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, or steering wheel, plug it in, and play games the way that nature intended?

It's not that difficult, as the next two chapters will show. I'll start by giving some background on JInput (https://jinput.dev.java.net/), an open-source API for game controller discovery and polled input.

I'll explain how to program with JInput by describing three small applications that display information about the input devices attached to your PC. I'll also develop a GamePadController class which offers a simplified interface to my game pad. GamePadController is utilized in two examples: a Swing application called GamePadViewer in this chapter, and a revised version of Arms3D (the articulated arms example from chapter N4) in the next chapter.

Arms3D illustrates how a game pad can enhance the user's playing experience, replacing a confusing mix of keys with more intuitive buttons and sticks. The game pad also allows multiple inputs to be processed at once, so that different parts of the grabber's arms can rotate and move at the same time.

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 a Mac OS X version as well. Posts to the JInput forum at 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 May 2006 Windows version, which I downloaded as jinput_windows_20060514.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 the first draft of this chapter.

2. First the Game Pad and Windows

A lot of heartaches can be avoided 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.

My game pad is of Chinese origin, without a brand name. 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.

It offers a direction pad (D-Pad) on the left, two analog sticks in the center, and 12 buttons. The "ANALOG" button located between the sticks in Figure 1 switches them on and off. Functionally, the device is very similar to the popular Logitech Dual Action game pad.

Eagle-eyed readers may note that there only seem to be ten buttons (excluding the "ANALOG" button). The other two are built into the analog sticks, which can be pressed down to act as buttons.

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. This is done through the Game Controllers application accessible through the Control Panel in Windows XP.

Figure 3 shows part of the Game Controllers main window.

Figure 3. Part of the Game Controllers Window.

Figure 3 gives the first indication of how Windows 'sees' the game pad: as a "USB Vibration Joystick", which isn't particularly informative. More details are available via the "Properties" button, which leads to testing and calibration windows for the device.

Its important that the game pad be calibrated so that Windows correctly interprets its inputs. Poor calibration will manifest itself at the JInput level as values which are incorrectly rounded or have the wrong sign.

Part of the Game Controllers "Function Test" window is shown in Figure 4.

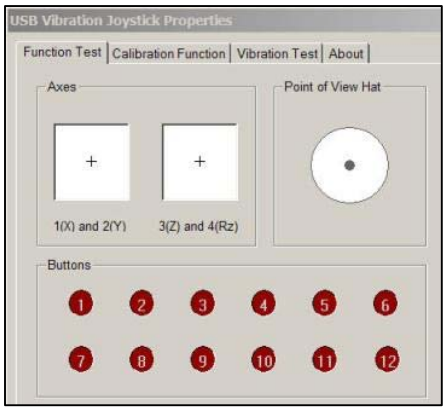

Figure 4. Testing the Game Pad.

When the "ANALOG" button is on, the game pad's two analog sticks affect the crosshairs inside the axes squares, the D-Pad draws an arrow inside the right-hand circle, and button presses light up the radio boxes. When the "ANALOG" button is off, stick movement is ignored, and the D-Pad affects the crosshair in the left-hand square.

When the analog sticks are active, the crosshairs can be moved anywhere in their boxes. The position of each crosshair is specified using two axes: X and Y for the left-hand stick, and Z and Rz for the right.

For the D-Pad, it's only possible to draw an arrow in eight compass positions around the edge of the circle. The test window calls the D-Pad a "Point of View" Hat (POV Hat), terminology that I'll employ from now on.

This game pad also has a vibration, or rumbler, feature, which is testable via a separate "Vibration Test" window. Rumbling can be switched on for the left or right sides of the pad, or both sides at once.

After you've finished calibration and testing, you can be confident that the game pad is recognized by the OS, and you'll also have a good idea about your game pad's input capabilities. My no-name game pad has two analog sticks, a POV hat, 12 buttons, and rumbler capability.

Another important check is to find out whether DirectX can see your device, since JInput uses DirectX under Windows. An easy way of doing this is with the dxdiag utility, Window'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.

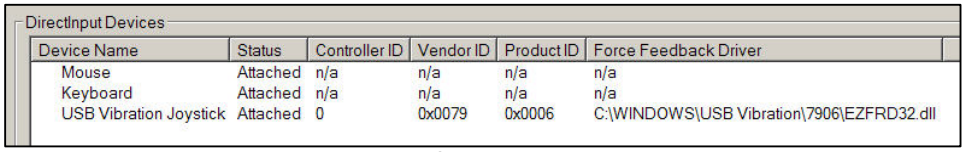

Figure 5. Part of the dxdiag Input Tab.

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

3. Installing and Testing JInput

As I mentioned earlier, I downloaded jinput_windows_20060514.zip from https://jinput.dev.java.net. It came from the 'Win32' folder under the "Documents & Files" menu item. It contains three files: jinput-windows.jar (the high-level Java part of the API) and jinput-dx8.dll and jinput-raw.dll (the OS-level parts).

