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

Project Wonderland (v0.5): Developing a HUD-Enabled Module

by Jordan Slott (jslott@dev.java.net)

Introduction

In this tutorial, you will learn how to create and display visual components on the Project Wonderland HUD (Heads-up Display). You can find the entire source code for this module in the "unstable" section of the Project Wonderland modules workspace, under the **top-map/** directory. For instructions on downloading this workspace, see Download, Build and Deploy Project Wonderland v0.5 Modules.

This tutorial is designed for Project Wonderland v0.5 User Preview 2.

Expected Duration: 60 minutes

Prerequisites

This tutorial is geared towards advanced Project Wonderland developers. Before completing this tutorial, you should have already completed the following:

- Download, Configure, Build and Run from the Wonderland v0.5 Source
- Download, Build and Deploy Project Wonderland v0.5 Modules
- Project Wonderland: Working with Modules
- Writing Client and Server Plugins

In these tutorials you learned how to download and compile the Wonderland source code, run the Wonderland server, compile an example module project, and install the module into your Wonderland server. You also learned how to develop a "plugin" for Project Wonderland.

What is a HUD?

The Project Wonderland HUD is an area encompassing the entire scene where 2D windows appear above the 3D scene (Figure 1). Typically, the HUD is used to display 'control panel' types of graphical user interfaces. Unlike the 3D scene, the appearance of the HUD (i.e. which HUD windows are visible and their position) is not shared amongst all in-world participants: each user may have different HUD windows visible from other users.

Figure 1: The Project Wonderland HUD (click on the image to view a full-sized version)

HUD windows may either be visible (e.g. the Users and Shortcut windows in Figure 1 above) or be iconified (lower-left hand corner). Also, HUD windows may have a frame decoration (e.g. Users, Shortcuts) or have no frame decoration (e.g. FPS meter). On HUD window frame decorations, two buttons exists to iconify the HUD window or close it.

The HUD Class

The HUD class (package **org.jdesktop.Wonderland.client.hud**) represents a HUD area. A single client may, in fact, contain multiple independent HUDs, although for now, you'll just interact with the "main" HUD that encompasses the entire 3D scene window. To fetch the "main" HUD, use the HUDManagerFactory class as follows:

```
HUD mainHUD = HUDManagerFactory.getHUDManager().getHUD("main");
```

A HUD consists of a collection of HUDComponent objects (discussed below) that represent each individual window on the HUD (whether they are currently visible or invisible, iconified or maximized). The following table summarizes some of the methods on the HUD class. This table is by no means an exhaustive list of all of the methods on the HUD class. Refer to the HUD API JavaDoc for further details.

Method	Description
Dimension getDisplayBounds()	Returns the dimensions of the HUD.
Iterator<HUDComponent> getComponents()	Returns an iteration of all components on the HUD.
void addComponent(HUDComponent)	Adds a component to the HUD.
void removeComponent(HUDComponent)	Removes a component from the HUD.

The HUD class has a standard set of methods to enumerate components that exist on the HUD, ask if a specific component exists on the HUD, add a new component to the HUD, and remove an existing component from the HUD. The HUD class also has convenience methods for creating different kinds of components to place on the HUD (The HUDComponent class is discussed below).

The following table summarizes some of the methods on the HUD class for creating new HUDComponents. Refer to the HUD API JavaDoc for further details.

Method	Description
HUDComponent createComponent(JComponent)	Create a new HUD component that displays a Java(TM) Swing JComponent.
HUDDialog createDialog(String, MESSAGE_TYPE, BUTTONS)	Creates an input dialog with a prompt message, message type and buttons.
HUDMessage createMessage(String)	Creates a HUDComponent for displaying a simple message.

HUDButton createButton(String)	Creates a new HUD button.
HUDComponent createImageComponent(ImageIcon)	Creates a new HUD image.

The HUDComponent Class

The HUDComponent class represents individual windows on the HUD. Project Wonderland provides a standard set of HUD components that you may use: this set currently includes HUDButton (for a simple button), HUDDialog (for a simple dialog box), and HUDMessage (for a simple message). A powerful feature of Project Wonderland and its HUD is you may build your own specialized HUDComponent using the Java(TM) Swing GUI toolkit. The **top-map/** module, described later in this tutorial, demonstrates how to build your own customized HUDComponent.

