Learning jME

This will provide all you need to get started with jME. It assumes you have jME installed. If not, you can look to this installation guide. As a supplement to this guide, you can read the documentation at jME's documentation link, paying special attention to jME's wiki. Finally, never be afraid to ask questions (no matter how simple) at jME's forum. All of the code you see here is done using jME's latest CVS version. Also, all the source code here is in the CVS under package jmetest.TutorialGuide Please post any corrections on the jME forums, or email them to me directly.

Updated:

December 5, 2004

By

Jack Lindamood AKA Cep21

Email:

cep221@gmail.com

Contents

1) Hello World

Here we'll learn the basics of creating a jME program by exploring SimpleGame, Box, and rootNode.

2) Hello Node

This program introduces Nodes, Bounding Volumes, Sphere, Colors, Translation, and Scaling. You will learn basic scene graph manipulation.

3) Hello TriMesh

This program introduces the TriMesh class. You will learn how to make your own objects from scratch.

4) Hello States

This program introduces MaterialState, TextureState, LightState, and PointLight. You will learn how to assign states such as materials and textures to your scene.

5) Hello KeyInput

This program introduces KeyBindingManager, texture wrapping, and how to change TriMesh data after it is assigned. You will learn how to bind keys to actions.

6) Hello Animation

This program introduces LightNode and using Controllers. The controller we will use is called SpatialTransformer. You will learn how to do animation and lighting

7) Hello ModelLoading

This program introduces JmeBinaryReader, FileConverter, BinaryToXML, and LoggingSystem. You will learn how to convert formats to a jME scene graph.

8) Hello MousePick

This program introduces AbsoluteMouse, AlphaState, InputSystem, Ray, and Intersection. You will learn how to create your own mouse icon and determine if the user is clicking an item on screen.

9) Hello Keyframes

This program introduces KeyframeController. You will learn how to make your own animations and how to manipulate vertex information after an object is created.

10) Hello Intersection

This program introduces SoundNode, Skybox, Text, and SoundAPIController. You will learn how to play sounds, create text, and make your own controllers and input handlers. This also introduces rudimentary collision detection.

11) Hello SimpleGame

This program introduces WireframeState, Timer, draw(), camera creation, and displaysystem creation. You will learn how to create and customize your own SimpleGame class. It will also take away some of the mystery of SimpleGame and what it does.

12) Hello LOD

This program introduces AreaClodMesh, BezierCurve, CurveController, and CameraNode. You will learn how to use AreaClodMesh to increase FPS, as well as how to move the camera along a curved path.

13) Hello Terrain

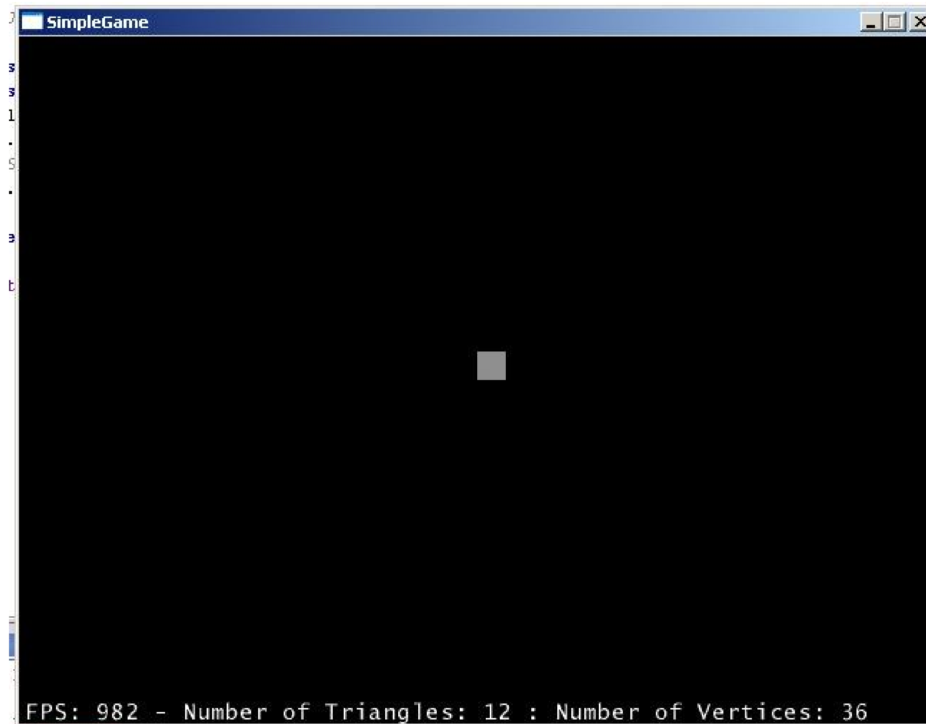
This program introduces jME's terrain utility classes. You will learn how to use ProceduralTextureGenerator, ImageBasedHeightMap, MidPointHeightMap, and TerrainBlock. All of this will allow you to create nice looking terrain with little effort.

1. Hello World

Here we'll learn the basics of creating a jME program, by exploring SimpleGame, Box, and rootNode.

OK, let's just dive in. Here's as basic a program as you can get:

```
import com.jme.app.SimpleGame;
import com.jme.scene.shape.Box;
import com.jme.math.Vector3f;

/**
 * Started Date: Jul 20, 2004<br><br>
 * Simple HelloWorld program for jME
 *
 * @author Jack Lindamood
 */
public class HelloWorld extends SimpleGame{
    public static void main(String[] args) {
        HelloWorld app = new HelloWorld();    // Create Object
        // Signal to show properties dialog
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();    // Start the program
    }

    protected void simpleInitGame() {
      // Make a box
      Boxb = new Box("Mybox",
             new Vector3f(0,0,0),
             new Vector3f(1,1,1));
      rootNode.attachChild(b);    // Put it in the scene graph
    }
}
```



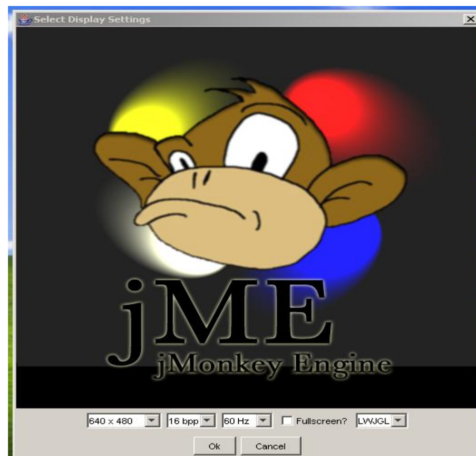
Pretty short, right? The real meat of our program begins with the following:

```
public class HelloWorld extends SimpleGame{
```

SimpleGame does a lot of initialization for us behind our back. If you really want to, you can look at its code, but for now just realize that it creates all the basics needed for rendering. You'll want to extend it on all your beginning programs.

```
app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
```

You know that picture of a monkey you see when the program is first run – the one that lets you select what resolution you want to run at?



Well, this command makes it show. As the name suggests, it always shows the properties dialog on every run. You'll never see that dialog box if you change it to the following:

```
SimpleGame.NEVER_SHOW_PROPS_DIALOG
```

Not too difficult.

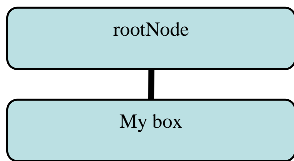
```
        app.start();    // Start the program
```

The function start() is basically an infinite while loop. First it initializes the jME system, and then the while loop does two things per iteration: first, it tells everything in your game that it needs to move, and secondly, it renders everything. Basically, it gets the game going.

```
      protected void simpleInitGame() {
        // Make a box
        Boxb = new Box("Mybox",
             new Vector3f(0,0,0),
             new Vector3f(1,1,1));
        rootNode.attachChild(b);    // Put it in the scene graph
      }
```

The function simpleInitGame() is abstract in SimpleGame, so you're forced to implement it every time you extend SimpleGame. Just by looking at the code, it's obvious that two things happen. First I make a box (it's the thing you saw on the screen). Second, I attach the box to the root of my scene graph. The object rootNode is of class Node which is created by SimpleGame for you. You'll attach everything to it or one of its children. Notice I gave *b* 3 parameters: a string and two Vector3f objects. Every Node or Box or Circle or Person or ***anything*** in your scene graph will have a name. Usually you want the name to be specific for each object. I called this one "My box", but really you could have called it anything. The next two parameters specify the corners of the Box. It has one corner at the origin, and another at x=1, y=1, z=1. Basically, it's a unit cube.

OK, I've created the Box, but I have to tell it I want it rendered, too. That's why I attach it to the rootNode object. Your scene graph basically looks like this:



The object rootNode is at the top and "My box" is below it. So, when SimpleGame tries to draw rootNode it will try to draw "My box" as well. That's it! Now, on to something more complex.

Challenge:

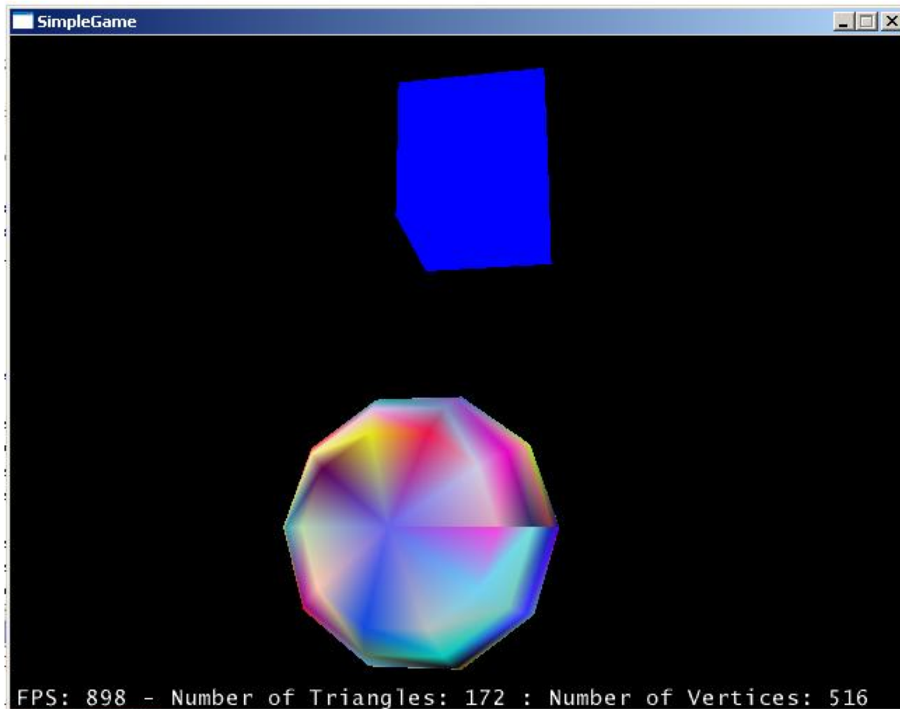
Try to draw the letter E with box objects.

2. Hello Node

This program introduces Nodes, Bounding Volumes, Sphere, Colors, Translation and Scaling.

```
import com.jme.app.SimpleGame;
import com.jme.scene.Node;
import com.jme.scene.shape.Box;
import com.jme.scene.shape.Sphere;
import com.jme.math.Vector3f;
import com.jme.bounding.BoundingSphere;
import com.jme.bounding.BoundingBox;
import com.jme.renderer.ColorRGBA;

/**
 * Started Date: Jul 20, 2004<br><br>
 *
 * Simple Node object with a few Geometry manipulators.
 *
 * @author Jack Lindamood
 */
public class HelloNode extends SimpleGame {
    public static void main(String[] args) {
        HelloNode app = new HelloNode();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        Box b=new Box("My Box",new Vector3f(0,0,0),new Vector3f(1,1,1));
        // Give the box a bounds object to allow it to be culled
        b.setModelBound(new BoundingSphere());
        // Calculate the best bounds for the object you gave it
        b.updateModelBound();
        // Move the box 2 in the y direction up
        b.setLocalTranslation(new Vector3f(0,2,0));
        // Give the box a solid color of blue.
        b.setSolidColor(ColorRGBA.blue);

        Sphere s=new Sphere("My sphere",10,10,1f);
        // Do bounds for the sphere, but we'll use a BoundingBox this time
        s.setModelBound(new BoundingBox());
        s.updateModelBound();
        // Give the sphere random colors
        s.setRandomColors();

        // Make a node and give it children
        Node n=new Node("My Node");
        n.attachChild(b);
        n.attachChild(s);
        // Make the node and all its children 5 times larger.
        n.setLocalScale(5);

        // Remove  lighting for rootNode so that it will use our
        //basic colors.
        lightState.detachAll();
```

```
        rootNode.attachChild(n);
    }
}
```



The first new thing you see is:

```
        // Give the box a bounds object to allow it to be culled
        b.setModelBound(new BoundingSphere());
```

Bounding Volumes (such as BoundingSphere and BoundingBox) are the key to speed in jME. What happens is your object is surrounded by a bounds, and this bounds allows jME to do a quick, easy test to see if your object is even viewable. So in your game, you'll make this really complex person and surround him with a sphere, for instance. A sphere is a really easy object mathematically, so jME can tell very fast if the sphere around your object is visible. If the sphere surrounding your guy isn't visible (which is an easy test), then jME doesn't even bother trying to draw your really large, very complex person. In this example, I surround my box with a sphere, but it makes more sense to surround it with a BoundingBox. Why? Well trying to draw a sphere around a box isn't as tight a fit as a Box around a Box (obviously). I just use a sphere as an example:

```
        // Calculate the best bounds for the object you gave it
        b.updateModelBound();
```

When I start, I give my Box object bounds, but it's an empty bounds. I now have to "update" the bounds so that it surrounds the object correctly. Now we'll move it up:

```
        // Move the box 2 in the y direction up
```

```
		b.setLocalTranslation(new Vector3f(0,2,0));
```

Now we start moving our object. Vector3f has the format x, y, z, so this moves the object 2 units up. Remember how our Box looks above the Sphere? That's because I moved it up some. If I had used *Vector3f(0,-2,0)* then it would have moved down two. Moving objects in jME is extremely simple. Now color it:

```
		// Give the box a solid color of blue.
		b.setSolidColor(ColorRGBA.blue);
```

Just looking at the function, you can tell I give my box a solid color. As you can guess by the color type, the color is blue. ColorRGBA.red would make it…. red! You can also use new ColorRGBA(0,1,0,1). These four numbers are: Red/Green/Blue/Alpha and are percentages from 0 to 1. Alpha is lingo for how opaque it is. (The opposite of opaque is transparent. Opaque would be a wall; not opaque would be a window.) So, an alpha of 0 would mean we couldn't see it at all, while .5 would mean we see half way thru it. New ColorRGBA(0,1,1,1) would create a color that is green and blue. Now let's make a sphere:

```
		Sphere s=new Sphere("My sphere",10,10,1f);
```

jME can't draw curves. It draws triangles. It's very hard to draw a circle with triangles, isn't it? Because of this, you have to get as close to a circle as you can. The first two numbers represent that closeness. If you change 10, 10 to 5,5 you'll see a pretty bad looking sphere. If you make them 30, 30 you'll see a very round looking sphere but the problem is that it's made of a lot of triangles. More triangles mean slower FPS. The last number, 1, is the radius of the sphere. Just like my Node and my Box, Sphere needs a name which I give "My sphere". For our sphere, let's be a bit prettier with colors:

```
		// Give the sphere random colors
		s.setRandomColors();
```

This gives each vertex of the sphere a random color, which creates that psychodelic effect you saw. Now on to the node:

```
		// Make a node and give it children
		Node n=new Node("My Node");
		n.attachChild(b);
		n.attachChild(s);
```

Instead of attaching our box and sphere to the rootNode, we create a new node *n* to attach them too. I also wanted *n* (and all its children) to be five times bigger, so I did this:

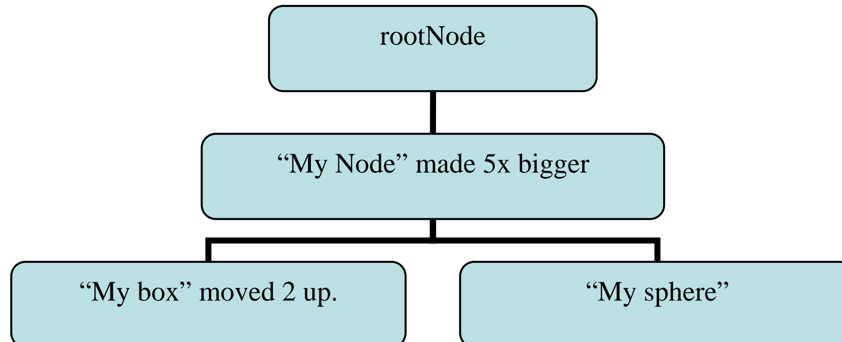
```
		// Make the node and all its children 5 times larger.
		n.setLocalScale(5);
```

;

You'll notice it's similar to the function call setLocalTranslation in name. There's also a setLocalRotation, which we'll get into later. As the name suggests, this makes everything five times larger. Finally, you'll notice a strange line at the end of my code:

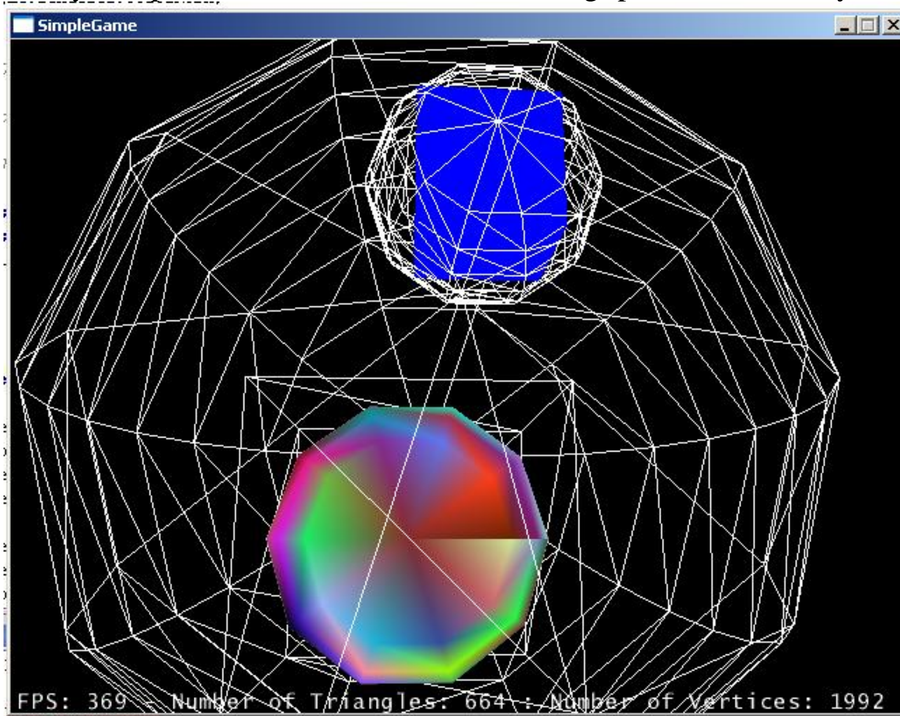
```
        // Remove  lighting for rootNode so that it will use our
        //basic colors.
        lightState.detachAll();
```

You see, per vertex colors (which is what I'm doing here) isn't really lighting. It's a cheap, easy way to create colors. You'll notice there's no shading for these colors at all. If you don't use this line, jME won't try to use per vertex colors. The scene graph looks like this:



You'll notice that because "My box" and "My sphere" are children of "My Node", they are made five times bigger, as well. If I move "My Node" down 10, then "My Box" and "My sphere" would move down 10, as well. This is the ease of making a game with a scene graph.

While you're running the program, press "B" to see the bounding in action. It is a neat way to visualize the parent/child scene graph relationship. You can see the BoundingSphere around your box, the BoundingBox around your sphere, and the automatic BoundingSphere around "My node":



Challenge:

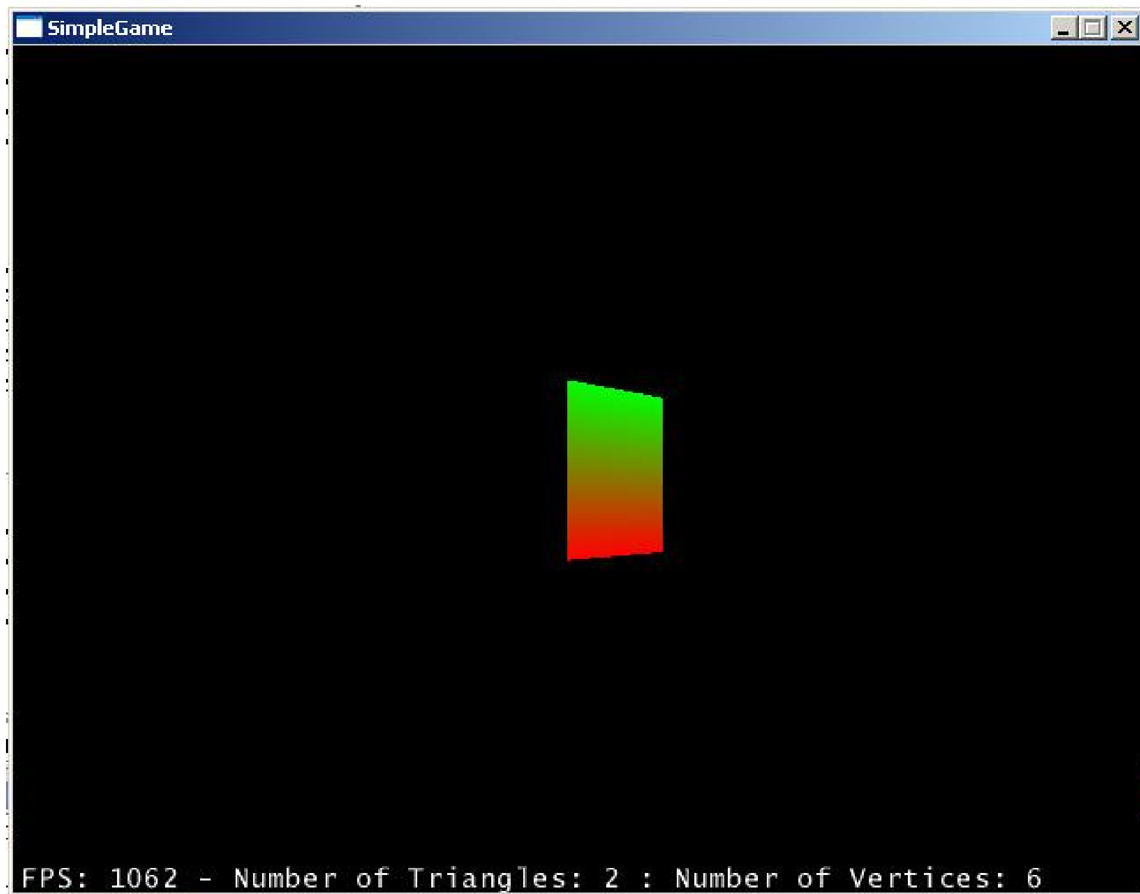
Try to draw an X with box objects.

