Open Inventor Programming Guide
Basic Annotation

Contents:

Section 1 – Annotation

Section 1.1 – Overview

Annotation is loosely defined as extra information that helps to understand the main part of the data. In a visualization application this may include both 2D and 3D information. Examples of 2D annotation include titles, legends, logos and colormaps. Examples of 3D annotation include bounding boxes, axes, text labels and the compass or gnomon that shows the orientation of the data set. All of these items are straightforward to implement in Open Inventor but not necessarily provided as built-in objects. In this document we'll show how to implement some commonly used annotation graphics so you can spend more time implementing your application specific features.

Programming languages: The Open Inventor API is available in specific versions for developers using C++, Java or C# (as well as other .NET conforming languages). The discussion in this document is generally applicable to all languages, but for simplicity the example code is currently only shown in C++. Because the API is intentionally similar in all languages, in most cases it is straightforward to convert the example code into Java or C#. For example:

```
C++:  pViewer->setBackgroundColor( SbColor( 0, .5f, .5f ) );

C#:   Viewer.SetBackgroundColor( new SbColor( 0, .5f, .5f ) );
```

Example programs: Starting with Open Inventor version 8.0, the example programs referenced in this document are installed with the SDK in directory: $OIVHOME/src/Inventor/samples/annotation.

Section 1.2 – Background Features

One type of basic annotation is screen positioned images, for example, the company or application logo. This and a few other small things fall loosely under features of the Open Inventor viewer class, but also include the SoBackground nodes. The following is a quick discussion of some features you may know about, but might not be taking full advantage of.

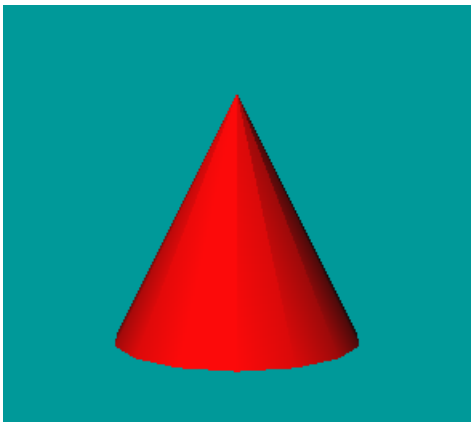
Background color:
By default the background of Open Inventor's 3D window is black (0,0,0). You can change the

background to any RGB value like this:

```
pViewer->setBackgroundColor( SbColor( 0, .5f, .5f ) );
```

You can also change the background color without re-compiling and re-linking, using an environment variable like this:

```
OIV_BACKGROUND_COLOR 0 .5 .6
```



Solid background

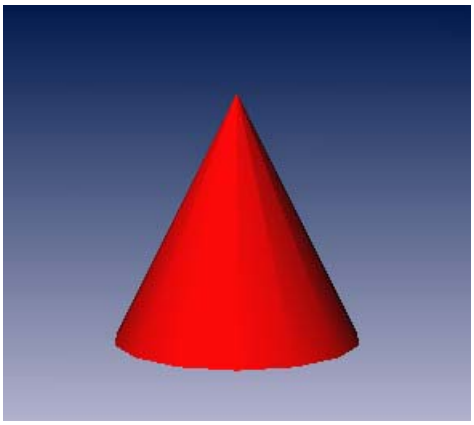
Note: When we say “environment variable”, we mean either an actual shell environment variable or the equivalent keyword in an Open Inventor configuration file (see the help page for SoPreferences). A configuration file is particularly convenient when starting an application from the IDE on Windows, since changing an environment variable doesn't take effect until you restart Visual Studio.

Why would you want to change the background color? There are some cases where it is required, for example, when using MIN_INTENSITY composition in VolumeViz, you must change the background to full white to get a useful image. Changing the background color is occasionally useful as a debugging technique. If there's any possibility that your "missing geometry" is just unlit and completely black, try changing the background color to see if it shows up. Highly colorful backgrounds like the image above may not be appropriate, but neutral non-black backgrounds can be easier on the eyes in data visualization applications.

But wait, we can do so much more with the background!

