NUI Chapter 15. The Wiimote, the PC, and Java

The wiimote (or more formally, the Wii Remote) is best known as a controller for Nintendo's Wii console, but I'll be using it as an IO device for a Window's PC, and access its features via Java. Here's a brief list of topics:

- button detection (e.g. presses, releases, and held buttons);
- motion sensing, which supplies the wiimote's orientation and acceleration along three axes. I'll also convert the accelerometer information into basic flick gestures;
- IR tracking to locate the wiimote in 3D. An important element of successfully utilizing IR is calibrating the software;
- the Nunchuk attachment: its joystick, motion sensors, and buttons;
- a full-screen Swing application that employs the Wiimote as the input device.

I'll use the wiiuseJ library (http://code.google.com/p/wiiusej/) to implement these examples, and the commercial BlueSoleil Bluetooth stack (http://www.bluesoleil.com/) running on Windows 7 (both 32- and 64-bit) to connect to the wiimote.

1. The Wii Remote and Associated Hardware

Figure 1 shows the wiimote's buttons, arrow keys and LEDs, but its real novelty is its 3-axis accelerometer, an infrared (IR) optical sensor on its front, a rumbler for vibrating the device, a built-in speaker, and expansion plug. It utilizes Bluetooth to communicate with the Wii console (no cables required), and I'll use the same communication approach to link it to the PC.

Figure 1. The Wiimote.

Figure 2 shows the top and bottom of the internals of a wiimote.

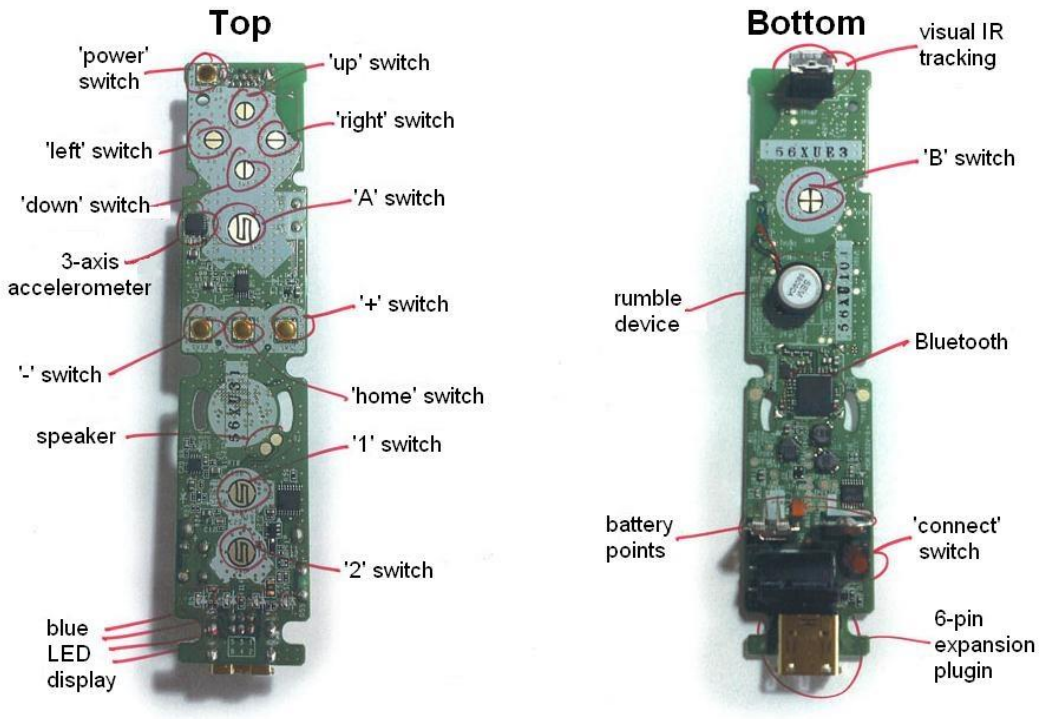

Figure 2. The Wiimote's Internals.

The expansion plug allows a range of different devices to be attached, turning the wiimote into a 'gun' (the Wii Zapper), a 'steering wheel' (the Wii Wheel), a balance board, and many others (gloves, swords, a snooker cue). Probably, the most common expansion device is the Nunchuk (shown in Figure 3).

Figure 3. The Nunchuk.

The Nunchuk comes with its own accelerometer, joystick, and two buttons.

The wiimote's IR sensor can be utilized with any source of infrared, but Nintendo recommends the Wii Sensor Bar (Figure 4).

Figure 4. The Sensor Bar.

The bar is about 20 cm long and features ten infrared LEDs, five at each end. The LEDs at the two ends are pointed slightly outwards, and the ones closest to the center are aimed slightly inwards. The IR light is invisible to the human eye, but can be seen in camera images.

A sensor bar like the one in Figure 4 allows the 3D position of the wiimote to be calculated, which turns out to be extremely useful, and the basis of many of the famous wiimote hacks. For example, Johnny Chung Lee's demos employ IR tracking to implement head and finger tracking, and a multipoint interactive whiteboard (as explained at http://johnnylee.net/projects/wii/).

The wiimote was praised at its release in 2006, but users began to notice that its functionality was sometimes unpredictable, especially in applications utilizing its accelerometer and IR tracking. These issues could be programmed around by using software calibration, and techniques such as data smoothing and weighting, and a slew of different APIs appeared for reading and manipulated the wiimote. A comprehensive list can be found at the WiiBrew website, at http://wiibrew.org/wiki/Wiimote_Driver.

Aside from software solutions, Nintendo released the Wii MotionPlus expansion device (Figure 5) in the middle of 2009.

Figure 5. The Wii MotionPlus attached to the base of the wiimote.

The MotionPlus contains a gyroscope for supplying more accurate rotational information. This can be used by programmers to augment (i.e. improve) the accelerometer and IR position data generated by the wiimote. The usefulness of this

extra hardware led eventually (at the end of 2010) to its incorporation inside the wiimote, now renamed as the *Wii Remote **Plus***, which is shown in Figure 6.

Figure 6. The Wii Remote **Plus**.

The only obvious external difference between the original wiimote (Figure 1) and the Wii Remote Plus are the words "Wii MotionPlus INSIDE" printed on the device below the "Wii" label, as in Figure 7.

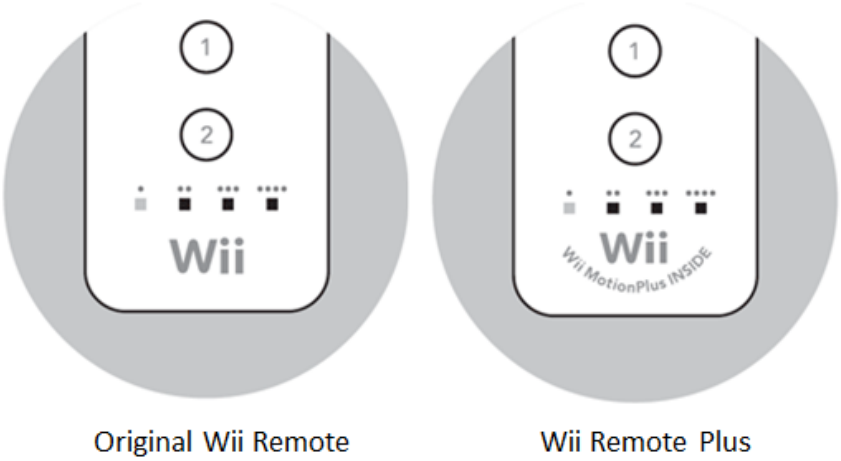

Figure 7. Which Wiimote is it?

In addition, the original wiimote is model number RVL-003, but the Wii Remote Plus is RVL-036, which is printed on the end of the device (see Figure 8) and inside the battery pack area.

Figure 8. The Wiimote's Model Number (RVL-003).

Internally there were so many changes between the wiimote and the Wiimote Remote Plus, that most programming libraries (including wiiuseJ) don't work with the Remote Plus. This incompatibility has a good and a bad side. Fortunately, original wiimotes can be obtained cheaply: I bought one on eBay for about US $15, with a Nunchuk included in the deal. I must also mention that the wiimote is a common counterfeit item on eBay. Always buy from a seller with a good reputation, and closely study the wording used in the advert and any pictures of the device. If the seller uses the phrase "Wii *compatible*", then the device is almost certainly a copy. Make sure that the pictures show the name "Wii' on the front of the device, and Nintendo on the back (as in Figure 9).

Figure 9. A Wiimote (probably) made by Nintendo.

You don't need to purchase a Wii MotionPlus (Figure 5), the gyroscope expansion device. Most software libraries don't support it, or contain warnings about their support being not fully tested. This means that when you're wiimote buying on eBay, if you encounter the word "Plus", either in "Wii MotionPlus" or in "Wii Remote Plus", that means "**don't buy**". The original wiimote (full name: "Wii Remote") doesn't use a "Plus".

There are two great sources of information on the wiimote hardware. One is the overview offered by Wikipedia at http://en.wikipedia.org/wiki/Wii_Remote, which links to many other pages related to the Wii console, games, and extensions. A more detailed hardware overview can be found at the WiiBrew site (http://wiibrew.org/wiki/Wiimote). WiiBrew has separate pages for wiimote expansion gear, such as the Nunchuk (http://wiibrew.org/wiki/Wiimote/Extension_Controllers/Nunchuk) and the MotionPlus (http://wiibrew.org/wiki/Wiimote/Extension_Controllers/Wii_Motion_Plus).

2. The Wiimote and the PC

Before I start programming, the wiimote must be connected to the PC, which turns out to be a tricky proposition. The various elements involved are shown in Figure 10.

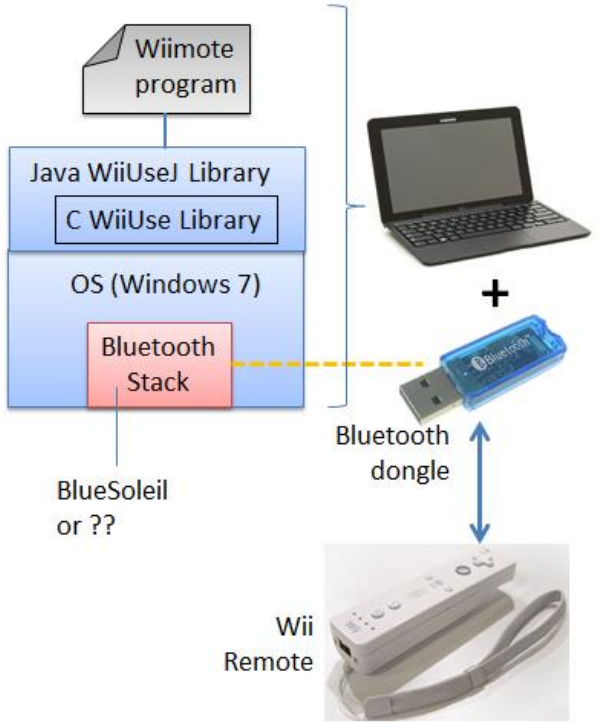

Figure 10. Linking a Wiimote to a PC and Java.

Windows 7 comes with a Bluetooth stack, but on both my test machines it only partially detected the wiimote, treating it as a mouse-like HID device. This meant that button presses could be detected, but not its motion sensing or IR tracking.

A good discussion of the various Windows Bluetooth stacks suited for the wiimote can be found at the Wiimote Project site (http://www.wiimoteproject.com//bluetooth-and-connectivity-knowledge-center/a-summary-of-windows-bluetooth-stacks-and-their-connection).

My solution involves a standard USB Bluetooth dongle and the BlueSoleil stack (http://www.bluesoleil.com/). BlueSoleil is a commercial product, with the current

version 10 selling for around US$30, but it's reliable, with great documentation, and seems to handle a much wider range of 'non-standard' Bluetooth devices than the stack that comes with Windows 7. Incidentally, I'm using version 6.2 of BlueSoleil, which deals fine with the wiimote.

