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

Project Wonderland v0.5: Capturing mouse input (Part 2)

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

Purpose

In this tutorial, you will enhance the simple new Cell you have created in Part 1 of this tutorial series by registering a listener to receive mouse button events using the Input API new to Project Wonderland v0.5. You will modify your shape Cell to switch between displaying a sphere and cube whenever the user clicks on your Cell.

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

You can find the entire source code for this module, including code for future tutorials in the "unstable" section of the Project Wonderland modules workspace. For instructions on downloading this workspace, see Download, Build and Deploy Project Wonderland v0.5 Modules.

Expected Duration: 30 minutes

Prerequisites

Before completing this tutorial, you should have already successfully completed Part 1 of this tutorial series. You will be extending the functionality you implemented there.

The Wonderland Input API

The Wonderland v0.5 Input API provides a common API and framework so that client-side components can receive events for keyboard and mouse input. The framework is general enough so that, in the future, it will also support event delivery for other sorts of input devices, although the purpose of this tutorial is to describe keyboard and mouse input handling.

The Input API is rooted in the InputManager abstract base class (package **org.jdesktop.Wonderland.client.input**). For the 3D graphics system employed by Wonderland (jME), the concrete class is InputManager3D (package **org.jdesktop.Wonderland.client.jme.input**). To obtain the singleton instance of this class:

```
InputManager3D manager = InputManager3D.getInputManager();
```

For the purposes of this tutorial, you do not need to interact with this class directly at all, although it is used to set focus (for example, for keyboard events) and to register global listeners (listeners not tied to a particular graphical Entity). Please consult the Wonderland Input API Specification for more details.

The Input API supports event delivery based upon both 'picking' and 'focus'.

Picking

In 3D graphics, "picking" is the act of selecting an object in 3D space using a mouse. Handling mouse input is the focus of this tutorial.

Event Focus

Event delivery based upon focus mainly applies to keyboard events, but can also apply to mouse events. The concept of keyboard event focus is actually quite common: using a normal desktop grahical user interface, the application window that has been selected (and is typically on top of other application windows and highlighted somehow to indicate it has keyboard event focus) receives the keyboard events.

The concept of event focus is not limited to delivering events to a single graphical Entity in Wonderland; it is up to the listener whether to honor the current focus or not. This lets Wonderland implement both a desktop graphical user interface paradigm where only a single "application" (in our case, a single graphics Entity) has focus, and also allows for a listener to receive all keyboard events in the system.

This tutorial does not detail handling focus-related and keyboard events. For more details, please consult the Wonderland Input API Specification.

Changing shapes in ShapeCellRenderer.java

In this tutorial, your modifications will occur entirely in the **ShapeCell.java** and **ShapeCellRenderer.java** files. You should use your existing code from previous tutorials as a base, even though you'll be modifying several of the methods in there. Conceptually, the changes are straightforward: you'll register for mouse events and when the user clicks the mouse button, you will render a sphere if the current shape is a cube, and vice versa.

You'll first modify the **ShapeCellRenderer.java** class so that invoking the **updateShape()** method will update the shape drawn based upon the current shape stored in **ShapeCell.java**. The **updateShape()** will query the Cell what the current shape type is, generate a new shape mesh based upon the type, and update the scene graph Node with the new shape mesh. You can implement this method as follows:

```
    public void updateShape() {
        final String name = cell.getCellID().toString();
        final String shapeType = ((ShapeCell) cell).getShapeType();

        SceneWorker.addWorker(new WorkCommit() {
            public void commit() {
                node.detachAllChildren();
                node.attachChild(getShapeMesh(name, shapeType));
                node.setModelBound(new BoundingBox());
                node.updateModelBound();

                ClientContextJME.getWorldManager().addToUpdateList(node);
            }
        });
    }
```

Note that whenever you change any part of the scene graph, you must tell the Wonderland world manager of the node you have updated via the **addToUpdateList()** method. This tells the 3D graphics subsystem to redraw it properly. Any time you modify the jME Nodes or primitives while attached to the scene, you have to make the changes within a special thread. In the case above, you are removing an active node and adding a new one. Since the jME itself is not multi-threaded safe, the MT Game layer ensures that all changes to the jME scene graph happen in a multi-threaded safe manner. You may use the SceneWorker class to run code on the special MT Game "render" thread.

