

#### Get Involved

java-net Project  
Request a Project  
Project Help Wanted Ads  
Publicize your Project  
Submit Content

#### Get Informed

About java.net  
Articles  
Weblogs  
News  
Events  
Also in Java Today  
java.net Online Books  
java.net Archives

#### Get Connected

java.net Forums  
Wiki and Javapedia  
People, Partners, and Jobs  
Java User Groups  
RSS Feeds

#### Search

Web and Projects:

 »

Online Books:

 »

Advanced Search

[Home](#) | [Changes](#) | [Index](#) | [Search](#) | Go

## Wonderland Release 0.5 3D GUI Development: Button Box (Part 1)

by Deron Johnson ([deronj@dev.java.net](mailto:deronj@dev.java.net)) *Version 1.0 Fri Mar 13 11:19:21 PDT 2009*

### Purpose

This is the first tutorial in a series on developing 3D GUIs in Wonderland Release 0.5. In this series we will examine the Wonderland Release 0.5 APIs which allow you to create visible 3D objects and make them react to input events, as well as how to implement user interfaces which can be manipulated by multiple Wonderland users. In Part 1 of this series, this tutorial, we will look at how to create a simple set of buttons which reacts to mouse button clicks on a single client.

**Expected Duration: 60 minutes**

### Prerequisites

Before completing this tutorial, you should have already completed the following:

- [Developers Getting Started \(tutorial\)](#)
- [Wonderland Web-Based Administration Guide](#)
- [Project Wonderland: Working with Modules](#)
- [Extending Wonderland: Creating a New Cell Type \(Part 1\)](#)
- [Extending Wonderland: Creating a New Cell Type \(Part 2\)](#)
- [Extending Wonderland: Creating a New Cell Type \(Part 3\)](#)
- [Extending Wonderland: Creating a New Cell Type \(Part 4\)](#)

### The Button Box

In this tutorial we will create a set of 3D buttons which consists of the following objects:

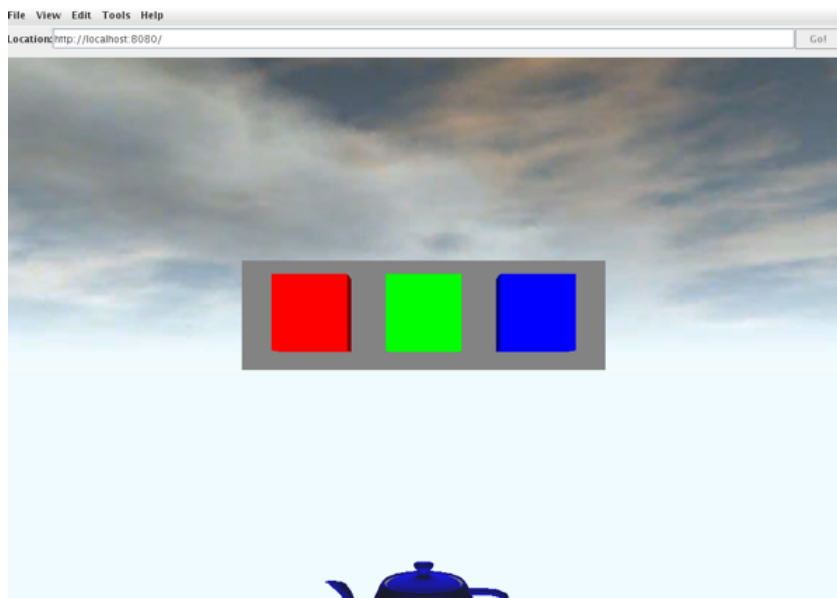
- A gray base.
- A red button.
- A green button.
- A blue button.

All objects will be input sensitive, that is to say, when you move the cursor over the object and click the left mouse button the object will react. For now, we will merely print a message to stdout when this happens. In future tutorials we will see how the button box can be used to control an animation.

In this tutorial the base and all of the buttons will be in the shape of a 3D box. The buttons are embedded in the base box and partially stick out of the front of it. For now all of the objects will be static--they won't move. We'll look at ways to add motion to the button box in future installments of this tutorial.

This tutorial is admittedly somewhat involved. The Wonderland team is aware of this, and we have future plans to simplify the process of creating 3D geometry which responds to events.

Here is a picture of the button box which you will learn to create (Figure 1): (Click on the image to see a full-sized version)



### Running the Example

In order to run the example program, create a file named **buttonboxtest1-wlc.xml** in the your current wfs directory (By default this is **~/wonderland-server/0.5-dev/wfs/worlds/celltest-wfs**) and copy the following into it and save.

```
<buttonboxtest1-cell>
  <position-component>
    <origin>
      <x>0.0</x>
      <y>3.0</y>
```

```

        <z>5.0</z>
    </origin>
    <rotation>
        <x>0.0</x>
        <y>1.0</y>
        <z>0.0</z>
        <angle>3.141569</angle>
    </rotation>
</position-component>
</buttonboxtest1-cell>

```

Then restart the Wonderland Darkstar server and run the Wonderland client.

