Interactive 3D Worlds with Coin and Qt

Drawing in 3D

In a realistic 3D world the observer becomes an actor whose actions provoke reactions that leave tracks in virtual reality. Qt and Coin allow you to program animated and interactive 3D worlds quickly and easily. We take a look at how you can interact with your new 3D world and create new effects.

BY STEPHAN SIEMEN

A three dimensional scene appears far more realistic if it is animated and the user can interact with it. The previous article in our Coin series [2] introduced the concept of animation with a plane circumnavigating a revolving globe. However, the scene did not allow the user to interact with, or modify the scene, although it was possible to use *SoQtExaminerViewer* to view the scene from different perspectives.

Our interactive example in this article is a simple drawing program that allows the user to draw arbitrary dots in a 3D scene. This illustration is based on an example from Inventor Mentor ([3], Chapter 10, example 2), however, we have expanded on it and moved from Motif to using Qt and SoQt.

Figure 2 shows the scene graph for the drawing program. It requires only four nodes, and the user can edit three of them dynamically (only the lighting remains constant). Pressing the center mouse button rotates the camera about the source. *dotCoordinates* and *dots* are the most interesting nodes:

- *dotCoordinates* are objects of the *SoCoordinate3* class that expects an arbitrary number of 3D coordinates.
- *dots* are objects of the *SoDotSet* class that draws all the coordinates for the *dotCoordinates* node as a dot in a 3D space.

The user can add new dots by pressing the left mouse button. Clicking the right mouse button deletes all the dots from the scene.

Coin 3D Update

The new version of Coin was released shortly before this article was printed. Version 2.0 contains the VRML extension as mentioned in this article and can be downloaded from [5].

The Right Choice

A program that needs to reflect a user's wishes interactively, needs to select and modify individual objects in the scene graph. This example adds new dots to *dotCoordinates* and tells the *dots* node to draw the new dots.

Nodes can be accessed and selected by reference to their position in the graph. Each group node provides a method called *getChild()* for this purpose. Figure 2 shows how the program references individual nodes via the root of the scene graph. However, this simple method is extremely error-prone: if

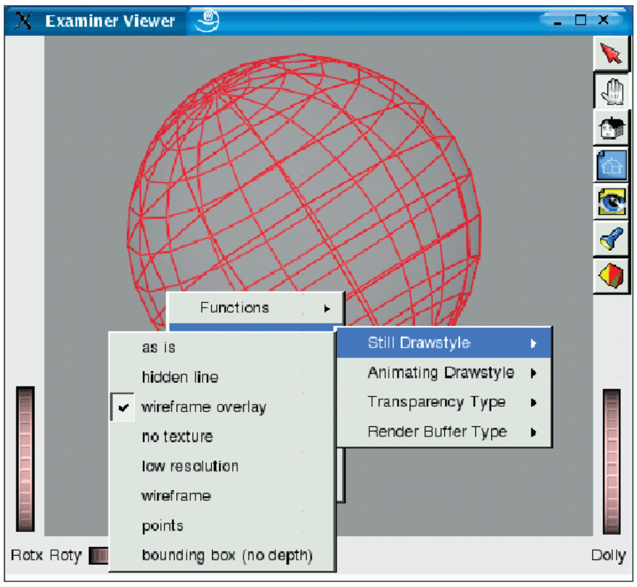

Figure 1: The default right-mouse-button menu gives many options, such as rendering only a wireframe of the scene.

nodes are added to the scene graph later, the indices change. Also, larger scene graphs tend to become unmanageable – in this case you might prefer to use the *setName()* method to assign a unique name to each node. A *SoSearchAction* class object can then be used to discover and change the position of the object.

The *newDot()* in Listing 1 adds a new dot to the scene. It expects to be passed a pointer to the rendering area and the coordinates of the dot. Lines 5 to 7 discover the pointers for the third and fourth nodes in the scene graph starting at the root node. Finally, lines 10 and 12 modify the characteristics of the node.

Adding Dots to the Scene

When a user adds dots to a scene the program has to correlate the two-dimensional mouse position in the window to the three-dimensional position in the scene. Our example uses a simple approach placing all the dots on a plane that transects the origin.

Figure 3 shows the geometry of the volume that *SoPerspectiveCamera* sees as a section of the 3D scene. If the camera range is set to infinity, rendering can take a while. This is why the *nearDistance* and *farDistance* parameters restrict the volume of the camera viewfinder. In our example *nearDistance* is set to *1* and *farDistance* to *7*. As the distance between the camera and the origin is *4*, the focus of the scene is exactly in middle of *nearDistance* and *farDistance*. The focal point is traversed by the plane (highlighted in green) on which the user can draw dots.