The HUDComponent interface itself contains no methods, however, it inherits methods from the HUDObject interface. Each HUDObject has a name (i.e. that appears in the frame title of the HUD window) and a size among its collection of attributes. The following table summarizes some of the methods on the HUDComponent class. This table is by no means an exhaustive list of all of the methods on the HUDComponent class. Refer to the HUDComponent API JavaDoc for further details.

Method	Description
void setName(String)	Sets the name of the HUD component.
void setWidth(int)	Sets the width of the HUD component.
void setHeight(int)	Sets the height of the HUD component.
void setLocation(Point)	Sets the position (x, y) of the HUD component on the HUD.
void setVisible(boolean)	Sets whether the HUD component is visible or not.
void setVisible(boolean, long)	Sets whether the HUD component is visible or not, after a given delay.
void setTransparency(Float)	Sets the transparency of the HUD component.
void setMinimized()	Iconifies (or minimizes) the HUD component.
void setMaximized()	De-iconifies (or maximizes) the HUD component.
boolean isMinimized()	Returns whether the HUD component is iconified or not.
void setDecoratable(boolean)	Sets whether the HUD component should be decorated with a frame header or not.
void setIcon(ImageIcon)	Sets the icon to use when the HUD component is minimized.

HUD components also provide events for various lifecycle-related state changes. The **addEventListener()** method adds a new listener for these events; the **removeEventListener()** method removes an existing listener. The event listener implements the HUDEventListener interface. The HUDEventListener is given a HUDEvent that represents the event.

The following tables lists some of the kinds of event types represented by the HUDEvent object. By no means is the table an exhaustive list. Refer to the HUDEvent.HUDEventType API JavaDoc for further details.

Event Type	Description
ADDED	A HUD component has been added.
REMOVED	A HUD component has been removed.
APPEARED	A HUD component is visible.
DISAPPEARED	A HUD component is no longer visible.
MOVED	A HUD component has moved.
RESIZED	A HUD component has resized.
MINIMIZED	A HUD component is minimized.
MAXIMIZED	A HUD component is maximized.
CHANGED_TRANSPARENCY	A HUD component transparency changed.
CHANGED_NAME	A HUD component name changed.
CLOSED	A HUD component has been closed.

Creating a Custom HUD: The Top Map Tutorial

The remainder of this tutorial will describe an example of creating a custom HUDComponent using the Java(TM) Swing GUI toolkit. This example displays the world from above looking down on your avatar. The HUD window lets you control the elevation, letting you view more of the world from above. The top map is shown on the lower right-hand corner of Figure 2 below. To make the top map visible, use the menu item found on the Windows menu.

Figure 2: The top map on the HUD (click on the image to view a full-sized version)

Feel free to download the source code (found in the **top-map/** directory of the "unstable" section of the **Wonderland-modules** workspace), compile it, and deploy it to your Project Wonderland server and make use of it.

Developing a Custom HUDComponent

There are two classes that define the custom HUDComponent for the **top-map** module: CaptureJComponent and TopMapJPanel . The CaptureJComponent extends the Java(TM) Swing JComponent class and simply draws a BufferedImage into the component area. This class takes the BufferedImage as an argument in its constructor: other parts of the module will draw the 3D scene into this BufferedImage .

```
public class CaptureJComponent extends JComponent {

    private BufferedImage image = null;

    public CaptureJComponent(BufferedImage image) {
        super();
        this.image = image;
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (image != null) {
            g.drawImage(image, 0, 0, null);
        }
    }

    public BufferedImage getBufferedImage() {
        return image;
    }
}
```

The second class, TopMapJPanel , contains the CaptureJComponent and displays a spinner control to set the elevation of the top map. This GUI was designed in the Netbeans GUI builder (which is why there is both TopMapJPanel .java and TopMapJPanel .form files) and extends the JPanel class. (Your custom HUD component may extend JComponent or any of its subclasses, e.g. JPanel).

