Part 2. Introducing Simple3D

Simple3D simplifies Java3D in a few different ways:

1) It builds the top-level scene graph for a program using Java 3D's SimpleUniverse class, and also adds a background, a set of lights, and an optional floor grid and x-, y-, z- axes. The camera is augmented to allow both mouse-based and key-based navigation.

2) The scene is displayed in a JFrame that uses a JOGL panel to hold the Canvas3D object (I went with JOGL rather than a JPanel since it seems more stable). The Swing coding also employs a few tricks to improve rendering stability across multiple monitors and graphics cards. In addition, a menu bar includes a item for taking screenshots, information about the camera controls, and a way to restore the camera to its starting position.

3) Simple3D contains an ever-growing collection of static methods for common tasks:

- scene graph node creation;
- applying colors and textures;
- loading, examining, and saving OBJ models;
- translating, rotating, and scaling;
- common forms of animation;
- using vertex and fragment shaders.

4) The Simple3D JAR includes several helper classes, some of which I've 'borrowed' from other people. These include ColorFactory by lobochief@users.sourceforge.net, InterpolatorData by Marco, and OBJWriter by Emmanuel Puybaret. Full details of my borrowings are included in the source code in the JAR. I'm also utilizing an updated version of Daniel Selman's j3dTree.jar to visualize scene graphs.

In the rest of this part, I'll introduce these features through five small examples, beginning with Earth.java which displays a rotating textured sphere, as pictured in Figure 1.

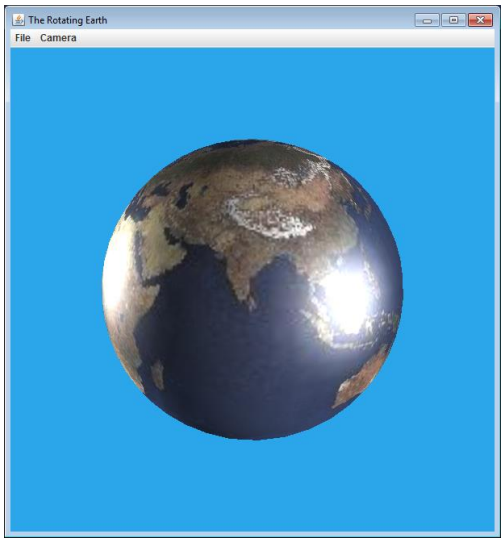

Figure 1. The Rotating Earth.

1. A World of Coding

Even a simple program like Earth.java requires a very large amount of boilerplate code to set up the scene graph infrastructure. Most textbooks and tutorials depict this as something like Figure 2.

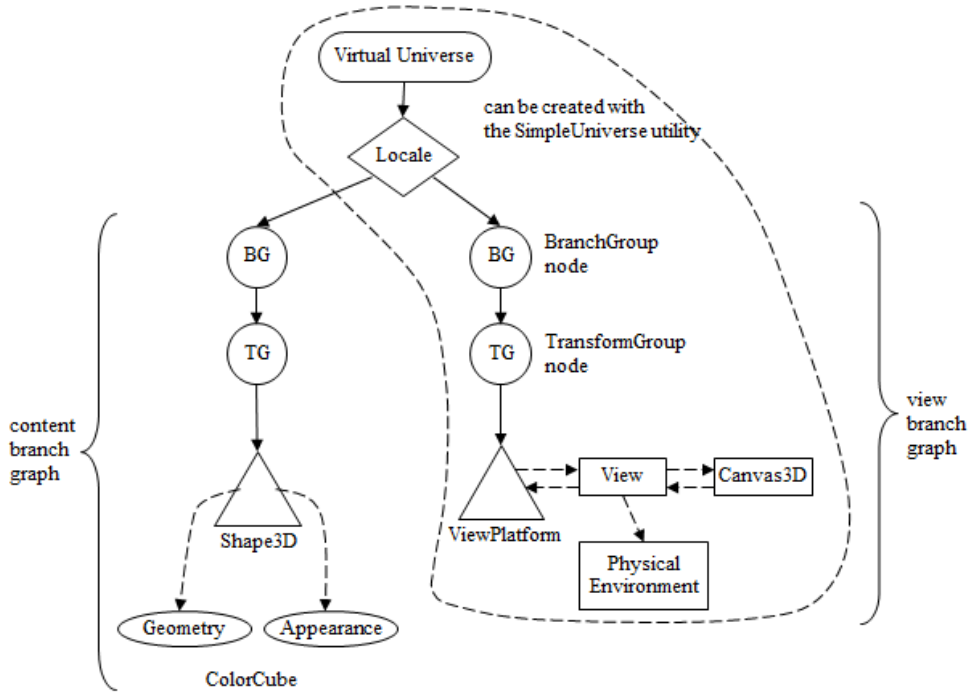

Figure 2. A Scene Graph for Displaying a Colored Cube.

