

Chapter 28.9. Building a Game Pad Controller with JInput

Playing PC games with a keyboard and mouse can sometimes feel like playing tennis in a tuxedo – entirely possible but not quite right. Wouldn't it be so much cooler to pull out that game pad, joystick, power glove, or steering wheel, plug it in, and play games the way that nature intended?

It's not that difficult, as this chapter shows. I'll use the JInput API to connect a game pad to the first-person shooter from chapter 24 (FPSShooter3D). Aside from making me feel like a high-rollin' games wizard (ha, ha), the game pad allows me to extend FPSShooter3D's input processing. In this version, a player can move, rotate, and fire their gun at the same time.

I'll start by giving some background on JInput. Then I'll explain how to choose an input device, configure it in Windows, and examine it with JInput.

I'll describe how to program with the JInput API by developing three small applications that display information about the input devices attached to the PC. I'll use this information to write a GamePadController class which offers a simple interface to my game pad. GamePadController is utilized in two examples: a Swing application called GamePadViewer, and FPSShooter3D.

1. JInput

JInput (<https://jinput.dev.java.net>) is a cross-platform API for the discovery and polling of input devices, ranging from the familiar (keyboard, mouse) to more fun varieties (e.g. joysticks and game pads).

The range of supported devices depends on the underlying OS. On Windows, JInput employs DirectInput, on Linux it relies on `/dev/input/event*` device nodes, and there's an OS X version as well. Posts to the JInput forum at [javagaming.org](http://www.javagaming.org) (<http://www.javagaming.org/forums/index.php?board=27.0>) suggest that JInput runs well on Windows and Linux, but is less fully implemented on OS X. I've only tested my code on Windows, and would welcome comments by readers using other platforms.

JInput first appeared in 2003, as part of a collection of open-source technologies (JOGL, JOAL, JInput) initiated by the Game Technology Group at Sun Microsystems. JInput was developed by members of the JSR 134 expert group (a group concerned with Java gaming).

I'm using the August 2005 Windows version of JInput, which I downloaded as `jinput_windows_2005-08-29.zip` from <https://jinput.dev.java.net>. It came from the Win32 folder under the "Documents & Files" menu item.

Recent JInput releases differ from the popular version 1.1 from 2004. In particular, the old Axis class has been renamed to Component, and the component Identifier classes have been reorganized and expanded. Unfortunately, this means that older JInput examples, notably those in the excellent tutorial by Robert Schuster at

<https://freefodder.dev.java.net/tutorial/jinputTutorialOne.html>, won't work without some modifications.

If you're looking for up-to-the-minute information, then visit the JInput javagaming.org forum at <http://www.javagaming.org/forums/index.php?board=27.0>. The current JInput maintainer is an active member, and helped me considerably while I was writing this chapter.

The latest beta versions of JInput can be found at <http://www.newdawnsoftware.com/resources/jinput/>. When a version becomes 'official', it's moved to the <https://jinput.dev.java.net> site.

2. First the Game Pad and Windows

A lot of heartache can be spared by choosing your game-playing device with care. There's two points to look out for: first, the device should definitely support the OS that you're using. This usually isn't an issue with Windows, but you may need to do some research for Linux or OS X. The other good idea, based on comments in the JInput forum, is to choose a device that uses a USB connector.

I bought a basic game pad, the Genius MaxFire G-08XU, for a very reasonable US\$ 8. Top and front views of the device are shown in Figures 1 and 2.



Figure 1. Top View of the Game Pad.



Figure 2. Front View of the Game Pad.

The game pad offers a single direction pad (DPad) with eight different positions, and eight buttons (labeled '1' to '8').

When Windows detects a new input device it either installs one of its own device drivers, or requests one from you. After this process is finished, it's important to calibrate and test the device within Windows. This is done through the Game Controllers application accessible through the Control Panel (it's called Gaming Options in Windows 98).

Figure 3 shows part of the Game Controllers main window – Windows has detected the game pad.

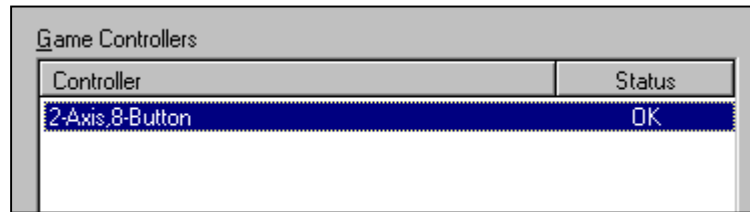


Figure 3. Part of the Game Controllers Window.

Figure 3 gives the first indication of how Windows 'sees' the game pad – as a device with two axes and eight buttons. An axis can generate more than two values (unlike a button), and game pads use axes to return information about the device's position, rotation, velocity, or even acceleration.

The game pad should be calibrated so that Windows correctly interprets DPad movements and button presses. Poor calibration will manifest itself at the JInput level as values which are incorrectly rounded or have the wrong sign.

The game pad can be tested through the Game Controllers Test window, part of which is shown in Figure 4.

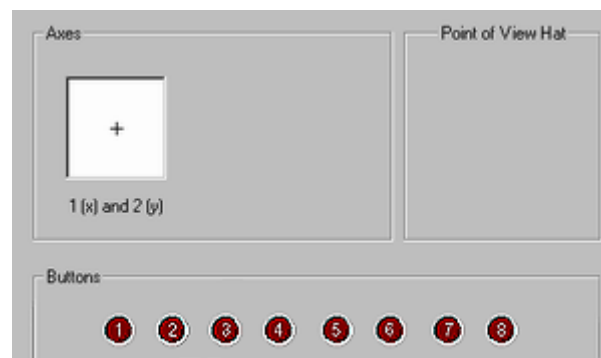


Figure 4. Testing the Game Pad.

When I press the DPad, the "+" symbol moves to the corresponding edge or corner. The two axes represent the DPad's x- and y- directions according to the text message in the "Axes" area of the Test window. Also, as I press the game pad's buttons, the corresponding numbered circles in the Test window turn bright red. This tells me that Windows can see all the buttons, and what labels they've been assigned.

After calibration and testing, I can be confident that the game pad is recognized by the OS, and I also have a good idea about the game pad's input capabilities: x- and y- axes on the DPad, and 8 buttons.

My final check is to find out whether DirectX can see the device, since JInput uses it under Windows. An easy way of doing this is with the dxdiag utility, Windows's DirectX diagnostic tool, which can be started via Window's "run" menu item. Figure 5 shows part of its Input tab, which reports what devices can be communicated with by DirectInput.

DirectInput Devices					
Device Name	Status	Controller ID	Vendor ID	Product ID	Force Feedback Driver
Mouse	Attached	n/a	n/a	n/a	n/a
Keyboard	Attached	n/a	n/a	n/a	n/a
2-Axis,8-Button	Attached	0	0x0583	0xA000	n/a

Figure 5. Part of the dxdiag Input Tab.

The game pad is listed, so both Windows and DirectX are happy with my game pad. Now it's time to try out JInput.

3. Installing and Testing JInput

The August 2005 Windows version of JInput consists of two files: jinput.jar (the high-level Java part of the API) and jinput-dxplugin.dll (the OS-level part).

It's likely that you'll want to include these files with your application when you distribute it, so the simplest installation strategy is to copy them into the application folder. Figure 6 shows a simple "JInput Application" directory containing the JInput files and a batch file, runTest.bat, that I'll explain in a moment.

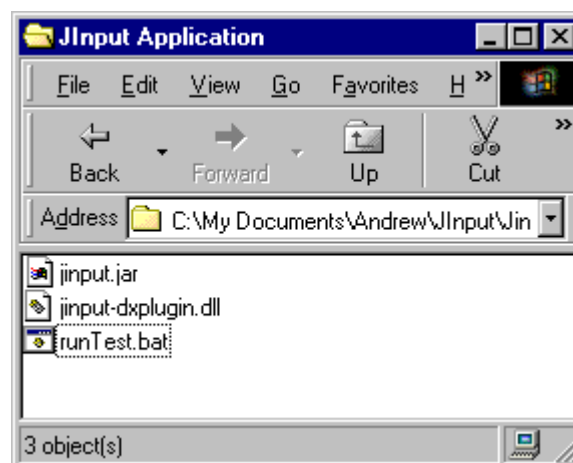


Figure 6. A Basic JInput Application Folder.