Gradient Background:
The *SoGradientBackground* node creates a vertical gradient background that exactly fills the window and is smoothly shaded from field *color0* to field *color1*. The default values produce a gradient shaded from dark blue to light blue. This is very attractive for presentations. In fact we like it so much it's the default background in our Avizo and amira applications! For example:

```
SoGradientBackground *pGrdBackg = new SoGradientBackground();
pGrdBackg->color0 = SbColor( 0.0f, 0.1f, 0.3f );
pGrdBackg->color1 = SbColor( 0.7f, 0.7f, 0.8f );
```



Gradient background

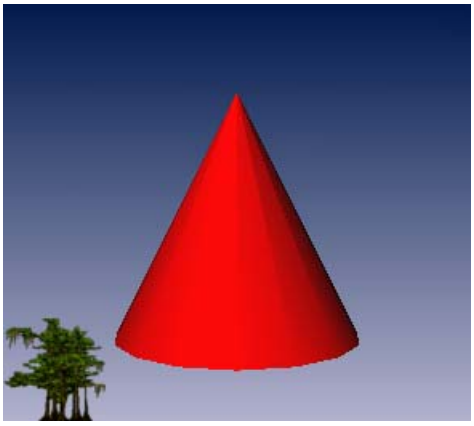
Generally background nodes should be placed at the beginning of the scene graph so that geometry is drawn on top of the background. Background nodes have no size, are not pickable, and do not modify the OpenGL depth buffer.

Image Background

The *SoImageBackground* node provides several convenient modes for placing an image in the background of the window. Just like the texture image nodes, the image can be specified as either a filename or an image in memory. Images containing transparency are blended, allowing them to overlay, for example, a gradient background. Note that the *SoImage* node can also be used to place an image in the scene, but the position of the image is specified in 3D coordinates. SoImageBackground positions images relative to the window.

The *style* field controls how the image is mapped into the window. The "corner" styles place the image, at its native size, in any corner or the center of the window. This is very convenient for displaying logos for demos. For example:

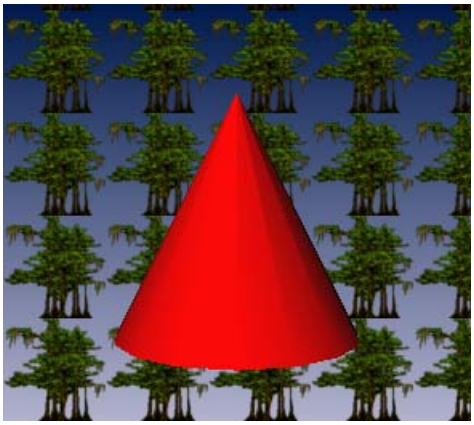
```
SoImageBackground *pImgBackg = new SoImageBackground();
pImgBackg->filename = imgFilename;
pImgBackg->style = SoImageBackground::LOWER_LEFT;
```



Partially transparent image background

The other two styles fill the entire window. The STRETCH style stretchs or shrinks a single copy of the image to fill the window. This has somewhat limited utility because the image may be distorted or low

resolution after stretching. However the TILE style fills the window with multiple copies of the image, calculated so that every copy is the image's native size. This can be useful for slightly unrealistic, but convenient special effects.



Tiled background

A nice demo of these nodes is provided in the Open Inventor SDK: $OIVHOME/src/Inventor/examples/Features/BackgroundNode. The demo includes a DialogViz user interface that allows you to toggle the background nodes on and off and modify their specific features.

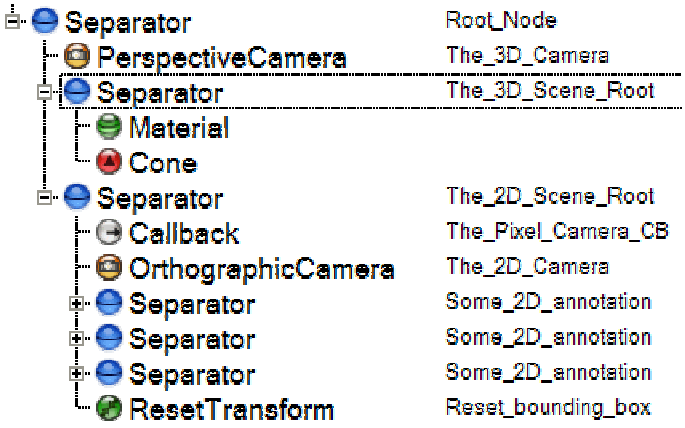
Section 1.3 – 2D Text Annotation

3D applications often need to do a bit of 2D graphics too, particularly screen aligned annotations. These can include images, for example company logos; text, for example the data set name; and 2D graphics like color maps and histograms. Open Inventor does not provide an explicit 2D mechanism, but it does provide a mechanism by allowing multiple cameras (*SoCamera* nodes) in the scene graph.

There are many uses for multiple 3D cameras of course. For example, a *collection* of cameras of which only one is currently in use. A collection of cameras could be useful for storing multiple viewpoints when saving the scene graph. You could also use a collection of cameras under a switch (SoSwitch node) to jump between different viewpoints at run time. There are also cases where you need multiple views simultaneously, for example front and side views, a working (zoomed in) view and a global (bird's eye) view, etc.