The TopMapJPanel class exports a listener interface to changes in the elevation set in the GUI. The following excerpts of code from the TopMapJPanel class implements this listener interface:

```
    private Set<ElevationListener> listenerSet = null;

    public TopMapJPanel() {
        listenerSet = new HashSet<ElevationListener>();
        ....
    }

    public void addElevationListener(ElevationListener listener) {
        synchronized (listenerSet) {
            listenerSet.add(listener);
        }
    }

    public void removeElevationListener(ElevationListener listener) {
        synchronized (listenerSet) {
            listenerSet.remove(listener);
        }
    }

    private void fireElevationListener(float elevation) {
        synchronized (listenerSet) {
            for (ElevationListener listener : listenerSet) {
                listener.elevationChanged(elevation);
            }
        }
    }

    public interface ElevationListener {
        public void elevationChanged(float elevation);
    }
```

The CaptureJComponent is added to the TopMapJPanel in the constructor of TopMapJPanel . It is added to **topMapPanel**, which is a JPanel created in the Netbeans GUI builder. The TopMapJPanel also creates the BufferedImage object that is passed into the CaptureJComponent constructor. The follow code excerpt from TopMapJPanel illustrates this:

```
    private BufferedImage bufferedImage = null;
    private CaptureJComponent mapComponent = null;

     public TopMapJPanel() {
        ...

        // Create the BufferedImage into which we will draw the camera scene
        bufferedImage = new BufferedImage(MAP_WIDTH, MAP_HEIGHT,
                BufferedImage.TYPE_INT_RGB);

        // Create and add a CaptureJPanel to the panel
        mapComponent = new CaptureJComponent(bufferedImage);
        mapComponent.setPreferredSize(new Dimension(MAP_WIDTH, MAP_HEIGHT));
        topMapPanel.add(mapComponent);

        ...
    }
```

The Top Map Camera Entity

While the TopMapJPanel and CaptureJComponent displays the top map in the HUD, the TopMapCameraEntity is an MT-Game Entity object that is inserted into the 3D world and is responsible for rendering the 3D scene, from above the avatar at a certain elevation looking down, into a BufferedImage object. The MT-Game layer in Project Wonderland defines an Entity object that represents a graphics entity in the world. For more information about MT-Game, see the MT-Game Programming Guide, Developing a New Cell - Part 1 tutorial and the 3D Button Box tutorial.

In these tutorials you learned how to draw basic shapes using the jMonkeyEngine API and insert them into a 3D scene graph. The TopMapCameraEntity is only slightly different from these examples: rather than insert visible shapes into the 3D scene graph, you will insert a "camera" that knows how to render its scene view into an off-screen pixel buffer. That pixel buffer is then copied into the BufferedImage object you created in the TopMapJPanel class (which is then drawn into your HUD window).

The TopMayCameraEntity class definition and constructor look something like this:

```
public class TopMapCameraEntity extends Entity implements RenderUpdater {

    ....
```

```
    private CaptureJComponent captureComponent = null;
    private float elevation = 0.0f;
    private ViewCell viewCell = null;

    public TopMapCameraEntity(CaptureJComponent capture, float elevation) {
        super("Top Camera Entity");
        this.captureComponent = capture;
        this.elevation = elevation;

        viewCell = ViewManager.getViewManager().getPrimaryViewCell();
        if (viewCell == null) {
            LOGGER.warning("Unable to find primary view cell, is null.");
            return;
        }

        createTopMap();
    }

    ....
}
```

The constructor takes the CaptureJComponent you created elsewhere (just as a reminder: this class renders a BufferedImage into a Java(TM) Swing JComponent) as an argument. This constructor fetches the "primary" ViewCell from the ViewManager and stores it away. A ViewCell is a subclass of Cell and represents your avatar. There can only be one "primary" ViewCell per client, since each client represents only a single avatar. There are other ViewCells in the Cell hierarchy that represent all of the other avatars in the system, however. The TopMapCameraEntity needs the primary ViewCell so that it can determine the position of the avatar. The constructor assumes the primary ViewCell is not null (it can be null when the client is first starting up or when the client disconnects from one server and connects to another server).

Next, the TopMapCameraEntity invokes the **createTopMap()** method to create the camera in the 3D scene graph. Feel free to look over the implementation of the **createTopMap()** method, parts of it are discussed below.

The following code creates the 3D scene graph for the camera:

```
        cameraNode = new Node();
        cameraNode.setLocalTranslation(x, y, z);
        float angle = (float)Math.toRadians(90.0f);
        Quaternion rot = new Quaternion().fromAngleAxis(angle, Vector3f.UNIT_X);
        cameraNode.setLocalRotation(rot);
        CameraNode cn = new CameraNode("Top Camera", null);
        cameraNode.attachChild(cn);
```