Aside from the Java parts of the JInput API, jinput.jar also contains three test applications, ControllerReadTest, ControllerTextTest, and RumbleTest. From within the "JInput Application" directory, they're called like so:

```
java -cp jinput.jar;. net.java.games.input.test.ControllerReadTest
java -cp jinput.jar;. net.java.games.input.test.ControllerTextTest
java -cp jinput.jar;. net.java.games.input.test.RumbleTest
```

Typing these long lines is rather tiring, so I've packaged most of the command line in `runTest.bat`. It contains:

```
java -cp jinput.jar;. net.java.games.input.test.%1
```

This means that the program calls become:

```
runTest ControllerReadTest
runTest ControllerTextTest
runTest RumbleTest
```

`ControllerReadTest` opens a window for each input device (controller) it detects, and reports the current settings for the components. A component is a 'widget' on the device which generates input, such as a button or a DPad axis.

On my machine, `ControllerReadTest` opens four windows, for the keyboard, the mouse buttons, the mouse ball, and the game pad. (A quirk of `JInput` is that the mouse is represented by two `JInput` controllers.)

Figure 7 shows the window for the game pad.

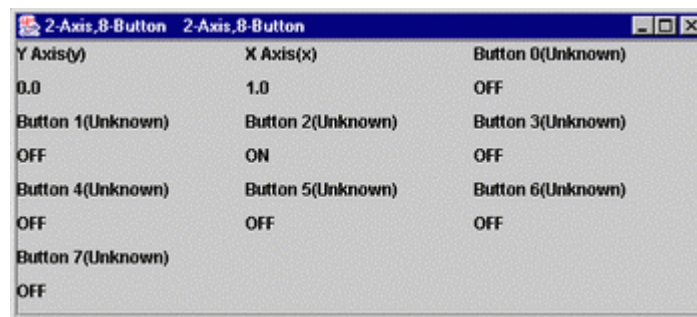


Figure 7. The Game Pad Window in `ControllerReadTest`.

The window is updated as I press the DPad and buttons. When the image in Figure 7 was generated, I was pressing the DPad on the right side, and holding down button "3". This is shown as a 1.0 value for the X Axis and an "ON" value for Button 2.

Experimenting with the DPad and buttons inside `ControllerReadTest` lets me find out what information is delivered by `JInput`. Figure 8 shows the x- and y- axis values output when I press different part of the DPad.

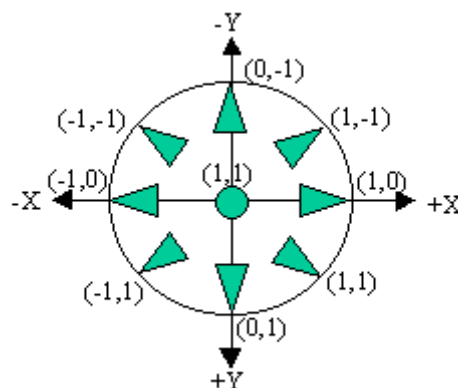


Figure 8. DPad (x, y) Values Reported by `JInput`.

One surprising aspect of Figure 8 is that the y-axis is reversed from its usual orientation. Also, all the values are normalized to be in the range -1.0 to 1.0 . Pressing the DPad in the middle is reported as $(1.0,1.0)$, which is the same as the lower right hand direction. When the DPad isn't pressed, the x- and y- values are both 0.0 .

The window in Figure 7 displays every component's name and identifier (the identifier is in brackets after the name). These are useful to know when I want to access a particular component in my JInput code.

One minor problem highlighted by Figure 7 is that all the buttons are assigned an "Unknown" identifier, rather than something more descriptive such as "button". This issue will be fixed in future versions of JInput.

3.1. The Other JInput Test Applications

Aside from ControllerReadTest, there are two other test applications in jinput.jar: ControllerTextTest and RumblerTest.

ControllerTextTest writes details about every device to standard output: it's a very long list, since there's information about every key on the keyboard. Consequently, it's a good idea to redirect the output into a text file:

```
java -cp jinput.jar;. net.java.games.input.test.ControllerTextTest
> devices.txt
```

The details supplied about my game pad are:

```
2-Axis,8-Button    2-Axis,8-Button
Type: stick
Component Count: 10
Component 0: Y Axis
  Identifier: y
  ComponentType: Absolute Analog
Component 1: X Axis
  Identifier: x
  ComponentType: Absolute Analog
Component 2: Button 0
  Identifier: Unknown
  ComponentType: Absolute Digital
Component 3: Button 1
  Identifier: Unknown
  ComponentType: Absolute Digital
Component 4: Button 2
  Identifier: Unknown
  ComponentType: Absolute Digital
Component 5: Button 3
  Identifier: Unknown
  ComponentType: Absolute Digital
Component 6: Button 4
  Identifier: Unknown
  ComponentType: Absolute Digital
Component 7: Button 5
  Identifier: Unknown
  ComponentType: Absolute Digital
Component 8: Button 6
  Identifier: Unknown
  ComponentType: Absolute Digital
```

```
Component 9: Button 7
  Identifier: Unknown
  ComponentType: Absolute Digital
```

The first two lines give the device name and type. Rather strangely, JInput believes the game pad is a joystick (a "stick"), which is actually quite common for simple game pads. Each component has an index number: for instance, the x-axis is component 1. The component names and identifiers are the same as those shown by ControllerReadTest in Figure 7.

I'll explain the "ComponentType" information later.

The RumbleTest application checks every device to see if it can generate force feedback. For my devices (a keyboard, mouse, and basic game pad), it reports "Found 0 rumblers".

4. Three JInput Applications

I'll cover the basics of JInput programming by describing three applications:

- ListControllers: it prints a list of all the detected input devices.
- ControllerDetails: it outputs a list of all of the components that form part of a specified controller.
- TestController: this displays the numbers generated by a given component as the user manipulates it.

These examples all read input from the command line and report to standard output, which allows me to put off the issue of integrating Swing and JInput until later in the chapter.

The compilation and execution of these programs will be carried out in the "JInput Applications" folder shown in Figure 6, using:

```
javac -classpath jinput.jar;. <java file>
```

and

```
java -cp jinput.jar;. <class file>
```

For example:

```
javac -classpath jinput.jar;. ListControllers.java
java -cp jinput.jar;. ListControllers
```

4.1. Listing the Controllers

On my test machine, ListController's produces the following output:

```
> java -cp jinput.jar;. ListControllers
JInput version: 1.0.0-b01
JInput logging being sent to: jinputLog.txt
0. Mouse Mouse, mouse
1. Keyboard (Keyboard), buttons
```

```
2. 2-Axis,8-Button 2-Axis,8-Button , stick
```

Each controller is assigned an index number, and the controller's name and type are listed. A controller's index number will be used as an input to ControllerDetails and TestController later on, so it's worth noting that the game pad is labeled as controller number 2.

The JInput version is reported incorrectly; it should be at least version 1.1; this bug will be fixed in a future JInput API release.

jinputLog.txt will contain JInput's start-up log, which would otherwise clutter up standard output. In earlier versions of JInput, the log data could be quite extensive. In the current version, jinputLog.txt will end up containing:

```
OS name is: Windows 98
DX8 plugin is supported
OS name is: Windows 98
DX8 plugin is supported
Creating 2-Axis,8-Button polling = false
```

This test was run on a machine running Windows 98 and DirectX 9.0, but jinput-dxplugin.dll only needs the DirectX 8.0 API.