Aside from installing BlueSoleil, I also disabled Window's Bluetooth device installer software (bth.inf and bth.pnf in C:\Windows\inf\) by renaming (I added "BAK" to the extensions). I rebooted the PC, and **only then** plugged in the Bluetooth dongle, which duly became a device managed by BlueSoleil.

The Wiimote Project webpage mentioned above, and several blogs around the Web, claim success with other Bluetooth stacks, such as those for Dell and Toshiba or the Widcomm stack. I've no experience of using those, so can't comment on their capabilities.

The wiimote must be made 'discoverable' to the Bluetooth dongle, which can be done most easily by holding down the wiimote's "1" and "2" buttons simultaneously. Keep pressing those button during the following steps. Now turn to the PC, and access BlueSoleil's Classic View via its tray icon. Right click on the orange 'sun' in the center of the window, and select "Search Devices". A 'constellation' of discovered devices will appear, like those in Figure 11.

Figure 11. Search Results in BlueSoleil Classic View.

The joystick icon indicates a HID device, which is probably the wiimote. Right click on the icon, and select "Get Device Name", and the hexadecimal string label will be replaced by the name "Nintendo RVL-CNT-01", which you can confirm by right clicking on the icon and selecting "Properties".

The wiimote has been discovered, but there's no communication link between the wiimote and PC yet. Bluetooth *pairing* isn't required for the wiimote (so ignore the "Pair" menu item when right clicking on the joystick icon. Instead, press the "Search Services" item, and after a few seconds a new menu item will appear, "Connect Bluetooth Human Interface Device" (see Figure 12).

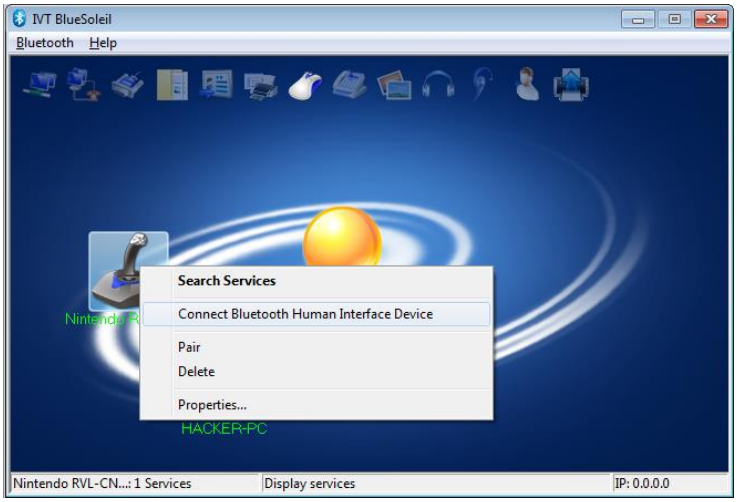

Figure 12. About to Connect the Wiimote to the PC.

Select that "Connect" menu item, and a communications link will be created, indicating by a dotted line with a moving red ball (Figure 13).

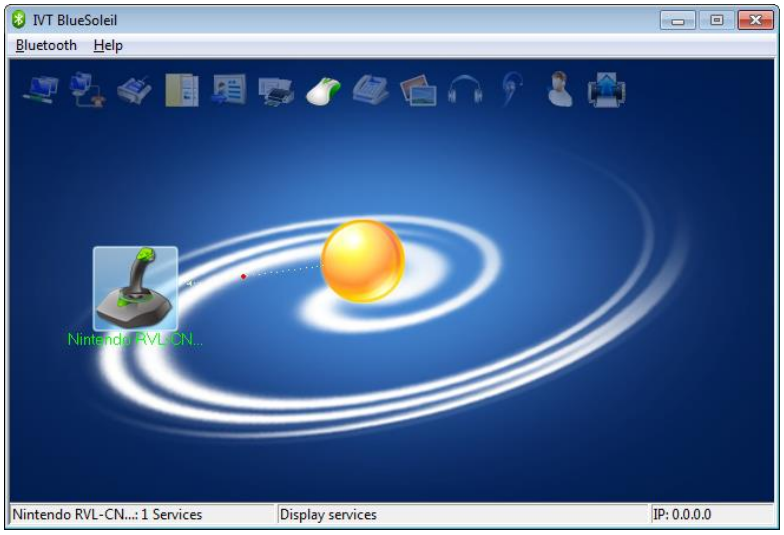

Figure 13. Wiimote and PC are Connected.

At this stage you can stop pressing the "1" and "2" buttons on the wiimote (which you've been holding down since before Figure 11).

The wiimote name displayed by BlueSoleil ("Nintendo RVL-CNT-01") is another way of confirming that you have an original wiimote. If the name is "Nintendo RVL-CNT-01-**TR**" then you've got the Wii Remote Plus, which most wiimote libraries (including wiiuseJ) can not communicate with.

2.1. Testing the Wiimote's Capabilities

Although the wiimote is connected to your PC, there's still the question of whether all of its capabilities are available (e.g. buttons, motion sensing, IR tracking, the rumbler,

sound generator, and the LEDs). A simple way of checking these is to download the WiinRemote tool from http://onakasuita.org/wii/index-e.html; you should select the latest version, which is currently WiinRemote_v2007.1.13.zip. It allows you to test the wiimote without having to write any code. Its GUI is shown in Figure 14.

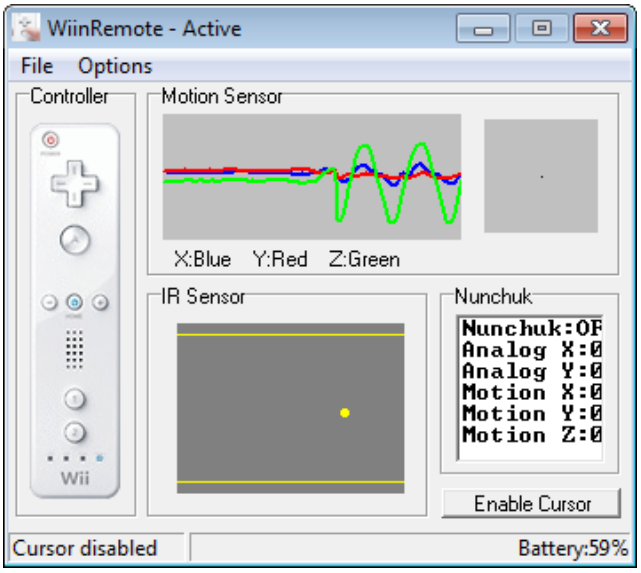

Figure 14. The WiinRemote Tool.

Incidentally, wiinRemote doesn't support the wiimote's sound generator, so there's no way to make it play a tune. That's also a common restriction of most wiimote libraries, including wiiuseJ.

2.2. WiiuseJ

WiiuseJ (http://code.google.com/p/wiiusej/) is a Java API built on top of the popular wiiuse C library, and offers versions for Windows (32- and 64- bit) and Linux. It can access almost all of the wiimote, except for the sound generator. Only three expansion devices are recognized: the Nunchuk, the Classic Controller, and the Guitar Hero 3 Controller. Most notably absent is support for the Motion Plus attachment (the gyroscope).

WiiuseJ employs version 0.12 of the wiiuse C library, which is maintained at http://sourceforge.net/projects/wiiuse/. There's no need to download wiiuse since wiiuseJ already contains copies of its libraries compiled for different versions of Windows and Linux. The original developer of wiiuse, Michael Laforest, stopped working on it at version 0.12 in 2008, but several people created forks and have carried on its development, including Ryan Pavlik at https://github.com/rpavlik/wiiuse. The current version contains support for the Mac, and extensions such as the Motion Plus and the Wii Balance Board. Unfortunately, wiiuseJ hasn't 'followed' the fork and made these features accessible from Java.

The wiiuseJ website includes API documentation and a small example. However, the best overview of the library is a hard-to-find web page explaining the internals of the wiiuse C library. Although it describes the data structures and events employed by

wiiuse, almost all of it is useful for understanding wiiuseJ. The page is available through the Wayback Machine at http://web.archive.org/web/*/http://wiiuse.net/, but I've also saved a PDF copy at http://fivedots.coe.psu.ac.th/~ad/jg/nui15a/.

The main difference between wiiuseJ and wiiuse (aside from the obvious one of language) is that wiiuse uses non-blocking polling to check a wiimote for data. while wiiuseJ is built around a WiimoteListener interface. The programmer implements the listener, coding methods for the events that he wants to process. I'll explain the details in the examples in the rest of this chapter.

WiiuseJ comes as a JAR file (wiiusej.jar), which implements the API *and* contains three test programs. One of them is a GUI application, similar to the wiinRemote tool, and is shown in Figure 15.

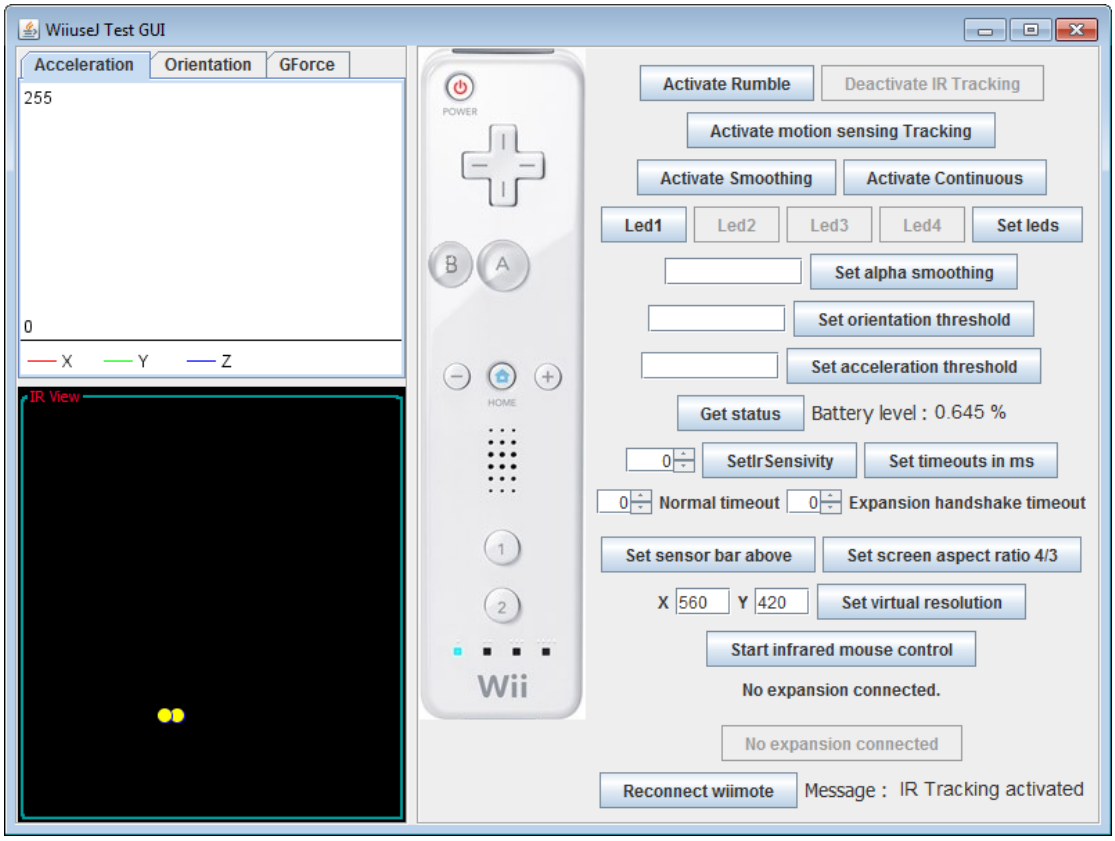

Figure 15. The WiiuseJ GUI Test Program.

If a Nunchuk is connected to the wiimote, then a Nunchuk tab appears in the GUI, which when selected displays its joystick, accelerometer, and button details (see Figure 16).