The interesting case for 2D annotations is when we have both a "3D" camera, usually an SoPerspectiveCamera, and a "2D" camera, usually an SoOrthographic camera, in the scene graph at the same time. This is perfectly straightforward for Open Inventor because the camera is really just a property node that happens to change the OpenGL viewing and projection matrices. Multiple cameras works for annotation because the viewer can only control one camera at a time. So if the viewer is attached to the 3D camera, the user can move around the scene as usual and the 2D camera does not change.

We use the hierarchy of the scene graph to manage which geometry is viewed by which camera. The "3D" part of the scene graph will contain our application geometry and the "2D" part will contain our annotation geometry. The scene graph might look like this:



Note: This *tree view* of the scene graph is conveniently produced by the IvTune tool provided with the Open Inventor SDK. You can display the IvTune window, browse and modify the scene graph while your application is running, by pressing Shift-F12. We assigned meaningful names to (some of) the nodes using the SoNode::setName() method as usual.

By default the orthographic camera sets up a view volume that is -1 to 1 in both X and Y. So effectively we can define and position geometry in a kind of normalized device coordinates (NDC). Normalized means we can position our annotation geometry independent of the actual window size on the screen. To position some text in the upper left corner we might do this:

```
SoSeparator   *pTextSep1 = new SoSeparator();
SoTranslation *pTran1    = new SoTranslation();
SoText2       *pText1    = new SoText2();
pTran1->translation = SbVec3f( -.99f, .9f, 0 );
pText1->string = "Upper Left";
pTextSep1->addChild( pTran1 );
pTextSep1->addChild( pText1 );
```

Here's a potential gotcha. "viewAll" is a convenient and frequently used operation that automatically adjusts the camera so that the entire scene is visible. When the viewAll() method is called it applies an SoGetBoundingBoxAction to the scene graph, then adjusts the view volume to contain this box. So, in some cases (specifically when the extent of the application is less than 1), our 2D annotation geometry might affect the size of this bounding box, with surprising (at least to the user) results. To avoid this problem we add an **SoResetTransform** node as the last child of the 2D scene root (see scene graph structure above) and set its *whatToReset* field to BBOX. This effectively makes our 2D geometry invisible to the GetBoundingBox action.

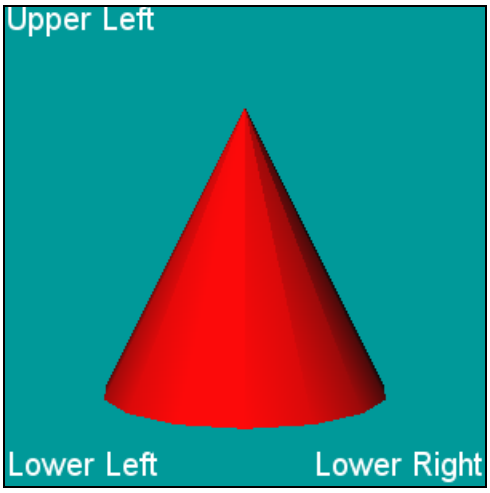
```
SoResetTransform *pResetBbox = new SoResetTransform();
pResetBbox->whatToReset = SoResetTransform::BBOX;
p2dSep->addChild( pResetBbox );
```

Here's another small gotcha. Some Open Inventor programs rely on the viewer to automatically create a camera. What happens is that when setSceneGraph() is called, the viewer searches the scene graph for an existing camera. If no camera is found, the viewer creates a camera and inserts it in the "viewer scene graph" above the application scene graph. If at least one camera *is* found, the viewer attaches to the first camera found. So if we put a 2D camera in the scene graph for annotation, we cannot rely on the viewer to create a 3D camera because the viewer will just find and attach to the 2D camera. The best solution is to explicitly create both cameras.

Some Open Inventor programs also rely on the viewer automatically doing a "view all" when the scene graph is first loaded (to ensure that the entire scene is initially visible). In fact the automatic “view all” *only* happens when the viewer automatically creates a camera (otherwise it would be modifying an existing camera). So in this case we need to initialize our 3D camera either by knowing good initial settings or by explicitly calling the viewer's viewAll() method. Finally, note that the viewAll() method does not automatically save the new camera settings as the "Home" position. So normally we will call the viewer's saveHomePosition() method after the initial viewAll(), because the user will expect the Home button to return to this view.

```
myViewer->setSceneGraph( pRoot );
myViewer->viewAll();
myViewer->saveHomePosition();
```