ListControllers main() method obtains an array of controllers, and prints their names and types:

```
public static void main(String[] args)
{
    System.out.println("JInput version: " + Version.getVersion());

    Controller[] cs = getControllers();
    if (cs.length == 0) {
        System.out.println("No controllers found");
        System.exit(0);
    }

    // print the name and type of each controller
    for (int i = 0; i < cs.length; i++)
        System.out.println(i + ". " +
            cs[i].getName() + ", " + cs[i].getType());
} // end of main()
```

getController() builds the array of controllers:

```
private static Controller[] getControllers()
// obtain an array of all the input devices (controllers)
{
    PrintStream origStdout = redirectOut();
    // send stdout to a log file

    ControllerEnvironment ce =
        ControllerEnvironment.getDefaultEnvironment();
    Controller[] controllers = ce.getControllers();

    System.setOut(origStdout); // reset stdout
```

```

    return controllers;
} // end of getControllers()

```

It uses JInput's ControllerEnvironment class to ask the OS about its input devices.

The creation of the ControllerEnvironment object, and the retrieval of the controllers generates log messages. They're redirected from standard output to jinputLog.txt with redirectOut(), but the stream is switched back at the end of getControllers(). redirectOut() utilizes System.setOut() to move the stream.

The use of the static keyword for the getControllers() method has no bearing on JInput. I've used static methods in ListControllers, ControllerDetails, and TestController simply to avoid having to define classes.

4.2. Viewing Controller Details

The ControllerDetails application prints components and rumblers information for a specified controller. The data can be written to a text file or be sent to the screen.

ControllerDetails is based on a similar application by Robert Schuster in his JInput tutorial at <https://freefodder.dev.java.net/tutorial/jinputTutorialOne.html>.

ListControllers labels the game pad as controller number 2, so its information is obtained like so:

```

> java -cp jinput.jar;. ControllerDetails 2
JInput logging being sent to: jinputLog.txt
Details for: 2-Axis,8-Button 2-Axis,8-Button , stick, Unknown
Components: (10)
0. Y Axis, y, absolute, normalized, analog, 0.0
1. X Axis, x, absolute, normalized, analog, 0.0
2. Button 0, button, absolute, normalized, digital, 0.0
3. Button 1, button, absolute, normalized, digital, 0.0
4. Button 2, button, absolute, normalized, digital, 0.0
5. Button 3, button, absolute, normalized, digital, 0.0
6. Button 4, button, absolute, normalized, digital, 0.0
7. Button 5, button, absolute, normalized, digital, 0.0
8. Button 6, button, absolute, normalized, digital, 0.0
9. Button 7, button, absolute, normalized, digital, 0.0
No Rumblers
No subcontrollers

```

The "Details for:" line gives the name and type of the controller, and its port type. The port type should be "Usb", but is shown as "Unknown" (due to DirectX).

The ten components consist of x- and y-axes and eight buttons, which confirms the information that I gathered from Windows and JInput's ControllerReadTest.

Each component has an index which I'll need later for TestController. For instance, the y-axis is component index 0.

I'll explain the data on the component lines below.

Mouse information can be stored in mouseDevice.txt by calling:

```
java -cp jinput.jar;. ControllerDetails 0 mouseDevice.txt
```

The file will contain:

```
JInput logging being sent to: jinputLog.txt
Details for: Mouse Mouse, mouse, Unknown
No Components
No Rumbler
No. of subcontrollers: 2
-----
Subcontroller: 0
Details for: Mouse Mouse ball, ball, Unknown
Components: (3)
0. X-axis, x, relative, arbitrary, analog, 0.0
1. Y-axis, y, relative, arbitrary, analog, 0.0
2. Wheel, slider, relative, arbitrary, analog, 0.0
No Rumbler
No subcontrollers
-----
Subcontroller: 1
Details for: Mouse Mouse buttons, buttons, Unknown
Components: (3)
0. Button 0, Left, absolute, normalized, digital, 0.0
1. Button 1, Right, absolute, normalized, digital, 0.0
2. Button 2, Middle, absolute, normalized, digital, 0.0
No Rumbler
No subcontrollers
```

This illustrates how a controller can contain subcontrollers. The top-level mouse controller has two subcontrollers, one for the mouse ball, the other for its buttons.

ControllerDetails' main() method obtains an array of controllers in the same way as ListControllers, and examines one of them.

```
public static void main(String[] args)
{
    if (args.length < 1)
        System.out.println("Usage: ControllerDetails <index> [<fnm>");
    else {
        Controller[] cs = getControllers();
        if (cs.length == 0) {
            System.out.println("No controllers found");
            System.exit(0);
        }

        int index = extractIndex(args[0], cs.length);
            // get controller index from the command line

        PrintStream ps = getPrintStream(args);
        printDetails(cs[index], ps); // print details for the controller
        ps.close();
    }
} // end of main()
```

extractIndex() parses the command line argument to extract the index integer, and checks that it's valid (i.e. between 0 and cs.length-1).

getPrintStream() links a PrintStream to the file named on the command line; if no filename is supplied then it uses System.out (stdout) instead.

printDetails() reports on the specified controller's components and rumblers. If it finds any subcontrollers, then it recursively visits them, and reports their details.

```
private static void printDetails(Controller c, PrintStream ps)
{
    ps.println("Details for: " + c.getName() + ", " +
              c.getType() + ", " + c.getPortType() );

    printComponents(c.getComponents(), ps);
    printRumblers(c.getRumblers(), ps);

    // print details about any subcontrollers
    Controller[] subCtrls = c.getControllers();
    if (subCtrls.length == 0)
        ps.println("No subcontrollers");
    else {
        ps.println("No. of subcontrollers: " + subCtrls.length);
        // recursively visit each subcontroller
        for (int i = 0; i < subCtrls.length; i++) {
            ps.println("-----");
            ps.println("Subcontroller: " + i);
            printDetails( subCtrls[i], ps);
        }
    }
} // end of printDetails()
```

Examining a Component

A component can be many things: a button, a slider, a dial. However, every component has a name and type (called, rather misleadingly, an identifier), and generates a value when manipulated. A component has four attributes: relative (or absolute), normalized (or arbitrary), analog (or digital), and a dead zone setting

Relative/Absolute. A relative component returns a value relative to the previously output value. For example, a relative positional component will return the distance moved since it was last polled. An absolute component produces a value that doesn't depend on the previous value. The attribute is tested with Component.isRelative().

Normalized/Arbitrary. A normalized component outputs values between -1.0f and 1.0f, with 0.0f representing 'not manipulated'. An arbitrary component has no range restrictions. The attribute is tested with Component.isNormalized().

Analog/Digital. An analog component can have more than two values. For instance, a game pad x-axis component can return three different values corresponding to being pressed on the left, on the right, or not being pressed at all. A digital component only has two possible values, which is suitable for boolean devices such as buttons. This attribute is tested with Component.isAnalog().

A fourth attribute, mainly for joystick devices, is the **dead zone** value. It specifies a threshold before the component switches from 0.0f (representing 'off') to an 'on' value. This mechanism means that slight changes in joystick position can be ignored.

printComponents() is defined as:

```

private static void printComponents(
    Component[] comps, PrintStream ps)
{ if (comps.length == 0)
  ps.println("No Components");
  else {
    ps.println("Components: (" + comps.length + ")");
    for (int i = 0; i < comps.length; i++)
      ps.println( i + ". " +
        comps[i].getName() + ", " +
        getIdentifierName(comps[i]) + ", " +
        (comps[i].isRelative() ? "relative" : "absolute") + ", " +
        (comps[i].isNormalized() ? "normalized":"arbitrary")+ ", " +
        (comps[i].isAnalog() ? "analog" : "digital") + ", " +
        comps[i].getDeadZone());
  }
} // end of printComponents()

```

getIdentifierName() augments the **Component.getIdentifier()** method:

```

private static String getIdentifierName(Component comp)
{
  Component.Identifier id = comp.getIdentifier();
  if (id == Component.Identifier.Button.UNKNOWN)
    return "button"; // an unknown button
  else if (id == Component.Identifier.Key.UNKNOWN)
    return "key"; // an unknown key
  else
    return id.getName();
}

```

If the component's identifier is UNKNOWN, then the returned string is set to "button" or "key" depending on the identifier type (it would otherwise be the string "Unknown"). As I mentioned earlier, this identifier problem will be fixed in a future version of JInput.