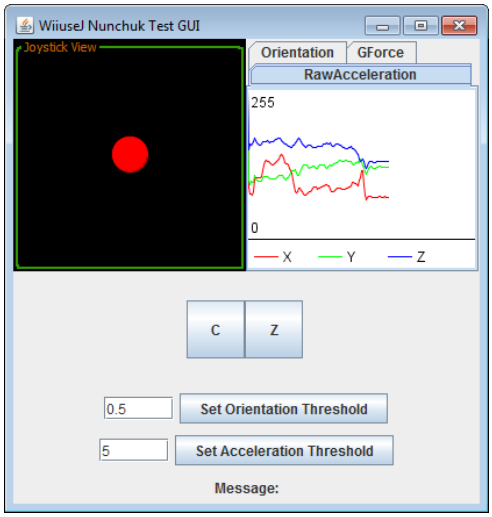

Figure 16. Testing the Nunchuk with WiiuseJ.

The readme file included with my code examples explains how to start this GUI application.

The source code can be downloaded from the WiiuseJ website, as part of the source for the wiiuseJ JAR. It contains several JPanel subclasses for displaying the IR, orientation, and acceleration information (in "WiiuseJ 0.12b src\Java\src\wiiusej\utils\"), which can be useful if you want to graphically display wiimote data in your code.

The JAR also includes a non-GUI test program that writes to standard output, and an application for testing the Classic Controller expansion device.

3. Overview of my Examples

In the rest of this chapter, I'll go through examples that show how to use different features of the wiimote (and the Nunchuk). I'll start by briefly listing each one:

- **WiiSimple.java**: a basic example of how to use the WiimoteListener interface. Most of the other examples are based on this one.
- **Buttons.java**: demonstrates wiimote button usage, and how to activate the rumbler and LEDs.
- **Motion.java**: shows how to access motion information (the wiimote's current orientation and acceleration along the x-, y-, and z- axes). It also implements flick detection.
- **Nunchuk.java**: captures Nunchuk information (i.e. its button states, joystick, and motion).
- **IRTracker.java**: demonstrates the wiimote's IR tracking. You'll need a sensor bar for this and the next example.
- **WiiPosition.java**: shows how to calibrate the wiimote's viewing range for the sensor bar. The code is embedded in a Swing application to illustrate how the wiimote can be used as an input device for a GUI program.

4. The Wiimote Simply

All my examples subclass wiiuseJ's WiimoteListener interface. The following WiiSimple class illustrates the coding approach:

```
public class WiiSimple implements WiimoteListener
{

  static {   // attempt to load the wiiuse library (wiiuse.dll)
    try {
      System.loadLibrary("wiiuse");
    }
    catch (UnsatisfiedLinkError e) {
      System.err.println("wiiuse library failed to load\n" + e);
      System.exit(1);
    }
  }

  private Wiimote wiimote;

  private boolean showStatus = true;
                  // flag for showing status info only once
  private int eventCount = 0;
                  // for labeling event print-outs


  public WiiSimple()
  {
    // look for a Bluetooth connected wiimote
    Wiimote[] wiimotes = WiiUseApiManager.getWiimotes(1, false);
            // "false" means do not trigger a rumble on connection
    if ((wiimotes == null) || (wiimotes.length == 0)){
      System.out.println("No wiimote found");
      return;
    }
    else
      System.out.println("No. of wiimotes found: " +
                                          wiimotes.length);

    wiimote = wiimotes[0];
    wiimote.setTimeout((short)20, (short)20);
                          // default of 10 causes packet timeouts

    // wiimote.activateMotionSensing();
                          // uncomment this for motion sensing
    // wiimote.activateMotionSensing();
                          // uncomment this for IR tracking

    wiimote.addWiiMoteEventListeners(this);
    wiimote.getStatus();    // trigger a status event
  }  // end of WiiSimple()


  public void onStatusEvent(StatusEvent e)
  { if (showStatus) {    // show status event once
      System.out.println("\n" + e);
      showStatus = false;
    }
  }  // end of onStatusEvent
```



```
  public void onButtonsEvent(WiimoteButtonsEvent e)
  {  // processes button events, such as...
    System.out.println("Button event (" + (eventCount++) +
                                                 "): " + e);
    // exit using an "A" button press
    if (e.isButtonAPressed()) {
      WiiUseApiManager.shutdown();
      System.out.println("\nWiimote (" + e.getWiimoteId() +
                                                 ") shutdown");
      System.exit(1);
    }
  }  // end of onButtonsEvent()


  public void onDisconnectionEvent(DisconnectionEvent e)
  {  System.out.println("\nDisconnection): " + e);
     /* should be activated when wiimote is disconnected,
        but isn't */
  }


  public void onMotionSensingEvent(MotionSensingEvent e)
  {  /* process motion events (if motion sensing is activated */ }

  public void onIrEvent(IREvent e)
  {  /* process IR events (if IR tracking sensing is activated */ }

  public void onExpansionEvent(ExpansionEvent e)
  {  /* called if expansion plugin is being used */ }

  // insertion/removal events for three expansion devices

  public void onClassicControllerInsertedEvent(
                                ClassicControllerInsertedEvent e) {}
  public void onClassicControllerRemovedEvent(
                                ClassicControllerRemovedEvent e) {}

  public void onGuitarHeroInsertedEvent(GuitarHeroInsertedEvent e) {}
  public void onGuitarHeroRemovedEvent(GuitarHeroRemovedEvent e) {}

  public void onNunchukInsertedEvent(NunchukInsertedEvent e) {}
  public void onNunchukRemovedEvent(NunchukRemovedEvent e) {}

  // ----------------
  public static void main(String[] args)
  {  new WiiSimple();  }

}  // end of WiiSimple class
```

The class starts with a static block that attempts to load the wiiuse library (wiiuse.dll on Windows) utilized by the wiiuseJ JAR.

The WiiSimple constructor begins by looking for a Bluetooth connected wiimote. WiiuseJ can handle multiple wiimotes at once, but I've not had the opportunity to test that feature.

The wiimote automatically delivers button events, but if motion sensing or IR tracking is required then their event handling must be explicitly activated. The code for doing that is commented out in WiiSimple().

WiiSimple implements the WiimoteListener interface, which contains 12 abstract event-handling methods:

```
public abstract void onStatusEvent(StatusEvent e);
public abstract void onDisconnectionEvent(DisconnectionEvent e);

public abstract void onButtonsEvent(WiimoteButtonsEvent e);
public abstract void onMotionSensingEvent(MotionSensingEvent e);
public abstract void onIrEvent(IREvent e);

public abstract void onExpansionEvent(ExpansionEvent e);

public abstract void onNunchukInsertedEvent(NunchukInsertedEvent e);
public abstract void onNunchukRemovedEvent(NunchukRemovedEvent e);

public abstract void onGuitarHeroInsertedEvent(
                              GuitarHeroInsertedEvent e);
public abstract void onGuitarHeroRemovedEvent(
                              GuitarHeroRemovedEvent e);

public abstract void onClassicControllerInsertedEvent(
                              ClassicControllerInsertedEvent e);
public abstract void onClassicControllerRemovedEvent(
                              ClassicControllerRemovedEvent e);
```

WiiSimple uses onStatusEvent(), onButtonsEvent(), and onDisconnectionEvent(), but the other methods have empty bodies because WiiSimple isn't listening for motion or IR events, and isn't utilizing the wiimote's expansion plugin. Later examples will show how those features are employed.

A status event is triggered by calling Wiimote.getStatus() at the end of the constructor. This prints information like the following:

```
/*********** STATUS EVENT : WIIMOTE ID :1 ********/
--- connected : true
--- Battery level : 0.58
--- Leds : 1
--- Speaker enabled : false
--- Attachment ? : 0
--- Rumble ? : false
--- Continuous ? : false
--- IR active ? : false
--- Motion sensing active ? : false
```

It shows that the wiimote has been correctly initialized (e.g. IR and motion sensing are inactive). The setting of the showStatus boolean to false disables further printing by onStatusEvent().

4.1. Polling Timeouts

Although wiiuseJ implements a listener interface, the underlying wiiuse C library utilizes non-blocking polling – a wiiuse programmer is expected to write a loop which periodically calls a wiiuse_poll() function to read a new event sent by the connected wiimotes. The code might look something like:

```
/* C code: not Java */
while(1) {
  if wiiuse_poll(wiimotes, 1)) {
    /* poll the wiimote (assuming there's only 1) */
    switch(wiimotes[0]->event) {
      /* process the event */
    }
  }
}
```

The default polling frequency is every 10 ms for wiiuseJ, which seems to be too fast since many 'packet timeouts' error messages are reported. This problem is greatly reduced by increasing the timeout period to 20 ms by calling Wiimote.setTimeout() in WiiSimple().

You may be wondering why Wiimote.setTimeout() has two 20 ms arguments? The first is the timeout for wiimote polling (shown above), and the second is for polling expansion units connected to the wiimote.

4.2. Reporting a Button Event

The onButtonsEvent() method is called whenever the state of a button changes, which occurs when one is pressed, released, or held down. For example, when I press the wiimote's "1" button, the contents of the WiimoteButtonsEvent object are printed:

```
Button (0): /******** Buttons for Wiimote generic Event ********/
/******** Buttons ********/
--- Buttons just pressed : 2
--- Buttons just released : 0
--- Buttons held : 0
```

Nothing more is output until the buttons state changes again, which occurs when I release the "1" button, and there's a new call to onButtonsEvent():

```
Button (1): /******** Buttons for Wiimote generic Event ********/
/******** Buttons ********/
--- Buttons just pressed : 0
--- Buttons just released : 2
--- Buttons held : 0
```

The number 2 in the two reports is the WiimoteButtonsEvent.WIIMOTE_BUTTON_ONE constant, representing the "1" button. Unfortunately wiiuseJ's API documentation doesn't explicitly state the values for these button constants, for which you'll need to examine the WiimoteButtonsEvent source code. Table 1 lists their hexadecimal values.

WiimoteButtonEvent Name	Value
WIIMOTE_BUTTON_TWO	0x0001
WIIMOTE_BUTTON_ONE	0x0002
WIIMOTE_BUTTON_B	0x0004
WIIMOTE_BUTTON_A	0x0008
WIIMOTE_BUTTON_MINUS	0x0010
WIIMOTE_BUTTON_ZACCEL_BIT6	0x0020
WIIMOTE_BUTTON_ZACCEL_BIT7	0x0040
WIIMOTE_BUTTON_HOME	0x0080
WIIMOTE_BUTTON_LEFT	0x0100
WIIMOTE_BUTTON_RIGHT	0x0200
WIIMOTE_BUTTON_DOWN	0x0400
WIIMOTE_BUTTON_UP	0x0800
WIIMOTE_BUTTON_PLUS	0x1000
WIIMOTE_BUTTON_ZACCEL_BIT4	0x2000
WIIMOTE_BUTTON_ZACCEL_BIT5	0x4000
WIIMOTE_BUTTON_UNKNOWN	0x8000
WIIMOTE_BUTTON_ALL	0x1F9F

Table 1. Button Constants and their Values.

Figure 17 shows a wiimote with its buttons labeled.

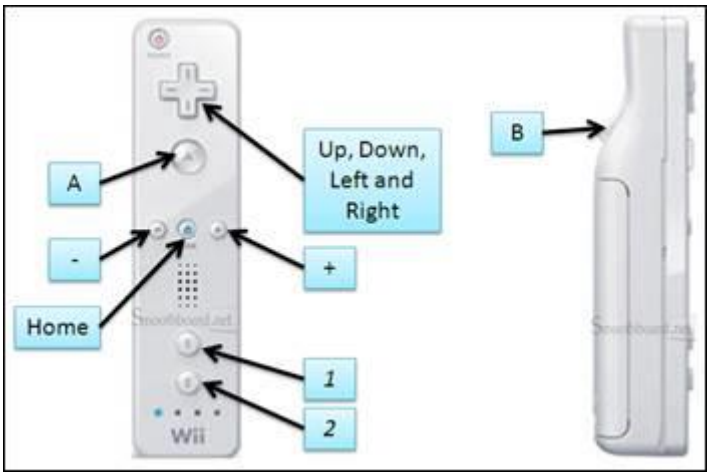

Figure 17. Wiimote Button Names.