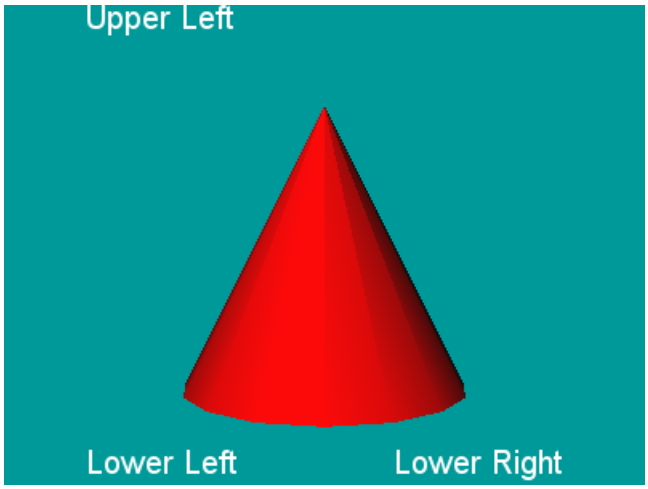
A simple example program is provided with this document: “Annotation2D_P1.zip”. It should look similar to the following image. If you build and run this program you will see that the cone can be rotated, zoomed, etc without any effect on the 2D annotation strings.



First 2D annotation example

You can also check out the VolRend demo ($OIVHOME/src/VolumeViz/VolRend) to see this technique used for the data set title, color map and histogram.

Unfortunately 2D annotation using normalized device coordinates has some problems. The annotation geometry is effectively positioned at some *percentage* of the window. So making the window larger makes the annotation positions drift away from the edges of the window. Even worse, if the user can reshape the window, for example from square to rectangular, the annotations completely abandon the window edges, as shown here:



Not exactly what we had in mind...

What we really need is to be able to position and draw annotations directly in pixel coordinates! A convenient way to do this would be to have a "pixel space" camera node that automatically updates itself to match the current viewport dimensions. Open Inventor doesn't have a built-in mechanism to do this, but it does provide the tools we need to build one:

- 2D View
  The *SoOrthographicCamera* node can be considered a 2D view. To make it a pixel space camera we just need to define a view volume that goes from 0 to N-1, where N is the viewport width or height in pixels.

- Current viewport dimensions
  At the beginning of each traversal Open Inventor stores the current viewport dimensions in the *SoViewportRegionElement* (SoLOD and other nodes use this information). So we can get the current viewport at traversal time and update the pixel space camera.

- Automatic update
  The *SoCallback* node allows us to create a "quick and dirty" custom node by executing an application defined function every time it is traversed. (It might be more elegant to create an actual custom node, but we don't need it for a simple example.)

Every time we need a pixel space camera we'll create an orthographic camera and a callback node to modify it with the current viewport dimensions. Every time the callback node is traversed it will execute the following function, where *userData* is some data specific to this instance, for example, the camera node to be modified:

```
void pixelCameraCB( void *userData, SoAction *pAction )
{  . . .  }
```

In this function we can get the current viewport dimensions from the traversal state like this:

```
SoState *pState = pAction->getState();
const SbViewportRegion &vpRegion =
  SoViewportRegionElement::get( pState );
const SbVec2i32 viewport =
  vpRegion.getViewportSizePixels_i32();
```

Now we need to understand a little bit about how Open Inventor camera nodes set up an orthographic view volume. The view volume height is defined by the camera's *height* field. This will be the viewport height in pixels. The view volume width is defined by the camera's *aspectRatio* field. Aspect ratio is the viewport width divided by the viewport height. Open Inventor first computes a view volume centered around 0,0. In other words from -N/2 to N/2, where N is the width or the height. Next it applies a translation to the view volume using the value of the camera's *position* field. In order to transform the (-N/2)..(N/2) view volume into the desired 0..(N-1) view volume, we just need to do an additional translation by (N-1)/2.

```
pCamera->height      = (float)vpHeight;
pCamera->aspectRatio = (float)vpWidth / vpHeight;
float xRadius = 0.5f * (vpWidth - 1);
float yRadius = 0.5f * (vpHeight - 1);
pCamera->position = SbVec3f( xRadius, yRadius, 2 );
```

Creating the callback node is simple. When we set the callback, the first parameter is the function to call and the second parameter is the value to pass as the "userData" parameter.

```
SoCallback *p2dCamCB = new SoCallback();
p2dCamCB->setCallback( pixelCameraCB, (void*)pInfo );
```

That will take care of positioning and drawing relative to the lower left corner of the window. For example, the coordinate 5,5 is five pixels right and five pixels up from the lower left corner (in OpenGL). But we still have a problem positioning in the other corners because we don't know the maximum pixel value in advance and we don't want to modify our geometry every time the window size changes. We need a way to specify positions relative to the right and top edges of the window as well. We can do that by instantiating a pixel space camera with one or both of the axes inverted. An inverted X axis goes from -(N-1) to 0. So the coordinate -5,5 is five pixels left and five pixels up from the lower right corner of the window. We'll need to instantiate a separate camera for each corner, but that's not a problem.