Many up-market gaming devices offer force feedback, which is delivered via a **Rumbler** object in JInput. The presence of rumpblers is reported by **printRumpblers()**:

```

private static void printRumpblers(Rumbler[] rumpblers, PrintStream ps)
{
  if (rumpblers.length == 0)
    ps.println("No Rumpblers");
  else {
    ps.println("Rumpblers: (" + rumpblers.length + ")");
    for (int i=0; i < rumpblers.length; i++)
      ps.println(i + ". " +
        rumpblers[i].getAxisName() + " on axis " +
        rumpblers[i].getAxisIdentifier().getName());
  }
} // end of printRumpblers()

```

A rumbler has a name and identifier. There's also a **Rumbler.rumble()** method for delivering a certain level of feedback to the component.

My Game Pad Again

The output from ControllerDetails for my game pad is:

```
0. Y Axis, y, absolute, normalized, analog, 0.0
1. X Axis, x, absolute, normalized, analog, 0.0
2. Button 0, button, absolute, normalized, digital, 0.0
3. Button 1, button, absolute, normalized, digital, 0.0
4. Button 2, button, absolute, normalized, digital, 0.0
5. Button 3, button, absolute, normalized, digital, 0.0
6. Button 4, button, absolute, normalized, digital, 0.0
7. Button 5, button, absolute, normalized, digital, 0.0
8. Button 6, button, absolute, normalized, digital, 0.0
9. Button 7, button, absolute, normalized, digital, 0.0
```

This shows that all the components return absolute and normalized values. The 0.0's at the end of every line mean that none of the components use dead zones.

The button values will probably be 1.0f and 0.0f for 'on' and 'off', and the axes will generate -1.0f, 0.0f, or 1.0f. However, I need to check these numbers, which is the purpose of the TestController application.

4.3. Testing a Controller

TestController shows me the numbers generated by a component when I manipulate it. The program is called with a controller index and a component index. For instance:

```
java -cp jinput.jar;. TestController 2 0
```

The '2' denotes the game pad, and the '0' is for the DPad's y-axis. The component index comes from the output of ListControllers, and the component index from ControllerDetails.

When a component is 'pressed', a value is printed to the screen. Keeping the component pressed doesn't generate multiple outputs, and releasing the component doesn't trigger an output either. These design decisions mean that the screen isn't swamped with numbers.

In the example below, I pressed the DPad at the top and bottom several times. I terminated the program by typing ctrl-c.

```
> java -cp jinput.jar;. TestController 2 0
JInput logging being sent to: jinputLog.txt
No. of controllers: 3
Polling controller: 2-Axis,8-Button 2-Axis,8-Button , stick
No. of components: 10
Component: Y Axis
-1.0; -1.0; 1.0; 1.0; -1.0; 1.0; 1.0; 1.0; 1.0;
-1.0; 1.0; 1.0; 1.0; 1.0; 1.0;
>
```

The -1.0 values were printed when I pushed the DPad at the top, and the 1.0's appeared when I pressed it at the bottom or in the center. This output corresponds to

the numbers obtained from JInput's ControllerReadTest application (shown in Figure 7).

TestController shares a lot of code with my earlier examples:

```
public static void main(String[] args)
{
    if (args.length < 2) {
        System.out.println("Usage: TestController <index>
                               <component index>");
        System.exit(0);
    }

    // get a controller using the first index value
    Controller c = getController(args[0]);

    // get a component using the second index value
    Component component = getComponent(args[1], c);

    pollComponent(c, component);    // keep polling the component
} // end of main()
```

The component polling is carried out in pollComponent():

```
// global constant
private static final int DELAY = 75;    // ms    (polling interval)

private static void pollComponent(Controller c, Component component)
{
    float prevValue = 0.0f;
    float currValue;

    int i = 1;    // used to format the output
    while (true) {
        try {
            Thread.sleep(DELAY);    // wait a while
        }
        catch (Exception ex) {}

        c.poll();    // update the controller's components
        currValue = component.getPollData();    // get the current value
        if (currValue != prevValue) {    // the value has changed
            if (currValue != 0.0f) {    // don't show component 'releases'
                System.out.print(currValue + "; ");
                i++;
            }
            prevValue = currValue;
        }

        if (i%10 == 0) {    // after several outputs, put in a newline
            System.out.println();
            i = 1;
        }
    }
} // end of pollComponent()
```

The method repeatedly polls the specified component, sleeping for DELAY ms (75 ms) between each one. The polling is done in two steps: first Controller.poll() makes the controller update its components' values, then the new component value is retrieved by Component.getPollData().

To cut down the volume of data, a new value is only displayed if it's different from the previous one. Also, I've decided not to sure component 'releases' (e.g. when the user stops pressing the DPad), which would appear as 0's.

5. A Game Pad Controller

The ListControllers, ControllerDetails, and TestController examples let me examine the game pad to see what components it offers, and what values those components generate.

I can use that information to write a GamePadController class which hides the JInput classes, and makes it easier to access the DPad and button values. GamePadController is utilized in the Swing example and the Java 3D first-person shooter in the next sections.

It's surprisingly difficult to write a general-purpose class due to the wide variety of components that a game pad might possess. For example, the popular Logitech Dual Action game pad has one DPad, two analog joysticks, and 12 buttons.

Consequently, I won't write a one-size-fits-all class that tries to support every possible combination of components. Instead, GamePadController manages a single DPad offering up x- and y- axis values and 8 buttons, all with normalized and absolute attributes. In other words, GamePadController is aimed at my type of game pad. This approach keeps the class' interface simple, while illustrating how JInput processing can be hidden.

Figure 9 shows the class diagram for GamePadController, with only its public methods listed.

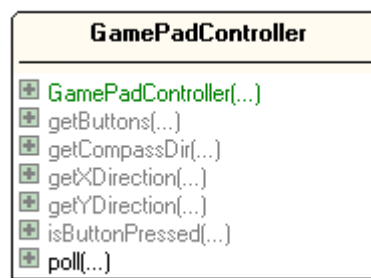


Figure 9. Class Diagram for GamePadController.

The axes values can be accessed individually (with getXDirection() and getYDirection()), or as a single compass direction (getCompassDir()). Rather than return floats, the methods serve up constants, such as UP, DOWN, LEFT and RIGHT for the directions, and NW, NORTH, NE, EAST, and so on, for the compass headings.

An individual button can be checked with `isButtonPressed()`, or all the button settings can be retrieved in an array (`getButtons()`). The 'on' and 'off' settings are represented by booleans.

The `poll()` method updates all the component values.

5.1. Initializing the Controller

`GamePadController()` obtains a list of controllers, then searches through it for the game pad. If a controller is found, then its components are searched to find the indices of the x- and y- axes and the buttons.

```
// globals
public static final int NUM_BUTTONS = 8;

private Controller controller; // for the game pad
private Component[] comps; // game pad components

// these variables will hold component indices
private int xAxisIdx, yAxisIdx; // for the x- and y- axes
private int buttonsIdx[]; // for the buttons

public GamePadController()
{
    // initialize components index stores
    xAxisIdx = -1;
    yAxisIdx = -1;
    buttonsIdx = new int[NUM_BUTTONS];
    for(int i=0; i < NUM_BUTTONS; i++)
        buttonsIdx[i] = -1;

    // get all the controllers
    Controller[] cs = getControllers();
    if (cs.length == 0) {
        System.out.println("No controllers found");
        System.exit(0);
    }

    controller = findGamePad(cs); // find the game pad controller
    System.out.println("Game controller: " +
        controller.getName() + ", " +
        controller.getType());

    // get the controller's components
    comps = controller.getComponents();

    getComponentsIndices(comps);
} // end of main()
```

`xAxisIdx`, `yAxisIdx`, and the `buttonsIdx[]` array will hold the index positions of their components. Storing the indices means that a particular component can be accessed quickly later, without needing to search through the controller's components information every time.