A comparison of Figure 17 with Table 1 raises the question of what the four ZACCEL constant are for? I've no idea, and can't find anywhere that these values are used in wiiuseJ.

The hexadecimal values were chosen so they could be combined, which occurs if more than one button is used at once. For example, if I press the "1" and "2" buttons together , then onButtonsEvent() reports:

```
Button (3): /******** Buttons for Wiimote generic Event ********/
/******** Buttons ********/
--- Buttons just pressed : 3
--- Buttons just released : 0
--- Buttons held : 0
```

The 3 combines the button constant values for the "1" and "2" buttons.

The onButtonsEvent() method in WiiSimple.java also calls WiimoteButtonsEvent.isButtonAPressed() which returns true if the wiimote's "A" button is pressed. This test is used to close the wiimote link, which should trigger a call to the onDisconnectionEvent() method, but is never called in my tests. According to the wiiuse documentation, a disconnection event can occur in three ways: when the Bluetooth link is lost, when the wiimote's power button is held down for a short time to switch off the device, or when the battery fails (or is removed from the device). None of these cause onDisconnectionEvent() to be called in wiiuseJ.

5. More Complex Button State Processing

The Buttons.java example illustrates how different button states can be examined, and how the rumbler and LEDs can be activated.

The program is similar to WiiSimple, so I'll only discuss the differences which occur in onButtonsEvent(). The method uses the "A" button to exit the application as before, but examines the buttons states in more detail:

```
public void onButtonsEvent(WiimoteButtonsEvent e)
// in Buttons.java
{
  if (haveButtonsChanged(e)) {
    buttonAction(e);
    lightsAction(e);
    rumbleAction(e);
  }

  // say goodbye using "A"
  if (e.isButtonAPressed()) {
    WiiUseApiManager.shutdown();
    System.out.println("Wiimote shutdown");
    System.exit(1);
  }
}  // end of onButtonsEvent()
```

haveButtonsChanged() checks that there really has been a button change:


```
private boolean haveButtonsChanged(ButtonsEvent e)
{
  return ((e.getButtonsJustPressed() != 0) ||
          (e.getButtonsJustReleased() != 0) ||
          (e.getButtonsHeld() != 0) );
}
```

The wiiuse library supports four different states for each button: "just pressed", "just released", "held" and "currently pressed". This last state can be viewed as a combination of the "just pressed" and "held" states. This separation appears as four state-testing methods in ButtonsEvent: isButtonJustPressed(), isButtonJustReleased(), isButtonHeld(), and isButtonPressed(). I don't need to call e.isButtonPressed() because e.getButtonsJustPressed() and e.getButtonsHeld() already deal with the two possible cases.

Why is a call to haveButtonsChanged() needed at all? Surely onButtonsEvent() is only called when there's a button event, which means that some buttons have changed? This is true when the wiimote is used without expansion devices, but something strange happens when a Nunchuk is plugged in. Even if the wiimote and Nunchuk are left completely alone, a stream of button events is generated. If one of these events is printed it looks like:

```
Button (0): /******** Buttons for Wiimote generic Event ********/
/******** Buttons ********/
--- Buttons just pressed : 0
--- Buttons just released : 0
--- Buttons held : 0
```

Why this occurs is a mystery, but haveButtonsChanged() performs the useful service of filtering out these 'empty' button events.

If haveButtonsChanged () succeeds then buttonAction() tests the button event in detail.

```
private void buttonAction(WiimoteButtonsEvent e)

{
  if (e.isButtonOneJustPressed())
    System.out.println("One pressed");
  if (e.isButtonOneHeld())
    System.out.println("+ One held");
  if (e.isButtonOneJustReleased())
    System.out.println("  One released");

  if (e.isButtonTwoJustPressed())
    System.out.println("Two pressed");
  if (e.isButtonTwoHeld())
    System.out.println("+ Two held");
  if (e.isButtonTwoJustReleased())
    System.out.println("  Two released");

  /* more of the same, for the other 9 wiimote buttons
     shown in Figure 17 */
  //    :
```



```
} // end of buttonAction()
```

Note the WiimoteButtonsEvent type of the buttonAction() argument, which is a subclass of the ButtonsEvent type employed by haveButtonsChanged(). There are four button subtypes in wiiuseJ: for the wiimote, Nunchuk, Classic Controller, and Guitar Hero. I'll use Nunchuk button events in the Nunchuk example later.

Inside buttonAction(), the incoming event is tested to see what button it represents (there are 11 different ones on the wiimote, as can be counted in Figure 17). For each button, there are three if-tests for three of the four possible states of a button ("just pressed", "held", "just released"). I don't bother calling WiimoteButtonsEvent.isButtonOnePressed() because "currently pressed" is already covered by the "just pressed" and "held" states which are tested.

The output generated by buttonAction() when I press and release the "1" button followed by the "2" is:

```
One pressed
  One released
Two pressed
  Two released
```

One problem is the calculation of the "held" state for a button. If I press, *hold*, and release a single button (such as "1"), the "held" state is never detected, only "just pressed" and "just released":

```
One pressed
  One released
```

A "held" state is only identified when a different wiimote event occurs at the same time. For instance, if I press and hold "1", and then press a different button (e.g."2"), then its "just pressed" event will allow an "held" event for "1" to be reported:

```
One pressed
+ One held
Two pressed
```

"+ One held" is only printed when I press another button, or some other event occurs in the wiimote.

There is a "continuous" flag which can be set for the wiimote, by calling:

```
wiimote.activateContinuous();
```

This should cause events to be generated in each polling cycle even when the wiimote's state hasn't changed. This method does not seem to work, although the wiimote status report says that "continuous" is set to true.

As a consequence, I wouldn't suggest the use of "held" button states in wiiuseJ programming, unless there's going to be a steady stream of events from the wiimote. Such a stream is generated when motion sensing and/or IR tracking is switched on, but not when only button states are being monitored.

Activating the Lights and Rumbler

The lightsAction() and rumbleAction() methods in Buttons.java switch on/off the LEDs and rumbler depending on which buttons have just been pressed.

lightsAction() uses the wiimote's arrow keys to decide which LEDs to light up:

```
private void lightsAction(WiimoteButtonsEvent e)
// use the arrow keys to affect the LEDs
{
  if (e.isButtonLeftJustPressed())
    wiimote.setLeds(true, false, false, false);

  if (e.isButtonRightJustPressed())
    wiimote.setLeds(false, false, false, true);

  if (e.isButtonUpJustPressed())
    wiimote.setLeds(false, true, false, false);

  if (e.isButtonDownJustPressed())
    wiimote.setLeds(false, false, true, false);
}  // end of lightsAction()
```

Wiimote.setLeds() allows multiple LEDs to be on (i.e. set to true) at once.

rumbleAction() employs the "+" and "-" buttons to activate and deactivate the rumbler.

```
private void rumbleAction(WiimoteButtonsEvent e)
// use the +/- buttons to switch rumbler on/off
{
  if (e.isButtonPlusJustPressed()) {
    System.out.println("Rumble activated");
    wiimote.activateRumble();
  }

  if (e.isButtonMinusJustPressed()) {
    System.out.println("Rumble deactivated");
    wiimote.deactivateRumble();
  }
}  // end of rumbleAction()
```

For some reason, Wiimote.deactivateRumble() becomes 'buggy' if an arrow key is pressed while the rumbler is on. Subsequently, the "-" button's deactivateRumble() call has no effect until another arrow key is pressed.

6. Motion Sensing

Motion sensing must be enabled explicitly, which is done in the constructor of the Motion.java example:

```
wiimote.activateMotionSensing();
```


This starts sending a stream of MotionSensingEvent objects to onMotionSensingEvent(). Each event has three main parts: the wiimote's current orientation, g-force accelerations, and integer accelerations.

The wiimote's accelerometer is very sensitive, so wiiuseJ offers several methods for adjusting its settings. For example, an orientation change of just 0.5 degrees will trigger a motion event, but this can be changed by calling Wiimote.setOrientationThreshold(). There's a similar method for adjusting the acceleration reporting threshold, called Wiimote.setAccelerationThreshold(). Also, the acceleration values can be smoothed with Wiimote.setAlphaSmoothingValue(). It employs an exponential moving average with a default value of 0.07, which can range between 0 and 1.

I included the following in the constructor for the Motion class:

```
wiimote.setOrientationThreshold(3.0f);    // was 0.5 degrees
wiimote.setAccelerationThreshold(10);
               // range is 0-255; default threshold is 5
wiimote.activateSmoothing();
wiimote.setAlphaSmoothingValue(0.2f);
                // Accelerometer smoothing [0-1]; was 0.07
```

onMotionSensingEvent() reports a deluge of data, of three types: orientations about three axes, g-forces around those axes, and the raw, unsmoothed integer accelerometer data. In practice, if you want to study how a particular value changes then it's a good idea to comment out all the other prints in the following code.

```
public void onMotionSensingEvent(MotionSensingEvent e)
{
  if (showMotionSettings) {    // report only at start-up
    reportMotionSettings(e);
    showMotionSettings = false;
  }

  Orientation ori = e.getOrientation();
  System.out.printf("%d. Pitch(x): %.1f  Roll(y): %.1f
                                        Yaw(z):  %.1f\n",
        eventCount++, ori.getPitch(), ori.getRoll(), ori.getYaw());

  GForce gforce = e.getGforce();
  System.out.printf("     %d. Gravity x: %.2f  y: %.2f  z: %.2f\n",
        eventCount++, gforce.getX(), gforce.getY(), gforce.getZ());

  reportFlicks(gforce);

  RawAcceleration ra = e.getRawAcceleration();
  System.out.println("  " + ra);
}  // end of onMotionSensingEvent()
```

When the application first starts, reportMotionSettings() reports the orientation and acceleration thresholds:

```
private void reportMotionSettings(MotionSensingEvent e)
{
  System.out.println("===== Motion Sensing Settings ======");
```



```
  System.out.println("Orientation threshold: " +
                                  e.getOrientationThreshold());
  System.out.println("Acceleration threshold: " +
                                  e.getAccelerationThreshold());
  if (e.isSmoothingActive())
    System.out.println("Alpha smoothing: " +
                                   e.getAlphaSmoothing());
  else
    System.out.println("Smoothing not active");
  System.out.println();
}  // end of reportMotionSettings()
```

The wiimote orientation information is specified around its x-, y-, and z- axes, as illustrated in Figure 18.

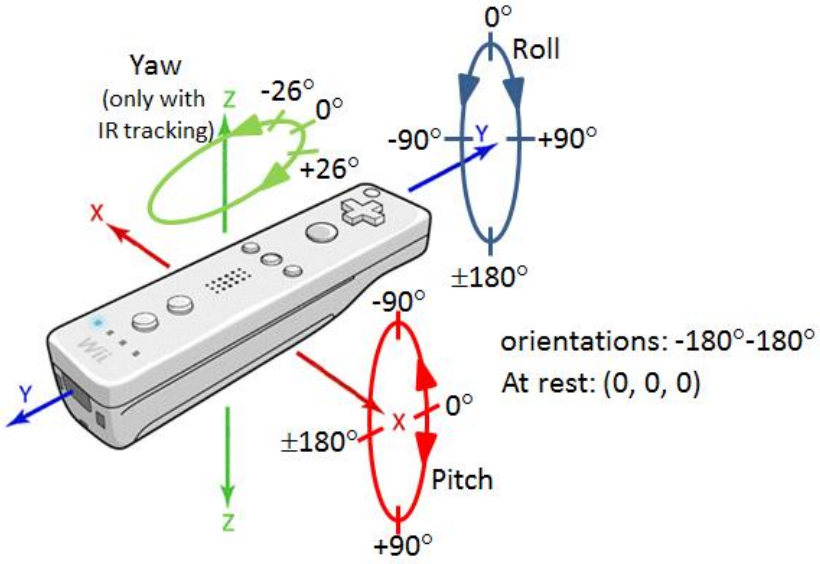

Figure 18. The Orientation Axes of the Wiimote.