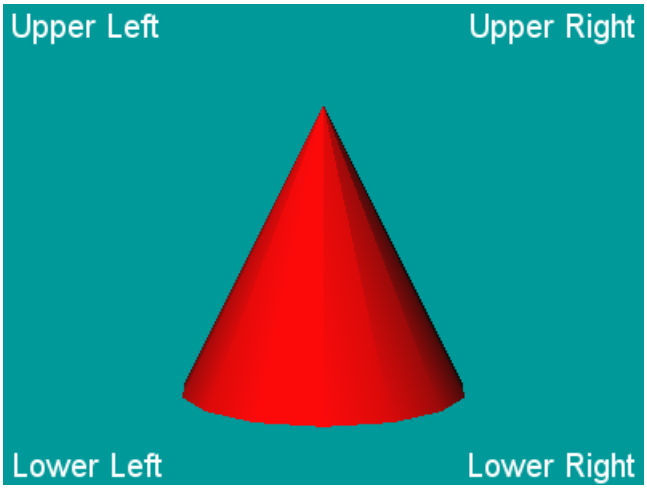
In the example program we define a convenience function *addPixelSpaceCamera* that adds both the camera node and the callback node that modifies it, into the specified group node. The *invertX* and *invertY* parameters specify if the axes are inverted. For example for the lower left corner we would pass FALSE and FALSE, but for the upper left corner we would pass FALSE and TRUE.

```
void addPixelSpaceCamera( SoGroup *pParent,
                          SbBool invertX,
                          SbBool invertY )
{ . . . }
```

Placing a 2D annotation text string in the upper left corner can now be done as follows. The text string will be positioned 5 pixels from the left edge and 20 pixels down from the top edge (because the Y axis is inverted). The Y position is 20 because the example program sets the font size to 20 and we know the actual characters are normally slightly smaller than the font size.

```
SoSeparator *pTextSep1 = new SoSeparator();
addPixelSpaceCamera( pTextSep1, FALSE, TRUE );
SoTranslation *pTran1 = new SoTranslation();
pTran1->translation = SbVec3f( 5, -20, 0 );
SoText2 *pText1 = new SoText2();
pText1->string = "Upper Left";
```

A simple example program is provided with this document: “Annotation2D_P2.zip”. It should look similar to the following image. If you build and run this program you will see that the window can be resized and reshaped without any effect on the 2D annotation strings.

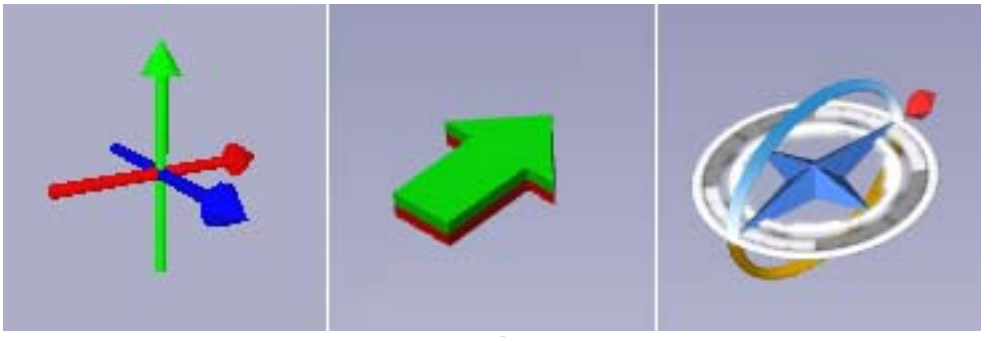


Much better!

Section 1.4 – Gnomon Annotation

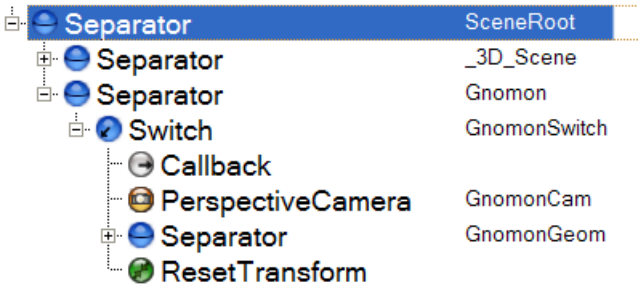
Actually this annotation is 3D and dynamic! But like the previous examples, it is locked to a specific region of the window. This annotation is called a "gnomon" (pronounced NO-mon) or sometimes just a "compass". Its purpose is to continuously display the 3D orientation of the scene. This helps users understand what they are looking at.