3) Hello TriMesh

This program introduces the TriMesh class and how to make your own objects from scratch. We will make a flat rectangle:

```
import com.jme.app.SimpleGame;
import com.jme.scene.TriMesh;
import com.jme.math.Vector3f;
import com.jme.math.Vector2f;
import com.jme.renderer.ColorRGBA;
import com.jme.bounding.BoundingBox;

/**
 * Started Date: Jul 20, 2004<br><br>
 *
 * Demonstrates making a new TriMesh object from scratch.
 *
 * @author Jack Lindamood
 */
public class HelloTriMesh extends SimpleGame {
    public static void main(String[] args) {
        HelloTriMesh app = new HelloTriMesh();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        // TriMesh is what most of what is drawn in jME actually is
        TriMesh m=new TriMesh("My Mesh");

        // Vertex positions for the mesh
        Vector3f[] vertexes={
            new Vector3f(0,0,0),
            new Vector3f(1,0,0),
            new Vector3f(0,1,0),
            new Vector3f(1,1,0)
        };

        // Normal directions for each vertex position
        Vector3f[] normals={
            new Vector3f(0,0,1),
            new Vector3f(0,0,1),
            new Vector3f(0,0,1),
            new Vector3f(0,0,1)
        };

        // Color for each vertex position
        ColorRGBA[] colors={
            new ColorRGBA(1,0,0,1),
            new ColorRGBA(1,0,0,1),
            new ColorRGBA(0,1,0,1),
            new ColorRGBA(0,1,0,1)
        };

        // Texture Coordinates for each position
```

```
        Vector2f[] texCoords={
            new Vector2f(0,0),
            new Vector2f(1,0),
            new Vector2f(0,1),
            new Vector2f(1,1)
        };

        // The indexes of Vertex/Normal/Color/TexCoord sets.  Every 3
        // makes a triangle.
        int[] indexes={
            0,1,2,1,2,3
        };

        // Feed the information to the TriMesh
        m.reconstruct(vertexes,normals,colors,texCoords,indexes);

        // Create a bounds
        m.setModelBound(new BoundingBox());
        m.updateModelBound();

        // Attach the mesh to my scene graph
        rootNode.attachChild(m);

        // Let us see the per vertex colors
        lightState.setEnabled(false);
    }
}
```



The first new thing here is:

```
        // TriMesh is what most of what is drawn in jME actually is
        TriMesh m=new TriMesh("My Mesh");
```

WARNING: *Do not ever use the constructor new TriMesh(). Always use new TriMesh("String name"). The empty constructor is for internal use only and will not render correctly. The same goes for Node, Box, Sphere, and anything else that extends com.jme.scene.Spatial;*

If you were to look at the class header for the Box and Sphere class we were using, you'd see
`public class Box extends TriMesh`
and
`public class Sphere extends TriMesh`

TriMesh is the parent class of both Box and Sphere. They are made from it. You'll notice that most things actually drawn are from TriMesh.

```
        // Vertex positions for the mesh
```

```
        Vector3f[] vertexes={
            new Vector3f(0,0,0),
            new Vector3f(1,0,0),
            new Vector3f(0,1,0),
            new Vector3f(1,1,0)
        };
```

This creates positions for the corners of the rectangle we're about to make. Now let's give each vertex a normal:

```
        // Normal directions for each vertex position
        Vector3f[] normals={
            new Vector3f(0,0,1),
            new Vector3f(0,0,1),
            new Vector3f(0,0,1),
            new Vector3f(0,0,1)
        };
```

The normal for vertex[0] is normal[0], just like the normal for vertex[i] is normal[i]. Normals are a very common 3D graphics concept. For more information on how normals work with triangles, please visit jME's math links. Basicly, they point in a direction where the light is its brightest (but not we don't use lighting at all in this example). After normals, each vertex can have a color:

```
        // Color for each vertex position
        ColorRGBA[] colors={
            new ColorRGBA(1,0,0,1),
            new ColorRGBA(1,0,0,1),
            new ColorRGBA(0,1,0,1),
            new ColorRGBA(0,1,0,1)
        };
```

In the last program, we assigned a solid blue color to every vertex of our box (ColorRGBA.blue which is the equivalent of new ColorRGBA(0,0,1,1)). In this program, we give the first two vertexes the color red, and the last two green. You'll notice the first two vertex positions are vertexes[0]=(0,0,0) and vertexes[1]=(1,0,0) which are the lower part of our rectangle and the last two are vertexes[2]=(0,1,0) and vertexes[3]=(1,1,0) which are the upper parts. Looking at the picture you can see how the rectangle does a smooth transition from red to green from bottom to top. Next, we assign texture coordinates:

```
        // Texture Coordinates for each position
        Vector2f[] texCoords={
            new Vector2f(0,0),
            new Vector2f(1,0),
            new Vector2f(0,1),
            new Vector2f(1,1)
        };
```

If you've worked with any 3D graphics before, the concept of texture coordinates is the same in jME as it is anywhere. Vector2f is just like a Vector3f, but it has 2 floats (notice the 2f in Vector2f) instead of 3 floats (notice the 3f in Vector3f). These two floats are x, y. I'll go into texturing later, but for now I

throw this in just so you can get the pattern of how constructing a TriMesh works. Finally, I have to make indexes for my TriMesh:

```
// The indexes of Vertex/Normal/Color/TexCoord sets.  Every 3
// makes a triangle.
int[] indexes={
    0,1,2,1,2,3
};
```

TriMesh means Triangle mesh. It is a collection of triangles. Your indexes array must always be a length of modulus 3 (3,6,9,12,15,ect). That's because triangles always have 3 coordinates. Notice this is made up of two triangles. If I had {0,1,2,1,2,3,2,3,0}, that would be 3 triangles. Lets look at the first set of 0,1,2. This means that my TriMesh object's first triangle is drawn by connecting vertexes [0] -> vertexes [1] -> vertexes [2]. Vertex [0] has a normal of normals [0], a color of colors [0], a texture coordinate of texCoords [0]. The next triangle is drawn from vertexes [1] -> vertexes [2] -> vertexes [3]. Note that it would be illegal to have the following:

```
int[] indexes={
    0,1,2,1,2,4
};
```

This is illegal because there are no vertexes[4]. After making all our data, we finally feed it into the TriMesh, then attach it after creating a bounds

```
// Feed the information to the TriMesh
m.reconstruct(vertexes,normals,colors,texCoords,indexes);

// Create a bounds
m.setModelBound(new BoundingBox());
m.updateModelBound();

// Attach the mesh to my scene graph
rootNode.attachChild(m);

// Let us see the per vertex colors
lightState.setEnabled(false);
```

The last line will make more sense after you've completed a few more chapters. For now, just realize that it turns on the per-vertex colors we need to see the rainbow effect on the box.

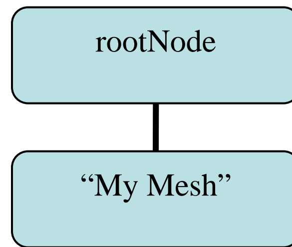
Note that I don't use the texCoords really in this example (They only have meaning when you use a picture on your object). I could have used the following:

```
m.reconstruct(vertexes,normals,colors,texCoords,indexes);
```

I could even have done the following:

```
m.reconstruct(vertexes,null,null,null,indexes);
```

Then I could have drawn a grey, boring TriMesh. Here is a picture of the scene graph:



Challenge:

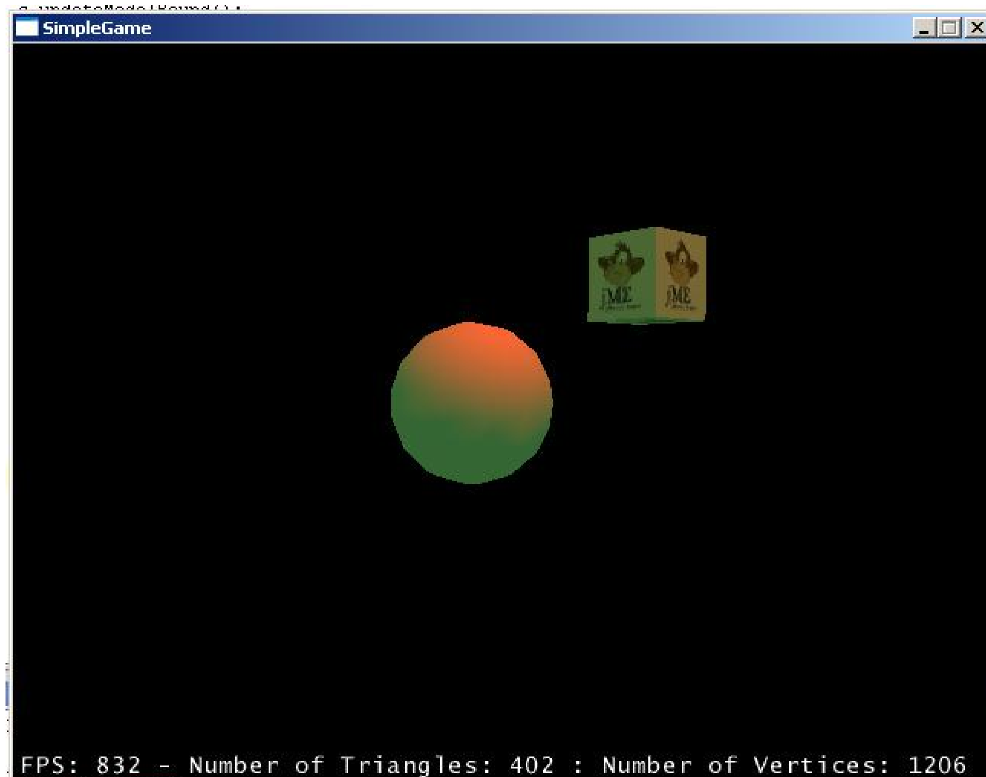
Try to create a TriMesh from scratch that looks like a pyramid.

4) Hello States

This program introduces MaterialState, TextureState, LightState, and PointLight.

```
import com.jme.app.SimpleGame;
import com.jme.scene.shape.Box;
import com.jme.scene.shape.Sphere;
import com.jme.scene.Node;
import com.jme.scene.state.TextureState;
import com.jme.scene.state.MaterialState;
import com.jme.scene.state.LightState;
import com.jme.math.Vector3f;
import com.jme.util.TextureManager;
import com.jme.image.Texture;
import com.jme.renderer.ColorRGBA;
import com.jme.light.PointLight;
import com.jme.bounding.BoundingBox;
import com.jme.bounding.BoundingSphere;

import java.net.URL;


/**
 * Started Date: Jul 20, 2004<br><br>
 *
 * Demonstrates using RenderStates with jME.
 *
 * @author Jack Lindamood
 */
public class HelloStates extends SimpleGame {
    public static void main(String[] args) {
        HelloStates app = new HelloStates();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {

        // Create our objects.  Nothing new here.
        Box b=new Box("my box",new Vector3f(1,1,1),new Vector3f(2,2,2));
        b.setModelBound(new BoundingBox());
        b.updateModelBound();
        Sphere s=new Sphere("My sphere",15,15,1);
        s.setModelBound(new BoundingSphere());
        s.updateModelBound();
        Node n=new Node("My root node");

        // Get a URL that points to the texture we're going to load
        URL monkeyLoc;
        monkeyLoc=HelloStates.class.
                   getClassLoader().getResource(
                  "jmetest/data/images/Monkey.tga");

        // Get a TextureState
        TextureState ts=display.getRenderer().createTextureState();
        // Use the TextureManager to load a texture
```

```
        Texture t =
                  TextureManager.loadTexture(monkeyLoc,
                                              Texture.MM_LINEAR,
                                              Texture.FM_LINEAR,true);
        // Assign the texture to the TextureState
        ts.setTexture(t);

        // Get a MaterialState
        MaterialState ms=display.getRenderer().createMaterialState();
        // Give the MaterialState an emissive tint
        ms.setEmissive(new ColorRGBA(0f,.2f,0f,1));

        // Create a point light
        PointLight l=new PointLight();
        // Give it a location
        l.setLocation(new Vector3f(0,10,5));
        // Make it a red light
        l.setDiffuse(ColorRGBA.red);
        // Enable it
        l.setEnabled(true);

        // Create a LightState to put my light in
        LightState ls=display.getRenderer().createLightState();
        // Attach the light
        ls.attach(l);


        // Signal that b should use renderstate ts
        b.setRenderState(ts);
        // Signal that n should use renderstate ms
        n.setRenderState(ms);
        // Detach all the default lights made by SimpleGame
        lightState.detachAll();
        // Make my light effect everything below node n
        n.setRenderState(ls);

        // Attach b and s to n, and n to rootNode.
        n.attachChild(b);
        n.attachChild(s);
        rootNode.attachChild(n);
    }
}
```



Here, the first new line of code is where we locate our Monkey image:

```
// Get a URL that points to the texture we're going to load
URL monkeyLoc;
monkeyLoc=HelloStates.class.
          getClassLoader().getResource(
         "jmetest/data/images/Monkey.tga");
```

What we are doing is locating the Monkey.tga image that we'll use to put on our cube. After the image, we need the TextureState:

```
// Get a TextureState
TextureState ts=display.getRenderer().createTextureState();
```

There's a new one! The object "display" is defined in SimpleGame. It gets whatever renderer I'm using (in my example I was using LWJGL) and from that renderer it creates a texturestate. So this would give me a TextureState that works with LWJGL. This is jME's abstraction. If I were using JOGL, it would give me a TextureState that works with JOGL. The code is independent of the rendering environment. Another way to write this line would be the following:

```
// Get a TextureState
TextureState ts=
   DisplaySystem.getDisplaySystem.getRenderer().createTextureState();
```

This is because display is equal to DisplaySystem.getDisplaySystem() in SimpleGame. Now we have our texture state, so let's attach a texture to it:

```
// Use the TextureManager to load a texture
Texture t =
         TextureManager.loadTexture(monkeyLoc,
                                    Texture.MM_LINEAR,
                                    Texture.FM_LINEAR,true);
```

We use a class called TextureManager to do this. It manages loading of textures so that all we need to do is supply the URL and it will load correctly. The weird variables MM_LINEAR and FM_LINEAR just let us know how to filter the texture. For now, just use these. The "true" at the end is a mipmap flag. I won't get into any of those here (use the wiki for a more in depth explanation). For now, just use these three all the time. After I've created a Texture with my URL, I need to feed it to my TextureState:

```
// Assign the texture to the TextureState
ts.setTexture(t);
```

Next let's make a MaterialState. While a TextureState gives things pictures, MaterialStates give things tints or colors. The difference between MaterialStates and per-vertex coloring is that the former uses lighting so it looks much better:

```
// Get a MaterialState
MaterialState ms=display.getRenderer().createMaterialState();
```

Notice I get a MaterialState very similarly to how I get a TextureState. After I get the MaterialState, I decided to give it an emissive color:

```
// Give the MaterialState an emissive tint
ms.setEmissive(new ColorRGBA(0f,.2f,0f,1));
```

This makes it look a tiny bit green, which is why you see a green tint at the bottom. There's setEmissive, setSpecular, setDiffuse, and setAmbient as well as shininess and alpha characteristics. These work just like any other modeling environment. For more detail, see the wiki.

After my two states, I'll make a light to make them show up:

```
// Create a point light
PointLight l=new PointLight();
// Give it a location
l.setLocation(new Vector3f(0,10,5));
```

This creates a point light. Think of a PointLight as a lightbulb. It's a light, at a point. Which point, you ask? Well, that's why I use setLocation. It moves my light to (x=0,y=10,z=5). Lights can have colors, of course, so I give my light a red color:

```
// Make it a red light
l.setDiffuse(ColorRGBA.red);
```

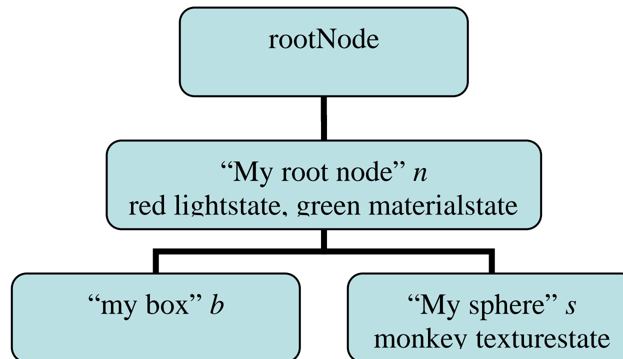
After creating the PointLight, I have to make a LightState to attach it too:

```
// Create a LightState to put my light in
LightState ls=display.getRenderer().createLightState();
// Attach the light
ls.attach(l);
```

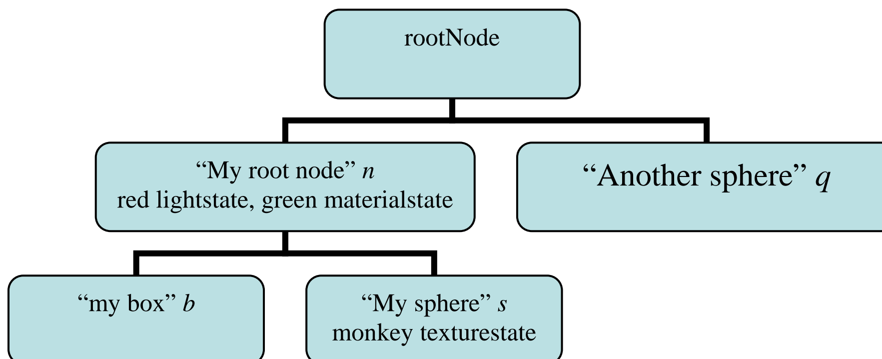
Notice I make a LightState the same way I make the other states. After I make my states, I have to assign them to the places I want them to effect:

```
// Signal that b should use renderstate ts
b.setRenderState(ts);
// Signal that n should use renderstate ms
n.setRenderState(ms);
// Detach all the default lights made by SimpleGame
lightState.detachAll();
// Make my light effect everything below node n
n.setRenderState(ls);
```

Notice I have to call detachAll() on lightState. That is because SimpleGame makes a light for us and puts it in lightState. I only want to use my lights so I detach all of lightState's lights. Finally, I build my scene graph:



Again, the scene graph inheritance comes into play. The box "my box" is affected by a red lightstate (the red light) and green materialstate (the green tint) because it is a child of *n*: The same with *s*. Notice the next scene graph:



If my scene graph looked like this, q would not be affected by *n*'s light at all, no matter how close it was to the lights location. That's because it isn't a child of *n*. This allows you to control what your lights are lighting.

Challenge:

Make a program like the above scene graph.

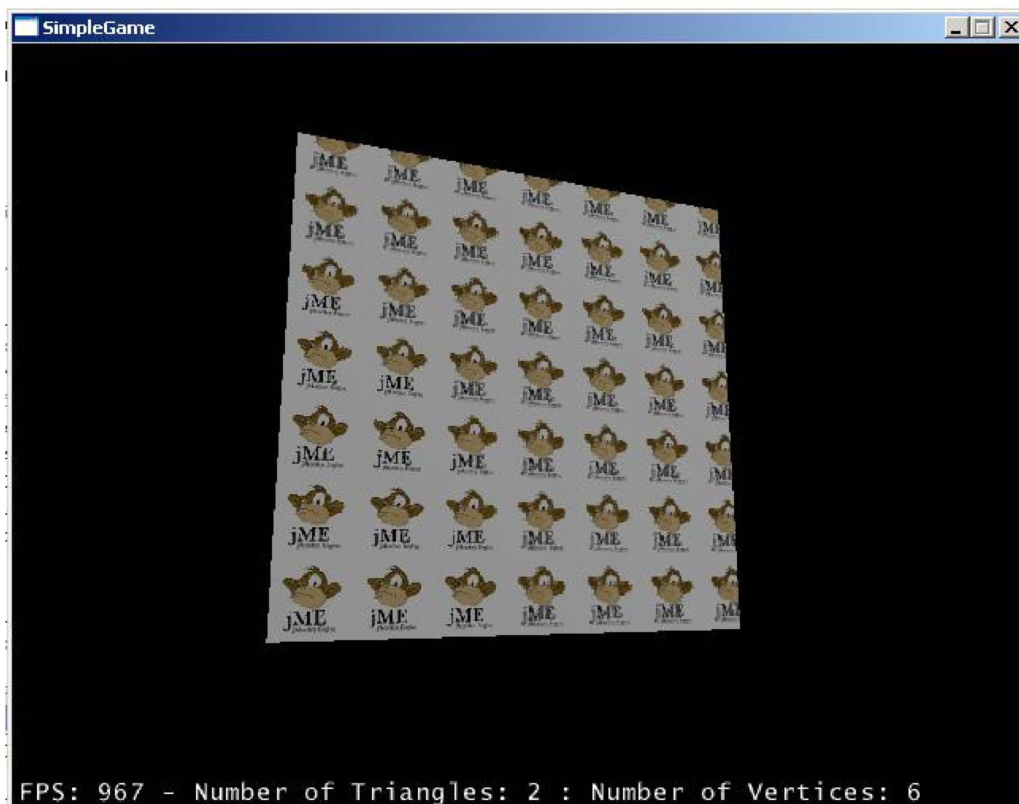
5) Hello KeyInput

This program introduces KeyBindingManager, texture wrapping, and how to change TriMesh data after it is assigned.

```
import com.jme.app.SimpleGame;
import com.jme.scene.TriMesh;
import com.jme.scene.state.TextureState;
import com.jme.math.Vector3f;
import com.jme.math.Vector2f;
import com.jme.util.TextureManager;
import com.jme.image.Texture;
import com.jme.input.KeyBindingManager;
import com.jme.input.KeyInput;

import java.net.URL;

/**
 * Started Date: Jul 21, 2004<br><br>
 *
 * This program demonstrates using key inputs to change things.
 *
 * @author Jack Lindamood
 */
public class HelloKeyInput extends SimpleGame {
    // The TriMesh that I will change
    TriMesh square;
    // A scale of my current texture values
    float coordDelta;
    public static void main(String[] args) {
        HelloKeyInput app = new HelloKeyInput();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        // Vertex positions for the mesh
        Vector3f[] vertexes={
            new Vector3f(0,0,0),
            new Vector3f(1,0,0),
            new Vector3f(0,1,0),
            new Vector3f(1,1,0)
        };

        // Texture Coordinates for each position
        coordDelta=1;
        Vector2f[] texCoords={
            new Vector2f(0,0),
            new Vector2f(coordDelta,0),
            new Vector2f(0,coordDelta),
            new Vector2f(coordDelta,coordDelta)
        };
```

```
        // The indexes of Vertex/Normal/Color/TexCoord sets.  Every 3
        // makes a triangle.
        int[] indexes={
            0,1,2,1,2,3
        };
        // Create the square
        square=new TriMesh("My Mesh", vertexes, null, null,
                           texCoords, indexes);
        // Point to the monkey image
        URL monkeyLoc=
                HelloKeyInput.class.getClassLoader().
                getResource("jmetest/data/images/Monkey.tga");
        // Get my TextureState
        TextureState ts=display.getRenderer().createTextureState();
        // Get my Texture
        Texture t=TextureManager.loadTexture(monkeyLoc,
                                             Texture.MM_LINEAR,
                                             Texture.FM_LINEAR,true);
        // Set a wrap for my texture so it repeats
        t.setWrap(Texture.WM_WRAP_S_WRAP_T);
        // Set the texture to the TextureState
        ts.setTexture(t);

        // Assign the TextureState to the square
        square.setRenderState(ts);
        // Scale my square 10x larger
        square.setLocalScale(10);
        // Attach my square to my rootNode
        rootNode.attachChild(square);

        // Assign the "+" key on the keypad to the command "coordsUp"
        KeyBindingManager.getKeyBindingManager().set(
            "coordsUp",
            KeyInput.KEY_ADD);

        // Adds the "u" key to the command "coordsUp"
        KeyBindingManager.getKeyBindingManager().add(
            "coordsUp",
            KeyInput.KEY_U);

        // Assign the "-" key on the keypad to the command "coordsDown"
        KeyBindingManager.getKeyBindingManager().set(
            "coordsDown",
            KeyInput.KEY_SUBTRACT);
    }

    // Called every frame update
    protected void simpleUpdate(){

        // If the coordsDown command was activated
        if (KeyBindingManager.getKeyBindingManager().
                                isValidCommand("coordsDown",true)){
            // Scale my texture down
            coordDelta-=.01f;
            // Get my square's texture array
            Vector2f[] texes=square.getTextures();
```

```
                // Change the values of the texture array
                texes[1].set(coordDelta,0);
                texes[2].set(0,coordDelta);
                texes[3].set(coordDelta,coordDelta);
                // Tell the square TriMesh that I have changed values in
                 // it's array
                square.updateTextureBuffer();
            }
            // if the coordsUp command was activated
            if (KeyBindingManager.getKeyBindingManager().
                                    isValidCommand("coordsUp",true)){
                // Scale my texture up
                coordDelta+=.01f;
                // Assign each texture value manually.
                square.setTexture(1,new Vector2f(coordDelta,0));
                square.setTexture(2,new Vector2f(0,coordDelta));
                square.setTexture(3,new Vector2f(coordDelta,coordDelta));
            }
        }
}
```



The first new thing you see here is how I create my square:

```
            // Create the square
            square=new TriMesh("My Mesh", vertexes, null, null,
                               texCoords, indexes);
```

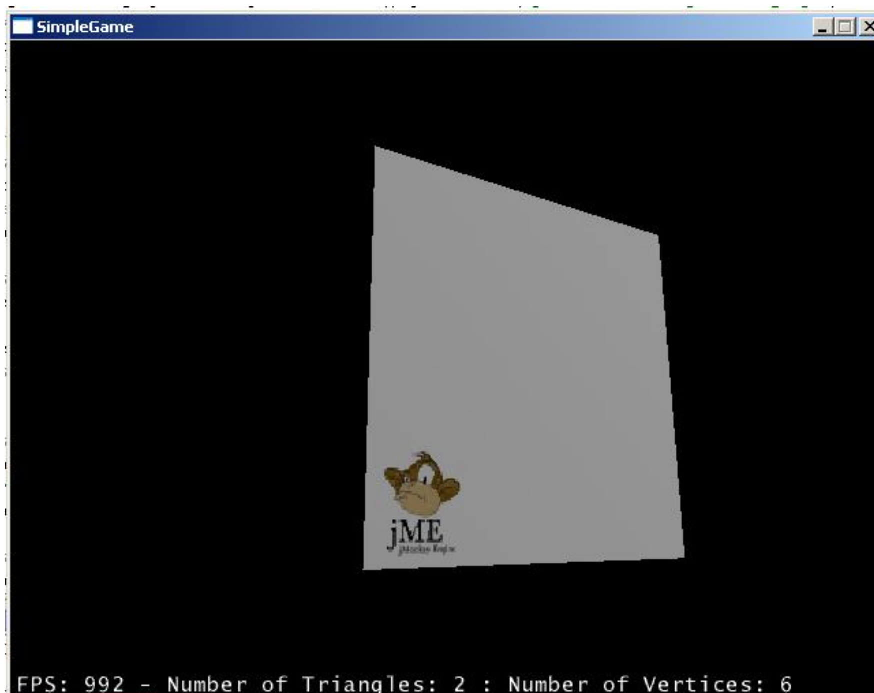
This is equivalent to the following:

```
// Create the square
square=new TriMesh("My Mesh");
square.reconstruct(vertexes, null, null, texCoords, indexes);
```

So it's just like example 3, but I put the reconstruct into the constructor. Not too difficult, right? The next new thing to show you is how I set a wrap attribute:

```
// Set a wrap for my texture so it repeats
t.setWrap(Texture.WM_WRAP_S_WRAP_T);
```

Without this, my picture would just get smaller. It wouldn't show up 20 million times like it does in the picture. As an example, comment this line out and run the program. You'll see something like the following:



After attaching the TextureState and nodes correctly, I begin to bind commands to keys:

```
// Assign the "+" key on the keypad to the command "coordsUp"
KeyBindingManager.getKeyBindingManager().set(
    "coordsUp",
    KeyInput.KEY_ADD);
```

Key input is abstracted away just like the rendering, so I have to use display.getRenderer() in the same way I also have to use KeyBindingManager.getKeyBindingManager() . This function sets the command "coordsUp" to the key KEY_ADD (which is the + key). Later, we won't check "Did they press the +

key". Instead we will check "Did the coordsUp action happen". This lets me assign multiple keys to the same command, and lets users customize their keys very easily. Speaking of this, next I assign the *U* key to coordsUp as well:

```
        // Adds the "u" key to the command "coordsUp"
        KeyBindingManager.getKeyBindingManager().add(
            "coordsUp",
            KeyInput.KEY_U);
```

Notice the "add" function call, not the "set" function call. This "adds" key *U* to the "coordsUp" command. So either key *U* or + will trigger "coordsUp". Next I add a key for coordsDown:

```
        // Assign the "-" key on the keypad to the command "coordsDown"
        KeyBindingManager.getKeyBindingManager().set(
            "coordsDown",
            KeyInput.KEY_SUBTRACT);
```

It's just like the one I assigned for "coordsUp". Next you see a new function we haven't used before:

```
    // Called every frame update
    protected void simpleUpdate(){
```

I am overriding the simpleUpdate() function in SimpleGame. This function is called every frame. By putting queries here, I can poll for a key every frame. Which is exactly what I do right off the bat:

```
        // If the coordsDown command was activated
        if (KeyBindingManager.getKeyBindingManager().
                                isValidCommand("coordsDown",true)){
```

Notice I don't ask "Did they press the + key or the *U* key". I don't need to know which key is "coordsDown". I just ask "Did a coordsDown key get pressed". The Boolean at the end signals if I allow users to hold the button down and execute the command multiple times. Change the true to false, and you'll notice you only get once change per button press. On a down command, I shrink my texture cords for each point of my square:

```
            // Scale my texture down
            coordDelta-=.01f;
```

Next I get my texture array for the square and change the values in the array:

```
            // Get my square's texture array
            Vector2f[] texes=square.getTextures();
            // Change the values of the texture array
            texes[1].set(coordDelta,0);
            texes[2].set(0,coordDelta);
            texes[3].set(coordDelta,coordDelta);
```

Texture coordinate numbers are a value between 0 and 1 that signals which part of my picture is put in the rectangle. If they go from 0 to .5 then the lower half is shown. If they go from .5 to 1 then the upper

half is shown. Because I set the command *wrap* the texture will wrap so that a value of 1.5 would look like my entire picture plus the first half. That is why as coordDelta gets large, you see many monkeys. The *set* command of Vector2f just changes the vector's values. So for example:

```
            texes[1].set(coordDelta,0);
```

Changes texes[1] to an x of coordDelta and a y of 0. After I change my texture values, I must tell my square that the values of its textures array have changed. This lets the square recreate its texture buffer:

```
            // Tell the square TriMesh that I have changed values in
            // it's array
            square.updateTextureBuffer();
```

If I don't call this function, the square won't know its texture buffer needs to be updated. Next, I check for the "coordsUp" command:

```
        // if the coordsUp command was activated
        if (KeyBindingManager.getKeyBindingManager().
                                isValidCommand("coordsUp",true)){
```

If this command is activated, then I shrink my texture cords. I set up this function a little bit different than the last one just as an example:

```
            // Scale my texture up
            coordDelta+=.01f;
            // Assign each texture value manually.
            square.setTexture(1,new Vector2f(coordDelta,0));
            square.setTexture(2,new Vector2f(0,coordDelta));
            square.setTexture(3,new Vector2f(coordDelta,coordDelta));
```

First, notice I never ask for the texture array. I also don't have to tell square that the texture array has changed. The reason is that each function call of setTexture updates my texture buffer as it is called. Each is one way of doing the same thing. In general, use the first if most values in your array are going to change, and use the second if only a few values are going to change. That's it. Simple!