`getControllers()` calls `ControllerEnvironment.getControllers()` to get an array of controllers. While this method is executing, standard output is redirected to a file, so any log details generated by `JInput` won't appear on screen.

`findGamePad()` loops through the controllers array looking for a game pad.

```
private Controller findGamePad(Controller[] cs)
{
    Controller.Type type;
    int index = 0;
    while(index < cs.length) {
        type = cs[index].getType();
        if ((type == Controller.Type.GAMEPAD) ||
            (type == Controller.Type.STICK))
            break;
        index++;
    }
    if (index == cs.length) {
        System.out.println("No game pad found");
        System.exit(0);
    }

    return cs[index];
} // end of findGamePad()
```

The only unusual aspect of this method is that it checks if the controller type is a `GAMEPAD` or a `STICK` (i.e. a joystick). This extra test is due to the output generated by `ControllerDetails`, which lists my game pad as a joystick.

5.2. Checking the Components

`getComponentsIndicies()` records the indices of the x- and y- axes and buttons in `xAxisIdx`, `yAxisIdx`, and `buttonsIdx[]`. All the components should be absolute and normalized, and there should be at least `NUM_BUTTONS` buttons

```
private void getComponentsIndicies(Component[] comps)
{
    if (comps.length == 0) {
        System.out.println("No Components found");
        System.exit(0);
    }

    boolean foundXAxis = false;
    boolean foundYAxis = false;
    int numButtons = 0;
    Component c;

    for(int i=0; i < comps.length; i++) {
        c = comps[i];
        if ((c.getIdentifier() == Component.Identifier.Axis.X) &&
            hasRequiredAttrs(c)) { // deal with x-axis component
            if (foundXAxis)
                System.out.println("Found another x-axis: " +
                    c.getName() + "! Ignoring it.");
            else {
                xAxisIdx = i; // store x-axis index
                System.out.println("Found: " + c.getName() + "; index: " + i);
            }
        }
    }
}
```

```

        foundXAxis = true;
    }
}
else if ((c.getIdentifier() == Component.Identifier.Axis.Y) &&
        hasRequiredAttrs(c)) { // deal with y-axis component
    if (foundYAxis)
        System.out.println("Found another y-axis: " +
            c.getName() + "! Ignoring it.");
    else {
        yAxisIdx = i; // store y-axis index
        System.out.println("Found: " + c.getName()+ "; index: " + i);
        foundYAxis = true;
    }
}
else if (isButton(c)) { // deal with a button
    if (numButtons == NUM_BUTTONS) // already enough buttons
        System.out.println("Found an extra button. Ignoring it");
    else {
        buttonsIdx[numButtons] = i; // store button index
        System.out.println("Found: " + c.getName()+ "; index: " + i);
        numButtons++;
    }
}
else // some other kind of component
    System.out.println("Skipping: " + c.getName());
} // end of for-loop

// check if enough axes and buttons
if (!foundXAxis) {
    System.out.println("No x-axis component found");
    System.exit(0);
}
if (!foundYAxis) {
    System.out.println("No y-axis component found");
    System.exit(0);
}
if (numButtons < NUM_BUTTONS) {
    System.out.println("Too few buttons (" + numButtons +
        "); expecting " + NUM_BUTTONS);
    System.exit(0);
}
} // end of getComponentsIndicies()

```

`getComponentIndicies()` ignores extra x- and y- axes components, and any buttons over the required `NUM_BUTTONS` amount. However, if an x- or y- axis isn't found, or there's less than `NUM_BUTTONS` buttons, then the program exits.

`hasRequiredAttrs()` makes sure that a component has absolute and normalized attributes:

```

private boolean hasRequiredAttrs(Component c)
{ if (!c.isRelative() && c.isNormalized())
    return true;
  return false;
}

```

isButton() returns true if the supplied component is a button. The component needs to be digital with suitable attributes, and its identifier class name must end with "Button" (i.e. it must be Component.Identifier.Button).

```
private boolean isButton(Component c)
{
    if (!c.isAnalog() && hasRequiredAttrs(c)) {
        // a suitable digital component
        String className = c.getIdentifier().getClass().getName();
        if (className.endsWith("Button"))
            return true;
    }
    return false;
} // end of isButton()
```

I used the class name to identify the button since Component.Identifier.Button defines numerous button identifiers, and I didn't want to check the component against each one.

5.3. Polling the Device

An application will utilize the GamePadController by calling its poll() method to update the component's values, and then one or more of its get methods to retrieve the new values.

GamePadController's poll() method calls poll() in the controller:

```
public void poll()
{ controller.poll(); }
```

5.4. Reading the Axes

A convenient way of representing DPad x- and y- axis values are as compass directions, as shown in Figure 10.

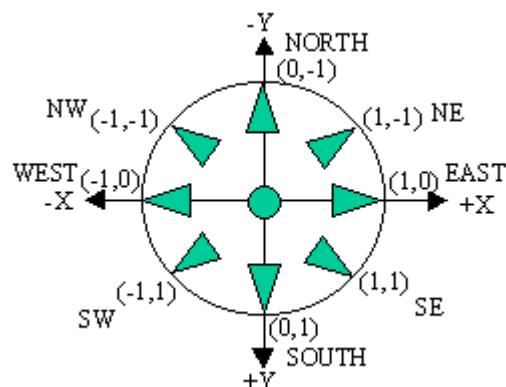


Figure 10. Compass Directions for the DPad.

The axes values in Figure 10 come from Figure 8.

`getCompassDir()` returns a compass direction by interpreting the x- and y- axis data according to Figure 10:

```
// global public DPad compass positions
public static final int NUM_COMPASS_DIRS = 9;

public static final int NW = 0;
public static final int NORTH = 1;
public static final int NE = 2;
public static final int WEST = 3;
public static final int NONE = 4;
public static final int EAST = 5;
public static final int SW = 6;
public static final int SOUTH = 7;
public static final int SE = 8;

public int getCompassDir()
{
    float xc = comps[xAxisIdx].getPollData();
    float yc = comps[yAxisIdx].getPollData();

    if ((yc == -1.0f) && (xc == -1.0f)) // (y,x)
        return NW;
    else if ((yc == -1.0f) && (xc == 0.0f))
        return NORTH;
    else if ((yc == -1.0f) && (xc == 1.0f))
        return NE;
    else if ((yc == 0.0f) && (xc == -1.0f))
        return WEST;
    else if ((yc == 0.0f) && (xc == 0.0f))
        return NONE;
    else if ((yc == 0.0f) && (xc == 1.0f))
        return EAST;
    else if ((yc == 1.0f) && (xc == -1.0f))
        return SW;
    else if ((yc == 1.0f) && (xc == 0.0f))
        return SOUTH;
    else if ((yc == 1.0f) && (xc == 1.0f))
        return SE;
    else {
        System.out.println("Unknown (x,y): (" + xc + ", " + yc + ")");
        return NONE;
    }
} // end of getCompassDir()
```

The x- and y- axis numbers are retrieved from their components by using their index positions and `getPollData()`:

```
float xc = comps[xAxisIdx].getPollData();
float yc = comps[yAxisIdx].getPollData();
```

The compass constants in `GamePadController` are public so they can be utilized in other classes which use the data returned from `getCompassDir()`.

The x- and y- axis values can be read individually by using `getXDirection()` and `getYDirection()`. These methods return integer direction constants rather than `JInput` floats. `getXDirection()` is defined as:

```
// global public DPad axis positions for (x,y)
public static final int LEFT = 9;      // for the x-axis
public static final int RIGHT = 10;
public static final int UP = 11;      // for the y-axis
public static final int DOWN = 12;

public int getXDirection()
// return the x-axis value (RIGHT, LEFT, or NONE)
{
    float xc = comps[ xAxisIdx ].getPollData();
    if (xc == 1.0f)
        return RIGHT;
    else if (xc == -1.0f)
        return LEFT;
    else
        return NONE;
} // end of getXDirection()
```

getYDirection() is similar, but returns UP, DOWN, or NONE.

The direction constants are public so that other classes can use them.