When the scene exists in an arbitary coordinate system, for example in computer aided engineering, the gnomon might be drawn as a small set of XYZ axes. When the scene is geo-referenced, for example in seismic interpretation, the gnomon might look like part of a real world compass. We will only assume the gnomon is defined by some 3D geometry that is symmetric around the point 0,0,0 (this is only a simplifying assumption, not a requirement).


Some example gnomons

Similar to the 2D Text Annotation project, we will create a second camera node to render the gnomon and a callback node to allow us to dynamically modify the camera during scene graph traversal. The scene graph (below) is very similar to the previous project, including a ResetTransform node to keep the

gnomon geometry out of the viewer's bounding box calculation. One difference is that we're using an SoPerspectiveCamera to render the gnomon because we want to define its appearance using 3D geometry. We've also added a Switch node so we can conveniently turn the gnomon off.



Scenegraph including 3D scene and gnomon

Here are the key requirements for the callback function:

1. Restrict rendering of the gnomon to a pixel region.
2. Ensure that the gnomon is rendered "in front of" everything.
3. Match the gnomon orientation to the scene orientation.

Requirement 1.
To restrict rendering of the gnomon to a pixel region, for example the lower left corner of the window, we will modify the OpenGL viewport. We don't currently have a Viewport node, but it's easy to modify the viewport in a callback function. We could call OpenGL directly, but setting the *SoViewportRegionElement* in the traversal state is safer and more convenient. This will modify the OpenGL viewport and also ensure that any other nodes in the scene graph that depend on the viewport are rendered correctly.

```
// This gnomon position is in the lower-left corner.
SbViewportRegion vport( GNOMON_WIDTH, GNOMON_HEIGHT );
SoViewportRegionElement::set( state, vport );
```

Requirement 2.
To ensure that the gnomon is rendered in front of the application scene the simplest solution is to render the scene first, clear the OpenGL depth buffer, then render the gnomon. We could do fancy things with the background of the gnomon area, but as they are purely esthetic we won't bother in this example. Currently the only way to clear the depth buffer during traversal is to make a direct call to OpenGL. You may have to include the OpenGL header file and modify your project/make file to link in the OpenGL library (neither is necessary on Windows - see the example program).

```
// Reset depth buffer so gnomon is always "on top"
glClear( GL_DEPTH_BUFFER_BIT );
```

Requirement 3.
Matching the gnomon orientation to the scene orientation requires two steps. First we have to get the current orientation of the scene camera. We could try to keep a pointer to the scene camera, but this approach is very "fragile" and not recommended. For example, if the user changes the camera type by clicking the viewer button, the viewer destroys the current camera and creates a new one, leaving our pointer invalid. A safe approach is get the viewing matrix from the traversal state and extract the rotation part of the matrix (the cameraRotation variable in the code below). Open Inventor conveniently keeps

separate model and view matrices (unlike OpenGL). Note that the viewing matrix is applied to geometry, so we actually need the inverse of this matrix. The following code gets the viewing matrix from the traversal state, requests the inverse matrix, then (all in the same line) requests a decomposition so we can get just the rotation part of the matrix.

```
// Get camera rotation from viewing matrix.
SbMatrix vMat = SoViewingMatrixElement::get( state );
SbVec3f tran, scale;
SbRotation cameraRotation, orient;
vMat.inverse().getTransform( tran, cameraRotation, scale, orient );
```

Now we have to update the gnomon camera. We'll pass a pointer to this camera as the callback function's "userData". This should be safe because the viewer does not control this camera and the user cannot affect it directly. We can update the camera orientation directly with the cameraRotation we extracted above:

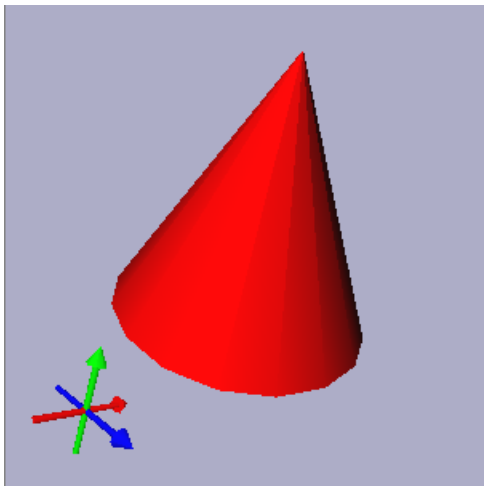
```
// Get ptr to the gnomon camera
SoCamera *camera = (SoCamera*)userData;

// Set the new orientation for the gnomon camera
camera->orientation = cameraRotation;
```

