Wonderland Swing Developer's Guide
Deron Johnson
Version 1.2
Wed Oct 7 13:29:58 PDT 2009

Changes since 1.1
The creation of the component contained in the WindowSwing must be performed on the AWT Event Dispatch Thread.

1. Introduction

The purpose of this document is to describe how to develop Wonderland modules which use Swing GUIs. In Wonderland 0.5, module developers can create 2D rectangular windows which have a Swing **`JComponents`** mapped onto them. The user can interact with the Swing controls in these JComponents just like a normal Swing GUI. The key difference is that these windows live in the 3D world itself and so they can be seen by all nearby Wonderland users. The 3D position and orientation of these windows can be changed by a user and other users will see the result.

Swing is a mature toolkit with a lot of functionality and GUI controls. Wonderland module users can use Swing JComponents to display textual and numeric data values. Buttons and sliders can be used to allow users to control the module's behavior, and JTextFields and JTextAreas can be used for text input. Most Swing features work in Wonderland, so all of these capabilities are available to Wonderland module developers. In addition to using Swing to implement modules which only have 2D, flat windows, Wonderland developers can also use Swing to implement mixed 2D/3D modules. These are modules which use a mixture of 2D Swing components and 3D objects as well. We'll see an example of this below. (TODO: add this for 0.5 FCS).

Note: the Wonderland client core itself uses Swing to implement the Heads-Up-Display (HUD).

It is important to understand that the GUI you build in Wonderland with Swing components is only local to a single client; the components are not shared among the clients of other users in the world. For example, if a panel with a button is displayed and a user clicks the button other users do not see the button click. So Swing in Wonderland is **a local GUI only, not a shared one.** If you want other users to be notified of changes made to the GUI, and output provided via the GUI, you must arrange to share this information yourself in your module. For example, you can register a listener which is notified when a Swing button is pressed. You can then send a message to the server, as well as instances of your module running in other clients (via normal Wonderland interprocess communication mechanisms). The server and these remote module instances can take the appropriate action.

Wonderland supports almost all commonly used Swing features used by lightweight Swing components. However, there are a few esoteric features that aren't supported. Refer to Appendix 1 for a list of these. Most developers will not be impacted by these limitations. Also, it should be emphasized that you can only use lightweight components, such as **`JPanel`**. You may not use heavyweight components such as **`JFrame`** or **`JDialog`**.

(Note: if you really need to use a frame in your Wonderland swing module, you should consider using the **`JInternalFrame`** Swing component, which is a lightweight component).

2. Prerequisites

This document assumes that the reader is familiar with Swing programming. If you are unfamiliar with Swing programming, here is a tutorial which will help you get started.

This document also assumes that the reader is familiar with how to build a Wonderland module. Here is the first part of a four-part tutorial series which describes how to build a Wonderland module. A Wonderland module is an application program developed specifically to run in the Wonderland environment.

3. App Base Modules

The Wonderland *App Base* is a module which allows 2D applications to run inside the 3D virtual world. It provides support for both Swing applications as well as shared, conventional applications (such as X11 apps). This document focuses on Swing applications. The App Base provides a set of utilities to Swing module developers. The most important of these is a class called **`WindowSwing`**, which is a 3D object. This object takes a lightweight Swing **`JComponent`** as an attribute. The image of this **`JComponent`** is continuously rendered into a texture map which is then mapped onto the 3D object, which is a flat rectangle. In this document, we will also refer to a **`WindowSwing`** as a *swing window.* Note that these swing windows are different from the "top level" Swing windows that you might use to display a JDialog or JOptionPane as a peer of the Wonderland client main window in the native window system. In this document a swing window is a **`WindowSwing`** which can actually be displayed with a 3D orientation in the 3D world.

A developer-written module which uses **`WindowSwing`** is called an *app module*. Specifically it is a *Swing app module*. A Swing app module must provide a cell which is a subclass of **`App2DCell`**. This is located in the package **`org.jdesktop.wonderland.modules.appbase.client.cell`**. In addition, the cell must create an *app* object which is an subclass of **App2D**. Once the app object has been created, one or more **WindowSwings** can be created. These windows are associated with, or added to, the app object. When the cell becomes visible to a user, the module developer should call **`setVisible`** on one or more of the windows.

