COM3503/4503/6503: 3D Computer Graphics

Dr Steve Maddock

Exercise sheet 6: Complex scenes

1 Introduction

This exercise sheet will develop a scene graph class structure. Before that, however, Section 2 will cover some changes required to the Mesh structure to support the use of a scene graph. As we develop the scene graph, I'll also add some buttons to the GUI which can be used to alter particular variables that control the scene.

In the examples below, I've used separate folders as we progress through the programs. It is important to keep the code in separate folders. This is because some of the classes are updated from folder to folder – a version in one folder will not necessarily work in another folder.

You will need to compile programs using javac M01.java, javac M02.java, etc rather than javac *.java. if you use javac *.java in Week6_3_scene_graph, you'll get errors. This is because of the extra classes inside M03.java, M04.java and M05.java.

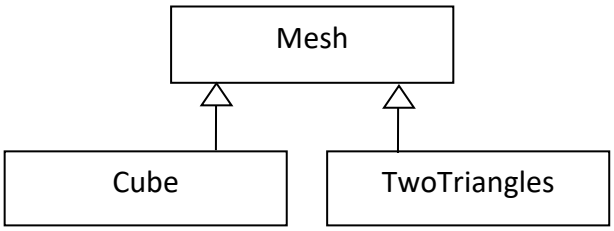

Figure 2.1: The Mesh class and subclasses

2 Mesh

Program: Week6_2_mesh_change\M01.java

In previous exercise sheets, we have used a Mesh class to store a set of vertices and connectivity information and to offer a render method that uses a shader and textures. The main programs have made use of Cube and TwoTriangles classes (which extend the Mesh class, as illustrated in Figure 2.1) for a few objects.

Program Listing 2.1 shows the main parts of a new Mesh class, with the new parts highlighted in bold. The main changes are the inclusion of three new attributes, camera, perspective and light, methods to set these attributes, and two new render methods.

The render method used to be:

```
public abstract void render(GL3 gl, Light light, Vec3 viewPosition, Mat4 perspective,
Mat4 view);
```

Now, the relevant parameters are attributes of the Mesh class, so do not need to be included in the parameter list for the render method.

The two render methods serve different purposes. **`render(GL3 gl, Mat4 model)`** is used in conjunction with the scene graph that will be introduced later. A model matrix can be supplied as a parameter to override the model matrix stored as an attribute in the class. The other render method, **`render(GL3 gl)`**, makes use of the model matrix attribute.

```
import gmaths.*;
import java.nio.*;
import com.jogamp.common.nio.*;
import com.jogamp.opengl.*;

public abstract class Mesh {
  protected float[] vertices;
  protected int[] indices;
  private int vertexStride = 8;
  private int vertexXYZFloats = 3;
  private int vertexNormalFloats = 3;
  private int vertexTexFloats = 2;
  private int[] vertexBufferId = new int[1];
  protected int[] vertexArrayId = new int[1];
  private int[] elementBufferId = new int[1];
  protected Material material;
  protected Shader shader;
  protected Mat4 model;
  protected Camera camera;
  protected Mat4 perspective;
  protected Light light;

  public Mesh(GL3 gl) {
    material = new Material();
    model = new Mat4(1);
  }

  public void setModelMatrix(Mat4 m) {
    model = m;
  }

  public void setCamera(Camera camera) {
    this.camera = camera;
  }

  public void setPerspective(Mat4 perspective) {
    this.perspective = perspective;
  }

  public void setLight(Light light) {
    this.light = light;
  }

  public void dispose(GL3 gl) {
    gl.glDeleteBuffers(1, vertexBufferId, 0);
    gl.glDeleteVertexArrays(1, vertexArrayId, 0);
    gl.glDeleteBuffers(1, elementBufferId, 0);
  }

  protected void fillBuffers(GL3 gl) {  // same as previous version of Mesh }

  public abstract void render(GL3 gl, Mat4 model);

  public void render(GL3 gl) {
    render(gl, model);
  }

}
```

Program listing 2.1: Mesh.java.

Figure 2.2 shows the output from M01.java. A new Sphere class is used. Program Listing 2.2 shows M01_GLEventListener.initialise(). I've moved the class's attributes to the start of the listing so that it is easier to see what is happening. Five textures are loaded. A TwoTriangles instance is created and stored in tt1. The model matrix for this mesh is set to scale by 16 in the X and z coordinates to make the floor bigger than the default 1x1. textureId0 is used, since the Shader stored in the TwoTriangles class expects only a single texture.