## Example Source

The source code for this example is integrated into the Wonderland 0.5 subversion trunk. It is in the directory **modules/samples/buttonboxtest1/src**. We will be referring to this source throughout this tutorial and will include excerpts from it as necessary. (Note that this tutorial, unlike some of the other tutorials, does not include the complete source of the program in its text).

One interesting thing to note about the **ButtonBoxTest1** cell is that it extends **SimpleShapeCell**. So, in order to build properly, the buttonbox module **build.xml** contains the following lines to add the necessary jars to the various class paths:

```

<property name="module-common.classpath" value="${current.dir}/../../world/testcells
<property name="module-client.classpath" value="${current.dir}/../../world/testcells
<property name="module-server.classpath" value="${current.dir}/../../world/testcells

```

## Primary APIs Used

This tutorial concentrates on how to use the following Wonderland APIs. These APIs are provided as a part of the Wonderland bundle.

1. MonkeyEngine 2.0 (also known in Wonderland-speak as JME). (Not to be confused with Java Micro Edition).

This API provides basic rendering and scene graph management capabilities. You can learn more about jMonkeyEngine at <http://jmonkeyengine.com>. The Flag Rush tutorial series is a particularly good introduction to JME: [http://www.jmonkeyengine.com/wiki/doku.php?id=flag\\_rush\\_tutorial\\_series\\_by\\_mojomonkey](http://www.jmonkeyengine.com/wiki/doku.php?id=flag_rush_tutorial_series_by_mojomonkey).

(Note: Flag Rush and other JME programs do not run without changes in Wonderland. This is because they use a different JME "Game" classes. Wonderland requires JME code to use the MTGame game classes).

JME provides the ability to specify 3D objects in your scene: how big they are, where they are, what they look like and how they relate to each other. It allows you to do common, simple 3D graphics operations such as lighting and texture as well as advanced techniques such as particle systems.

2. MTGame.

MTGame is a library for scene graph rendering and concurrency management. It has been developed as part of the Wonderland 0.5 project but is a separate open source project in its own right (<https://mtgame.dev.java.net>).

MTGame provides the overall structure in which the JME code runs and manages the dynamic behavior of 3D objects, such as animation and other changes that happen to the scene graph over time. In particular, MTGame takes care of rendering the scene at (or as close as possible to) the configured frame rate. Using MTGame you can specify **Processor** objects which perform certain developer-provided actions at appropriate times during the rendering process. Using these you can animate various 3D object attributes such as position, color, lighting, etc.

For more information, refer to the [MT Game Programming Guide](#)

3. The Wonderland Input API.

This is a package, **org.jdesktop.wonderland.client.input**, which provides the ability to express interest in user input and other events. It is also sometimes referred to as the Event API. For more information refer to the [Wonderland Input API Specification](#).

## Cell Renderers

The concept of a Wonderland "cell" was introduced in the previous tutorials. Using a cell you can create a Wonderland module which provides unique visual objects and behavior. In this tutorial we will create a cell called **ButtonBoxTest1**.

Wonderland cells can have multiple visual and behavior representations. Different representations may be used by different Wonderland clients. For example, a cell can provide a 3D representation which is used by a 3D Wonderland client and it can also provide a 2D representation which would be used by a 2D Wonderland client. The network protocol between the server and client part of the cell could be the same in both cases, but what the user sees is different and the way the behavior is accomplished may be different. A specific cell representation is implemented in a *cell renderer*. When a cell is loaded, the client determines the render type it is using and your cell must return an cell renderer of this type.

Since at the present time the only Wonderland client is a 3D client based on JME and MTGame we will see how to create a cell renderer which uses these 3D graphics APIs. We ignore 2D renderers for now.

The Wonderland client calls the cell's **createCellRenderer** method to create the cell renderer. Here is how we implement this method for the button box. This is an excerpt from the class **ButtonBoxTest1**.

```

/**
 * Create a cell renderer which is appropriate for the client.
 */
@Override
protected CellRenderer createCellRenderer(RendererType rendererType) {
    switch (rendererType) {
        case RENDERER_2D:
            // Note: There is currently no 2D version of the button box
            return null;
        case RENDERER_JME:
            cellRenderer = new ButtonBoxTestRenderer(this);
            break;
    }

    return cellRenderer;
}

```

In the next section will examine the **ButtonBoxTestRenderer** class.

