

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](#).

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.

3) [Hello TriMesh](#)

This program introduces the TriMesh class and how to make your own objects from scratch.

4) [Hello States](#)

This program introduces MaterialState, TextureState, LightState, and PointLight

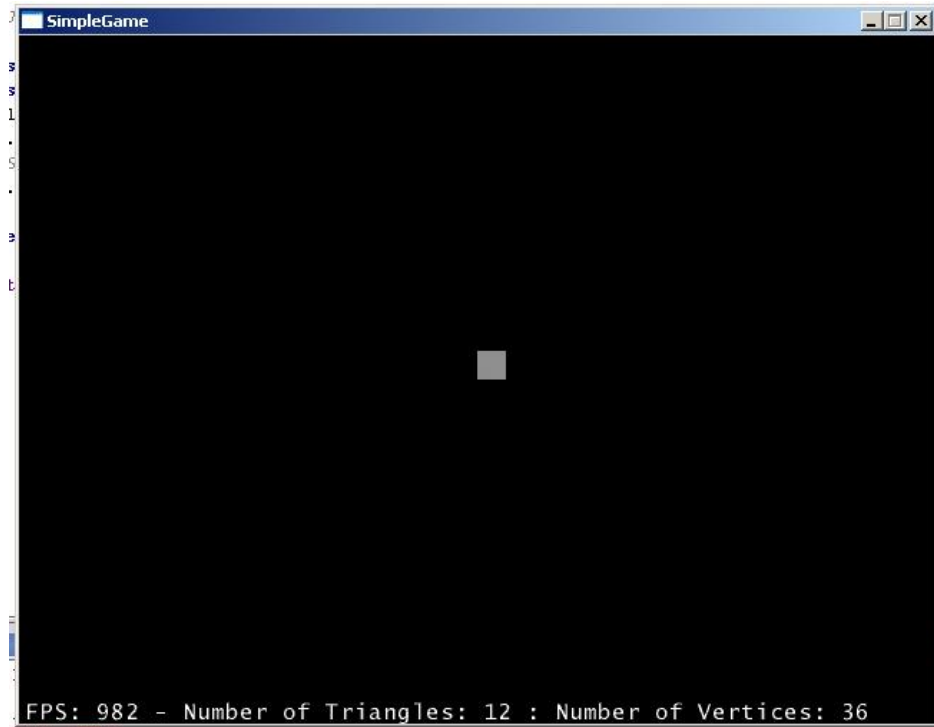
1. Hello World

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
        app.setDialogBehaviour(SimpleGame.ALWAYS_SHOW_PROPS_DIALOG);
        // Signal to show properties dialog
        app.start(); // Start the program
    }
    protected void simpleInitGame() {
        Box b=new Box("My box",new Vector3f(0,0,0),new Vector3f(1,1,1)); // Make a box
        rootNode.attachChild(b); // Put it in the scene graph
    }
}
```



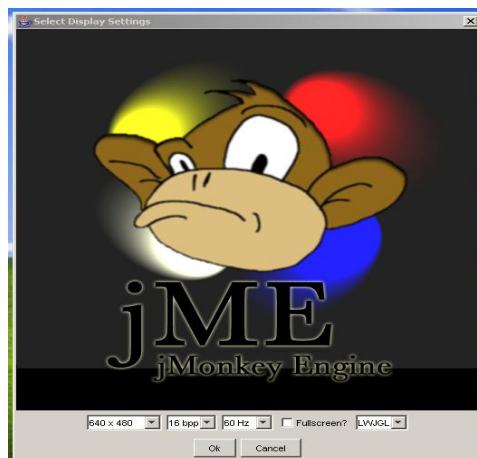
Pretty short, right? The real meat of our program begins with the following:

```
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.