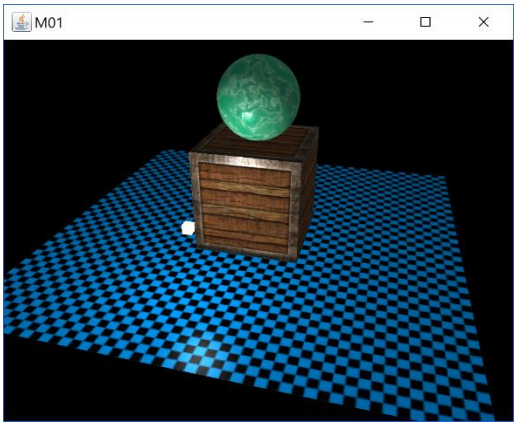

Figure 2.2: Output from M01.java

```
private Camera camera;
private Mat4 perspective;
private Mesh tt1, cube, sphere;
private Light light;

public void initialise(GL3 gl) {
  createRandomNumbers();
  int[] textureId0 = TextureLibrary.loadTexture(gl, "textures/chequerboard.jpg");
  int[] textureId1 = TextureLibrary.loadTexture(gl, "textures/container2.jpg");
  int[] textureId2 = TextureLibrary.loadTexture(gl,
                                                "textures/container2_specular.jpg");
  int[] textureId3 = TextureLibrary.loadTexture(gl, "textures/jade.jpg");
  int[] textureId4 = TextureLibrary.loadTexture(gl, "textures/jade_specular.jpg");

  tt1 = new TwoTriangles(gl, textureId0);
  tt1.setModelMatrix(Mat4Transform.scale(16,1f,16));

  cube = new Cube(gl, textureId1, textureId2);
  Mat4 model = Mat4.multiply(Mat4Transform.scale(4,4,4),
                             Mat4Transform.translate(0,0.5f,0));
  cube.setModelMatrix(model);

  sphere = new Sphere(gl, textureId3, textureId4);
  model = Mat4.multiply(Mat4Transform.scale(3,3,3),
                        Mat4Transform.translate(0,0.5f,0));
  model = Mat4.multiply(Mat4Transform.translate(0,4,0), model);
  sphere.setModelMatrix(model);

  light = new Light(gl);
  light.setCamera(camera);

  tt1.setLight(light);
  tt1.setCamera(camera);
  cube.setLight(light);
  cube.setCamera(camera);
  sphere.setLight(light);
  sphere.setCamera(camera);
}
```

Program listing 2.2: M01_GLEventListener.initialise().

Next, an instance of Cube is created. This used the container diffuse and specular maps used in previous programs. The model matrix transformation first translates the cube 0.5 in the positive y direction (so as to put the bottom of the cube at the point (0,0,0)), and then scales by 4 to create a 4x4x4 cube which has the centre of its base at (0,0,0). The new Sphere class is then used to create a sphere that is transformed to sit on top of the cube (which has a y height of 4 – this should be stored in a variable rather than using literals).

A light is then created. Each of the meshes (including the light) needs to be given a reference to the camera, since the Shaders they use need to know the view transformation matrix which is stored in the Camera instance. The TwoTriangles, Cube and Sphere instances also need to be given a reference to the Light instance, since their Shaders will need information stored in the Light instance.

The render method is given in Program Listing 2.3. This makes use of the new Mesh.render(gl) method in Program Listing 2.1. Note the call to updatePerspectiveMatrices(). This must be done in case the user has resized the window. For efficiency, the call could be moved to M01_GLEventListener.reshape() since that method is automatically called when the user resizes the display area. We'll leave it in render for now, just so it's clear what is happening. The light position is changing every frame, so light.setPosition() must be called with the new position. Every Mesh has a reference to the light (see Program Listing 2.2 where this was set up) so can easily retrieve the new light attributes when a Shader is used.

```
private void render(GL3 gl) {
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);

  updatePerspectiveMatrices();

  light.setPosition(getLightPosition());   // changing light position each frame
  light.render(gl);

  tt1.render(gl);
  cube.render(gl);
  sphere.render(gl);
}

private void updatePerspectiveMatrices() {
  // needs to be changed if user resizes the window
  perspective = Mat4Transform.perspective(45, aspect);
  light.setPerspective(perspective);
  tt1.setPerspective(perspective);
  cube.setPerspective(perspective);
  sphere.setPerspective(perspective);
}

// The light's position is continually being changed, so needs to be calculated for each frame.
private Vec3 getLightPosition() {
  double elapsedTime = getSeconds()-startTime;
  float x = 5.0f*(float)(Math.sin(Math.toRadians(elapsedTime*50)));
  float y = 2.7f;
  float z = 5.0f*(float)(Math.cos(Math.toRadians(elapsedTime*50)));
  return new Vec3(x,y,z);
}
```

Program listing 2.3: M01_GLEventListener.render().

The setting of all the perspective matrices in each Mesh instance is a little messy, so we'll return to that in a later section when we look at efficiency issues. We could also consider a list or array structure to contain all the meshes to tidy things up. However, that is left as an exercise.

Exercises

1. Stack another sphere on top of the one in Figure 2.2

3 The scene graph

Program: Week6_3_scene_graph \M02.java. The output is given in Figure 3.1. One cuboid is stacked on top of another. You may think that this is similar to M01.java in the previous section. However, a scene graph is used. The flexibility that this gives us will become clearer in later programs. For now, we need to study the scene graph.

