Chapter 28.10 (N14). The P5 Glove

The P5 Virtual Reality Glove (Figure 1) is a low-price data glove suitable for gaming and 3D virtual environments.

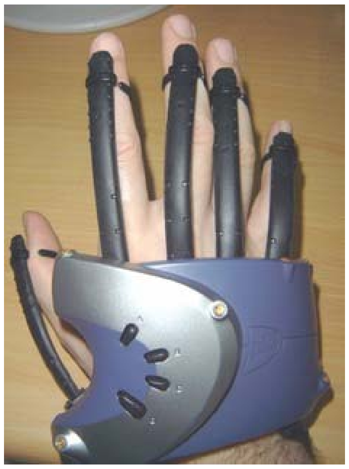


Figure 1. The P5 Virtual Reality Glove.

The user moves their hand in front of a receptor 'tower' (shown on the right of Figure 2), which contains two infrared sensors. They detect the visible LEDs on the glove (there's eight altogether), and convert them into an (x, y, z) position for the glove, and an orientation in terms of pitch, yaw, and roll.

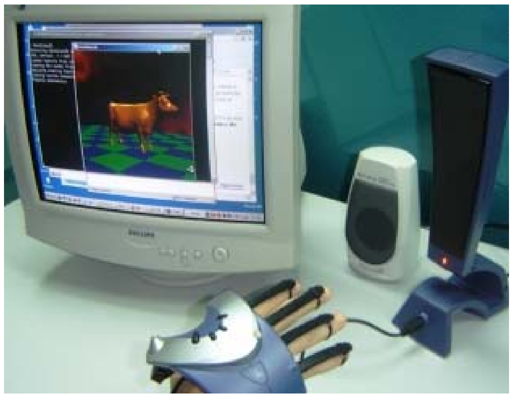


Figure 2. The Glove and the Receptor Tower.

The glove is plugged into the tower, which is connected to the PC's USB port.

The glove also has bend sensors in its fingers, and four buttons on the top.

I'll explain how to communicate with the P5 glove using Java, and implement a Glove class that hides most of the interfacing details.

However, the chapter's main focus is on developing a Java 3D application called HandView3D, a variant of ObjView3D from chapter 7 ??. This time the camera moves forwards, backwards, and turns left or right based only on glove movements; no key presses are necessary.

A musical cow is standing in the middle of the scene (visible in Figure 2). As the user approaches or retreats, the music's intensity varies, as does the mix of sound coming from the speakers. The spatial effects are managed by my JOAL sound manager class from the last chapter. Perhaps mercifully, the music can be paused by the user clenching their fist or pressing a button on the glove. Another fist-clench, or button press, resumes the audio accompanyment.

One advantage of using the P5 is that several operations can be carried out at once; for example, a user can move the camera forwards and left at the same time. This is possible because the P5's data is accessed using polling.

1. P5 Software and Support

The P5 is an amazing piece of hardware, but it's driver software is less impressive. Fortunately, hackers and hobbyists have done a great job of supplying replacements.

Carl Kenner's site (http://www.geocities.com/carl_a_kenner/p5glove.html) has links to P5 drivers for Windows, Linux, and the Mac. I downloaded Kenner's *Dual Mode Driver Beta 3* for Windows. It's 'dual mode' since it supplies position data in either relative or absolute terms. A relative position is an offset from the last glove position, while an absolute position is relative to a fixed origin in space.

Compared to the P5's official software, Kenner's driver offers improved data filtering, better accuracy, better access to finger and LED information, and many other goodies. Most importantly for my needs, it has a Java API. (Other languages are supported as well, including Delphi, Visual Basic, C, and Visual C++.)

Three great sources of glove information are the P5 Glove mini wiki at Wikia at http://scratchpad.wikia.com/wiki/P5_Glove, the P5 Community page at http://www.zzz.com.ru/index.php?area=pages&action=view_page&page_id=11, and the p5glove Yahoo group at http://groups.yahoo.com/group/p5glove/.

I haven't mentioned a company website, because the P5's manufacturers, Essential Reality, sadly went out of business in the middle of 2004. As a consequence, it's possible to find brand new P5 gloves for sale for around US$50 on eBay; Essential Reality was selling them for US$150. Professional data gloves cost far more.

