes appear in bold.

3D mesh visualization (1)



PoMeshCrossSection

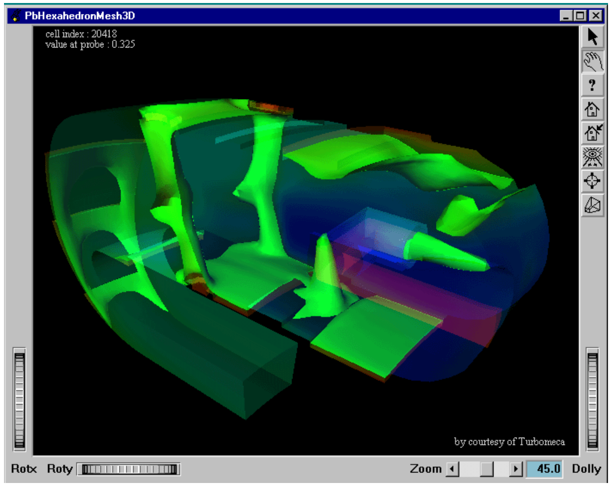
SoSFPlane plane



PoMeshCrossContour

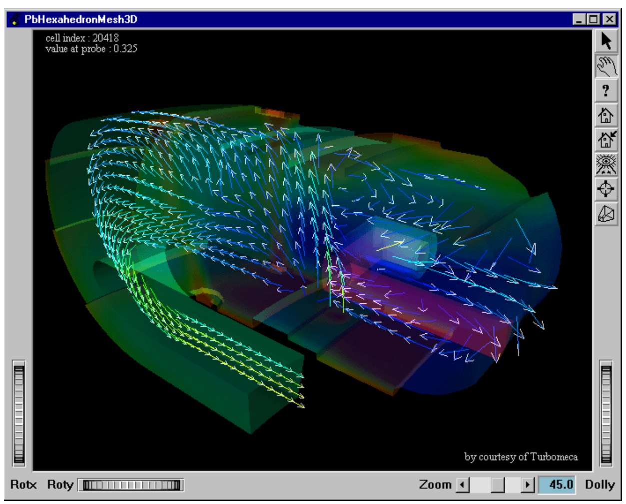
SoSFPlane plane

3D mesh visualization (2)



PoMeshLevelSurf

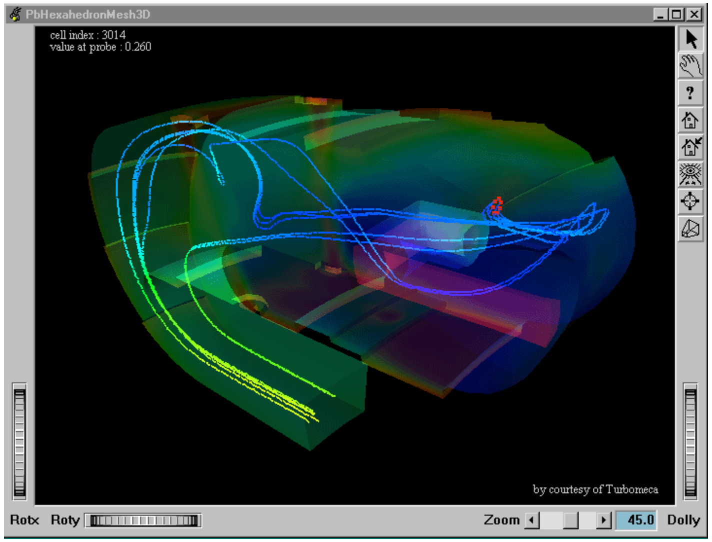
SoSFFloat levelValue	0.0
SoSFEnum surfOrientation	ORIENTED_TO_MAX
SoSFInt valuesIndexForLevel	-1