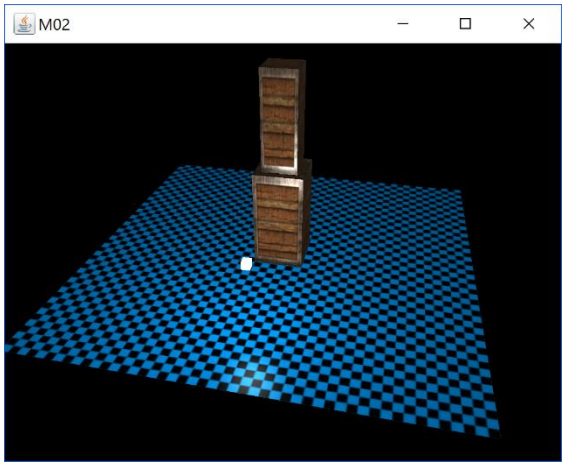

Figure 3.1: Output from M02.java

Figure 3.2 gives the scene graph for a two-branch structure from Lecture 03. This is the scene graph used to produce Figure 3.1, albeit using cubes instead of spheres. Program Listing 3.1 gives the code to set up the scene graph.

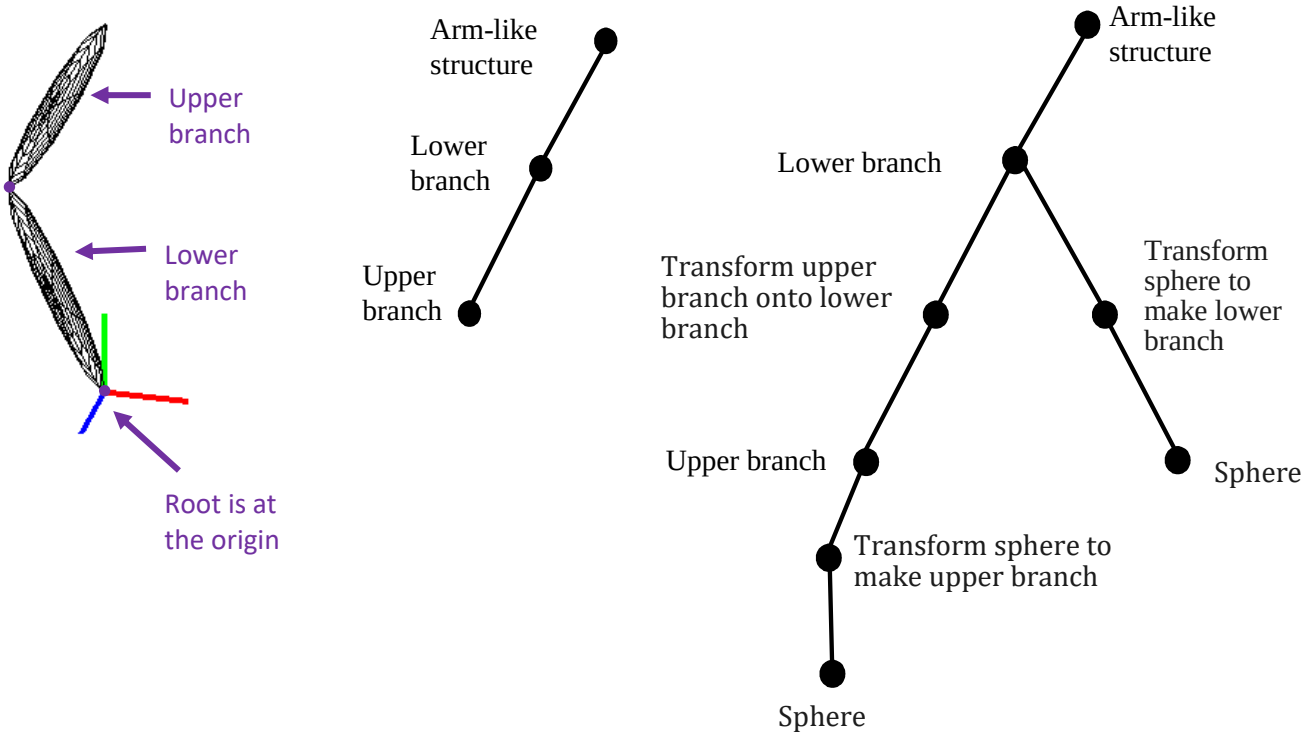

Figure 3.2: A two-branch structure

```
private Camera camera;
private Mat4 perspective;
private Mesh floor, cube;
private Light light;
private SGNode twoBranch;

private void initialise(GL3 gl) {
  createRandomNumbers();
  int[] textureId0 = TextureLibrary.loadTexture(gl, "textures/chequerboard.jpg");
  int[] textureId1 = TextureLibrary.loadTexture(gl, "textures/container2.jpg");
  int[] textureId2 = TextureLibrary.loadTexture(gl, "textures/container2_specular.jpg");

  floor = new TwoTriangles(gl, textureId0);
  floor.setModelMatrix(Mat4Transform.scale(16,1f,16));

  cube = new Cube(gl, textureId1, textureId2);

  light = new Light(gl);
  light.setCamera(camera);

  floor.setLight(light);
  floor.setCamera(camera);
  cube.setLight(light);
  cube.setCamera(camera);

  twoBranch = new NameNode("two-branch structure");
    NameNode lowerBranch = new NameNode("lower branch");
      Mat4 m = Mat4Transform.scale(2,4,2);
      m = Mat4.multiply(m, Mat4Transform.translate(0,0.5f,0));
      TransformNode lowerBranchTransform
          = new TransformNode("scale(2,4,2); translate(0,0.5,0)", m);
        MeshNode lowerBranchShape = new MeshNode("Cube(0)", cube);
      TransformNode translateToTop
          = new TransformNode("translate(0,4,0)",Mat4Transform.translate(0,4,0));
        NameNode upperBranch = new NameNode("upper branch");
          m = Mat4Transform.scale(1.4f,3.9f,1.4f);
          m = Mat4.multiply(m, Mat4Transform.translate(0,0.5f,0));
          TransformNode upperBranchTransform
              = new TransformNode("scale(1.4f,3.9f,1.4f);translate(0,0.5,0)", m);
            MeshNode upperBranchShape = new MeshNode("Cube(1)", cube);

  twoBranch.addChild(lowerBranch);
    lowerBranch.addChild(lowerBranchTransform);
      lowerBranchTransform.addChild(lowerBranchShape);
    lowerBranch.addChild(translateToTop);
      translateToTop.addChild(upperBranch);
      upperBranch.addChild(upperBranchTransform);
      upperBranchTransform.addChild(upperBranchShape);
  twoBranch.update(); // IMPORTANT – must be done every time any part of the scene graph changes
  // Following two lines can be used to check scene graph construction is correct
  //twoBranch.print(0, false);
  //System.exit(0);
}
```

Program listing 3.1: Initialising the scene graph for the two-branch structure.