2. Using Kenner's Java API

After downloading and unzipping Kenner's Dual Mode driver (e.g. from http://zzz.com.ru/zzz_original_site/roid/A_DualModeDriverBeta3.zip), the files required for Java programming are P5DLL.dll, and CP5DLL.java in the subdirectory include/com/essentialreality/.

It's easiest to work with a CP5DLL JAR:

```
> javac –d . CP5DLL.java   // create the com.essentialreality package
> jar cvf CP5DLL.jar com   // roll the package up into a JAR
```

The resulting JAR file, CP5DLL.jar, should be copied into <JAVA_HOME>\jre\lib\ext and <JRE_HOME>\lib\ext. On my WinXP test machine, they're the directories c:\Program Files\Java\jdk1.6.0\jre\lib\ext and c:\Program Files\Java\jre1.6.0\lib\ext.

The DLL, P5DLL.dll can be placed in a directory of your choice. I chose "c:\Program Files\P5 Glove" the directory created by Essential Reality's P5 installer.

Every call to java.exe requires a command line option to supply the path to the DLL's location. For instance, to run the P5 application P5Example.java needs:

```
> java -Djava.library.path="c:\Program Files\P5 Glove" P5Example
```

I'm lazy so I use a DOS batch file instead:

```
> runP5 P5Example
```

runP5.bat contains:

```
@echo off
echo Executing P5 code
java -Djava.library.path="c:\Program Files\P5 Glove" %1
echo Finished.
```

The CP5DLL class consists of numerous public constants, variables, and methods, and three inner classes: P5Data, P5Info, and P5State (as illustrated by Figure 3). P5Data can be ignored since it's only included for backward compatibility; it's functionality has been superceded by P5Info and P5State.

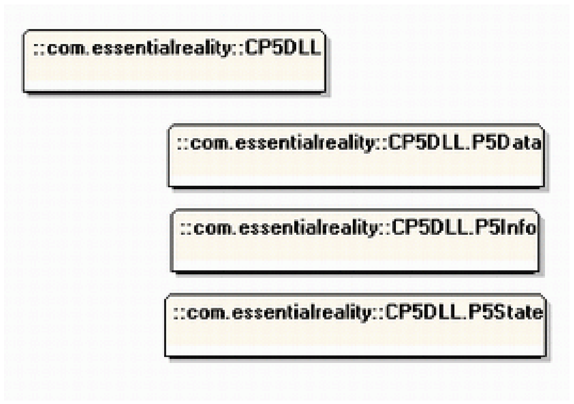

::com.essentialreality::CP5DLL
::com.essentialreality::CP5DLL.P5Data
::com.essentialreality::CP5DLL.P5Info
::com.essentialreality::CP5DLL.P5State

Figure 3. The CP5DLL Classes.

The 40+ public methods in the top-level CP5DLL class are used to initialize and configure the glove (or gloves) connected to the PC. For example, it's possible to adjust the amount of data filtering.

The 40+ public variables in P5State expose the glove's current position, rotation, button values, LEDs, and much more. There's an update() method which should be called whenever the values need to be refreshed.

Most of the 25+ public variables in P5Info hold device details, such as the manufacturer name and the version numbers. As with P5State, there's an update() method which initializes the values.

The documentation for Kenner's Java API is rather sparse; for instance, there's no Java example in the distribution. The best place for finding out about the API is the p5glove Yahoo group (http://groups.yahoo.com/group/p5glove/), which has a searchable interface; Kenner is an active member.

The only Java example I could find was the fun "Robot Glove" Java application by Eric Lundquist (http://www.robotgroup.net/index.cgi/RobotGlove). Inputs from the P5 glove are converted into commands sent to a Lynxmotion robot arm via the Java Communications API: as your hand moves, so does the robot.

