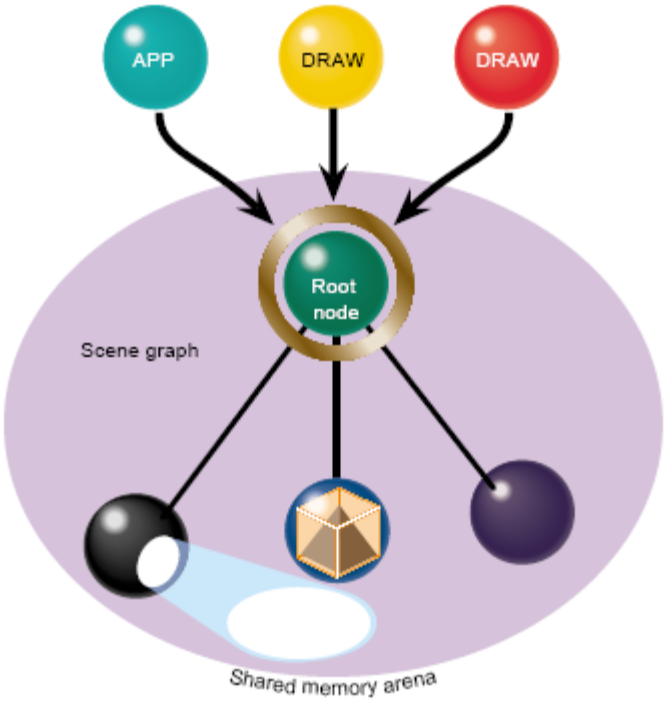
SCENEGRAPH PROGRAMMING WITH OPENGL PERFORMER

A LAB CREATED BY TOMMY TROEN

Graphics created by:
A.J. Diamond, Donald Schmitt and Company.



ABSTRACT

This lab is an introductory lab in OpenGL Performer and its concepts. In part 1 of the lab you will create your own sample application with all the basic parts that must be present in an OpenGL Performer application. This application can be used a skeleton for future work. In part 2 of the lab you will get an introduction to the OpenGL Performer scene graph and create an application where you build a scene graph from scratch.

You do not need to be a graphics-guru to complete this lab but it will help the overall experience if you are familiar with a few general graphics concepts. The lab uses the C++ programming language for the examples but it does not require that you are an expert in this either.

INTRODUCTION

OpenGL Performer is an application development environment, which provides a programming interface (with ANSI C and C++ bindings) for creating real-time graphics applications with emphasis on performance. It is built atop the OpenGL graphics library and is available for the IRIX® operating system, 32-bit and 64-bit Linux®, Windows® XP and Windows® 2000. Typical OpenGL Performer applications are in the field of virtual reality, visual simulations, interactive entertainment and computer-aided design. OpenGL Performer aims to maximize the graphics performance of any application.

Fig 1. Simulated Hotel Lobby, created by Design Vision Inc, Toronto
(This image is located in the Getting Started Guide by OpenGL Performer)

Many graphics applications, such as Virtual Reality applications, consist of very complex scenes and interactivity. The scenes may have many related and connected objects, which is not easily managed with OpenGL alone. OpenGL Performer makes use of scene graphs to overcome this.

A scene graph provides high performance rendering for less effort and eases the management of interaction and management of large numbers of objects in a scene.

So what is a scene graph? It is a data structure that stores all information about the virtual world. This includes all objects and behaviours in a scene and all other entities that affect the objects such as lights and sounds. The scene graph specifies a hierarchical grouping of 3D objects and object parts using a graph (DAG, Directed Acyclic Graph).