## The Button Box 3D Cell Renderer

Here is how we implement the cell renderer for the button box. It relies on a class called **ButtonBox**, which we will examine later.

```
/**
 * This renders a 3D representation of button box test cell.
 * This is a button box with three buttons: a red button, a green button,
 * and a blue button.
 */
public class ButtonBoxTestRenderer extends BasicRenderer {

    private static float BASE_HEIGHT = 1.5f;
    private static float BASE_DEPTH = 1f;

    private static int NUM_BUTTONS = 3;
    private static float BUTTON_WIDTH = 1f;
    private static float BUTTON_HEIGHT = 1f;
    private static float BUTTON_DEPTH = 0.5f;
    private static float BUTTON_SPACING = 0.5f;

    private static ColorRGBA BASE_COLOR = ColorRGBA.gray;
    private static ColorRGBA[] BUTTON_COLORS = new ColorRGBA[] {
        ColorRGBA.red, ColorRGBA.green, ColorRGBA.blue
    };

    private ButtonBox buttonBox;

    public ButtonBoxTestRenderer(Cell cell) {
        super(cell);
    }
}
```

The first thing to notice is that we create this class by extending **BasicRenderer**. **BasicRenderer** is a generic superclass for creating cell renderers. You almost always want to subclass it; it provides a lot of convenience.

Next, we define a series of constants: the height and depth of the base (note: the width will be determined by the number and dimensions of the buttons), the number and dimensions of buttons and how large a gap is between each button. We also define the colors of the base and the buttons.

## Entities

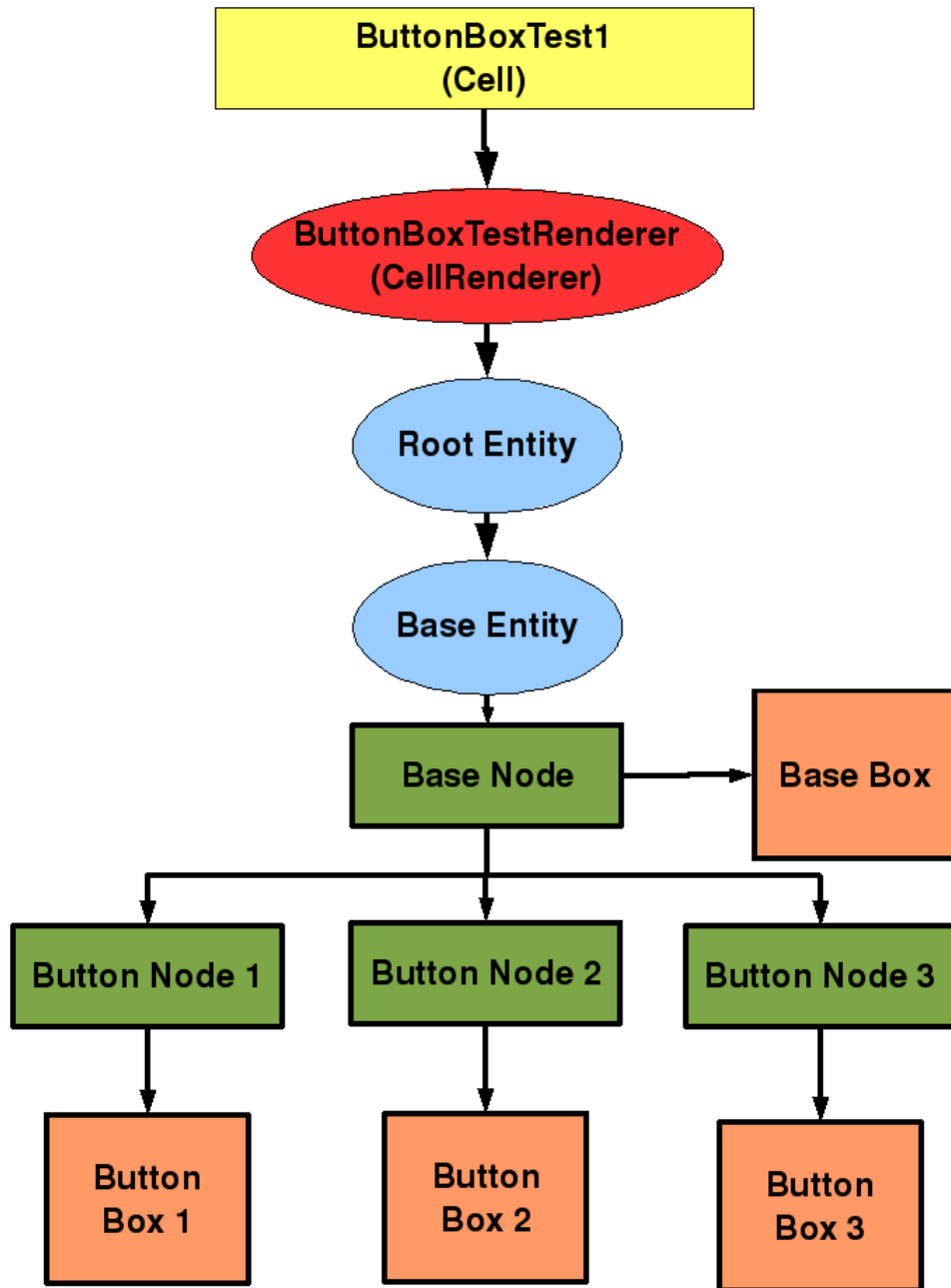
*Entities* are an important concept in Wonderland. **Entity** is a class provided by MTGame. An entity is basically a collection of visual objects and behaviors. The visual objects of an entity are specified in a *scene graph*. A scene graph is a tree of *nodes*. Each entity has a *root node* which refers to the top-most node in the scene graph.