The first part of method initialise() creates the pieces that are used in the scene graph: a floor represented as an instance of TwoTriangles, a cube and a light. Only one cube is needed. The same cube can be used to draw each part of the two-branch structure, but with different model transformations used in each case.

The two-branch structure, twoBranch, is an instance of SGNode, declared as an attribute of the class. Figure 3.3 shows the class structure for an SGNode. The link from an SGNode to itself is an indicator that this is a recursive structure. In fact, an SGNode contains an ArrayList of child nodes, each of which is an SGNode.

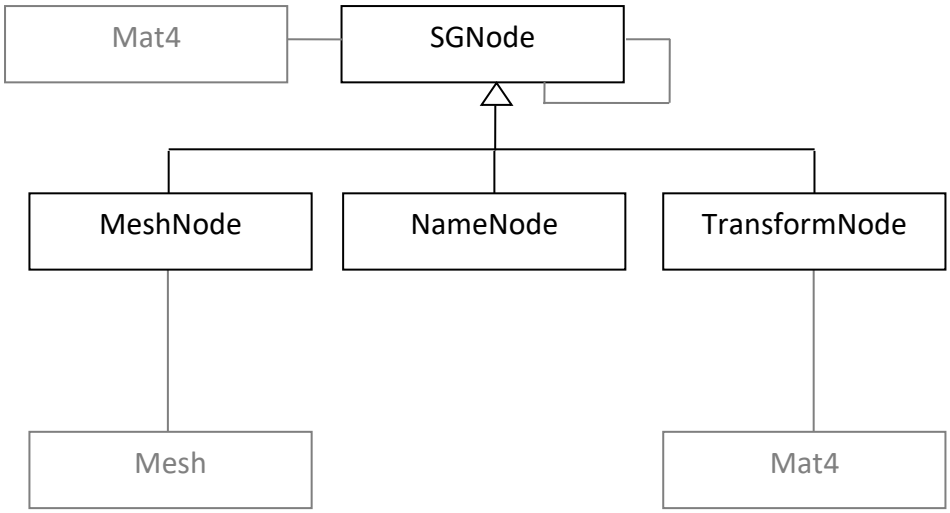

Figure 3.3: A scene graph structure

Three other classes extend SGNode: a MeshNode contains a reference to a Mesh instance; a NameNode is used solely to make the scene graph hierarchy clearer by allowing nodes that contain nothing but a String to represent their name; and a TransformNode is used to represent a transformation (a Mat4 instance) to be applied to its children in the scene graph. Cross-referencing this structure to Figure 3.2, we have a mixture of NameNodes (Arm-like structure, Lower branch, Upper branch), MeshNodes (Sphere) and TransformNodes (Transform upper branch onto lower branch, Transform sphere to make lower branch, Transform sphere to make upper branch). This is reflected in the code in Program Listing 3.1. SGNode contains a method called addChild() which is used to build the scene graph. I've used indentation in the program code to make the hierarchy clearer, although some may argue that it makes the code a little more difficult to read. It's personal choice.

You should be able to match the code in Program Listing 3.1 to the nodes in Figure 3.2. For example, the cube is used in two places in the scene graph hierarchy, but each time is preceded by a different TransformNode. As we discussed in lectures, when drawing an object, it will be transformed by all the transformations in its ancestry all the way up to the root node. The lower branch cube is subject to a scale and a translation. The upper branch cube is subject to a translation, a scale and another translation.

The line `twoBranch.update();` is important. This is called after the scene graph is built. It starts at the root and traverses the scene graph updating the 'world transformation' stored in every node (every SGNode contains a Mat4 instance, as illustrated in Figure 3.3), combining transformations from TransformNodes as it descends the parent-child hierarchy in the scene graph. When it has completed, every node will store a world transformation that is correct for that node in the scene graph. Thus, a MeshNode will have the correct world transformation stored and ready to be used when rendering the Mesh. Whenever the scene graph is changed, the SGNode.update() method must be called.

