An Introduction to OpenSceneGraph

Norman Murray

© *Review date September 28, 2012*

Contents

Chapter 1

An Introduction to OpenSceneGraph

1.1 Introduction

OpenSceneGraph is an open source, cross platform graphics toolkit for the development of high performance graphics applications such as flight simulators, games, virtual reality and scientific visualisation. Based around the concept of a scene graph, it provides an object oriented framework on top of OpenGL freeing the developer from implementing and optimising low level graphics calls, and provides many additional utilities for rapid development of graphics applications.

1.1.1 Features

With OpenSceneGraph the goal is to make the benefits of scene graph technology freely available to all, both commercial and non-commercial users. Written entirely in Standard C++ and OpenGL, it makes full use of the Standard Template Library (STL) and Design Patterns, and leverages the open-source development model to provide a development library that is legacy-free and focused on the needs of end users. Some of the features of OpenSceneGraph include:

Performance: Supports view-frustum culling, occlusion culling, small feature culling, Level Of Detail (LOD) nodes, OpenGL state sorting, vertex arrays, vertex buffer objects, OpenGL Shader Language and display lists as part of the core scene graph. These together make the OpenSceneGraph one of the highest performance graphics toolkit available.

Productivity: The core scene graph encapsulates the majority of OpenGL functionality including the latest extensions, provides rendering optimizations such as culling and sorting, and a whole set of add-on libraries which make it possible to develop high-performance graphics applications very rapidly. The application developer is freed to concentrate on content and how that content is controlled rather than low level coding.

Database loaders: For reading and writing databases, the database library (osgDB) adds support for a wide variety of database formats via an extensible dynamic plugin mechanism - the distribution now includes 55 separate plugins for loading various 3D database and image formats.

Portability: The core scene graph has been designed to have minimal dependency on any specific platform, requiring little more than Standard C++ and OpenGL. This has allowed the scene graph to be rapidly ported to a wide range of platforms—originally developed on IRIX, then ported to Linux, then to Windows, then FreeBSD, Mac OSX, Solaris, HP-UX, AIX and even PlayStation2!

You will be using OpenSceneGraph for this component of the module and also for Rob Aspin's component on Visual Simulation. More information on OpenSceneGraph can be found at its website `http://www.openscenegraph.org/`.

A scene graph, is a structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene. A scene graph is a collection of nodes in a graph or tree structure. A node may have many children but often only a single parent. Nodes in a scene graph (generally) represent entities or objects in the scene. For example, in Figure 1.1, we can see a human who is represented by a node for their torso, linked to this are two nodes to represent their legs, and two nodes for the feet. You will learn about creating complex scene graphs such as an articulated robot in the Visual Simulation course.

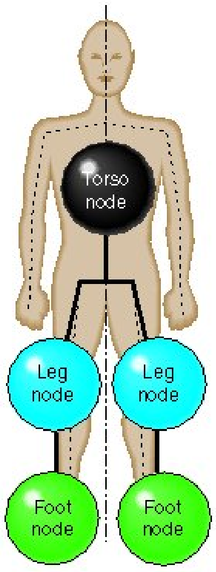


Figure 1.1: A simple scenegraph with linked nodes

1.2 Building a Simple Viewer

OpenSceneGraph is a vast software library and we have no way of learning how to use it all within the teaching of this course. We will be coding in C++ and using Visual Studio 2010. The machines you use have Visual Studio 2010 and you are free to download Visual Studio 2010 to your own computers from the DreamSpark website at `www.dreamspark.com`

I will supply you with an empty Visual Studio solution/project to which you can add your code. If you double click on a *.sln* file, Visual Studio 2010 will open, see Figure 1.2.

We will begin by creating a simple viewer, a window in to our 3D world and adding some objects to it, and examine how we can move around the world. Begin by adding the following code to the currently empty raaViewer.cpp file:

```
// A simple program to view a model

// the following define the functions that are needed by our program
// #include is similar to import in Java
#include "stdafx.h"
#include <iostream>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>

// this is similar to main(String arg[]) in Java
int main(int argc, char** argv)
{
  // create a node to store our 3D model(s)
  osg::Node *g_pModel=0;
```

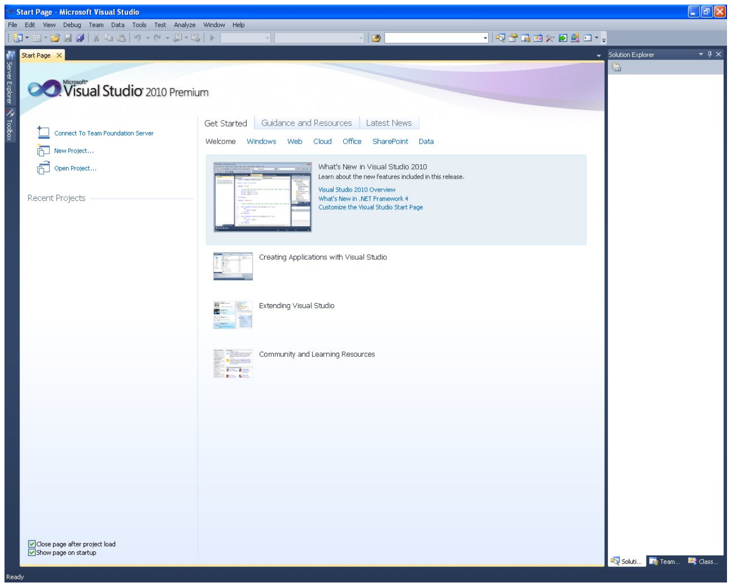

Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Analyze Window Help
Start Page
Microsoft Visual Studio 2010 Premium
Connect To Team Foundation Server
New Project...
Open Project...
Recent Projects
Get Started
Guidance and Resources
Latest News
Welcome Windows Web Cloud Office SharePoint Data
What's New in Visual Studio 2010
Learn about the new features included in this release.
Visual Studio 2010 Overview
What's New in .NET Framework 4
Customize the Visual Studio Start Page
Creating Applications with Visual Studio
Extending Visual Studio
Community and Learning Resources
Close page after project load
Show page on startup
Solution Explorer
Ready