Each entity has an extendable set of *components*. The behaviors of an entity are specified in these components. For example, an entity can potentially have an audio component, which plays an audio clip, a processor component, which animates some attribute of the scene graph, and also components which communicate with the server cell. There is also a component which allows us to attach event listeners to an entity.

When a cell is first loaded into the Wonderland client, **BasicRenderer** creates a *root entity* for the cell. In this tutorial, we will create a single entity (`baseEntity`) and will attach all the nodes of our scene graph to it. We will then attach this entity to the cell renderer's root entity so it will be displayed in the cell.

(Note: You can also build your scene graph so that there is one entity for every scene graph node. But this is more complicated and is a subject for a later tutorial).

Here is a diagram which illustrates the scene graph we will build, and how it is connected to the cell (Figure 2).



### The createSceneGraph Method

Let's now look in detail at how we build the graph depicted above. We start with the `createSceneGraph` method of `ButtonBoxTestRenderer`. This method is called when the Wonderland client needs to build the entity tree and scene graph for the cell. This is typically done when the cell is first loaded into the client. The method can also be called if the cell's resources were reclaimed because the cell wasn't visible and now the cell becomes visible again. This method is defined in `ButtonBoxTestRenderer`.

```

@Override
protected Node createSceneGraph(Entity entity) {

    // Create the root node of the test and position it appropriately.
    // We do this by moving the cell's transform into the sceneRoot node.
    Node sceneRoot = new Node("Button Box Test Scene Root Node");

    // Attach root node to to the root entity by placing it into an attached render
    RenderComponent rc =
        ClientContextJME.getWorldManager().getRenderManager().
            createRenderComponent(sceneRoot);
    rootEntity.addComponent(RenderComponent.class, rc);

    // Create the button box
  
```

```

        buttonBox = new ButtonBox(BASE_HEIGHT, BASE_DEPTH, NUM_BUTTONS, BUTTON_WIDTH,
                                   BUTTON_HEIGHT, BUTTON_DEPTH, BUTTON_SPACING);

```

The first thing we do is create a root node for the overall scene graph.

Next, we create a **RenderComponent** and attach it to the entity. Nothing can be displayed (i.e. rendered by the graphics system) without a **RenderComponent**. We place our root node inside this render component and add it to the root entity. Now the root entity knows about the root node.

Next, we create an instance of a **ButtonBox**. A lot of magic happens inside the **ButtonBox** class. The **ButtonBox** will create all of the necessary subentities, scene graph nodes, and geometry objects. We will examine this in detail in the next section.

## The ButtonBox Constructor

Let's take a look at what happens when the **ButtonBox** constructor is called.

```

/**
 * Create a new instance of ButtonBox. The width of the base depends on the width of
 * buttons, the number of buttons and the inter-button spacing.
 * @param baseHeight The height of the base.
 * @param baseDepth The depth of the base.
 * @param numButtons The number of buttons.
 * @param buttonWidth The width of the buttons. Each button has the same width.
 * @param buttonHeight The height of the buttons. Each button has the same height.
 * @param buttonDepth The depth of the buttons. Each button has the same depth.
 * Each button will stick out depth/2 in front of the base.
 * @param buttonSpacing The space between buttons.
 */
public ButtonBox(float baseHeight, float baseDepth, int numButtons, float buttonWidth,
                 float buttonHeight, float buttonDepth, float buttonSpacing) {

    if (numButtons <= 0) {
        throw new RuntimeException("Invalid number of buttons");
    }

    this.baseHeight = baseHeight;
    this.baseDepth = baseDepth;
    this.numButtons = numButtons;
    this.buttonWidth = buttonWidth;
    this.buttonHeight = buttonHeight;
    this.buttonDepth = buttonDepth;
    this.buttonSpacing = buttonSpacing;

    // Create all of the necessary objects and assemble them together
    createGeometries();
    createBaseNode();
    createCollisionComponent();
    createButtonNodes();
    baseEntity = createEntity("Base Entity", baseNode);
}

```

First, we verify that **numButtons** is a positive integer. Then we transfer the constructor arguments into the instance variables. Then we call methods which create the geometry objects, the base node, the collision component, the button nodes and, finally, the base entity. Let's take a look at each one of these in turn.