The y-axis runs forwards-backwards, with forwards typically pointing towards the PC screen. The rotation (the roll) around the axis can vary between -180 and 180 degrees with the wiimote starting at 0 degrees when lying flat in the user's hand. A roll to the right is positive, a roll to the left is negative.

The z-axis runs upwards-downwards, with its rotation (the yaw) values varying from about -26 to 26 degrees as the user turns the wiimote to the left and right. Yaw is **not** calculated by the accelerometer, but derived from IR tracking data. This means that if you don't have IR tracking activated, and don't have a sensor bar to point the wiimote at, then the yaw will always be 0. I'll discuss IR tracking in a later section, but activating it requires a one line addition to the Motion() constructor:

```
wiimote.activateIRTRacking();
```

The x-axis runs left-to-right, usually parallel to the monitor, and its rotation (the pitch) values can vary between -180 and 180 degrees. If the wiimote is rotated upwards then the angle becomes negative, while aiming the wiimote downwards creates a positive pitch.

The wiimote reports acceleration as gravity forces (g-forces), along each axis, as in Figure 19.

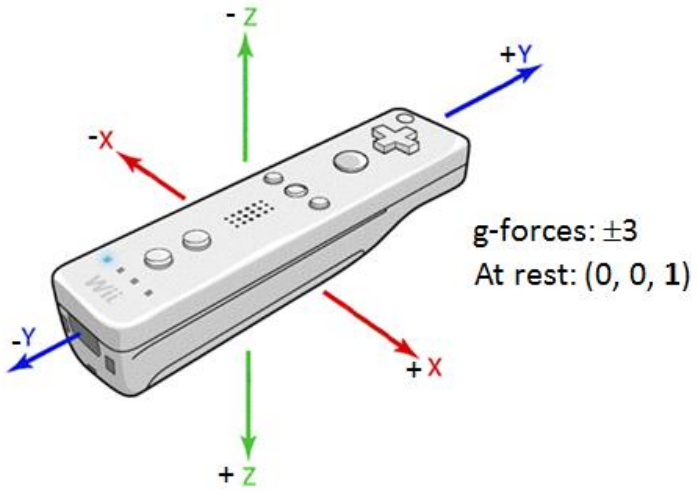

Figure 19. The G-Force Axes of the Wiimote.

The forces can range between -3 and +3. Along the y-axis, a positive g-force is generated when you lunge towards the screen. A positive x-axis g-force occurs when you quickly wave your wiimote to the right. The z-axis g-force is a little different in that even when the wiimote is at rest, it contributes a downwards g-force of 1g, due to gravity. If you quickly lower the wiimote, this value will increase.

There's no need to activate IR tracking in order to generate any of these g-force readings.

It's also possible to read the accelerometer data in an integer form. as in Figure 20.

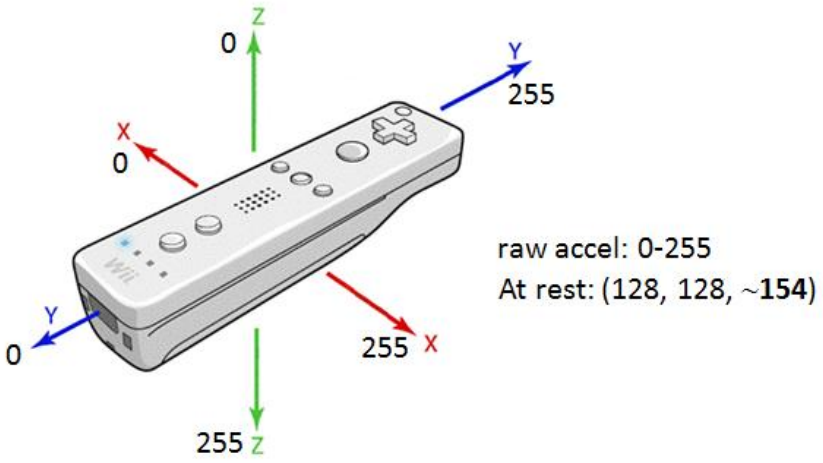

Figure 20. Raw Acceleration Axes of the Wiimote.

The accelerations are returned as integers ranging between 0 and 255. Rather confusingly, 0 does not mean "no acceleration", but a maximum acceleration backwards (along the y-axis), to the left (for the x-axis), and upwards (for the z-axis). No acceleration is represented by the mid-range value, 128. When the wiimote is at

rest, there is a downwards acceleration due to gravity, which my wiimote reports as about 154. This is somewhat surprising since I was expecting that accelerations were linearly mapped to g-forces, which suggests that the value should be around 170.

We've already seen this integer way of specifying acceleration back in the Motion() constructor, when I set the acceleration threshold:

```
wiimote.setAccelerationThreshold(10);
               // range is 0-255; default threshold is 5
```

This thresholding applies both to the integer accelerations and g-forces.

The Basics of Flick Gestures

The g-force information (or integer accelerations) can be easily transformed into gesture information, based on the directions shown in Figure 19. reportFlicks() examines the g-forces along the three axes, and if they exceed hardwired positive or negative thresholds then a direction string is printed.

```
private void reportFlicks(GForce gf)
{
  float xForce = gf.getX();
  if (xForce > 1.5f)
    System.out.println("Right");
  else if (xForce < -1.5f)
    System.out.println("Left");

  float yForce = gf.getY();
  if (yForce > 1.5f)
    System.out.println("  Forward");
  else if (yForce < -1.5f)
    System.out.println("  Back");

  float zForce = gf.getZ();
  if (zForce > 1.8f)
    System.out.println("    Down");
  else if (zForce < -1.0f)
    System.out.println("    Up");
}  // end of reportFlicks()
```

Note that the thresholds for the z-axis g-force are a little different from those for the x- and y- axes, since the z-axis always has a constant 1g downward force due to gravity.

Typical output is:

```
Right
Right
Right
Right
Right
  Forward
Left
Left
Left
Left
Left
```



```
Left
  Forward
  Forward
  Forward
  Forward
  Forward
```

The series of "Left" and "Right" reports occur when the user swings the wiimote to the left and right. The "Forward" messages indicate that the user has jumped towards the screen.

7. The Nunchuk

The Nunchuk (see Figure 3) adds several features to the basic wiimote, including: two buttons (called "C" and "Z"), a joystick, and its own motion sensing information (i.e. its own orientation and accelerometer values along three axes).

From a programming viewpoint, three listener methods come into play for the first time:

- onNunchukInsertedEvent(), called when the Nunchuk is plugged into the wiimote's expansion slot;
- onNunchukRemovedEvent(), triggered when the Nunchuk is removed from the slot;
- onExpansionEvent(), activated whenever an event is sent from an expansion device.

One confusing aspect of Nunchuk programming is that its button and motion sensing data do not arrive via the WiimoteListener's onButtonsEvent() and onMotionSensingEvent() methods but through onExpansionEvent(). Also, it's unnecessary to activate Nunchuk motion sensing by calling:

```
wiimote.activateMotionSensing();   // not needed
```

Nunchuk motion detection is on by default.

The Nunchuk also has its own orientation and acceleration threshold settings:

```
wiimote.setNunchukOrientationThreshold(3.0f);    // was 0.5 degrees
wiimote.setNunchukAccelerationThreshold(10);     // default is 5
```

The complicated part of Nunchuk.java is the coding of onExpansionEvent(). Since every expansion device delivers events to this listener method, it's necessary to do some checking of the event to decide how to process it:

```
public void onExpansionEvent(ExpansionEvent e)
// cast expansion event to a Nunchuk event, then process it
{
  if (e instanceof NunchukEvent)
    processNunchuk((NunchukEvent) e);
  else
    System.out.println("Unknown Expansion Event: " + e);
```



```
}  // end of onExpansionEvent()
```

The available subclasses of ExpansionEvent are NunchukEvent, ClassicControllerEvent, and GuitarHeroEvent.

processNunchuk() examines the NunchukEvent object passed to it, processing the joystick, button, and motion information separately.

```
private void processNunchuk(NunchukEvent ne)
{
  if (ne.isThereNunchukJoystickEvent()) {   // joystick
    JoystickEvent je = ne.getNunchukJoystickEvent();
    float magnitude = je.getMagnitude();
    if (magnitude > 0.1)
      System.out.printf("Joystick angle %.1f; magnitude: %.2f\n",
                                je.getAngle(), magnitude);
  }

  NunchukButtonsEvent nbe = ne.getButtonsEvent();  // button
  if (haveButtonsChanged(nbe))
    nunchukButtons(nbe);

  if (ne.isThereMotionSensingEvent()) {     // motion
    MotionSensingEvent nme =  ne.getNunchukMotionSensingEvent();
    Orientation ori = nme.getOrientation();
    // System.out.printf("%d. Pitch(x): %.1f  Roll(y): %.1f
                                              Yaw(z): %.1f\n",
    //  eventCount++, ori.getPitch(), ori.getRoll(), ori.getYaw());
       // yaw is always 0 for the nunchuk

    GForce gforce = nme.getGforce();
    // System.out.printf("%d. Gravity x: %.2f  y: %.2f  z: %.2f\n",
    //  eventCount++, gforce.getX(), gforce.getY(), gforce.getZ());

    RawAcceleration ra = nme.getRawAcceleration();
    // System.out.println("  " + ra);
  }
}  // end of processNunchuk()
```

The joystick returns angle and magnitude data depending on its position, as shown in Figure 21.

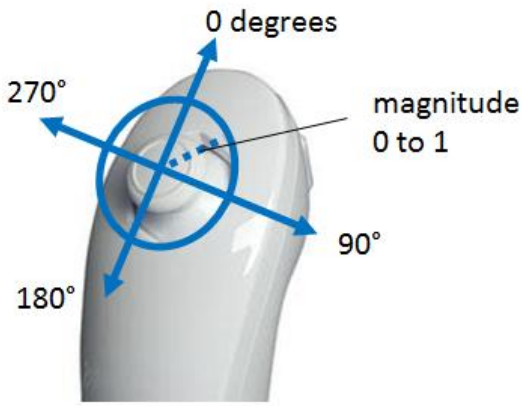

Figure 21. The Nunchuk Joystick.

The angle can range between 0 and 360 degrees, but may also be undefined if the joystick isn't currently being touched. This explains the if-test of the magnitude in processNunchuk() before printing out the data; if the magnitude is close to 0 (0.01 in the code above) then it probably means that the joystick is not being pressed. Also note that all the joystick code is surrounded by a NunchukEvent.isThereNunchukJoystickEvent() call since the Nunchuk event may not contain joystick information at all.

The Nunchuk's "C" and "Z" buttons (which are on its front) are dealt with by nunchukButtons():

```
private void nunchukButtons(NunchukButtonsEvent e)
{
  if (e.isButtonCJustPressed())
    System.out.println("C pressed");
  if (e.isButtonCHeld())
    System.out.println("+ C held");
  if (e.isButtonCJustReleased())
    System.out.println("  C released");

  if (e.isButtonZJustPressed())
    System.out.println("Z pressed");
  if (e.isButtonZeHeld())    // note spelling mistake
    System.out.println("+ Z held");
  if (e.isButtonZJustReleased())
    System.out.println("  Z released");
}  // end of nunchukButtons()
```

The coding style is the same as the button processing in my earlier Buttons.java example. One difference is that "held" events are correctly generated by the Nunchuk. For example, if I press, hold and release the "C" button, then the following is reported:

```
C pressed
+ C held
  C released
```

This improved behavior is probably because the Nunchuk is also generating a nearly continual stream of motion events, allowing "held" states to be correctly calculated.

The motion sensing offered by the Nunchuk is very similar (but not exactly the same) as the wiimote's. processNunchuk() can print the orientation information around the 3 axes, the g-forces, and the integer acceleration. The prints are commented out inside my method simply to reduce the volume of data that's dumped to the screen.