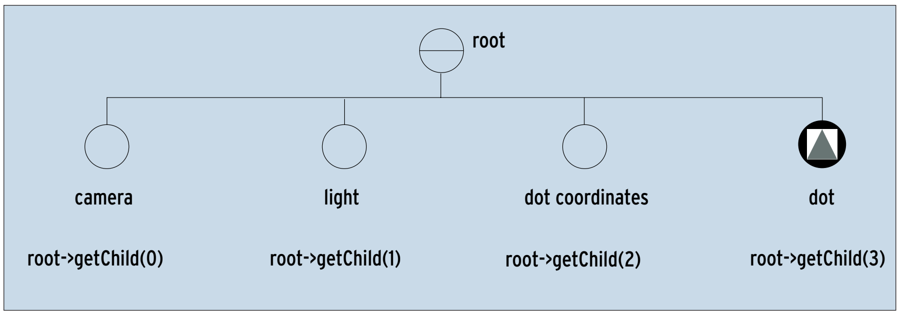

Figure 2: The neat scene graph for a 3D drawing program. The four children of the root node can be queried using the *getChild()* method

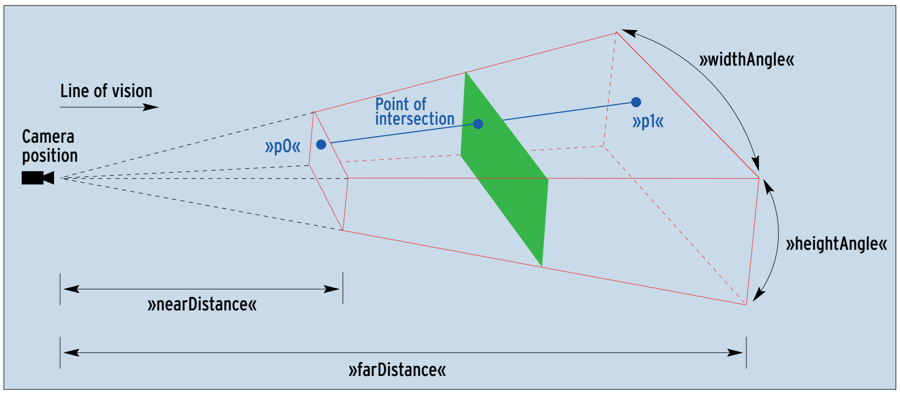

Figure 3: In Coin the observer looks at a 3D scene through a camera. The camera only shows a section of the world that is limited by four margins and two depth layers (red)

Listing 1: New Dots

```
01 // Add new dots to the scene
02 void newDot(SoQtRenderArea *renderArea, const SbVec3f dot)
03 {
04    // Pointer to other nodes in the graph
05    SoGroup *root = (SoGroup *) renderArea->getSceneGraph();
06    SoCoordinate3
*dotCoordinates = (SoCoordinate3 *) root
->getChild(2);
07    SoDotSet *dots = (SoDotSet *) root->getChild(3);
08
09    // Add coordinates
10    dotCoordinates->dot.set1Value(dotCoordinates->dot.getNum(), dot);
11    // Get the dot node to draw all the coordinates
12    dots->numDots. setValue(dotCoordinates->dot.getNum());
13 }
```

By default units of distance are relative to one another in Open Inventor, however, you can use the *SoUnit* node to assign a specific unit of distance.

The Mouse in the Volume

Listing 2 maps the x and y mouse coordinates on this volume. Imagine this as a ray that vertically transects the camera volume. Figure 3 shows this ray in blue. The individual steps are as follows:

- Lines 7 to 9 convert the mouse coordinates from the X11 window to the rendering area units (value between *0* and *1*). The origin is bottom left, in contrast to top left for X11.
- Line 17 gets an *SbViewVolume* class object that describes the visible section of the scene, thus providing a correct viewing perspective.
- Line 21 extends the mouse pointer, converting the dot into a line that extends vertically into the image. The end points are stored in *p0* and *p1*.
- The drawing layer is half-way through the image and transects the line exactly at its mid-point.

This is only one of many approaches to transferring a mouse position to a scene. More complex approaches map the mouse pointer to three dimensional objects.

Additional details are available in the online documentation for Coin [6] and Inventor Mentor [3], [4]. The *SoPickAction* and *SoPickedDot* classes are particularly useful.