The dotted line around the view branch refers to Java 3D's SimpleUniverse class which can generate a branch with a Canvas3D object suitable for a Swing application. However, even with this helper class, a scene graph for a typical application still looks like Figure 3.

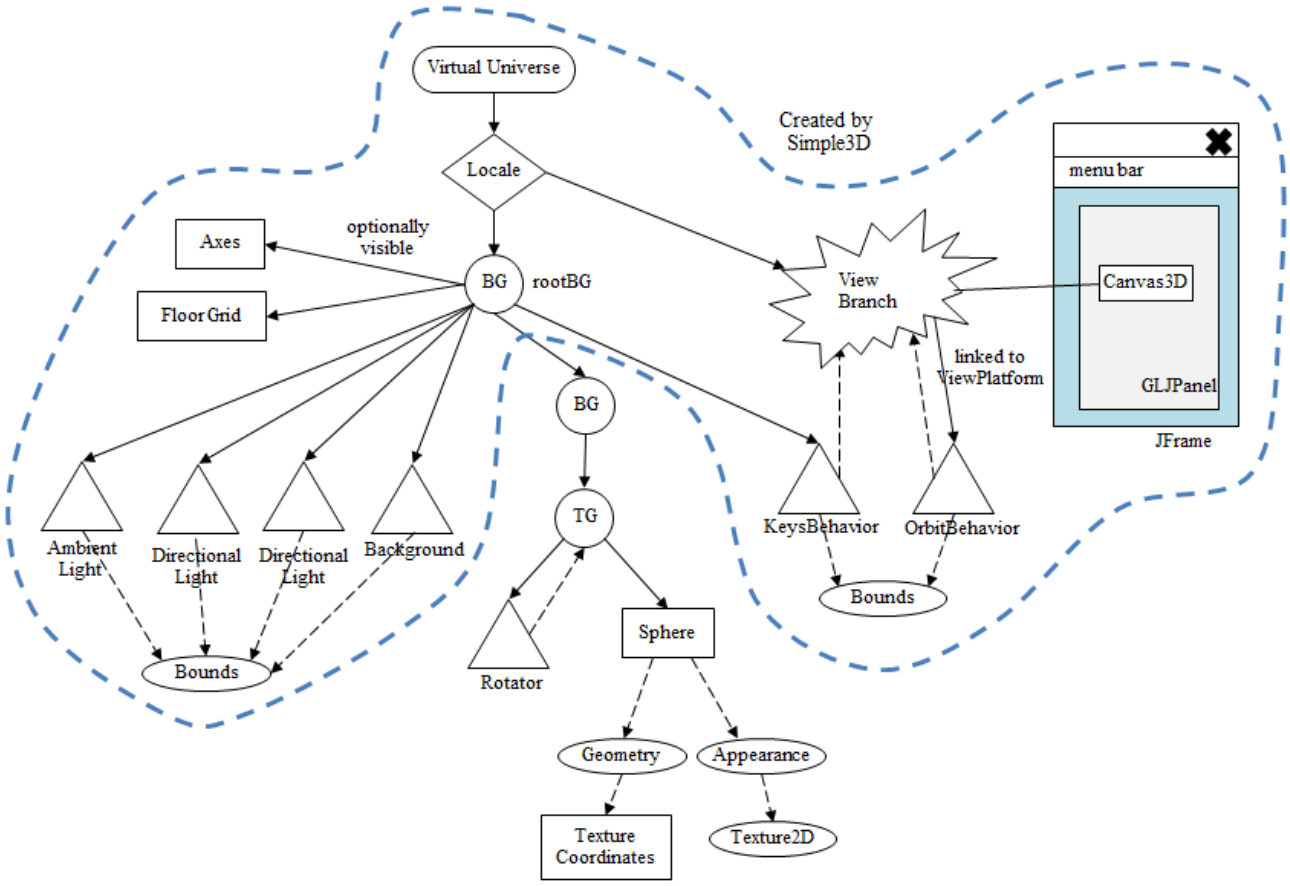

Figure 3. The Scene Graph for the Rotating Earth.

Almost every application needs several lights, a background, and a floor and/or axes. These latter elements are necessary so that objects in the scene can be accurately positioned and visually compared. Over on the view branch side, the built-in OrbitBehavior class is frequently utilized to move the camera around the scene with mouse clicks and drags, but there's often a need for additional keyboard-based controls, and Swing code for handling the Java 3D's Canvas3D object.