Figure 1.2: Microsoft Visual Studio 2010

```
  // load a model giving its location relative to the running program
  g_pModel = osgDB::readNodeFile("../../Data/robot.osg");

  // if we can't load the model then quit
  if(!g_pModel){
    std::cout << "Model not found" << std::endl;
    return 0;
  }

  // create a viewer - a window on to our 3D world
  osgViewer::Viewer viewer;

  // set the scene to render (the model(s) we have just loaded)
  viewer.setSceneData(g_pModel);

  // set up windows
  viewer.realize();

  // starts the viewer
  return viewer.run();

  return 0;
}
```

As shown in the above program, the simplest program using osgViewer loads a model and creates an instance of the viewer. Pay particular reference to the comments included within the program that explain what is happening on each line of code.

1.2.1 Compiling your program

Once you have typed in the source code, you need to turn it into machine code that the computer can execute/run.

To compile your world you can select Build→Build Solution or press F7.

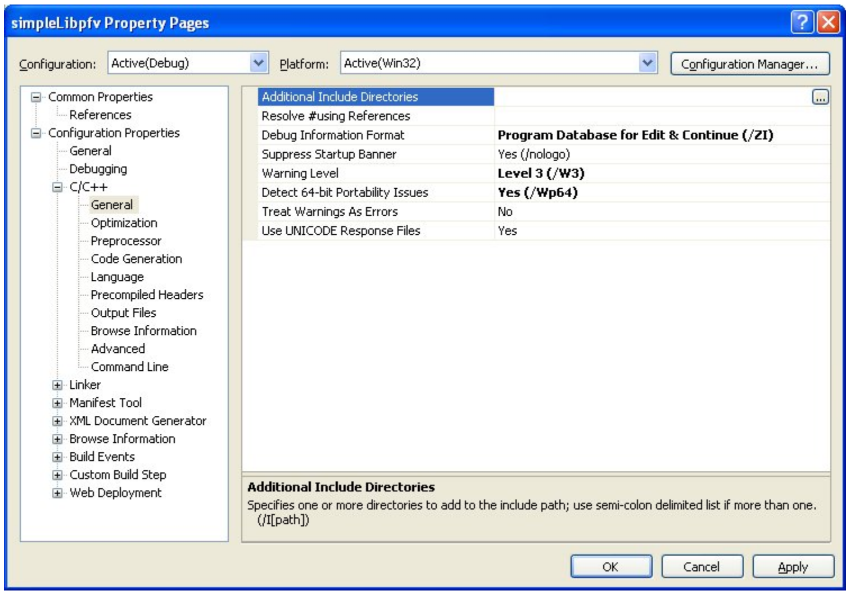
The project will not compile. This is because the project does not know where to look for the files in the `#include` lines of your program code. These include files are like `import` statement in Java. They define the methods that are available in the OpenSceneGraph libraries.

To specify where to look for the `#include` files:

- Select Project→Properties...
- Select Configuration Properties→C/C++→General (see Figure 1.3)
- Click on the Additonal Include Directories box and enter the following directories:

  ../OpenSceneGraph-vs2010/include

After specifying the include directories you can attempt to compile the program again.



Configuration: Active(Debug) Platform: Active(Win32)

Additional Include Directories	
Resolve #using References	
Debug Information Format	**Program Database for Edit & Continue (/ZI)**
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	**Yes (/Wp64)**
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

Figure 1.3: Where to specify the include directories

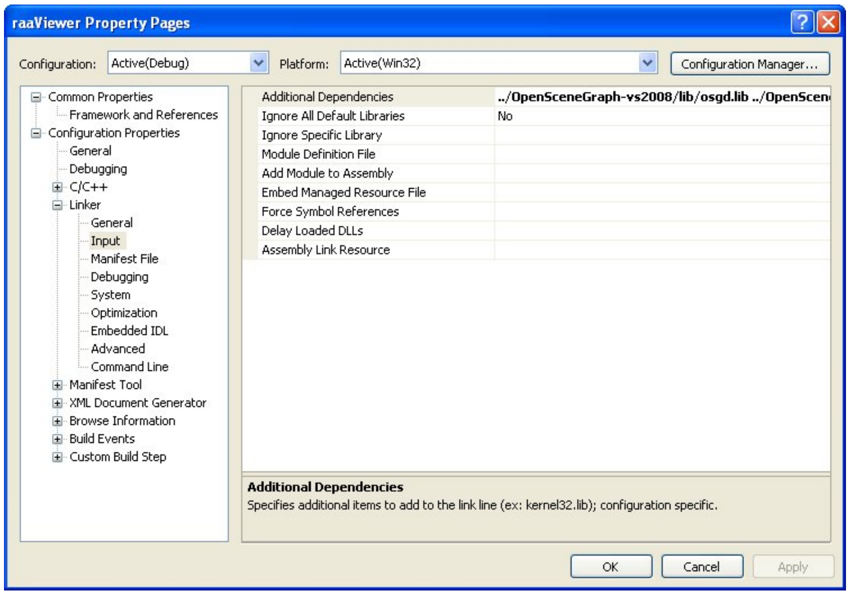
1.2.2 Linking your program

If you look at your errors, you should find that your program has compiled, but it has not linked. Your program needs to link with the OpenSceneGraph libraries that contain the machine code of the OpenSceneGraph functions that you have used.