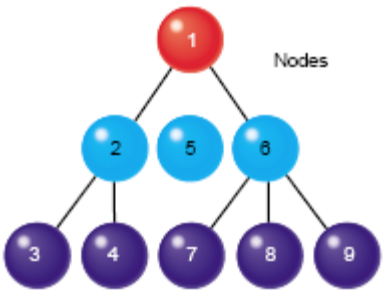


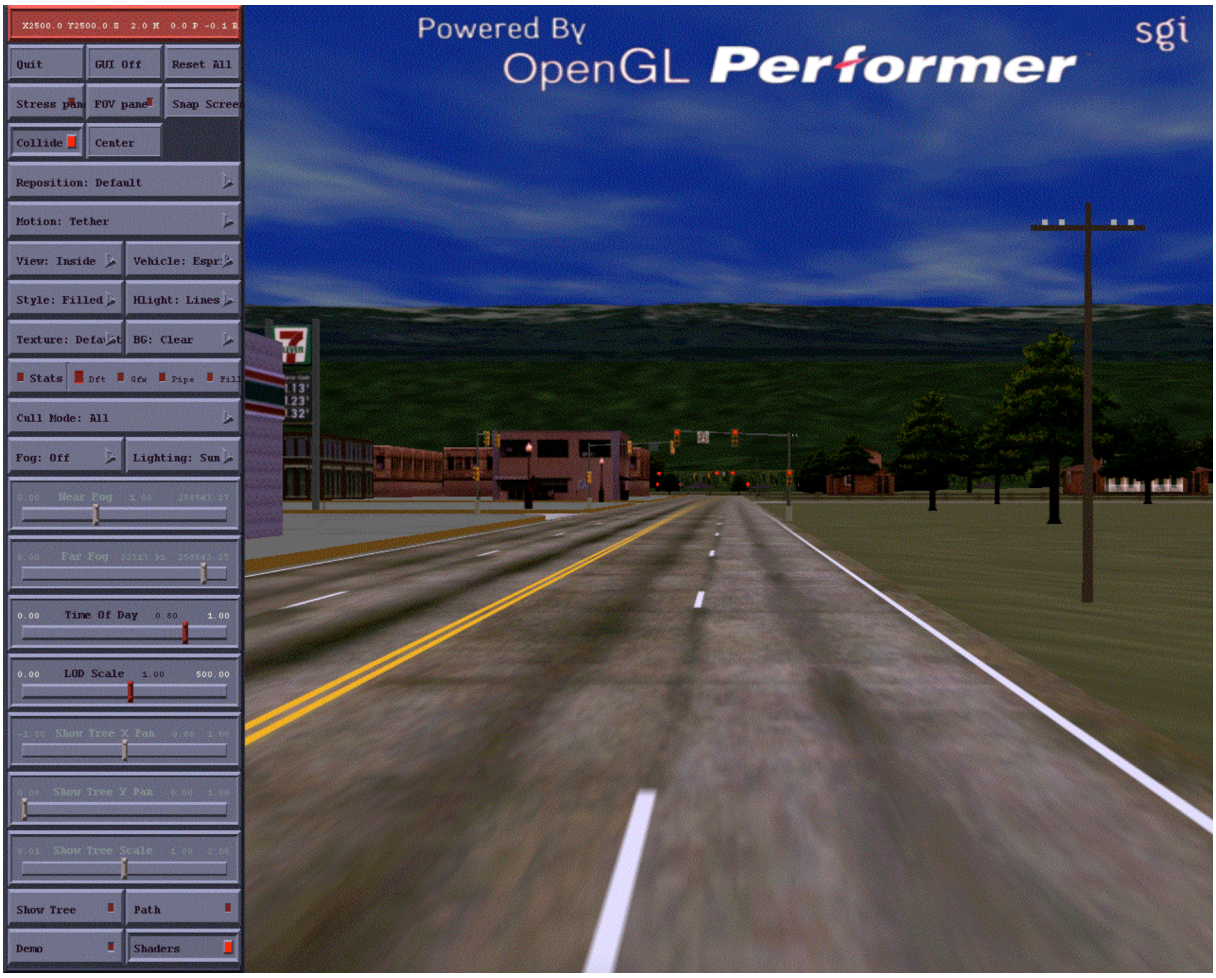
Figure 2: Structure of a scene graph, source: OpenGL Performer Getting Started Guide

The scene graph may be created in the application or loaded in from disk. The latter is the most common approach, where the scene graph is created in a modelling package such as Maya or 3DS MAX. The scene graph can then be interactively modified during rendering (program execution).