## The ButtonBox Geometries

Let's take a look at the **createGeometries** method:

```

/**
 * Create the geometry objects. Create a box for the base and a box for each button.
 */
private void createGeometries() {

    // Create the base geometry.
    //
    // Note: the JME documentation for Box(xExtent, yExtent, zExtent) is a big vague
    // on what "extent" means. Extent is half of the desired dimension.
    float baseWidth = numButtons * buttonWidth + (numButtons+1) * buttonSpacing;
    baseBox = new Box("Base Box", new Vector3f(0f, 0f, 0f),
                     baseWidth/2f, baseHeight/2f, baseDepth/2f);
    baseBox.setModelBound(new BoundingBox());
    baseBox.updateModelBound();

    // Create the button geometries and ensure their bounds are initialized
    buttonBoxes = new Box[numButtons];
    float buttonX = -baseWidth / 2f + buttonSpacing + buttonWidth / 2f;
    float buttonY = 0;
    float buttonZ = baseDepth / 2f + buttonDepth / 2f;
    for (int i = 0; i < numButtons; i++) {
        buttonBoxes[i] = new Box("Button Box " + i, new Vector3f(buttonX, buttonY, buttonZ),
                                buttonWidth/2f, buttonHeight/2f, buttonDepth/2f);
        buttonBoxes[i].setModelBound(new BoundingBox());
        buttonBoxes[i].updateModelBound();
        buttonX += buttonWidth + buttonSpacing;
    }
}

```

The first thing we do in this method is to calculate how wide the base needs to be. It needs to be wide enough to fit all of the buttons, with space in between and on the ends. Next, we create a 3D box for the base using the JME **Box** class, which is itself a subclass of the JME **Geometry** class. We tell

it to center this box at the origin (0,0,0) in the box's local coordinate system. (The base will ultimately be positioned at the center of the cell).

After you have created a geometry object you must attach a JME **Bounds** object to it and size the bounds to completely enclose the geometry. In our example we will use a subclass of **Bounds** called **BoundingBox**. We use **baseBox.setModelBound** to connect an instance of **BoundingBox** to the geometry. We then call **baseBox.updateModelBound** to size this bounding box so that it encloses the geometry of the box.

Once the base has been created, we then create the buttons. We loop over the number of buttons, placing each button centered at a different x coordinate. In this way we evenly space the buttons across the base. And we define **buttonZ** so that half of each button will be sticking out of the front face of the base box. And, as we did for the base box, we define and update a bound for each button.

At this point all of the necessary geometry objects are defined.

## Creating the Base Node

Creating the base node is very simple. We just create a new node and attach the base box geometry to it. Then we set its color using **setBoxColor**. (We will look at this method later).

```
/**
 * Create the base scene graph node.
 */
private void createBaseNode() {
    baseNode = new Node("Base Node");
    baseNode.attachChild(baseBox);
    setBoxColor(baseNode, baseBox, baseColor);
}
```

## The Collision Component

Next, we need to create an MTGame collision component for the base node. This object is used by the MTGame picking system, which is used by the Wonderland input system. To make a scene graph node input sensitive you need to make it pickable. And to make it pickable you need to associate it with a collision component. So we call **createCollisionComponent** to create the component. We will see how the collision component is used in the next sections.

```
/**
 * Creates the collision component for the topmost scene graph node.
 */
private void createCollisionComponent () {
    collisionSystem = (JMECollisionSystem) ClientContextJME.getWorldManager().
        getCollisionManager().loadCollisionSystem(JMECollisionSystem.class);
    cc = collisionSystem.createCollisionComponent(baseNode);
}
```

## Creating the Button Nodes

The next method called by the **ButtonBox** constructor is **createButtonNodes**.

```
/**
 * Create the button nodes and attach them to the base node.
 */
private void createButtonNodes() {

    // Create a scene graph node for each button. And, because these nodes are not
    // directly connected to the entity, we need to explicitly make them pickable.
    // by adding them to the collision component of the topmost node.
    buttonNodes = new Node[numButtons];
    for (int i = 0; i < numButtons; i++) {
        buttonNodes[i] = new Node("Button Node " + i);
        collisionSystem.addReportingNode(buttonNodes[i], cc);
        buttonNodes[i].attachChild(buttonBoxes[i]);
        baseNode.attachChild(buttonNodes[i]);
        setBoxColor(buttonNodes[i], buttonBoxes[i], baseColor);
    }
}
```

For each button we create a **Node**. We then attach the node to the collision component (**cc**). This makes the node input sensitive. The box geometry for the button is then attached to the button node and the button node is attached to the base node of the scene graph. Finally the color of the box is initialized to the same color as the base box.