5.5. Reading the Buttons

getButtons() returns all the button values in a single array; each value is represented by a boolean.

```
public boolean[] getButtons()
{
    boolean[] buttons = new boolean[ NUM_BUTTONS ];
    float value;
    for(int i=0; i < NUM_BUTTONS; i++) {
        value = comps[ buttonsIdx[i] ].getPollData();
        buttons[i] = ((value == 0.0f) ? false : true);
    }
    return buttons;
} // end of getButtons()
```

A JInput button value (a float) is read by selecting the relevant component using its index, and calling getPollData():

```
value = comps[ buttonsIdx[i] ].getPollData();
```

A single button can be accessed with isButtonPressed():

```
public boolean isButtonPressed(int pos)
{
    if ((pos < 1) || (pos > NUM_BUTTONS)) {
        System.out.println("Button position out of range (1-" +
            NUM_BUTTONS + "): " + pos);
        return false;
    }

    float value = comps[ buttonsIdx[pos-1] ].getPollData();
```

```

    // array range is 0-NUM_BUTTONS-1
    return ((value == 0.0f) ? false : true);
} // end of isButtonPressed()

```

The supplied button position (`pos`) is expected to be in the range 1 to `NUM_BUTTONS`, matching the labels on the game pad.

5.6. Other Approaches

`GamePadController` is designed for a 2-axis, 8-button game pad. Its interface is simpler to use than `JInput` directly, but it can't deal with other game pad configurations. As you might expect, there are other ways of packaging up `JInput`

Xj3D (<http://www.xj3d.org/>) is a toolkit for building X3D-based applications which uses `JInput` to support input devices such as data gloves, joysticks, tracking devices, and game pads. (X3D is the ISO standard for real-time 3D graphics, a successor to VRML, but with extras such as humanoid animation and NURBS.)

Xj3D devices are accessible via a variety of classes (e.g. `GamepadDevice`, `JoystickDevice`, `TrackerDevice`, `WheelDevice`, and `MouseDevice`), each of which maintains a game state class (e.g. `GamepadState`, `JoystickState`, `WheelState`). `GamepadState` supports several buttons, two sticks, a slider, and a DPad. The documentation for these classes can be found at <http://www.xj3d.org/javadoc/>.

The Lightweight Java Game Library (**LWJGL**, <http://lwjgl.org>) utilizes `JInput` with a wrapper for manipulating component axes and buttons. The LWJGL tutorials page has an article on how to use it (<http://lwjgl.org/wiki/doku.php/lwjgl/tutorials/input/basiccontroller>).

6. Swing and JInput

The `GamePadViewer` application is shown in Figure 11.

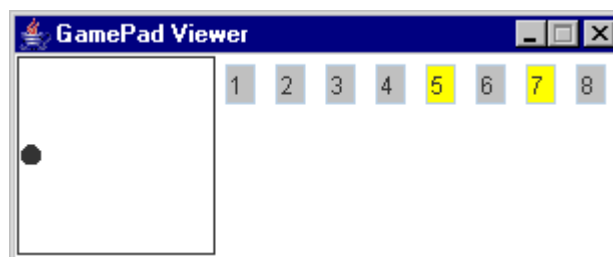


Figure 11. The `GamePadViewer` Application.

It's meant to emulate the Test window in the Game Controller application in Windows (see Figure 4). The left-hand canvas contains a dot that represents the current position of the DPad, while the eight textfields show the status of the game pad's buttons. A yellow textfield means the button with that number is being pressed.

Figure 11 shows the situation when the DPad is being pressed on the left side, and buttons 5 and 7 are being held down.

Figure 12 shows GamePadViewer's class diagrams, with only the public methods visible.

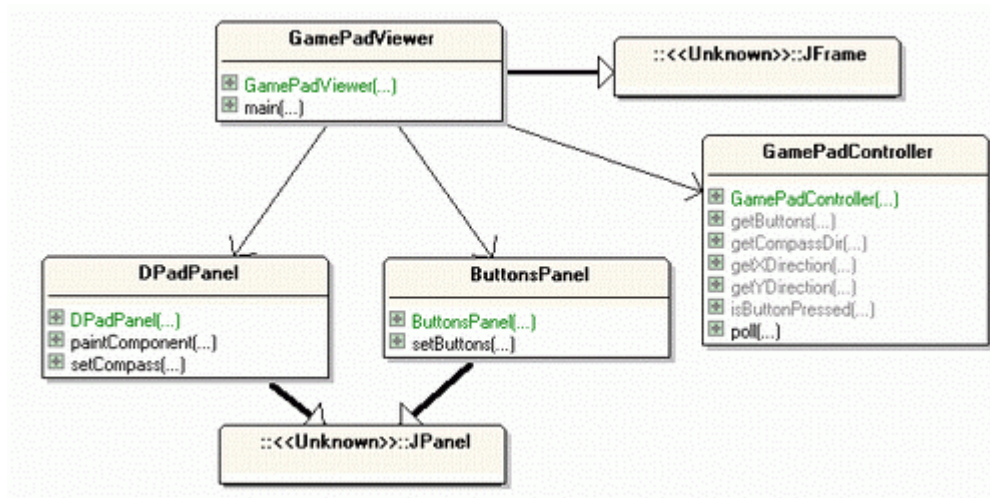


Figure 12. Class Diagrams for GamePadViewer.

The GamePadViewer class is the top-level JFrame, which constructs its GUI from a DPadPanel object for the left-hand canvas, and a ButtonsPanel for the buttons.

Periodically, GamePadViewer polls GamePadController, and gathers data about the DPad and buttons. It passes that data to the GUI classes via DPadPanel.setCompass() and ButtonsPanel.setButtons(), which update their GUI elements.

6.1. Constructing the Application

The GamePadViewer constructor gets DPadPanel and ButtonsPanel to build their parts of the GUI. It initiates polling by calling startPolling().

```

// globals
private GamePadController gpController;
private DPadPanel dpadPanel; // shows DPad
private ButtonsPanel buttonsPanel; // shows buttons

private Timer pollTimer; // timer which triggers the polling

public GamePadViewer()
{
    super("GamePad Viewer");

    gpController = new GamePadController();

    Container c = getContentPane();
    c.setLayout( new BorderLayout() );

    dpadPanel = new DPadPanel();
    c.add(dpadPanel, "West");

    buttonsPanel = new ButtonsPanel();
    c.add(buttonsPanel, "Center");
  
```

```

addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { pollTimer.stop(); // stop the timer
      System.exit(0);
    }
});

pack();
setResizable(false);
setVisible(true);

startPolling();
} // end of GamePadViewer()

```

`startPolling()` creates an `ActionListener` object that polls the game pad and updates the GUI. It also starts a timer which activates the object every `DELAY` ms.

```

// global
private static final int DELAY = 75; // ms (polling interval)

private void startPolling()
{
    ActionListener pollPerformer = new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            gpController.poll();

            // get new data from the game pad
            int compassDir = gpController.getCompassDir();
            boolean[] buttons = gpController.getButtons();

            // update the GUI
            dpadPanel.setCompass(compassDir);
            buttonsPanel.setButtons(buttons);
        }
    }; // end of ActionListener

    pollTimer = new Timer(DELAY, pollPerformer);
    pollTimer.start();
} // end of startPolling()

```

`startPolling()` illustrates the standard technique for integrating `JInput` polling and Swing GUI updates. The tricky issue is that changes to the GUI must be performed from the event-dispatching thread. `startPolling()` does this by employing `javax.swing.Timer`, which schedules its `ActionListener` argument in that thread.

It's important that the `ActionListener` code executes quickly, since the GUI can't respond to user actions while the code is being run.

`DpadPanel.setCompass()` stores the new compass direction in the `dpadPanel` object, and then triggers a repaint. The black circle's drawing position is determined by looking up an array of (x,y) coordinates, using the compass heading as an index.

`ButtonsPanel.setButtons()` cycles through the supplied boolean array, and changes the backgrounds of its textfields accordingly: yellow means "on", gray is "off".