Try it out!

OpenGL Performer provides a demo application, Perfly, where you can view sample scenes. This is a good place to start getting to know OpenGL Performer and its functionality.

Go to the start menu. Click Programs > SGI OpenGL Performer > Performer town. The demo application will start and give you a tour of a town from the perspective of a car. (The graphics cards in the pc's in the lab are inadequate to the requirements of OpenGL Performer, so the application may behave a bit strange.)



You can play about with scene using the buttons to your left. To exit press 'Quit'.

PART 1 – Create a basic application

In this section you will be introduced to the basic concepts of a Performer application and build your own skeleton application which you can use as a basis for future applications.

How does the scene you have created (in the application or in a modelling package) end up on the computer screen?

As mentioned in the introduction, OpenGL Performer uses a scene graphs to describe the virtual worlds in an application. The scene graph encapsulates all of the visual data in a scene. A view into a virtual world is described by a channel, which is equivalent to a camera moving throughout the scene. The channel shows a slice of the scene from a specified perspective and only contains the visual information visible to the viewer. The visual data from the channel is rendered by a software pipeline into a window on the computer screen. (The pipe is the software abstraction of the hardware graphics pipeline) See the path from scene to screen visually described in Figure 2:

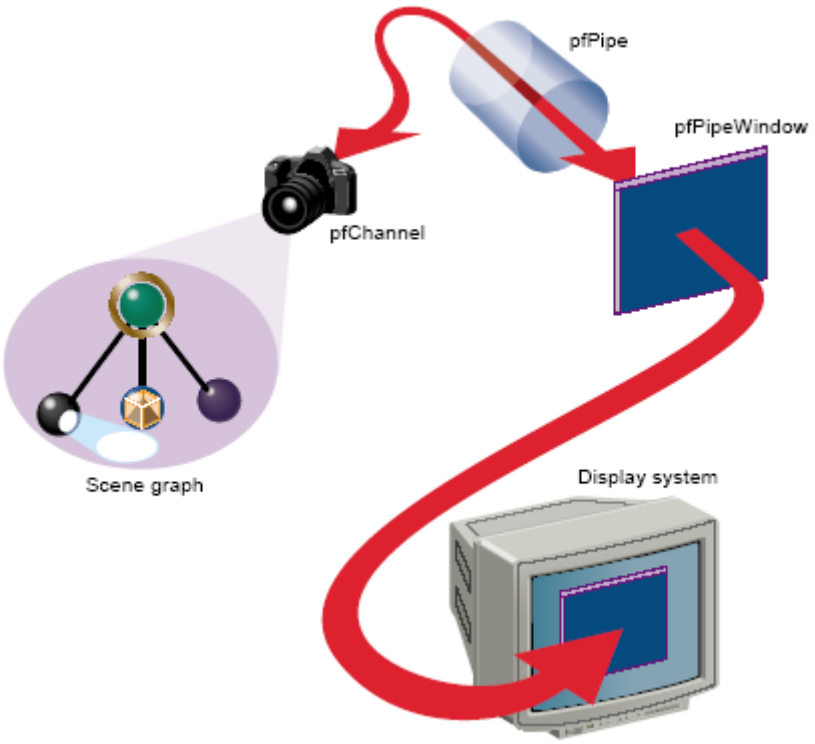


Figure 3: Data to display. (The figure is originally from the OpenGL Performer Getting Started guide.)

.

Basic parts of a Performer application

These parts should be present in every Performer application:

1. Initialize OpenGL Performer
2. Acquire a pipe.
3. Create a channel and a window.
4. Associate the channel with a window
5. Load or create a scene graph and associate it with the channel
6. Position the channel and update the scene
7. Call pfFrame to draw the scene
8. Create the simulation loop to return to step 6.

Start programming!

This application will load in model from a file and rotate the model for a specified time.