Challenge:

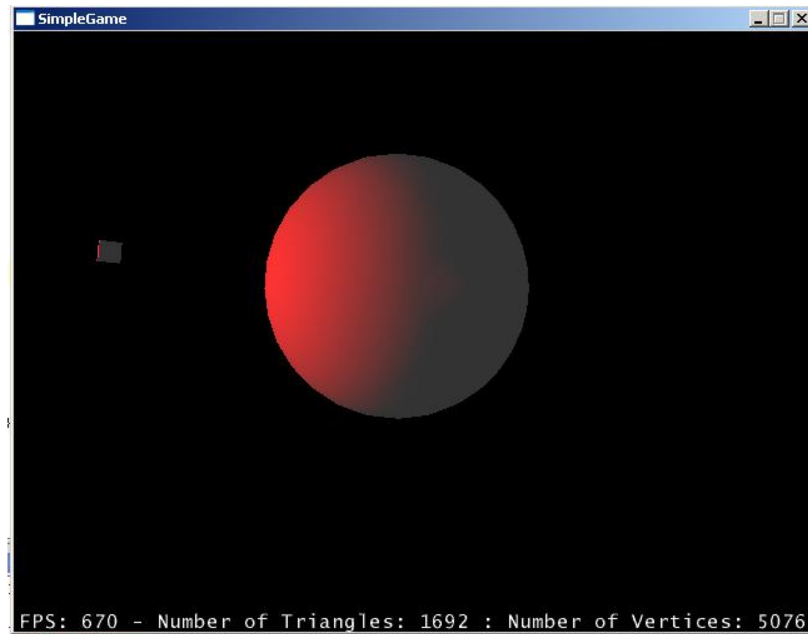
Add keys to make the square get bigger when +/- are pressed, instead of increasing the texture coordinates.

6) Hello Animation

This program introduces LightNode and using Controllers. The controller we will use is called SpatialTransformer.

```
import com.jme.app.SimpleGame;
import com.jme.scene.shape.Box;
import com.jme.scene.shape.Sphere;
import com.jme.scene.Node;
import com.jme.math.Vector3f;
import com.jme.math.Quaternion;
import com.jme.math.FastMath;
import com.jme.animation.SpatialTransformer;
import com.jme.light.PointLight;
import com.jme.light.SimpleLightNode;
import com.jme.renderer.ColorRGBA;

/**
 * Started Date: Jul 21, 2004<br><br>
 *
 * This class demonstrates animation via a controller, as well as LightNode.
 *
 * @author Jack Lindamood
 */
public class HelloAnimation extends SimpleGame {
    public static void main(String[] args) {
        HelloAnimation app = new HelloAnimation();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        Sphere s=new Sphere("My sphere",30,30,5);
        // I will rotate this pivot to move my light
        Node pivot=new Node("Pivot node");

        // This light will rotate around my sphere.  Notice I don't
        // give it a position
        PointLight pl=new PointLight();
        // Color the light red
        pl.setDiffuse(ColorRGBA.red);
        // Enable the light
        pl.setEnabled(true);
        // Remove the default light and attach this one
        lightState.detachAll();
        lightState.attach(pl);

        // This node will hold my light
        SimpleLightNode ln=
              new SimpleLightNode("A node for my pointLight",pl);
        // I set the light's position thru the node
        ln.setLocalTranslation(new Vector3f(0,10,0));
        // I attach the light's node to my pivot
        pivot.attachChild(ln);
```

```
        // I create a box and attach it too my lightnode.
        // This lets me see where my light is
        Box b=new Box("Blarg", new Vector3f(-.3f,-.3f,-.3f),
                               new Vector3f(.3f,.3f,.3f));
        ln.attachChild(b);

        // I create a controller to rotate my pivot
        SpatialTransformer st=new SpatialTransformer(1);
        // I tell my spatial controller to change pivot
        st.setObject(pivot,0,-1);

        // Assign a rotation for object 0 at time 0 to rotate 0
        // degrees around the z axis
        Quaternion x0=new Quaternion();
        x0.fromAngleAxis(0,new Vector3f(0,0,1));
        st.setRotation(0,0,x0);

        // Assign a rotation for object 0 at time 2 to rotate 180
        // degrees around the z axis
        Quaternion x180=new Quaternion();
        x180.fromAngleAxis(FastMath.DEG_TO_RAD*180,new Vector3f(0,0,1));
        st.setRotation(0,2,x180);

        // Assign a rotation for object 0 at time 4 to rotate 360
        // degrees around the z axis
        Quaternion x360=new Quaternion();
        x360.fromAngleAxis(FastMath.DEG_TO_RAD*360,new Vector3f(0,0,1));
        st.setRotation(0,4,x360);

        // Prepare my controller to start moving around
        st.interpolateMissing();
        // Tell my pivot it is controlled by st
        pivot.addController(st);

        // Attach pivot and sphere to graph
        rootNode.attachChild(pivot);
        rootNode.attachChild(s);
    }
}
```



The first new thing in this program is LightNode:

```
        // This node will hold my light
        SimpleLightNode ln=
              new SimpleLightNode("A node for my pointLight",pl);
        // I set the light's position thru the node
        ln.setLocalTranslation(new Vector3f(0,10,0));
        // I attach the light's node to my pivot
        pivot.attachChild(ln);
```

Only objects that extend Spatial can be attached to a scene graph. Node extends Spatial. Box extends TriMesh, which extends Geometry, which extends Spatial. PointLight extends Light, but is not a spatial. LightNode is a bridge between Light and Spatial. When I set the LightNode's local translation, I am actually setting my lights position. The difference is that this light's position is relative to the LightNode. For example, if light node's parent is translated up 10 and the lightnode is translated up 10, then the light is at 20 y. But, if I light.setLocation() to 10, then the light's exact position is 10 no matter what parent it's attached too. Usually, you'll use LightNode to position your lights just like I did. The next new thing is when I create my controller to rotate my pivot:

```
        // I create a controller to rotate my pivot
        SpatialTransformer st=new SpatialTransformer(1);
        // I tell my spatial controller to change pivot
        st.setObject(pivot,0,-1);
```

SpatialTransformer extends Controller. Controllers 'control' things. They are the java3d equivalent of Behaviors. Every frame, any Spatial in the graph has its Controllers updated first. SpatialTransformer changes a set of Spatial's rotations, translations, and scales over time. The first part creates a SpatialTransformer that will change one object. I set the object pivot to index 0 on the second part. The -1 at the end of setObject is a parent index. Don't confuse this with an

object's parent in the scene graph. It's a little different. For now, always use -1. Now that SpatialTransformer knows *what* to update, lets tell it *how* to update it:

```
        // Assign a rotation for object 0 at time 0 to rotate 0
        // degrees around the z axis
        Quaternion x0=new Quaternion();
        x0.fromAngleAxis(0,new Vector3f(0,0,1));
        st.setRotation(0,0,x0);
```

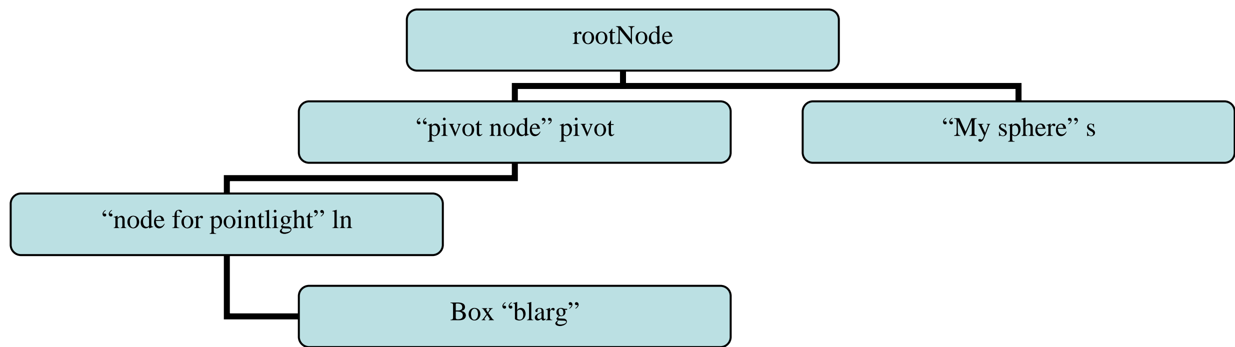
The fromAngleAxis command means that Quaternion x0 is equivalent to rotating 0 degrees about a vector drawn to x=0,y=0,z=1. After calculating the correct rotation, I say that index 0 (which is pivot) should at time=0 rotate by x0. Next I assign a rotation of 180 degrees at time=2. Because the fromAngleAxis function takes rotations in radians, and not in degrees, I have to convert 180 to radians. I use FastMath to do that. FastMath is an equivalent of the Math library, but uses all floats instead of doubles. This is because everything in jME uses floats, not doubles:

```
        // Assign a rotation for object 0 at time 2 to rotate 180
        // degrees around the z axis
        Quaternion x180=new Quaternion();
        x180.fromAngleAxis(FastMath.DEG_TO_RAD*180,new Vector3f(0,0,1));
        st.setRotation(0,2,x180);
```

After setting a rotation for time=4 to rotate the object back, I wrap up using my Controller:

```
        // Prepare my controller to start moving around
        st.interpolateMissing();
        // Tell my pivot it is controlled by st
        pivot.addController(st);
```

In order to correctly interpolate rotations/scales/translations for times I didn't specify to SpatialTransformer, I call interpolateMissing(). This is just a requirement for SpatialTransformer to work correctly. After interpolating, I add it as a controller (not as a child) to pivot. This means the controller will be updated and the object rotated every frame. My scene graph looks like the following:



Challenge:

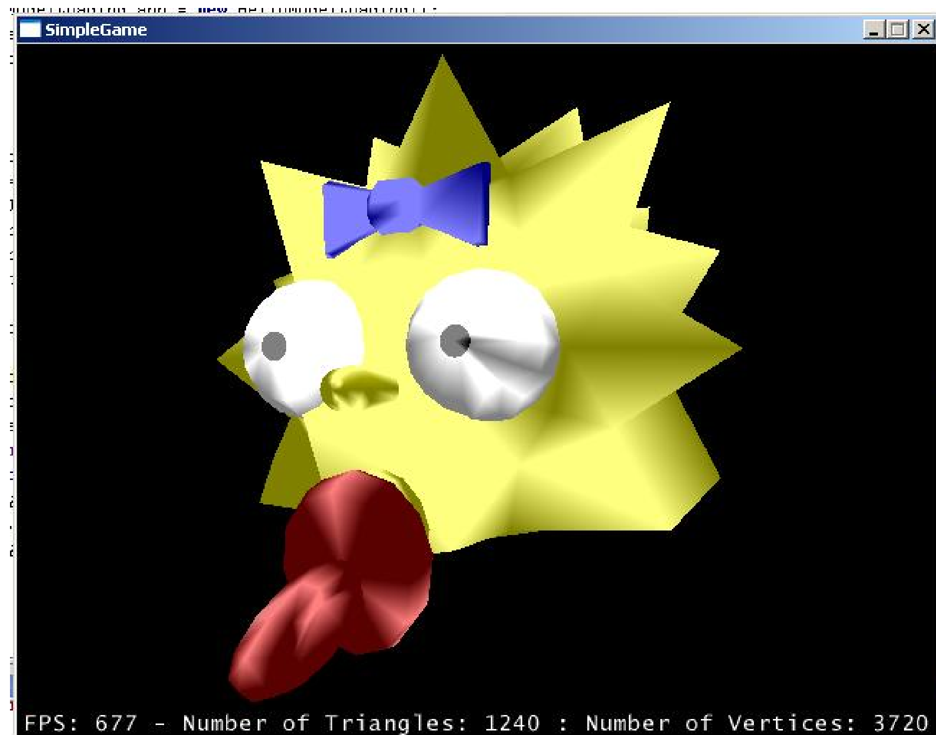
Make a sphere that rotates around a light.

7) Hello ModelLoading

This program introduces JmeBinaryReader, FileConverter, BinaryToXML, and LoggingSystem. You will learn how to convert formats to a jME scene graph.

```
import com.jme.app.SimpleGame;
import com.jme.scene.model.XMLparser.Converters.ObjToJme;
import com.jme.scene.model.XMLparser.Converters.FormatConverter;
import com.jme.scene.model.XMLparser.JmeBinaryReader;
import com.jme.scene.model.XMLparser.BinaryToXML;
import com.jme.scene.Node;
import com.jme.util.LoggingSystem;

import java.net.URL;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.OutputStreamWriter;
import java.util.logging.Level;

/**
 * Started Date: Jul 22, 2004<br><br>
 *
 * Demonstrates loading formats.
 *
 * @author Jack Lindamood
 */
public class HelloModelLoading extends SimpleGame {
    public static void main(String[] args) {
        HelloModelLoading app = new HelloModelLoading();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        // Turn the logger off so we can see the XML later on
        LoggingSystem.getLogger().setLevel(Level.OFF);
        app.start();
    }

    protected void simpleInitGame() {
        // Point to a URL of my model
        URL model=
                  HelloModelLoading.class.getClassLoader().
                  getResource("jmetest/data/model/maggie.obj");

        // Create something to convert .obj format to .jme
        FormatConverter converter=new ObjToJme();
        // Point the converter to where it will find the .mtl file from
        converter.setProperty("mtllib",model);

        // This byte array will hold my .jme file
        ByteArrayOutputStream BO=new ByteArrayOutputStream();
        // This will read the .jme format and convert it into a scene graph
        JmeBinaryReader jbr=new JmeBinaryReader();
        // Use this to visualize the .obj file in XML
        BinaryToXML btx=new BinaryToXML();
        try {
            // Use the format converter to convert .obj to .jme
            converter.convert(model.openStream(), BO);
```

```
            // Send the .jme binary to System.out as XML
            btx.sendBinarytoXML(
                            new ByteArrayInputStream(BO.toByteArray()),
                            new OutputStreamWriter(System.out));

            // Tell the binary reader to use bounding boxes instead of
            // bounding spheres
            jbr.setProperty("bound","box");

            // Load the binary .jme format into a scene graph
            Node maggie=jbr.loadBinaryFormat(
                          new ByteArrayInputStream(BO.toByteArray()));
            // shrink this baby down some
            maggie.setLocalScale(.1f);
            // Put her on the scene graph
            rootNode.attachChild(maggie);
        } catch (IOException e) {   // Just in case anything happens
            System.out.println("Damn exceptions!" + e);
            e.printStackTrace();
            System.exit(0);
        }
    }
```



Pretty short program, eh? The first new thing here is when I turn off jME's logging system:

```
        // Turn the logger off so we can see the XML later on
        LoggingSystem.getLogger().setLevel(Level.OFF);
```

jME has its own LoggingSystem to log different types of warnings in your game. You can play around with it yourself if you're interested. I do this simply to turn it off so you can see the .obj's XML equivalent in later on.

Next I make my obj converter:

```
        // Create something to convert .obj format to .jme
        FormatConverter converter=new ObjToJme();
```

To understand file loading with jME, one must understand that jme doesn't support directly loading any file format other than its own binary format called "jME binary." There are classes such as ObjToJme or MaxToJme or Md2ToJme for example that convert from the given format to jME's binary. They all extend FormatConverter. Because the file we're using is an obj format file, we use ObjToJme to convert it to binary. Next, we assign the mtl library:

```
        // Point the converter to where it will find the .mtl file from
        converter.setProperty("mtllib",model);
```

Normally, it is sufficient to simply convert without setting any properties. Obj format is different though because it splits its information between the .obj file and a .mtl file. The obj holds geometry and the mtl holds material information. Because of this, we have to set a property to tell the converter which directory to find mtl libraries in. For what properties are needed for what formats, consult the javadoc of each format you're using. After telling the converter to find mtl files in the same directory as the model, we create a binary reader:

```
        // This byte array will hold my .jme file
        ByteArrayOutputStream BO=new ByteArrayOutputStream();
        // This will read the .jme format and convert it into a scene graph
        JmeBinaryReader jbr=new JmeBinaryReader();
        // Use this to visualize the .obj file in XML
        BinaryToXML btx=new BinaryToXML();
```

jME works purely with streams for converting files: InputStreams to read files and OutputStreams to send their contents. What you would do in your game is write a program to convert your .obj file to .jme to disk with FileOutputStream, then simply read the .jme in every time your game is run with a FileInputStream. What we am doing here, though, is simply writing to a byte stream then reading back from that stream later. We will later use BinaryToXML to convert the .obj file to XML so you can see it in human readable format. After creating my converter, I simply call convert:

```
            // Use the format converter to convert .obj to .jme
            converter.convert(model.openStream(), BO);
```

Convert takes an InputStream which is the .obj (or .3ds or .md3 or .ms3d or…), and sends it to an OutputStream as .jme. If anything goes wrong with converting or reading a file, an IOException is thrown, which is why I have to surround it with a try/catch. After convert, BO

now has the .obj file, but in jME's binary format. Next, simply as an example, I send the jME binary to System.out as XML:

```
// Send the .jme binary to System.out as XML
btx.sendBinarytoXML(
                new ByteArrayInputStream(BO.toByteArray()),
                new OutputStreamWriter(System.out));
```

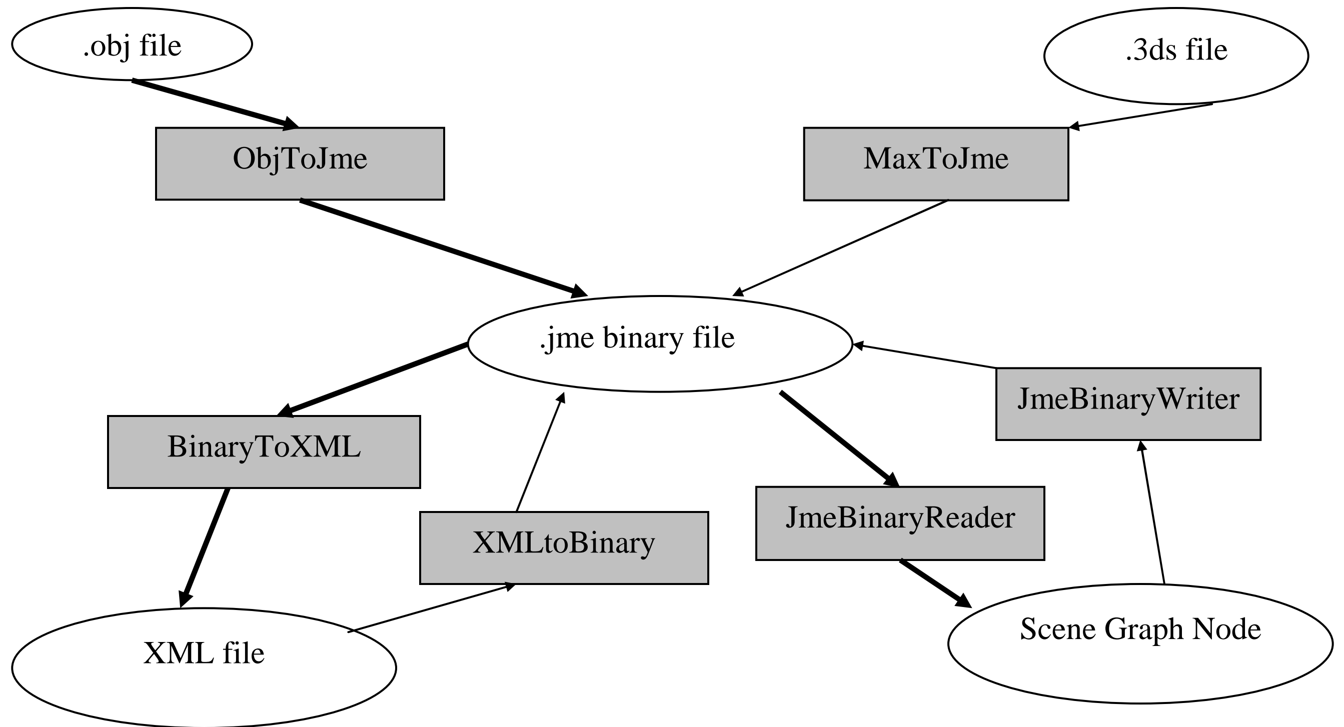
Not only does jME support sending binary to XML, it also supports sending XML to binary. This allows you to write easy export scripts for your modeling software and export it to XML so that jME can then read it in. Now, again as an example, I will instruct my JmeBinaryReader to read in the binary format and use BoundingBox instead of BoundingSphere around my model:

```
// Tell the binary reader to use bounding boxes instead of
// bounding spheres
jbr.setProperty("bound","box");
```

Again, for a complete list of settable properties, look at JmeBinaryReader's javadoc. Finally, all I have to do is load the jME binary into a scene graph:

```
// Load the binary .jme format into a scene graph
Node maggie=jbr.loadBinaryFormat(
              new ByteArrayInputStream(BO.toByteArray()));
```

In your games, before release you would have already converted your obj file to jME's binary and would only need to simply read it in. A flow chart of jME's file loading system is as follows. I highlight the path we took in our program. Note that there are more than just 2 supported formats. I only list .obj and .3ds here for the sake of size:



Challenge:

Find and load your own .obj file from the internet.