Engines, Sensors & Callbacks

Open Inventor provides two techniques for programming animations: engines

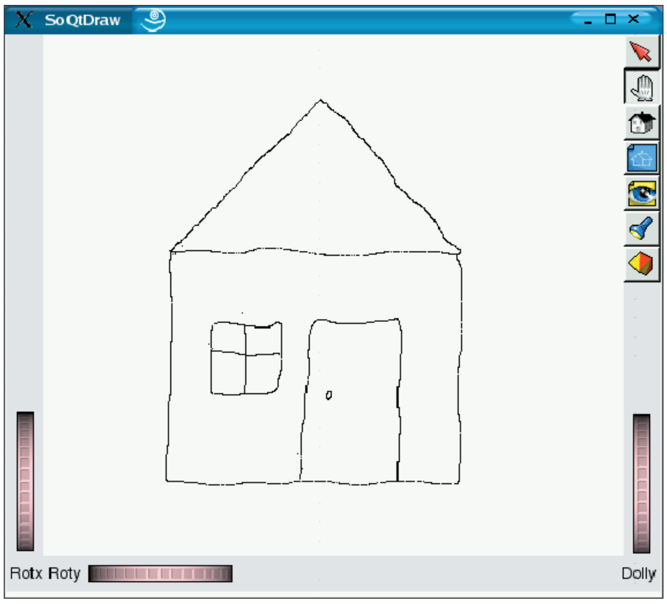

Figure 4a: The example program initially looks like a 2D drawing program. You use the left mouse button to draw arbitrary figures

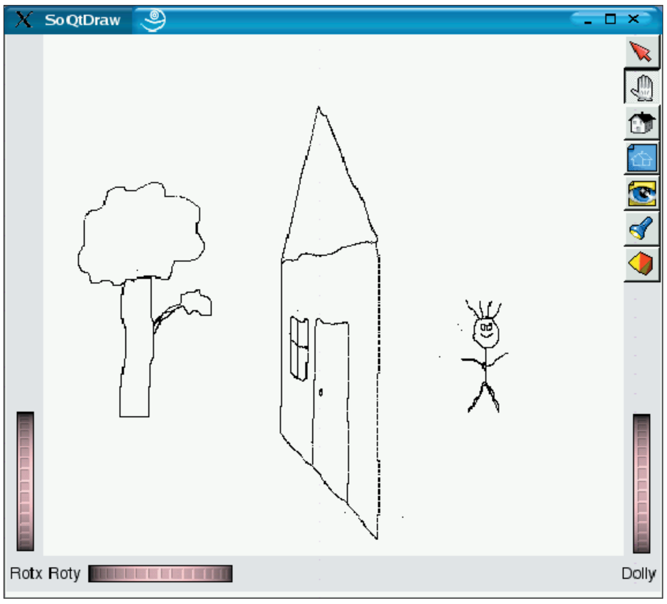

Figure 4b: When the user moves the camera the 3D capabilities of the program become apparent. New dots are shown on a new layer

and sensors. Engines are objects that link to node fields in the scene graph and change their values. Changes are effected either by logical links or by time functions. As the engines are part of the scene graph they can be stored in files along with their characteristics.

In [2] we used two engines to rotate a globe. Sensors are objects that recognize changes in the scene graph and react by referring to a callback function. In comparison to engines, sensors provide the programmer with more potential, as they allow original code to be added. But this also means that sensors are external to the scene graph and cannot be stored in the scene graph files.

Our example uses a sensor to rotate the camera in the scene. When the user presses the center mouse button the code in Listing 3a applies the timer sensor, *SoTimerSensor*, to the camera. The sensor refers to its callback function (Listing 3b) at regular intervals to gradually rotate the camera. Other sensors, such as *SoNodeSensor*, react to changes in monitored nodes and react by calling their own callback functions.

One advantage that callbacks offer is the fact that they can do more than modify the scene graph. A sensor can, for example, write to the terminal whenever a characteristic of a monitored object changes. However, callbacks should not contain too much code, as they may be called frequently, and this could impact program speed.

Listing 2: Mouse based projections

```
// Uses the mouse position to calculate
// a 3D dot in the scene
void projection(SoQtRenderArea *renderArea,
   int mouseX, int mouseY, SbVec3f &interface)
{
   // Position of mouse in rendering area:
   SbVec2s size = renderArea->getSize(
   float x = float(mouseX) / size[0];
   float y = float(size[1] - mouseY) / size[1];

   // Pointer to camera
   SoGroup *root = (SoGroup *) renderArea->getSceneGraph();
   SoCamera *camera = (SoCamera *) root->getChild(0);

   // Volume visible to camera
   SbViewVolume cameraVolume;
   cameraVolume = camera->getViewVolume();

   // Mouse line vertical to scene
   // The other endpoint is created by mirroring on the plane
   SbVec3f p0, p1;
   cameraVolume.projectDotToLine(SbVec2f(x,y), p0, p1);

   // The center of the line is the position we are looking for
   // on the surface, which goes through the origin
   interface = (p0 + p1) / 2.0;
}
```

Callback Parameters

In Coin, callbacks have two parameters. The first is a pointer to a *void* type that allows the programmer to pass a reference to an arbitrary object to the function. Thus, Listing 3b expects a *SoCamera* class object, Listing 3a having first passed the reference to the sensor constructor. The second parameter is a *SoSensor* type pointer that passes information from the calling sensor. However, Listing 3b does not use this parameter.