I extracted the files, and placed them in their own folder in a convenient location; I chose d:\jinput\bin (see Figure 6).

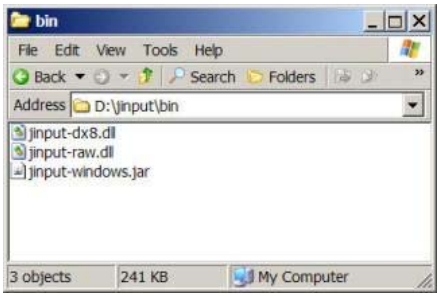

Figure 6. A Basic JInput Application Folder.

Two other essential downloads from https://jinput.dev.java.net/ are the API documentation and the tests JAR, both available from the 'core' folder under the "Documents & Files" menu item

The tests JAR, jinput-tests-20060514.jar, contains three test applications, ControllerReadTest, ControllerTextTest, and RumbleTest. Calling them can be a bit tricky since the java command line must include classpath and library path settings for the JInput software in d:\jinput\jar, and must correctly name the required application inside the tests JAR.

The applications are called like so:

```
java -cp d:\jinput\bin\jinput-windows.jar;.
     -Djava.library.path="d:\jinput\bin"
          net.java.games.input.test.ControllerReadTest

java -cp d:\jinput\bin\jinput-windows.jar;.
     -Djava.library.path="d:\jinput\bin"
          net.java.games.input.test.ControllerTextTest

java -cp d:\jinput\bin\jinput-windows.jar;.
     -Djava.library.path="d:\jinput\bin"
          net.java.games.input.test.RumbleTest
```

Of course, the directory paths will depend on where you placed the JInput library files.

Typing these long lines is rather tiring, so I packaged the command line inside runTest.bat. It contains:

```
java -cp d:\jinput\bin\jinput-windows.jar;.
     -Djava.library.path="d:\jinput\bin"
          net.java.games.input.test.%1
```

This means that the program calls become:

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

3.1. Examining the Input Devices

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, stick axis, or hat key.

On my machine, ControllerReadTest opens three sub-windows, for the keyboard, the mouse, and the game pad. Figure 7 shows the window for the game pad.

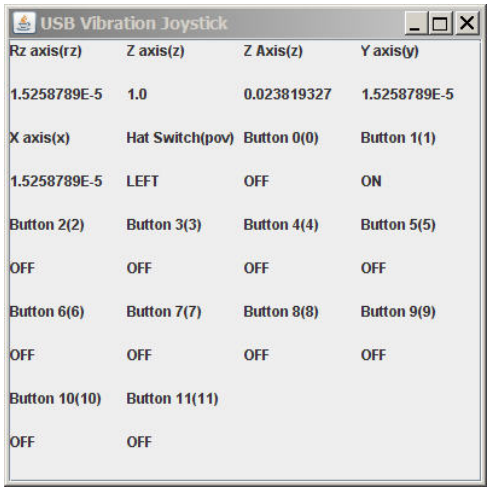

Figure 7. The Game Pad Window in ControllerReadTest.

The window is updated as I press the sticks, hat, and buttons. When the image in Figure 7 was generated, I was pressing the POV hat on the left side, holding down the button labeled "2", and pushing the right-hand stick directly to the right. That's shown as a "LEFT" value for Hat Switch (pov), "ON" for Button 1(1), and a 1.0 value for the Z axis(z).

Experimenting with the sticks, hat, and buttons inside ControllerReadTest is a great way of finding out what information is delivered by JInput.

Figure 8 illustrates the value ranges generated by the left and right sticks. The left stick produces floats between 1.0 and -1.0 for the x- and y- axes, and the right stick similar values for the z- and rz- axes.

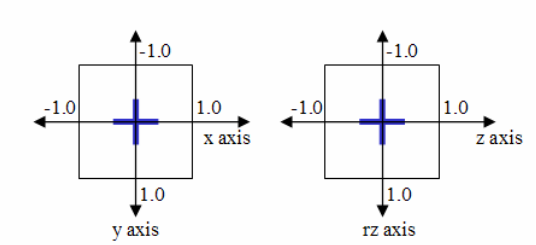

Figure 8. Axes values for the Left and Right Analog Sticks.

One surprising aspect of Figure 8 is that the y- and rz- axes are reversed from their usual orientations. Also, when a stick isn't being used, the axes values are *close* to 0.0f, but not exactly zero (as can be seen in Figure 7).

Hat (D-Pad) positions are reported as float constants, as summarized by Figure 9.

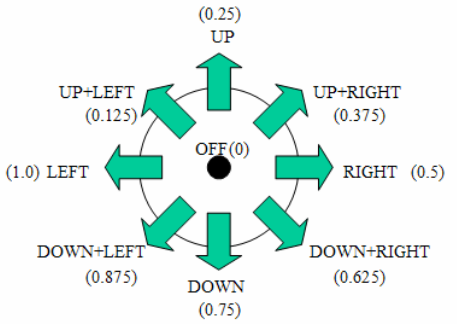

Figure 9. Hat Constants.