The blue dashed line in Figure 3 indicates that all of this is managed by Simple3D. The programmer only has to build the subgraph representing the things in the scene; in this case a rotating textured sphere. The complete code for Earth.java is:

```
public class Earth
{

  public static void main(String[] args)
  /*
      bg --> rotTG --> sphere
  */
  { Simple3D j3d = new Simple3D("The Rotating Earth");

    // a textured sphere
    Sphere sphere = Simple3D.texSphere(0.4f, "images/earth1.jpg");

    // rotating with a period of 4000ms
    TransformGroup rotTG = Simple3D.orbit(sphere, 4000);
```

```
    BranchGroup bg = Simple3D.createBG(rotTG);
    j3d.rootBG.addChild(bg);
  }  // end of main()


}  // end of Earth class
```

The program utilizes three static methods from Simple3D: texSphere() to make a textured sphere, orbit() to set up a rotation interpolator for the sphere, and createBG() to create a BranchGroup node to manage the interpolator's TransformGroup. This is nicely summarized by the scene graph comment at the start of main(). This branch is connected to the rootBG BranchGroup, the main connection point between Simple3D's graph and the user's work.

texSphere() is joined in Simple3D by texCylinder(), texCone(), and texBox() for texturing Java 3D's Primitive shapes, and there are general purpose color() and texture() methods for creating Appearance nodes for other shapes. These allow the intricacies of a how colors and textures are utilized in a shape's geometry and appearance to be hidden.

The orbit() method plays a similar role for the rotation interpolator and its Alpha node, allowing us to avoid having to think about animation details until later (probably in Part 5).

Simple3D includes several other node creation methods apart from createBG() (e.g. for TransformGroup, Shape3D, Text3D) which hide messy details such as the setting of capability bits.

2. Visualizing the Scene Graph

From my own experiences when teaching Java 3D labs, it's clear that student's like the conceptual simplicity of the scene graph, but are put off by the size of actual real-life graphs. This is readily apparent when I've employed Daniel Selman's excellent j3dTree library to display those graphs.

The ViewEarthSG.java program below uses a (slightly modified) version of j3dTree to generate a diagram for the Earth.java example.

```
  public static void main(String[] args)
  /*
      bg --> rotTG --> sphere
  */
  {
    Simple3D j3d = new Simple3D("View Earth's Scenegraph",
                                                  false, true);
    Java3dTree j3dTree = new Java3dTree();
                // create a window for the scene graph

    // a textured sphere
    Sphere sphere = Simple3D.texSphere(0.4f, "images/earth1.jpg");

    // rotating with a period of 4000ms
    TransformGroup rotTG = Simple3D.orbit(sphere, 4000);
```

```
    BranchGroup bg = Simple3D.createBG(rotTG);

    j3dTree.setCapabilities(bg);    // set graph capabilities

    j3d.rootBG.addChild(bg);

    j3dTree.showGraph(j3d.su);     // fill the display
  }  // end of main()
```

J3dTree instantiates a JFrame consisting of a JTree hierarchy for the scene graph at the top and a text area listing the capability settings for the currently selected node. Figure 4 shows the situation when the rootBG BranchGroup node has been chosen.