When a user drags the mouse or presses a button, the X Server notes the action and passes it to the program as a so called event. Qt uses the *QEvent* class to provide a simple approach to providing event handling. SoQt allows you to add a callback function to the rendering area. SoQt then calls the callback whenever a *QEvent* type event occurs. In our example the function is called *eventHandler()* and is added to the rendering area as follows:

```
renderArea->setEventCallback(
  eventHandler, renderArea);
```

The code for the *eventHandler()* function is shown in Listing 4.

Event Handling in a Switch Block

The callback function mainly comprises a Switch-Case construct that queries various events. Again, this function should not be too bulky, as it is called whenever event occurs and could thus impact seriously on a program performance. However, Qt also provides more intelligent ways of handling individual events without needing to evaluate each event. The Qt online documentation provides more detail.

To trap events, the program needs to initialize an event queue that remains active throughout runtime. As this queue is not terminated until the program ends, it has to be started with the last command in *main()*.

The complete program is available from [9]. After compiling and launching the program, it should appear as shown in Figures 4a and 4b. You can use the left mouse button to draw arbitrary two dimensional figures, and rotate them with the control elements at the edge of the window. Before we complete this mini-series on Coin and Open Inventor, let's first take a look at one or two more features.

VRML Import

In the previous article [2], we learnt that it is quite easy to convert VRML 1.0 files to Open Inventor format. The next Coin version (2.0) promises more support for VRML versions 1.0 or later.

Some VRML nodes have different properties, despite keeping the same name. To use these new nodes in Open Inventor it was necessary to assign new names to them.

The *SoVRML* prefix was used for classes that behave differently in later VRML versions. Thus *SoSphere* and *SoVRMLSphere* will describe a globe in future. Refer to the Coin 2.0 beta documentation for more details [6].

OpenGL Is Alive

Users with 3D graphics in OpenGL code can continue to use them after moving to Open Inventor. As the latter is based on OpenGL, and uses the OpenGL state machine, it is quite simple to integrate OpenGL code in an Open Inventor application. You would typically add the code to a callback function that the scene graph calls. OpenGL code cannot be stored in files, as it does not belong to the graph itself.

Nodekits are another Open Inventor technique – a kind of blackbox that conceals an internal scene graph of its own. They can be used just like any other Open Inventor class and possess author-definable properties. Nodekits are a perfect solution for creating libraries.

Complex Objects

More complex surfaces and geometric models, such as Bezier curves, are quite easy to model in Open Inventor. This type of object uses the following principle: Just like the *SoDotSet* node in the previous example, all of the points that make up the geometry of the object must be stored in one or more nodes of the scene graph. The geometry that defines the form is added later. This makes it easy to change the geometry, but retain the same anchor points.

Listing 3a: Sensor

```
// The timer sensor passes time impulses to the camera
// while the center mouse button is held down
ticker = new SoTimerSensor(tickerCallback, camera);
ticker->setInterval(UPDATE_RATE);
```

Listing 3b: Callback

```
void tickerCallback(void *data, SoSensor *)
{
   // Pointer to camera
   SoCamera *camera = (SoCamera *) data;
   SbRotation rot;
   SbMatrix   mtx;
   SbVec3f    pos;

   // Rotate camera
   pos = camera->position.getValue();
   rot = SbRotation(SbVec3f(0,1,0), ROTATION_ANGLE);
   mtx.setRotate(rot);
   mtx.multVecMatrix(pos, pos);
   camera->position.setValue(pos);

   // Correct orientation of camera
   camera->orientation.setValue(camera
->orientation.getValue() * rot);
}
```