The hat consists of four buttons (for left, right, up, and down); the other constants are generated by holding down adjacent keys. For example, DOWN+LEFT is reported when left and down are pressed together.

The float values for these constants are given in the Component.POV class documentation.

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 you want to access a particular component in your JInput code.

One minor problem with my game pad, highlighted by Figure 7, is that there are two z-axes! One is called "Z **a**xis(z)", the other "Z **A**xis(z)" – note the case change for "axis". When the "ANALOG" button is switched on, the second z-axis ("Z **A**xis(z)") responds in the same way as the x axis for the left stick, so is superfluous. The first z-axis ("Z **a**xis(z)") is used with the right hand stick, so is what I need in my code.

3.2. The Other JInput Test Applications

Aside from ControllerReadTest, there are two other test applications in jinput-tests-20060514.jar: ControllerTextTest and RumblerTest.

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

```
runTest ControllerTextTest > devices.txt
```

The details supplied about my game pad are:

```
USB Vibration Joystick
Type: Stick
Component Count: 18
Component 0: Rz axis
    Identifier: rz
    ComponentType: Absolute Analog
Component 1: Z axis
    Identifier: z
    ComponentType: Absolute Analog
Component 2: Z Axis
    Identifier: z
    ComponentType: Absolute Analog
Component 3: Y axis
    Identifier: y
    ComponentType: Absolute Analog
Component 4: X axis
    Identifier: x
    ComponentType: Absolute Analog
Component 5: Hat Switch
    Identifier: pov
    ComponentType: Absolute Digital
Component 6: Button 0
    Identifier: 0
    ComponentType: Absolute Digital
Component 7: Button 1
    Identifier: 1
    ComponentType: Absolute Digital
Component 8: Button 2
    Identifier: 2
    ComponentType: Absolute Digital
Component 9: Button 3
    Identifier: 3
    ComponentType: Absolute Digital
Component 10: Button 4
    Identifier: 4
    ComponentType: Absolute Digital
Component 11: Button 5
    Identifier: 5
    ComponentType: Absolute Digital
Component 12: Button 6
```



```
    Identifier: 6
    ComponentType: Absolute Digital
Component 13: Button 7
    Identifier: 7
    ComponentType: Absolute Digital
Component 14: Button 8
    Identifier: 8
    ComponentType: Absolute Digital
Component 15: Button 9
    Identifier: 9
    ComponentType: Absolute Digital
Component 16: Button 10
    Identifier: 10
    ComponentType: Absolute Digital
Component 17: Button 11
    Identifier: 11
    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. Each component has an index number: for instance, the x-axis is component 4. 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 game pad, it reports "Found 5 rumblers", and progresses to test each one. It's rather hard to tell the differences between them, but the first rumbler setting seem to activate the left rumbler, the second setting controls the right rumbler, and the other three trigger both rumblers together.

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 (controllers).
- 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 is carried out with the help of two batch files which add in the necessary classpath and library path settings. compileJI.bat contains:

```
javac -classpath d:\jinput\bin\jinput-windows.jar;. %1
```


runJI.bat holds:

```
java -cp d:\jinput\bin\jinput-windows.jar;.
     -Djava.library.path="d:\jinput\bin" %1 %2 %3
```

The %1, %2, and %3 arguments allow at most three arguments to the java call.

For example:

```
compileJI ListControllers.java
runJI ListControllers
```

4.1. Listing the Controllers

On my test machine, ListControllers produces the following output:

```
> runJI ListControllers
Executing ListControllers with JInput...
JInput version: 2.0.0-b01
0. Mouse, Mouse
1. Keyboard, Keyboard
2. USB Vibration Joystick, 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 a input to ControllerDetails and TestController later on, so it's worth noting that the game pad is controller number 2.

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());

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

  // 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()
```

It uses JInput's ControllerEnvironment class to ask the OS about its input devices, which are retrieved as an array of Controller objects.

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 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:

```
> runJI ControllerDetails 2
Executing ControllerDetails with JInput...
Details for: USB Vibration Joystick, Stick, Unknown
Components: (18)
0. Rz axis, rz, absolute, analog, 0.0
1. Z axis, z, absolute, analog, 0.0
2. Z Axis, z, absolute, analog, 0.0
3. Y axis, y, absolute, analog, 0.0
4. X axis, x, absolute, analog, 0.0
5. Hat Switch, pov, absolute, digital, 0.0
6. Button 0, 0, absolute, digital, 0.0
7. Button 1, 1, absolute, digital, 0.0
8. Button 2, 2, absolute, digital, 0.0
9. Button 3, 3, absolute, digital, 0.0
10. Button 4, 4, absolute, digital, 0.0
11. Button 5, 5, absolute, digital, 0.0
12. Button 6, 6, absolute, digital, 0.0
13. Button 7, 7, absolute, digital, 0.0
14. Button 8, 8, absolute, digital, 0.0
15. Button 9, 9, absolute, digital, 0.0
16. Button 10, 10, absolute, digital, 0.0
17. Button 11, 11, absolute, digital, 0.0
Rumblers: (5)
0. null on axis; no name
1. null on axis; no name
2. null on axis; no name
3. null on axis; no name
4. null on axis; no name
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 restrictions in DirectX).