Now the gnomon camera is rotated to the right direction, but it probably isn't pointing at the gnomon geometry anymore! We now need to move the rotated camera so the gnomon is in front of the camera. Specifically, the "lookAt" point, or point of rotation, for an Open Inventor camera is located *focalDistance* (a field of SoCamera) units from the camera, measured along the view direction vector. We know this point should be the center of the gnomon, so we need to move the (rotated) camera to a position that is focalDistance units away from the gnomon center, along the view direction vector. (This is similar to what the ExaminerViewer does when you "spin" the camera around the point of rotation and it's a useful technique to know.)

Get the focal distance from the gnomon camera. We already know the lookAt point (the center of our gnomon, which is 0,0,0 in this case). Convert the rotation (SbRotation) to a matrix (SbMatrix) and extract the direction vector (see Graphics Gems or other reference for a explanation of this trick). Compute a position *distance* units away from 0,0,0 along the direction vector.

```
float distance = camera->focalDistance.getValue();
SbMatrix mx;
mx = cameraRotation;
SbVec3f direction( -mx[2][0], -mx[2][1], -mx[2][2] );
camera->position = SbVec3f(0,0,0) - distance * direction;
```

A simple example program is provided with this document: "Annotation2D_P3.zip". It should look similar to the following image. If you build and run this program you will see that rotating the scene using left-mouse or thumbwheels causes the gnomon to rotate in sync, but panning or zooming the scene does not affect the gnomon:



Gnomon showing orientation of cone.

Section 1.5 – Bounding Box Annotation

In the previous sections we looked at several kinds of basic 2D annotation, including pixel positioned text (also applicable to legends, color maps, etc) and a gnomon or compass to show the current scene orientation. Now let's look at some basic 3D annotation. This is annotation that is part of the 3D scene. It is normally viewed by the same 3D camera used for the application's data, so it is moved, rotated and scaled with the rest of the scene. 3D annotation is additional geometry that helps the user understand the scene and interpret the data. Typical examples are bounding boxes, axes, text labels and so on.

First let's look at constructing a visual representation of a bounding box. This is useful for various kinds of data, but particularly for volume data. In this case we may only be rendering a few slices through the volume or a subvolume (ROI). The visual bounding box shows us the full extent of the volume. This allows us to see how the portion of the volume currently being rendered relates to the overall volume. It also allows us to see how far we can move a slice or subvolume before hitting the limit of the volume. I find it useful to keep the code to build this in my "utilities" directory, so it's easy to add a bounding box to test programs.

It is not difficult to construct geometry to represent the bounding box. It's basically just a "wireframe" box. But it provides a nice example of constructing Open Inventor geometry and shows a few techniques that might not be obvious when first starting to use Inventor. For example, we sometimes see wireframe boxes constructed using an SoCube node and an SoDrawStyle node with the style field set to LINES. This may be the quickest solution and is visually correct to a first approximation, but there are some subtle issues. For example, Inventor picking still considers the sides of the box to be polygons. For another, we can only apply a single color to all the edges.

We could define a new class, but to keep the example as simple as possible we'll just create a factory function that returns an SoSeparator node. If we ever need to change the extent of the bounding box, we can just delete the current one and create a new one. The function *makeBBox()* returns a Separator because we'll need multiple nodes and the Separator groups them together for us and because we don't want the property nodes for the box to affect anything else in the scene graph. We'll specify the extent of the box using an SbBox3f object. The two main properties of the box we might want to control are the color and the line width. We'll make the default color red and the default line width 2, so the box is easier

to see.

Here is the declaration of that function:

```
SoSeparator *makeBBox(
    SbBox3f box,
    SbColor color=SbColor(1,0,0),
    float   width=2 );
```

The first thing we'll do is disable lighting for the box. This makes it easier to see because it will have a constant color, independent of the view and light directions. To turn off lighting in Open Inventor we actually set the lighting model, like this:

```
// The box will be easier to see without lighting
SoLightModel *pLModel = new SoLightModel;
pLModel->model = SoLightModel::BASE_COLOR;
```

Next we'll set the line width for the box:

```
// And with wide lines
SoDrawStyle *pStyle = new SoDrawStyle;
pStyle->lineWidth = width;
```

And we'll also specify that the box is not *pickable*. This ensures that the box doesn't prevent the user from selecting geometry or using draggers and manipulators inside the box.

```
// The box should be unpickable
SoPickStyle *pPickable = new SoPickStyle;
pPickable->style = SoPickStyle::UNPICKABLE;
```