7. FPShooter3D with JInput

Back in chapter 24, I described a 3D first-person shooter application called FPShooter3D. The player's movements, and gun firing, are controlled by the user pressing keys on the keyboard. Figure 13 shows a screenshot.



Figure 13. The FPShooter3D Application.

7.1. The Problem with Key Presses

Keyboard processing is handled by KeyBehavior, a subclass of Java 3D's Behavior class. KeyBehavior's processStimulus() is called whenever a key press is detected.

```
public void processStimulus(Enumeration criteria)
// KeyBehavior responds to a keypress
{
    WakeupCriterion wakeup;
    AWTEvent[] event;

    while( criteria.hasMoreElements() ) {
        wakeup = (WakeupCriterion) criteria.nextElement();
        if( wakeup instanceof WakeupOnAWTEvent ) {
            event = ((WakeupOnAWTEvent)wakeup).getAWTEvent();
            for( int i = 0; i < event.length; i++ ) {
                if( event[i].getID() == KeyEvent.KEY_PRESSED )
                    processKeyEvent( (KeyEvent)event[i] );
            }
        }
    }
    wakeupOn( keyPress );
} // end of processStimulus()
```

Although this code can handle multiple key presses, usually the event[] array will only contain the key press that triggered the method call. This means that it's not generally possible for a player to hold down multiple keys, and obtain a combination of key actions. For instance, the user can't move and fire the gun simultaneously.

This drawback is fixed in the JInput version of FPShooter3D, where the user can move, rotate, and fire the gun *at the same time*. This is possible because the new behaviour (in GamePadBehavior) is driven by periodic polling of the game pad rather than by keypress events.

7.2. Changing FPSShooter3D

Figure 14 gives the class diagrams for the new FPSShooter3D, showing only the visible methods.

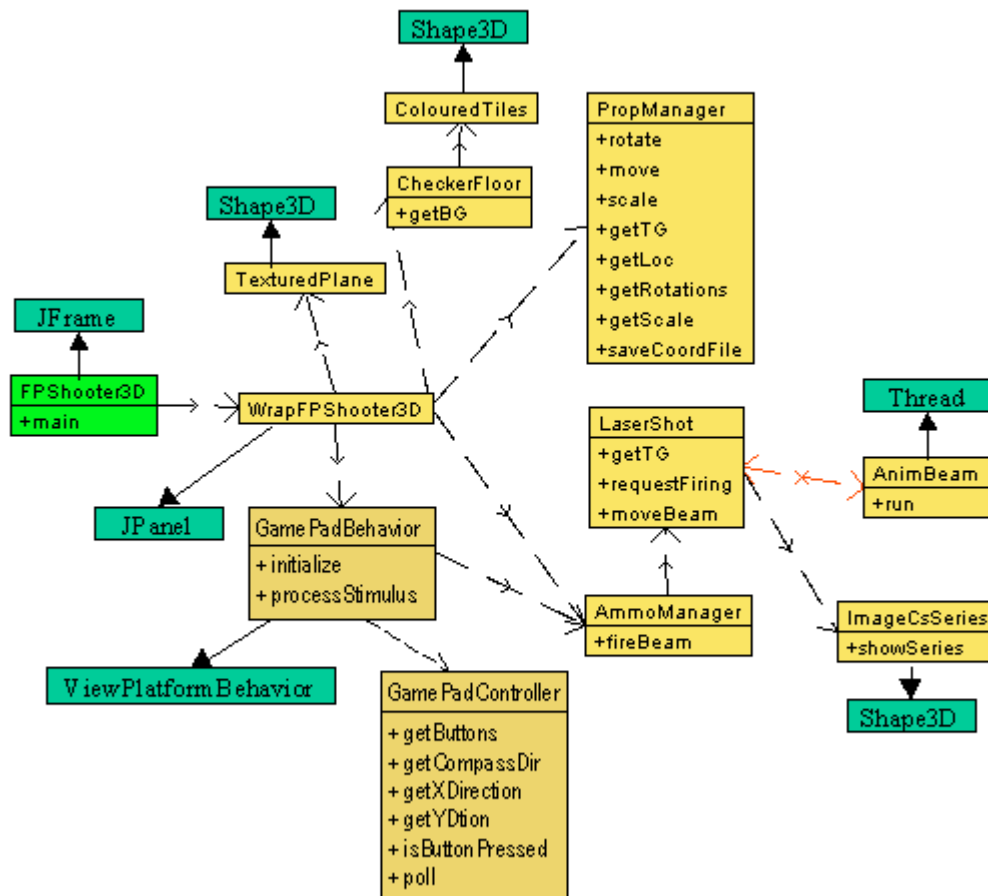


Figure 14. Class Diagrams for the JInput-enabled FPSShooter3D.

The application is quite complex, but most of it is the same as in chapter 24. The main changes are:

- the KeyBehavior class has been replaced by GamePadBehavior;
- the WrapFPSShooter3D class, which sets up the 3D world, now creates and initializes a GamePadBehavior object rather than a KeyBehavior instance;
- GamePadBehavior utilizes GamePadController to access the game pad.

I won't talk about the unchanged parts of FPSShooter3D; if you're unfamiliar with those classes, please look back at chapter 24. Also, the GamePadController is the same as earlier, so I won't explain that again either.

All that's left to describe is GamePadBehavior, and the small piece of code in WrapFPSShooter3D that calls it.

7.3. The Game Pad Behavior

GamePadBehavior is a time-driven behavior, triggered into action every DELAY ms.

```
public class GamePadBehavior extends ViewPlatformBehavior
{
    private static final int DELAY = 75;    // ms (polling interval)
    // more constants

    private WakeupCondition wakeUpCond;
    private AmmoManager ammoMan;
    private GamePadController gamePad;
    // more globals go here

    public GamePadBehavior(AmmoManager am)
    { ammoMan = am;
      gamePad = new GamePadController();
      wakeUpCond = new WakeupOnElapsedTime(DELAY);
    } // end of GamePadBehavior()

    public void initialize()
    { wakeupOn(wakeUpCond); }

    //more methods go here
} // end of GamePadBehavior
```

GamePadBehavior extends ViewPlatformBehavior so the scene graph's ViewPlatform's transform group, targetTG, is available. GamePadBehavior adjusts the player's camera position by applying translations and rotations to targetTG.

The AmmoManager object (ammoMan) is used to fire 'bullets' at the robot target.

Both of these techniques come from KeyBehavior. The major change is the use of a WakeupOnElapsedTime condition in GamePadBehavior; KeyBehavior uses WakeupOnAWTEvent to wake up when there are key presses.

7.4. Polling the Game Pad

processStimulus() is radically different from the version in KeyBehavior. When it's triggered, it polls the game pad controller, gets the latest DPad and button values, and updates the camera accordingly.

```
public void processStimulus(Enumeration criteria)
{
    gamePad.poll();

    // get new data from the game pad
    int compassDir = gamePad.getCompassDir();
    boolean[] buttons = gamePad.getButtons();

    processCompass(compassDir);
    processButtons(buttons);

    wakeupOn(wakeUpCond);    // make sure we are notified again
} // end of processStimulus()
```

The next scene graph frame is rendered only when processStimulus() has finished, so all the camera changes (and any shots) will appear in the scene at the same time.

7.5. Processing the DPad

The compass direction can be one of nine possible values (the eight compass headings, and NONE). I decided to allocate the moves and rotations shown in Figure 15 to the eight headings.

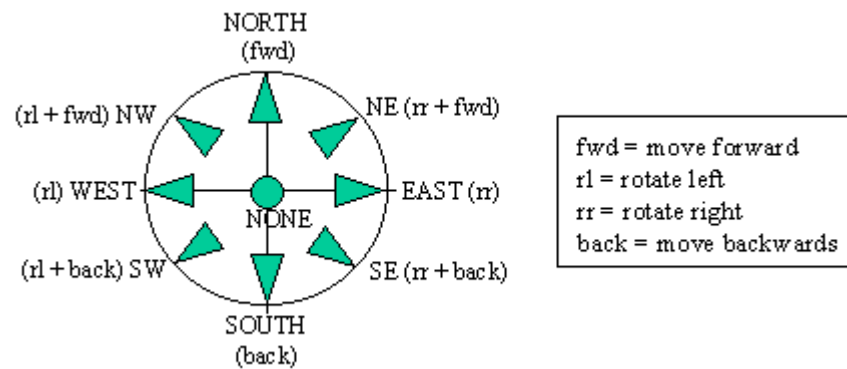


Figure 15. Moves and Rotations for the Compass Headings.