The 18 components consist of five axes, a hat, and 12 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 3.

The rumbler information is rather sparse, since there's no identifier or name for any of the rumblers.

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

Information can also be sent to a file. For example, details on the mouse can be stored in mouseDevice.txt by calling:


```
runJI ControllerDetails 0 mouseDevice.txt
```

The file will contain:

```
Details for: Mouse, Mouse, Unknown
Components: (6)
0. X-axis, x, relative, analog, 0.0
1. Y-axis, y, relative, analog, 0.0
2. Wheel, z, relative, analog, 0.0
3. Button 0, Left, absolute, digital, 0.0
4. Button 1, Right, absolute, digital, 0.0
5. Button 2, Middle, absolute, digital, 0.0
No Rumblers
No subcontrollers
```

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 {
    ControllerEnvironment ce =
          ControllerEnvironment.getDefaultEnvironment();
    Controller[] cs = ce.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()
```

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

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 three attributes: relative (or absolute), 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().

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 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() in ControllerDetails 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].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").

The presence of rumblers is reported by printRumblers():

```
private static void printRumblers(Rumbler[] rumblers,
                                        PrintStream ps)
{ if (rumblers.length == 0)
    ps.println("No Rumblers");
  else {
    ps.println("Rumblers: (" + rumblers.length + ")");
    Component.Identifier rumblerID;
    for (int i=0; i < rumblers.length; i++) {
      rumblerID = rumblers[i].getAxisIdentifier();
      ps.print(i + ". " + rumblers[i].getAxisName() +
                                          " on axis; ");
      if (rumblerID == null)
        ps.println("no name");
      else
        ps.println("name: " + rumblerID.getName() );
    }
  }
}  // end of printRumblers()
```

A rumbler may have a name and identifier. There's also a Rumbler.rumble() method to make the component vibrate. The setting can range between 0.0f and 1.0f, with 1.0f meaning full on.

My Game Pad Again

Here's the component output from ControllerDetails for my game pad once again:

```
0. Rz axis, rz, absolute, analog, 0.0
1. Z axis, z, absolute, analog, 0.0
2. Z Axis, z, absolute, analog, 0.0
3. Y axis, y, absolute, analog, 0.0
4. X axis, x, absolute, analog, 0.0
5. Hat Switch, pov, absolute, digital, 0.0
6. Button 0, 0, absolute, digital, 0.0
7. Button 1, 1, absolute, digital, 0.0
```



```
8. Button 2, 2, absolute, digital, 0.0
9. Button 3, 3, absolute, digital, 0.0
10. Button 4, 4, absolute, digital, 0.0
11. Button 5, 5, absolute, digital, 0.0
12. Button 6, 6, absolute, digital, 0.0
13. Button 7, 7, absolute, digital, 0.0
14. Button 8, 8, absolute, digital, 0.0
15. Button 9, 9, absolute, digital, 0.0
16. Button 10, 10, absolute, digital, 0.0
17. Button 11, 11, absolute, digital, 0.0
```

The list shows that all the components return absolute values. The 0.0's at the end of every line indicate that none of the components use dead zones, which is a little problematic for the analog sticks. It probably means that they'll be unlikely to return 0 when switched 'off'.

The next step is to test the various components to get an idea of the values they return when activated. For example, what numbers do the buttons deliver when pressed and released? What floats are generated when the hat buttons are clicked? What values are returned when an analog stick is moved around? These questions are answered by the TestController application in the next subsection.

4.3. Testing a Controller

TestController displays the numbers generated by a component when it's manipulated. The program is called with controller and component index arguments. For instance:

```
runJI TestController 2 0
```

The '2' denotes the game pad, and the '0' is for the right-hand analog stick's vertical axis (rz-axis). The controller index value 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 a component doesn't trigger an output either. These design decisions mean that the screen isn't swamped with numbers.

In the example below, I moved the right-hand analog stick up and down several times (i.e. along the rz-axis); I terminated the program by typing ctrl-c.