We'll store the coordinates and colors for the box in an *SoVertexProperty* node. The box is defined by the coordinates of the eight corners, which we get by querying the bounds of the box. For simplicity we'll specify a single color for the entire box, but we could easily use a different color for each edge or a different color for each axis. Note that the color was passed in as an SbColor (three floats) but the VertexProperty expects an RGBA value. SbColor provides the *getPackedValue()* method to do this conversion.

```
// Create a cube outlining the specified box
float xmin, xmax, ymin, ymax, zmin, zmax;
box.getBounds( xmin,ymin, zmin, xmax, ymax, zmax );

SoVertexProperty *pProp = new SoVertexProperty;
pProp->vertex.set1Value( 0, SbVec3f(xmin,ymin,zmin) );
pProp->vertex.set1Value( 1, SbVec3f(xmax,ymin,zmin) );
pProp->vertex.set1Value( 2, SbVec3f(xmax,ymax,zmin) );
pProp->vertex.set1Value( 3, SbVec3f(xmin,ymax,zmin) );
pProp->vertex.set1Value( 4, SbVec3f(xmin,ymin,zmax) );
pProp->vertex.set1Value( 5, SbVec3f(xmax,ymin,zmax) );
pProp->vertex.set1Value( 6, SbVec3f(xmax,ymax,zmax) );
pProp->vertex.set1Value( 7, SbVec3f(xmin,ymax,zmax) );
pProp->orderedRGBA.set1Value( 0, color.getPackedValue() );
```

The box will be drawn by an *SoIndexedLineSet* node. This node allows us to draw multiple lines, each with multiple segments. Each line is specified by a list of indices identifying the coordinates to be used.

Each list of indices is terminated by a -1 value. Using indices allows us to use the coordinates multiple times without duplicating them. It also allows us to use a predefined set of indices (defining the shape of the geometry) without knowing the actual coordinates until runtime. In this case we're drawing the four segments of the front face, four segments of the back face, and then four single segments to connect them. The index values here correspond to the index values in the set1Value calls above.

```
// Draw it with a line set
int coordIndices[] = {0,1,2,3,0,-1,4,5,6,7,4,-1,
                      0,4,-1, 1,5,-1, 2,6,-1, 3,7};
int numCoordIndices = sizeof(coordIndices)/sizeof(int);
```

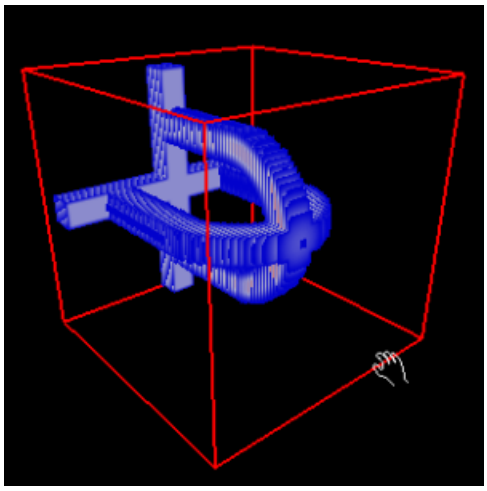
After creating the node, we store the SoVertexProperty node containing our coordinates and colors in the *vertexProperty* field and store the indices in the *coordIndex* field. The *setValues* method takes a start index, the number of values to set and an array of values.

```
SoIndexedLineSet *pLines = new SoIndexedLineSet;
pLines->vertexProperty = pProp;
pLines->coordIndex.setValues(
    0, numCoordIndices, coordIndices );
```

Finally we assemble the scene graph and return the root SoSeparator object:

```
SoSeparator *pBoxSep = new SoSeparator;
pBoxSep->addChild( pLModel );
pBoxSep->addChild( pPickable );
pBoxSep->addChild( pStyle );
pBoxSep->addChild( pLines );
return pBoxSep;
```

A simple example program is provided with this document: “Annotation3D_P1.zip”. Here is an image of the bounding box with the syn_64 volume data set.



Bounding box

What happens when the scene bounding box changes? For example when we load a new volume with a different extent. The easiest thing is to simply delete the existing bounding box geometry and recreate it with another call to the makeBBox() function. However we can easily imagine more elaborate implementations such as creating a custom nodekit or a custom node. A custom node could even

automatically resize itself based on the current volume data in the traversal state list. But that seems like a lot of work to solve this small problem.

Notice that we could easily make the box convey even more information to the user. For example we could color the edges of the box according to their corresponding coordinate axis, providing the user with visual cues about the orientation as well as the extent of the scene.

- End -