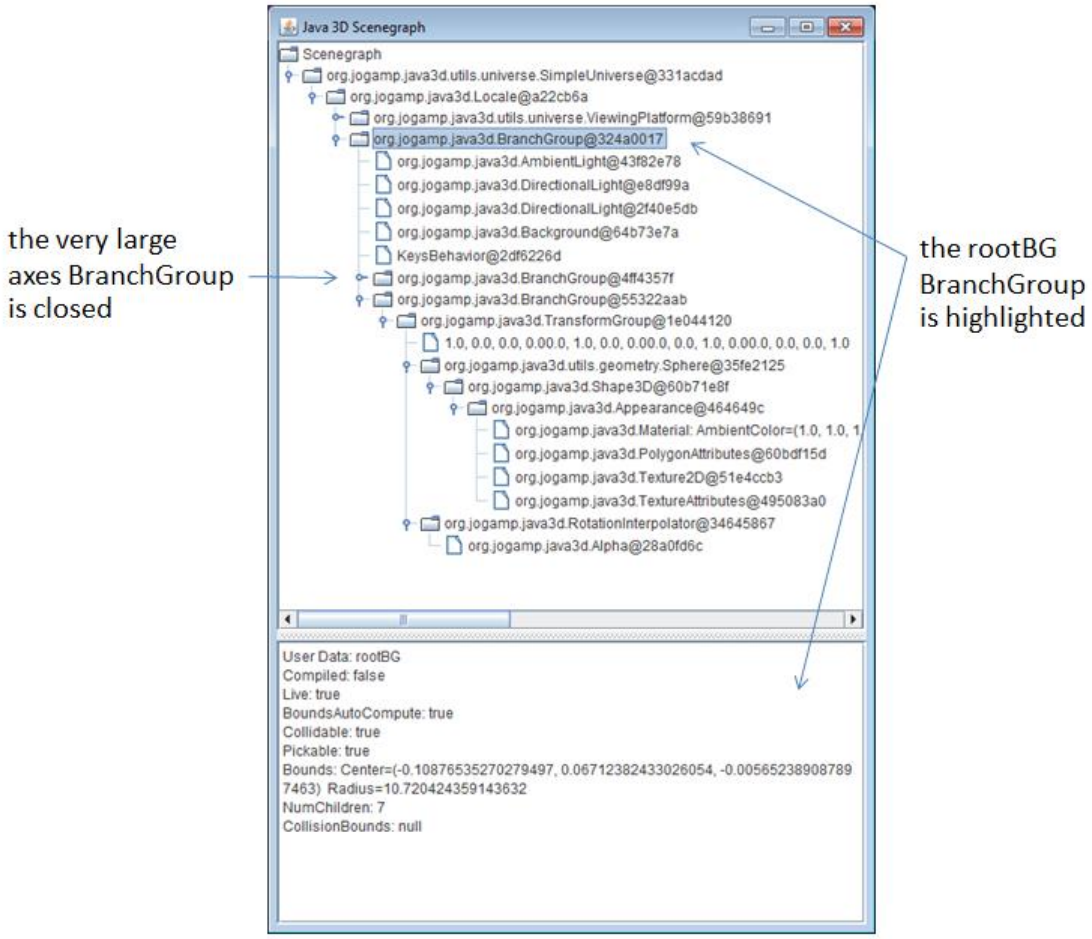

Figure 5. The j3dTree Window.

The advantage, and drawback, of j3dTree is that it shows *all* of a scene graph, including the view branch and the content branches generated by Simple3D.

The Simple3D constructor in this program is called like so:

```
Simple3D j3d = new Simple3D("View Earth's Scenegraph", false, true);
```

The last two arguments specify whether to display the floor grid and the axes. Figure 6 shows the result.

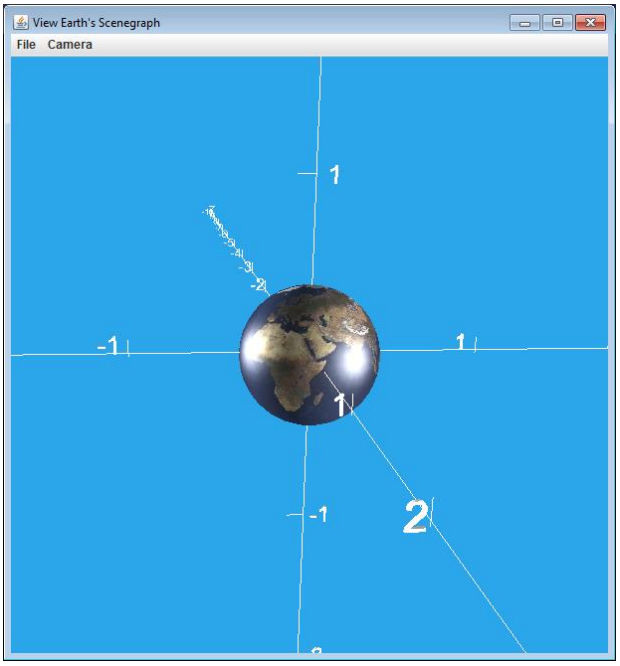

Figure 6. A Rotating Earth with Axes.

The axes contribute an enormous new subbranch to the j3dTree display, which can make it hard to find the few lines assigned to the textured sphere. However, the axes branch can be closed, as in Figure 5.

3. Positioning Objects

The axes displayed by ViewEarthSG.java make it much more obvious that the sphere is rotating clockwise around the y-axis, centered on the origin. It also emphasizes that Java 3D employs a "right-hand" rule for rotations as illustrated by Figure 7.