```
> runJI TestController 2 0
Executing TestController with JInput...
No. of controllers: 3
Polling controller: USB Vibration Joystick, Stick
No. of components: 18
Component: Rz axis
1.0; -0.8110323; -1.0; -0.9448844; -1.0; -1.0; -0.92126346;
1.0; 0.9763485; 0.36220336; 1.0; 0.46456087; 1.0; -0.4016022;
-1.0; Failed to poll device: Device is released
Controller no longer valid
Terminate batch job (Y/N)? y
>
```


The ctrl-c generates a rather messy series of diagnostic messages, starting with "Failed to poll device". Also, the abnormal termination of the batch file requires the user to type "y" to return to the DOS prompt.

The -1.0 values were printed when I pushed the stick fully down, and the 1.0's appeared when I pushed it up. This output matches the numbers obtained from JInput's ControllerReadTest application (shown in Figure 7).

The output is complicated by intermediate values reported when the stick is moved slowly from it's center (near to 0) towards an edge (where it reports +/-1.0).

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 = 40;  // ms  (polling interval)
private static final float EPSILON = 0.0001f;
                      // to deal with values near to 0.0


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 current value
    if (currValue != prevValue) {  // the value has changed
      if (Math.abs(currValue) > EPSILON) {
        // only show values not near to 0.0f
        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 (40 ms) between each one. The interval has to be short in order to catch rapid presses.

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 don't show component 'releases' (i.e. when the user stops pressing a button, the hat, or stick), which would appear as 0's, or values close to 0.0f.

5. A Game Pad Controller

My GamePadController class makes the accessing of the game pad's sticks, hat, buttons, and rumbler simpler. It's utilized in the Swing example in the next section, and the Arms3D application in the next chapter.

It's surprisingly difficult to write a general-purpose game pad controller due to the wide variety of components that a game pad might possess – buttons, sticks, hats, D-Pads, and rumblers. Consequently, I haven't written a one-size-fits-all class that tries to support every possible combination of components. Instead, GamePadController manages a device with 12 buttons, two analog sticks, a hat, and rumbler. In other words, GamePadController is aimed at my game pad. This keeps the class simple, while illustrating how JInput processing can be hidden.

Figure 10 shows the class diagram for GamePadController, with only its public methods listed.

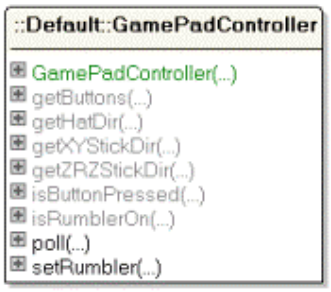

Figure 10. Class Diagram for GamePadController.

GamePadController simplifies the interface to the components by returning the stick and hat positions as compass directions. getXYStickDir() returns a compass value for

the left stick (which manages the x- and y- axes), getZRZStickDir() returns a compass direction for the right stick (it deals with the z- and rz axes), and getHatDir() does the same for the hat.

The possible compass values are shown in Figure 11.

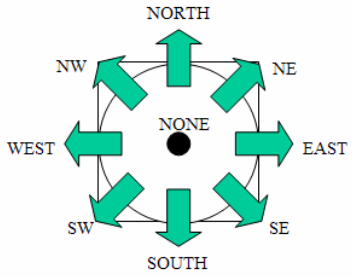

Figure 11. Compass Directions for the Sticks and Hat.

This compass approach hides the numerical details of the stick axes, and offers a single way of thinking about the sticks and POV hat. A drawback is that the compass view restricts the values a stick can return; for example, it's not possible to get a position half-way between SW and SOUTH.

A button is 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 rumbler is switched on and off with setRumbler(), and its current status retrieved with isRumblerOn(). GamePadController only offers access to a single rumbler, and the choice of which one is hard-wired into the class.

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 sticks axes, the POV hat, the buttons, and a rumbler.

```
// globals
private Controller controller;  // for the game pad


public GamePadController()
{
  // get the controllers
  ControllerEnvironment ce =
       ControllerEnvironment.getDefaultEnvironment();
  Controller[] cs = ce.getControllers();
  if (cs.length == 0) {
    System.out.println("No controllers found");
    System.exit(0);
  }
  else
    System.out.println("Num. controllers: " + cs.length);
```



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

  // collect indices for the required game pad components
  findCompIndices(controller);

  findRumblers(controller);
} // end of GamePadController()
```

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);
  }
  else
    System.out.println("Game pad index: " + i);

  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

findCompIndices() records the indices of the sticks axes, the POV hat, and the buttons in xAxisIdx, yAxisIdx, zAxisIdx, rzAxisIdx, povIdx, and the buttonsIdx[].