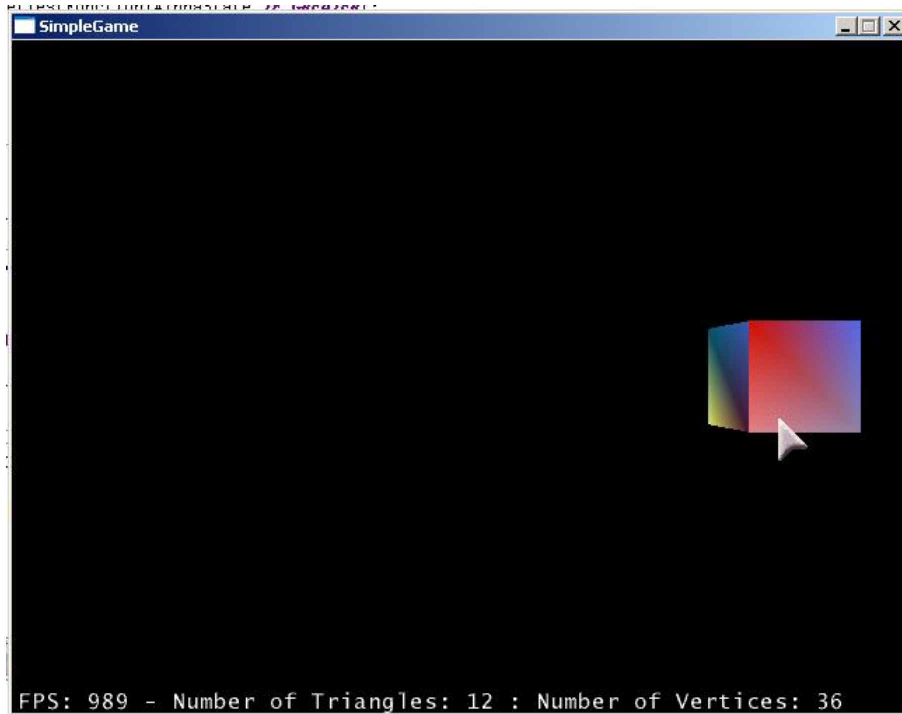
8) Hello MousePick

This program introduces AbsoluteMouse, AlphaState, InputSystem, Ray, and Intersection. You will learn how to create your own mouse icon and determine if the user is clicking an item on screen.

```
import java.net.URL;

import com.jme.app.SimpleGame;
import com.jme.bounding.BoundingBox;
import com.jme.image.Texture;
import com.jme.input.AbsoluteMouse;
import com.jme.input.InputSystem;
import com.jme.input.MouseInput;
import com.jme.intersection.BoundingPickResults;
import com.jme.intersection.PickResults;
import com.jme.math.Ray;
import com.jme.math.Vector2f;
import com.jme.math.Vector3f;
import com.jme.scene.shape.Box;
import com.jme.scene.state.AlphaState;
import com.jme.scene.state.TextureState;
import com.jme.util.TextureManager;

/**
 * Started Date: Jul 22, 2004 <br>
 * <br>
 *
 * Demonstrates picking with the mouse.
 *
 * @author Jack Lindamood
 */
public class HelloMousePick extends SimpleGame {
  // This will be my mouse
  AbsoluteMouse am;

  // This will be he box in the middle
  Box b;

  PickResults pr;

  public static void main(String[] args) {
        HelloMousePick app = new HelloMousePick();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
  }

  protected void simpleInitGame() {
        // Create a new mouse. Restrict its movements to the display screen.
        am = new AbsoluteMouse("The Mouse",
                display.getWidth(), display.getHeight());

        // Get a picture for my mouse.
        TextureState ts = display.getRenderer().createTextureState();
        URL cursorLoc;
        cursorLoc = HelloMousePick.class.getClassLoader().getResource(
                        "jmetest/data/cursor/cursor1.png");
```

```
        Texture t = TextureManager.loadTexture(cursorLoc,
                        Texture.MM_LINEAR,
                        Texture.FM_LINEAR, true);
        ts.setTexture(t);
        am.setRenderState(ts);

        // Make the mouse's background blend with what's already there
        AlphaState as = display.getRenderer().createAlphaState();
        as.setBlendEnabled(true);
        as.setSrcFunction(AlphaState.SB_SRC_ALPHA);
        as.setDstFunction(AlphaState.DB_ONE_MINUS_SRC_ALPHA);
        as.setTestEnabled(true);
        as.setTestFunction(AlphaState.TF_GREATER);
        am.setRenderState(as);

        // Get the mouse input device and assign it to the AbsoluteMouse
        am.setMouseInput(InputSystem.getMouseInput());
        // Move the mouse to the middle of the screen to start with
        am.setLocalTranslation(new Vector3f(display.getWidth() / 2,
                        display.getHeight() / 2, 0));
        // Assign the mouse to an input handler
        input.setMouse(am);
        // Create the box in the middle. Give it a bounds
        b = new Box("My Box", new Vector3f(-1, -1, -1),
                        new Vector3f(1, 1, 1));
        b.setModelBound(new BoundingBox());
        b.updateModelBound();
        // Attach Children
        rootNode.attachChild(b);
        rootNode.attachChild(am);
        // Remove all the lightstates so we can see the per-vertex colors
        lightState.detachAll();
        pr = new BoundingPickResults();
}

protected void simpleUpdate() {
        // Get the mouse input device from the jME mouse
        MouseInput thisMouse = am.getMouseInput();
        // Is button 0 down? Button 0 is left click
        if (thisMouse.isButtonDown(0)) {
                Vector2f screenPos = new Vector2f();
                // Get the position that the mouse is pointing to
                screenPos.set(am.getHotSpotPosition().x,
                              am.getHotSpotPosition().y);
                // Get the world location of that X,Y value
                Vector3f worldCoords = display.getWorldCoordinates(
                                                     screenPos, 0);
                // Create a ray starting from the camera, and
                // going in the direction of the mouse's location
                Ray mouseRay = new Ray(cam.getLocation(), worldCoords
                               .subtractLocal(cam.getLocation()));
                // Does the mouse's ray intersect the box's world bounds?
                pr.clear();
                rootNode.findPick(mouseRay, pr);

                for (int i = 0; i < pr.getNumber(); i++) {
```

```
                                pr.getPickData(i).getTargetMesh().setRandomColors();
                        }
                }
        }
}
```



The first new part here is when I declare my AbsoluteMouse:

```
    // This will be my mouse
    AbsoluteMouse am;
```

An AbsoluteMouse holds an absolute position for the mouse currently on the screen. There is also a RelativeMouse, which doesn't bother with the mouse's actual position but only with the position relative to the last movement. An AbsoluteMouse is, obviously, like the one we're using here. A RelativeMouse is like +mouselook on Quake or your favorite shooter game. It doesn't care about the mouse's position on the screen, but just where you're moving the mouse. The next new thing is when I create the AbsoluteMouse object:

```
            // Create a new mouse. Restrict its movements to the display screen.
            am = new AbsoluteMouse("The Mouse",
                    display.getWidth(), display.getHeight());
```

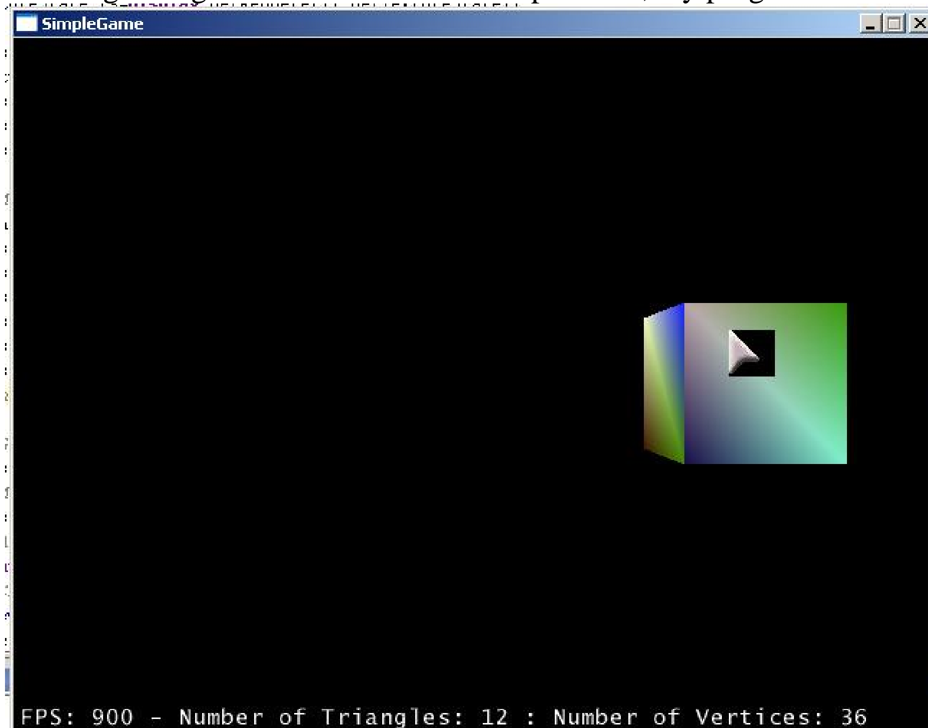
```
AbsoluteMouse extends Mouse
Mouse extends Geometry
Geometry extends Spatial
```

As we know, all Spatials must have a name. The name of this spatial is "The Mouse". It makes sense. All things that are drawn are Geometry objects. So to draw the mouse on the screen, we make it a

Geometry. The two numbers at the end restrict the movement of the mouse. If it was 40,40 then the mouse could only move in the lower 40,40 of the screen. We simply restrict it to the size of the display. The object "display" is created for us in SimpleGame. The next new item is AlphaState:

```
// Make the mouse's background blend with what's already there
AlphaState as = display.getRenderer().createAlphaState();
as.setBlendEnabled(true);
as.setSrcFunction(AlphaState.SB_SRC_ALPHA);
as.setDstFunction(AlphaState.DB_ONE_MINUS_SRC_ALPHA);
as.setTestEnabled(true);
as.setTestFunction(AlphaState.TF_GREATER);
am.setRenderState(as);
```

That's a lot of strange things there. If I didn't use AlphaState, my program would look like this:



That's because my cursor is a square picture. Doesn't look too nice, does it? What I need is to tell jME to blend the cursor's transparent color with what's already there. That's what my AlphaState does. Color blending for AlphaStates is divided into a source function and destination function. jME's alphastates work in a similar way to OpenGL alpha states. For more information, look up alpha states for OpenGL and how they are used. For now, just use what I told you. Now that we've setup how our mouse will "look", we have to actually tie the mouse input device to the mouse on the screen:

```
// Get the mouse input device and assign it to the AbsoluteMouse
am.setMouseInput(InputSystem.getMouseInput());
```

InputSystem holds input information for whatever rendering environment you're using. In my example, it returns a LWJGLMouseInput because I'm using LWJGL. This lets jME separate whatever

environment you're using and the input devices associated with it. Next, I move the mouse to the center of the screen for the user:

```
// Move the mouse to the middle of the screen to start with
am.setLocalTranslation(new Vector3f(display.getWidth() / 2,
                       display.getHeight() / 2, 0));
```

Again, I use display.getWidth() to figure out the width of the window. Notice the setLocalTranslation I use is just like the one I use for other Spatials. Next, I tie the mouse to an input handler:

```
// Assign the mouse to an input handler
input.setMouse(am);
```

The object input is created in SimpleGame. By default, it's the thing that lets us move around. I change it to work with my AbsoluteMouse instead of aiming. In your own game, you would use your own InputHandler. An InputHandler is a class that lets you handle your inputs separately from your main game code. After creating my box, I attach both my mouse and the box to rootNode. After I deactivate my lightstate, you see a strange line by itself at the end of simpleInitGame()

```
pr = new BoundingPickResults();
```

This creates a container for our "pick results" which we will use during simpleUpdate. Next, you see simpleUpdate. Remember, it is called every frame. First, we get the input device associated with am:

```
// Get the mouse input device from the jME mouse
MouseInput thisMouse = am.getMouseInput();
```

Note that this is equivalent to the following:

```
MouseInput thisMouse = InputSystem.getMouseInput();
```

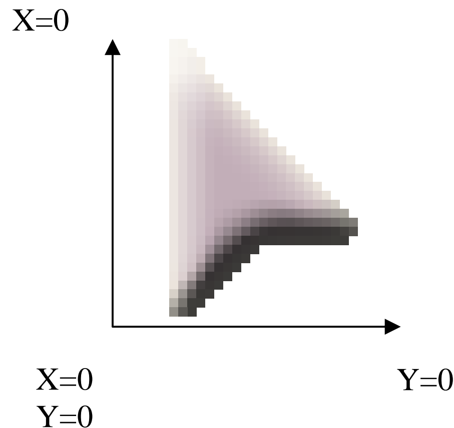
I suggest you get the MouseInput from am instead of the system, though, as it abstracts the two. After getting the mouse, we check for a button press:

```
// Is button 0 down? Button 0 is left click
if (thisMouse.isButtonDown(0)) {
```

We could have checked for isButtonDown(1) or isButtonDown(2) and so on if your mouse has that many buttons on it. Button 0 happens to be left click in windows. If the button is pressed, we check to see if they are pressing it on the box, and if so we change the box's color. The first step in that is getting to where the mouse is on the screen:

```
Vector2f screenPos = new Vector2f();
// Get the position that the mouse is pointing to
screenPos.set(am.getHotSpotPosition().x,
              am.getHotSpotPosition().y);
```

We get the mouse's absolute X and Y position. To understand why I use getHotSpotPosition, lets look at a picture of my mouse:



Notice that the actual cursor icon points to the top. Because of this, my cursor is actually pointing to its position plus whatever the icon's height is. You can set your hotspot to wherever your mouse pointer actually is. But default, the hotspot is at x=0, y=ImageHeight so we're ok with the default. After getting the correct cursor position, I get the real world value for wherever it's pointing:

```
// Get the world location of that X,Y value
Vector3f worldCoords = display.getWorldCoordinates(
                                              screenPos, 0);
```

This translates a position on the screen to a position in my world. The 3rd float I pass in, 0, is the distance away from the screen's position that this 3D position will take. 0 would be the screen position right in front of you. 10 would be the screen position 10 units away from you. Now that I know at which point my cursor is located in the real world, I need to create a ray using that point:

```
// Create a ray starting from the camera, and
// going in the direction of the mouse's location
Ray mouseRay = new Ray(cam.getLocation(), worldCoords
               .subtractLocal(cam.getLocation()));
```

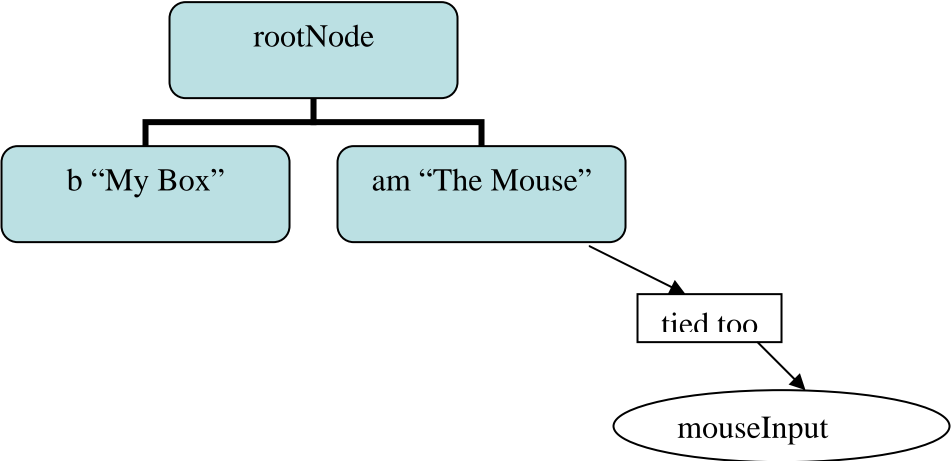
This creates a ray starting wherever my camera is and going in whichever direction I'm clicking. Rays in jME are defined with a starting location and a direction. I subtract my world coordinate from my camera's location to get a direction relative to the camera's location. There's some math going on there. Sadly, you'll need to use math to create any good 3D game. For more math information, check out jME's math links. Next, I check to see if the ray I have intersects the BoundingBox for my box. If so, I change colors. To do that, I first clear any previous pick results I may have had.

```
pr.clear();
```

Next, I have to initiate the "pick" call. I am going to see if any children of rootNode intersect with the ray I just created, and if so their results are stored in pr. For each item that intersected that ray, I'm changing its color.

```
                    rootNode.findPick(mouseRay, pr);

                    for (int i = 0; i < pr.getNumber(); i++) {
                            pr.getPickData(i).getTargetMesh().setRandomColors();
                    }
```

Here is a picture of what the scene graph would look like:



Challenge:

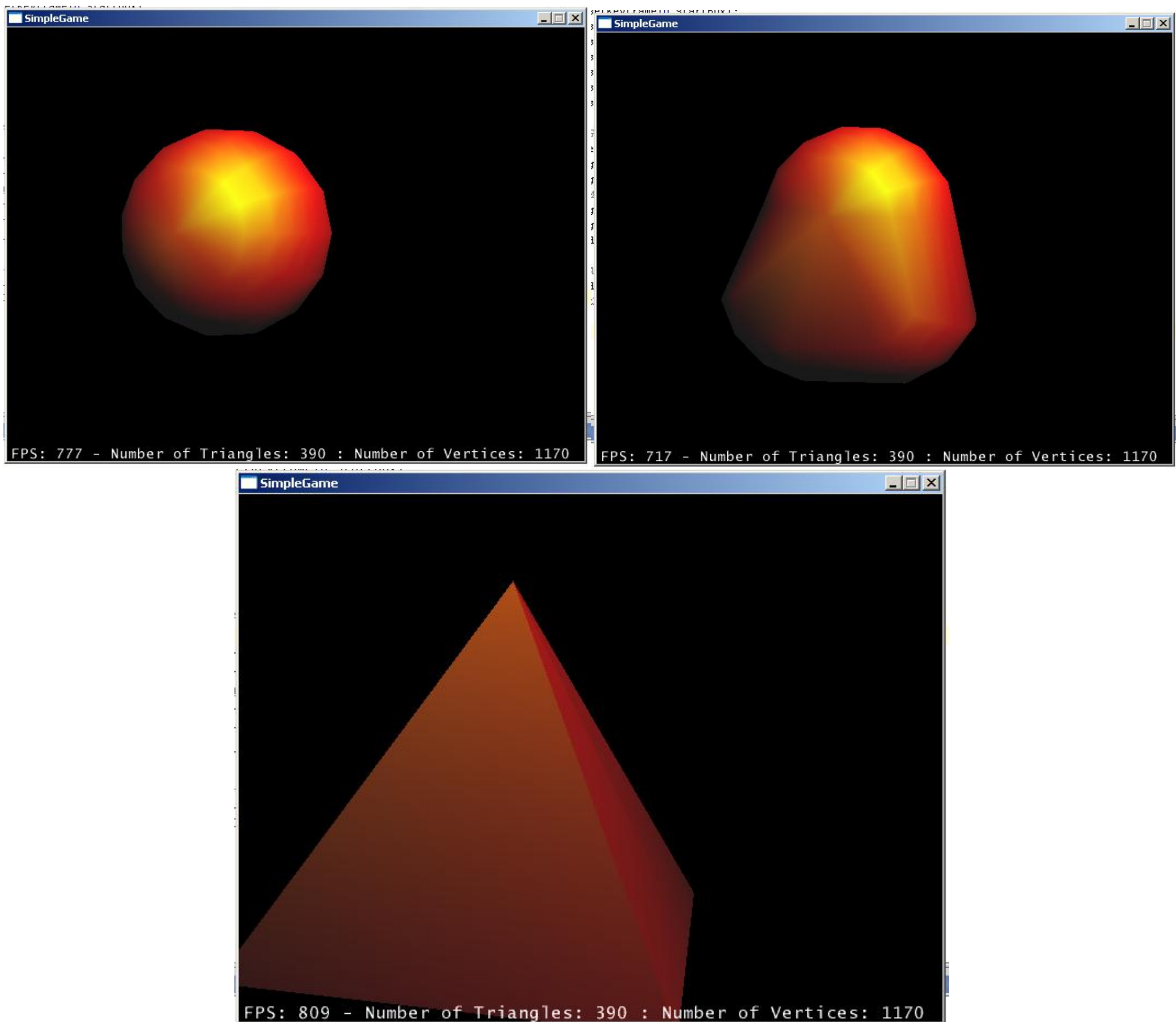
Change your mouse’s hotspot so that the bottom of the mouse is where clicks register.
Shrink the box whenever it is clicked.

9) Hello Keyframes

This program introduces KeyframeController. You will learn how to make your own animations and how to manipulate vertex information after an object is created.

```
import com.jme.app.SimpleGame;
import com.jme.scene.shape.Sphere;
import com.jme.scene.TriMesh;
import com.jme.scene.Controller;
import com.jme.scene.state.MaterialState;
import com.jme.renderer.ColorRGBA;
import com.jme.math.Vector3f;
import com.jme.math.FastMath;
import com.jme.animation.KeyframeController;

/**
 * Started Date: Jul 23, 2004<br><br>
 *
 * Demonstrates making your own keyframe animations.
 *
 * @author Jack Lindamood
 */
public class HelloKeyframes extends SimpleGame {
    public static void main(String[] args) {
        HelloKeyframes app = new HelloKeyframes();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        // The box we start off looking like
        TriMesh startBox=new Sphere("begining box",15,15,3);
        // Null colors,normals,textures because they aren't being updated
        startBox.setColors(null);
        startBox.setNormals(null);
        startBox.setTextures(null);

        // The middle animation sphere
        TriMesh middleSphere=new Sphere("middleSphere sphere",15,15,3);
        middleSphere.setColors(null);
        middleSphere.setNormals(null);
        middleSphere.setTextures(null);

        // The end animation pyramid
        TriMesh endPyramid=new Sphere("End sphere",15,15,3);
        endPyramid.setColors(null);
        endPyramid.setNormals(null);
        endPyramid.setTextures(null);

        Vector3f[] boxVerts=startBox.getVertices();
        Vector3f[] sphereVerts=middleSphere.getVertices();
        Vector3f[] pyramidVerts=endPyramid.getVertices();

        for (int i=0;i<sphereVerts.length;i++){
            Vector3f boxPos=boxVerts[i];
```

```
            Vector3f spherePos=sphereVerts[i];
            Vector3f pyramidPos=pyramidVerts[i];

            // The box is the sign of the sphere coords * 4
            boxPos.x =FastMath.sign(spherePos.x)*4;
            boxPos.y =FastMath.sign(spherePos.y)*4;
            boxPos.z =FastMath.sign(spherePos.z)*4;

            if (boxPos.y<0){     // The bottom of the pyramid
                pyramidPos.x=boxPos.x;
                pyramidPos.y=-4;
                pyramidPos.z=boxPos.z;
            }
            else    // The top of the pyramid
                pyramidPos.set(0,4,0);
        }

        // The object that will actually be rendered
        TriMesh renderedObject=new Sphere("Rendered Object",15,15,3);
        renderedObject.setLocalScale(2);

        // Create my KeyframeController
        KeyframeController kc=new KeyframeController();
        // Assign the object I'll be changing
        kc.setMorphingMesh(renderedObject);
        // Assign for a time, what my renderedObject will look like
        kc.setKeyframe(0,startBox);
        kc.setKeyframe(.5f,startBox);
        kc.setKeyframe(2.75f,middleSphere);
        kc.setKeyframe(3.25f,middleSphere);
        kc.setKeyframe(5.5f,endPyramid);
        kc.setKeyframe(6,endPyramid);
        kc.setRepeatType(Controller.RT_CYCLE);

        // Give it a red material with a green tint
        MaterialState redgreen=display.getRenderer().createMaterialState();
        redgreen.setDiffuse(ColorRGBA.red);
        redgreen.setSpecular(ColorRGBA.green);
        // Make it very affected by the Specular color.
        redgreen.setShininess(10f);
        redgreen.setEnabled(true);
        renderedObject.setRenderState(redgreen);

        // Add the controller to my object
        renderedObject.addController(kc);
        rootNode.attachChild(renderedObject);
    }
}
```



The first strange thing you see here is:

```
        // Null colors,normals,textures because they aren't being updated
        startBox.setColors(null);
        startBox.setNormals(null);
        startBox.setTextures(null);
```

This happens right after I create startBox. To understand what I'm doing, you must first understand how KeyframeController works. It animates geometry information from one TriMesh to another, and then to another, etc. So what I am going to do is make one TriMesh that looks like a cube, another that looks like a sphere, and another that looks like a pyramid. If array values are null for any TriMesh in my KeyframeController, then those arrays won't be interpolated. Because I don't need to interpolate color/texture/normal I set those null. One requirement of KeyframeController is that all the Keyframes

and the mesh that I'm transforming all have the same number of vertexes in them. All KeyframeController does is to interpolate from one to the next, which is why the array lengths must all be equal. In this example, the simplest way to do that is to make the same object multiple times and change each vertex value as it needs to be changed, which is what I do. After I make my objects, I get my vertex positions and loop thru each vertex value:

```
Vector3f[] boxVerts=startBox.getVertices();
Vector3f[] sphereVerts=middleSphere.getVertices();
Vector3f[] pyramidVerts=endPyramid.getVertices();

for (int i=0;i<sphereVerts.length;i++){
```

To make my box, I simply take my sphere values and change them to either -4 or 4 depending upon their sign:

```
    // The box is the sign of the sphere coords * 4
    boxPos.x =FastMath.sign(spherePos.x)*4;
```

Notice I use FastMath to calculate the sign. My pyramid is a little more complex. The explanation is more math-based than jME-based. Sadly, math keeps coming up in 3D graphics:

```
    if (boxPos.y<0){    // The bottom of the pyramid
        pyramidPos.x=boxPos.x;
        pyramidPos.y=-4;
        pyramidPos.z=boxPos.z;
    }
    else    // The top of the pyramid
        pyramidPos.set(0,4,0);
```

After creating my "frames" and my object that will actually be rendered, I create a KeyframeController:

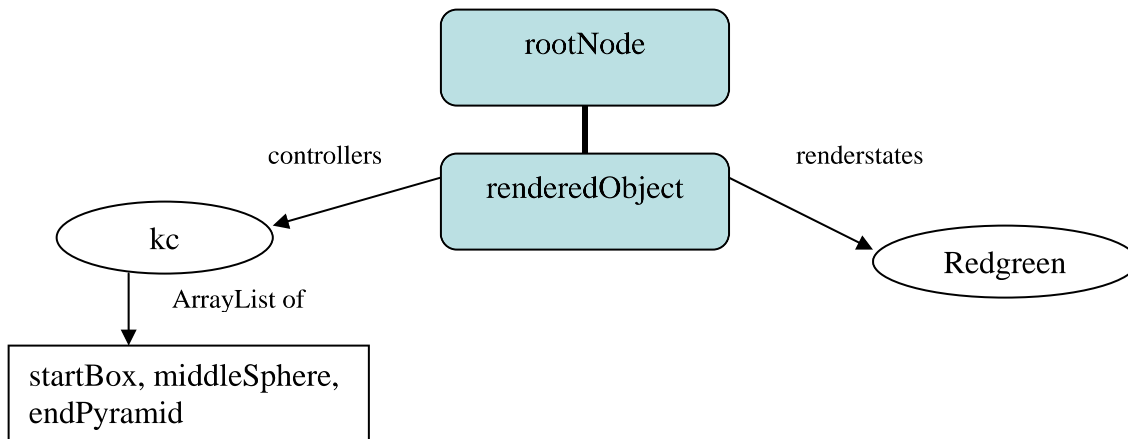
```
// Create my KeyframeController
KeyframeController kc=new KeyframeController();
// Assign the object I'll be changing
kc.setMorphingMesh(renderedObject);
```

This assigns the mesh my KeyframeController will animate. Next, I assign time values to frames:

```
// Assign for a time, what my renderedObject will look like
kc.setKeyframe(0,startBox);
kc.setKeyframe(.5f,startBox);
kc.setKeyframe(2.75f,middleSphere);
kc.setKeyframe(3.25f,middleSphere);
kc.setKeyframe(5.5f,endPyramid);
kc.setKeyframe(6,endPyramid);
kc.setRepeatType(Controller.RT_CYCLE);
```

For example, from 0 to .5f seconds, animate between startBox and startBox (which basically look the same). From time .5 to 2.75 animate between startBox and middleSphere. And so on. A repeat type of RT_CYCLE cycles my animation. Once it reaches 6 seconds, it goes backwards towards 0 again.