Download the files needed for this tutorial from:
http://www.macs.hw.ac.uk/~tt16/Performer/OpenGL_Performer_Lab.zip

NOTE: The files must be unzipped to the D:/ disk of your computer.
This is because .NET runtime doesn't fully trust network share locations, e.g. your home directory. If you put the files in your home directory, the code will not execute as fully trusted and you may receive unexpected security exceptions.

Locate the Part1 folder and double-click on the Part1.sln file. This will open Visual Studio .NET. You may have to set some paths and dependencies for the project, so have a look at the readme.txt file in the same folder.

If the file basic.cxx isn't open, open it from the solution panel to your right in Visual Studio (under the folder 'Source').

All code in the steps listed below must be inserted in the 'main' method.
Now let's get started! Good Luck!

1. Start of by declaring the variables for the required elements in the application's main method.

```
pfScene *scene;     //the root of the scenegraph

pfNode *root;       //the node in which we will place the loaded model

pfPipe *p;          //Graphics pipeline to perform the rendering

pfPipeWindow *pw;   //the graphics window

pfChannel *chan;    //the view which is rendered on the pipe

pfSphere bsphere;   //a sphere which we will use to determine the
                    //extent of the scene's geometry

char *filename = "teapot.nff";//name of file to be loaded in
```

2. Now you have to initialise and configure OpenGL Performer.

```
pfInit();
// Use default multiprocessing mode based on number of processors.
pfMultiprocess(PFMP_DEFAULT);

// Load all loader DLL's before pfConfig().
// This method determines what loader to use by the format of the
// file extension. The actual loading of the file will take place
// later. It takes a filename as a parameter.
pfdInitConverter(filename);

// Configure multiprocessing mode and start parallel
// processes.
pfConfig();
```

3. With all this in place it is time to load in the actual model from a file, this model is then placed in the node we declared in step 1:

```
// Read a single file, of any known type.
 root = pfdLoadFile(filename);
 if (root == NULL)
 {
   pfExit();
   exit(-1);
 }
```

pfdLoadFile() loads a scenegraph structure/graphics model from disk and constructs a graph from the data. It supports a number of different file types.

In part 2 you will create the graph structure that is added to the root node yourself.

4. Create the root node of the scene graph and add the root node of the model to the scene. The loaded file is placed in the node 'root'.

```
// Attach loaded file to a pfScene.
 scene = new pfScene;
 scene->addChild(root);
```

5. Add lighting to your scene:

```
// Create a pfLightSource and attach it to scene.
 scene->addChild(new pfLightSource);
```

6. Set the extent of the scene's geometry with the sphere created in step 1. This is used later to determine what is viewed in the channel.

```
// determine extent of scene's geometry
scene->getBound (&bsphere);
```

7. Acquire the pipe and associate it with a new window. The pipe renders the data received from the channel (the camera of the scene) into a window.

```
 // Configure and open GL window
 p = pfGetPipe(0);
 pw = new pfPipeWindow(p);
 pw->setWinType(PFPWIN_TYPE_X);
 pw->setName("My basic application");
 pw->setOriginSize(0,0,500,500);
 pw->open();
```

8. Create the channel and associate with the channel, p, and the root node of the scene graph, scene. Set up the viewing and draw the channel.

```
 // Create and configure a pfChannel.
 chan = new pfChannel(p);
 chan->setScene(scene);
 chan->setNearFar(1.0f, 10.0f * bsphere.radius);
 chan->setFOV(45.0f, 0.0f);
 chan->setTravFunc(PFTRAV_DRAW, DrawChannel);
```

9. Finally, in the simulation loop set the new computed viewing position. The simulation loop drives the application and repeats endlessly until the application exits. In our case after 60 seconds.

```
    // Simulate for 60 seconds.
  while (t < 60.0f)
  {
    float      s, c;
    pfCoord        view;

    // Go to sleep until next frame time.
    pfSync();

    // Compute new view position.
    t = pfGetTime();
    pfSinCos(45.0f*t, &s, &c);
    view.hpr.set(45.0f*t, -10.0f, 0);
    view.xyz.set(2.0f * bsphere.radius * s,
                 -2.0f * bsphere.radius *c,
                 0.5f * bsphere.radius);

    chan->setView(view.xyz, view.hpr);

    // Initiate cull/draw for this frame.
    pfFrame();
  }

  // Terminate parallel processes and exit.
  pfExit();

     return 0;
```

The simulation loop has generally three steps: Update the scene, e.g location of an object, Update the camera, e.g. zoom in on scene, redraw the frame.