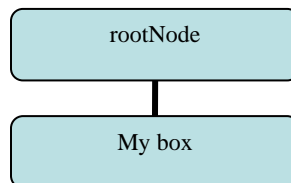
```
app.start();
```

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() {  
    Box b=new Box("My box",new Vector3f(0,0,0),new Vector3f(1,1,1));  
    rootNode.attachChild(b);  
}
```

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.

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.render.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();

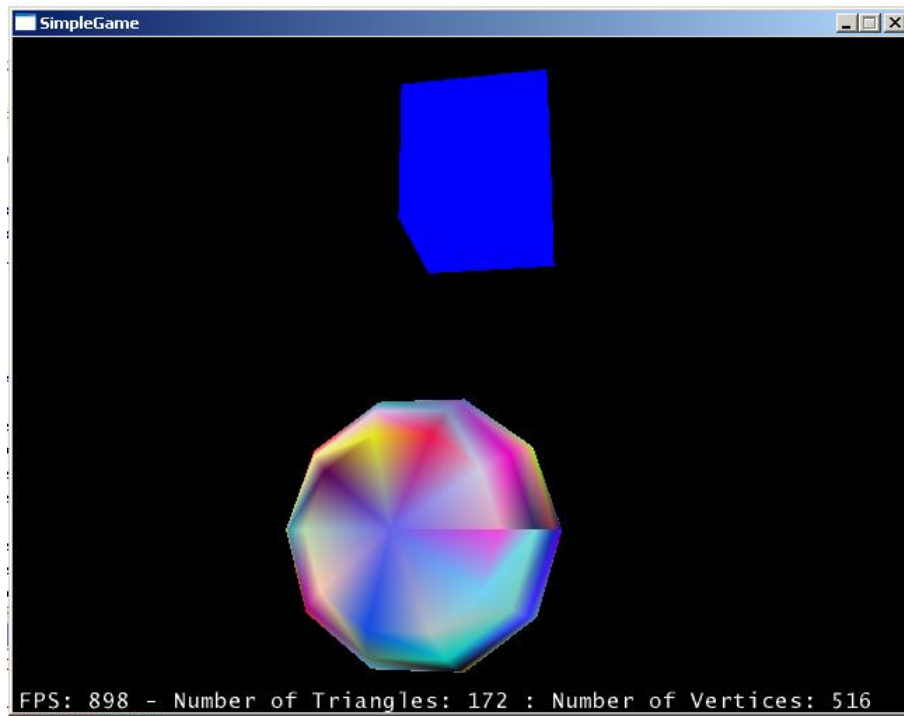
        // Make a node and give it children
        Node n=new Node("My Node");
        n.attachChild(b);
        n.attachChild(s);
        // Make the node and all its children 5 times larger.
        n.setLocalScale(5);

        // 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 psychedelic 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);
```