Finally, I create my material state, attach my controller, and attach the TriMesh I want to be rendered to my rootNode. The scene graph looks something like this:



Challenge:

Try to morph Maggie.obj to a sphere. Hint: You will need to use a new sphere for each material.

10) Hello Intersection

This program introduces SoundNode, Skybox, and SoundAPIController. You will learn how to play sounds, create text, and make your own controllers and input handler. This also introduces rudimentary collision detection.

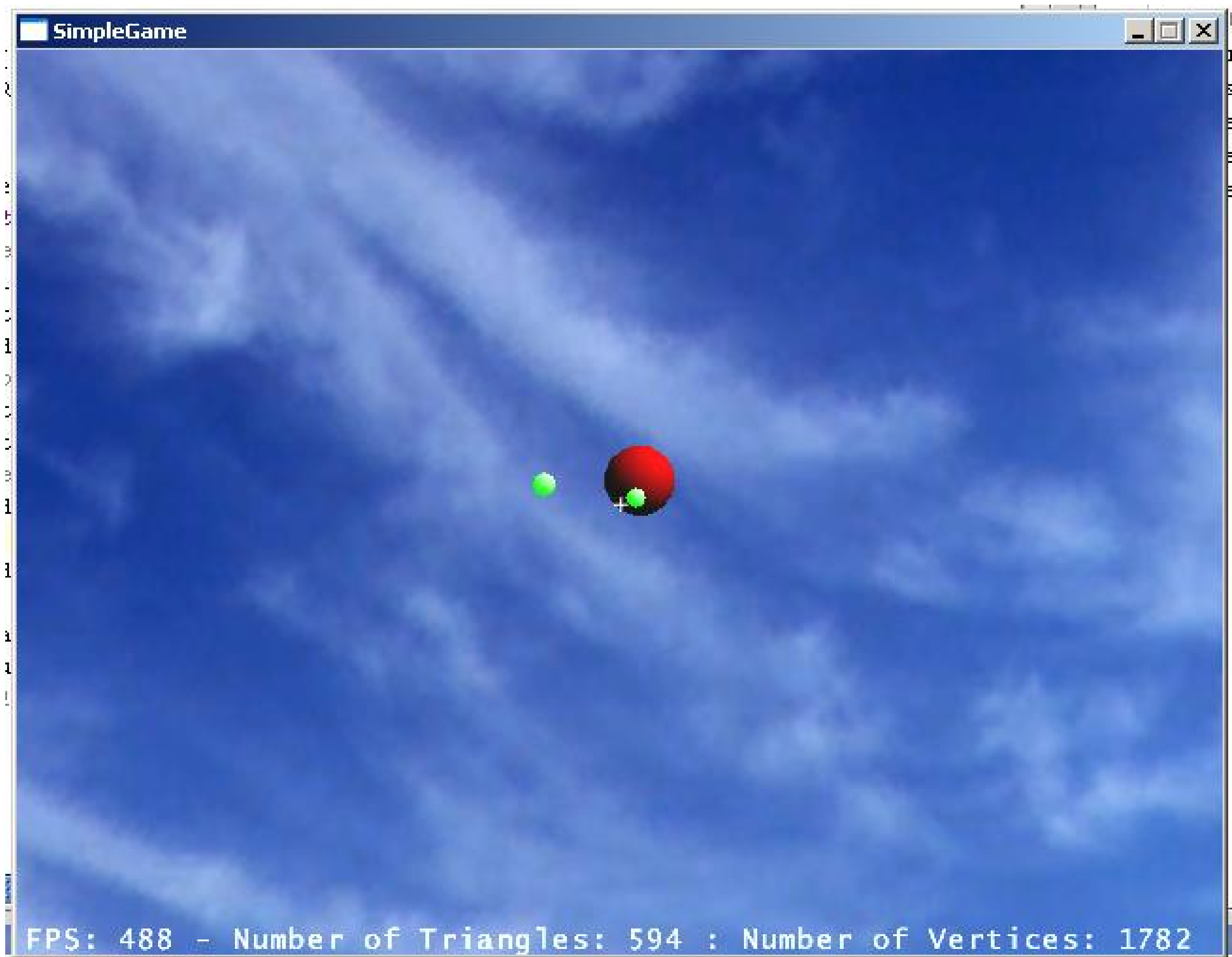
```
package jmetest.TutorialGuide;

import java.net.URL;
import java.util.Random;

import com.jme.app.SimpleGame;
import com.jme.bounding.BoundingSphere;
import com.jme.image.Texture;
import com.jme.input.KeyInput;
import com.jme.input.action.AbstractInputAction;
import com.jme.intersection.Intersection;
import com.jme.math.Vector3f;
import com.jme.renderer.ColorRGBA;
import com.jme.scene.Controller;
import com.jme.scene.Skybox;
import com.jme.scene.Text;
import com.jme.scene.TriMesh;
import com.jme.scene.shape.Sphere;
import com.jme.scene.state.MaterialState;
import com.jme.scene.state.TextureState;
import com.jme.sound.SoundAPIController;
import com.jme.util.TextureManager;
import com.jme.sound.SoundPool;
import com.jme.sound.scene.ProgrammableSound;
import com.jme.sound.scene.SoundNode;

/**
 * Started Date: Jul 24, 2004 <br>
 * <br>
 *
 * Demonstrates intersection testing, sound, and making your own controller.
 *
 * @author Jack Lindamood
 */
public class HelloIntersection extends SimpleGame {
    /** Material for my bullet */
    MaterialState bulletMaterial;

    /** Target you're trying to hit */
    Sphere target;

    /** Location of laser sound */
    URL laserURL;

    /** Location of hit sound */
    URL hitURL;

    /** Used to move target location on a hit */
    Random r = new Random();
```

```java
/** A sky box for our scene. */
Skybox sb;

/**
 * The programmable sound that will be in charge of maintaining our sound
 * effects.
 */
ProgrammableSound laserSound;

ProgrammableSound targetSound;

/** The node where attached sounds will be propagated from */
SoundNode snode;

/**
 * The ID of our laser shooting sound effect. The value is not
 * important. It should just be unique in our game to this sound.
 */
private int laserEventID = 1;

private int hitEventID = 2;

public static void main(String[] args) {
    HelloIntersection app = new HelloIntersection();
    app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
    app.start();
}

protected void simpleInitGame() {
    setupSound();
    BoundingSphere.useExactBounds=true;

    /** Create a + for the middle of the screen */
    Text cross = new Text("Crosshairs", "+");

    // 8 is half the width of a font char
    /** Move the + to the middle */
    cross.setLocalTranslation(new Vector3f(display.getWidth() / 2f - 8f,
            display.getHeight() / 2f - 8f, 0));
    fpsNode.attachChild(cross);
    target = new Sphere("my sphere", 15, 15, 1);
    target.setModelBound(new BoundingSphere());
    target.updateModelBound();
    rootNode.attachChild(target);

    /** Create a skybox to suround our world */
    sb = new Skybox("skybox", 200, 200, 200);
    URL monkeyLoc = HelloIntersection.class.getClassLoader().getResource(
            "jmetest/data/texture/clouds.png");
    TextureState ts = display.getRenderer().createTextureState();
    ts.setTexture(TextureManager.loadTexture(monkeyLoc,
            Texture.MM_LINEAR,
            Texture.FM_LINEAR, true));
    sb.setRenderState(ts);
```

```
        // Attach the skybox to our root node, and force the rootnode to show
        // so that the skybox will always show
        rootNode.attachChild(sb);
        rootNode.setForceView(true);

        /**
         * Set the action called "firebullet", bound to KEY_F, to
         * performAction FireBullet
         */
        input.addKeyboardAction("firebullet", KeyInput.KEY_F,
                                new FireBullet());

        /** Make bullet material */
        bulletMaterial = display.getRenderer().createMaterialState();
        bulletMaterial.setEmissive(ColorRGBA.green);

        /** Make target material */
        MaterialState redMaterial =
                display.getRenderer().createMaterialState();
        redMaterial.setDiffuse(ColorRGBA.red);
        target.setRenderState(redMaterial);
    }

    private void setupSound() {
        /** init sound API acording to the rendering enviroment you're using
        */
        SoundAPIController.getSoundSystem(properties.getRenderer());
        /** Set the 'ears' for the sound API */
        SoundAPIController.getRenderer().setCamera(cam);
        snode = new SoundNode();
        /** Create program sound */
        targetSound = new ProgrammableSound();
        /** Make the sound softer */
        targetSound .setLooping(false);
        targetSound.setMaxDistance(140f);

        laserSound = new ProgrammableSound();
        laserSound.setLooping(false);


        /** locate laser and register it with the prog sound. */

        laserURL = HelloIntersection.class.getClassLoader().getResource(
                "jmetest/data/sound/laser.ogg");
        hitURL = HelloIntersection.class.getClassLoader().getResource(
                "jmetest/data/sound/explosion.ogg");
        // Ask the system for a program id for this resource
        int programid = SoundPool.compile(new URL[] { laserURL });
        int hitid = SoundPool.compile(new URL[] { hitURL });
        // Then we bind the programid we received to our laser event id.
        laserSound.bindEvent(laserEventID, programid);
        targetSound.bindEvent(hitEventID, hitid);
        snode.attachChild(laserSound);
        snode.attachChild(targetSound);
    }
```

```java
    class FireBullet extends AbstractInputAction {
        int numBullets;

        FireBullet() {
            setAllowsRepeats(false);
        }

        public void performAction(float time) {
            System.out.println("BANG");
            /** Create bullet */
            Sphere bullet = new Sphere("bullet" + numBullets++, 8, 8, .25f);
            bullet.setModelBound(new BoundingSphere());
            bullet.updateModelBound();
            /** Move bullet to the camera location */
            bullet.setLocalTranslation(new Vector3f(cam.getLocation()));
            bullet.setRenderState(bulletMaterial);
            /**
             * Update the new world locaion for the bullet before I add a
             * controller
             */
            bullet.updateGeometricState(0, true);
            /**
             * Add a movement controller to the bullet going in the camera's
             * direction
             */
            bullet.addController(new BulletMover(bullet, new Vector3f(cam
                    .getDirection())));
            rootNode.attachChild(bullet);
            bullet.updateRenderState();
            /** Signal our sound to play laser during rendering */
            laserSound.setPosition(cam.getLocation());
            snode.onEvent(laserEventID);
        }
    }

    class BulletMover extends Controller {
        /** Bullet that's moving */
        TriMesh bullet;

        /** Direciton of bullet */
        Vector3f direction;

        /** speed of bullet */
        float speed = 10;

        /** Seconds it will last before going away */
        float lifeTime = 5;

        BulletMover(TriMesh bullet, Vector3f direction) {
            this.bullet = bullet;
            this.direction = direction;
            this.direction.normalizeLocal();
        }

        public void update(float time) {
            lifeTime -= time;
```

```
            /** If life is gone, remove it */
            if (lifeTime < 0) {
                rootNode.detachChild(bullet);
                bullet.removeController(this);
                return;
            }
            /** Move bullet */
            Vector3f bulletPos = bullet.getLocalTranslation();
            bulletPos.addLocal(direction.mult(time * speed));
            bullet.setLocalTranslation(bulletPos);
            /** Does the bullet intersect with target? */
            if (Intersection.intersection(bullet.getWorldBound(), target
                    .getWorldBound())) {
                System.out.println("OWCH!!!");
                targetSound.setPosition(target.getWorldTranslation());
                target.setLocalTranslation(new Vector3f(r.nextFloat() * 10, r
                        .nextFloat() * 10, r.nextFloat() * 10));
                lifeTime = 0;

                snode.onEvent(hitEventID);
            }
        }
    }

    /**
     * Called every frame for rendering
     */
    protected void simpleRender() {
        // Give control to the sound in case sound changes are needed.
        SoundAPIController.getRenderer().draw(snode);
    }

    /**
     * Called every frame for updating
     */
    protected void simpleUpdate() {
        // Let the programmable sound update itself.
        snode.updateGeometricState(tpf, true);
    }
}
```



Whew! This is a deep program and our first one that actually does something kind of fun! OK, first new thing is the sound:

```
/**
 * The programmable sound that will be in charge of maintaining our sound
 * effects.
 */
ProgrammableSound laserSound;

ProgrammableSound targetSound;

/** The node where attached sounds will be propagated from */
SoundNode snode;

/**
 * The ID of our laser shooting sound effect. The value is not
 * important. It should just be unique in our game to this sound.
 */
private int laserEventID = 1;
```

```
    private int hitEventID = 2;
```

Sounds have to be rendered just like nodes in a scene graph. ProgrammableSound extends SoundSpatial extends Object. So because programSound isn't a "Spatial" in the sense of a TriMesh or Geometry or Node, it can't be rendered with the regular rendering environment. It is rendered with a sound system, while objects are rendered with a display system. In this system, we are going to use a Programmable sound. A programmable sound is a node that has various links to URLs in it. Each URL is a sound that can be fired on an event. The event ID for our laser sound is 1. The number isn't as important as being unique. We'll get more into using it later, but look at how I setup the sound:

```
        /** init sound API acording to the rendering enviroment you're using
        */
        SoundAPIController.getSoundSystem(properties.getRenderer());
        /** Set the 'ears' for the sound API */
        SoundAPIController.getRenderer().setCamera(cam);
```

The first line creates a sound system. Sound isn't enabled by default. We are creating a sound system that is specific to the renderer we are using. So if we're using LWJGL it will create a LWJGL usable sound system. Next we setup the 'ears' for our sound by placing them where the camera is. Afterwards, we create our sound:

```
        snode = new SoundNode();
        /** Create program sound */
        targetSound = new ProgrammableSound();
        /** Make the sound softer */
        targetSound .setLooping(false);
        targetSound.setMaxDistance(140f);
```

We say here our sound can be heard up to 140 units away and won't loop. We do the same with our laser sound, but don't give it a max distance.

```
        laserSound = new ProgrammableSound();
        laserSound.setLooping(false);
```

Next, we tie the sounds into the Sound nodes:

```
        /** locate laser and register it with the prog sound. */

        laserURL = HelloIntersection.class.getClassLoader().getResource(
                "jmetest/data/sound/laser.ogg");
        hitURL = HelloIntersection.class.getClassLoader().getResource(
                "jmetest/data/sound/explosion.ogg");
        // Ask the system for a program id for this resource
        int programid = SoundPool.compile(new URL[] { laserURL });
        int hitid = SoundPool.compile(new URL[] { hitURL });
        // Then we bind the programid we received to our laser event id.
        laserSound.bindEvent(laserEventID, programid);
        targetSound.bindEvent(hitEventID, hitid);
        snode.attachChild(laserSound);
        snode.attachChild(targetSound);
```

SoundPool.compile takes an array of URLs and compiles them into a "SoundPool" that we can use in our program. Those sounds are identified by an integer. After compiling, we tell snode that whenever event laserEventID is called, it should play the sounds in SoundPool that are identified by programid. Whenever the event hitEventID is triggered, we should play the sounds identified by hitid. We only use one sound, but we could easily have a laser sound followed by a … bang tied into our programid by using an array of 2 URLs. After our sounds, we put our crosshair on the screen:

```
/** Create a + for the middle of the screen */
Text cross = new Text("Crosshairs", "+");

// 8 is half the width of a font char
/** Move the + to the middle */
cross.setLocalTranslation(new Vector3f(display.getWidth() / 2f - 8f,
        display.getHeight() / 2f - 8f, 0));
fpsNode.attachChild(cross);
```

Text extends Geometry. So this means Text needs a name. We call it "crosshairs". Text just displays text on the screen. This one will display a "+". The translation of the text isn't its real world coordinates like other things, but is its actual screen coordinates. The Z of setLocalTranslation is ignored by Text so 0 is fine. I set my + to the middle of the screen in this example. You will notice I don't attach it too rootNode but to fpsNode. fpsNode is created in SimpleGame and has one child by default. That child is the text you see at the bottom of the screen. I attach my *cross* to fpsNode because fpsNode has a special texturestate already in it. Let's look at that texture state for a second:

Text will look at the ASCII you give it and match it to a place on a picture and display a rectangle for each letter. Taking advantage of this, you can easily modify a picture file to have any text you want to create your own custom fonts for your games. After creating the +, I create the skybox for my world:

```
/** Create a skybox to suround our world */
sb = new Skybox("skybox", 200, 200, 200);
URL monkeyLoc = HelloIntersection.class.getClassLoader().getResource(
        "jmetest/data/texture/clouds.png");
TextureState ts = display.getRenderer().createTextureState();
```

```
        ts.setTexture(TextureManager.loadTexture(monkeyLoc,
                Texture.MM_LINEAR,
                Texture.FM_LINEAR, true));
        sb.setRenderState(ts);
```

This simply creates a box that is 200x200x200 units big. The box is made so that textures can appear on the inside of it. I assign clouds as my texture. The final new thing in simpleInitGame is when I bind a key to an action:

```
        input.addKeyboardAction("firebullet", KeyInput.KEY_F,
                                new FireBullet());
```

input is created in SimpleGame. It is of class InputHandler. By default, SimpleGame gives input the information it needs to do simple keyboard and mouse movements in our world. Here, I am adding KEY_F to be the action called "firebullet" which is handled by FireBullet() object. If you've done action binding for AWT, this is similar. This simply states that when you press the F key, FireBullet.performAction will be called. Lets look in FireBullet's constructor:

```
        FireBullet() {
            setAllowsRepeats(false);
        }
```

This function setAllowsRepeats(false) means that if a user holds down the F key, they won't see thousands of bullets going across their screen. If you wanted to turn this semi-automatic into an automatic, simply use setAllowsRepeats(true) in the constructor. Notice that FireBullet extends AbstractInputAction, which allows us to bind it through input.addKeyboardAction(). Next is performAction:

```
            System.out.println("BANG");
            /** Create bullet */
            Sphere bullet = new Sphere("bullet" + numBullets++, 8, 8, .25f);
            bullet.setModelBound(new BoundingSphere());
            bullet.updateModelBound();
            /** Move bullet to the camera location */
            bullet.setLocalTranslation(new Vector3f(cam.getLocation()));
            bullet.setRenderState(bulletMaterial);
```

Whenever the F key is pressed, first I create my bullet. Next I move it to the camera's position (after all that's where the bullet will come from) and then give it a greenish color:

```
            /**
             * Update the new world locaion for the bullet before I add a
             * controller
             */
            bullet.updateGeometricState(0, true);
```

I call the function updateGeometricState(0,true) because I need to update the bullet's geometric information (more importantly its boundingsphere). This gives the bullet the correct worldbounds so

that the immediately following update() for BulletMover() will know the correct location for the bullet. The true simply states that bullet is starting the updateGeometricState call. This is needed because the function is recursive for all of bullet's children (if it was able to have any). Behind our backs, every frame SimpleGame calls rootNode.updateGeometricState(time_per_frame,true) to move and update all of rootNode's children. Next, I give the bullet a controller to move it:

```
/**
 * Add a movement controller to the bullet going in the camera's
 * direction
 */
bullet.addController(new BulletMover(bullet, new Vector3f(cam
        .getDirection())));
```

This moves the bullet in the camera's direction. We'll get into BulletMover later, but for now lets finish our F key press:

```
rootNode.attachChild(bullet);
bullet.updateRenderState();
```

Next we attach the nodes and call updateRenderState() on the bullet. Again, updateRenderState() is called behind our backs in SimpleGame after simpleInitGame(). The function takes all the render states we are trying to apply to a spatial and actually applies them, as well as any children states. Without updating our renderstates for our bullet, it would look red because it doesn't know about the green. Finally, we signal to programSound that it should initiate event laserEventID, which as you remember we tied to the laser sound:

```
/** Signal our sound to play laser during rendering */
laserSound.setPosition(cam.getLocation());
snode.onEvent(laserEventID);
```

Now, lets look at BulletMover(). Notice it is a controller just like KeyframeController and SpatialController. Its purpose is to move a spatial in a given direction. Every time updateGeometricState() is called on the bullet (to which the BulletMover() is attached) or bullet's parent (rootNode), the function update is called on all of bullet's controllers. Lets look at update() :

```
public void update(float time) {
```

The *time* is the time between successive frames. Why? Because in that's the float value given to updateGeometricState() in SimpleGame's update method. Next, I see if I can remove my bullet:

```
lifeTime -= time;
/** If life is gone, remove it */
if (lifeTime < 0) {
    rootNode.detachChild(bullet);
    bullet.removeController(this);
    return;
}
```

If the bullet has been alive for more than 5 seconds, I remove it. Simple enough, right? Next, I move the bullet in its direction:

```
                /** Move bullet */
                Vector3f bulletPos = bullet.getLocalTranslation();
                bulletPos.addLocal(direction.mult(time * speed));
                bullet.setLocalTranslation(bulletPos);
```

The float *time* is the value given to update() and the float speed is just a constant for the bullet that makes it move faster. Next, I check to see if the bullet has hit my target:

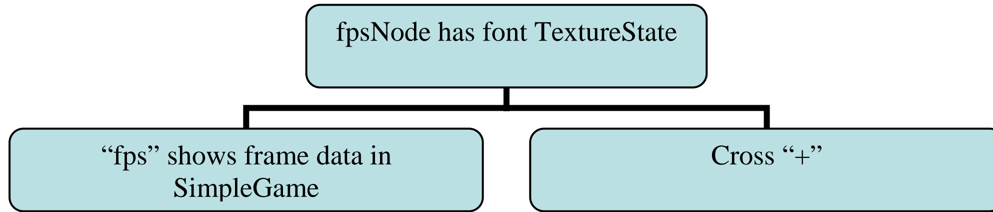
```
                /** Does the bullet intersect with target? */
                if (Intersection.intersection(bullet.getWorldBound(), target
                        .getWorldBound())) {
```

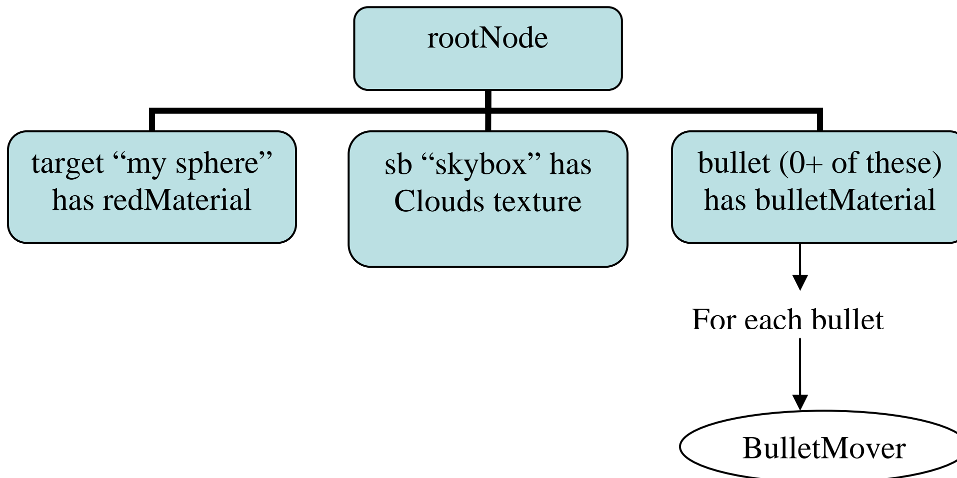
Note that I am checking if the bullet's worldBound intersects the target's worldbound. A BoundingVolume for a mesh can be bigger than the actual mesh, which means that two bounds can intersect while the objects don't. If this were a real game, I would add a more complex intersection method after this one to see if the two TriMesh objects intersect, but for now we're keeping it simple. If the bounds intersect, I simply move the target and kill the bullet. The only new things left are in simpleRender and simpleUpdate:

```
        /**
         * Called every frame for rendering
         */
        protected void simpleRender() {
            // Give control to the sound in case sound changes are needed.
            SoundAPIController.getRenderer().draw(snode);
        }

        /**
         * Called every frame for updating
         */
        protected void simpleUpdate() {
            // Let the programmable sound update itself.
            snode.updateGeometricState(tpf, true);
        }
```

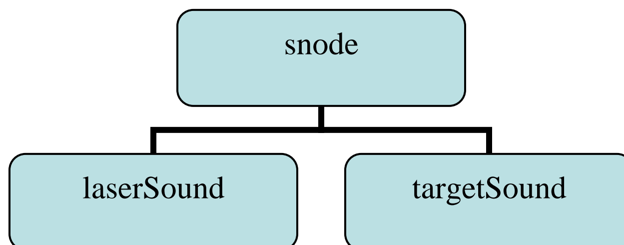
I have to update and render my sounds just like SimpleGame does for my rootNode, and so that's what I'm doing here. The variable "tpf" is created in SimpleGame. It is the "Time per frame," and it is updated every frame.