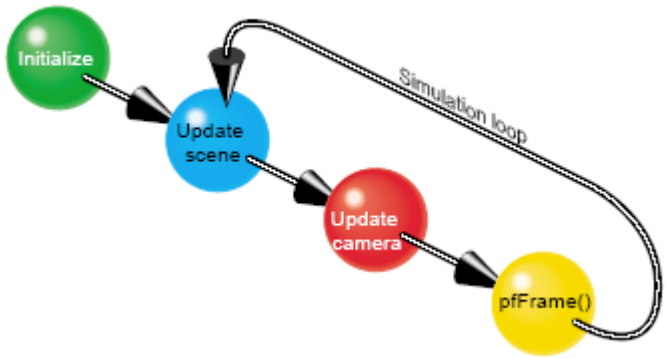


Figure 4: Simulation loop, source: OpenGL Performer Getting Started Guide

Part 2 – Build a scene graph

In part 1, we loaded in the structure of the scene graph from a file. This is the most common approach when creating a graphics application. However there are times where parts of the scene graph have to be build up inside the application code or you want to apply changes to the scene graph at runtime. This part will introduce the basic concepts about the OpenGL Performer scene graph.

The OpenGL Performer scene graph

A scene graph is a hierarchical structure that consists of many different node types. Each node type contains different data such as the geometric data and appearance of an object or the transformation to orient or position an object in the scene. The node types in an OpenGL Performer scene graph can be divided into two general classifications:

- Group nodes: can take an arbitrary number of child nodes, associate nodes into hierarchies

- Leaf nodes: contain all the descriptive data of objects in the virtual world. Cannot have child nodes.

The root of an OpenGL Performer scene graph is the pfScene node.

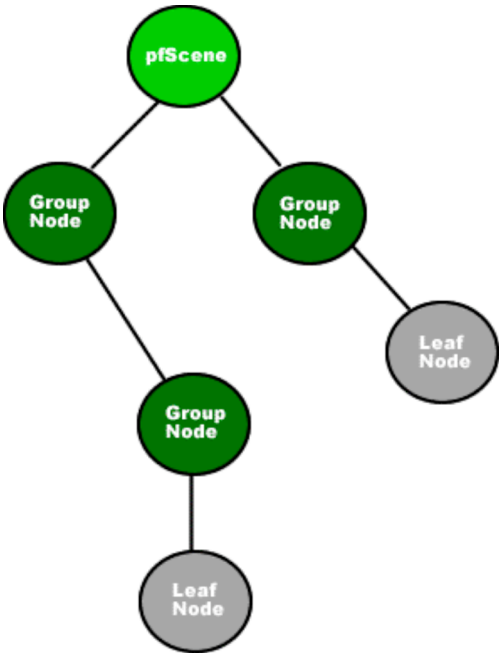


Figure 5: A sample scene graph

All nodes in an OpenGL Performer scene graph are used to describe a virtual world. This means that objects associated with the view of the virtual world, such as pfChannel, are not included in the scene graph. Some scene graphs include the view of a scene such as in a Java3D scene graph where the view is actually a part of the virtual universe.

Geometry and transformations
When creating geometry you specify its size, orientation and location in its own space using the local coordinate system. The geometry can be placed in relationship to other shapes in the same scene or into the coordinate system of the root node known as the world space. If you wish to reposition, reorient or rescale a shape in the scene graph you will need to use a transformation node. A transformation node is a group node and applies the transformation to all nodes in the branch directly below the transformation node. Transformations are not carried over from one branch to another.