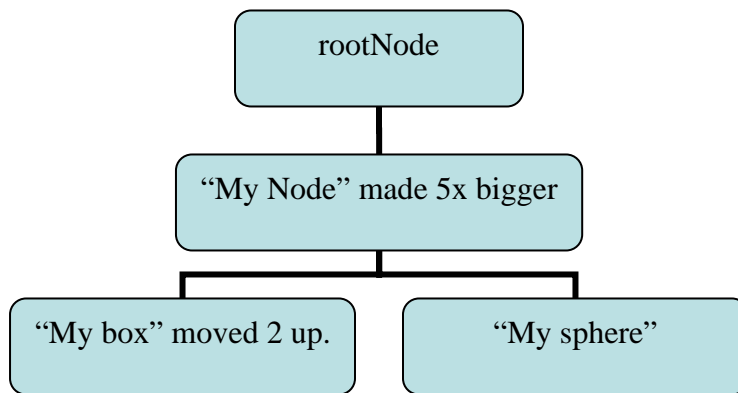
Instead of attaching our box and sphere to the `rootNode`, we create a new node `n` to attach them too. I also wanted `n` (and all its children) to be five times bigger, so I did this:

```
// 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:

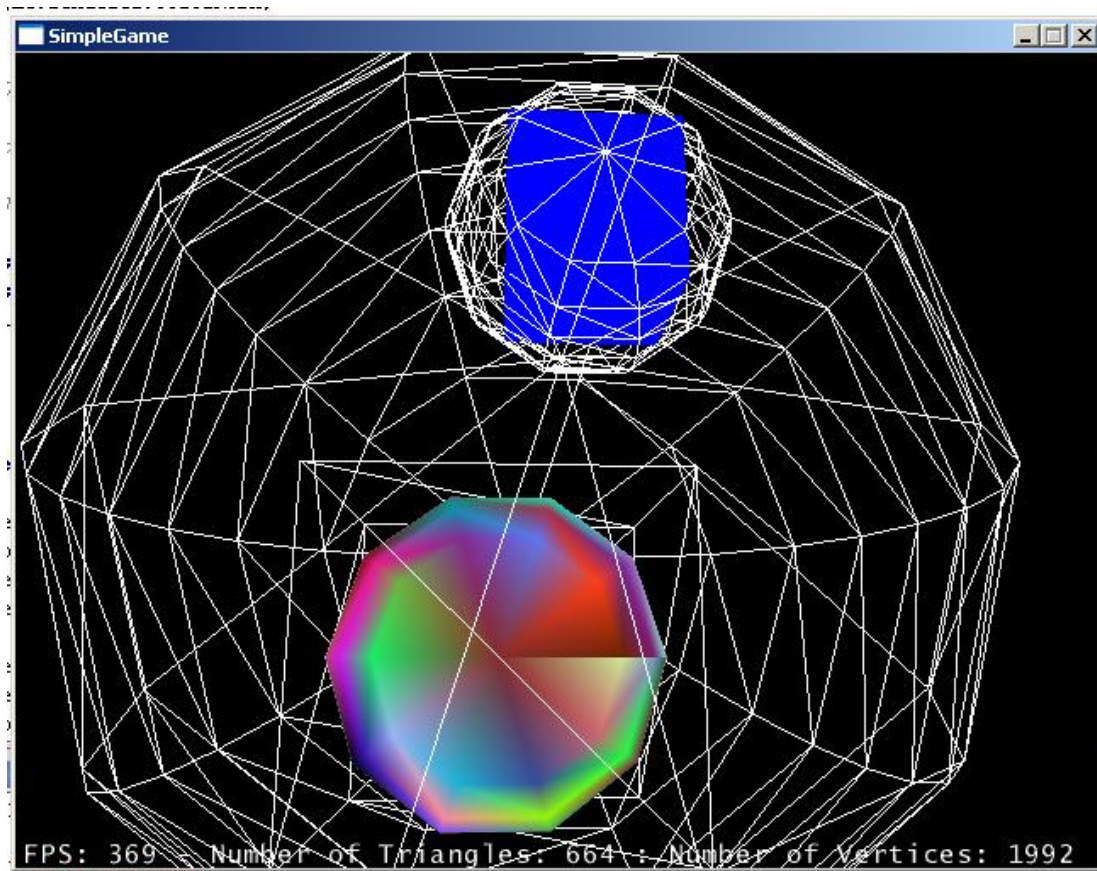
```
// Remove lighting for rootNode so that it will use our basic colors.  
lightState.detachAll();
```

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":



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)
    };

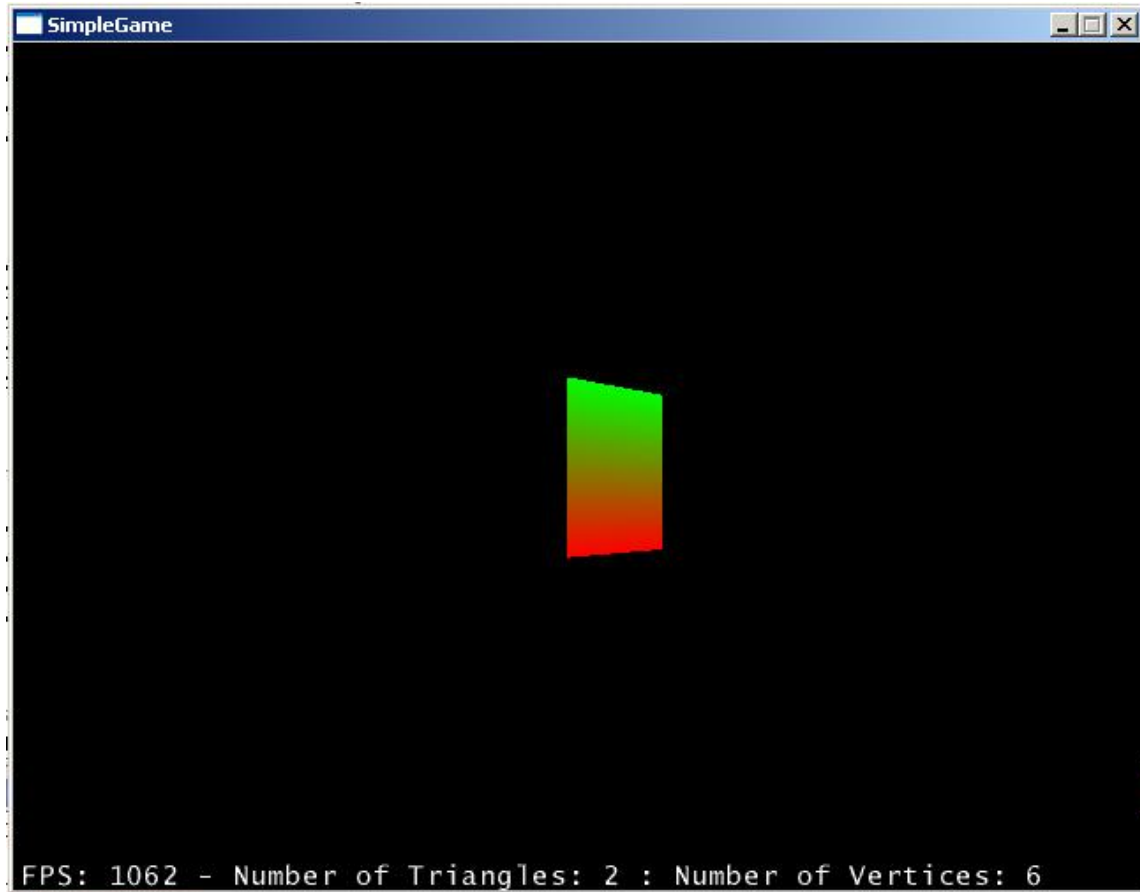
    // The indexes of Vertex/Normal/Color/TexCoord sets. Every 3 makes a triangle.
    int[] indexes={
        0,1,2,1,2,3
    };

    // 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);
}
}