It sets the translation of the jME Node to an (x, y, z) that was computed from the primary ViewCell . It also rotates the camera so that it points downward. Here, two nodes are used: **cameraNode** holds the transform of the camera, while **cn** holds the CameraNode object. The **cn** Node is a child of **cameraNode**. The **CameraNode** Node is a special jME Node that holds a Camera. The following code creates the Camera and attaches it to the **CameraNode**:

```
        // have created.
        CameraComponent cc = rm.createCameraComponent(
                cameraNode,      // The Node of the camera scene graph
                cn,              // The Camera
                width,           // Viewport width
                height,          // Viewport height
                90.0f,           // Field of view
                1.0f,            // Aspect ratio
                1.0f,            // Front clip
                3000.0f,         // Rear clip
                false            // Primary?
                );
```

The CameraComponent class is defined by MT-Game. In this method invocation, the **createTopMap()** method specifies the width and height of the buffer to render the camera's 3D scene into, and various parameters such as the aspect ratio, the front and rear clipping planes, and the field of view.

Finally, it associates a special TextureRenderBuffer with the CameraComponent , and adds the CameraComponent to the TopMapCameraEntity (Entities can have a collection of Components; they are not to be confused with Cell Components which although represent a similar concept to Entity components, are entirely different). The TextureRenderBuffer is the pixel buffer into which the 3D scene is drawn; it is also defined by MT-Game.

```
        textureBuffer = (TextureRenderBuffer) rm.createRenderBuffer(
                RenderBuffer.Target.TEXTURE_2D, width, height);
        textureBuffer.setIncludeOrtho(false);

        ....
```

```
        textureBuffer.setEnable(false);
        textureBuffer.setCameraComponent(cc);
        rm.addRenderBuffer(textureBuffer);
        textureBuffer.setRenderUpdater(this);

        ....

        addComponent(CameraComponent.class, cc);
```

The act of rendering the 3D scene the camera sees is handled automatically by MT-Game. During each frame, the 3D scene of the camera, according to the parameters defined for it (e.g. width, height, aspect ratio, etc) are rendered into the TextureRenderBuffer object. And during each frame you'll want to copy the pixels from the TextureRenderBuffer into the BufferedImage object you have associated with CaptureJComponent . The call to the **TextureRenderBuffer.setRenderUpdater()** results in the **RenderUpdater.update()** method being called each frame.

You can see the implementation of the **update()** method, that copies the pixels from TextureRenderBuffer to BufferedImage , below:

```
    public void update(Object arg0) {
        BufferedImage bi = captureComponent.getBufferedImage();
        ByteBuffer bb = textureBuffer.getTextureData();
        fill(bi, bb, bi.getWidth(), bi.getHeight());
        captureComponent.repaint();
    }
```

One final note about the implementation of the TopMapCameraEntity . The position of the camera must track the position of the avatar. The TopMapCameraEntity listens for changes in the transform of the primary ViewCell and updates the **cameraNode** transform. Note that it only tracks the position, but not the rotation (so if you turn your avatar around, the map does not also rotate), but feel free to enhance the TopMapCameraEntity to support this if you so desire. Note that we must make any changes to the **cameraNode** in the special MT-Game Render thread, because the jME library is not thread-safe. The SceneWorker utility class lets you execute code on the MT-Game Render thread.

```
        listener = new TransformChangeListener() {
            public void transformChanged(Cell cell, ChangeSource source) {
                CellTransform transform = cell.getWorldTransform();
                final Vector3f translation = transform.getTranslation(null);
                SceneWorker.addWorker(new WorkCommit() {
                    public void commit() {
                        float x = translation.getX();
                        float y = elevation;
                        float z = translation.getZ();
                        cameraNode.setLocalTranslation(x, y, z);
                        wm.addToUpdateList(cameraNode);
                    }
                });
            }
        };
        viewCell.addTransformChangeListener(listener);
```

The Top Map Client Plugin

Finally, everything must be tied together and initialized upon client start-up. This is achieved by the TopMapClientPlugin class, that achieves the following:

- Create and add/remove a main menu item to display/hide the top map HUD window
- Create the top map HUD component and the TopMapCameraEntity and associate the two
- Add the TopMapCameraEntity to the world when the top map HUD window is visible
- Listen for changes in the elevation set by the user in the HUD window and change the elevation of the camera