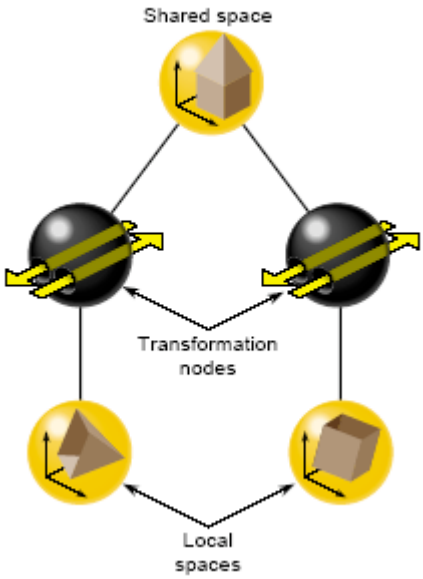


Figure 6: Transformation nodes, source OpenGL Performer Getting Started Guide

OpenGL Performer provides three different transformation nodes:

- pfFCS – used for dynamically transforming shapes to create movement
- pfDCS – used to transform shapes when the transformations might change over time.
- pfSCS – used for transformation values that do not change once they are set.

Each transformation node provides methods to scale, rotate and translate (move) its child nodes. Multiple transformations can be applied and are applied in the following order: scale, rotate and translate.

We will be using the pfSCS node when creating our sample program.

Start programming!

We will create a model that will consist of two spheres and a cube. The scene graph for this application will look like the one in figure 7.

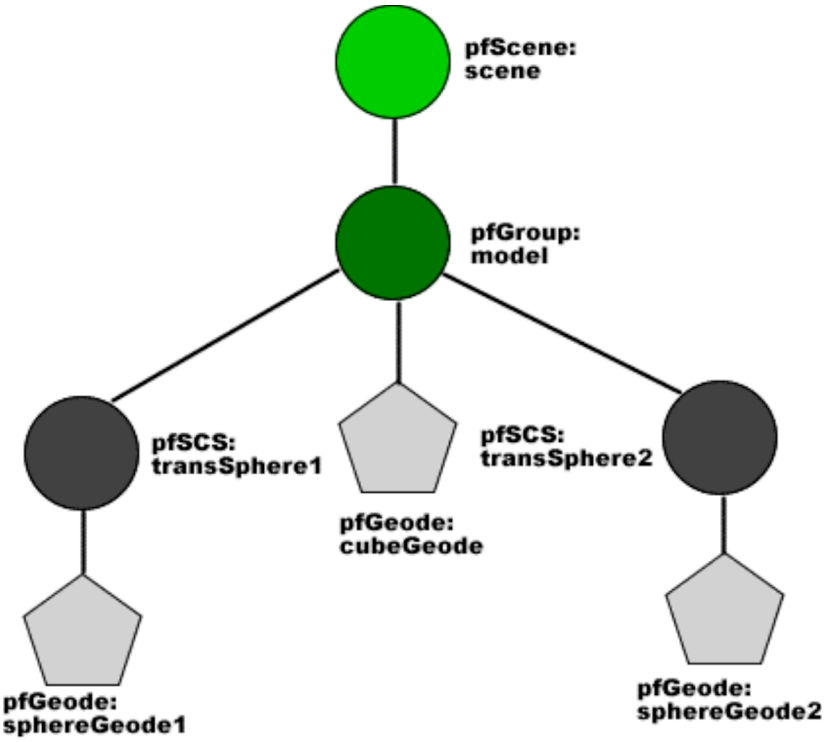


Figure 7: Our scene graph

Locate the Part2 folder in the OpenGL Performer directory you unzipped in part 1 and double-click on the Part2.sln file. This will open Visual Studio .NET. All paths should now have been properly set.

If the file scenegraph.cxx isn't open, open it from the solution panel to your right in Visual Studio (under the folder 'Source').