We need to tell Visual Studio where the libraries are and what the libraries are called, see Figure 1.4:

- Select Project→Properties...
- Select Configuration Properties→Linker→Input
- Click on Additional Dependencies and add:

  ../OpenSceneGraph-vs2010/lib/osgd.lib
  ../OpenSceneGraph-vs2010/lib/osgViewerd.lib
  ../OpenSceneGraph-vs2010/lib/osgDBd.lib
  ../OpenSceneGraph-vs2010/lib/osgUtild.lib
  ../OpenSceneGraph-vs2010/lib/osgGAd.lib



Additional Dependencies	../OpenSceneGraph-vs2008/lib/osgd.lib ../OpenScen
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

Figure 1.4: Where to specify the libraries

1.2.3 Running your program

You can try to compile and run your program again (F5). Your program runs but then stops very quickly and you may see that it outputs `Model not found` to the console window before it closes. To stop the console window from disappearing so quickly, you can press Ctrl F5 to start debugging the program. The program is expecting a filename as input to the program to specify what model to display in our world. To specify command line arguments within Visual Studio we:

- Select Project→Properties...
- Select Configuration Properties→Debugging
- Set the Working Directory box to be: ../OpenSceneGraph-vs2010/bin
- Select Linker→General
- Set output file $(OutDir)$(ProjectName).exe

You can now run your program and you should see as output Figure 1.5.

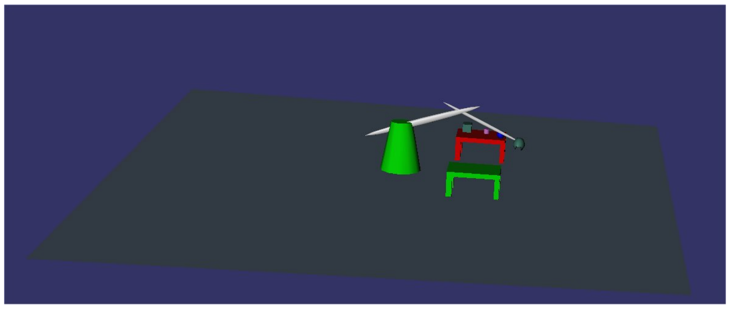


Figure 1.5: Your running program

Now we need to investigate this program and what it allows us to do. We can see a robot (you will learn how to animate the robot in Rob Aspin's Visual Simulation course). You will notice that by using your mouse, you can manipulate the scene:

- using the left mouse button and mouse movement you can rotate around world
- using the middle mouse button and mouse movement you can move the world to the left and right, and up and down
- using the right mouse button and mouse movement you can zoom in and out
- space resets your view of the world

This is called a Trackball manipulator, but where did it come from, as we did not specify one. If you go to Visual Studio and right click on `run` on the following line of code: `return viewer.run();` and select Go To Definition, you will be taken to file where the `run` method is defined. In the comment above the method definition you will see that the `run` method that we call adds a trackball manipulator to our world if there is no manipulator in the world. We are also positioned in the world where we can see the objects that have been placed in the world and we can also move the camera about the world.

1.2.4 Some modifications

You can find more models to view and links to websites to obtain models at: `http://www.openscenegraph.org/projects/osg/wiki/Downloads/SampleDatasets`. You can also find some models on Blackboard. Downloads some models to your Data directory and alter the Projects command arguments to view different models from your Data directory. You can add multiple models as command line arguments and observe what happens.

1.3 Some Other Camera Manipulators

Our world exists, and we can alter our view of the world using a trackball manipulator. We shall now look at adding some different types of manipulator to our world. Modify your program by adding the three `#include` lines at the top of the program and their use, just after the creation of the viewer: **new lines are shown in larger font size:**

```
// A simple program to view a model

// the following define the functions that are needed by our program
// #include is similar to import in Java
#include "stdafx.h"
#include <iostream>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>

// new camera manipulators
#include <osgGA/FlightManipulator>
#include <osgGA/DriveManipulator>
#include <osgGA/KeySwitchMatrixManipulator>
#include <osgGA/TrackballManipulator>

// this is similar to main(String arg[]) in Java
int main(int argc, char** argv)
{
  // create a node
  osg::Node *g_pModel=0;

  // load a model giving its location relative to the running program
  g_pModel = osgDB::readNodeFile("../../Data/robot.osg");

  // if we can’t load the model then quit
  if(!g_pModel){
    std::cout << "Model not found" << std::endl;
    return 0;
  }

  // create a viewer
  osgViewer::Viewer viewer;

  // adding three camera manipulators controlled by keys 1 to 3
  osgGA::KeySwitchMatrixManipulator* pKeyswitchManipulator =
    new osgGA::KeySwitchMatrixManipulator();
  pKeyswitchManipulator->addMatrixManipulator( '1', "Trackball",
    new osgGA::TrackballManipulator() );
  pKeyswitchManipulator->addMatrixManipulator( '2', "Flight",
    new osgGA::FlightManipulator() );
  pKeyswitchManipulator->addMatrixManipulator( '3', "Drive",
    new osgGA::DriveManipulator() );
  viewer.setCameraManipulator(pKeyswitchManipulator);

  // set the scene to render (the model we have just loaded)
  viewer.setSceneData(g_pModel);

  // set up windows
  viewer.realize();

  // starts the viewer
  return viewer.run();

  return 0;
}
```