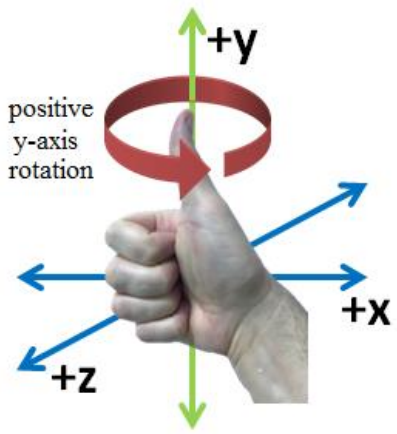

Figure 7. The Right-Hand Rule is Okay!

Specifying other axial rotations is dealt with by including a third argument in Simple3D.orbit(), as in:

```
TransformGroup rotTG = Simple3D.orbit(sphere, 4000, Simple3D.X_AXIS);
```

which will make the sphere rotate towards the camera around the x-axis.

Moving the sphere away from the origin is a little more complicated, since it introduces the importance of the graph hierarchy in determining how multiple transformations (i.e. a rotation and a translation) are combined.

SunEarth.java has the Earth rotate clockwise around a Sun positioned at the origin. Figure 8 shows the astronomical magic in action:

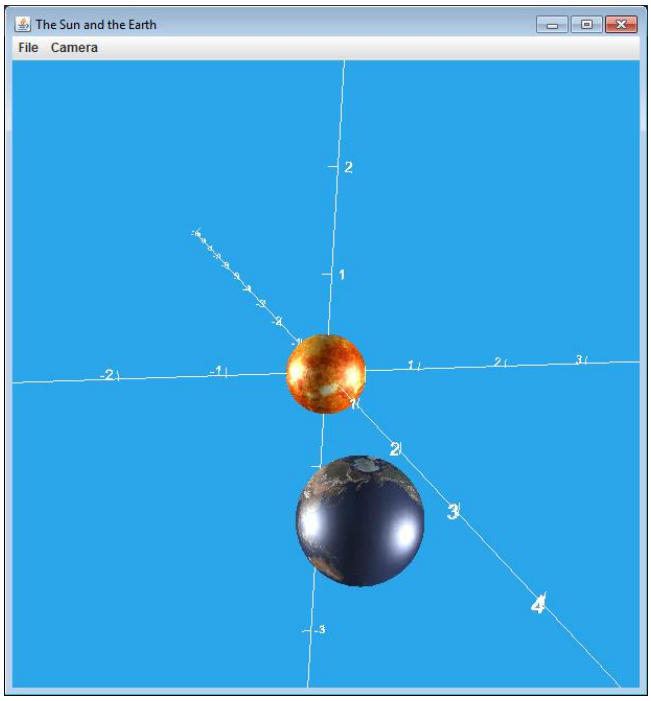

Figure 8. The Wonder of the Heavens.

The code:

```
  public static void main(String[] args)
  /*
       sceneBG -->  rotTG --> posTG --> earth
                      |
                      ----> sun
  */
  {
    Simple3D j3d = new Simple3D("The Sun and the Earth",
                                        0, 1, 10, false, true);

    // build the shapes
    Sphere earthSphere = Simple3D.texSphere(0.4f,
                                              "images/earth1.jpg");
```

```
    Sphere sunSphere = Simple3D.texSphere(0.4f, "images/sun.jpg");

    // position the Earth along the -z axis
    TransformGroup posTG = Simple3D.setTranslation(0,0,-3);
    posTG.addChild(earthSphere);   // posTG --> earth

    // make the earth rotate with a period of 4000ms at posTG
    TransformGroup rotTG = Simple3D.orbit(posTG, 4000);

    // build scene
    BranchGroup sceneBG = new BranchGroup();
    sceneBG.addChild(sunSphere);
    sceneBG.addChild(rotTG);

    j3d.rootBG.addChild(sceneBG);
  }  // end of main()
```

The importance of the hierarchy is illustrated by the scene graph comment – the Earth is moved to (0,0,-3) *before* the rotation is applied. The code is easily changed to reverse the order of these operations, and the Earth's orbit changes accordingly.

I'll fully explore the consequences of the scene graph hierarchy for positioning and rotating objects in Part 3. It's definitely an issue that confuses new Java 3D programmers, but Simple3D helps by hiding the irrelevant parts of the larger graph, and by offering a range of methods for simplifying translations, rotations, and scaling. These methods fall into three broad groups:

1. those that create a new TransformGroup: the "set" methods;
2. those that overwrite an existing TransformGroup: "To" methods;
3. those that incrementally change an existing one: "By" methods;

In all cases, the Transform3D object nestled inside the TransformGroup is hidden.