Storing the indices means that a particular component can be accessed quickly later, without needing to search through the game controller's components information every time.

```
// globals
private Component[] comps;  // holds the components

// comps[] indices for specific components
private int xAxisIdx, yAxisIdx, zAxisIdx, rzAxisIdx;
                           // indices for the analog sticks axes
private int povIdx;        // index for the POV hat
```



```
private void findCompIndices(Controller controller)
{
  comps = controller.getComponents();
  if (comps.length == 0) {
    System.out.println("No Components found");
    System.exit(0);
  }
  else
    System.out.println("Num. Components: " + comps.length);

  // get indices for sticks axes: (x,y) and (z,rz)
  xAxisIdx = findCompIndex(comps,
                   Component.Identifier.Axis.X, "x-axis");
  yAxisIdx = findCompIndex(comps,
                   Component.Identifier.Axis.Y, "y-axis");

  zAxisIdx = findCompIndex(comps,
                   Component.Identifier.Axis.Z, "z-axis");
  rzAxisIdx = findCompIndex(comps,
                   Component.Identifier.Axis.RZ, "rz-axis");

  // get POV hat index
  povIdx = findCompIndex(comps,
                   Component.Identifier.Axis.POV, "POV hat");

  findButtons(comps);
}  // end of findCompIndices()
```

findCompIndices() delegates the index searches to findCompIndex() for the axes and POV hat, and to findButtons() for the buttons.

```
private int findCompIndex(Component[] comps,
                      Component.Identifier id, String nm)
{
  Component c;
  for(int i=0; i < comps.length; i++) {
    c = comps[i];
    if ((c.getIdentifier() == id) && !c.isRelative()) {
      System.out.println("Found " + c.getName() + "; index: " + i);
      return i;
    }
  }

  System.out.println("No " + nm + " component found");
  return -1;
}  // end of findCompIndex()
```

findCompIndex() searches through comps[] for the supplied component ID, returning the corresponding array index, or -1 if the identifier can't be found. The component must also be absolute.

findButtons() performs a similar search through comps[], collecting the indices for a maximum of NUM_BUTTONS buttons, which it stores in buttonsIdx[].

```
// globals
public static final int NUM_BUTTONS = 12;
private int buttonsIdx[];   // indices for the buttons
```



```
private void findButtons(Component[] comps)
{
  buttonsIdx = new int[NUM_BUTTONS];
  int numButtons = 0;
  Component c;

  for(int i=0; i < comps.length; i++) {
    c = comps[i];
    if (isButton(c)) {    // deal with a button
      if (numButtons == NUM_BUTTONS)   // already enough buttons
        System.out.println("Found an extra button; index: " +
                                              i + ". Ignoring it");
      else {
        buttonsIdx[numButtons] = i;  // store button index
        System.out.println("Found " + c.getName() + "; index: " + i);
        numButtons++;
      }
    }
  }

  // fill empty spots in buttonsIdx[] with -1's
  if (numButtons < NUM_BUTTONS) {
    System.out.println("Too few buttons (" + numButtons +
                         "); expecting " + NUM_BUTTONS);
    while (numButtons < NUM_BUTTONS) {
      buttonsIdx[numButtons] = -1;
      numButtons++;
    }
  }
}  // end of findButtons()
```

findButtons() ignores extra buttons, and if there aren't enough, then it fills the empty spots in buttonsIdx[] with -1's.

isButton() returns true if the supplied component is a button. The component needs to be digital and absolute, 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() && !c.isRelative()) {  // digital & absolute
    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. Finding the Rumblers

The rumblers are accessed using Controller.getRumblers(), and one is chosen to be the rumbler offered by the class.

```
// globals
private Rumbler[] rumblers;
private int rumblerIdx;      // index for the rumbler being used


private void findRumblers(Controller controller)
{
  // get the game pad's rumblers
  rumblers = controller.getRumblers();
  if (rumblers.length == 0) {
    System.out.println("No Rumblers found");
    rumblerIdx = -1;
  }
  else {
    System.out.println("Rumblers found: " + rumblers.length);
    rumblerIdx = rumblers.length-1;    // use last rumbler
  }
}  // end of findRumblers()
```

findRumblers() stores the index of the last rumbler in the rumblers array. There's no particular reason for this choice, except that in my game pad it vibrates the left and right sides of the game pad together.

5.4. 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();  }
```

5.5. Reading the Stick Axes

The two sticks return axes values as floats between -1.0 and 1.0 (see Figure 8). GamePadController simplifies this interface by returning compass directions (as shown in Figure 11).

The left analog stick manages the x- and y- axes, while the right stick deals with the z- and rz- axes. GamePadController's public stick methods call getCompassDir() with the relevant component indices for the axes.

```
public int getXYStickDir()
// return the (x,y) analog stick compass direction
{
  if ((xAxisIdx == -1) || (yAxisIdx == -1)) {
    System.out.println("(x,y) axis data unavailable");
```



```
    return NONE;
  }
  else
    return getCompassDir(xAxisIdx, yAxisIdx);
} // end of getXYStickDir()


public int getZRZStickDir()
// return the (z,rz) analog stick compass direction
{
  if ((zAxisIdx == -1) || (rzAxisIdx == -1)) {
    System.out.println("(z,rz) axis data unavailable");
    return NONE;
  }
  else
    return getCompassDir(zAxisIdx, rzAxisIdx);
} // end of getXYStickDir()
```

getCompassDir() retrieves the floats for the required axes, and converts them into a compass heading.