We are now ready to render the scene graph. However, it is worth mentioning the two commented out lines at the end of Program Listing 3.1. If you uncomment these and run the program, the scene graph will be displayed and then the program will quit before displaying any objects. (Rather than System.exit(0), you could also wait for user input and then continue the program.) The SGNode.print() method can be useful in debugging to make sure the scene graph matches what you think you have created.

Program Listing 3.2 gives the M02_GLEventListener.render() method. This is similar to the render method in Program Listing 2.3. The main difference is the call to twoBranch.draw(gl), which renders the scene graph. (I've called the method draw() to distinguish it from render(), which is used as the method name in M02_GLEventListener and also for a Mesh. Some people might prefer drawGraph() to make it even clearer.) This method starts at the root of the scene graph and recursively calls draw on the children of a node. When a MeshNode is encountered, the method Mesh.render(gl, model) (see Program Listing 2.1) is used for the Mesh instance, e.g. mesh.render(gl, worldTransform). The world transform stored in the MeshNode instance is passed as the model matrix when rendering the Mesh instance. That's why we needed this method in Program Listing 2.1

Exercise

1. Replace the two-branch structure with the tree-like structure given in Lecture 03. This structure had two upper branches on top of the lower branch.

```
private void render(GL3 gl) {
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
  updatePerspectiveMatrices();
  light.setPosition(getLightPosition());   // changing light position each frame
  light.render(gl);
  floor.render(gl);
  twoBranch.draw(gl);
}

private void updatePerspectiveMatrices() {
  // needs to be changed if user resizes the window
  perspective = Mat4Transform.perspective(45, aspect);
  light.setPerspective(perspective);
  floor.setPerspective(perspective);
  cube.setPerspective(perspective);
}

// The light's position is continually being changed, so needs to be calculated for each frame.
private Vec3 getLightPosition() {
  double elapsedTime = getSeconds()-startTime;
  float x = 5.0f*(float)(Math.sin(Math.toRadians(elapsedTime*50)));
  float y = 2.7f;
  float z = 5.0f*(float)(Math.cos(Math.toRadians(elapsedTime*50)));
  return new Vec3(x,y,z);
  //return new Vec3(5f,3.4f,5f);
}
```

Program listing 3.2: Rendering the two-branch structure.

4 SGNode

Program Listings 4.1, 4.2, 4.3 and 4.4 give the classes for SGNode, MeshNode, NameNode and TransformNode, respectively.

```
import gmaths.*;
import java.util.ArrayList;
import com.jogamp.opengl.*;

public class SGNode {

  protected String name;
  protected ArrayList<SGNode> children;
  protected Mat4 worldTransform;

  public SGNode(String name) {
    children = new ArrayList<SGNode>();
    this.name = name;
    worldTransform = new Mat4(1);
  }

  public void addChild(SGNode child) {
    children.add(child);
  }

  public void update() {
    update(worldTransform);
  }

  protected void update(Mat4 t) {
    worldTransform = t;
    for (int i=0; i<children.size(); i++) {
      children.get(i).update(t);
    }
  }

  protected String getIndentString(int indent) {
    String s = ""+indent+" ";
    for (int i=0; i<indent; ++i) {
      s+="  ";
    }
    return s;
  }

  public void print(int indent, boolean inFull) {
    System.out.println(getIndentString(indent)+"Name: "+name);
    if (inFull) {
      System.out.println("worldTransform");
      System.out.println(worldTransform);
    }
    for (int i=0; i<children.size(); i++) {
      children.get(i).print(indent+1, inFull);
    }
  }

  public void draw(GL3 gl) {
    for (int i=0; i<children.size(); i++) {
      children.get(i).draw(gl);
    }
  }

}
```

Program listing 4.1: SGNode.java.

SGNode does most of the work. It contains member variables for a name, a list of children and a world transform matrix. Method addChild() adds a node to the list of children. There are two versions of update(). The public update() is used to propagate changes to the world transformation matrix recursively to each of the children. This uses the internal protected update(Mat4) to pass the updated worldTranformMatrix down the hierarchy. TransformNode overrides this second version of the method and combines in the transformation stored in the TransformNode, as shown in Program Listing 4.4. Method print is used for printing a version of the scene graph, used for debugging purposes. Finally, method draw() recursively calls the draw method for each of the children. The draw() method is overridden in MeshNode, so that mesh.render() is called before then recursively drawing each of its children. Class NameNode does very little. Its sole purpose is for nodes in the scene graph like lowerBranch in Figure 3.2, which help to make the structure of the scene graph clearer. SGNode could have been used instead of NameNode, but I think it helps to make the purpose of the node clearer. One could also argue that SGNode should be an abstract class to make this even clearer.

```
import com.jogamp.opengl.*;

public class MeshNode extends SGNode {

  protected Mesh mesh;

  public MeshNode(String name, Mesh m) {
    super(name);
    mesh = m;
  }

  public void draw(GL3 gl) {
    mesh.render(gl, worldTransform);
    for (int i=0; i<children.size(); i++) {
      children.get(i).draw(gl);
    }
  }

}
```