Here is a graph of my game:





And here is a scene graph of my sound nodes:



Challenge:

Add a sound for when the sphere is hit.
Make the target sphere move.

11) Hello SimpleGame

This program introduces WireframeState, Timer, draw(), camera creation, and displaysystem creation. You will learn how to create and customize your own SimpleGame class. It will also take away some of the mystery of SimpleGame and what it does.

```
import com.jme.app.SimpleGame;
import com.jme.app.BaseGame;
import com.jme.renderer.Camera;
import com.jme.renderer.ColorRGBA;
import com.jme.scene.Node;
import com.jme.scene.Text;
import com.jme.scene.shape.Box;
import com.jme.scene.state.*;
import com.jme.input.*;
import com.jme.util.Timer;
import com.jme.util.TextureManager;
import com.jme.util.LoggingSystem;
import com.jme.system.DisplaySystem;
import com.jme.system.JmeException;
import com.jme.math.Vector3f;
import com.jme.image.Texture;
import com.jme.light.PointLight;

import java.util.logging.Level;

/**
 * Started Date: Jul 29, 2004<br><br>
 *
 * Is used to demonstrate the inner workings of SimpleGame.
 *
 * @author Jack Lindamood
 */
public class HelloSimpleGame extends BaseGame {
    public static void main(String[] args) {
        HelloSimpleGame app = new HelloSimpleGame();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    /** The camera that we see through. */
    protected Camera cam;
    /** The root of our normal scene graph. */
    protected Node rootNode;
    /** Handles our mouse/keyboard input. */
    protected InputHandler input;
    /** High resolution timer for jME. */
    protected Timer timer;
    /** The root node of our text. */
    protected Node fpsNode;
    /** Displays all the lovely information at the bottom. */
    protected Text fps;
    /** Simply an easy way to get at timer.getTimePerFrame(). */
    protected float tpf;
    /** True if the renderer should display bounds. */
    protected boolean showBounds = false;
```

```java
/** A wirestate to turn on and off for the rootNode */
protected WireframeState wireState;
/** A lightstate to turn on and off for the rootNode */
protected LightState lightState;

/** Location of the font for jME's text at the bottom */
public static String fontLocation = "com/jme/app/defaultfont.tga";

/**
 * This is called every frame in BaseGame.start()
 * @param interpolation unused in this implementation
 * @see com.jme.app.AbstractGame#update(float interpolation)
 */
protected final void update(float interpolation) {
  /** Recalculate the framerate. */
  timer.update();
  /** Update tpf to time per frame according to the Timer. */
  tpf = timer.getTimePerFrame();
  /** Check for key/mouse updates. */
  input.update(tpf);
  /** Send the fps to our fps bar at the bottom. */
  fps.print("FPS: " + (int) timer.getFrameRate() + " - " +
            display.getRenderer().getStatistics());
  /** Call simpleUpdate in any derived classes of SimpleGame. */
  simpleUpdate();

  /**
  Update controllers/render states/transforms/bounds for
  rootNode.
  */
  rootNode.updateGeometricState(tpf, true);

  /**
  If toggle_wire is a valid command (via key T), change
  wirestates.
  */
  if (KeyBindingManager
      .getKeyBindingManager()
      .isValidCommand("toggle_wire", false)) {
    wireState.setEnabled(!wireState.isEnabled());
    rootNode.updateRenderState();
  }
  /**
  If toggle_lights is a valid command (via key L),
  change lightstate.
  */
  if (KeyBindingManager
      .getKeyBindingManager()
      .isValidCommand("toggle_lights", false)) {
    lightState.setEnabled(!lightState.isEnabled());
    rootNode.updateRenderState();
  }
```

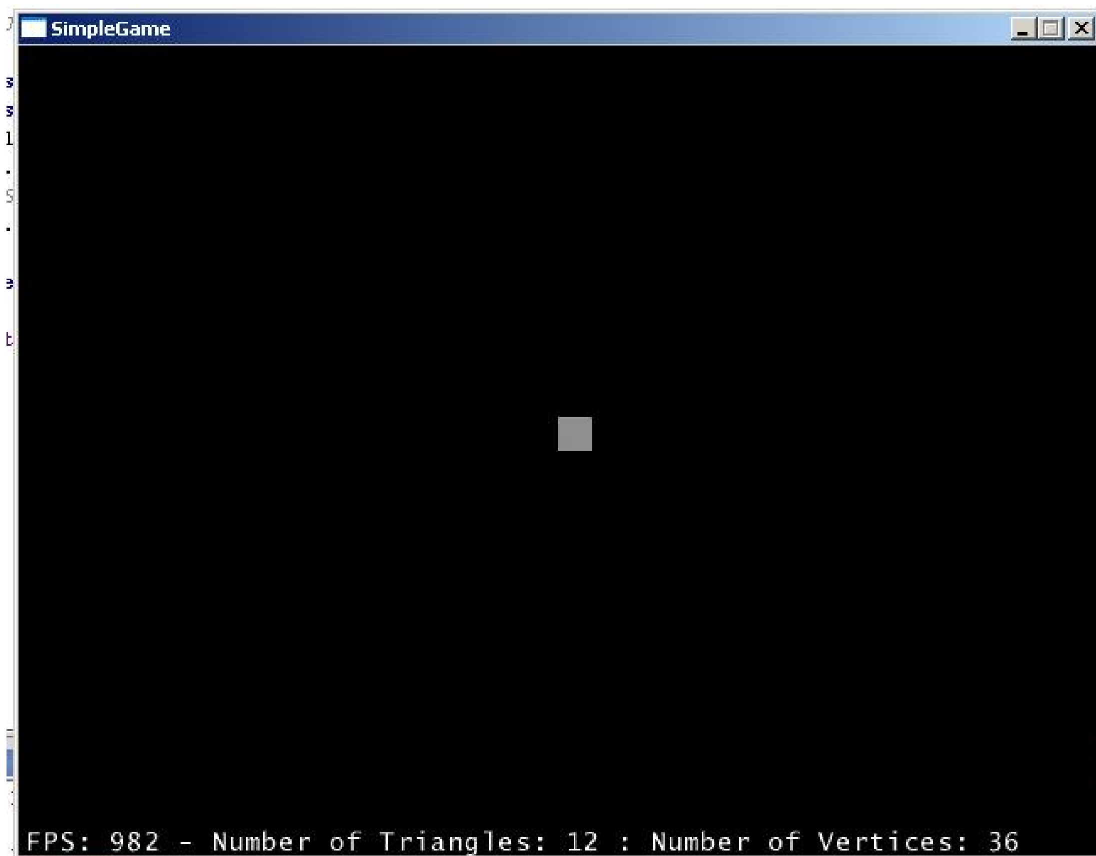
```
    /**
    If toggle_bounds is a valid command (via key B),
    change bounds.
    */
    if (KeyBindingManager
        .getKeyBindingManager()
        .isValidCommand("toggle_bounds", false)) {
      showBounds = !showBounds;
    }

    /**
    If camera_out is a valid command (via key C), show camera
    location.
    */
    if (KeyBindingManager
        .getKeyBindingManager()
        .isValidCommand("camera_out", false)) {
      System.err.println("Camera at: " +
                          display.getRenderer().getCamera().getLocation());
    }

  }

  /**
   * This is called every frame in BaseGame.start(), after update()
   * @param interpolation unused in this implementation
   * @see com.jme.app.AbstractGame#render(float interpolation)
   */
  protected final void render(float interpolation) {
    /**
    Reset display's tracking information for
    number of triangles/vertexes
    */
    display.getRenderer().clearStatistics();
    /** Clears the previously rendered information. */
    display.getRenderer().clearBuffers();
    /** Draw the rootNode and all its children. */
    display.getRenderer().draw(rootNode);
    /**
    If showing bounds, draw rootNode's bounds, and the
    bounds of all its children.
    */
    if (showBounds)
      display.getRenderer().drawBounds(rootNode);
      /** Draw the fps node to show the fancy information at the bottom. */
    display.getRenderer().draw(fpsNode);
      /** Call simpleRender() in any derived classes. */
    simpleRender();
  }

  /**
   * Creates display, sets up camera, and binds keys.
   * Called in BaseGame.start() directly after
   * the dialog box.
   * @see com.jme.app.AbstractGame#initSystem()
   */
```

```
    protected final void initSystem() {
      try {
        /**
        Get a DisplaySystem acording to the renderer
        selected in the startup box.
        */
        display = DisplaySystem.getDisplaySystem(properties.getRenderer());
         /** Create a window with the startup box's information. */
        display.createWindow(
            properties.getWidth(),
            properties.getHeight(),
            properties.getDepth(),
            properties.getFreq(),
            properties.getFullscreen());
        /**
        Create a camera specific to the DisplaySystem that works
        with
        the display's width and height
        */
        cam =
            display.getRenderer().createCamera(
            display.getWidth(),
            display.getHeight());

      } catch (JmeException e) {
        /**
        If the displaysystem can't be initialized correctly,
        exit instantly.
        */
        e.printStackTrace();
        System.exit(1);
      }

        /** Set a black background.*/
      display.getRenderer().setBackgroundColor(ColorRGBA.black);

      /** Set up how our camera sees. */
      cam.setFrustumPerspective(45.0f,
                                (float) display.getWidth() /
                                (float) display.getHeight(), 1, 1000);
      Vector3f loc = new Vector3f(0.0f, 0.0f, 25.0f);
      Vector3f left = new Vector3f( -1.0f, 0.0f, 0.0f);
      Vector3f up = new Vector3f(0.0f, 1.0f, 0.0f);
      Vector3f dir = new Vector3f(0.0f, 0f, -1.0f);
      /** Move our camera to a correct place and orientation. */
      cam.setFrame(loc, left, up, dir);
      /** Signal that we've changed our camera's location/frustum. */
      cam.update();
      /** Assign the camera to this renderer.*/
      display.getRenderer().setCamera(cam);

      /** Create a basic input controller. */
      input = new FirstPersonHandler(this, cam, properties.getRenderer());
      /** Signal to all key inputs they should work 10x faster. */
      input.setKeySpeed(10f);
      input.setMouseSpeed(1f);
```

```
    /** Get a high resolution timer for FPS updates. */
    timer = Timer.getTimer(properties.getRenderer());

    /** Sets the title of our display. */
    display.setTitle("SimpleGame");
    /**
    Signal to the renderer that it should keep track of
    rendering information.
    */
    display.getRenderer().enableStatistics(true);

    /** Assign key T to action "toggle_wire". */
    KeyBindingManager.getKeyBindingManager().set(
        "toggle_wire",
        KeyInput.KEY_T);
    /** Assign key L to action "toggle_lights". */
    KeyBindingManager.getKeyBindingManager().set(
        "toggle_lights",
        KeyInput.KEY_L);
    /** Assign key B to action "toggle_bounds". */
    KeyBindingManager.getKeyBindingManager().set(
        "toggle_bounds",
        KeyInput.KEY_B);
    /** Assign key C to action "camera_out". */
    KeyBindingManager.getKeyBindingManager().set(
        "camera_out",
        KeyInput.KEY_C);
  }

  /**
   * Creates rootNode, lighting, statistic text, and other basic
   * render states.
   * Called in BaseGame.start() after initSystem().
   * @see com.jme.app.AbstractGame#initGame()
   */
  protected final void initGame() {
      /** Create rootNode */
    rootNode = new Node("rootNode");

    /** Create a wirestate to toggle on and off.  Starts disabled with
     * default width of 1 pixel. */
    wireState = display.getRenderer().createWireframeState();
    wireState.setEnabled(false);
    rootNode.setRenderState(wireState);

    /** Create a ZBuffer to display pixels closest to the camera
    above farther ones.  */
    ZBufferState buf = display.getRenderer().createZBufferState();
    buf.setEnabled(true);
    buf.setFunction(ZBufferState.CF_LEQUAL);

    rootNode.setRenderState(buf);

    // -- FPS DISPLAY
    // First setup alpha state
```

```
/** This allows correct blending of text and what is already
rendered below it*/
AlphaState as1 = display.getRenderer().createAlphaState();
as1.setBlendEnabled(true);
as1.setSrcFunction(AlphaState.SB_SRC_ALPHA);
as1.setDstFunction(AlphaState.DB_ONE);
as1.setTestEnabled(true);
as1.setTestFunction(AlphaState.TF_GREATER);
as1.setEnabled(true);

// Now setup font texture
TextureState font = display.getRenderer().createTextureState();
/** The texture is loaded from fontLocation */
font.setTexture(
    TextureManager.loadTexture(
    SimpleGame.class.getClassLoader().getResource(
    fontLocation),
    Texture.MM_LINEAR,
    Texture.FM_LINEAR,
    true));
font.setEnabled(true);

// Then our font Text object.
/** This is what will actually have the text at the bottom. */
fps = new Text("FPS label", "");
fps.setForceView(true);
fps.setTextureCombineMode(TextureState.REPLACE);

// Finally, a stand alone node (not attached to root on purpose)
fpsNode = new Node("FPS node");
fpsNode.attachChild(fps);
fpsNode.setRenderState(font);
fpsNode.setRenderState(as1);
fpsNode.setForceView(true);

// ---- LIGHTS
/** Set up a basic, default light. */
PointLight light = new PointLight();
light.setDiffuse(new ColorRGBA(1.0f, 1.0f, 1.0f, 1.0f));
light.setAmbient(new ColorRGBA(0.5f, 0.5f, 0.5f, 1.0f));
light.setLocation(new Vector3f(100, 100, 100));
light.setEnabled(true);

/** Attach the light to a lightState and the lightState to rootNode. */
lightState = display.getRenderer().createLightState();
lightState.setEnabled(true);
lightState.attach(light);
rootNode.setRenderState(lightState);

/** Let derived classes initialize. */
simpleInitGame();

/** Update geometric and rendering information for both the
rootNode and fpsNode. */
rootNode.updateGeometricState(0.0f, true);
```

```
      rootNode.updateRenderState();
      fpsNode.updateGeometricState(0.0f, true);
      fpsNode.updateRenderState();
    }

    protected void simpleInitGame() {
        rootNode.attachChild(new Box("my box",
                             new Vector3f(0,0,0),
                             new Vector3f(1,1,1)));
    }

    /**
       * Can be defined in derived classes for custom updating.
       * Called every frame in update.
       */
    protected void simpleUpdate() {}

      /**
       * Can be defined in derived classes for custom rendering.
       * Called every frame in render.
       */
    protected void simpleRender() {}

    /**
     * unused
     * @see com.jme.app.AbstractGame#reinit()
     */
    protected void reinit() {
    }

    /**
     * Cleans up the keyboard.
     * @see com.jme.app.AbstractGame#cleanup()
     */
    protected void cleanup() {
      LoggingSystem.getLogger().log(Level.INFO, "Cleaning up resources.");
      input.getKeyBindingManager().getKeyInput().destroy();
      InputSystem.getMouseInput().destroy();
    }
}
```



Wow. So much you've taken for granted happens in SimpleGame. No place to start but the beginning:

```
    public static void main(String[] args) {
        HelloSimpleGame app = new HelloSimpleGame();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }
```

Let's look at app.start():

```
  public final void start() {
    LoggingSystem.getLogger().log(Level.INFO, "Application started.");
    try {
      getAttributes();
      initSystem();
      assertDisplayCreated();
      initGame();
      //main loop
      while (!finished && !display.isClosing()) {
        //update game state, do not use interpolation parameter
        update( -1.0f);
        //render, do not use interpolation parameter
        render( -1.0f);
        //swap buffers
```

```
        display.getRenderer().displayBackBuffer();
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
    cleanup();
    LoggingSystem.getLogger().log(Level.INFO, "Application ending.");
    display.reset();
    quit();
  }
```

Don't let it scare you. Here's the order:

1) getAttributes(): Get the attributes the user selects from the monkey dialog box
2) initSystem(): Call initSystem() to create the "system" part
3) assertDisplayCreated(): Make sure the system was created
4) initGame(): Create the "game" part beginning
5) while (): While the game is running
6) update(-1): Call update() to change object values
7) render(-1): Call render() to draw object values
8) displayBackBuffer() : Display what we draw all at once
9) loop until complete
10) cleanup() -- quit() : close out the application

This basic order exist in most rendering systems. Lets look at initSystem():

```
    protected final void initSystem() {
      try {
        /**
        Get a DisplaySystem acording to the renderer
        selected in the startup box.
        */
        display = DisplaySystem.getDisplaySystem(properties.getRenderer());
         /** Create a window with the startup box's information. */
        display.createWindow(
            properties.getWidth(),
            properties.getHeight(),
            properties.getDepth(),
            properties.getFreq(),
            properties.getFullscreen());
```

Here we use the values you input in the monkey screen to create our game's settings. Notice DisplaySystem is a singleton class that we initialize with the static function getDisplaySystem. What good is color without eyes? Next we create our camera:

```
        /**
        Create a camera specific to the DisplaySystem that works
        with
        the display's width and height
        */
        cam =
            display.getRenderer().createCamera(
            display.getWidth(),
```

```
                display.getHeight());
```

This creates a camera that uses the renderer's given width and height. Next I simply set the background color; nothing too complex here:

```
        /** Set a black background.*/
    display.getRenderer().setBackgroundColor(ColorRGBA.black);
```

Now for the math craziness! Next I setup the view frustum:

```
    /** Set up how our camera sees. */
    cam.setFrustumPerspective(45.0f,
                              (float) display.getWidth() /
                              (float) display.getHeight(), 1, 1000);
```

This frustum is how jME culls away objects that are not being viewed:

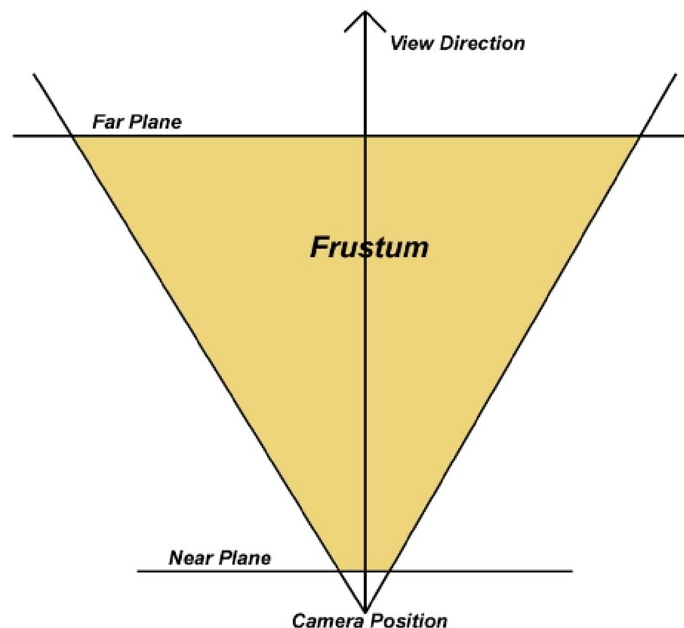


Figure 1: 2D View Frustum

I got this picture from [here](). This link also contains a very good article about Frustum Culling, which I highly recommend if you're interested in the inner workings of jME. As you can see, a frustum in our case is just a pyramid with the tip cut off. We define our frustum with the view angle, ratio of width/height on the screen, and the near/far planes. It looks something like the picture, except it's 3D. Everything we will see is in the orange. After defining what my camera will see, I move the camera back a bit and signal to the camera that I've made changes:

```
    Vector3f loc = new Vector3f(0.0f, 0.0f, 25.0f);
    Vector3f left = new Vector3f( -1.0f, 0.0f, 0.0f);
    Vector3f up = new Vector3f(0.0f, 1.0f, 0.0f);
```

```
        Vector3f dir = new Vector3f(0.0f, 0f, -1.0f);
        /** Move our camera to a correct place and orientation. */
        cam.setFrame(loc, left, up, dir);
        /** Signal that we've changed our camera's location/frustum. */
        cam.update();
        /** Assign the camera to this renderer.*/
        display.getRenderer().setCamera(cam);
```

The function setFrame() defines the camera's location with a position of the camera, the direction it's looking, and the directions of left and up according to the camera. After all of the camera information is set, I must update the camera (similar to updating renderstates, geometry, and sounds) and set the camera to the renderer. Next, we create basic movement keys using a predefined class called FirstPersonHandler:

```
        /** Create a basic input controller. */
        input = new FirstPersonHandler(this, cam, properties.getRenderer());
        /** Signal to all key inputs they should work 10x faster. */
        input.setKeySpeed(10f);
        input.setMouseSpeed(1f);
```

FirstPersonHandler extends InputHandler. InputHandler classes are used to control the inputs for your game from keyboards and mice. In your own game you would make a class extend InputHandler and define your own ways to handle inputs. For here, we just use FirstPersonHandler. It defines the mouse look and ASDW movement keys. Notice we set the key speed 10x faster. This tells the key readers in FirstPersonHandler to work 10x faster. You could, for example, use this in your game to toggle running and walking. Next, I get a timer to use for my FPS updates:

```
        /** Get a high resolution timer for FPS updates. */
        timer = Timer.getTimer(properties.getRenderer());
```

This timer uses 'ticks' to get a very accurate measure of time between frames. We will use the timer in our updating. Near the end, I enable statistic counting and set a title for my display:

```
        /** Sets the title of our display. */
        display.setTitle("SimpleGame");
        /**
        Signal to the renderer that it should keep track of
        rendering information.
        */
        display.getRenderer().enableStatistics(true);
```

Finally, I bind the keys T,L,B,C to various commands and am done with my system initialization.