You will also need to add the following import statements:

```
import org.jdesktop.mtgame.processor.WorkProcessor.WorkCommit;
import org.jdesktop.Wonderland.client.jme.ClientContextJME;
import org.jdesktop.Wonderland.client.jme.SceneWorker;
```

Handling user input events in ShapeCell.java

In the **ShapeCell.java** class, you must now register to receive notification whenever someone has clicked a mouse button while over your shape Cell. In Wonderland v0.5, client Cell classes register input event listeners on Entity objects.

You will need to add the following import statements to your java class beneath the package statement:

```
import org.jdesktop.Wonderland.common.cell.CellStatus;
import org.jdesktop.Wonderland.client.input.EventClassListener;
import org.jdesktop.Wonderland.client.jme.input.MouseButtonEvent3D;
import org.jdesktop.Wonderland.client.jme.input.MouseEvent3D.ButtonId;
import org.jdesktop.Wonderland.client.input.Event;
```

Next, you will override another method defined in Cell: **setCellStatus()**. This method is called when the Cell's status on the client has changed. The client-side Cell status is a set of states which designates whether the Cell is loaded in memory, "near" the avatar, or currently visible. The following Cell status types exist, defined by the **CellStatus** class:

- **DISK**: Cell is on disk with no memory footprint
- **INACTIVE**: Cell geometry is in memory, but not being rendered
- **ACTIVE**: Cell is 'close' to avatar
- **RENDERING**: Cell is being rendered in the world
- **VISIBLE**: Cell is in view frustum

In your implementation of the **setCellStatus()** method you will add a listener when the Cell becomes **RENDERING** (that is, the Cell is being rendered in the world) and removes the listener when the Cell goes back into the **INACTIVE** state. The system guarantees that whenever the status of the Cell changes, it passes through all intermediate states. The **setCellStatus()** method also gives an indication of which direction the state changes are happening: if **increasing** is true, it means the state is changing towards the **VISIBLE** state; if **increasing** is false, it means the state is changing towards the **DISK** state.

First, define a member variable in your ShapeCell class to store the listener (that you will define below):

```
    private MouseEventListener listener = null;
```

Next, define your **setCellStatus()** method as follows:

```
    @Override
    public void setStatus(CellStatus status, boolean increasing) {
        super.setStatus(status, increasing);

        if (status == CellStatus.INACTIVE && increasing == false) {
            if (listener != null) {
                listener.removeFromEntity(renderer.getEntity());
                listener = null;
            }
        }
        else if (status == CellStatus.RENDERING && increasing == true) {
            if (listener == null) {
                listener = new MouseEventListener();
                listener.addToEntity(renderer.getEntity());
            }
        }
    }
```

When the Cell becomes **RENDERING**, create a new instance of MouseEventListener and add it -- note that you add listeners to Entities, not Cells themselves. You can simply add a listener to the "root" Entity that is created for you in your Cell renderer. When the Cell becomes **INACTIVE** (that is, when your avatar is really far away from it), you will remove the listener.

The MouseEventLister class is an inner class of your ShapeCell class. You can define it as follows:

```
    class MouseEventListener extends EventClassListener {
        @Override
        public Class[] eventClassesToConsume() {
            return new Class[] { MouseButtonEvent3D.class };
        }

        // Note: we don't override computeEvent because we don't do any computation in t

        @Override
        public void commitEvent(Event event) {
            MouseButtonEvent3D mbe = (MouseButtonEvent3D)event;
            if (mbe.isClicked() == false || mbe.getButton() != ButtonId.BUTTON1) {
                return;
            }
            shapeType = (shapeType.equals("BOX") == true) ? "SPHERE" : "BOX";
            renderer.updateShape();
        }
    }
```

The MouseEventListener class extends the EventClassListener class, which is an abstract base class to help you implement event listeners. It also implements the EventListener interface. It is rare that you should need to implement the EventListener interface yourself, and the abstract base classes provide much needed functionality (e.g. an implementation of the **addToEntity()** method you used above, for example).

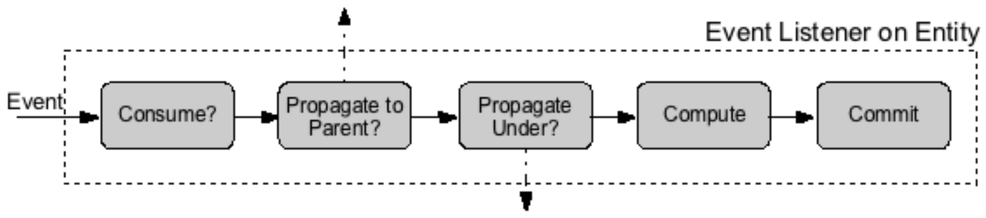
The **eventClassesToConsume()** method returns an array of Java Class objects of the events that the listener handles. In this case, the listener consumes only mouse button events (MouseButtonEvent3D.class). All of these events must extend the Wonderland *Event* class (package **org.jdesktop.Wonderland.client.input**). The following event classes are defined in the **org.jdesktop.Wonderland.client.jme.input** package:

Input Event Class	Description
MouseButtonEvent3D	Represents mouse presses, releases, and clicks, including the number of clicks
MouseMovedEvent3D	Represents all mouse motion, excluding any dragged motion
MouseDraggedEvent3D	Represents all mouse drags (mouse motion while pressing a mouse button)
MouseEnterExitEvent3D	Represents whenever the mouse enters or exits a Cell
MouseWheelEvent3D	Represents mouse wheel motion, including the wheel motion as a number of "clicks" it moved
KeyEvent3D	Represents all key presses, releases, and "types" (a press and release)

Processing of an Event in an Entity

The event processor in the Wonderland input system follows a very specific set of steps when it delivers events to your event listeners (see Figure 1).

Figure 1. Event delivery mechanism for the Input API



These steps are as follows:

- **Consume:** The event listener is asked whether it handles (or consumes) the event type. Your event class implements the **eventClassesToConsume()** and returns an array of event class types that it handles.
- **Propagate to Parent:** The event listener is asked whether the event should be propagated to parent Entities. A Entity is contained within its parent Entity (see Figure 2 below). By default, events are propagated to their parents, but event classes can override the **propagatesToParent()** method to control this behavior.
- **Propagate Under:** The event listener is asked whether the event should be propagated to Entities 'under' the Entity (or visually behind an Entity in a 3D sense). By default, events are propagated to Entities under them, but event classes can override the **propagatesToUnder()** method to control this behavior.
- **Compute:** The event listener performs any complex or time-consuming tasks in preparation to process the event, but does not actually take action on the event. Event classes can override the **computeEvent()** method to perform these tasks.
- **Commit:** The event takes action on the event, but should have performed any time-consuming tasks in the Compute stage. Event classes must override the **commitEvent()** method to take action on the event.

The ordering of method invocations in the event delivery process is not entirely as strict as presented in Figure 1 (e.g. **eventClassesToConsume()** may be called multiple times, and methods may be called out of order). Refer to the Wonderland Input API Specification for more details on the method call ordering constraints.

Figure 2. Picking mechanism for the Input API

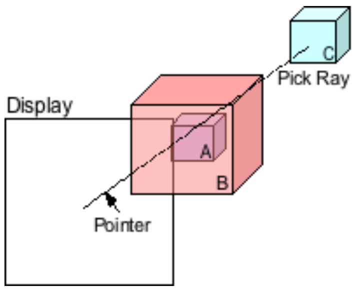


Figure 2 illustrates the concepts of "parents" of Entities and Entities that are "under" other Entities. In the figure, three Entities (A, B, C) exist, where Entity A is contained within the larger Entity B. Entity B is the parent of Entity A. Entity C exists visually behind both Entity A and B, and therefore is "under" both Entity A and B. The pick ray (dotted line) between the window camera and mouse pointer intersects all three Entities. The event it delivered to Entity B first, which then decides whether to propagate the event to Entity A (via the **propagatesToParent()** method) and whether to propagate the event to Entity C (via the **propagatesToUnder()** method).

Strictly speaking, parent Entities do not need to form a spatial relationship with its children. That is, a child Entity need not be situated entirely within its parent. The spatial ordering of parent and child Entities, as shown in Figure 2, is the most common case, however.

Running Your New Shape Cell

You can now compile your module and re-deploy it into Wonderland. When you run your client with the shape Cell and click on it, it should change shape from a cube to a sphere. Please refer to previous tutorials for help on compiling and installing your module.

- Wonderland Web-Based Administration Guide
- Project Wonderland v0.5: Working with Modules
- Project Wonderland v0.5: Developing a New Cell (Part 1)

Next Steps

In the next tutorial, you will learn how to texture your shape, enhancing its appearance. By using textures, you will learn how to interact with artwork repositories.

Part 3 - Texturing the Shape Cell Type
Part 4 - Synchronizing the State of the Shape Cell Type

Topic **ProjectWonderlandDevelopingNewCell05Part2** . { Edit | Ref-By | Printable | Diffs r1 | More }

Participation.

Revision r1 - 2009-12-25 - 02:48:12 - Main.jslott
Parents: WebHome > ProjectWonderland