We have added two new camera manipulators. These are call flight and drive. We have also added a `KeySwitchMatrixManipulator`. This allows us to switch between different camera manipulators. By pressing keys 1, 2, or 3 we can switch between the three types of camera manipulator that we have added to our world. These allow you to move the camera through the world, rather than manipulating the view of the whole world. For each of these viewers:

- the left mouse button accelerates,
- the right mouse button decelerates,
- and the middle mouse button (or left and right simultaneously) stops dead.
- space key resets your position in the world

Practice using each of these camera manipulators and note any problems that you have with them. Could you develop a better camera manipulator?

1.4 Summary

So far we have looked at creating a simple 3D world by loading models into that world. We have also looked at adding different types of camera manipulator to the world to view it. Topics that we will cover next will include positioning objects in the world and moving them. We will also look at creating our own camera manipulators.

Chapter 2

Windows, Transforms and Event Handlers

2.1 Introduction

This practical will extend on what was learnt in the initial practical where we created a simple world and examined different options for moving through the 3D world. In this practical we will look at creating a resizeable window, so that the OpenSceneGraph application does not dominate our screen. Then we will look at positioning objects in the scene, creating a basic scene graph so that our world contains more than one object, moving objects in the world and finally using the keyboard to control the movement of the objects in our 3D world.

2.2 Resizeable Window

Currently, when your program runs, it takes over the full screen. It may be necessary to be able to see other applications that you have running at the same time as our 3D world. To do this we can set the size and position of our window. To do this we need to set some of the options or *traits* of the window. These allow us to set the windows *x* and *y* position, to position the window on our screen. We can also set the *width* and *height* of the window. We also set the *windowDecoration* to *true*. This sets a border around the window and buttons to minimise and resize the window. We also set *doubleBuffer* to *true*. Finally, we apply these traits or properties to our graphics context and set the camera's viewport to have the size of our window, so when we draw/render the scene it fits in our window. Modify your program by adding the following lines to your program after the first line shown below that is already in your program: **new lines are shown in larger font size:**

```
// setup viewer
osgViewer::Viewer viewer;
```

```
osg::GraphicsContext::Traits *pTraits = new osg::GraphicsContext::Traits();
pTraits->x = 20;
pTraits->y = 20;
pTraits->width = 600;
pTraits->height = 480;
pTraits->windowDecoration = true;
pTraits->doubleBuffer = true;
pTraits->sharedContext = 0;

osg::GraphicsContext *pGC = osg::GraphicsContext::createGraphicsContext(pTraits);
osg::Camera *pCamera = viewer.getCamera();
pCamera->setGraphicsContext(pGC);
pCamera->setViewport(new osg::Viewport(0,0, pTraits->width, pTraits->height));
```

2.2.1 Tutorial

Now alter the code above by attempting the following:

- Modify the x and y values of pTraits, to reposition your screen at start up. What happens if you set the values outside the screen size?
- Modify the width and height values to obtain a window size you are happy with.
- What happens when you set windowDecoration to false?
- What happens when you set doubleBuffer to false? Do you have any idea what double buffering is? Can you guess or try looking online?

2.3 Positioning Objects

In the last practical, we looked at adding models or objects to our world, and then moving through the world using trackball, flight or drive controls. One aspect we did not look at was being able to position objects in our world. When you position an object in the world it has a position. These are referred to as the x, y, and z values, this positions the object in three dimensions in our world. You can also rotate objects in the world, altering the direction that they are facing. A final component of a model that you can change in our 3D world (but not so easily in the real world) is that we can change the scale or size of an object, meaning we can make it larger or smaller.

To be able to position our object we need to store the values that will transform our object to have the correct position, orientation and size (transformation). First we need to add a `PositionAttitudeTransform` node to our scene. This will be used to store our model's transformation. Near the top of your program after the first line shown below, add the `#include` line and the declaration of a `PositionAttitudeTransform` that we will use to store our model's transformation:

```
osg::Node *g_pModel=0;

#include <osg/PositionAttitudeTransform>
osg::PositionAttitudeTransform *g_pModelTransform=new osg::PositionAttitudeTransform();
```

Now we need to build our scene graph. So that the `PositionAttitudeTransform` affects our model we need to create a simple graph. A scene graph is a collection of nodes in a graph or tree structure. A node may have many children but often only a single parent, with the effect of a parent applied to all its child nodes; an operation performed on a group automatically propagates (feeds down) its effect to all of its members. In many programs, associating a geometrical transformation matrix at each group level and concatenating such matrices together is an efficient and natural way to process such operations. A common feature, for instance, is the ability to group related shapes/objects into a compound object which can then be moved, transformed, selected, etc. as easily as a single object. For example, a car could have a body and attached to the body are four wheels. When we apply a transformation to the car's body, the wheels will move with the car. We can apply a rotation to the car wheels, that are children of the car body, with this rotation turning the car's wheels but having no effect on the car's body.

To create our scene graph, we will add the model as a child to the `PositionAttitudeTransform`. That means that any positional information we place in our `PositionAttitudeTransform` will affect any and all models attached to the `PositionAttitudeTransform`. To create our scene graph, we can simply add the model that we are loading in as a child of the transform—the extra line is shown below the line where we read in our model:

```
// load a model giving its location relative to the running program
g_pModel = osgDB::readNodeFile("../../Data/robot.osg");

g_pModelTransform->addChild(g_pModel);
```

This creates a simple graph with the transform at the top of the graph, with the model being a child, and so inheriting any positioning information from the parent node. You can see this graph in Figure 2.1.