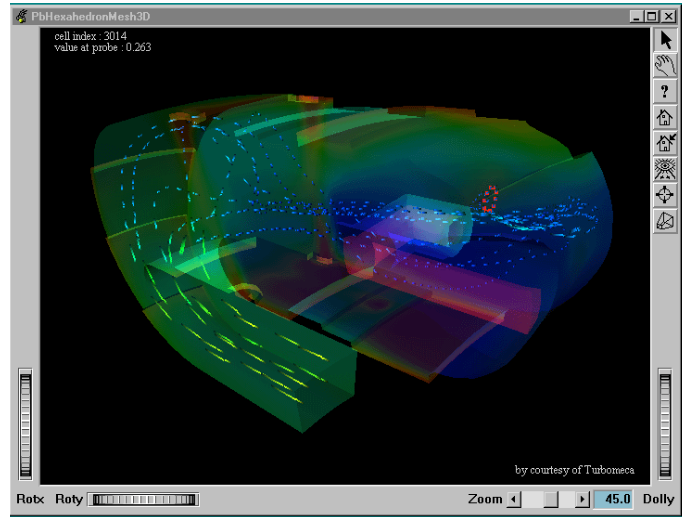


PoMesh3DVecGridCrossSection

SoSFPlane plane	Z=0 plane
SoSFFloat gridSpacing	0.05
SoSFEnum projectionType	NO_PROJECTION

3D mesh visualization (3)





PoStreamLines

SoSFFloat lineWidth

PoStreamTadpoleMotion

SoSFInt	pulseFrequency	5
SoSFFloat	shiftStart	0.0
SoSFFloat	timeStep	1.0
SoSFBool	isStartRandomized	TRUE
SoSFBool	isBlinking	TRUE
SoSFInt	viewFrame	0
SoSFFloat	blinkSpeed	3.0
SoSFColor	backColor	SbColor(0.0,0.0,0.)
SoSFColor	particleColor	SbColor(0.0,0.9,0.9) …..

Differences between Open Inventor for C++ and Open Inventor for Java (1)

- Features not available in Open Inventor for Java:
  - Calling OpenGL directly from Java
  -

- Features available in Open Inventor for Java but with a different interface
  - Pointers, values : All objects are handled through references. No more pointers or value but only references.
  - Operator redefinition : Several C++ classes redefine standard operators. This is not possible in Java(TM). Alternate methods are provided to allow this kind of operation. For example, the SbVec3f operator* method is replaced by SbVec3f.multiply().

Differences between Open Inventor for C++ and Open Inventor for Java (2)

Some methods with parameters given as references on basic types :

```
in C++:
void getAntialiasing(SbBool &smoothing, int &numPasses) const
```

```
in Java(TM):
public void getAntialiasing(boolean [] smoothing, int [] numPasses)
```

To call such method with Java, the user must define an array of 1 element for each parameter :

```
boolean [] smoothing = new boolean [1];
int [] numpasses = new int [1];
area.getAntialiasing( smoothing, numpasses);
```

Differences between Open Inventor for C++ and Open Inventor for Java (3)

- Callback mechanism : the Callback mechanism is very often used by Open inventor C++. Callback functions are defined as follows :

```
typedef <return_type> functionCB(void * userData, type1 arg1,..., typen argn);
static <return_type> myFunctionCB(void * userData, type1 arg1,..., typen argn) {
...
}
The callback is registered by calling a addCallback method:
addCallback((functionCB*) myFunctionCB, this);
```

With Open Inventor for Java, each type of callback is implemented by a specific class. To create a new callback, the user must extend this class and overwrite the default invoke method of the class.

```
class ProcessKeyEvents extends SoEventCallbackCB {
  private SwRenderArea area;
  public ProcessKeyEvents (area) {
    this.area = area;
  }
   public void invoke(SoEventCallback cb) {
     if (SoKeyboardEvent.isKeyPressEvent(cb.getEvent(),
SoKeyboardEvent.P)) {
              ...
        cb.setHandled();
     }
   }
}
```

```
eventCB.addEventCallback(SoKeyboardEvent.class,
                         new ProcessKeyEvents(area), null);
```