4. Please, No More Spheres

Simple3D.loadModel() loads a Wavefront OBJ model, along with its associated textures. For example, the following command line call to LoadModel.java:

```
> run LoadModel models/house/home.obj
```

loads the model shown in Figure 9.

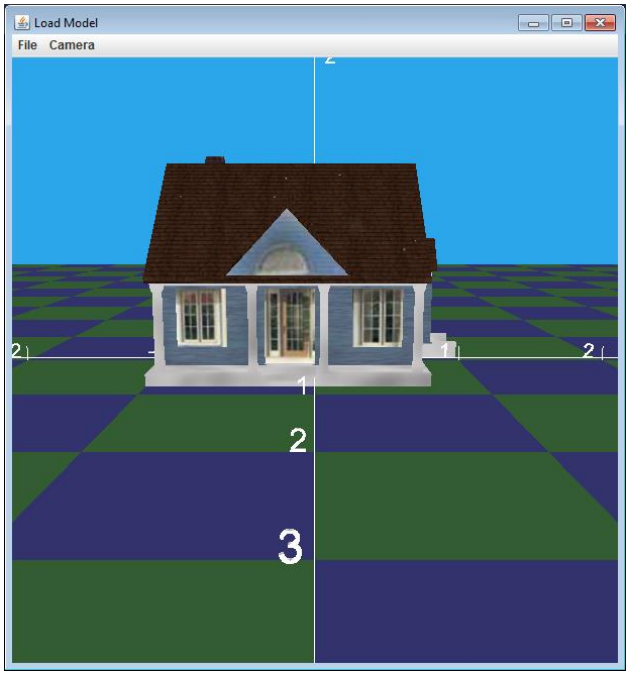

Figure 9. Little House on the Floor Grid.

The program uses the version of the Simple3D constructor that display a green-and-blue checkered floor grid along with the axes:

```
  public static void main(String[] args)
  {
    if (args.length != 1) {
      System.out.println("Usage: run LoadModel <filename>");
      System.exit(1);
    }

    Simple3D j3d = new Simple3D("Load Model", true, true);

    BranchGroup modelBG = Simple3D.loadModel(args[0]);

    // Simple3D.colorShape(modelBG, "blue");
    // Simple3D.textureShape(modelBG, "images/wood.jpg");

    if (modelBG != null)
      j3d.rootBG.addChild(modelBG);
  }
```

loadModel() reads OBJ and MTL files, and also looks for a user-defined "Zero" text file. "homeZero.txt" for the house is:

```
// House positioning at (0,0,0)
pos: 0 0.7 0
rots: -90 0 0
scale:  1
```

It specifies how the model should be positioned, rotated, and scaled before it's made visible in the scene. In this case, the house is moved 0.7 units up the y-axis, rotated by 90 degrees counterclockwise around the x-axis, and its scaling is left unchanged. The effect of these operations can be judged by 'hiding' the file from loadModel() by renaming it. The house is then positioned as in Figure 10.

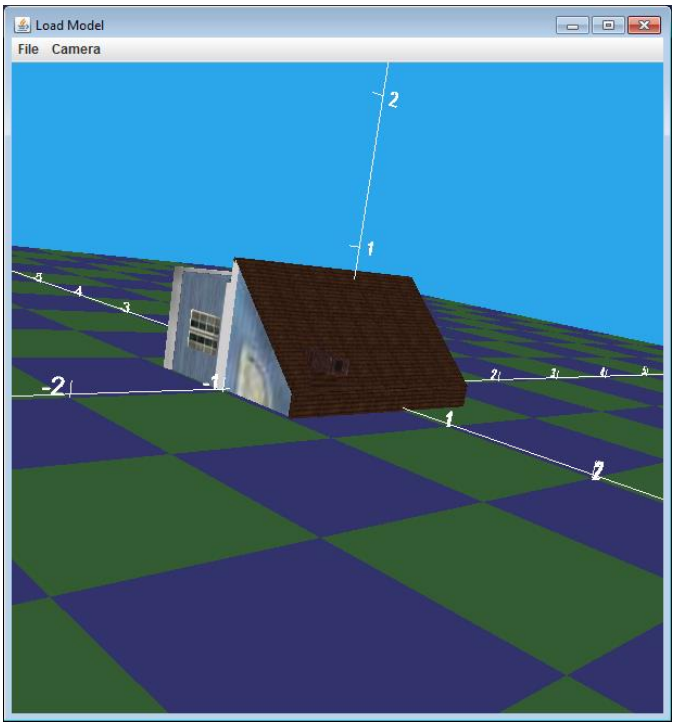

Figure 10. Little House Under the Prairie.