3. Examining the Glove's Data

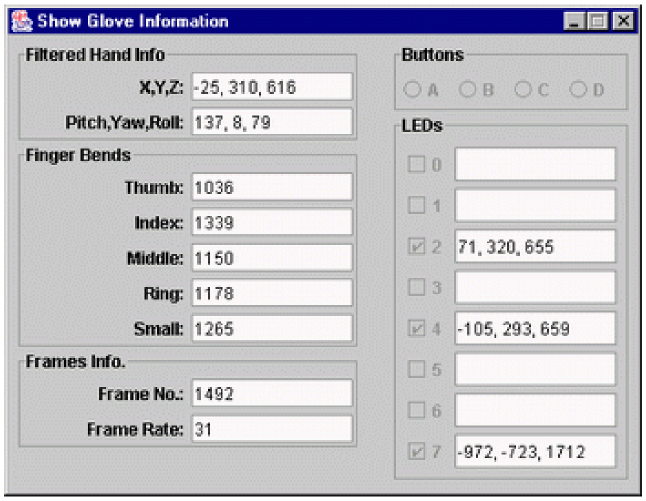
The ShowGlove application described in this section allows the different kinds of glove data to be examined easily.

Later, I'll use this program to decide how to map glove gestures (i.e. particular positions or orientations of the glove) to camera movement actions in HandView3D.

It's fairly easy to come up with suitable gestures; for instance, turning left is triggered by rolling the glove to the left. But, how *far* should the glove be rolled before that 'counts' as a camera turn? Is a 20 degree roll enough, or 40 degrees? The numerical details can be worked out by looking at the values reported by ShowGlove as different glove gestures are tried out.

3.1. What ShowGlove Displays

The ShowGlove GUI is shown in Figure 4.



Show Glove Information

Filtered Hand Info
X,Y,Z: -25, 310, 616
Pitch,Yaw,Roll: 137, 8, 79

Finger Bends
Thumb: 1036
Index: 1339
Middle: 1150
Ring: 1178
Small: 1265

Frames Info.
Frame No.: 1492
Frame Rate: 31

Buttons
A B C D

LEDs
0
1
✓ 2 71, 320, 655
3
✓ 4 -105, 293, 659
5
6
✓ 7 -972, -723, 1712

Figure 4. The ShowGlove GUI.

Filtered versions of the glove's position and orientation are shown in the "Filtered Hand Info" panel. The P5 filters smooth out 'jumps' in the data caused by the tower failing to detect some of the LEDs. The reported numbers are averages of the current position/rotation *and* the ten previous values. This filtering strategy can be adjusted via methods in CP5DLL.

The (x, y, z) numbers are absolute values, relative to coordinate axes shown in Figure 5. The origin is centered near the base of the tower.

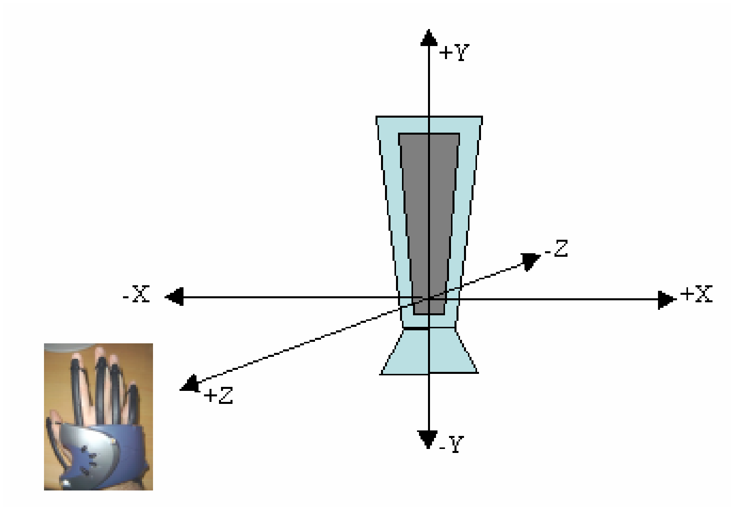


Figure 5. Coordinate Axes for the Glove.

The glove's orientation is given in terms of pitch, yaw, and roll, which are illustrated in Figure 6.

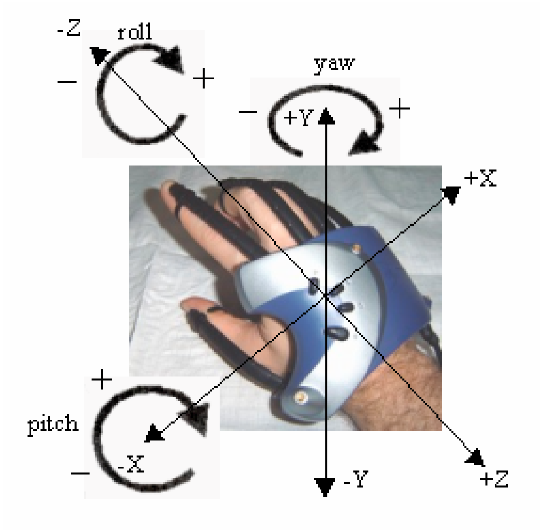


Figure 6. Pitch, Yaw, and Roll for the Glove.