Program listing 4.2: MeshNode.java.

```
import gmaths.*;
import com.jogamp.opengl.*;

public class NameNode extends SGNode {

  public NameNode(String name) {
    super(name);
  }

}
```

Program listing 4.3: NameNode.java.

```
import gmaths.*;
import com.jogamp.opengl.*;

public class TransformNode extends SGNode {

  private Mat4 transform;

  public TransformNode(String name, Mat4 t) {
    super(name);
    transform = new Mat4(t);
  }

  public void setTransform(Mat4 m) {
    transform = new Mat4(m);
  }

  public void update(Mat4 t) {
    worldTransform = t;
    t = Mat4.multiply(worldTransform, transform);
    for (int i=0; i<children.size(); i++) {
      children.get(i).update(t);
    }
  }

  public void print(int indent, boolean inFull) {
    System.out.println(getIndentString(indent)+"Name: "+name);
    if (inFull) {
      System.out.println("worldTransform");
      System.out.println(worldTransform);
      System.out.println("transform node:");
      System.out.println(transform);
    }
    for (int i=0; i<children.size(); i++) {
      children.get(i).print(indent+1, inFull);
    }
  }

}
```

Program listing 4.4: TransformNode.java.

5 Altering transformations in the scene graph

Program: Week6_3_scene_graph\M03.java. The output is shown in Figure 5.1.

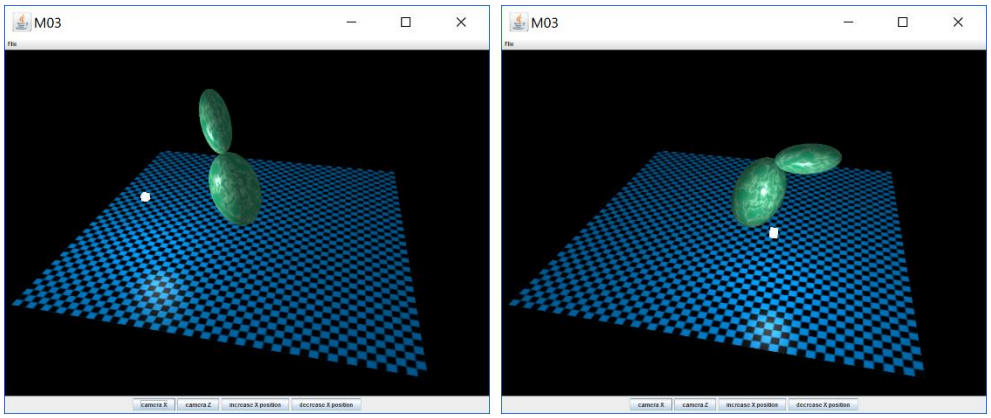

Figure 5.1: Two frames from M03.java

This program is similar to the M02.java. A sphere is used instead of a cube, so that the two 'branches' are now ellipsoids. The main difference, however, is that the rotations of the upper and lower branch change on every frame that is rendered. Program Listing 5.1 shows the initialise() method. This is similar to Program Listing 3.1. The differences are highlighted in bold. Three TransformNodes are declared at the class level, so are then accessible anywhere in the class. Some extra attributes are also declared to keep track of certain values. xPosition stores the current xPosition of the base of the two-branch structure. The variables storing rotations are used to rotate either the whole two-branch structure or to rotate the upper branch about its connection to the lower branch. (It may be helpful to re-read the notes from Lecture 03 at this point.)

```
private TransformNode translateX, rotateAll, rotateUpper;
private float xPosition = 0;
private float rotateAllAngleStart = 25, rotateAllAngle = rotateAllAngleStart;
private float rotateUpperAngleStart = -60, rotateUpperAngle = rotateUpperAngleStart;

private void initialise(GL3 gl) {
  // create objects code goes here

  twoBranch = new NameNode("two-branch structure");
    translateX = new TransformNode("translate("+xPosition+",0,0)",
                     Mat4Transform.translate(xPosition,0,0));
      rotateAll = new TransformNode("rotateAroundZ("+rotateAllAngle+")",
                       Mat4Transform.rotateAroundZ(rotateAllAngle));
        NameNode lowerBranch = new NameNode("lower branch");
          Mat4 m = Mat4Transform.scale(2.5f,4,2.5f);
          m = Mat4.multiply(m, Mat4Transform.translate(0,0.5f,0));
          TransformNode makeLowerBranch
              = new TransformNode("scale(2.5,4,2.5); translate(0,0.5,0)", m);
            MeshNode cube0Node = new MeshNode("Sphere(0)", sphere);
        TransformNode translateToTop
              = new TransformNode("translate(0,4,0)",Mat4Transform.translate(0,4,0));
          rotateUpper = new TransformNode("rotateAroundZ("+rotateUpperAngle+")",
                            Mat4Transform.rotateAroundZ(rotateUpperAngle));
            NameNode upperBranch = new NameNode("upper branch");
              m = Mat4Transform.scale(1.4f,3.1f,1.4f);
              m = Mat4.multiply(m, Mat4Transform.translate(0,0.5f,0));
              TransformNode makeUpperBranch
                  = new TransformNode("scale(1.4f,3.1f,1.4f);translate(0,0.5,0)", m);
                MeshNode cube1Node = new MeshNode("Sphere(1)", sphere);

  twoBranch.addChild(translateX);
    translateX.addChild(rotateAll);
      rotateAll.addChild(lowerBranch);
        lowerBranch.addChild(makeLowerBranch);
          makeLowerBranch.addChild(cube0Node);
        lowerBranch.addChild(translateToTop);
          translateToTop.addChild(rotateUpper);
            rotateUpper.addChild(upperBranch);
              upperBranch.addChild(makeUpperBranch);
                makeUpperBranch.addChild(cube1Node);
  twoBranch.update();  // IMPORTANT – must be done every time any part of the scene graph changes
}
```