Why Wavefront OBJ?

The Wavefront OBJ format is even older than Java 3D, and naturally lacks many modern graphics features. These 'drawbacks' actually make OBJ a good choice for beginner Java 3D users. Java 3D comes with built-in support for loading OBJ files, although some care must be taken to ensure that a model's associated MTL data is rendered correctly (these issues are taken care of by loadModel()). Also, the OBJ and MTL formats are so simple that both are text-based. This makes it easy to read and manually edit a file, and thereby see how vertices, faces, normals, texture coordinates, and different forms of lighting work. In addition, the OBJ format has been around for so long, that just about every graphics tool supports it. For instance, on Windows 10 you can examine a file with the "3D Viewer" app, or install a freeware tool such as MeshLab.

The models/shapes folder in the Simple3D download includes OBJ files for several geometric shapes. pyramid.obj is the simplest:

```
# OBJ file created by ply_to_obj.c
#
g Object001
```

```
v   0   0   0
v   1   0   0
v   1   1   0
v   0   1   0
v   0.5   0.5   1.6

f   5   2   3
f   4   5   3
f   1   3   2
f   5   1   2
f   4   1   5
f   1   4   3
```

This file is displayed by MeshLab on the left of Figure 11, and the loadModel() rendering is on the right (with the floor grid turned off).

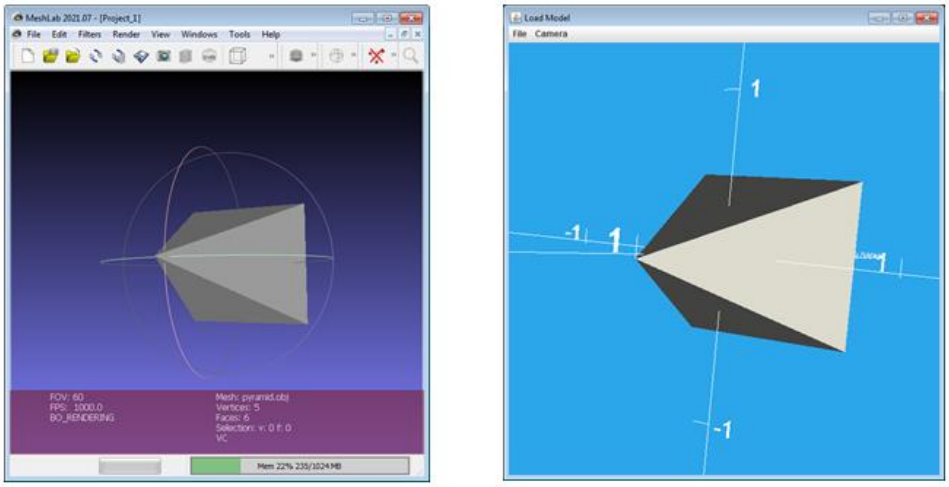

Figure 11. MeshLab and LoadModel Views of pyramid.obj.

Simple3D contains methods for colorizing or adding a texture to an unadorned model.

For example, if the line:

```
Simple3D.textureShape(modelBG, "images/wood.jpg");
```

is uncommented in LoadModel.java then the pyramid will be displayed as in Figure 12.

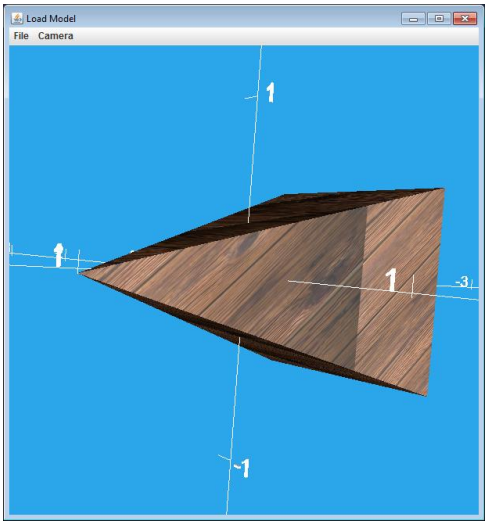

Figure 12. A Wooden Pyramid.

Part 4 will go into more detail on creating shapes in Java 3D, both programmatically, and with the OBJ format. One particularly useful part of Simple3D is the OBJWriter class, written by Emmanuel Puybaret as part of his Sweet Home 3D application (http://www.sweethome3d.com/). It allows a scene graph shape to be saved as an OBJ model.