Pitch is the rotation about the x-axis, yaw acts around the y-axis, and roll uses the z-axis. A positive pitch rotates the hand upwards, a positive yaw turns it to the right, and a positive roll turns the top of the hand to face right.

The fingers bend data depends on the glove's calibration settings. (The P5 is calibrated via its Windows control panel, which comes as part of its installation software.) On my glove, out-stretched fingers typically show a value of around 1000, while bent fingers register around 400.

There are four buttons on the top face of the glove, labeled 'A', 'B', 'C', and 'D'. These are mapped to the same-named radio buttons in the ShowGlove GUI. When a button is pressed, the radio button is selected, the exception being the 'D' button. When 'D' is pressed, the glove automatically switches off, so the button's change isn't detected by CP5DLL, and not shown by ShowGlove.

There are eight LEDs on the glove, numbered as in Figure 7.

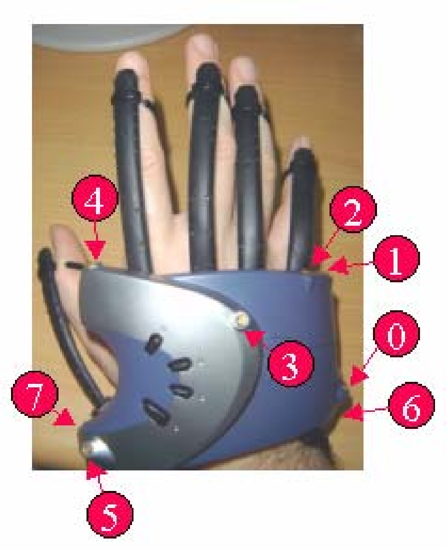


Figure 7. LEDs on the Glove.

Those numbers are used as labels in the LEDs panel in ShowGlove.

The reliability of the position and orientation data depends on the LEDs being visible to the tower. One of the problems is that the user's fingers may block LEDs on the front edge of the glove (e.g. LEDs 1, 2, and 4) from being seen. The simplest solution is for the user to keep his fingers bent, and to bend his wrist so that the top of his hand is aimed slightly towards the tower. Hand positioning is one of the hidden training issues with the P5 glove, analogous to the training needed when first using a mouse.

Another trick for making the top of the user's hand more visible is to place the tower on a box, so it's slightly higher than the hand.

3.2. ShowGlove Overview

The class diagrams for ShowGlove are given in Figure 8. Only the public methods are shown.

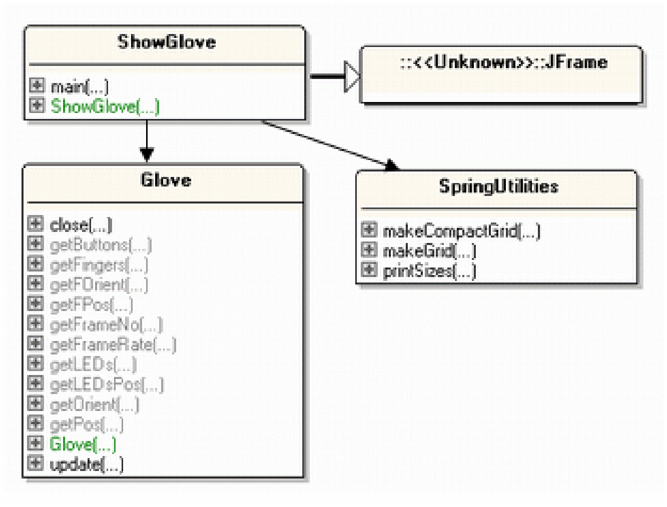


Figure 8. Class Diagrams for ShowGlove.

The ShowGlove class creates the GUI, and polls the glove every GLOVE_DELAY ms (200 ms) to gather data to populate the GUI's various text fields, radio buttons, and check boxes. The polling phase calls update() in the Glove class followed by its get methods.