Now that the system is initialized, I setup the game parts. I start with creating my rootNode, then I create renderstates for my rootNode:

```
        /** Create a wirestate to toggle on and off.  Starts disabled with
         * default width of 1 pixel. */
        wireState = display.getRenderer().createWireframeState();
        wireState.setEnabled(false);
```

```java
        rootNode.setRenderState(wireState);

        /** Create a ZBuffer to display pixels closest to the camera
        above farther ones.  */
        ZBufferState buf = display.getRenderer().createZBufferState();
        buf.setEnabled(true);
        buf.setFunction(ZBufferState.CF_LEQUAL);

        rootNode.setRenderState(buf);
```

The first is a wirestate. That's used whenever you press T. It draws the meshes as wires, not completely filled triangles. Afterwards I create a ZBufferState that is used to tell rootNode how to render pixels that are on top of each other on the screen. The intuitive way is to draw the ones closest to your eye, which is what I do by setting buf to draw LEQUAL or pixels less than or equal to the pixel already there. After rootNode has its renderstates, I create the ones I need for the fpsNode which will be used to display the text at the bottom of the screen. First is the alpha state:

```java
        /** This allows correct blending of text and what is already
        rendered below it*/
        AlphaState as1 = display.getRenderer().createAlphaState();
        as1.setBlendEnabled(true);
        as1.setSrcFunction(AlphaState.SB_SRC_ALPHA);
        as1.setDstFunction(AlphaState.DB_ONE);
        as1.setTestEnabled(true);
        as1.setTestFunction(AlphaState.TF_GREATER);
        as1.setEnabled(true);
```

This is needed for the same reason I needed it for the mouse example. It allows correct blending of the texture that represents the text and the background it is drawn on. After the correct alphastate, I set the correct text:

```java
        // Now setup font texture
        TextureState font = display.getRenderer().createTextureState();
        /** The texture is loaded from fontLocation */
        font.setTexture(
            TextureManager.loadTexture(
            SimpleGame.class.getClassLoader().getResource(
            fontLocation),
            Texture.MM_LINEAR,
            Texture.FM_LINEAR,
            true));
        font.setEnabled(true);
```

The fpsNode will use *font* to display the correct text on the screen. Next, I create the Text object I use at the bottom of my screen:

```java
        // Then our font Text object.
        /** This is what will actually have the text at the bottom. */
        fps = new Text("FPS label", "");
        fps.setForceView(true);
        fps.setTextureCombineMode(TextureState.REPLACE);
```

I force it to be used, and allow users the option of replacing fps's font if they want. By default it will just use fpsNode's font. Finally, I set up my fpsNode. I purposely don't attach it to rootNode because text shouldn't be manipulated the same way as world objects.

```
        // Finally, a stand alone node (not attached to root on purpose)
        fpsNode = new Node("FPS node");
        fpsNode.attachChild(fps);
        fpsNode.setRenderState(font);
        fpsNode.setRenderState(as1);
        fpsNode.setForceView(true);
```

The last thing I create are lights, which allow me to see what is going on. Lights are attached to LightStates, which are attached to Spatials. First, I create the light:

```
        /** Set up a basic, default light. */
        PointLight light = new PointLight();
        light.setDiffuse(new ColorRGBA(1.0f, 1.0f, 1.0f, 1.0f));
        light.setAmbient(new ColorRGBA(0.5f, 0.5f, 0.5f, 1.0f));
        light.setLocation(new Vector3f(100, 100, 100));
        light.setEnabled(true);
```

It is a PointLight (like a lightbulb) with some basic ambient and diffuse colors, located at position 100,100,100. The next step after creating a light is to attach it to a LightState and put the LightState into my rootNode so the light can shine on all the objects below rootNode:

```
        /** Attach the light to a lightState and the lightState to rootNode. */
        lightState = display.getRenderer().createLightState();
        lightState.setEnabled(true);
        lightState.attach(light);
        rootNode.setRenderState(lightState);
```

None of this should be new. After letting users modify their game like they want with simpleInitGame, geometry and renderstates are updated:

```
        /** Let derived classes initialize. */
        simpleInitGame();

        /** Update geometric and rendering information for both the
        rootNode and fpsNode. */
        rootNode.updateGeometricState(0.0f, true);
        rootNode.updateRenderState();
        fpsNode.updateGeometricState(0.0f, true);
        fpsNode.updateRenderState();
```

The *0* is a time value passed to all controllers and the *true* signals that the function should update things at and below the object. The functions updateGeometricState and updateRenderState must be called whenever geometry or renderstates are changed. After setting up the game correctly, we enter an infinite while loop of update/render. First, let's look at update. The first thing in update is to get the correct time between frames:

```
        /** Recalculate the framerate. */
        timer.update();
        /** Update tpf to time per frame according to the Timer. */
        tpf = timer.getTimePerFrame();
```

This is done by updating my timer object and asking for the time in seconds between this update call and the last. Once I have the correct time per frame, I check for key updates inside input, update the text at the bottom of the screen, and allow users to update their own things if they wish:

```
        /** Check for key/mouse updates. */
        input.update(tpf);
        /** Send the fps to our fps bar at the bottom. */
        fps.print("FPS: " + (int) timer.getFrameRate() + " - " +
                  display.getRenderer().getStatistics());
        /** Call simpleUpdate in any derived classes of SimpleGame. */
        simpleUpdate();
```

Once the scene graph is changed, I update the geometry information of everything in the scene graph, from rootNode down:

```
        /**
        Update controllers/render states/transforms/bounds for
        rootNode.
        */
        rootNode.updateGeometricState(tpf, true);
```

The final part of update() is some simple checking for key presses. Note that if a key press changes a renderstate, the function updateRenderState() is called. After updating, rendering occurs. The first part of rendering is to clear statistical information:

```
        /**
        Reset display's tracking information for
        number of triangles/vertexes
        */
        display.getRenderer().clearStatistics();
```

Statistical information for jME include the vertex and triangle counters you see at the bottom. Whenever a vertex or triangle is rendered, a counter increases. This is just a simple vertex=0;triangle=0; so that stat information is reset every frame. Next we clear the buffers:

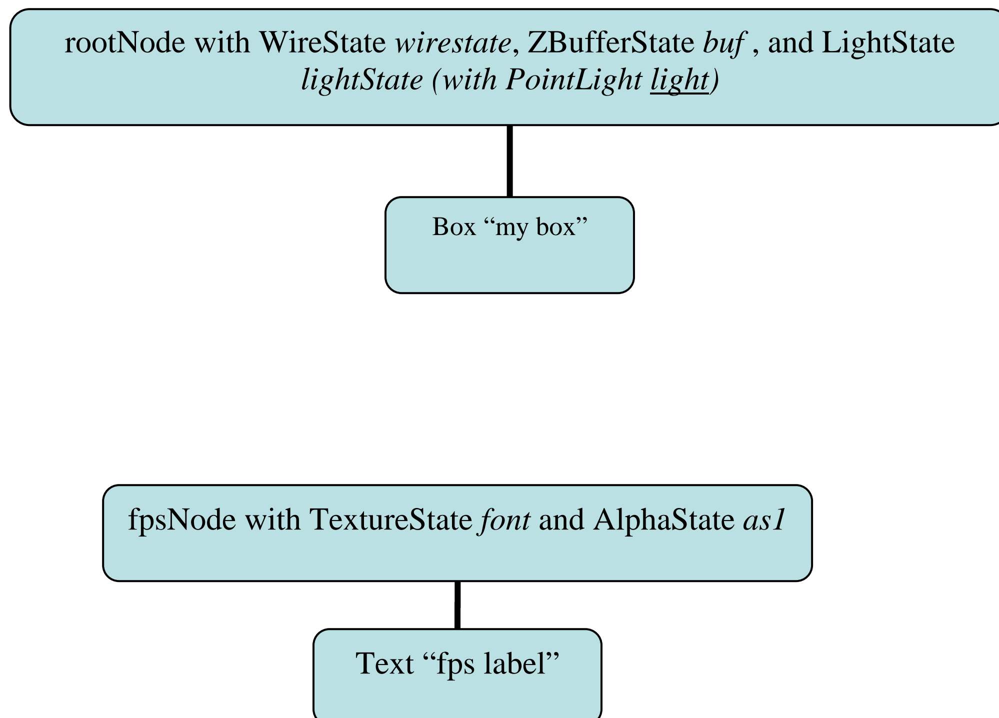
```
        /** Clears the previously rendered information. */
        display.getRenderer().clearBuffers();
```

This erases what we've previously drawn so we can draw something new. Without it, the frames would appear to stack on top of one another. Next, I draw the rootNode and fpsNode:

```
        /** Draw the rootNode and all its children. */
        display.getRenderer().draw(rootNode);
        /**
        If showing bounds, draw rootNode's bounds, and the
        bounds of all its children.
```

```
        */
        if (showBounds)
          display.getRenderer().drawBounds(rootNode);
          /** Draw the fps node to show the fancy information at the bottom. */
        display.getRenderer().draw(fpsNode);
          /** Call simpleRender() in any derived classes. */
        simpleRender();
```

I use display to get whatever rendering environment you're currently using (most likely LWJGL) and use its draw method to draw the node, which will automatically draw its children. Once done, I allow users to define their own rendering with simpleRender. That's it! All of that goes on behind our back when we use SimpleGame. Here's a scene graph of SimpleGame that is more complete than the one we've been using:



Challenge:

Display the camera's position every frame in the top, right corner of the scree

12. Hello LOD

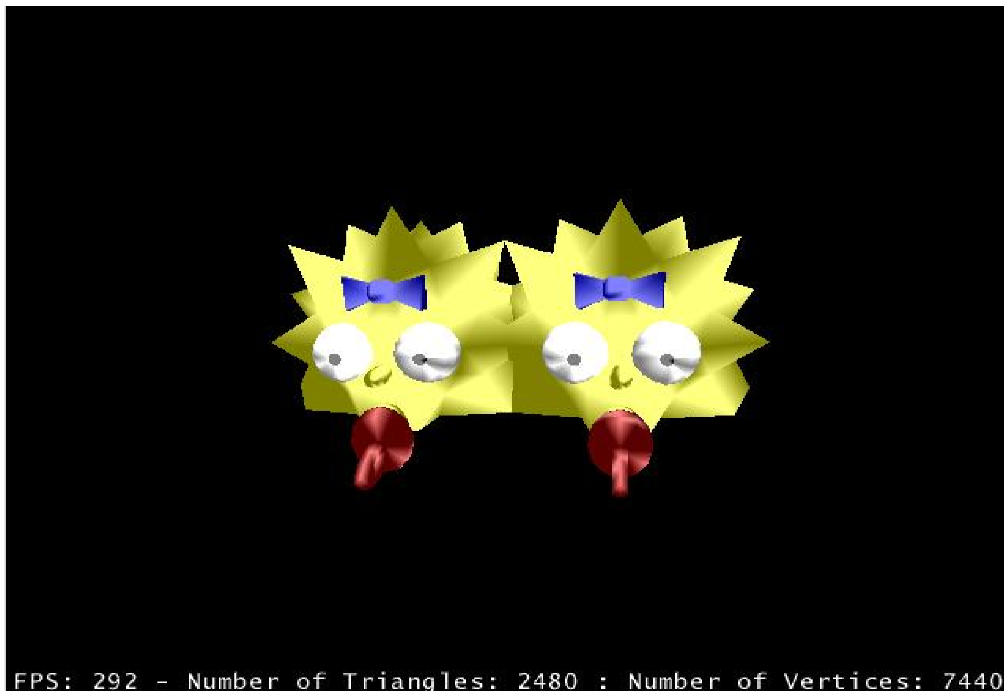
This program introduces AreaClodMesh, BezierCurve, CurveController, and CameraNode. You will learn how to use AreaClodMesh to increase FPS, as well as how to move the camera along a curved path.

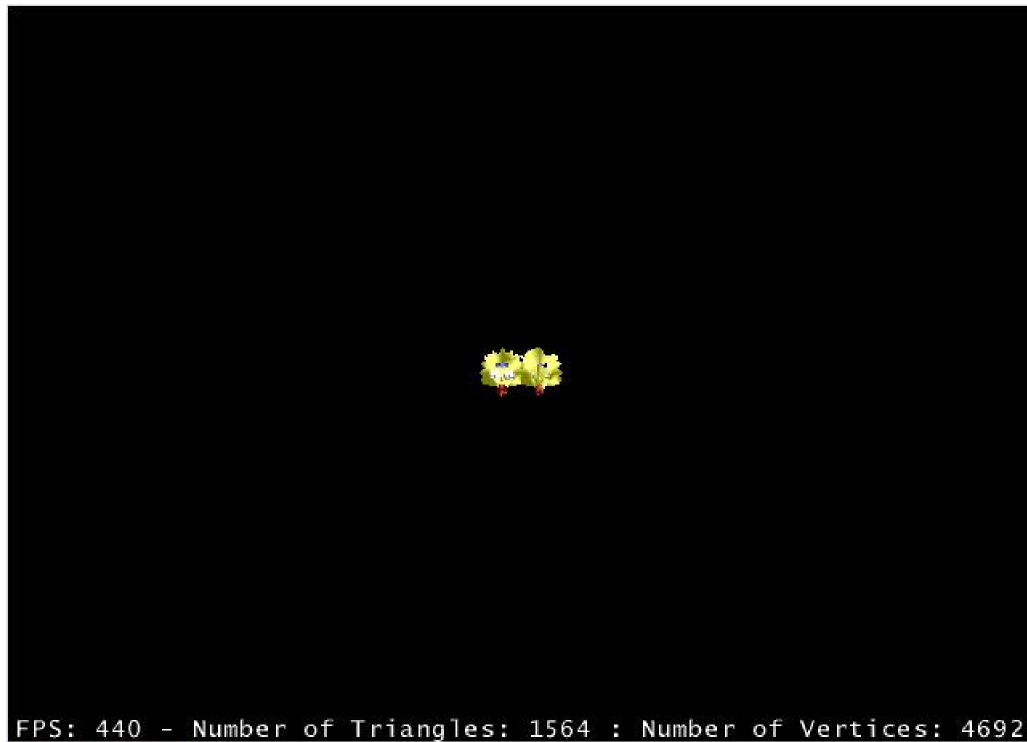
```
import com.jme.app.SimpleGame;
import com.jme.scene.model.XMLparser.Converters.FormatConverter;
import com.jme.scene.model.XMLparser.Converters.ObjToJme;
import com.jme.scene.model.XMLparser.JmeBinaryReader;
import com.jme.scene.Node;
import com.jme.scene.TriMesh;
import com.jme.scene.CameraNode;
import com.jme.scene.Controller;
import com.jme.scene.state.RenderState;
import com.jme.scene.lod.AreaClodMesh;
import com.jme.bounding.BoundingSphere;
import com.jme.math.Vector3f;
import com.jme.math.Matrix3f;
import com.jme.curve.CurveController;
import com.jme.curve.BezierCurve;
import com.jme.input.KeyInput;
import com.jme.input.action.KeyExitAction;

import java.net.URL;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ByteArrayInputStream;

/**
 * Started Date: Aug 16, 2004<br><br>
 *
 * This program teaches Complex Level of Detail mesh objects.  To use this
program, move
 * the camera backwards and watch the model disappear.
 *
 * @author Jack Lindamood
 */
public class HelloLOD extends SimpleGame {

    CameraNode cn;

    public static void main(String[] args) {
        HelloLOD app = new HelloLOD();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        // Point to a URL of my model
        URL model =
                HelloModelLoading.class.getClassLoader().
                getResource("jmetest/data/model/maggie.obj");
```

```
// Create something to convert .obj format to .jme
FormatConverter converter=new ObjToJme();
// Point the converter to where it will find the .mtl file from
converter.setProperty("mtllib",model);

// This byte array will hold my .jme file
ByteArrayOutputStream BO=new ByteArrayOutputStream();
// This will read the .jme format and convert it into a scene graph
JmeBinaryReader jbr=new JmeBinaryReader();

// Use an exact BoundingSphere bounds
BoundingSphere.useExactBounds=true;

Node meshParent=null;
try {
    // Use the format converter to convert .obj to .jme
    converter.convert(model.openStream(), BO);

    // Load the binary .jme format into a scene graph
    Node maggie = jbr.loadBinaryFormat(new
                        ByteArrayInputStream(BO.toByteArray()));
    meshParent=(Node) maggie.getChild(0);

} catch (IOException e) {   // Just in case anything happens
    System.out.println("Damn exceptions!" + e);
    e.printStackTrace();
    System.exit(0);
}

// Create a clod duplicate of meshParent.
Node clodNode=getClodNodeFromParent(meshParent);

// Attach the clod mesh at the origin.
clodNode.setLocalScale(.1f);
rootNode.attachChild(clodNode);

// Attach the original at -15,0,0
meshParent.setLocalScale(.1f);
meshParent.setLocalTranslation(new Vector3f(-15,0,0));

rootNode.attachChild(meshParent);

// Clear the keyboard commands that can move the camera.
input.clearKeyboardActions();
input.clearMouseActions();
// Insert a keyboard command that can exit the application.
input.addKeyboardAction("exit",KeyInput.KEY_ESCAPE,
                        new KeyExitAction(this));

// The path the camera will take.
Vector3f[]cameraPoints=new Vector3f[]{
    new Vector3f(0,5,20),
    new Vector3f(0,20,90),
    new Vector3f(0,30,200),
    new Vector3f(0,100,300),
    new Vector3f(0,150,400),
```

```java
        };
        // Create a path for the camera.
        BezierCurve bc=new BezierCurve("camera path",cameraPoints);

        // Create a camera node to move along that path.
        cn=new CameraNode("camera node",cam);

        // Create a curve controller to move the CameraNode along the path
        CurveController cc=new CurveController(bc,cn);

        // Cycle the animation.
        cc.setRepeatType(Controller.RT_CYCLE);

        // Slow down the curve controller a bit
        cc.setSpeed(.25f);

        // Add the controller to the node.
        cn.addController(cc);

        // Attach the node to rootNode
        rootNode.attachChild(cn);
    }

    private Node getClodNodeFromParent(Node meshParent) {
        // Create a node to hold my cLOD mesh objects
        Node clodNode=new Node("Clod node");
        // For each mesh in maggie
        for (int i=0;i<meshParent.getQuantity();i++){
            // Create an AreaClodMesh for that mesh.  Let it compute
            // records automatically
            AreaClodMesh acm=new AreaClodMesh("part"+i,
                                 (TriMesh) meshParent.getChild(i),null);
            acm.setModelBound(new BoundingSphere());
            acm.updateModelBound();

            // Allow 1/2 of a triangle in every pixel on the screen in
            // the bounds.
            acm.setTrisPerPixel(.5f);

            // Force a move of 2 units before updating the mesh geometry
            acm.setDistanceTolerance(2);

            // Give the clodMesh node the material state that the
            // original had.
            acm.setRenderState(meshParent.getChild(i).
                         getRenderStateList()[RenderState.RS_MATERIAL]);

            // Attach clod node.
            clodNode.attachChild(acm);
        }
        return clodNode;
    }

    Vector3f up=new Vector3f(0,1,0);
    Vector3f left=new Vector3f(1,0,0);
```

```
    private static Vector3f tempVa=new Vector3f();
    private static Vector3f tempVb=new Vector3f();
    private static Vector3f tempVc=new Vector3f();
    private static Vector3f tempVd=new Vector3f();
    private static Matrix3f tempMa=new Matrix3f();

    protected void simpleUpdate(){
        // Get the center of root's bound.
        Vector3f objectCenter=rootNode.getWorldBound().getCenter(tempVa);

        // My direction is the place I want to look minus the location
        // of the camera.
        Vector3f lookAtObject = tempVb.set(objectCenter).
                        subtractLocal(cam.getLocation()).normalizeLocal();

        // Left vector
        tempMa.setColumn(0,up.cross(lookAtObject,tempVc).normalizeLocal());
        // Up vector
        tempMa.setColumn(1,left.cross(lookAtObject,tempVd).normalizeLocal());
        // Direction vector
        tempMa.setColumn(2,lookAtObject);

        cn.setLocalRotation(tempMa);
    }
}
```





This program loads a model then goes through the model's children and creates a Complex Level of Detail mesh for each child and puts the two versions side by side so you can compare. The camera's movement is controlled by the program through a CameraNode.

The beginning is simply a copy from HelloModelLoading, so I'm not going to go over it. The first new part is when I create a clod duplicate from my original Maggie model.

```
    private Node getClodNodeFromParent(Node meshParent) {
        // Create a node to hold my cLOD mesh objects
        Node clodNode=new Node("Clod node");
        // For each mesh in maggie
        for (int i=0;i<meshParent.getQuantity();i++){
            // Create an AreaClodMesh for that mesh.  Let it compute
            // records automatically
            AreaClodMesh acm=new AreaClodMesh("part"+i,
                             (TriMesh) meshParent.getChild(i),null);
            acm.setModelBound(new BoundingSphere());
            acm.updateModelBound();

            // Allow 1/2 of a triangle in every pixel on the screen in
            // the bounds.
            acm.setTrisPerPixel(.5f);

            // Force a move of 2 units before updating the mesh geometry
            acm.setDistanceTolerance(2);

            // Give the clodMesh node the material state that the
            // original had.
```

```
            acm.setRenderState(meshParent.getChild(i).
                        getRenderStateList()[RenderState.RS_MATERIAL]);

            // Attach clod node.
            clodNode.attachChild(acm);
        }
        return clodNode;
    }
```

A ClodMesh is a "complex level of detail mesh". It allows users to trim away a few triangles at a time from a mesh, resulting in a figure that looks similar to the original but uses less information. This makes rendering quicker. For example, characters in your game need to look really good so you may put 5,000 polygons in them but those polygons are wasted when the character is 1,000ft away. From 1,000ft the character may as well have 400 polygons. The difference won't be too noticable to the user playing your game, but will be very big for their computer. As a general rule objects that take up more space on the screen should have more polygons.

A ClodMesh can only be created from a TriMesh. Because Maggie is actually multiple TriMesh objects inside one model (one trimesh is yellow, another is blue, and so on), I have to create new ClodMesh objects for each TriMesh. The type of ClodMesh I create is called an AreaClodMesh. AreaClodMesh uses the area on the screen the object's bounding volume occupies to determine how many triangles the mesh should have. Close objects will have all their triangles and far objects will have fewer triangles.

```
            AreaClodMesh acm=new AreaClodMesh("part"+i,
                            (TriMesh) meshParent.getChild(i),null);
```

The first parameter is the name of the ClodMesh. All spatials must have a name. The second parameter is the TriMesh to create a ClodMesh from and the third is the ClodMesh's *records*. Records tell the ClodMesh how to collapse triangles. When null is passed, the records are created for us.

```
            // Allow 1/2 of a triangle in every pixel on the screen in
            // the bounds.
            acm.setTrisPerPixel(.5f);
```

This function tells the ClodMesh how quickly to collapse triangles. I've set it intentionally small which is why you can tell how much Maggie changes. In your game, you would set this value where the collapsing of the TriMesh isn't noticeable. The ClodMesh will calculate how much area the bounding volume of *acm* takes on the screen and use that to figure out how to collapse the mesh's triangles.

```
            // Force a move of 2 units before updating the mesh geometry
            acm.setDistanceTolerance(2);
```

This forces a collapse update every time the camera's distance is changed by 2 units. A collapse update every frame would be extremely slow. Playing with this value allows the AutoClodMesh to update less frequently resulting in higher FPS.

```
            // Give the clodMesh node the material state that the
            // original had.
            acm.setRenderState(meshParent.getChild(i).
```

```
                              getRenderStateList()[RenderState.RS_MATERIAL]);
```

Because the *acm* has the original's geometry doesn't mean it has the original's render states. Because of this, I set the *acm's* material state to be the same as the original's material state. After creating a correct AreaClodMesh Node, I setup the key actions.

```
        // Clear the keyboard commands that can move the camera.
        input.clearKeyboardActions();
        input.clearMouseActions();
        // Insert a keyboard command that can exit the application.
        input.addKeyboardAction("exit",KeyInput.KEY_ESCAPE,
                                new KeyExitAction(this));
```

In this program I don't want users controlling the camera like they normally would with the keyboard so I clear all mouse and keyboard input actions. I have to reinsert a KeyExitAction otherwise users would have no way of closing the application. Finally too create the path for my camera.

```
        // The path the camera will take.
        Vector3f[]cameraPoints=new Vector3f[]{
            new Vector3f(0,5,20),
            new Vector3f(0,20,90),
            new Vector3f(0,30,200),
            new Vector3f(0,100,300),
            new Vector3f(0,150,400),
        };
        // Create a path for the camera.
        BezierCurve bc=new BezierCurve("camera path",cameraPoints);
```

I define points that will be the path of my camera and create a BezierCurve out of them. A curve is actually a spatial (which is why it needs a name), but this curve won't be rendered; instead it will be used as a path. After I have the curve setup, I create a CameraNode to move.

```
        // Create a camera node to move along that path.
        cn=new CameraNode("camera node",cam);
```

A CameraNode is a Spatial that can be translated and rotated just like spatials so that it can move around and with objects. The variable *cam* is created in SimpleGame. The above is equivalent to the following:

```
        // Create a camera node to move along that path.
        cn=new CameraNode("camera node",display.getRenderer().getCamera());
```

With the camera created, I can now create a controller for it.

```
        // Create a curve controller to move the CameraNode along the path
        CurveController cc=new CurveController(bc,cn);

        // Cycle the animation.
        cc.setRepeatType(Controller.RT_CYCLE);

        // Slow down the curve controller a bit
        cc.setSpeed(.25f);
```

```
        // Add the controller to the node.
        cn.addController(cc);

        // Attach the node to rootNode
        rootNode.attachChild(cn);
```

All of the above is pretty routine creation. A CurveController takes a spatial to move and a curve to move along. The only final part is rotating my CameraNode to face the two Maggie objects.

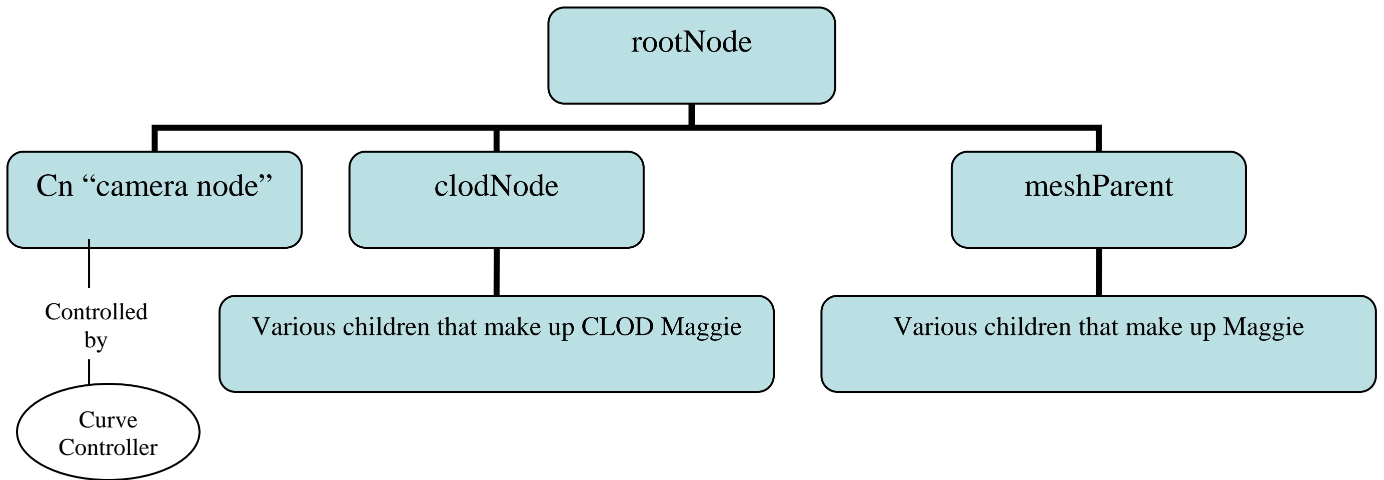
```
    protected void simpleUpdate(){
        // Get the center of root's bound.
        Vector3f objectCenter=rootNode.getWorldBound().getCenter(tempVa);

        // My direction is the place I want to look minus the location
       // of the camera.
        Vector3f lookAtObject = tempVb.set(objectCenter).
                        subtractLocal(cam.getLocation()).normalizeLocal();

        // Left vector
        tempMa.setColumn(0,up.cross(lookAtObject,tempVc).normalizeLocal());
        // Up vector
        tempMa.setColumn(1,left.cross(lookAtObject,tempVd).normalizeLocal());
        // Direction vector
        tempMa.setColumn(2,lookAtObject);

        cn.setLocalRotation(tempMa);
    }
```

Notice I created a lot of temporary static variables for this program. They are there just to assist with some math calls I will need in simpleUpdate. The rotation matrix of the CameraNode is used to orient the camera object. The CurveController is only there for the camera's position. The first two lines setup a vector that points from the camera towards the object. The last four are a little strange though:

```
        // Left vector
        tempMa.setColumn(0,up.cross(lookAtObject,tempVc).normalizeLocal());
        // Up vector
        tempMa.setColumn(1,left.cross(lookAtObject,tempVd).normalizeLocal());
        // Direction vector
        tempMa.setColumn(2,lookAtObject);

        cn.setLocalRotation(tempMa);
```

The 3x3 Rotation matrix is used to rotate the camera. The first column is the left vector the camera should use, the second is the camera's up vector, and the third is the camera's direction it is facing. I use the cross product of the up vector and the direction to get my left vector (similarly with the up vector). The cross of two vectors is the vector perpendicular to each. For more information on cross product, hit google. After the correct rotation matrix for the camera, I simply assign it to the camera node.