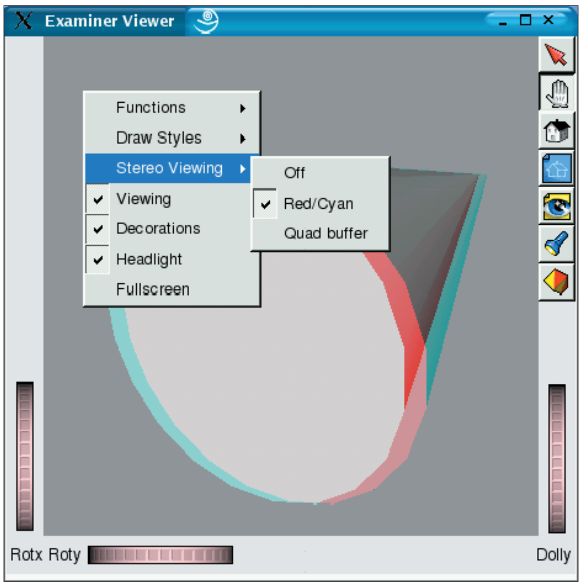

Figure 5: Coin allows you to create color stereo graphics of scenes. Red/cyan specs give depth to the 3D cone in the eyes of the user

Stereo Graphics

Stereo graphics help to make three dimensional objects more realistic. What really happens is that both eyes see slightly different images of the same object from slightly different angles. The brain extracts the three dimensional information from the differences between the two. To display realistic 3D, a 2D screen must display two different images for the left and right eye.

Most techniques use a bespoke hardware for this purpose, such as 3D specs, a special graphics adapter or screen. Coin supports so called Anaglyph Stereo, using two different colors to superimpose two images and send them to the viewer. Spectacles with different colored lenses separate the images. Figure 5 shows an example: Coin displays the white cone in red and cyan, if the corresponding stereo option is enabled in the menu.

Listing 4: Event handling

```
SbBool eventHandler(void *data, QEvent *anEvent)
{
   // Pointer to rendering area
   SoQtRenderArea *renderArea = (SoQtRenderArea *) data;
   QMouseEvent *MouseEvent;
   SbVec3f vector;
   SbBool handled = TRUE;

   // What event has happened?
   switch(anEvent->type()) {
      case QEvent::MouseButtonPress:
         // Mouse key pressed
         MouseEvent = (QMouseEvent *) anEvent;

         if(MouseEvent->button() == Qt::LeftButton) {
            // Left mouse key: new dot
            Projection(renderArea,MouseEvent->x(),
                       MouseEvent->y(), vector);
            newDot(renderArea, vector);
         }
         else if(MouseEvent->button() == Qt::MidButton) {
            // Center mouse key: rotate camera
            ticker->schedule();
         }
         else if(MouseEvent->button() == Qt::RightButton) {
            // right mouse key: delete all dots
            deleteDots(renderArea);
         }
         break;
      case QEvent::MouseButtonRelease:
         // Mouse button is released
         MouseEvent = (QMouseEvent *) anEvent;
         if(MouseEvent->button() == Qt::MidButton) {
            // Center mouse key: camera stable
            ticker->unschedule();
         }
         break;
      case QEvent::MouseMove:
         // The Mouse is moved
         MouseEvent = (QMouseEvent *) anEvent;
         if(MouseEvent->state()) {
            // Key pressed
            Projection(renderArea,MouseEvent->x(),
                       MouseEvent->y(), vector);
            newDot(renderArea, vector);
         }
         break;
   }
   return handled;
}
```

Useful Online Sources

If you need more information on enhancements and features for Open Inventor, you might like to try the Coin [5] and TGS [7] web sites. The Coin homepage not only provides online documentation, but also a good assortment of forums and FAQs. TGS have enhanced the original Open Inventor version and also provide a considerable collection of information and FAQs. ■

INFO

[1] Stephan Siemen, Virtual World: Linux Magazine, Issue 28, p72

[2] Stephan Siemen, Moving Objects: Linux Magazine, Issue 29, p72

[3] Josie Wernecke, The Inventor Mentor. Release 2: Addison-Wesley 1994, ISBN 0-201-62495-8

[4] Summary of Inventor Mentor by SGI: *http://www.sgi.com/software/inventor/vrml/TIMSummary.html*

[5] Coin: *http://www.coin3d.org*

[6] Documentation for Coin libraries: *http://doc.coin3d.org*

[7] TGS: *http://www.tgs.com*

[8] Additional information: *http://prswww.essex.ac.uk/stephan/3D/*

[9] Files for the articles: *ftp://ftp.linux-magazin.de/pub/listings/magazin/2003/04/3d/*

THE AUTHOR

Dr. Stephan Siemen works as a scientist at the University of Essex (UK) where he is involved with creating software for 3D representation of weather systems and teaches computer graphics and programming. Additional information on this subject is available from his website: http://prswww.essex.ac.uk/stephan/3D/.