(Aside: This collision component used in the process of 3D *picking* which the Wonderland input system performs on mouse events. For each mouse event received, the Wonderland client figures out what object is "underneath" the mouse event by doing a 3D pick. This is done by "firing" a ray into the scene and determining which Node first intersects the ray. Nodes not associated with a collision component are ignored during this picking process).

## The Base Entity

The final step performed by the **ButtonBox** constructor is to create the entity for the button box. This called **baseEntity** and is created by the following line:

```
baseEntity = createEntity("Base Entity", baseNode);
```

Here is the **createEntity** method:

```
/**
 * Create a pickable, renderable entity which renders the given scene graph.
 * Such an entity has both a render component and a collision component.
 * @param node The top node of the entity's scene graph.
 * @return The new entity.
 */
private Entity createEntity(String name, Node node) {

    // Create the entity
    Entity entity = new Entity(name);

    // Make the entity renderable by attaching a render component which refers
    // to the given scene graph.
```

```

        RenderComponent rc = ClientContextJME.getWorldManager().getRenderManager().
            createRenderComponent(node);
        entity.addComponent(RenderComponent.class, rc);

        // Make the entity pickable by attaching a collision component.
        entity.addComponent(CollisionComponent.class, cc);

        return entity;
    }

```

First, we create the entity. Then we create the render component for it that has the base node attached. To create the render component, we must call the **createRenderComponent** method of the MTGame **RenderManager**. We use the static accessor expression **ClientContextJME.getWorldManager().getRenderManager()** to get the render manager and we pass the entity's node as an argument to the create method. We then add the render component to the entity. The entity's node is now attached to the entity.

Finally, we take the collision component previously created and add it to the entity. This makes the base node of the entity able to receive events.

### Connecting the Button Box to the Cell

This completes our trip through the button box constructor. We now return to **ButtonBoxTestRenderer.createSceneGraph**, after the **ButtonBox** constructor returns. The next thing the code does is to assign colors to the base and the buttons. But let's skip this for a moment and look at the final step of connecting the buttonbox to the renderer. We will examine setting colors in the next section.

The final step is to attach the button box entity to the root entity of the cell renderer, with this line:

```

// Attach button box to the top-most entity
buttonBox.attachToEntity(entity);

```

Here is the method **ButtonBox.attachToEntity** method:

```

/**
 * Attach this button box as a subentity of the given parent entity.
 * @param parentEntity The parent entity
 */
public void attachToEntity(Entity parentEntity) {
    BasicRenderer.entityAddChild(parentEntity, baseEntity);
}

```

At this point we have completely initialized the button box, the cell renderer and the cell to the desired state (depicted in Figure 2).

### Button Box Colors

In the previous section we glossed over the setting of the base and button colors. We will now examine this in detail. Here is the cell renderer code which assigns these colors.

```

buttonBox.setBaseColor(BASE_COLOR);
for (int i=0; i<NUM_BUTTONS; i++) {
    buttonBox.setButtonColor(i, BUTTON_COLORS[i]);
}

```

Here are the button box **setBaseColor** and **setButtonColor** methods:

```

/**
 * Set the color of the base.
 * @param color The new color of the base.
 */
public void setBaseColor(ColorRGBA color) {
    baseColor = color;
    setBoxColor(baseBox, color);
}

/**
 * Set the color of the specified button.
 * @param buttonIndex The button whose color is to be changed.
 * @param color The new button color.
 * @throws java.lang.IllegalArgumentException if buttonIndex is too large or too small
 */
public void setButtonColor(int buttonIndex, ColorRGBA color) throws IllegalArgumentException {
    if (buttonIndex < 0 || buttonIndex >= numButtons) {
        throw new IllegalArgumentException("Invalid button index");
    }
    if (buttonColors == null) {
        buttonColors = new ColorRGBA[numButtons];
    }
    buttonColors[buttonIndex] = color;
    setBoxColor(buttonBoxes[buttonIndex], color);
}

```

These methods both invoke the private method **setBoxColor** on the appropriate box.