```
// global public stick and hat 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;  // default value
public static final int EAST = 5;
public static final int SW = 6;
public static final int SOUTH = 7;
public static final int SE = 8;


private int getCompassDir(int xA, int yA)
{
  float xCoord = comps[xA].getPollData();
  float yCoord = comps[yA].getPollData();

  int xc = Math.round(xCoord);
  int yc = Math.round(yCoord);

  if ((yc == -1) && (xc == -1))   // (y,x)
    return NW;
  else if ((yc == -1) && (xc == 0))
    return NORTH;
  else if ((yc == -1) && (xc == 1))
    return NE;
  else if ((yc == 0) && (xc == -1))
    return WEST;
  else if ((yc == 0) && (xc == 0))
    return NONE;
  else if ((yc == 0) && (xc == 1))
    return EAST;
  else if ((yc == 1) && (xc == -1))
    return SW;
  else if ((yc == 1) && (xc == 0))
    return SOUTH;
  else if ((yc == 1) && (xc == 1))
```



```
    return SE;
  else {
    System.out.println("Unknown (x,y): (" + xc + "," + yc + ")");
    return NONE;
  }
}  // end of getCompassDir()
```

The axes value are retrieved by using their index positions and Component.getPollData():

```
float xCoord = comps[xA].getPollData();
float yCoord = comps[yA].getPollData();
```

They're rounded to integers (either 0 or 1), and a series of if-tests determine which compass setting should be returned. The rounding shows that position information is lost, but with the aim of simplifying the interface.

The compass constants in GamePadController are public so they can be utilized in other classes which use the compass data.

5.6. Reading the POV Hat

The polling of the hat returns a float corresponding to different combinations of key presses (see Figure 9). getHatDir() maps this to the same compass directions as used by the sticks.

```
public int getHatDir()
{
  if (povIdx == -1) {
    System.out.println("POV hat data unavailable");
    return NONE;
  }
  else {
    float povDir = comps[povIdx].getPollData();
    if (povDir == POV.CENTER)  // 0.0f
      return NONE;
    else if (povDir == POV.DOWN)  // 0.75f
      return SOUTH;
    else if (povDir == POV.DOWN_LEFT)  // 0.875f
      return SW;
    else if (povDir == POV.DOWN_RIGHT)  // 0.625f
      return SE;
    else if (povDir == POV.LEFT)  // 1.0f
      return WEST;
    else if (povDir == POV.RIGHT)  // 0.5f
      return EAST;
    else if (povDir == POV.UP)  // 0.25f
      return NORTH;
    else if (povDir == POV.UP_LEFT)  // 0.125f
      return NW;
    else if (povDir == POV.UP_RIGHT)  // 0.375f
      return NE;
    else  { // assume center
      System.out.println("POV hat value out of range: " + povDir);
      return NONE;
    }
  }
```



```
}  // end of getHatDir()
```

5.7. Reading the Buttons

getButtons() returns all the button values in a single array, each value 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 is 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.

5.8. Using the Rumbler

The rumbler is switched on or off with setRumbler().

```
// global
private boolean rumblerOn = false;   // is rumbler is on or off


public void setRumbler(boolean switchOn)
{
  if (rumblerIdx != -1) {
    if (switchOn)
```



```
      rumblers[rumblerIdx].rumble(0.8f);  // almost full on
    else  // switch off
      rumblers[rumblerIdx].rumble(0.0f);
    rumblerOn = switchOn;    // record rumbler's new status
  }
}  // end of setRumbler()
```

The rumbler vibrates with a value of 0.8f; the maximum value (1.0f) seems a bit too 'violent' for my game pad.

The rumbler's current status is retrieved with isRumberOn(), which returns the rumblerOn value.