The Glove class is a wrapper (a facade) around Kenner's CP5DLL class, hiding the glove's initialization, data collection, and its closedown at the end of the application.

The SpringUtilities class comes from the J2SE 1.4.2 tutorial; it contains utility methods for manipulating the SpringLayout class. SpringLayout permits components to be positioned in a grid formation without each cell having to be the same size (as is the case with GridLayout). I use SpringLayout for the LEDs panel (see Figure 4) since the check boxes column needs much less horizontal space than the text fields in the adjacent column.

I won't describe the GUI creation code in ShowGlove – it's quite standard, and not relevant to how the glove is accessed.

3.3. Initializing and Terminating the Glove

The ShowGlove constructor creates a Glove instance, and sets up a window listener which closes down the P5 glove when the JFrame is closed.

```
// global variables
private Glove glove;

// for glove polling
private Timer pollTimer;   // timer which triggers the polling
private DecimalFormat df;
```

```
public ShowGlove()
{
  super("Show Glove Information");

  glove = new Glove();
  initGUI();

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

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

  df = new DecimalFormat("0");    // no decimal places
  startPolling();
}  // end of ShowGlove()
```

The windowClosing() method also stops the polling timer.

3.4. Polling the Glove

startPolling() sets up a timer to be activated every GLOVE_DELAY ms. The timer triggers an update of the glove's data and refreshes the GUI.

```
// time delay between glove polling
private static final int GLOVE_DELAY = 200;  // ms


private void startPolling()
{
  ActionListener pollPerformer = new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
      glove.update();  // call update() before showing the data
      showHandInfo();
      showFingersInfo();
      showFramesInfo();
      showButtonsInfo();
      showLEDsInfo();
    }
  };

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

The GUI must be updated in the event-dispatching thread, so a javax.swing Timer is utilized which executes the action handler (pollPerformer) in that thread.

After the call to Glove.update(), the show methods (e.g. showHandInfo()) call Glove's get methods. For example:

```
// global GUI variable – it displays the filtered glove
   position (X,Y,Z) and the filtered Pitch,Yaw,Roll. */
private JTextField handInfo[];


private void showHandInfo()
// show the filtered (x,y,z) and (pitch,yaw,roll)
{ showVals(handInfo[0], glove.getFPos());
  showVals(handInfo[1], glove.getFOrient());
}

private void showVals(JTextField tf, float vals[])
{ tf.setText( df.format(vals[0]) + ", " +
              df.format(vals[1]) + ", " +
              df.format(vals[2]) );  }
```

The Glove.getFPos() and Glove.getFOrient() methods return the filtered position and orientation as three-element arrays. Their data is formatted and written to the two text fields in the "Filtered Hand Info" panel (see Figure 4).

4. The Glove

The Glove class is a facade for Carl Kenner's CP5DLL class; it initializes the glove, closes it down, and offers a small set of get methods. These supply:

- the glove's (x,y,z) position, both filtered and raw data;
- the glove's pitch, yaw, and roll, both filtered and raw data;
- finger bend amounts;
- button presses;
- visible LEDs, and their values;
- the current frame number, and frame rate.

A typical usage pattern for the Glove class is to call its update() method, followed by one or more get methods.

The P5 glove's default configuration is utilized, except that the z-axis is reversed, so the positive direction is towards the user (as shown in Figure 5). Normally, the negative z-axis is pointing outwards. Also, the P5's mouse mode is switched off, which stops glove movements and finger bends being interpreted as mouse moves and button presses.

The Glove() constructor creates a CP5DLL instance, and initializes it.

```
// links to the P5 glove(s)
private CP5DLL gloves;
private CP5DLL.P5State gloveState;
```

```
public Glove()
{
  gloves = new CP5DLL();

  // initialize the glove
  if (!gloves.P5_Init()) {
    System.out.println("P5 Initialization failed");
    System.exit(1);
  }

  gloves.P5_SetForwardZ(-1);  // so positive Z is towards the user
  gloves.P5_SetMouseState(-1, false);  // disable mouse mode