The format of the Nunchuk's orientation data is just like the wiimote's shown in Figure 18 – but replace the wiimote by a Nunchuk pointing at the screen. One difference is that the Nunchuk always reports a yaw of 0 degrees, and switching on IR tracking makes no difference.

The Nunchuk's g-forces are reported as in Figure 22 (which is similar to the wiimote's g-forces diagram in Figure 19).

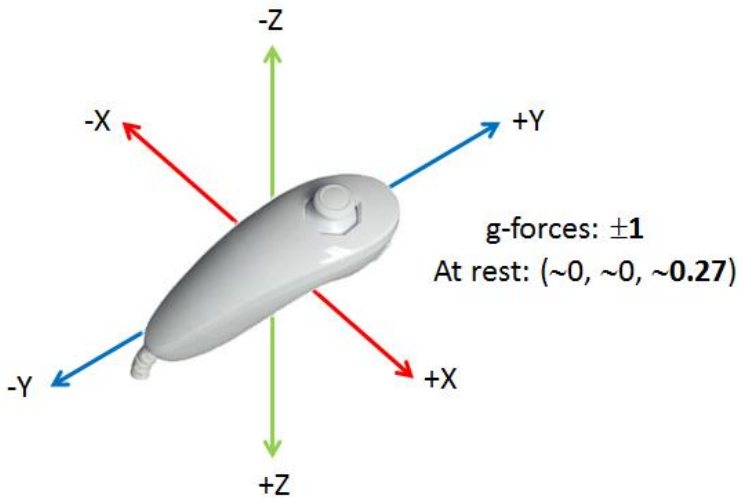

Figure 22. The G-Force Axes of the Nunchuk.

The range goes between -1 and 1 rather than -3 to 3, and the at-rest downward z-axis g-force appears to be about 1/3 of 1g.

The integer accelerations for the Nunchuk are almost identical to the wiimote's accelerations shown in Figure 20. However, the at-rest z-axis acceleration for the Nunchuk is about 170 units, which corresponds to 1g in g-forces.

For those interested in technical details about the Nunchuk, a good overview can be found in the "Nunchuk" section of the wiimote Wikipedia page (http://en.wikipedia.org/wiki/Wii_Remote#Nunchuk), and there's more technical information at WiiBrew (http://wiibrew.org/wiki/Wiimote/Extension_Controllers/Nunchuk).

8. IR Tracking

Wiimote IR tracking requires a sensor bar, such as the one in Figure 4. The 'sensor' part of the name is a bit misleading since the device is really just a source of infrared (IR) light. The imposing USB cable that links the Nintendo Sensor Bar to your PC is only a power source, and you can achieve much the same functionality with a couple of IR LEDs and a battery, or even two candles! Several instruction guides are available online, including at Instrutables.com (http://www.instructables.com/id/Cheapest-and-easiest-wii-sensor-bar/) and WikiHow (http://www.wikihow.com/Make-a-Sensor-Bar).

In retrospect, I wished I'd taken the maker approach, but I bought a sensor bar 'clone' on eBay instead. When I finally got a chance to examine it through a camera, I discovered that it only had 3 LEDs at each end (see Figure 23), and was missing the slightly outward facing and inward facing lights of the Nintendo device. The LEDs also seem a bit dim.

Figure 23. My non-Nintendo Sensor Bar.

The means that my wiimote has some trouble detecting the IR lights in the sensor bar. But this poor functionality can be seen as an opportunity, since reliable IR tracking often requires a calibration phase, even when using a Nintendo Sensor Bar.

By calibration, I mean determining the effective viewing range of the wiimote's IR sensor. For instance, how far can a user stand away from the sensor bar, how close to it, how far to the left and right?

According to the hardware specs at WiiBrew (http://wiibrew.org/wiki/Wiimote), the IR camera has an effective field of view of about 33 degrees horizontally and 23 degrees vertically. This is quite a narrow range when the wiimote is being used by someone sat in front of a large monitor with the sensor bar taped on top. Other interesting physical constraints are that the camera can only track up to four IR sources, and enlarges 128x96 resolution images to 1024x768 before generating point coordinates.

I'll be describing two example programs for IR tracking. The first, IRTracker.java, shows the standard coding approach. The second, WiiPosition.java, implements some calibration techniques on top of lower-level IR tracking.

8.1. Tracking with No Extra Frills

IRTracker.java reports the IR settings at start-up time, and then repeatedly prints the wiimote's calculated IR tracking (x, y, z) position. It also display wiiuseJ's 'absolute' (x, y) coordinate, and the actual data points detected by the wiimote's IR camera. Typical output is:

```
     :
Track: (299, 27, 639);          Abs: (291, 203)
  Corr: (483,194);  Raw: (525,192); Size: 3
  Corr: (99,213);  Raw: (147,125); Size: 4

Track: (299, 27, 639);          Abs: (289, 201)
  Corr: (481,192);  Raw: (523,189); Size: 3
  Corr: (97,210);  Raw: (144,124); Size: 4

Track: (299, 27, 640);          Abs: (287, 198)
  Corr: (479,191);  Raw: (520,188); Size: 3
  Corr: (96,206);  Raw: (142,123); Size: 4

Track: (299, 27, 639);          Abs: (284, 198)
  Corr: (476,191);  Raw: (516,187); Size: 4
  Corr: (92,205);  Raw: (137,123); Size: 4
     :
```


The "Track" coordinate is the calculated tracking position, Abs" stands for absolute, and the original detected points are displayed indented beneath. There may be as many as four detected points, but it's not unusual to only see one or two. The "Corr" value is the "Raw" coordinate after automatic smoothing, and "Size" is an indication of the size of the observed point. According to the documentation, the size can vary between 1 and 4 (4 being the largest), but I've seen 5's and 6's in some test runs.

The IRTracker() constructor turns on IR tracking, and supplies various settings:

```
wiimote.activateIRTRacking();
wiimote.activateMotionSensing();
                // IMPORTANT: improves the accuracy

wiimote.setIrSensitivity(5);
               // maximum; ranges between 1-5; default is 3

wiimote.setSensorBarAboveScreen();

wiimote.setScreenAspectRatio169();
  // does not seem to work; aspect ratio still 4:3 in status report

wiimote.setVirtualResolution(1920, 1080);    // desktop PC
// wiimote.setVirtualResolution(1366, 768);       // laptop
         // do this last, after bar position and aspect ratio
```

It's important to activate motion sensing as well as IR tracking, since the wiimote uses movement to improve the accuracy of the IR points it generates.

There are four adjustable IR settings – sensitivity, sensor bar position, screen aspect ratio, and virtual screen resolution. I increased the sensitivity to the maximum since my sensor bar produces quite dim IR sources, but there's an increased danger that the wiimote may start detecting other heat sources behind the PC.

The wiimote can be told that the sensor bar is positioned above the screen (as I have done), or beneath it, with a call to Wiimote.setSensorBarBelowScreen().

The sensor bar position, aspect ratio, and virtual screen settings apply to the calculated tracking position (the (x, y, z) coordinate labeled with "Track:" in the output shown above). The aspect ratio is 4:3 by default, or can be set to 16:9 as I have done. However, this doesn't seem to have any effect since the printed IR settings (which I'll show you in a minute) report the ratio as 4:3.

8.2. Reporting IR Events

With IR tracking activated, each IR event will trigger a call to onIrEvent (). That method's job is to print the information shown in the output on the previous page.

```
public void onIrEvent(IREvent e)
{
  if (showIRSettings) {
    reportIRSettings(e);    // show only at start-up
    showIRSettings = false;
  }

  if ((e.getX() != 0) || (e.getY() != 0)) {
```



```
    // don't print (0, 0, ??) to reduce the output
    System.out.printf("Track: (%d, %d, %.0f);  \t",
                          e.getX(), e.getY(), e.getZ());
    System.out.print("Abs: (" + e.getAx() + ", " + e.getAy() +")");
    System.out.println();

    IRSource[] irPts = e.getIRPoints();
    reportPts(irPts);
  }
}  // end of onIrEvent()
```

reportIRSettings() reports information about the IR settings:

```
private void reportIRSettings(IREvent e)
{
  System.out.println("============ IR Settings ================");
  if (e.isScreenAspectRatio169())
    System.out.println("Screen aspect ratio: 16/9");
  else if (e.isScreenAspectRatio43())
    System.out.println("Screen aspect ratio: 4/3");

  System.out.println("Virtual screen res: " + e.getXVRes() +
                                        "x" + e.getYVRes());

  // is sensor bar above/below the TV/monitor?
  if (e.isSensorBarAbove())
    System.out.println("Sensor bar position: above");
  else if (e.isSensorBarBelow())
    System.out.println("Sensor bar position: below");

  System.out.println("IR (X,Y) correction offsets: (" +
               e.getXOffset() + ", " + e.getYOffset() + ")");
  System.out.println("IR camera sensitivity: " +
               e.getIrSensitivity());
  System.out.println();
}  // end of reportIRSettings()
```

The output on my test machine is:

```
Screen aspect ratio: 4/3
Virtual screen res: 1919x1079
Sensor bar position: above
IR (X,Y) correction offsets: (0, 110)
IR camera sensitivity: 5
```

The virtual screen size matches the dimensions of my monitor, but the aspect ratio has not changed to 16:9. The correction offset is based on Wiimote. setSensorBarAboveScreen(), although it's unclear how the actual offset numbers are determined. For instance, I'd expect there to be an x-axis offset since most user's will position their sensor bar mid-way along the top-edge of the monitor.

Returning to onIrEvent(), there's a few quirks that need explaining.

The output is quite substantial, so it's reduced a little by discarding tracking points at (0, 0); this is almost always an indication that no point could be calculated.

The tracking point includes a z-axis coordinate, which appears to be based on the size integer of the detected IR points. Its accuracy depends greatly on how many points were detected, and represents the distance of the wiimote from the bar, measured in millimeters.

The x- and y- tracking point values are positioned relative to the user-specified virtual screen rectangle (1920, 1080 for my PC), with (0, 0) at the top-left. However, it's possible for the tracking point to be negative, or bigger than the screen size.

The absolute point (the (x, y) coordinate marked with "Abs" in the output shown earlier) is not very well documented. There's nothing equivalent to it in the wiiuse library documentation, and so I've had to make a guess about its role. It appears to be an average of the detected IR points, which isn't scaled to the virtual screen size. Also, when no points are detected, the absolute coordinate retains its last value, whereas the calculated tracking position is set to (0, 0, 0).

reportPts() reports the points detected by the wiimote's IR camera. Each point is located relative to a fixed virtual screen size of 1024 x 768, with (0,0) at the top-left. However, I've seen output where the points are negative and/or bigger than the screen size. The camera can record details about a maximum of four IR sources, but may also detect nothing. That means that IREvent.getIRPoints() may return a 0-element array or null, which needs to be handled by reportPts().

```
private void reportPts(IRSource[] irPts)
{
  if ((irPts != null) && (irPts.length > 0)) {
    for(IRSource ir : irPts) {
      System.out.println( "  Corr: (" + ir.getX() + "," + ir.getY() +
                          ");  Raw: (" + ir.getRx() + "," + ir.getRy() +
                          "); Size: " + ir.getSize());
    }
  }
  System.out.println();
}  // end of reportPts()
```

Each detected point is represented in two ways: as a corrected and a raw coordinate. The corrected value is a smoothed versions of the raw data, and it's these that are used to calculate the tracking and absolute coordinates.

Since these detected IR points are available, it's possible to implement your own tracking calculation, which is at the heart of the next IR program.

9. Calibrating IR Tracking

The second IR tracking example, WiiPosition, has two main purposes. The first is to demonstrate several tracking calibration techniques, including:

- calculating a IR tracking point from the detected IR sources in the program rather than relying on the wiimote's algorithm;
- determining the viewing boundaries of the wiimote IR camera relative to the sensor bar;

- displaying a detected width for the bar. With a little bit of math, this can be used to determine the z-axis distance of the wiimote from the screen.