4. The swingexample Module

In this section we will examine an example Swing app module called **`swingexample`**. We will take a look at the actual code of this module. The source code for this module is located in **`wonderland-modules/0.5/stable/swingexample`**.

The server-side and server-client common code is fairly conventional, so we won't go into details about these parts here. The reader is encouraged to check on the source code for more information on these parts. As for the other, more interesting files, we will only examine the most interesting parts of these files. Again, the reader is encouraged to examine the code in full in the source tree.

Here is how SwingExampleCell.java creates the cell:

```
public class SwingExampleCell extends App2DCell {

    /**
```

```
     * Create an instance of SwingExampleCell.
     *
     * @param cellID The ID of the cell.
     * @param cellCache the cell cache which instantiated, and owns, this
cell.
     */
    public SwingExampleCell(CellID cellID, CellCache cellCache) {
        super(cellID, cellCache);
    }
```

It's pretty basic. The real interesting part begins in the setStatus method.

```
    protected void setStatus(CellStatus status, boolean increasing) {
        super.setStatus(status, increasing);

        switch (status) {

            // The cell is now visible
            case ACTIVE:
                if (increasing) {
                    SwingExampleApp stApp = new SwingExampleApp("Swing
Example", clientState.getPixelScale());
                    setApp(stApp);

                    // Tell the app to be displayed in this cell.
                    stApp.addDisplayer(this);

                    // This app has only one window, so it is always top-level
                    try {
                        window = new SwingExampleWindow(this, stApp,
clientState.getPreferredWidth(),

clientState.getPreferredHeight(), true, pixelScale);
                    } catch (InstantiationException ex) {
                        throw new RuntimeException(ex);
                    }

                    // Both the app and the user want this window to be
visible
                    window.setVisibleApp(true);
                    window.setVisibleUser(this, true);
                }
```

When the cell becomes active (from a less active state) an instance of **`SwingExampleApp`** is created and assigned a name. It is created with the *pixel scale* contained in the **`clientState`** which has been obtained from the server. The pixel scale is the size of the 2D window pixels in the virtual world. The units of pixel scale are meters / pixel. Pixel scale is a **`Vector2f`** whose x coordinate specifies the pixel width and whose y coordinate specifies the pixel height. Once the app is created it is attached to the cell via **`setApp`**.

The next thing which is done, the call to **`addDisplay`**, tells the cell that it will be used to display the windows of the app.

Next, a window is created. This window is an instance of the class **`SwingExampleWindow`** which is a subclass of **WindowSwing**. When the window is created, it is given the cell, the app object and the preferred width and height of the window in pixels. (Like the pixel scale, the preferred width and height are also obtained from the client state received from the server). The window constructor also takes the pixelScale as an argument.

In the above code example, the boolean argument **true** is passed to the **decorated** argument **SwingExampleWindow** constructor. It indicates that the window is to be *decorated*, or surrounded with a frame. A frame is a rectangular border which is wrapped around the window on all sides. The frame displays information which is helpful to the user, such as the window title and which user has permission to interactively control the window. It also provides a close button which can be used to close the window and quit the application. Not all windows have a frame (as we will see below). Windows that do not have a frame are called *undecorated*.

Once the window has been created, it is made visible. First, the program must state that the application wants the app to be visible, by calling **setVisibleApp** with an argument of true. Next, the program must state that the user wants the app to be visible by calling **setVisibleUser**. Note that these are separate controls on the window; both must be true in order for the window to be visible to the user.

The final part of **setStatus** handles the case where the cell is no longer visible to the user.

```
           // The cell is no longer visible
            case DISK:
                if (!increasing) {
                    window.setVisibleApp(false);
                    window = null;
                }
                break;
        }
    }
}
```

This is self explanatory: the window is made invisible and the window is made garbage collectable (Note: it is not explicitly not necessary to set the window to null, but it can help in certain memory leak debugging situations).

(Note: in most real apps, you would not destroy the window in this way just because the window becomes invisible. You would store the window reference and use it to make the window visible when the cell again comes into view. But this is a simplistic example).