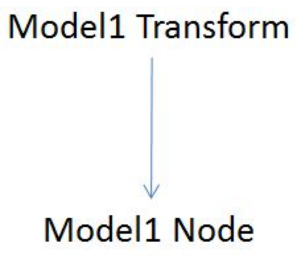


Figure 2.1: The basic scene graph that you have created

2.3.1 Rotating a Model

You can run the program, but will see no change—there are two reasons for this. First the model transform will contain an empty or identity matrix that has no effect on the position or orientation of the model. Immediately after the last line you entered above to create the scene graph add:

```
// set the rotation: here we create a roatation about each axis
// Note that we need to convert degrees to radians
g_pModelTransform->setAttitude(osg::Quat(
  osg::DegreesToRadians( 0.0f ),osg::Vec3(0,1,0),
  osg::DegreesToRadians( 0.0f ),osg::Vec3(1,0,0),
  osg::DegreesToRadians( 45.0f ),osg::Vec3(0,0,1)));
```

The above lines create a matrix and we set this matrix to contain a rotation of 45 degrees about the Z axis. What direction does the Z axis point. For an answer to this see Figure 2.2 that show the orientations of the axes in OpenSceneGraph.

The second change that is required is to make sure that we need to tell the `viewer` which node in the graph to begin drawing from. Currently the line:

```
viewer.setSceneData(g_pModel);
```

Tells the viewer to draw the model, so the parent `PositionAttitudeTransform` node is ignored. We can change the line to:

```
viewer.setSceneData(g_pModelTransform);
```

Now the viewer will begin drawing from the transform node and rotate the model correctly.

2.3.2 Tutorial

Now attempt the following:

- Alter the angles of the `setAttitude` command and see the effects of your changes and how they compare with Figure 2.2

2.3.3 Translating a Model

The above code allows us to rotate our model about a given axis. What about moving the model. To move the model we can use the `setPosition` command:

```
// set the position
g_pModelTransform->setPosition(osg::Vec3(0, 0, 20));
```

Run the program and see the changes. You should see that your object has moved up 20 units. Do you see a difference? If the answer is no then the question is why is there no difference in what you are seeing. The answer to that question is because we do not specify the initial starting point of the OpenSceneGraph camera. As we do not specify a position for our camera, OpenSceneGraph automatically positions the

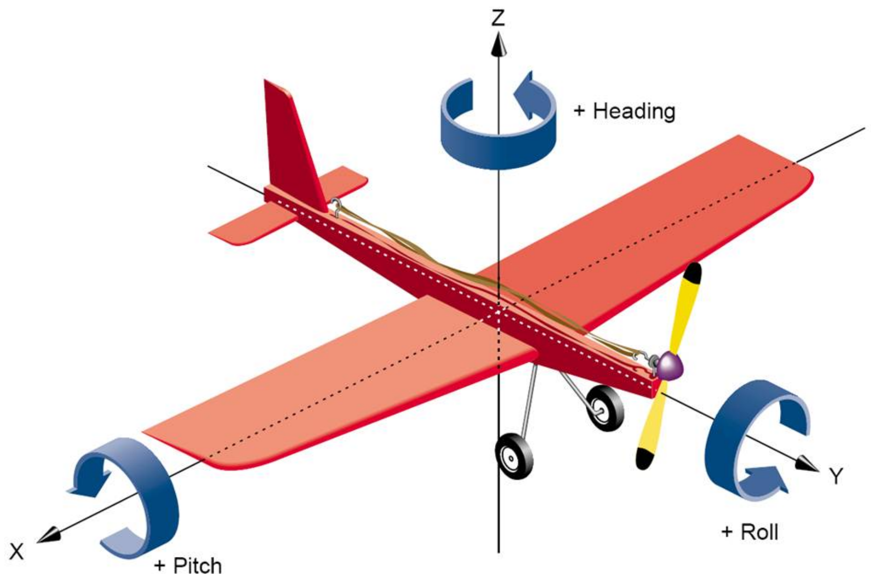


Figure 2.2: Axes in OpenSceneGraph

camera so that it can view the whole 3D world that we have created. Doing this automatically saves us from having to look around a large empty world for the objects we have added. It also means that as the model was moved 20 units up, the OpenSceneGraph camera was also moved up so that it could view the model.

2.3.4 Scaling Objects

You can also change the size or scale of your model with:

```
g_pModelTransform->setScale(osg::Vec3d(2.0f, 2.0f, 2.0f));
```

The above line will double the size of the plane. The `setScale` method takes in a vector that tells us how much we want to scale the model along the x, y, and z axes. Normally these three numbers are the same, but you can examine what happens when the numbers are different allowing you to stretch or squash the model in each of the axes.

2.4 Tutorial: Adding Another Model to the World

Now I want you to build a slightly more complicated scene graph with two models in the world. Look at Figure 2.3 and you will see that there are three transforms and two models. You will need to load two models and combine them with three transforms to create a scene. So you will need to add a world transform node. To this you can add two transform nodes as children. To these two transform nodes you can add different models. You will then be able to position the models by altering the individual transforms. So, you will have a world transform that allows you to move the entire world. In this world will be other transform nodes (remember each model will needs its own transform node so that you can alter the positions of the models in the world). So you will need to create a transform node for each model you add to the world and give them unique names, e.g. `g_pModel1Transform`, `g_pModel2Transform`, etc. If you get stuck at any point just ask for help. We can load mutliple models into a unqiue Nodes using the following:

```
g_pModel1 = osgDB::readNodeFile("../../Data/robot.osg");
g_pModel2 = osgDB::readNodeFile("../../Data/lz.osg");
```

Now try adding more models to the world, and positioning them in appropriate places.