If you are unfamiliar with client plugins, please read Writing a Client or Server "Plugin". Like in that tutorial, the **initialize()** method of the TopMapClientPlugin creates the main menu item:

```
    @Override
    public void initialize(ServerSessionManager loginInfo) {
        topMapMI = new JCheckBoxMenuItem(BUNDLE.getString("Top_Map"));
        topMapMI.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (topMapMI.isSelected() == true) {
                    if (hudComponent == null) {
                        hudComponent = createHUDComponent();
                    }
                    hudComponent.setVisible(true);
                    topMapEntity.setCameraEnabled(true);
                }
```

```
                else {
                    hudComponent.setVisible(false);
                    topMapEntity.setCameraEnabled(false);
                }
            }
        });

        elevationListener = new MapElevationListener();

        viewManagerListener = new MapViewManagerListener();
        ViewManager.getViewManager().addViewManagerListener(viewManagerListener);

        super.initialize(loginInfo);
    }
```

You'll notice (unlike the Client Plugin tutorial), that TopMapClientPlugin does not define its own **activate()** and **deactivate()** methods to add and remove the main menu item. Rather it waits until there is a primary ViewCell before it adds the main menu item. This guarantees that whenever the user selects the top map menu item, that a primary ViewCell exists and it can safely create a TopMapCameraEntity (recall: the TopMapCameraEntity constructor assumed that the primary ViewCell was non-null). The **ViewManager.addViewManagerListener()** provides events when the primary ViewCell changes.

The MapViewManagerListener receives notification when the primary ViewCell has changed, and removes any existing HUD component and menu item for the "old" primary ViewCell , and adds the main menu item if there is a "new" primary ViewCell :

```
    private class MapViewManagerListener implements ViewManagerListener {
        public void primaryViewCellChanged(ViewCell oldCell, ViewCell newCell) {
            if (oldCell != null) {
                if (hudComponent != null) {
                    HUD mainHUD = HUDManagerFactory.getHUDManager().getHUD("main");
                    mainHUD.removeComponent(hudComponent);
                }

                if (topMapEntity != null) {
                    topMapEntity.setCameraEnabled(false);
                    topMapEntity.dispose();
                    topMapEntity = null;
                }

                JmeClientMain.getFrame().removeFromWindowMenu(topMapMI);
            }

            if (newCell != null) {
                JmeClientMain.getFrame().addToWindowMenu(topMapMI, -1);
            }
        }
    }
```

The **actionPerformed()** method for the menu item (in the **initialize()** method above), creates the HUD component if it does not already exist and either displays or hides it depending upon whether the top map checkbox menu item is selected or not. It also enables and disables the camera accordingly: when the camera is disabled, it does not render the scene to the TextureRenderBuffer object, thereby improving performance when the top map is not visible.

To create the HUD component (in the **createHUDComponent()** method), the following fetches the "main" HUD and creates a new HUD component using the TopMapJPanel as the HUD JComponent. It fetches the name of the HUD window (that appears in its frame) from a ResourceBundle (a file that contains a set of keys and values), rather than hard-coding the String title in the code: this is a standard technique to internationalize (I18N) graphical user interfaces so that language-specific Strings may be easily defined for the GUI. The initial location of the HUD window is in the Southeast corner of the 3D scene window (i.e. the lower-right). There are nine possible initial locations of HUD windows (in order from the top-left to lower-right): NORTHWEST, NORTH, NORTHEAST, WEST, CENTER, EAST, SOUTHWEST, SOUTH, and SOUTHEAST.

```
        HUD mainHUD = HUDManagerFactory.getHUDManager().getHUD("main");
        TopMapJPanel panel = new TopMapJPanel();
        hudComponent = mainHUD.createComponent(panel);
        hudComponent.setName(BUNDLE.getString("Top_Map_Title"));
        hudComponent.setPreferredLocation(Layout.SOUTHEAST);
        mainHUD.addComponent(hudComponent);
```

Then, the TopMapCameraEntity is created, and given the CaptureJComponent (into which it copies the pixels from the rendered camera scene) and added to the world. The Entity is added directly to the world, so its scene graph is rooted in world coordinates (the ViewCell 's transforms are also in world coordinates).

```
        CaptureJComponent captureComponent = panel.getCaptureJComponent();
        float elevation = panel.getElevation();
        topMapEntity = new TopMapCameraEntity(captureComponent, elevation);
```

```
        WorldManager wm = ClientContextJME.getWorldManager();
        wm.addEntity(topMapEntity);
```

Conclusion

In this tutorial you learned the basic classes that comprise the Project Wonderland HUD API and some of its fundamental methods. You also learned about an example of using the HUD, that also used some advanced features of jME and MT-Game.