Assignment 3: Modeling and Scene Setting

Out: 3rd October 2015 **Due: 20th October 2015 at 11:55pm**

In this assignment you will interact with a scene graph implementation. Two scene graph models have been provided to you in XML files. A program that should read these models and render them is also provided to you. In this assignment, you will learn how to modify transformations at scene graph nodes to form "poses". You will then create an animated scene graph model of an amusement park ride and put it into your maze. Finally you will extend the scene graph implementation by drawing bounding boxes.

Starting code

A Scenegraphs project has been provided to you. It uses RapidXML, an XML parsing library. This is a header-only library. The RapidXML files are already present in your graphics folder.

The following are the main files and what they do:

1. Node: This is the basic Scene graph node. Each node has a name that can be used to refer to it. By itself it does very little. However several node types extend it.
2. GroupNode: A group node is a non-leaf node that can have multiple children. Each child is a generic Node type.
3. TransformNode: This node represents a transformation. The transformation transforms its child into its parent's coordinate system. The transform node has only one child, which may be a leaf, transform or a group node.
4. LeafNode: This represents a leaf in the scene graph, and is usually a triangle mesh of an actual model. Each LeafNode is associated with (a) an Object, which in turn is a TriangleMesh model and (b) material properties (i.e. color).
5. Material: This class represents the material of an object. For now, its ambient, diffuse and specular are all set to one "color".
6. Scenegraph: This represents a scene graph. It keeps track of the root of the scene graph tree, and renders the scene graph.
7. SceneXMLReader: This parses an XML file that contains a valid scene graph, and create the scene graph tree.

In main.cpp, you can specify the name of the XML file you wish to open. When you run this project, it will open the XML file, parse it, create the scene graph and show it.

While keeping the program running, open the XML file in Notepad or any other text editor. You can make changes to the file, save it, and then open your Scenegraphs program window and press 'R' to refresh the scene graph. This will help you to edit the scene graphs in XML while viewing the results interactively.

Part 1 : Learning to Y-M-C-A

Three other files have been provided to you that draw the humanoid: humanoid-Y.xml, humanoid-M.xml, and humanoid-A.xml (created by Nick Christensen, a former IT 356 student). When you load in these models one-by-one, you will see the humanoid in the "Y", "M" and "A" poses of the popular dance steps to the Y-M-C-A song. Go to http://www.ymcahilo.org/programs.htm to see a visual of these poses (scroll down to the bottom).

Look at the XML files and look for comments that point to the specific transformations that create these poses (e.g. Y pose, M pose, etc.). This will give you an idea about how to add transformations in specific places to move the scene graph accordingly.

Part 2: Amusement park ride (70 points)

In this part, you will create two new XML file that create an amusement park ride each. Then you will modify your code to animate this park ride.

Part 2A: Create the model (30 points)

Each student must create a model. This is the only part where you will be graded differently than your group partner. Please include the name of the creator in comments on top of the XML file.

You are free to create any model, so long as the following constraints are obeyed:

1. There should be at least 4 places to sit. (Model a seat as two boxes, one for the seat and another for the backrest).
2. Each seat should be attached to the main assembly in such a way that when animated, it is possible to transform the seat in at least two ways (e.g. the seats could rotate about themselves while moving in a circle, seats going back and forth as they are moving in a circle, seats rotating about a common axis while the axis itself swings like a pendulum, etc.)

Be sure to use the following features of the XML file that will help you model as well as animate:

a. Build the model "bottom up", like we built the merry-go-around example in class. Start with a seat. Then move it to its correct position by adding it to a transform node, etc. Build this interactively using your program, rather than typing in the entire file and then viewing it!
b. The only node that can store an animation transform is the TransformNode. So make sure you use a TransformNode for the parts that you wish to animate later. Look at the comments in the humanoid.xml file for examples.
c. Use descriptive names for nodes appropriately, using the "name=<value>" attribute to any node.

Part 2B: Animate the model (20 points)

1. After the scene graph is loaded, you should keep pointers to the nodes that you wish to transform in each frame of the animation. You can do this by using the "getNode()" function of the Node class.

   For example, to get a pointer to the entire head-and-neck assembly of the humanoid to make it "nod", you can use the code:

   ```
   Node * headAndNeck = root->getNode("headneck");
   ```

   You can also do this in parts. For example, to get a pointer to the left fore-arm, you can use the code:

   ```
         Node *leftForearm = NULL;
         Node * lefthand = root->getNode("lefthand");
         if (lefthand!=NULL)
         {
               //get the forearm node within the lefthand sub-tree
               leftForearm = lefthand->getNode("forearm");

         }
   ```

2. An empty animate function has been provided to you in the Scenegraph class. Fill this function to animate your specific model.

    a. After you read the scene graph, store the Node pointers for all the nodes you wish to animate.
    b. In animate: for each node obtained as above, determine its transformation for animation using "time".
    c. Set the node's animation transform to this transformation (DO NOT CHANGE the original transform)

Part 2C: Add to maze model (20 points)

Create a scene graph that comprises of a maze (at least 50x50) and your two park ride models placed within empty spaces within the maze.

Hints for creating this scene graph efficiently:

a. Convert your C++ code from Assignment 2 that sets up the maze to an XML file. This may be easier than you think. You can write a function that writes out an XML file, mimicking the logic that you already have for setting up the maze.
b. It is possible to refer to one XML file within another. Look at the "two-humanoids.xml" file for an example. Use this to build your scene graph possibly from different parts.

Part 3: Twin Views with maze solution (30 points)

Set up the program so that it shows the following on the screen:

1. The 3D scene with maze and rides in it. The trackball interface from Assignment 2 should rotate the entire 3D scene about the center of the maze. Set up the camera so that initially the maze is viewed from a suitable angle.
2. The upper right corner of the screen should show the 2D rendition of the same maze from Assignment 1 (a thumbnail view). You can achieve this by retaining the code from Assignment 1, and drawing it to a viewport in the upper right corner of the screen.
3. The 2D view should also show the solution to the maze (i.e. a path through the maze from the starting cell to the ending cell).
    a. The PathFinder.jar program provided with Assignment 1 generates, for each cell "C" the next cell along the shortest path from "C" to the ending cell. Use this information to determine the shortest path from the starting cell to the ending cell as follows: start at the starting cell and find the next cell, then find its next cell and so on until you reach the ending cell. Recall that you did this in IT 279.
    b. Create extra 2D lines that show this path in the thumbnail view.

Program details

1. Prepare your program so that when I build and run, your model will be seen entirely and animated. The trackball should be functional, and the thumbnail view should be clearly visible.
2. Your program should work in the lab! If you are working on your own machine, it is your responsibility to ensure that it works in the lab before you submit it.

What to submit

Submit the Visual studio project (please clean and remove the huge database file before submitting), and all the XML files necessary to run your program successfully in a single zipped file on Reggienet.