The file **SwingExampleApp.java** is very simple:

```
public class SwingExampleApp extends App2D {

    public SwingExampleApp(String name, Vector2f pixelScale) {
        super(name, new ControlArbMulti(), pixelScale);
        controlArb.setApp(this);
    }
}
```

The only interesting thing in this class is the use of **ControlArbMulti**. Every app must have a control arbiter (or *control arb*) associated with it. The control arb is specified for the app when it is created. A control arb specifies a policy for how and when users can take control of the app's windows and interactively manipulate them. In this case, `ControlArbMulti` states that the policy that we want for our window is that before our user can interact with the window he or she must first "take control" of it. (We'll go into details on this later). When a user takes control of an app, the keyboard and mouse events are directed to the windows of the app, and not to other world functions such as moving the avatar. In fact, the user will not be able to move the avatar when he or she is controlling an app. In order to move the avatar once more, control must be released. (Again, we'll go into details on this later).

Another aspect of `ControlArbMulti` is that it permits any number of users to have control at the same time. This is useful when implementing some types of shared applications, but great care must be taken to prevent the input of one user from conflicting with that of other users.

(We could have chosen a different type of control arb for our example: `ControlArbSingle`. This control arb requires that a controlling user first take control of the app (like `ControlArbMulti`). But it allows only one user to have control at a time. If a user takes control while another user already has control, the control permission is taken from the controlling user and is assigned to the new user).

5. The swingexample Window Class

In this section we will take a closer look at how the example window is created. We will examine the `SwingExampleWindow.java` file in detail.

```
public class SwingExampleWindow extends WindowSwing {

   ...

   private SwingExampleCell cell;

   /** The panel displayed within this window. */
   private TestPanel examplePanel;

   public SwingExampleWindow (SwingExampleCell cell, App2D app, int width, int
height, boolean decorated,
                      Vector2f pixelScale)
      throws InstantiationException
   {
          super(app, width, height, decorated, pixelScale);
      this.cell = cell;

          setTitle("Swing Example");

          try {
                SwingUtilities.invokeAndWait(new Runnable() {
                      public void run () {
                            // This must be invoked on the AWT Event Dispatch
Thread
                            examplePanel = new TestPanel();
                      });
      } catch (Exception ex) {
            throw new RuntimeException(ex);
          }

          // Parent to Wonderland main window for proper focus handling
        JmeClientMain.getFrame().getCanvas3DPanel().add(examplePanel);

          setComponent(examplePanel);
   }
}
```

First, in the window constructor, a title is specified. This string will be displayed in the frame of the window.

Next, a test panel is created. This class of this test panel is `TestPanel`, which is a subclass of the Swing `JPanel` class. This panel contains a single button. It was constructed using the GUI builder of netbeans.

The panel must be created on the AWT Event Dispatch Thread because of the thread safety rules of Swing. For more on this refer to Section 10.

Next, something very important happens: the panel is added as a child of the Wonderland client main window. This is extremely important! If you do not do this, your panel will not work correctly. **All JComponents displayed within a WindowSwing must be parented to the Wonderland client main window.**

Finally, the panel is placed into the `WindowSwing` by calling its `setComponent` method. That's all there is to it. You have now seen how to create and display in the world a window which contains a Swing `JPanel`. This `JPanel` can contain any number of components, although in this simple example we chose to use just a single component, a simple button.

Here is what the window looks like in the world:

6. Building the swingexample module

To build this module, go to the top directory of the module and type:

```
ant
```

Then, to install the module into Wonderland, type:

```
ant deploy
```

(You must do this while the web server is running).

The `build.xml` file of swingexample is very similar to most other Wonderland module build files. However, it is important to observe that because the module depends on the app base, it adds the app base client and server jars to the module class path properties with the following ant code:

```
  <pathconvert property="module-common.classpath">
         <path
```

```
location="${current.dir}/../appbase/build/client/appbase-client.jar"/>
        <path
location="${current.dir}/../appbase/build/client/appbase-client-cell.jar"/>
    </pathconvert>
    <property name="module-client.classpath"
value="${module-common.classpath}"/>
    <property name="module-server.classpath"
value="${current.dir}/../appbase/build/server/appbase-server.jar"/>
```

A quick way to create your own module which uses the app base is to copy the `swingexample` source code and modify it. If you do this, make sure that you configure the module class path properties in the same way.