```



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. 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);
```

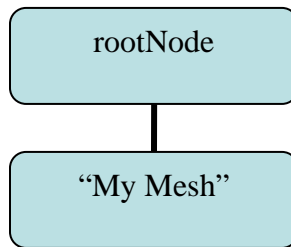
Note that because I don't use the texCoords really in this example, I could have used the following:

```
m.reconstruct(vertexes,normals,colors,null,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:



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().getTextureState();
// 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);
// Enable the TextureState
ts.setEnabled(true);

// Get a MaterialState
MaterialState ms=display.getRenderer().getMaterialState();
// Give the MaterialState an emissive tint
ms.setEmissive(new ColorRGBA(0f,.2f,0f,1));
// Enable the material state
ms.setEnabled(true);

// 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);

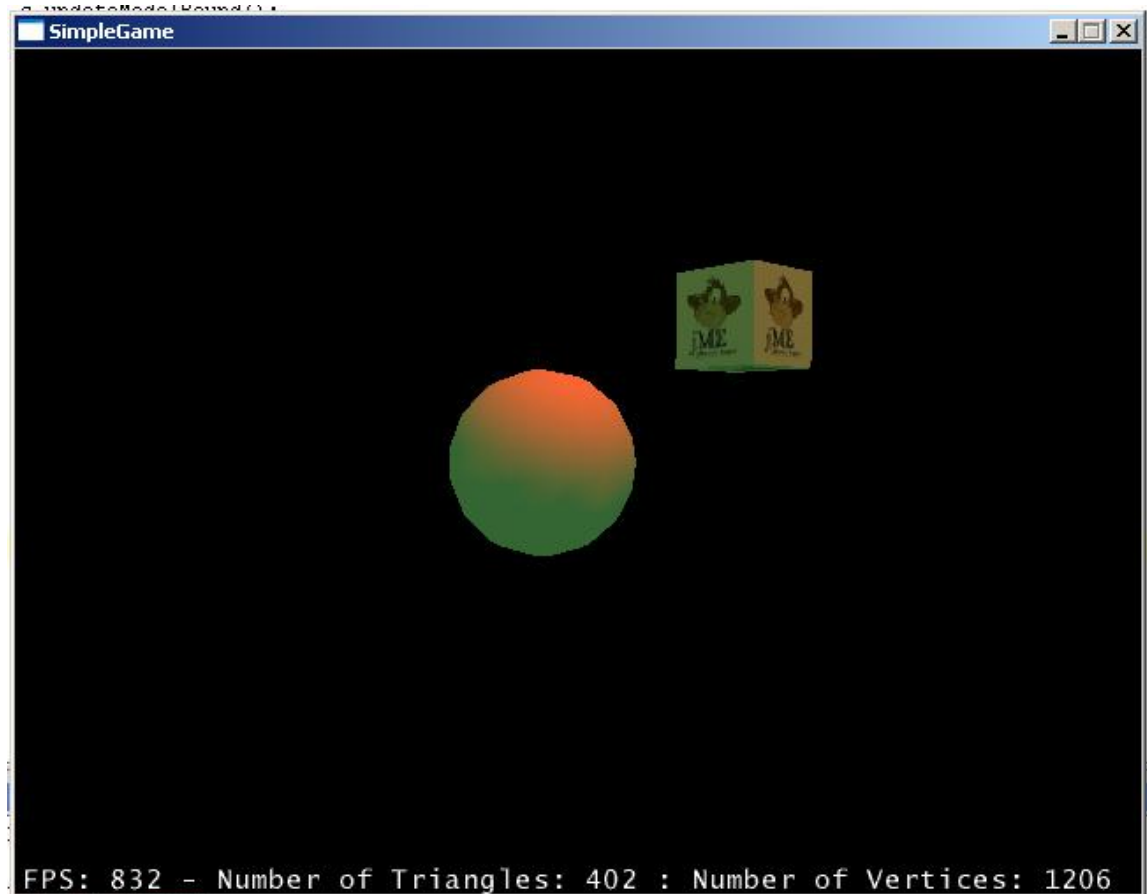
// Create a LightState to put my light in
LightState ls=display.getRenderer().getLightState();
// Attach the light
ls.attach(l);
// Enable the lightstate
ls.setEnabled(true);