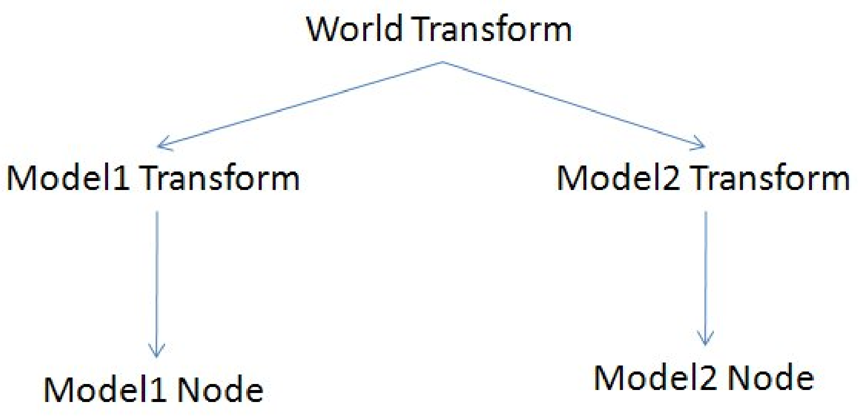


Figure 2.3: A scene graph for you to construct

2.5 Tutorial: Moving Objects

Hopefully, you have now created a simple scene graph with more than one object being displayed in your world. Previously we tried to translate an object, but we did not observe anything happening as the camera moved so we could see the object fully. We will now move one of the objects and observe how it moves in relation to the other object in our world. For my world I have loaded the `cessna.osg` model that represents a plane and the `lz.osg` that represents some ground in to my example. You can find more models to view and links to websites to obtain models at: `http://www.openscenegraph.org/projects/osg/wiki/Downloads/SampleDatasets`. Downloads some models to your Data directory. If you do this then you will see that the cessna plane appears below the ground. We can attempt to move it up using the line we used previously in Section 2.3.3. So you will need to create a transform and set the position using the `setPosition` method.

Once this is done you should now see that you can use `setPosition` to move objects in our world.

2.6 Using the Keyboard

Now that we have added multiple objects to the scene, we may want to control the objects in our world. Initially we will look at using the keyboard to move objects within the world. OpenSceneGraph provides a `GUIEventHandler` that we can extend to manage events that we are interested in. You can add the following class before your the `main` method.

```
osg::Vec3f g_pPlanePos;

class planeEventHandler : public osgGA::GUIEventHandler
{
public:
  virtual bool handle(const osgGA::GUIEventAdapter &ea,
    osgGA::GUIActionAdapter &aa, osg::Object *pObject,
    osg::NodeVisitor *pNodeVisitor)
  {
    switch(ea.getEventType())
    {
    case osgGA::GUIEventAdapter::KEYDOWN:
      {
        switch(ea.getKey())
        {
        case 'W':
        case 'w':
          {
            // move the plane 1 unit when ever 'w' is pressed
            g_pPlanePos.y() += 1.0f;
            g_pModelTransform->setPosition(g_pPlanePos);
            return true;
          }
        default:
          return false;
        }
    default:
      return false;
      }
    }
    return false;
  }
};
```

This creates a plane event handler. Within the code you will see a `switch` statement that checks to see if a key has been pressed. Within this is another `switch` statement that checks to see which key has been pressed. A global variable has been created at the top of the code that holds the position of the plane. You may want to initialise the `g_pPlanePos` vector to the position that you have moved to plane up to, to stop it jumping below the ground when you initially press the w key. Finally, before `viewer.realize();` in your `main` method add:

```
// add an event handler to manage some keyboard events
viewer.addEventHandler(new planeEventHandler());
```

This will create an instance of your `planeEventHandler` and add it to the viewer. Now whenever you press the W or w keys your plane should move.

2.6.1 Tutorial

Extend the `planeEventHandler` so that you can move the plane forwards and backwards, left and right and up and down. Also have a look at rotating the plane using the keyboard. Try printing the values of x, y and z and heading, pitch and roll to the console screen. What units are heading, pitch and roll stored as?

2.7 Summary

In this practical we have looked at creating a resizeable window, transforming objects in our world, creating a simple scene graph and transforming objects in the scene graph using keyboard input. We will extend this in later practicals by allowing user defined control of the camera and having the camera follow objects in the scene.

Chapter 3

Cameras and Matrix Manipulators

3.1 Introduction

In this practical we will look at positioning and moving the camera in our world. First we will create a camera follow manipulator. In this, the camera will follow an object that we control within the world. After this we will look at creating a walk camera manipulator that will simulate walking in the environment.