Program listing 5.1: M03_GLEventListener.initialise().

Program Listing 5.2 gives the render method. The method updateBranches() is called before the scene graph, twoBranch, is drawn. Method updateBranches() uses the elapsed system time to create new values for rotateAllAngle and rotateUpperAngle. The relevant TransformNodes in the scene graph are then updated accordingly.

Now, we see the power of the scene graph in action. By storing references to specific TransformNodes in the scene graph, we can change the transforms stored in them. We then update the scene graph (twoBranch.update()) to update all the world transformations stored in each dependent node, and then the scene graph can be redrawn using the new values.

```
private void render(GL3 gl) {
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
  updatePerspectiveMatrices();
  light.setPosition(getLightPosition());  // changing light position each frame
  light.render(gl);
  floor.render(gl);
  updateBranches();
  twoBranch.draw(gl);
}

private void updateBranches() {
  double elapsedTime = getSeconds()-startTime;
  rotateAllAngle = rotateAllAngleStart*(float)Math.sin(elapsedTime);
  rotateUpperAngle = rotateUpperAngleStart*(float)Math.sin(elapsedTime*0.7f);
  rotateAll.setTransform(Mat4Transform.rotateAroundZ(rotateAllAngle));
  rotateUpper.setTransform(Mat4Transform.rotateAroundZ(rotateUpperAngle));
  twoBranch.update();  // IMPORTANT – the scene graph has changed
}
```

Program listing 5.2: M03_GLEventListener.render().

That leaves us with the class attributes float xPosition and TransformNode translateX. You'll note in Figure 5.1 that there are a series of buttons at the bottom of the window. These are created in the constructor in M03.java. The relevant lines are given in Program Listing 5.3. Here, we can see that four JButtons are created and added to the SOUTH of the window. M03 implements the ActionListener interface, which means it needs to supply a method called actionPerformed(ActionEvent e) that responds to action events such as button presses in the GUI. This method detects which button is pressed and takes appropriate action. Here, two buttons are used to set the camera's position to two specific positions. The other two buttons call methods `glEventListener.incXPosition();` and `glEventListener.deccXPosition();`, respectively. Program Listing 5.4 gives these methods. Each changes the value of the variable xPosition and uses this to update the value in the TransformNode `translateX.` `translateX.update()` is then called. It is sufficient to update the scene graph from this node downwards to its children. There is no need to start at the root of the scene graph, since the parent nodes have not changed. When the scene graph is next drawn the new value will be used.

Exercise

1. Change the lower branch so that it is a cube instead of a sphere.
2. Adapt your answer to the exercise in Section 3 so that the x position of the tree-like structure can be controlled from button presses and each of the rotation transforms varies over time.

```
// import statements

public class M03 extends JFrame implements ActionListener {

// attributes and methods

  public M03(String textForTitleBar) {

    // adding the canvas and other code

    JMenuBar menuBar=new JMenuBar();
    this.setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
        JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(this);
        fileMenu.add(quitItem);
    menuBar.add(fileMenu);

    JPanel p = new JPanel();
      JButton b = new JButton("camera X");
      b.addActionListener(this);
      p.add(b);
      b = new JButton("camera Z");
      b.addActionListener(this);
      p.add(b);
      b = new JButton("increase X position");
      b.addActionListener(this);
      p.add(b);
      b = new JButton("decrease X position");
      b.addActionListener(this);
      p.add(b);
    this.add(p, BorderLayout.SOUTH);

    // and the rest of the method
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equalsIgnoreCase("camera X")) {
      camera.setCamera(Camera.CameraType.X);
      canvas.requestFocusInWindow();
    }
    else if (e.getActionCommand().equalsIgnoreCase("camera Z")) {
      camera.setCamera(Camera.CameraType.Z);
      canvas.requestFocusInWindow();
    }
    else if (e.getActionCommand().equalsIgnoreCase("increase X position")) {
      glEventListener.incXPosition();
    }
    else if (e.getActionCommand().equalsIgnoreCase("decrease X position")) {
      glEventListener.decXPosition();
    }
    else if(e.getActionCommand().equalsIgnoreCase("quit"))
      System.exit(0);
  }
```

Program listing 5.3: M03.java.

```
// ************************************************
/* INTERACTION
 *
 *
 */

 public void incXPosition() {
   xPosition += 0.5f;
   if (xPosition>5f) xPosition = 5f;
   updateX();
 }

 public void decXPosition() {
   xPosition -= 0.5f;
   if (xPosition<-5f) xPosition = -5f;
   updateX();
 }

 private void updateX() {
   translateX.setTransform(Mat4Transform.translate(xPosition,0,0));
   translateX.update(); // IMPORTANT - the scene graph has changed
 }
```

Program listing 5.4: Methods in M03_GLEventListener.java.