  // make sure there's only one glove
  int numGloves = gloves.P5_GetCount();
  if (numGloves > 1) {
    System.out.println("Too many gloves detected: " + numGloves);
    System.exit(1);
  }

  printGloveInfo();
  gloveState = gloves.state[0];  // store a ref to the glove state
} // end of Glove()
```

The CP5DLL instance, gloves, can potentially access multiple gloves, a feature I don't need. So a reference to the state of the single glove is stored in gloveState; it's used subsequently to update and access the glove's data.

printGloveInfo() employs the CP5DLL.P5Info class to print out details about the device. A code fragment shows the general approach:

```
CP5DLL.P5Info gloveInfo = gloves.info[0];
System.out.println(
        "Vendor ID: " + gloveInfo.vendorID +
        "; Product ID: " + gloveInfo.productID +
        "; Version: " + gloveInfo.version);
```

4.1. Updating and Accessing the Glove

Glove's update() method calls the update() method in CP5DLL.P5State:

```
public void update()
{ gloveState.update(); }
```

The get methods in Glove read the public variables exposed by CP5DLL.P5State. For instance, getFPos() and getFOrient() return the filtered position and orientation data from CP5DLL.P5State.

```
public float[] getFPos()
// filtered (x,y,z) position
{ return new float[]{gloveState.filterPos[0],
                     gloveState.filterPos[1],
                     gloveState.filterPos[2]};  }

public float[] getFOrient()
```

```
// filtered (pitch,yaw,roll)
{ return new float[] {gloveState.filterPitch,
                      gloveState.filterYaw,
                      gloveState.filterRoll};  }
```

4.2. Closing Down

Glove's close() method restores the mouse mode setting, and closes the connection to the P5.

```
public void close()
{ gloves.P5_RestoreMouse(-1);
  gloves.P5_Close();
}
```

Unfortunately, the call to restore the mouse mode has no effect on my Windows XP test machine. It’s necessary to go to the P5 Glove control panel to switch the mode back on.

5. A First-Person Shooter Glove Class

The FPSGlove class is another wrapper for Kenner's CP5DLL class, with many similarities to the previous section's Glove class. The class diagram for FPSGlove is given in Figure 9, showing only its public methods.

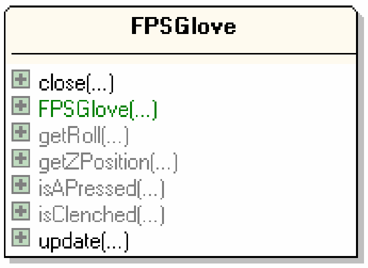


Figure 9. Class Diagram for FPSGlove.

FPSGlove's new methods:

- getZPosition() returns a constant representing the z-axis position of the glove relative to the tower. The constant can be NEAR, FAR or MIDDLE.
- getRoll() returns a constant representing the roll orientation of the glove. The constant can be ROLL_LEFT, ROLL_RIGHT, or LEVEL.
- isClenched() returns true if the user's fingers are bent enough to classify as a clenched fist.
- isAPressed() returns true if the "A" button on the glove has been pressed.

This new glove wrapper class only accesses the CP5DLL data needed by HandView3D, so is smaller than the Glove class. I've also simplified the class' interface, returning constants and booleans instead of arrays of floats.

5.1. Initializing the Glove

The FPSGlove() constructor is almost identical to the one in Glove. It initializes the link to the glove, and stores the glove state for future use.

```
// links to the P5 glove(s)
private CP5DLL gloves;
private CP5DLL.P5State gloveState;


public FPSGlove()
{
  gloves = new CP5DLL();

  // initialize the glove
  if (!gloves.P5_Init()) {
    System.out.println("P5 Initialization failed");
    System.exit(1);
  }

  gloves.P5_SetForwardZ(-1);    // so positive Z is towards the user
  gloves.P5_SetMouseState(-1, false);  // disable mouse mode

  // make sure there's only one glove
  int numGloves = gloves.P5_GetCount();
  if (numGloves > 1) {
    System.out.println("Too many gloves detected: " + numGloves);
    System.exit(1);
  }

  gloveState = gloves.state[0];  // store a ref to the glove state
} // end of FPSGlove()
```

5.2. Updating and Closing

The update() and close() methods are unchanged from the Glove class:

```
public void update()
{ gloveState.update(); }

public void close()
{ gloves.P5_RestoreMouse(-1);  // restore mouse mode (does not work)
  gloves.P5_Close();
}
```

5.3. Getting the Position

getZPosition() examines the filtered z-axis location of the glove, and returns one of the position constants NEAR, FAR, or MIDDLE.

The idea is that HandView3D will move the camera forward when the value is NEAR (i.e. near to the tower), move the camera back when the value is FAR (far from the tower), and not adjust the camera at all when the value is MIDDLE.

After some experimentation with ShowGlove, I decided that NEAR should start at less than 500 units from the tower, while FAR should start at over 900. Therefore, the MIDDLE region is between 500-900 units, which is a comfortable distance to rest my arm on the computer desk, and not have the camera move. By default, a glove unit is roughly 0.5mm.