The camera can move forwards, backwards, and rotate left or right. Some of the headings carry out a rotation and a move.

processCompass() is a large switch statement to determine which heading needs to be handled.

```
// globals
private static final double ROT_AMT = Math.PI / 36.0; // 5 degrees
private static final double MOVE_STEP = 0.2;

// hardcoded movement vectors
private static final Vector3d FWD = new Vector3d(0,0,-MOVE_STEP);
private static final Vector3d BACK = new Vector3d(0,0,MOVE_STEP);
private static final Vector3d LEFT = new Vector3d(-MOVE_STEP,0,0);
private static final Vector3d RIGHT = new Vector3d(MOVE_STEP,0,0);
private static final Vector3d UP = new Vector3d(0,MOVE_STEP,0);
private static final Vector3d DOWN = new Vector3d(0,-MOVE_STEP,0);

private void processCompass(int compassDir)
{
    switch (compassDir) {
        case GamePadController.NW: rotateY(ROT_AMT); doMove(FWD); break;
        // rotate left and fwd
        case GamePadController.NORTH: doMove(FWD); break;
        case GamePadController.NE: rotateY(-ROT_AMT); doMove(FWD); break;
        // rotate right and fwd

        case GamePadController.WEST: rotateY(ROT_AMT); break;
        case GamePadController.NONE: break; // do nothing
        case GamePadController.EAST: rotateY(-ROT_AMT); break;

        case GamePadController.SW: rotateY(ROT_AMT); doMove(BACK); break;
```

```

// rotate left and back
case GamePadController.SOUTH: doMove(BACK); break;
case GamePadController.SE: rotateY(-ROT_AMT); doMove(BACK); break;
// rotate right and back
default:
    System.out.println("Did not recognise compass dir: " +
        compassDir);
    break;
}
} // end of processCompass()

```

The rotateY() and doMove() methods are unchanged from KeyBehavior in chapter 24.

```

// globals for repeated calculations
private Transform3D t3d = new Transform3D();
private Transform3D toMove = new Transform3D();
private Transform3D toRot = new Transform3D();

private void rotateY(double radians)
// rotate about y-axis by radians
{
    targetTG.getTransform(t3d);
    // targetTG is the ViewPlatform's transform
    toRot.rotY(radians);
    t3d.mul(toRot);
    targetTG.setTransform(t3d);
} // end of rotateY()

private void doMove(Vector3d theMove)
{ targetTG.getTransform(t3d);
  toMove.setTranslation(theMove);
  t3d.mul(toMove);
  targetTG.setTransform(t3d);
} // end of doMove()

```

7.6. Processing the Buttons

The buttons on the top of the game pad (buttons 1-4; see Figure 1) move the camera down, right, left, or up. The front buttons (buttons 5-8; see Figure 2) make the gun fire.

All the buttons are checked, so combinations of button actions are possible.

```

private void processButtons(boolean[] buttons)
{
    if (buttons[0]) // button 1
        doMove(DOWN);
    if (buttons[1]) // button 2
        doMove(RIGHT);
    if (buttons[2]) // button 3
        doMove(LEFT);
    if (buttons[3]) // button 4
        doMove(UP);

    if (buttons[4] || buttons[5] || buttons[6] || buttons[7])
        ammoMan.fireBeam(); // buttons 5-8
}

```

```
} // end of processButtons()
```

7.7. Creating a Game Pad Behaviour

WrapFPSShooter3D creates a GamePadBehavior object, and makes it the controlling behaviour for the camera (the view platform):

```
GamePadBehavior padBeh = new GamePadBehavior(ammoMan);
padBeh.setSchedulingBounds(bounds);

ViewingPlatform vp = su.getViewingPlatform();
vp.setViewPlatformBehavior(padBeh);
```

The ammoMan argument to GamePadBehavior is a reference to the AmmoManager, which fires the gun.

8. Distributing a JInput Application

A potential problem with applications that use JInput is that it isn't part of the standard set of packages in the JDK/JRE. This means that an application must include JInput in its distribution, or a user won't be able to execute it straight 'out of the box'.

On Windows, the JInput API is just two files: jinput.jar and jinput-dxplugin.dll; both files are very small, 66 KB and 86 KB respectively.

Two ways of building Java distributions are:

- use a cross-platform, native installer, such as install4j, or
- employ Java Web Start (JWS), a Web-enabled installer for Java applications.

Both approaches can easily create a single downloadable file which contains Java classes, JARs, and DLLs. Appendix A and B explain how this is done for applications which use Java 3D. Java 3D poses the same problems as JInput: it's not a standard part of J2SE, and the additional libraries are a mix of JARs and DLLs (on Windows).

An example JWS deployment file for JInput can be found at <http://www.newdawnsoftware.com/resources/jinput/webstart/jinput2.jnlp>.

9. Java 3D and Input Devices

FPSShooter3D employs a time-based behaviour to periodically poll the game pad. However, Java 3D offers another mechanism, based around its InputDevice interface and sensor classes, which I'll briefly consider.

Device polling and data retrieval using JInput must be added to InputDevice's pollAndProcessInput() method. Depending on the device type, pollAndProcessInput() will be called by Java 3D regularly, or when there's a request for more device data by the application.

The class implementing InputDevice must use a Java 3D Sensor object to store a sequence of SensorRead objects for the data coming from the input device. A

SensorRead object contains a time-stamp, and a 3D transform for the sensor or the status of the device's buttons or switches.

Each time that pollAndProcessInput() is called, a new SensorRead object must be added to the end of the Sensor object's sequence.

Other parts of the application can retrieve SensorRead values directly from the Sensor sequence, or be sent sensor events. A listener can either be notified of button presses or sensor transforms.

The reason that Sensor maintains a sequence of SensorRead objects, is to allow the application to carry out predictive sensor calculations. It's possible to generate a new SensorRead object based on previous SensorRead data in the sequence.

There's a VirtualInputDevice demo in Sun's Java 3D API download, which illustrates many of these techniques. Rather than using listeners, the example employs a behaviour to access the Sensor object in each frame, and read the newest SensorRead value.

The InputDevice and sensor design pattern is overly complex for something as simple as a game pad. I've no need for predictive data calculation or a SensorRead class.

The InputDevice approach may be more suitable for devices such as headtrackers, where the device generates positional and rotational information that naturally maps to Java 3D transforms. Predictive calculations may be useful if the device is connected via a slow or erratic network link, so that inputs often arrive too slowly or not at all.

Most of chapter 13 in:

Java Media APIs: Cross-Platform Imaging, Media, and Visualization
Alejandro Terrazas, John Ostuni, Michael Barlow
SAMS, 2002

details how to use the InputDevice and sensor classes to build a headtracking application.

10. Alternatives to JInput

JXInput (<http://www.hardcode.de/jxinput/>) provides access to game controllers under Windows, and is quite similar to JInput. For each device, JXInput can manage up to 6 axes (3 positional and 3 rotational), 2 sliders, 4 hats, and 128 buttons. It supports callbacks and events, and has a Java 3D InputDevice implementation.

JXInput is only available under Windows, since it relies on DirectInput. It was developed in late 2002 by Joerg Plewe and Stefan Pfafferott, as part of their very entertaining FlyingGuns Java 3D game (<http://www.flyingguns.com/>).

JavaJoystick (<http://sourceforge.net/projects/javajoystick/>) isn't just for joysticks, but any input device with 2-6 degrees of freedom (which includes game pads). There's a listener class, JoystickListener, with callbacks for when a button or axis changes, and it's also possible to use polling. JavaJoystick is implemented on Windows and Linux. It dates from 2001, and hasn't been updated since 2003.