7. Running the swingexample module

To run this module, insert it in the world. In the Wonderland client window, select the main menu item **Insert -> Components**. This will bring up the component insertion dialog window. Select `swingexample` from the list and click left on the **Insert** button. In a second or two the `swingexample` window will appear. It will have a red border, or *frame*, indicating that you don't have control of the app. That is, you don't have permission to interact with it. Now *take control* of the app. This is done by clicking mouse right on the window frame to bring up the window menu. Then select the **Take Control** menu item. The window frame will turn green, indicating that you now have control of the app. (Another way to take control of the app is to press the Shift key and click mouse left on the frame or the window interior.

Now that you have control, you can click the mouse on the button. It will simply print the message "Button pressed" to stdout of the Wonderland client.

When you are done, you can release control of the window. This is done by clicking mouse right on the window frame to bring up the window menu. The select the **Release Control** menu item. There are other ways of releasing control. When you first took control, a `Release App Contro`l HUD button appeared. You can click on this to release control for all apps you control. You can also press the Shift key and click mouse left on the frame to release control of an individual app.

8. Controlling the Size of a WindowSwing

This section deals with how to set the size of a **WindowSwing** to the size you wish. Depending on the situation, the size can be controlled by the program, or by the user, or both.

8.1. Window Swing Sizing Modes

Swing supports two modes for the program to control the size of a `JComponent` container. One mode calculates the size of the component based on the preferred sizes of its contained child components. In this document, we will call this *preferred size* mode. The other mode allows the user to directly specify the size of the container, independent of the sizes of any of its children. We will call this *forced size* mode. Like Swing itself, the `WindowSwing` class supports both of these modes.

Initially **`WindowSwing`** defaults to preferred size mode; the system will calculate the size of the **`WindowSwing`** based on the preferred size of the component you specify in the call to **`setComponent`**. (If this component is a container, its preferred size depends on the preferred sizes of its children, the layout manager in effect and other factors). If this is the behavior you want, you don't need to do anything extra.

But if you want to force the window to have a specific size of your choice, you can use one of these two **`WindowSwing`** methods:

`setSize(int width, int height)`

`setSize(java.awt.Dimension dims)`, where **`dims`** is non-null.

If, at any time, you wish the window size to adapt itself to the size of its contained components, you can switch back to preferred size mode by calling:

`setSize(null)`

Note: you must first call **`setComponent`** before calling any of the **setSize** methods. Otherwise an exception will be thrown.

8.2. User Resizing of WindowSwings

By default, users cannot change the size of a **`WindowSwing`** themselves. If you want to allow users of your module to change the size of a **`WindowSwing`** you must invoke the following method:

`setUserResizable(true);`

This activates the *resize corner* which is in the bottom right hand corner of the frame. When the resize corner is active, the user can move the cursor over the corner and press-and-hold the left mouse button and drag the cursor to shrink or grow the window.

Note: if you allow the swing window to be resized it is your responsibility to communicate the size changes to other clients.

9. Window Types

Wonderland supports multiple types of windows: *primary* windows and *secondary* windows. A primary window is considered a main window of the application. A secondary window is one which is made to appear in support of an activity that is going on in one of the primary windows. For example, a secondary window could be a dialog window which queries the user for some additional information. You can specify the type of the window when you create it.

(Note: internally Wonderland supports a third type of window, *popup* windows. A popup window one that usually appears for a short period of time and is not decorated by a window frame. Popup menus are usually used to implement menus and tool tips. The WindowSwing API allows you to create a popup window, but this is an advanced usage and should be used with care. The average Swing module won't need to create its own popup windows).

9.1. Primary Windows

By default, when you create a new **`WindowSwing`**, it is a primary window. A primary window has a special ability: when you close it using the "X" button on the frame header quits the program. The user, however, is first given a chance confirm the quit. If the user confirms the quit, the Swing program running on the user's local client is stopped and the cell is destroyed. Note that this will stop the copy of the Swing program which is running on other connected clients.

Another thing about a primary window which is different than secondary windows is that in order to move it within the world you need to click Mouse Right on the window frame to bring up the Window Menu and select **`Edit`** to bring up the *Translate Affordance.* This consists of a set of arrows which lie along the cell local X, Y, and Z axes. You can drag these
arrows with Mouse Left to move the cell and the primary window is moved along with it.

IMPORTANT NOTE: In Wonderland 0.5 you should create only one primary window per app. The API supports the creation of multiple primaries, but there are currently bugs in this area. So just stick to one primary window.