Differences between Open Inventor for C++ and Open Inventor for Java example

```java
import java.awt.* ;
import java.awt.event.*;
import com.tgs.inventor.*;
import com.tgs.inventor.awt.* ;
import com.tgs.inventor.nodes.* ;

public class HelloCone  {
  public HelloCone() {
    // Make a scene containing a red cone
    SoSeparator root = new SoSeparator();
    root.addChild(new SoDirectionalLight());
    SoMaterial myMaterial = new SoMaterial();
    myMaterial.diffuseColor.setValue(1,0,0); // Red
    root.addChild(myMaterial);
    root.addChild(new SoCone());

    // Put the scene in myRenderArea
    SwSimpleViewer myRenderArea = new SwSimpleViewer();
    myRenderArea.setSceneGraph(root);

    Panel panel = new Panel(new BorderLayout()) ;
    panel.add(myRenderArea);

    Frame f = new Frame ("HelloCone");
    f.addWindowListener(new WindowListener());
    f.add(panel);
    f.pack();
    f.show();
  }

  class WindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }

  public static void main(String argv[]) {
    new HelloCone();
  }
}
```

```cpp
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

void main(int , char **argv)
{
  // Initialize Inventor. This returns a main window to use.
  // If unsuccessful, exit.
  Widget myWindow = SoXt::init(argv[0]); // pass the app name
  if (myWindow == NULL) exit(1);

  // Make a scene containing a red cone
  SoSeparator *root = new SoSeparator;
  SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
  SoMaterial *myMaterial = new SoMaterial;
  root->ref();
  root->addChild(myCamera);
  root->addChild(new SoDirectionalLight);
  myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Red
  root->addChild(myMaterial);
  root->addChild(new SoCone);

  // Create a renderArea in which to see our scene graph.
  // The render area will appear within the main window.
  SoXtRenderArea *myRenderArea = new
SoXtRenderArea(myWindow);

  // Make myCamera see everything.
  myCamera->viewAll(root, myRenderArea->getViewportRegion());

  // Put our scene in myRenderArea, change the title
  myRenderArea->setSceneGraph(root);
  myRenderArea->setTitle("Hello Cone");
  myRenderArea->show();

  SoXt::show(myWindow);  // Display main window
  SoXt::mainLoop();      // Main Inventor event loop
}
```

VolumeViz Overview

- ➔ Cross platform library to do voxel rendering based on Open Inventor.
- ➔ Rendering big volume of data (or part of it) with data mapping.
- ➔ Visualization of the internal volume with slices (Ortho Slice, Oblique Slice) or transparency (RGBA).
- ➔ Picking information.
- ➔ Hardware optimization used (2D textures, 3D textures, VolumePro board…)

VolumeViz Data Structure

- Node **SoVolumeData**

- 3D data matrix
  - Char (1 byte)
  - Short (2 bytes)
- Regular data set.
- Source
  - Memory
  - File (SoVolumeReader)

VolumeViz Coloration

- Node **SoTransferFunction**

- Association value – color.
- Use of paletted color (default) or real color.
  - Paletted color scale keeps memory and can be updated faster,
  - Real coloring allows lighting on the volume.
- Predefined color scale.

VolumeViz Rendering (1/2)

- Node **SoVolumeRender**

- Draw the data volume.
  - Slices displayed from back to front (textured polygons)
  - Composition
    - Max
    - Sum
    - Alpha Blending
  - Lighting
  - Texture interpolation
  - SoMaterial used (transparency, diffuseColor).

VolumeViz Rendering (2/2)

- A sub-volume can be rendered (**SoROI**).
- Nodes to make slices in the volume :
  - **SoOrthoSlice** : main plane,
  - **SoObliqueSlice** : any plane.

VolumeViz Picking information

- **SoObliqueSliceDetail** : Stores detail information about a picked voxel on an oblique slice.

- **SoOrthoSliceDetail** : Stores detail information about a picked voxel on an ortho slice.

- **SoVolumeRenderDetail** : Stores detail information about a picked voxel or pick ray in a data volume.