6 A robot

Program: Week6_3_scene_graph\M04.java. Two screen shots are shown in Figure 6.1. In this program, a robot is drawn. Buttons on the GUI can be used to make the robot perform actions (raise arms and lower arms) and to start and stop the animation – the robot swings its left arm forwards and backwards.

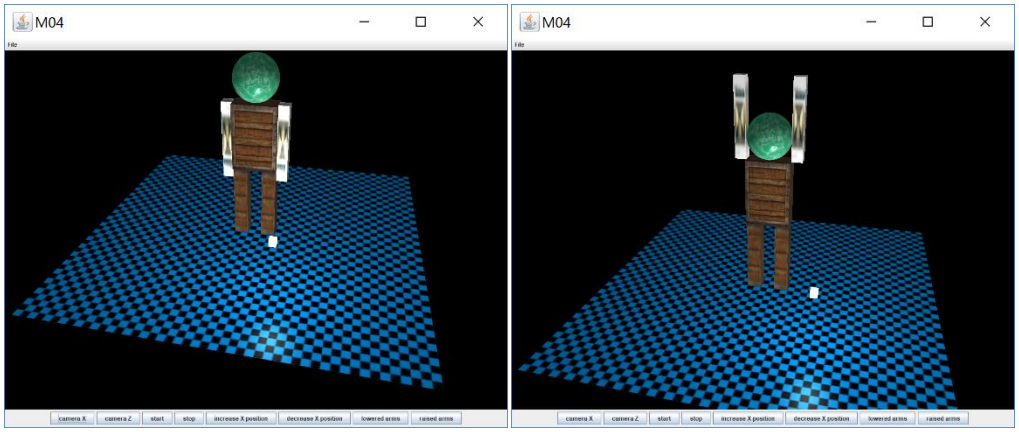

Figure 6.1: Output from M04.java

Exercise

1. Draw a scene graph for the robot in Figure 6.1. Then, compare your scene graph with the one in M04_GLEventListener.initialise().
2. Alter the program so that the right arm also swings backwards and forwards. It should swing forwards when the left arm swings backwards, and vice versa. The robot will look like it is marching.

7 Further thoughts

The initialise method is becoming quite long in the above programs. This is inevitable as the complexity of models increases and a scene graph is used. Separate classes could be used to deal with this. For example, a separate class could be created for the robot in Section 6. This would then have its own initialise method which would be called from the main initialise method in M04_GLEventListener.

The existing programs add a lot of attributes to the Mesh class, some of which it could be argued should not be a part of this class. For example, perspective, camera and light do not need to be part of the Mesh class. The Mesh should be concerned solely with Mesh properties. A better solution may be to move the perspective, camera and light attributes to the Shader class. A list of Shaders could be created. Then, a specific Shader could be attached to a Mesh class. The Mesh class could then use the Shader class to deal with setting the uniforms for perspective, camera view matrix and light. The same Shader could then be associated with different Mesh instances. In some ways this is much more flexible, as there are only so many different Shaders needed and different ones could be created for use during debugging. This is left as an exercise for you to consider.

Another thing to consider is the following statement from the OpenGL 3.3 specification:

> *"Shaders can declare named uniform variables, as described in the OpenGL Shading Language Specification. Values for these uniforms are constant over a primitive, and typically they are constant across many primitives. Uniforms are program object-specific state. They retain their values once loaded, and their values are restored whenever a program object is used, as long as the program object has not been re-linked."*
> [https://www.khronos.org/registry/OpenGL/specs/gl/glspec33.core.pdf]

This is saying that the value of a uniform in a Shader is persistent between uses of the same Shader. For example, consider that the light's values are set in the fragment shader. If the light never moves, nor switches on or off, then it is not necessary to keep setting the light's values every time the scene is rendered. A flag could be used in the main Java program so that once the uniform values for the light are set, they are not set again until the light changes. This would decrease the number of transfers of data from the CPU to the GPU, which would increase efficiency.

Another thing that might be considered in a larger system is to standardise the names of variables in the vertex and fragment shaders. We have done this to a certain extent, since we always use position, normal and texCoord as inputs to the vertex shader. However, we could also standardise the out variable names. Likewise, the names used in the fragment shader could be standardised. One other point to remember is that if a declared uniform is not used in either the vertex or fragment shader, then it is ignored (so attempting to bind a value to it may fail causing a GL error!). Another possible thing to consider for fragment shaders is to be able to make use of a variable number of texture maps. Again, standardisation of names can help here. Joey discusses this issue in his tutorial on models: https://learnopengl.com/#!Model-Loading/Mesh.

No doubt there are further things that could be considered to make the programs we have looked at more flexible and/or more efficient. However, we are now at a stage where fairly complex programs can be constructed. If you wish to learn more advanced OpenGL, then Joey's tutorials are a good place to start.