```
// public z-axis position constants (for closeness to tower)
public final static int NEAR = 0;
public final static int FAR = 1;
public final static int MIDDLE = 2;

// position constraints
private final static int NEAR_MIN = 500;
                       // how near to the tower before it counts
private final static int FAR_MIN = 900;   // how far back


public int getZPosition()
{
  float zPos = gloveState.filterPos[2];    // filtered z-axis pos
  if (zPos < NEAR_MIN)    // near to the tower
    return NEAR;
  else if (zPos > FAR_MIN)  // far from the tower
    return FAR;
  else
    return MIDDLE;
}  // end of getZPosition()
```

The NEAR, FAR, and MIDDLE constants are public so they can be utilized by other classes.

5.4. On a Roll

getRoll() examines the filtered roll value supplied by the P5, and returns one of the constants ROLL_LEFT, ROLL_RIGHT, or LEVEL. HandView3D uses these constants to turn the camera left, right, or not at all.

Once again, I've used threshold values for triggering the rotation – when the glove is rotated more than 80 degrees to the right, then ROLL_RIGHT is returned. When the glove is less than -40 degrees to the left, then ROLL_LEFT is the result. Between -40 and 80, the glove is assumed to be un-rotated, and returns LEVEL.

The greater threshold for rolling right is due to the way that I rest my right hand on the desk, rotated right by about 30 degrees. I don't want a rotation to occur when my hand is in that position.

```
// public roll orientation constants
public final static int ROLL_LEFT = 3;
public final static int ROLL_RIGHT = 4;
public final static int LEVEL = 5;

// rotation constraints
```

```
private final static int RIGHT_ROLL_MIN = 80;
              // how far to roll right before it counts
private final static int LEFT_ROLL_MIN = -40;  // roll left


public int getRoll()
{
  float roll = gloveState.filterRoll;   // the glove's roll
  if (roll < LEFT_ROLL_MIN)   // rolled left
    return ROLL_LEFT;
  else if (roll > RIGHT_ROLL_MIN)  // rolled right
    return ROLL_RIGHT;
  else
    return LEVEL;
}  // end of getRoll()
```

5.5. Clenching my Fist

When I clench my fist, I want the the music playing in HandView3D to pause (or resume if it’s already paused).

The data supplied by ShowGlove shows that clenching usually causes the thumb bend value to decrease to around 800, the little finger to around 500, and the other fingers to about 350. In other words, my glove doesn't generate particularly low bending values for the thumb and little finger.

As a consequence, I've coded isClenched() to returns true if at least FINGERS_BENT fingers (three, including the thumb) are bent to have values less than or equal to BEND_MIN (500).

```
// finger bend constraints
private final static int FINGERS_BENT = 3;
          // the minimum no. of fingers that need to be bent

private final static int BEND_MIN = 500;
          /* how far a finger must bend to count
             (a smaller value means more bend) */


public boolean isClenched()
{
  short[] fingerBends = gloveState.fingerAbsolute;

  boolean isClenched = false;
  int bentCount = 0;
  for(int i=0; i < fingerBends.length; i++) {
    if (fingerBends[i] <= BEND_MIN)  // bent enough to count
      bentCount++;
  }
  return (bentCount >= FINGERS_BENT);
}  // end of isClenched()
```

5.6. Pressing the "A" Button

isAPressed() returns true if the glove button labeled "A" is pressed.