The other aim of WiiPosition is to show how a wiimote can be used as an input device inside a Swing application. WiiPosition is a full-screen application (see Figure 24), using a JPanel as a drawing area. Its state is updated and redrawn inside a threaded loop, utilizing input from the wiimote.

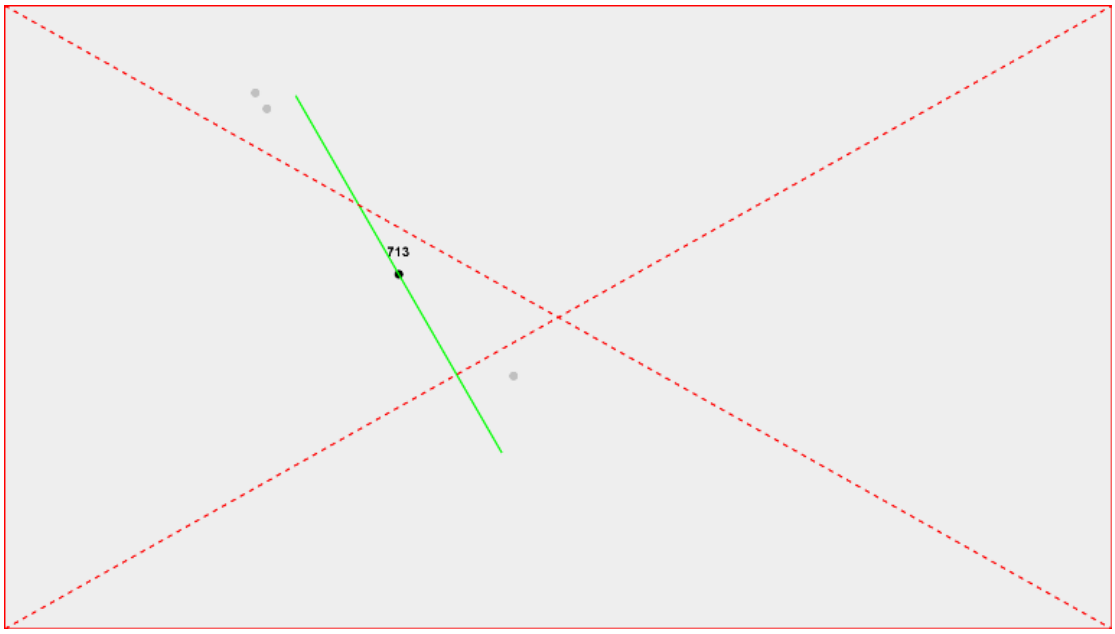

Figure 24. WiiPosition in Action.

The detected IR sources are drawn as gray circles (there are three shown in Figure 24). My code utilizes these points to calculate a tracking position, which is drawn as a black circle. A green line is drawn between the left and right groups of detected IR points to represent the width of the sensor bar, which is also written above the black circle ("713" in Figure 24). The green line is rotated to show the current roll angle of the wiimote around the z-axis In the figure, the user has turned the wiimote about 45 degrees to the right.

There are four solid red lines drawn on the screen to indicate the viewing boundaries beyond which the wiimote cannot see the sensor bar. In Figure 24, all four lines are in their starting positions at the edge of the screen. However, as the user moves the wiimote towards these boundaries, it will eventually lose sight of the sensor bar. At that point, the black tracking dot will change color to red, and no gray detected IR points will be drawn (because none are visible). The user can then use the wiimote arrow keys to move the red boundary lines to reflect the true viewing range of the wiimote.

Figure 25 shows a later stage in WiiPosition's execution after the right and bottom boundaries lines have been moved inwards.

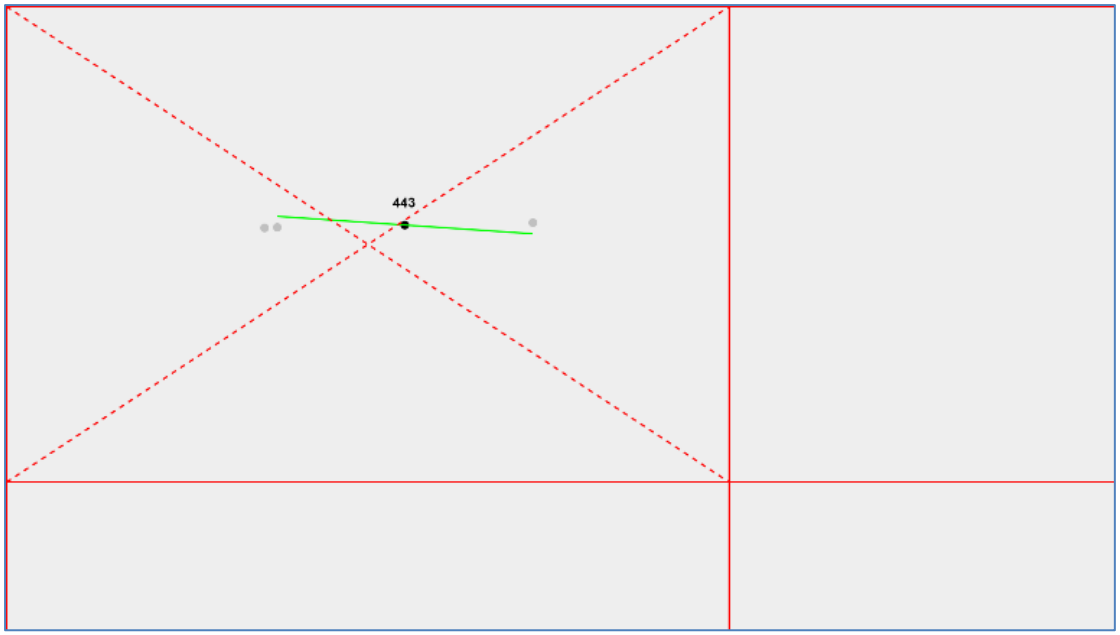

Figure 25. WiiPosition with Adjusted Boundary Lines.

The diagonal dotted lines make it easier to see the effective viewing area for the wiimote. If the user moves the wiimote outside of this region then it will lose sight of the sensor bar.

The green line has also changed between Figure 24 and 25 since the user has rotated the wiimote to the left so it is almost level. In addition, he has moved further away from the screen since the width of the line is reported as "443".

When the user wishes to quit, he presses the "A" button on the wiimote. The program finishes by printing the final tracking location (the black point), the width of the green line, and the boundary ranges:

```
IR current location: (661, 354)
Line width: 443
Sensor Bar Views. x-range: 0--1183, y-range: 0—772
```

9.1. Using the Calibration Data

The boundary ranges show that the wiimote will be unable to move the cursor in a full-screen application to every point on the screen. For example, a user sat in my position in front of the monitor will be unable to move the cursor further right than pixel 1183, and no lower than 772. This can be fixed by scaling the (x, y) data returned by the wiimote. The dimensions of my screen are 1920x1080, and so the x- and y- axes scale factors should be set to 1920/1183 and 1080/772 respectively. This will mean that when the wiimote points at the edges of the viewing region, the cursor will be positioned at the edges of the screen.

The line width (443, above) can be used to calculate z-axis distances from the screen for the wiimote. The simplest way is to require the user to employ a tape measure to find the real-world distance of the wiimote from the screen when the line width is 443 units. I measured this length as 64 cm. There's an inverse relationship between the

distance-to-screen and the line width – as the width increases, the distance from the screen will get smaller. This can be expressed as:

```
distance = k / width
```

where k is some constant. Since I've measured one point on this curve, I can calculate k, and use the equation to calculate other distances when the wiimote supplies other line widths.

Thus, k = 443*64, which means that when the line width is reported as 713 (for example, in Figure 24), then the wiimote is (443 * 64)/713 ≈ 40 cm from the screen.

9.2. The Wiimote and Swing

The WiiPosition class diagrams in Figure 26 give a good overview of the program's structure.

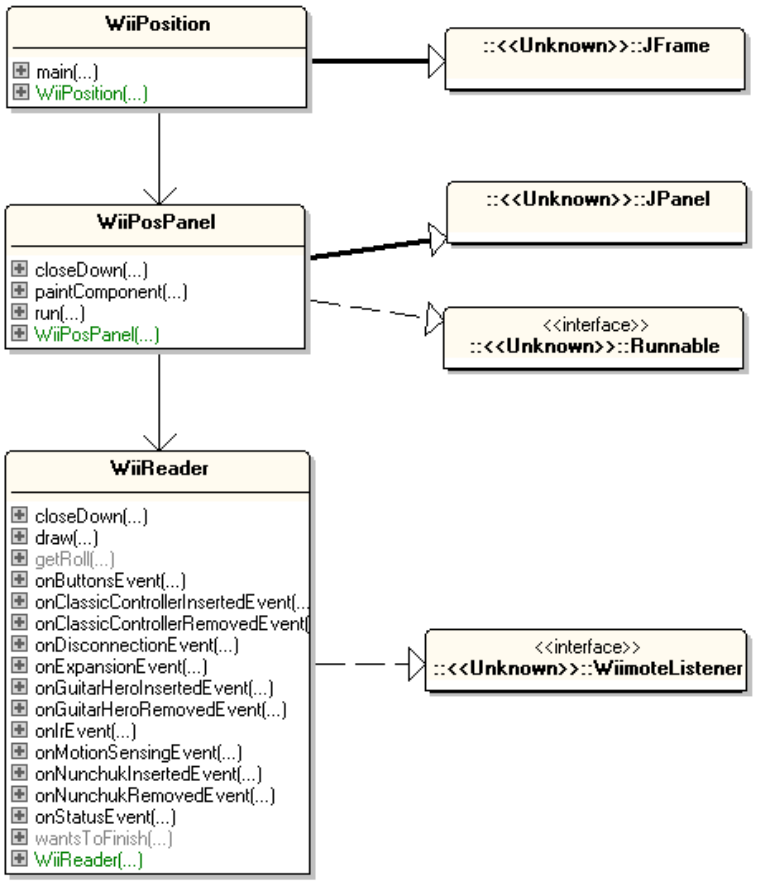

Figure 26. Class Diagrams for WiiPosition.

WiiPosition is a subclass of JFrame which creates a full-screen drawing area (a JPanel subclass), and also hides the cursor. WiiPosPanel starts a threaded loop inside its run() method which updates the application state, draws onto the JPanel, sleeps for a short time, and then repeats. WiiPosPanel's run() method:

```
// globals
```



```
private static final int DELAY = 25;  // ms  (redraw interval)

private boolean isRunning = false;   // used to stop the loop
private WiiReader wiimote;


public void run()
/* Implements the application loop: update / draw / sleep
   The sleep time tries to keep each cycle close to DELAY ms.
*/
{
  long duration;
  isRunning = true;
  while(isRunning && !wiimote.wantsToFinish()) {
    long startTime = System.currentTimeMillis();
    update();
    duration = System.currentTimeMillis() - startTime;
    repaint();

    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);
             // wait until DELAY time has passed
      }
      catch (Exception ex) {}
    }
  }
  wiimote.closeDown();
  System.exit(0);
}  // end of run()
```

The wiimote is accessed via a WiiReader class. An instance is created in the WiiPosPanel constructor:

```
wiimote = new WiiReader(pWidth, pHeight);  // start the wiimote
```

The two arguments are the panel's width and height (the dimensions of the screen), which are used to initialize the view boundaries (the red lines).

WiiPosPanel utilizes the WiiReader object in four places:

- in the loop condition;
- inside the update() method, to update the application's state;
- inside paintComponent() to draw the WiiReader parts of the display;
- at the end of the loop, to close down the wiimote.

The loop condition uses a call to WiiReader.wantsToFinish() to check if the wiimote has requested that the application terminates, which occurs when the user presses the "A" button. Unlike earlier examples, the pressing of this button doesn't immediately trigger an exit. Instead a flag is set so that run() can clean-up before the program exits.

The update() method in WiiPosPanel is very simple since this application doesn't really do anything:

```
private void update()
/* read the wiimote's data, and do something */
{
  System.out.println( wiimote.getRoll());
```



```
}  // end of update()
```

The general purpose of update() is to read the current state of the wiimote (e.g. its buttons, orientation, or tracking position) and use that information to affect the rest of the application (e.g. move a game sprite, activate a menu item).