3.2 Adding a New Type of Camera Movement

Now that you have a plane under your control, it would be good if we could make the camera follow the plane. To do this we are going to create a `MatrixManipulator`. Add the following code:

```
class FollowManipulator: public osgGA::MatrixManipulator {

  class UnknownTransformType {};

  public:

    FollowManipulator(osg::PositionAttitudeTransform *node):
      _mat(node) {
      heading = 0;
      pitch = 0;
      roll = 0;
      x = 0;
      y = 0;
      z = -150;
    }

    virtual bool handle(const osgGA::GUIEventAdapter&, osgGA::GUIActionAdapter&) {
      return false;
    }

    virtual void setByMatrix(const osg::Matrixd& m) { }

    virtual osg::Matrixd getMatrix() const {
      osg::Matrix mat, offset;

      // get the matrix of the object/plane
      mat = *(_mat->getWorldMatrices().begin());

      // create the offset matrix
      osg::PositionAttitudeTransform *_trans;
      _trans = new osg::PositionAttitudeTransform();
      _trans->setPosition(osg::Vec3(x, y, z));
      _trans->setAttitude(osg::Quat(
        osg::DegreesToRadians( roll ),osg::Vec3(0,1,0),
        osg::DegreesToRadians( pitch ),osg::Vec3(1,0,0),
        osg::DegreesToRadians( heading ),osg::Vec3(0,0,1)));
      offset = *(_trans->getWorldMatrices().begin());

                        // apply the offset
                        mat = osg::Matrixd::inverse(offset)*mat;

      // this is where the camera should be
      return mat;
    }

    virtual void setByInverseMatrix(const osg::Matrixd &m) { }

    virtual osg::Matrixd getInverseMatrix() const {
      return osg::Matrix::inverse(getMatrix());
    }

    private:

      // the matrix of the object being followed and the offset values
      osg::PositionAttitudeTransform *_mat;
      float heading, pitch, roll;
      float x, y, z;
};
```

To use the `FollowMatrixManipulator` we can add the following line:

```
pKeyswitchManipulator->addMatrixManipulator( '4', "Follow", new
  FollowManipulator(g_pPlaneTransform));
```

You will need to pass to the constructor of this class the transform node of the plane/object that we want the camera to follow. Now when you run your program and select the `FollowMatrixManipulator` you should see the camera positioned above the plane and when you move the plane the camera should follow the plane as it moves.

3.3 Modifying the Camera Follow Vector

Now we need to look at how to add keyboard control to the `FollowMatrixManipulator` so that we can modify the offset values that control the position of the camera relative to the plane. Replace the `handle` method already in your `FollowMatrixManipulator` with the following:

```
virtual bool handle(const osgGA::GUIEventAdapter& ea,osgGA::GUIActionAdapter& us)
{
  switch(ea.getEventType())
  {
  case(osgGA::GUIEventAdapter::KEYDOWN):
    if (ea.getKey()==' ')
    {
      // this could be used to reset your position as with the other cameras
      return true;
    }
    else if (ea.getKey()=='q')
    {
      roll += 10.0f;
      return true;
    }
    else if (ea.getKey()=='a')
    {
      roll -= 10.0f;
      return true;
    }
    return false;

  case(osgGA::GUIEventAdapter::FRAME):
    // this method is called each time the scene is drawn
    // use this method when you want continuous movement
    // see the flying assignment later
    return true;

  default:
    return false;
  }
}
```

Now I want you to modify the `handle` method so that you can alter the x,y, and z position offset and the heading, pitch and roll of the offset. You may find this `FollowMatrixManipulator` useful to add to your robot simulation. You can pass in the transform of one of the robot components and when you move the arm of the robot the camera will follow that component. It may make it easier to see when the robot can pick up objects.

3.4 Walk Camera

You have seen how to create a matrix manipulator to create a follower camera. Now I want you to create a new camera matrix manipulator. This time you will need to implement six keys that allow the camera to move forward and backwards, rotate left and right and move up and down. You make a copy of the `FollowManipulator` camera, giving it a new name and also removing the offset, so you can remove the following lines:

```
offset = *(_trans->getWorldMatrices().begin());

// apply the offset
mat = osg::Matrixd::inverse(offset)*mat;
```

and replace them with:

```
mat = _trans->getWorldMatrices().begin();
```

To create forward motion use the following:

$$x+ = sin(-h)$$
$$y+ = cos(-h)$$

Yes, both lines are needed to move forwards. When you start you may be facing along the x axis, so an increase of your x position will move you along this axis. If you rotate 90 degrees, you will now be facing along the y axis, so you need to alter the y position value to make you move forwards. If you rotate 45 degrees you are in between the two axes and you will need to alter your x and y position to make you move forwards. So, to move backwards use:

$$x- = sin(-h)$$
$$y- = cos(-h)$$

You may also need to set your initial z position at 100 so that you do not appear underground and set the pitch to 90 degrees so you are looking at the horizon. Also, remember that angles need to be given in radians, see Section 2.3.1 for conversion of angles from degrees to radians.

3.4.1 Some modifications

Add some interesting effects to your walking model. You could try things like:

- jumping
- running
- crouching
- kangaroo movement
- something else...

3.5 Flying

For the flying camera, you will need to be able use the keys to control the speed forwards/backwards to accelerate the viewpoint directly forwards and backwards, taking the current orientation into consideration. The ultimate effect of this motion model should be similar to an aircraft.

Given an orientation (h, p, r) and the current speed, we can determine the change in position travelled in a frame by:

$$changeinx = speed \times sin(-h) \times cos(p)$$
$$changeiny = speed \times cos(-h) \times cos(p)$$
$$changeinz = speed \times sin(p)$$

Remember that angles must be given in radians, as we have been doing. You will need to use the `osgGA::GUIEventAdapter::FRAME` event to manage the updating of the movement each frame, according to the current speed and orientation, see Section 3.3. You can extend your work with some extras to the system to aid the user. These could include feedback to the user, implementation of roll, loss of height if you are flying below a certain speed then you start to lose height, etc.