5. Finish with some Glitz

Most of the Java 3D related textbooks were published in the mid-2000s, so don't talk about its support for vertex and fragment shaders which came later. Simple3D offers a makeShaderApp() method which takes the names of vertex and fragment shader files, and an optional collection of attribute values, and compiles and executes the shaders, generating an Appearance node.

Figure 13 shows a teapot model before and after applying a "dimple" shader effect.

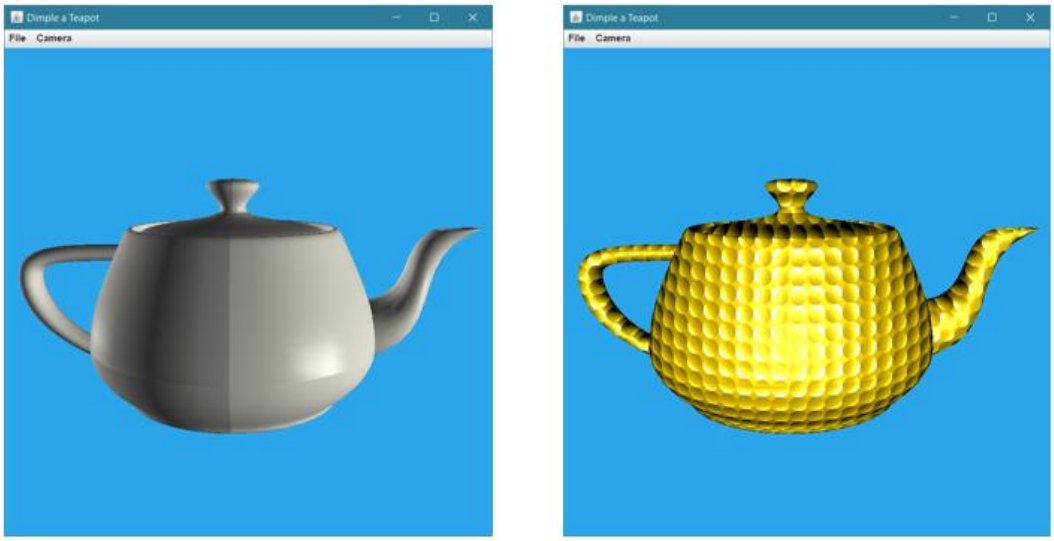

Figure 13. A Teapot before and after.

The code for TeapotDimples.java:

```
public class TeapotDimples
{
  private static final Color3f GOLD = ColorFactory.getColor("gold");
  private static final Color3f BLUE = ColorFactory.getColor("blue");


  public static void main(final String args[])
  {
    Simple3D j3d = new Simple3D("Dimple a Teapot");

    BranchGroup modelBG = Simple3D.loadModel("models/teapot.obj");

    // attributes for the dimple shaders
    Map<String,Object> dimpleAttrs = Map.of(
      "Density",        16.0f,
      "Size",           0.25f,
      "LightPosition",  new Point3f(0.0f, 0.0f, 0.5f),
      "Color",          GOLD   // BLUE
    );
    ShaderAppearance app = Simple3D.makeShaderApp(
                                        "shaders/dimple.vert",
                                        "shaders/dimple.frag",
                                        dimpleAttrs);
/*
    ShaderAppearance app = Simple3D.makeShaderApp(
                                        "shaders/polkadot3d.vert",
                                        "shaders/polkadot3d.frag");
*/
    Simple3D.applyShader(modelBG, app);
    j3d.rootBG.addChild(modelBG);
```

```
  } // end of main()

}  // end of TeapotDimples
```

The program includes a commented out line that adds polka dots to the teapot instead of dimples.

Several more shaders are available in the Simple3D shaders/ folder which I shamelessly lifted from the examples included in Java 3D. If you want to learn about shaders, then `https://learnopengl.com/Getting-started/Shaders` is a good place to start.

6. Onwards and Upwards

Parts 3 will look at how Simple3D simplifies the problem of combining translations, rotations, and scaling, especially when building composite objects from Java's 3D primitive shapes (i.e. using Cylinder, Cone, Box, and Sphere).

Part 4 will focus on creating shapes from Java 3D Geometry classes (e.g. the various subclasses of GeometryArray), and by utilizing the Wavefront OBJ format.

Part 5 will probably describe different ways of implementing animation, leading to a discussion of Behaviors.

I'm not certain about after part 5, but I'll probably look at various (recreational) math topics with a 3D emphasis – vectors, 3D curves and surfaces, turtle graphics, tiling, and building with blocks. Naturally, if you, dear reader, have any suggestions, please send me an e-mail (at `ad@fivedots.coe.psu.ac.th`).