1. Scene-Graph:

A scene graph is a data structure that is a combination of a tree and a graph. It basically stores nodes, that represent the images to be drawn, in a hierarchical fashion. This allows a single operation to be applied on multiple nodes that belong to the same tree/ sub-tree.

A scene graph is made up of a root node and 1 or more composite nodes. Each on the composite node has a name and can have a polygon mesh associated with it. The name is provided to facilitate a more efficient search for a node, in the tree. The polygon mesh represents the object to be drawn at that given node. This association between the nodes, their names and the meshes as stored as maps. One map associates the name of nodes, to the INode implementation, another map associates the names with the polygon meshes, while a third map associates the name of the node with the textures to be applied at that node.

To create a scene graph, we first create the root node. A reference of this node is passed to all the composite nodes in the tree, so that any node can call the functions of its associated scene graph. A composite node can either be a leaf or a tree structure itself.

This functionality is encapsulated in the following classes:

1. LeafNode

This class represents the leaf node in the scene graph. The leaf node is the only node that contains geometric data that is to be rendered on the screen. It provides the functionality to set the texture of the given node, create a copy of the given node, draw the node by passing the modelview stack to the renderer

2. GroupNode

This class represents a tree of a sub-tree. Each of the node has one more child nodes. Each of those child nodes can either be a leaf node or a sub-tree itself. It provides the functionality to search the tree recursively for a give node, draw the given node, create a copy of the tree, with the current node as it's root, to add a new child, return all the child nodes associated with the sub-tree.

3. AbstractNode

This is an abstract class and provides default/abstract implementations of method mentioned above. The classes LeafNode and GroupNode and extend this class and implement the methods as required.

4. TransformNode

This class represents a transformation. It has one child, since a transformation represents a conversion from the child's co-ordinate system to its parent's co-ordinate system. It provides the functionality to clone the given node, add a single child, check if a given node is its child or not, and finally draw the child. This is done be pushing the current modelview on to the stack and the applying the transformations to the stack top. Hence, preserving the order and ensuring that the transformations that are applied to the parent are also applied to the child.

In the given implementation of the scene graph, a node is represented using the INode interface. The INode interface provides abstract implementations to perform the following operations:

* Search a given node using the name of the node.
* Draw the scene graph by passing the modelview stack to the renderer.
* Return a clone of the current scene graph.
* To set a parent of the given node,
* Add a new child to current node under consideration.
* Set the transformations associated with this node.

Each node in the scene graph, implements one or more of these functions as per its requirements.

2. Drawing a scene graph:

The given implementation of the scene graph is traversed as a depth first search tree, following the pre-order traversal. As mentioned earlier, only the leaf nodes have actual geometry data to render. Hence, so as to draw images on the screen, the scene graph is traversed till the leaf node is reached, at which point the geometry data is passed to the renderer, and the associated rendered will draw the given image/ polygon mesh using the supplied modelview stack.

Transformations:

Since the scene graph follows a hierarchy, a transformation applied to the root of a tree or a sub-tree will be applied to all the nodes in that given tree/ sub-tree. Each node in the scene graph represents a local space. In other words, each node has its own co-ordinate system. Therefore, a transformation on a child-node can be viewed, as changing the co-ordinate system from the child's co-ordinate system to the parent's co-ordinate system.

To preserve the current positions of the object in the world, we first push the current modelview matrix on to the stack, to which all the transformations are applied, after which we recursively call the child of the given node, without popping the matrix, hence ensuring the proper order of the transformations. Once we reach the leaf node, the matrices are popped.

3. GL3ScenegraphRenderer:

The GL3ScenegraphRenderer Class is basically a renderer, a class that draws the image based on the modelview stack and the JOGL specific data provided to it. Each scene graph is associated with a renderer. The renderer can be used to set the rendering context, a rendering context is a container for state information. When you set the rendering context of the object as the current rendering context, the subsequent GL commands modify the context's state or the objects associated with that context. It can also be used to set the shader program, and the shader location. It can be used to draw the scene graph, or to draw the leaf node, or to set a texture to the given node. All this functionality is encapsulated in the GL3ScenegraphRenderer class.

Textual Renderer:

Implementing a textual renderer such that instead of loading images on the screen, the renderer would display all the operations that would occur (transformations, renders, etc).

Assumption: The scene-graph files are stored as .xml files, exactly the way they were provided.

The overall implementation of the scene graph will remain the same, i.e. the classes used to “construct” the scene graph (LeafNode, GroupNode, etc) will majorly remain the same, except the changes in the draw method.

Algorithm to implement a textual renderer:

Assumption: The scene graph is already created, the following algorithm is only to load the scene graph

1. Initialize the modelview stack.
2. If root is not null, traverse the scene graph as a pre-order DFS tree.
3. While the current node is not a leaf node:
4. Using the SceneXMLReader Class, identity all the groups and the transformations that are to be applied to those elements.
5. Traverse to the next child in the subtree.
6. Append the transformation matrix at this node to the current transformation matrix and print the transformations that were performed at the node a well as the group name for the given element, for identification.
7. If the current node is a leaf node, print “Rendering < group-name>” of the element from the xml file.
8. Repeat steps 2 – 4 till the entire scene-graph is traversed.

Program Implementation:

Design:

As mentioned in the problem statement, the scene depicts a floor of a building. The floor has 3 rooms, each room has a door and a window. The room on the right has the table with 3 objects on top of it. The program loads with camera just outside the room looking at the table through the window. The user can move the camera using the arrow keys. Up and Down to move the camera forward and backwards. Right and Left to move rotate the camera in the left and right directions. And “S” and “W” to rotate the camera up and down.

Pressing “Y” on the keyboard will switch the view to the centre of the floor, where the you will see the Y, M, C, A humanoid models. The camera controls and disabled for this scene. Pressing “R” on the keyboard will switch to the room view of the scene and camera controls will be enabled.

Working:

The camera controls are implemented as follows:

1. Pressing the keys “Left” or “Right” will rotate the camera 0.1 degrees per key-press in the appropriate direction, by translating the camera to the origin, then performing the rotations along the Y-axis and then undoing the transformation.
2. Pressing the keys “Up” or “Down” will move the position of the camera 2 pixels per key-press in the appropriate direction by simply changing the z co-ordinate of the camera.

1. Pressing the keys “W” and “S” will rotate the camera 0.1 degrees per key-press in the up and down direction respectively, by translating the camera to the origin, then performing the rotations along the X-axis and then undoing the translation as well as rotating the Up vector of the camera.
2. I have set up two cameras and pressing the keys “Y” and “R” will switch between the room scene and the floor scene respectively.

XML Files:

The entire scene is loaded as 3 XML files. One for the entire floor, one for the table-scene and one for the YMCA-Humanoids. Each of these files have references to individual Xml files for the said object. The individual files are stored in their separate folder, identified with their name.

**\*I couldn’t provide the functionality to move the camera in the direction that it is facing.**