Chapter 4

Using the Mouse

4.1 Introduction

In past practicals we have looked at creating a world and positioning objects in that world; creating camera manipulators that follow objects in the world and controlling the camera independently of any objects in the world and combining this with keyboard input. The last aspect that we need to investigate before attempting the assignment is the use of the mouse as input.

4.2 Mouse Input

We will introduce the mouse through the creation of a new `MatrixManipulator`. This `MatrixManipulator` is similar to the `WalkMatrixManipulator` that we created in the previous practical, except that we will be using the mouse to alter the value of the `heading` rather than using keys on the keyboard. As before, the class starts by initialising the cameras position and orientation. The method for creating the camera's matrix from a `PositionAttitudeTransform` is the same as for the `WalkMatrixManipulator`. The main difference is in the `handle` method that manages the mouse input. Type in the following code (an explanation of the code will follow):

```
class MouseManipulator: public osgGA::MatrixManipulator {

 class UnknownTransformType {};

 public:

  MouseManipulator() {
   heading = 0;
   pitch = 90;
   roll = 0;
   x = 0;
   y = 0;
   z = 100;
  }

  virtual void setByMatrix(const osg::Matrixd& m) { }

  virtual osg::Matrixd getMatrix() const {

   osg::Matrix mat;

   osg::PositionAttitudeTransform *_trans;
   _trans = new osg::PositionAttitudeTransform();
   _trans->setPosition(osg::Vec3(x, y, z));
   _trans->setAttitude(osg::Quat(
     osg::DegreesToRadians( roll ),osg::Vec3(0,1,0),
     osg::DegreesToRadians( pitch ),osg::Vec3(1,0,0),
     osg::DegreesToRadians( heading ),osg::Vec3(0,0,1)));
   mat = *(_trans->getWorldMatrices().begin());
   return mat;
  }

  virtual bool handle(const osgGA::GUIEventAdapter& ea,osgGA::GUIActionAdapter& us)
  {
   switch(ea.getEventType())
   {
   case(osgGA::GUIEventAdapter::PUSH):
    {
     if(ea.getButton()==osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON)
     {
      std::cout << "mouse push    X " << ea.getX() << " Y " << ea.getY() <<std::endl;
      startX = ea.getX();
      startY = ea.getY();
      currentX = startX;
      currentY = startY;
      return true;
     }
     return false;
    }

   case(osgGA::GUIEventAdapter::RELEASE):
    {
     if(ea.getButton()==osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON)
     {
      std::cout << "mouse release X " << ea.getX() << " Y " << ea.getY() <<std::endl;
      startX = currentX = ea.getX();
      startY = currentY = ea.getY();
      return true;
     }
     return false;
    }

   case(osgGA::GUIEventAdapter::DRAG):
    {
     if(ea.getButtonMask()==osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON)
     {
      std::cout << "mouse drag    X " << ea.getX() << " Y " << ea.getY() <<std::endl;
      currentX = ea.getX();
      currentY = ea.getY();
      return true;
     }
     return false;
    }

   case(osgGA::GUIEventAdapter::FRAME):
    heading += (startX - currentX)/100.0f;
    return true;

   default:
    return false;
   }
  }

  virtual void setByInverseMatrix(const osg::Matrixd &m) { }

  virtual osg::Matrixd getInverseMatrix() const {
   return osg::Matrix::inverse(getMatrix());
```

```
    }

  private:

    float heading, pitch, roll;
    float x, y, z;
    // contain initial value on mouse press and current mouse location
    int startX, startY, currentX, currentY;
};
```

You will need to add a new `MouseMatrixManipulator` to the `KeyswitchManipulator`:

```
pKeyswitchManipulator->addMatrixManipulator
  ( '6', "Mouse Example", new MouseManipulator());
```

When you run the program now and press '6' to set the `MouseMatrixManipulator` as the camera manipulator, you should find that by clicking and dragging with the left mouse button, you will alter the heading. The more you drag from the initial click point, the faster you will turn.

In the `handle` method, we look out for four different events: `PUSH, RELEASE, DRAG` and `FRAME`. The first three should be obvious: they are looking for mouse button events. With the `PUSH` and `RELEASE` events we can check to see which mouse button has been pressed when the event was created. With the `DRAG` event we cannot check which mouse button has been pressed (as we assume that for a `DRAG` event that a mouse button has been previously pressed) so we check to see if the left mouse button is being pressed. We do this by checking the mouse button mask. This contains all the mouse buttons that are currently being pressed, and we check to see of the left mouse button is currently being pressed.

For the movement that we want to create, we need to store the initial mouse position when the user presses the left mouse button. It is then the movement from this initial position that is used to calculate the speed that we are going to turn. The further we are from the point where we initially pressed the left mouse button down, the faster we will turn.

The camera's heading is altered in the `FRAME` event. This is called after the world has been drawn/rendered to the screen. We alter the heading in the `FRAME` event as if we included it in the `DRAG` event, the update would only happen each time we moved the mouse while the left mouse button was pressed. You can copy the line that alters the heading into the `DRAG` event and comment it out of the `FRAME` event to see the difference.

4.3 Tutorial

Some modifications that you can add to the `MouseMatrixManipulator`:

- allow movement of the mouse up and down to alter the pitch of the camera
- clamp the pitch so that you can only look up and down within a predefined range. For example, you can only look up and down 90 degrees, giving a 180 degree movement range.
- add keyboard controls so that you can move through the environment