Here is the **setBoxColor** method:

```

/**
 * Set the color of the given box.
 */
private static void setBoxColor(final Node node, final Box box, final ColorRGBA color) {
    ClientContextJME.getWorldManager().addRenderUpdater(new RenderUpdater() {
        public void update(Object arg0) {
            MaterialState ms = (MaterialState) box.getRenderState(RenderState.RS_MATERIAL);
            if (ms == null) {
                ms = DisplaySystem.getDisplaySystem().getRenderer().createMaterialState(

```

```

        box.setRenderState(ms);
    }
    ms.setAmbient(new ColorRGBA(color));
    ms.setDiffuse(new ColorRGBA(color));
    if (node != null) {
        ClientContextJME.getWorldManager().addToUpdateList(node);
    }
    }, null);
}

```

This method illustrates how to change the color of a lit object in Wonderland.

But before we dive into the details of this method, here is a brief digression on the subject of lighting.

By default, lighting is enabled in Wonderland. Each cell inherits a default set of lights and these lights are used to light the cell's scene graph during rendering. The rendering process calculates the amount of each light which is reflected by each object in the scene. For each object, the amount of reflected light is summed together over all lights and this total light is used to calculate the shading of the object and the colors of the individual pixels of the object. The amount (and hue) of the reflected light of an object depends on its *material* properties. By specifying properties of the material such as diffuse reflection, specular reflection, and shininess, we can approximately simulate the reflective properties of a real surface. By default, every JME geometry object has a white, non-shiny material. So we will need to change the material to something more interesting.

This method creates a material state (which is a special type of JME *render state*) and attaches it to the box (this is only done the first time). Then both the ambient and diffuse colors of the material state are set to the desired color. The ambient property value is the amount and hue of "general light floating around the scene" that is reflected by the box and the diffuse property indicates the amount and hue of the "non-shiny" lighting in the scene that is reflected. The diffuse property tells the graphics renderer how much light intensity and hue to reflect for each light in the scene.

The final step is to notify the MTgame world manager that something has changed. (Note that we only do this if the node for the box has definitely already been created).

There is a final bit of complexity that should be mentioned. This entire method takes place within an MTGame **RenderUpdater**. Whenever you modify an attribute of any JME object when it is (or may be) part of a displayed scene graph you must perform this modification inside a render updater. The render updater **update** method is a callback into your code which is called at a safe point in the MTGame render loop. This is a time when it is safe to update JME objects.

## Listening for Events

Modules can register interest in events that occur when the mouse cursor is "over" (that is, visibly on top of) one of their visible objects. This is done by using the Wonderland Input API to attach *event listeners* to these objects. This section discusses how to attach event listeners.

Event listeners are attached to entities. These listeners respond to events which occur over the root node of the entity and, if desired, child nodes also. Multiple event listeners can be attached to an entity in order to listen for different types of events.

Back in **ButtonBoxTest1**, the cell **setStatus** method tells us when the cell becomes visible (**status == ACTIVE**). When this happens we attach an event listener to the cell renderer.

```

/**
 * This is called when the status of the cell changes.
 */
@Override
public boolean setStatus(CellStatus status) {
    boolean ret = super.setStatus(status);

    switch (status) {

        // The cell is now visible
        case ACTIVE:
            // Make the button box mouse input sensitive when it becomes visible
            cellRenderer.addEventListener(new MyMouseListener());
            break;

        // The cell is no longer visible
        case DISK:
            // The button box no longer needs to be input sensitive because it is
            // no longer visible
            cellRenderer.removeEventListener();
            break;

    }

    return ret;
}

```

Note that we also detach the event listeners when the cell is no longer visible and has been removed from memory, in order to keep things clean.

In the cell renderer, the **addEventListener** and **removeEventListener** methods simply forward the request to the button box itself.

```

/**
 * Attach the event listener to the button box. This will allow the components of the
 * to be mouse input sensitive when they are visible, that is, the listener will receive
 * mouse events.
 * @param listener The listener for mouse events on the box.
 */
public void addEventListener (EventListener listener) {
    buttonBox.addEventListener(listener);
}

/**
 * Detaches the mouse event listener from the button box. The button box will no longer
 * be input sensitive.
 */
public void removeEventListener () {
    buttonBox.removeEventListener();
}

```



The **addEventListener** method uses the event listener **addToEntity** method to attach the listener to the button box's base entity. Likewise, the **removeEventListener** method uses the event listener's **removeFromEntity** method to detach the listener from the button box's base entity. Refer to the following code (from **ButtonBox**).

```
/**
 * Attach the event listener to the button box. This will allow the components of the
 * to be mouse input sensitive when they are visible, that is, the listener will receive
 * mouse events.
 * @param listener The listener for mouse events on the box.
 */
public void addEventListener (EventListener listener) {
    this.listener = listener;
    listener.addToEntity(baseEntity);
}

/**
 * Detaches the mouse event listener from the button box. The button box will no longer
 * be input sensitive.
 */
public void removeEventListener () {
    if (listener != null) {
        listener.removeFromEntity(baseEntity);
        listener = null;
    }
}
```

By default, an event listener added to an entity in this way is called for events which occur when the mouse is over the root node of the entity, in this case the base node. However, because we earlier explicitly added the button nodes as *reporting nodes* to the base entity's collision component, the event listener will be called for events which occur over these nodes as well.