```
public boolean isRumblerOn()
{  return rumblerOn;  }
```

5.9. Other Approaches

GamePadController is designed for a game pad with two analog sticks, a POV hat, 12 buttons, and a rumbler. 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 D-Pad. 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

My GamePadViewer application is shown in Figure 12.

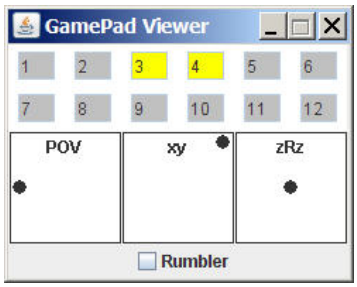

Figure 12. The GamePadViewer Application.

It's meant to emulate the "Function Test" window in the Game Controllers control panel in Windows (see Figure 4). The buttons are displayed as two rows of text fields, with yellow denoting that the numbered button is being pressed. The three boxes in the center are for the POV hat, and the left and right analog sticks. The dot's position in each box indicates the current hat/stick position.

The checkbox at the bottom of the window can be selected/deselected in order to switch the rumbler on/off.

Figure 12 shows the situation when the buttons 3 and 4 are pressed, the left-hand POV hat button is held down, and the left stick is pushed to the top-right. The rumbler is currently inactive.

Figure 13 shows GamePadViewer's class diagrams, with only the public methods visible.

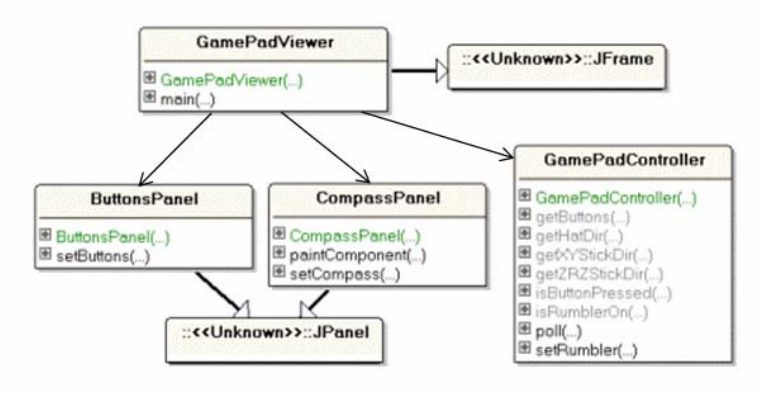

Figure 13. Class Diagrams for GamePadViewer.

The GamePadViewer class is the top-level JFrame, which constructs its GUI from a ButtonsPanel for the buttons, and three instances of CompassPanel for the POV hat and two sticks. The rumbler is managed by GamePadViewer.

GamePadViewer periodically polls GamePadController, and gathers data about the hat, sticks, and buttons. It passes that data to the GUI objects via ButtonsPanel.setButtons() and CompassPanel.setCompass(), which update their GUI elements.

GamePadViewer monitors the rumbler checkbox itself, and calls GamePadController.setRumbler() when necessary.

6.1. Constructing the Application

The GamePadViewer constructor creates the GUI, and initiates polling of the game pad by calling startPolling().

```
// globals
private GamePadController gpController;
private Timer pollTimer;   // timer which triggers the polling


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

  gpController = new GamePadController();

  makeGUI();

  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() uses a timer (pollTimer) to schedule the polling, which needs to be stopped when the application exits.

The GUI is conventional Swing code, so won't be described in detail here. GamePadViewer utilizes the ButtonsPanel class to manage the button text fields, and three instances of CompassPanel for the POV hat and sticks.

```
// globals for the GUI
private CompassPanel xyPanel, zrzPanel, hatPanel;
    // shows the two analog sticks and POV hat
private ButtonsPanel buttonsPanel;
private JCheckBox rumblerCheck;


private void makeGUI()
{
  Container c = getContentPane();
  c.setLayout(new BoxLayout(c, BoxLayout.Y_AXIS));
                  // vertical box layout
  buttonsPanel = new ButtonsPanel();
  c.add(buttonsPanel);
```



```
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
                    // three CompassPanels in a row
  hatPanel = new CompassPanel("POV");
  p.add(hatPanel);

  xyPanel = new CompassPanel("xy");
  p.add(xyPanel);

  zrzPanel = new CompassPanel("zRz");
  p.add(zrzPanel);

  c.add(p);

  // rumbler checkbox
  rumblerCheck = new JCheckBox("Rumbler");
  rumblerCheck.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent ie)
    {
      if (ie.getStateChange() == ItemEvent.SELECTED)
        gpController.setRumbler(true);  // switch on
      else  // deselected
        gpController.setRumbler(false);  // switch off
    }
  });
  c.add(rumblerCheck);
}  // end of makeGUI()
```

Selecting the rumbler checkbox triggers calls to GamePadController.setRumbler().

6.2. Polling the Game Pad

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 = 40;   // ms (polling interval)
      // needs to be fast to catch fast button pressing!

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

      // update the GUI:
      // get POV hat compass direction
      int compassDir = gpController.getHatDir();
      hatPanel.setCompass( compassDir );

      // get compass directions for the two analog sticks
      compassDir = gpController.getXYStickDir();
      xyPanel.setCompass( compassDir );

      compassDir = gpController.getZRZStickDir();
      zrzPanel.setCompass( compassDir );
```



```
      // get button settings
      boolean[] buttons = gpController.getButtons();
      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.

A CompassPanel.setCompass() call stores the compass direction in the CompassPanel object, and triggers a repaint. The black dot's position in the panel 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 text fields accordingly: yellow means "on", gray is "off".

7. Alternatives to JInput

JXInput (http://www.hardcode.de/jxinput/) is a cross-platform API for using the mouse, keyboard, and other input devices. However, non-standard devices, such as game pads, are only supported on Windows because of JXInput's use of DirectInput.

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 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.