WiiPosPanel requests a redraw by calling repaint() inside run(). Eventually, this will trigger a call to its paintComponent() method. It renders the graphical state of the application, *and* gives the WiiReader object a chance to draw any wiimote-related elements:

```
public void paintComponent(Graphics g)
/* draw current state of the application,
   and let WiiReader draw any wiimote-related things */
{
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;

  // use antialiasing
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

  wiimote.draw(g2d);
} // end of paintComponent()
```

The current version of paintComponent() doesn't have any application state to draw, but still calls WiiReader.draw() to let the wiimote's graphical information be rendered. WiiReader.draw() contains standard Java 2D code for drawing the IR points, green width line, and red boundary edges seen in Figures 24 and 25.

9.3. Calculating the IR Tracking Location

WiiReader implements WiimoteListener, in a similar way to earlier examples. Its primary aim is to detect IR sources so it can calculate a tracking location, but it also collects information about the roll of the wiimote, and listens for an "A" button press to signal that its time for the application to finish.

WiiReader maintains a variety of global data, much of which is used by its draw() method. Figure 27 shows a close-up of the WiiPosition window showing most of what WiiReader renders.

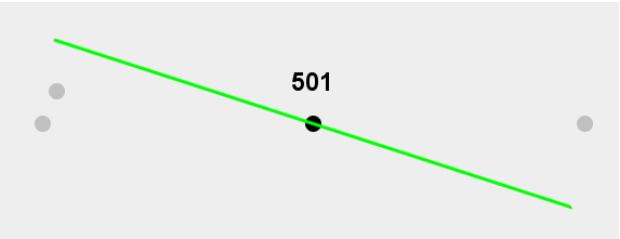

Figure 27. What WiiReader Draws.

Missing from Figure 27 are the four solid red boundary lines, and the two dashed red diagonal lines between their intersecting corners.

The black dot is the calculated tracking location, the three gray dots are the detected IR points, the green line is drawn between two selected IR points representing the two ends of the sensor bar, and the rotation of the line corresponds to the wiimote's current roll.

These pieces of information are stored as globals variables in WiiReader:

```
// globals in WiiReader
private Wiimote wiimote;    // wiiuseJ link to the wiimote

private IRSource[] irPts;   // detected IR points

private Point leftIR, rightIR;
    /* two IR points that are different from each other, and so
       are assumed to be the IR sources at the
       left and right ends of the sensor bar */

private int widthIR;   // distance between leftIR and rightIR points

private Point currLoc;    // current IR tracking point

private int xMinView, xMaxView, yMinView, yMaxView;
     // wiimote's view boundaries for the sensor bar

private int rollAngle = 0;    // roll detected for wiimote
```

The irPts[] array holds the detected IR points last read by the wiimote. This array may be empty or null if no points were observed.

The leftIR and rightIR points are two selected points from the array representing the two ends of the sensor bar. Some clever coding has to be employed if irPts[] doesn't contain two suitable points to ensure that leftIR and rightIR always have values.

widthIR is the distance between the leftIR and rightIR points, and currLoc is an average of the two.

The four 'view' integers represent the two x- axis and two y-axis boundary lines.

rollAngle is the roll angle returned by the wiimote's motion sensor.

9.4. Initializing the Wiimote

The wiimote link is initialized in the usual way, inside an initWimote() method called from the WiiRemote constructor:

```
// global
private boolean wantsToFinish = false;


private Wiimote initWiimote(int pWidth, int pHeight)
{
  System.out.println("Initializing wiimote");
  if (WiiUseApiManager.getNbConnectedWiimotes() == 0) {
    System.out.println("No wiimotes connected");
    wantsToFinish = true;
    return null;
  }
```



```
  Wiimote[] wiimotes = WiiUseApiManager.getWiimotes(1, false);
  if ((wiimotes == null) || (wiimotes.length == 0)){
    System.out.println("No wiimote found");
    wantsToFinish = true;
    return null;
  }
  Wiimote wiimote = wiimotes[0];
  wiimote.setTimeout((short)20, (short)20);

  wiimote.activateIRTRacking();
  wiimote.activateMotionSensing();

  wiimote.setIrSensitivity(5);
  wiimote.setSensorBarAboveScreen();
    /* don't both with virtual screen settings since
       only using detected IR points */

  wiimote.setOrientationThreshold(1);
    /* the amount an angle (roll, pitch, or yaw) must change
       before generating an event */

  wiimote.addWiiMoteEventListeners(this);
  wiimote.getStatus();
  return wiimote;
}  // end of initWiimote()
```

Rather than exiting if there's a problem, initWiimote() sets the global wantsToFinish boolean to true. This flag can be read by the Swing application calling WiiReader.wantsToFinish():

```
public boolean wantsToFinish()
{  return wantsToFinish;  }
```

initWiimote() doesn't use virtual screen settings. They're unnecessary since WiiReader only reads the detected IR points, which use a fixed screen size of 1024 x 768, with (0,0) at the top-left.

9.5. Processing IR Points

When the wiimote's IR camera detects some points, the onIrEvent() method is called:

```
// globals
private IRSource[] irPts;   // detected IR points
private Point currLoc;      // current IR tracking point


public void onIrEvent(IREvent e)
{
  irPts = e.getIRPoints();
  Point loc = locateIR(irPts);
  if (loc != null)
    currLoc = loc;
}  // end of onIrEvent()
```


locateIR() is a complex method since it has to choose two points from the detected IR sources to represent the two ends of the sensor bar (leftIR and rightIR). These are used to calculate the tracking point returned by the method.

Part of the method's complexity is dealing with the situation when the irPts[] array doesn't contain two suitable points. If no points were supplied then leftIR and rightIR are left unchanged, which leaves the tracking position unchanged. If only one point is found, then the method utilizes the old leftIR or rightIR to 'guess' at a reasonable value for the other point. Parts of this algorithm were inspired by a discussion of sensor bar point calculation at http://wiibrew.org/wiki/Wiimote/Pointing.

The locateIR() code:

```
// globals
private Point leftIR, rightIR;
private int widthIR;   // distance between leftIR and rightIR points


private Point locateIR(IRSource[] irPts)
{
  if ((irPts != null) && (irPts.length > 0)) {
    // there are some IR points to work with
    Point firstPt = new Point( irPts[0].getX(), irPts[0].getY());
    Point secondPt = new Point(0,0);  // dummy value

    // find a second point that is not close to the first
    boolean foundPair = false;
    int i=1;
    while (i < irPts.length) {
      secondPt.setLocation(irPts[i].getX(), irPts[i].getY());
      if (!isClose(firstPt, secondPt)) {
        foundPair = true;
        break;
      }
      i++;
    }

    if (foundPair) {    // got two different IR points
      if (firstPt.x > secondPt.x) {
        // swap so in left-right order
        Point temp =  firstPt;
        firstPt = secondPt;
        secondPt = temp;
      }

      // store the two points in the leftIR and rightIR globals
      leftIR.setLocation(firstPt);
      rightIR.setLocation(secondPt);

      // calculate width between points
      widthIR =  (int) Math.round(leftIR.distance(rightIR));
    }
    else  {  /* did not find a different second point, so try
                guessing  based on one of the leftIR/rightIR
                points calculated earlier. Use the movement of
                the first point relative to the IR point to
                update the other point. */
      if (isClose(firstPt, leftIR)) {    // guess based on leftIR
        int xOffset = firstPt.x - leftIR.x;
```



```
        int yOffset = firstPt.y - leftIR.y;
        leftIR.setLocation(firstPt);
        rightIR.translate(xOffset, yOffset);
                    // apply offset to old rightPt
        widthIR =  (int) Math.round(leftIR.distance(rightIR));
      }
      else if (isClose(firstPt, rightIR)) {  //guess based on rightIR
        int xOffset = firstPt.x - rightIR.x;
        int yOffset = firstPt.y - rightIR.y;
        rightIR.setLocation(firstPt);
        leftIR.translate(xOffset, yOffset);
                   // apply offset to old leftPt
        widthIR =  (int) Math.round(leftIR.distance(rightIR));
      }
    }
  }
  return averagePoints(leftIR, rightIR);
}  // end of locateIR()
```

leftIR and rightIR represent the two ends of the sensor bar, and so cannot be too close together. If two detected points are near to each other then they're probably adjacent IR LEDs on the same side of the bar. Closeness is defined using isClose():

```
// global
private static final double POINT_PROXIMITY = 80;
    // pixel distance between two points which is deemed 'close'

private boolean isClose(Point p0, Point p1)
{  return (p0.distance(p1) < POINT_PROXIMITY);   }
```

This approach may fail if the wiimote is a long way from the sensor bar, causing all the detected IR points to be less than POINT_PROXIMITY (80) pixels apart. However, my interest is in a user sitting (or standing) close to a PC, and so this problem is unlikely to occur.

If only one point is found, it is stored in the firstPt variable. This value is compared with the old values for leftIR and rightIR using isClose() to decide which point is closest. If leftIR (or rightIR) is closest, then the old rightIR (leftIR) is used for the other point. In addition, the offset of the new point from leftIR (rightIR) is applied to rightIR (leftIR).

As a side-effect of updating leftIR and rightIR, the width between them (widthIR) is re-calculated.

averagePoints() calculates the tracking position as the mid-point between leftIR and rightIR.

```
// globals
private int pWidth, pHeight;   // panel dimensions


private Point averagePoints(Point p0, Point p1)
{
  int x = (p0.x + p1.x)/2;
  if (x < 0)
    x = 0;
  else if (x > pWidth-1)
```



```
    x = pWidth-1;

  int y = (p0.y + p1.y)/2;
  if (y < 0)
    y = 0;
  else if (y > pHeight-1)
    y = pHeight-1;

  return new Point(x, y);
}  // end of averagePoints()
```

Extra tests are included to ensure that the mid-point cannot be positioned off the screen.

9.6. Moving the View Boundary Lines

WiiReader monitors the wiimote buttons, so that the viewing boundary lines can be adjusted, and the user can press "A" to finish:

```
// globals
private boolean wantsToFinish = false;
private Point currLoc;    // current IR tracking point

private int xMinView, xMaxView, yMinView, yMaxView;
          // wiimote's view boundaries for the sensor bar


public void onButtonsEvent(WiimoteButtonsEvent e)
{
 if (e.isButtonLeftPressed())
    xMinView = currLoc.x;
  if (e.isButtonRightPressed())
    xMaxView = currLoc.x;
  if (e.isButtonUpPressed())
    yMinView = currLoc.y;
  if (e.isButtonDownPressed())
    yMaxView = currLoc.y;

  if (e.isButtonAPressed())
    // set wantsToFinish, which is read by the outer Swing app
    wantsToFinish = true;
}  // end of onButtonsEvent()
```

The pressing of an arrow key causes a view boundary to be set to the current tracking location's x- or y- axis value. The idea is that the user will notice when a boundary had been reached because no IR sources will be drawn on screen (the gray dots in Figure 27). If the user doesn't notice their absence, WiiReader.draw() also changes the drawing color of the tracking point from black to red.

9.7. Roll with the Wiimote

WiiReader monitors the wiimote's orientation, updating the global rollAngle with the current roll. It is used inside WiiReader.draw() to rotate the green line on screen.


```
// global
private int rollAngle = 0;    // roll detected for wiimote


public void onMotionSensingEvent(MotionSensingEvent e)
{
  Orientation ori = e.getOrientation();
  if (ori != null)
    rollAngle = (int)Math.round(ori.getRoll());
                            // roll is around y-axis
}  // end of onMotionSensingEvent()
```

rollAngle also has a 'get' method, which allows the angle to be utilized by the wider application.

```
public int getRoll()
{  return rollAngle;  }
```

To make more of the wiimote's state visible to the rest of WiiPosition requires the addition of more 'get' methods for buttons, orientation, and acceleration data. Typically, they will be called inside WiiPosPanel.update() to update the application's state.