Now let's take a look at the event listener **MyMouseListener** (which is defined in **ButtonBoxTest1**) and see how this event listener gets called when an event occurs.

```
/**
 * A mouse event listener. This receives mouse input events from the button box.
 */
private class MyMouseListener extends EventClassListener {

    /**
     * This returns the classes of the Wonderland input events we are interested in
     */
    public Class[] eventClassesToConsume () {
        // Only respond to mouse events
        return new Class[] { MouseButtonEvent3D.class };
    }
}
```

First of all, it is a subclass of **EventClassListener**. The Wonderland Input API provides different types of event listeners. **EventClassListener** is an event listener which only responds to certain types of events. The **eventClassesToConsume** method of this listener returns an array which contains the classes of events of interest. In this case, we are only interested in receiving **MouseButtonEvent3D** events.

Finally, we must specify methods to call when an event is received. The Wonderland Input API provides two such methods: **computeEvent** and **commitEvent**.

The input system will first call the **computeEvent** of an event listener and then it will call the **commitEvent** method. The purpose of **computeEvent** is to allow the listener to do some arbitrary computation. This computation may take a significant amount of time and so this method is executed on some thread other than the main MTGame render thread. But, because of this, **computeEvent** is not allowed to make any changes to the scene graph.

After **computeEvent** is called, the input system will call the listener's **commitEvent** method. The implementation of this method must complete in a short amount of time, because it is executed on the main render thread of MTGame. (If you take too long in this method you will adversely impact the rendering rate and the responsiveness of the Wonderland client). But because this method is executed on the render thread it is allowed to make changes to the scene graph.

In general, you should implement your event listener to do any computations which don't involve the scene graph in the **computeEvent** method and then store the results of these computations in data members of the event listener. Then, when **commitEvent** is called, you can use these values to change the scene graph.

In the current tutorial, we will simply print a message to stdout when a mouse button click event is received. Because a print can take a significant amount of time we must perform it in the **computeEvent** method. And, because we are not making any changes to the scene graph, our event listener doesn't need to implement a **commitEvent** method. Here is the code of **computeEvent**.

```
/**
 * This will be called when a mouse event occurs over one of the components of the
 * button box.
 */
public void computeEvent (Event event) {

    // Only respond to mouse button click events
    MouseButtonEvent3D buttonEvent = (MouseButtonEvent3D) event;
    if (buttonEvent.isClicked() &&
        buttonEvent.getButton() == MouseButtonEvent3D.ButtonId.BUTTON1) {

        // For now, just print name of the clicked node
        System.out.println("Left mouse button click on " +
            ((MouseEvent3D)event).getNode().getName());
    }
}
```

The first thing we do is to cast the argument event into a **MouseButtonEvent3D**. (This is safe to do because in **eventClassesToConsume** we said we

were only interested in this type of event). Next we check whether it was the first mouse button (this is usually configured to be the left button for most users). We also make sure that the button event was a click (that is, a button press followed by a button release while the mouse pointer stayed in the same position). If these conditions are true then we know that we have the event we are looking for.

Finally, we get the node over which the event occurred by calling the `getNode` method of the event. Then we print out the name of this node.

### Try It

At this point, you should run the button box test program. When you click left on the base, you should get the following message:

```
Left mouse button click on Base Node
```

When you click on the leftmost (red) button, you should get:

```
Left mouse button click on Button Node 0
```


And, when you click on the rightmost (blue) button, you should get:

```
Left mouse button click on Button Node 2
```

### Next Steps

In future tutorials, we will see how the button box can be used to control an animation. We can use the buttons to start/stop an animation, to control direction and to control speed. We will also learn how to texture map symbols onto the buttons and how to make them look nicer. And we will add animations to the buttons themselves so that they move down and pop out in a natural way. If time permits we can also take a look at how to add a clicking sound.

Topic **ButtonBoxTutorialPart1** . { [Edit](#) | [Ref-By](#) | [Printable](#) | [Diffs](#) r3 < r2 < r1 | [More](#) }

 [java.net RSS](#)



[Feedback](#) | [FAQ](#) | [Terms of Use](#)  
[Privacy](#) | [Trademarks](#) | [Site Map](#)

Your use of this web site or any of its content or software indicates your agreement to be bound by these [Terms of Participation](#).

Copyright © 1995-2006 Sun Microsystems, Inc.

**O'REILLY COLLABNET**  
Powered by Sun Microsystems, Inc.,  
O'Reilly and CollabNet