Scene graph:



Challenge: *Rewrite HelloIntersection to use AreaClodMesh for the bullets and target.*

13) Hello Terrain

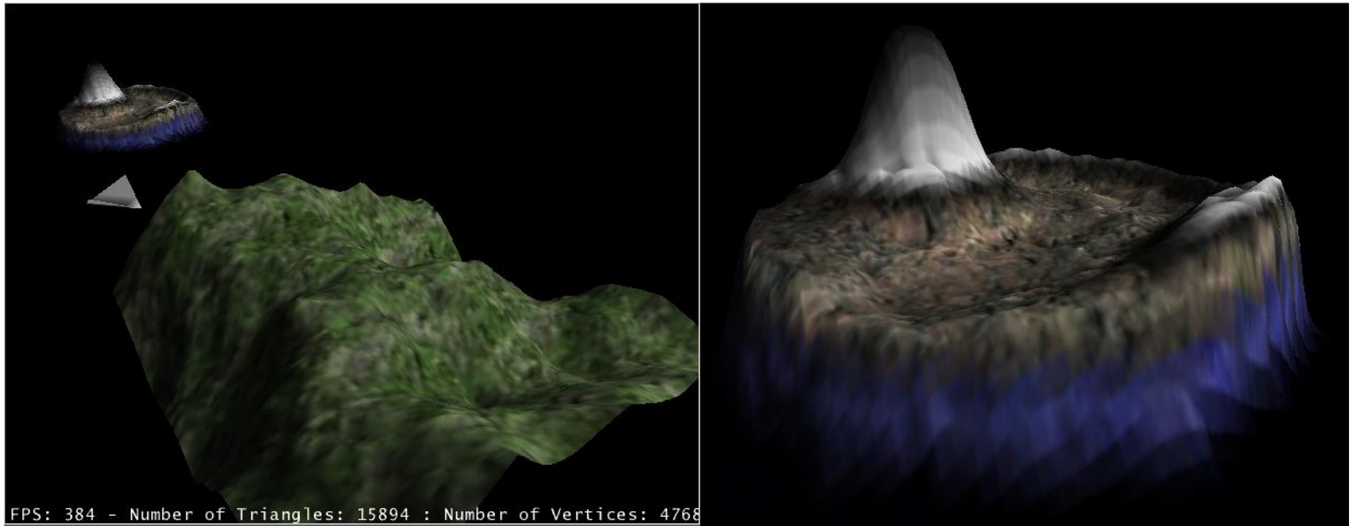
This program introduces jME's terrain utility classes. You will learn how to use ProceduralTextureGenerator, ImageBasedHeightMap, MidPointHeightMap, and TerrainBlock. All of this will allow you to create nice looking terrain with little effort.

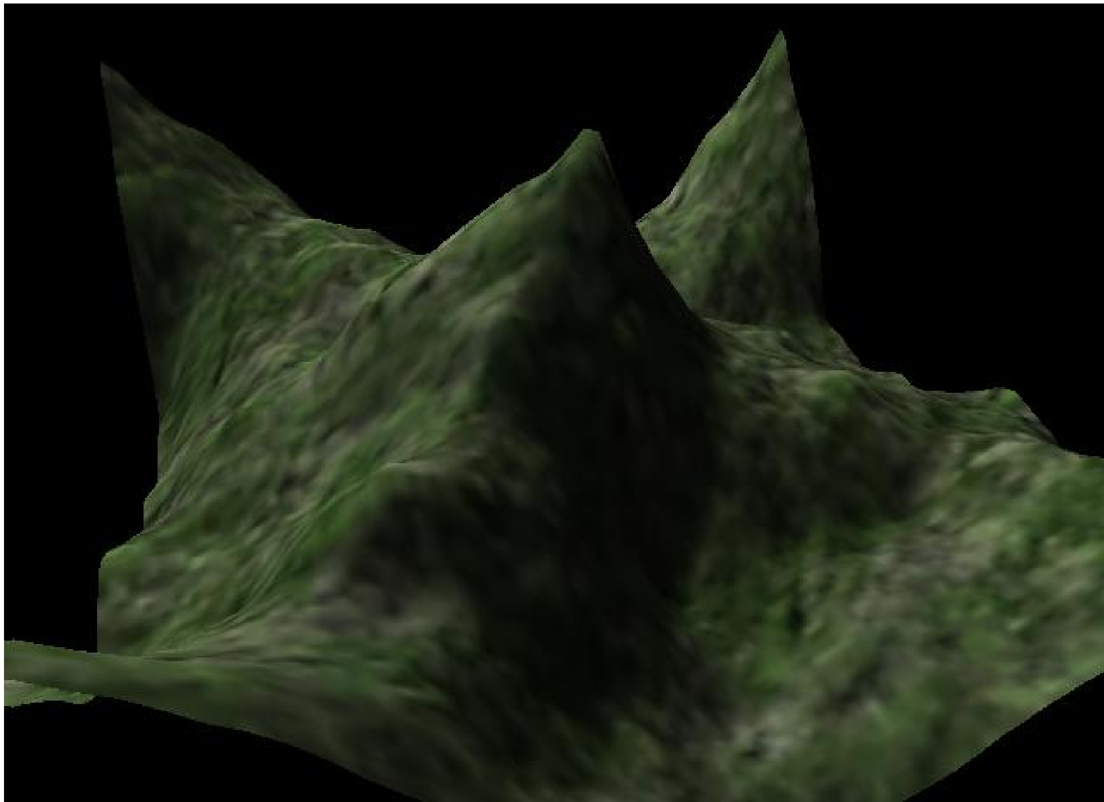
```
import com.jme.app.SimpleGame;
import com.jme.terrain.TerrainBlock;
import com.jme.terrain.util.MidPointHeightMap;
import com.jme.terrain.util.ImageBasedHeightMap;
import com.jme.terrain.util.ProceduralTextureGenerator;
import com.jme.math.Vector3f;
import com.jme.bounding.BoundingBox;
import com.jme.scene.state.TextureState;
import com.jme.util.TextureManager;
import com.jme.image.Texture;

import java.net.URL;

import javax.swing.*;

/**
 * Started Date: Aug 19, 2004<br><br>
 *
 * This program introduces jME's terrain utility classes and how
 * they are used.  It
 * goes over ProceduralTextureGenerator,
 * ImageBasedHeightMap, MidPointHeightMap, and
 * TerrainBlock.
 *
 * @author Jack Lindamood
 */
public class HelloTerrain extends SimpleGame {
    public static void main(String[] args) {
        HelloTerrain app = new HelloTerrain();
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        app.start();
    }

    protected void simpleInitGame() {
        // First a hand made terrain
        homeGrownHeightMap();
        // Next an automatically generated terrain with a texture
        generatedHeightMap();
        // Finally a terrain loaded from a greyscale image with
        // fancy textures on it.
        complexTerrain();
    }


    private void homeGrownHeightMap() {
        // The map for our terrain.  Each value is a height on the terrain
        int[] map=new int[]{
            1,2,3,4,
```

```
            2,1,2,3,
            3,2,1,2,
            4,3,2,1
        };

        // Create a terrain block.  Our integer height values will
        // scale on the map 2x larger x,
        // and 2x larger z.  Our map's origin will be the regular
        // origin, and it won't create an
        // AreaClodMesh from it.
        TerrainBlock tb=new TerrainBlock("block",4,
                new Vector3f(2,1,2),
                map,
                new Vector3f(0,0,0),
                false);

        // Give the terrain a bounding box.
        tb.setModelBound(new BoundingBox());
        tb.updateModelBound();

        // Attach the terrain TriMesh to our rootNode
        rootNode.attachChild(tb);
    }


    private void generatedHeightMap() {
        // This will be the texture for the terrain.
        URL grass=HelloTerrain.class.getClassLoader().getResource(
            "jmetest/data/texture/grassb.png");

        //  Use the helper class to create a terrain for us.  The
        // terrain will be 64x64
        MidPointHeightMap mph=new MidPointHeightMap(64,1.5f);
        // Create a terrain block from the created terrain map.
        TerrainBlock tb=new TerrainBlock("midpoint block",mph.getSize(),
                new Vector3f(1,.11f,1),
                mph.getHeightMap(),
                new Vector3f(0,-25,0),false);

        // Add the texture
        TextureState ts=display.getRenderer().createTextureState();
        ts.setTexture(
                TextureManager.loadTexture(grass,
                Texture.MM_LINEAR,Texture.FM_LINEAR,true)
        );
        tb.setRenderState(ts);

        // Give the terrain a bounding box.
        tb.setModelBound(new BoundingBox());
        tb.updateModelBound();

        // Attach the terrain TriMesh to rootNode
        rootNode.attachChild(tb);
    }

    private void complexTerrain() {
```

```
        // This grayscale image will be our terrain
        URL grayscale = HelloTerrain.
                             class.getClassLoader().
                             getResource("jmetest/data/texture/bubble.jpg");

        // These will be the textures of our terrain.
        URL waterImage=HelloTerrain.
                             class.getClassLoader().
                             getResource("jmetest/data/texture/water.png");
        URL dirtImage=HelloTerrain.
                             class.getClassLoader().
                             getResource("jmetest/data/texture/dirt.jpg");
        URL highest=HelloTerrain.
                             class.getClassLoader().
                             getResource("jmetest/data/texture/highest.jpg");


        //  Create an image height map based on the gray scale of our image.
        ImageBasedHeightMap ib=new ImageBasedHeightMap(
                new ImageIcon(grayScale).getImage()
        );
        // Create a terrain block from the image's grey scale
        TerrainBlock tb=new TerrainBlock("image icon",ib.getSize(),
                new Vector3f(.5f,.05f,.5f),ib.getHeightMap(),
                new Vector3f(0,0,0),false);

        //  Create an object to generate textured terrain from the
        // image based height map.
        ProceduralTextureGenerator pg=new ProceduralTextureGenerator(ib);
        //  Look like water from height 0-60 with the strongest
        // "water look" at 30
        pg.addTexture(new ImageIcon(waterImage),0,30,60);
        //  Look like dirt from height 40-120 with the strongest
        // "dirt look" at 80
        pg.addTexture(new ImageIcon(dirtImage),40,80,120);
        //  Look like highest (pure white) from height 110-256
        // with the strongest "white look" at 130
        pg.addTexture(new ImageIcon(highest),110,130,256);

        //  Tell pg to create a texture from the ImageIcon's it has recieved.
        pg.createTexture(256);
        TextureState ts=display.getRenderer().createTextureState();
        // Load the texture and assign it.
        ts.setTexture(
                TextureManager.loadTexture(
                        pg.getImageIcon().getImage(),
                        Texture.MM_LINEAR_LINEAR,
                        Texture.FM_LINEAR,
                        true,
                        true
                )
        );
        tb.setRenderState(ts);

        // Give the terrain a bounding box
        tb.setModelBound(new BoundingBox());
```

```java
        tb.updateModelBound();

        // Move the terrain in front of the camera
        tb.setLocalTranslation(new Vector3f(0,0,-50));

        // Attach the terrain to our rootNode.
        rootNode.attachChild(tb);
    }
}
```

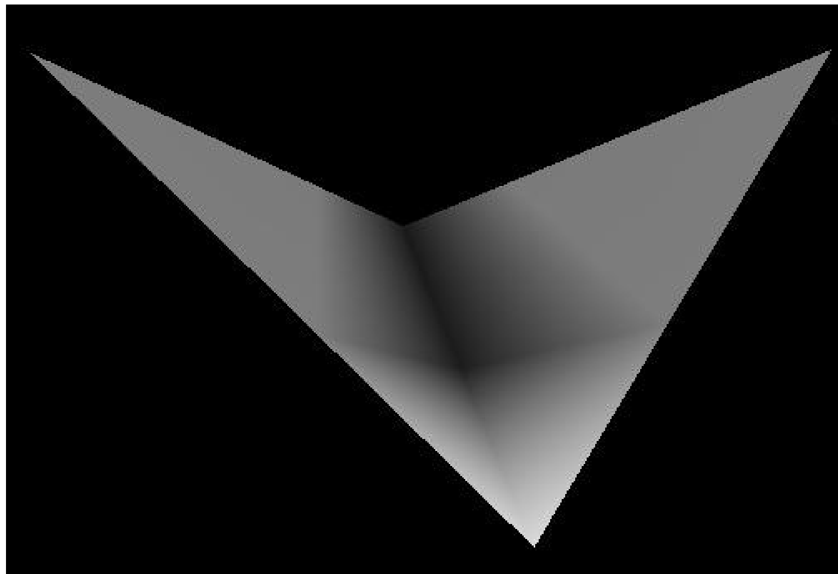




This program shows how jME's terrain package works. All Terrain comes from the base class TerrainBlock, which is a TriMesh. The TerrainBlock takes an array of integer values and creates a terrain from it. In this program I create 3 different types of terrain. The first is done by hand, the second uses a utility class to generate a terrain for us, and the third creates a terrain from a file. We'll start with the first.

```
// The map for our terrain.  Each value is a height on the terrain
int[] map=new int[]{
    1,2,3,4,
    2,1,2,3,
    3,2,1,2,
    4,3,2,1
};

// Create a terrain block.  Our integer height values will
// scale on the map 2x larger x,
// and 2x larger z.  Our map's origin will be the regular
// origin, and it won't create an
// AreaClodMesh from it.
TerrainBlock tb=new TerrainBlock("block",4,
        new Vector3f(2,1,2),
        map,
        new Vector3f(0,0,0),
        false);
```

Look at the *map* array. It is just an array of integer values. Each integer represents a height. This map will be low down the diagonal, and slope up just like the integer values do. This is exactly what it looks like when the program is run.



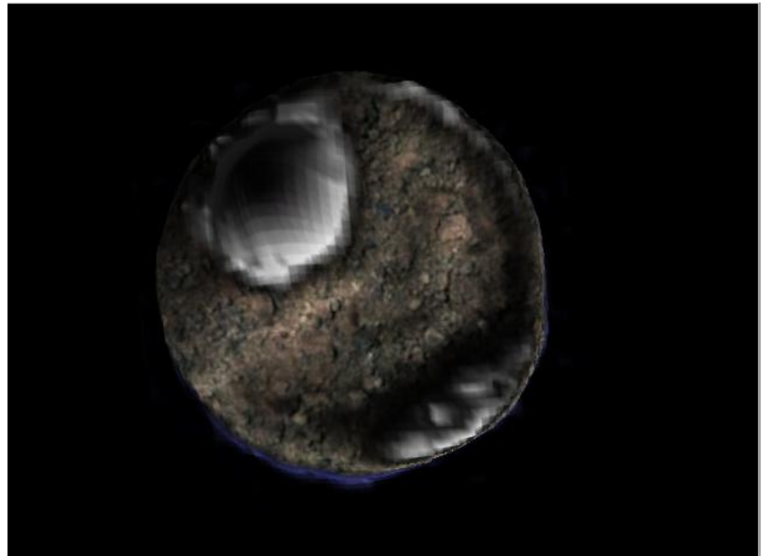
The terrain is a square. If you took the map [] and created a height for each number, it would look like this. This is how terrain is generated with TerrainBlock. The first parameter of TerrainBlock is the block's name (because all spatial objects must have a name). The next is the size of the block (4x4). The 3rd is a scale value for the terrain. We pass in integer terrain values, but what if we want float heights. The scale value for the terrain creates the terrain stretched twice as large along the x and z axis. Using this, you can manipulate the appearance and size of your terrain. The last two parameters are the origin for the terrain (0, 0, 0 for this example) and finally if we want to create the terrain as an AreaClodMesh. If true, the terrain will take up more memory but will on average generate faster FPS because it will be a clod.

Next, look at a generated terrain.

```
//  Use the helper class to create a terrain for us.  The
// terrain will be 64x64
MidPointHeightMap mph=new MidPointHeightMap(64,1.5f);
// Create a terrain block from the created terrain map.
TerrainBlock tb=new TerrainBlock("midpoint block",mph.getSize(),
        new Vector3f(1,.11f,1),
        mph.getHeightMap(),
        new Vector3f(0,-25,0),false);
```

Here we use MidPointHeightMap to generate a terrain for us. The terrain will be 64 by 64 and we give it a *roughness* value of 1.5. The roughness value changes how smooth the terrain becomes. You can play with it for the desired result. The only new thing about using TerrainBlock is we get our size and height map array from the midpoint height map.

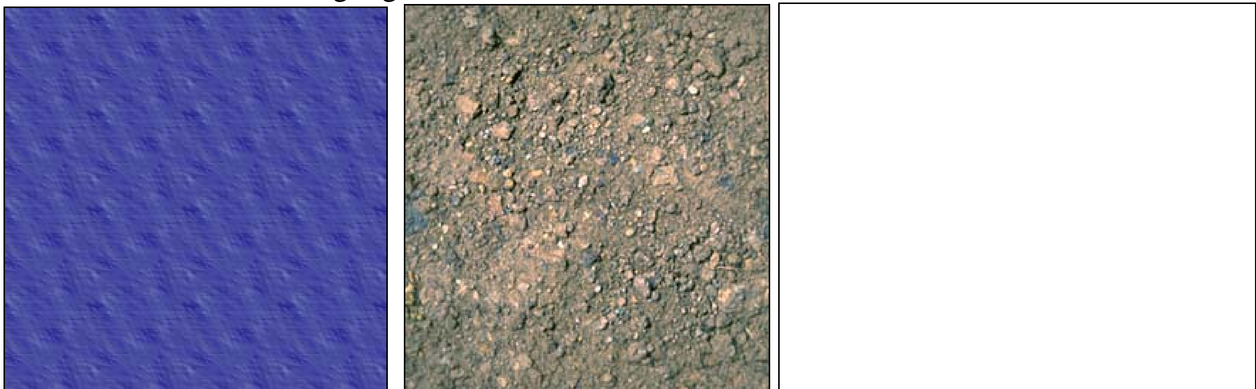
Finally we create a terrain from a gray scale image. First, take a look at the gray scale image we use, and then the terrain we create. Both views are from above.



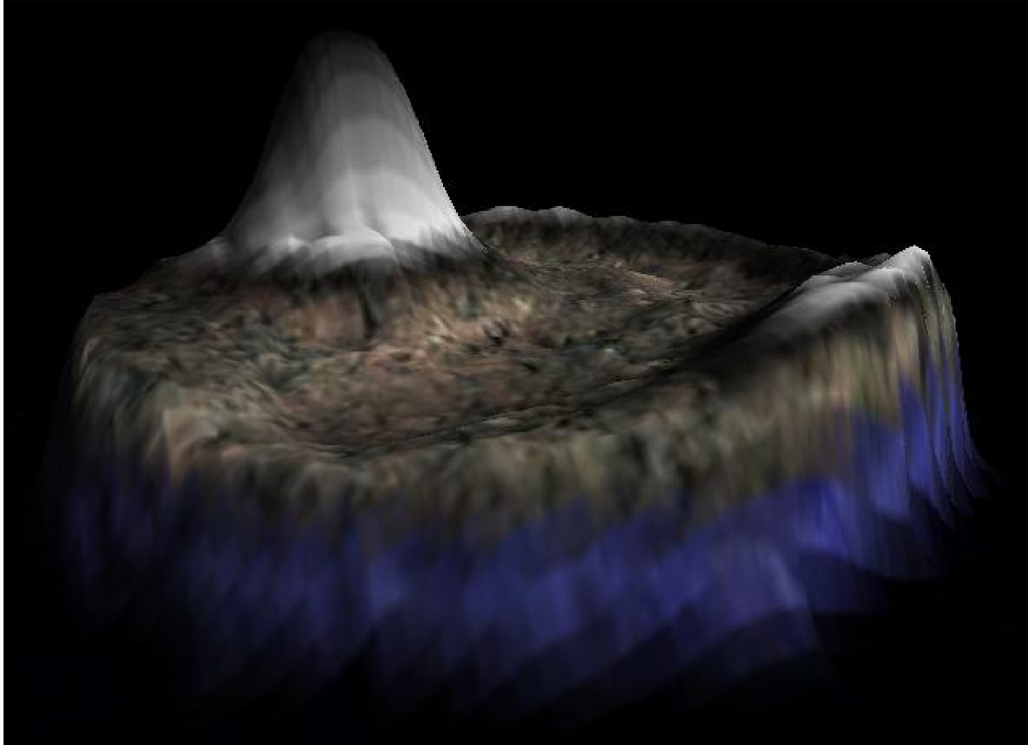
You'll notice the parts of my terrain that are the farthest up are the parts of the gray scale image that are the whitest. The class ImageBasedHeightMap creates a height map from an image where the tallest parts are white and the smallest parts are black.

```
        //  Create an image height map based on the gray scale of our image.
        ImageBasedHeightMap ib=new ImageBasedHeightMap(
                new ImageIcon(grayScale).getImage()
        );
```

We have to pass ImageBasedHeightMap an ***Image*** object of the grayscale so it can work. To do that, we create an ImageIcon from the image's URL and get an ***Image*** from that. After I have my height map and create a TerrainBlock, the next new thing is to create that strange texture I use. First, take a look at the three textures I'm mashing together.



The first is water.png, the second is dirt.jpg and the last is highest.jpg. Notice that highest is simply white. Now take another look at the complex terrain from the side.



Notice it's a blending that has the water image at the bottom, the dirt image in the middle, and the white image at the top. The next question is "How do I do that?" Well, to solve that we use ProceduralTextureGenerator. It takes a height map and some ImageIcons and produces a blend at the heights you specify for each texture.

```
//  Create an object to generate textured terrain from the
// image based height map.
ProceduralTextureGenerator pg=new ProceduralTextureGenerator(ib);
//  Look like water from height 0-60 with the strongest
// "water look" at 30
pg.addTexture(new ImageIcon(waterImage),0,30,60);
//  Look like dirt from height 40-120 with the strongest
// "dirt look" at 80
pg.addTexture(new ImageIcon(dirtImage),40,80,120);
//  Look like highest (pure white) from height 110-256
// with the strongest "white look" at 130
pg.addTexture(new ImageIcon(highest),110,130,256);
```

When I create pg, I pass it the height map (The actual height map object not an int []). It uses this height map to create the texture. Next I pass three image icons for the 3 parts. Let's look at one of those. The other two work in similar ways.

```
//  Look like water from height 0-60 with the strongest
// "water look" at 30
pg.addTexture(new ImageIcon(waterImage),0,30,60);
```

This function call tells pg to put the water image from height 0 to 60. The water image blends between 0 and 30. The blend is the strongest at 30. I do the same for the other two textures, and then signal to pg it needs to create an image icon from what it has.

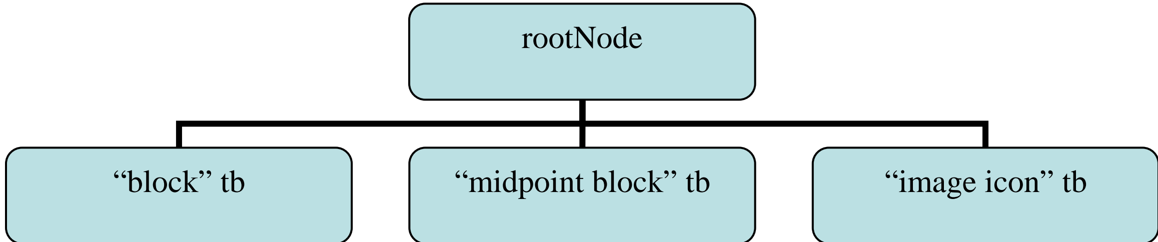
```
        //  Tell pg to create a texture from the ImageIcon's it has recieved.
        pg.createTexture(256);
```

You'll notice when I create the Texture object to use with the TextureState, I get the texture from pg.

```
        TextureState ts=display.getRenderer().createTextureState();
        // Load the texture and assign it.
        ts.setTexture(
                TextureManager.loadTexture(
                        pg.getImageIcon().getImage(),
                        Texture.MM_LINEAR_LINEAR,
                        Texture.FM_LINEAR,
                        true,
                        true
                )
        );
```

That's all there is too it. A Terrain class which is pretty important that I didn't touch on is TerrainPage. It creates a terrain and splits it into different BoundingBoxes so that the part of the Terrain you're not viewing can be culled.

Here's the obligatory scene graph:



Challenge: *Try to use TerrainPage. Using the function TerrainBlock.getHeight(), create a program that updates the camera's position to 5 above the terrain at every frame.*