9.2. Secondary Windows

Secondary windows are always specified with respect to a primary window. The primary window is the secondary window's parent. By default, the parent of a **`WindowSwing`** is null. The system complains about null parents, so you should always make sure you assign a parent to a secondary window. This parent can be the primary window or it can be another secondary window.

Note: you cannot have a secondary window in an app all by itself. The secondary window must have a primary window to be its parent.

You can create a secondary window in one of two ways:

1. Specify the type and parent with a special constructor when you create the WindowSwing. For example, there is a constructor which takes the type as an argument:

```
     public WindowSwing(App2D app, Type type, Window2D parent, int width, int
height, boolean decorated, Vector2f pixelScale)
```

Here is an example of how to use this constructor:

```
      WindowSwing window = new WindowSwing(app, WindowSwing.Type.SECONDARY,
primaryWindow, 200, 200, true, new Vector2f(0.1f, 0.1f))
```

Note: There are a few constructors which take the window type as an argument.

2. You can also create a primary **`WindowSwing`** using a type-less constructor and then turn it into a secondary using the **`setType`** and **`setParent`** methods. See the App Base javadoc for details on these methods.

You can control the position of a secondary window relative to its parent window using the **`WindowSwing.setPixelOffset(int x, int y)`** method. The offset specifies the distance (in pixels) from the **top left** corner of the parent to the **top left** corner of the secondary. There is also a way to specify the offset in cell local coordinates. Refer to the App Base javadoc for details.

10. Thread Safety of Swing Modules

Wonderland is a multithreaded program. Swing has special rules that must be followed when it is used in a multithreaded program. It is very important to understand that, in general, Swing objects **are not thread safe** (except where designated in the Swing javadoc). This means that you must take extra care when creating and manipulating Swing objects in Wonderland.

The basic Swing rule regarding threading is (to quote from the Swing docs):

"Almost all code that creates or interacts with Swing components must run on the AWT Event Dispatch Thread (or EDT)."

also:

"Some Swing component methods are labelled 'thread safe' in the API specification; these can be safely invoked from any thread. All other Swing component methods must be invoked from the EDT."

(For more info, on Swing threading practices, refer to the Swing concurrency tutorial).

What this means for Wonderland Swing developers is that you should never create or manipulate a Swing object from **`EventListener.commitEvent`** or **`Processor.commit`** methods (or almost never, as explained below).These methods run on the MTGame *Render Thread*. If you need to trigger from these methods some sort of code that involves Swing objects, you should always enclose the code in a call to **`SwingUtilities.invokeLater`** (or its, blocking counterpart, **`invokeAndWait`**). **`SwingUtilities`** is a standard part of Swing (as of Java 6). **`SwingUtilities.invokeLater`** executes a given runnable on the AWT event dispatch thread.

(Note: if you really need to use **`invokeAndWait`**, you need to take extra care, because it blocks waiting for the execution of its argument runnable to complete. In general, blocking on the MTGame render thread is not a good idea. It can adversely affect user interactivity).

Note: the **`jothjava`** module in the stable area of the **`wonderland-modules`** workspace provides an example of when and how to use **`invokeLater`**. For more on **`invokeLater`**, refer to the documentation of SwingUtilities in the Swing javadoc.

Another rule to follow is that you should never create a **`WindowSwing`** in code running on the MTGame Render Thread. This includes **`EventListener.commitEvent`** methods and **`Processor.commit`** methods. Instead, you must create the window from a generic thread or the AWT Event Dispatch Thread

Also, you should also use **invokeLater** when creating top-level Swing dialogs. These are Swing heavyweight windows which are displayed in the user's native desktop. **JOptionPane** is an example of this. For example, the following pattern in is a good one to follow:

```
public class MyDialog extends JFrame {
```

```
...
}

public void showMyDialog (final String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MyDialog(args).setVisible(true);
        }
    });
}
```

Also, if you have methods in **MyDialog** which can be called from some other part of the code besides a Swing event or action listener, you should also use **invokeLater** in these methods.

Appendix 1. Swing Features Not Supported in Wonderland

The following Swing features do not work in Wonderland. Their use should be avoided.

1. Swing components with custom repaint managers.

2. Swing components with custom event queues.

3. Custom popup factories.