All code required to get this application to function properly is listed on the next page.

Insert all of the code below the call to `pfConfig()` in the application's main method.

1. Create the shapes and place them in a pfGeode node.

```
//Create the shapes and place them in geometry nodes
pfVec4 *color1 = new pfVec4(0.0f, 0.5f, 0.5f, 1.0f);
pfGeode *sphereGeode1 = createSphere(1.5f,*color1,150);

pfVec4 *color2 = new pfVec4(1.0f, 0.2f, 0.5f, 1.0f);
pfGeode *sphereGeode2 = createSphere(1.5f,*color2,150);

pfVec4 *color3 = new pfVec4(0.5f, 0.5f, 0.5f, 1.0f);
pfGeode *cubeGeode = createCube(3.0f,1.0f,1.0f,*color3);
```

Geometry is created in a pfGeoSet which is a collection of one or more primitives, such as lines and triangles. To place this geometry in the scene graph you must attach the pfGeoSet to a pfGeode. A geode is a leaf node and is used to encapsulate geometry in the scene graph. The geode can have multiple pfGeoSet attached.

The methods createSphere() and createCube() are custom created help methods for this lab. The parameters for createSphere() are: radius, colour and number of triangles used to draw the shape (many triangles means a smoother shape).
The parameters for createCube() are: width(x), height(y), depth(z) and colour.
If you wish to learn more about how to create geometry, read the programming guide which is supplied with OpenGL Performer.

2. Create matrices which can be used to translate (move) the shapes. The macro `PFMAKE_TRANS_MAT(mat1,2.0f,0.0f,0.0f)` multiplies a vector into a matrix. The vector contains the new coordinates in the form: x,y,z. Associate the matrix with the pfSCS transformation node. Finally add the geodes containing the shapes to the transformation nodes.

```
pfMatrix mat1;
PFMAKE_TRANS_MAT(mat1,2.0f,0.0f,0.0f);
pfSCS* transSphere1 = new pfSCS(mat1);
transSphere1->addChild(sphereGeode1);

pfMatrix mat2;
PFMAKE_TRANS_MAT(mat2,-2.0f,0.0f,0.0f);
pfSCS* transSphere2 = new pfSCS(mat2);
transSphere2->addChild(sphereGeode2);
```

The SCS node has a static coordination system so the transformation values cannot be changed once it is created. The reason we are using the SCS node in this example is that we will only be positioning the shapes before the rendering of the scene.

3. Create a group node and add the two transformation nodes and the geode to it.

```
//Build hierarchy
pfGroup* model = new pfGroup;
model->addChild(transSphere1);
model->addChild(cubeGeode);
model->addChild(transSphere2);
```

We could have added the three nodes directly to the root of the scene graph, but by putting in the group node before the 3 nodes we make the program easier to maintain, e.g. if we wanted a another model attached to the scengraph we could divide them into branches.

4. Attach the group node to the root of the scene graph. We will also add some lighting to the scene.

```
scene = new pfScene;
scene->addChild(model);
// Create a pfLightSource and attach it to scene.
scene->addChild(new pfLightSource);
// determine extent of scene's geometry
scene->getBound (&bsphere);
```

5. Now compile and run the application and see the results of your work!

Extra exercise for those interested

Try to create another sphere or cube and position it in the scene graph. You could also try to translate or rotate the whole model (the 2 spheres and the cube) by only using one transformation.

Summary

This lab has only shown you a fraction of what you can do with OpenGL Performer. Next term we will be looking into slightly more complex applications and try to link it all together with Virtual Reality.

If you are interested in learning more about OpenGL Performer you can download a free demo version from:
http://www.sgi.com/products/software/performer/

This web-site also provides a lot of information about OpenGL Performer, including guides, mailing lists and resources.

The two guides that comes with OpenGL Performer are very extensive and great if you would like to learn more about the libraries available and how to develop applications using them.

You can find them on the SGI website or in the start up menu under SGI OpenGL Performer:
Getting Started Guide.pdf
Programming Guide.pdf