// 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().getTextureState();
```

There's a new one! "display" is an object defined in SimpleGame. It "gets" whatever renderer I'm using (In my example I was using LWJGL) and from that renderer gets a texturestate. So this would give me a TextureState that works with LWJGL. This is jME's abstraction. If I was 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

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

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 things about 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 3 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);  
// Enable the TextureState  
ts.setEnabled(true);
```

Notice I have to enable the TextureState. All states (TextureState, MaterialState, CullState, ect)are false by default, so I have to enable it to make it work. 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 this uses lighting so it looks much better.

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

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. My MaterialState has to be enabled after I make it

```
// Enable the material state  
ms.setEnabled(true);
```

Again, all *State objects must be enabled first or they don't operate. 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 say? 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);
```

Just like *States, I have to enable the light

```
// Enable it
l.setEnabled(true);
```

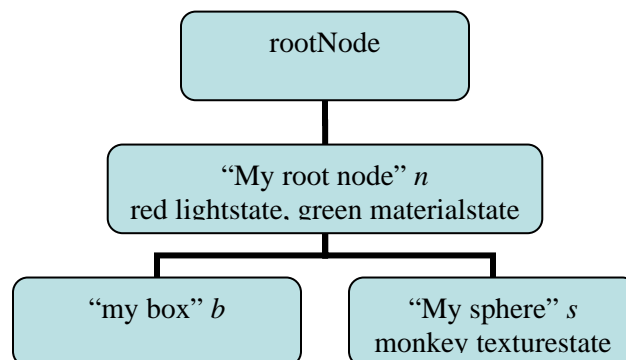
After creating the PointLight, I have to make a LightState to attach it too.

```
// Create a LightState to put my light in
LightState ls=display.getRenderer().getLightState();
// Attach the light
ls.attach(l);
// Enable the lightstate
ls.setEnabled(true);
```

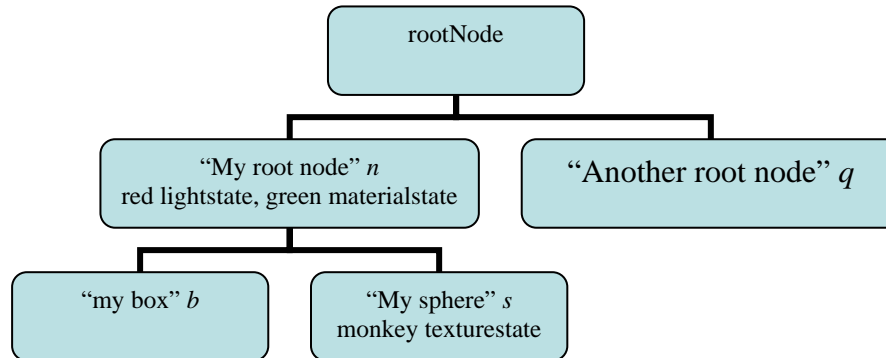
Notice I make a LightState the same way I make the other states, and I still have to enable the LightState. After I make my states, I have to assign them to the places I want them to effect

```
// 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.