```
public boolean isAPressed()
{ boolean[] buttonPresses = gloveState.button;
  return buttonPresses[0];  // the "A" button
}
```

HandView3D utilizes isAPressed() as another way of toggling the music on and off.

One issue is that a button press will only be recorded in gloveState if the button is depressed when update() is called. This won't be a problem if the update() calls occur frequently (e.g. every 100 ms, or more often), since a button press takes several tenths of a second to be carried out.

6. A Test-rig for FPSGlove

The MotionTest application is a test-rig for the FPSGlove class.

One of my aims is to assess the threshold values used in FPSGlove for detecting the glove's position and orientation. MotionTest is also used to check the initialization, update/get cycle, and termination phases of FPSGlove.

MotionTest has a deliberately simple user interface, so I don't need to worry about Swing or Java 3D. It prints a 'N' or 'F' when the glove is near or far from the receptor tower, a 'L' or 'R' if the glove is rolled to the left or right, and a '*' if the glove is clenched.

A typical execution is shown below:

```
> runP5 MotionTest
Executing P5 code: MotionTest ...
N N N N N N N N N N N N N N N N N N L N L
N L N L N L N L N L N L N L N L N L N L
F L F L F L F L F L F L F L F L F L F L
F L F L F L F L F L L L F F F * * * * *
* * * * * * * * N * N * N * N * N * N *
N * N *
>
```

The glove starts near the tower, then is rotated left. It stays rotated left as it's moved further away from the tower. The user clenches their fist when the glove is level and a middling distance from the tower, then the glove is moved nearer to the tower again.

The MotionTest constructor initializes FPSGlove, enters an update/get cycle, and terminates when the "A" button is pressed. Between each update/get iteration it sleeps for GLOVE_DELAY ms.

```
// delay between glove polling
private static final int GLOVE_DELAY = 100;  // ms

private int i = 1;   // used for formatting the text output
```

```
public MotionTest()
{
  FPSGlove glove = new FPSGlove();

  while(!glove.isAPressed()) {
    glove.update();   // update the glove settings

    // show position, roll, hand clenching
    showZPosition(glove);
    showRoll(glove);
    if (glove.isClenched()) {
      System.out.print("* ");
      i++;
    }

    if (i > 20) {    // lots of info printed, so add a newline
      i = 1;
      System.out.println();
    }

    try {   // sleep a bit
      Thread.sleep(GLOVE_DELAY);
    }
    catch(InterruptedException e) {}
  }

  glove.close();
}  // end of MotionTest()
```

After an update, the current position, roll, and firing status are all retrieved. This means that several letters may be printed during a single update/get iteration. For example, if the glove is near to the tower, rotated left, and the fingers are bent, then 'N', 'L', and '*' will be printed.

The z-axis position is reported as 'N' (near to the tower), or 'F' (far). showZPosition() utilizes FPSGlove.getZPosition().

```
private void showZPosition(FPSGlove glove)
{
  int handPos = glove.getZPosition();
  switch(handPos) {
    case FPSGlove.NEAR: System.out.print("N "); i++; break;
    case FPSGlove.FAR: System.out.print("F "); i++; break;
    case FPSGlove.MIDDLE: break;
    default: System.out.print("?? "); i++; break;  //shouldn't happen
  }
}  // end of showZPosition()
```

showRoll() prints a 'L' (the glove is rolled left) or 'R' (rolled right). It employs FPSGlove.getRoll().

```
private void showRoll(FPSGlove glove)
{
  int handOrient = glove.getRoll();
  switch(handOrient) {
    case FPSGlove.ROLL_LEFT: System.out.print("L "); i++; break;
    case FPSGlove.ROLL_RIGHT: System.out.print("R "); i++; break;
    case FPSGlove.LEVEL: break;
    default: System.out.print("?? "); i++; break;  //shouldn't happen
  }
}  // end of showRoll()
```

7. Visiting the Musical Cow

A screenshot of HandView3D is shown in Figure 10.

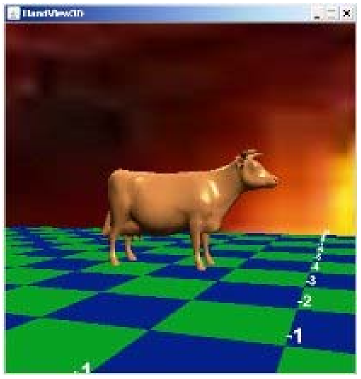


Figure 10. The HandView3D Application.

The 3D scene is a familar one – a checkboard floor, a background, lights. The cow is new, as is the music, spatially located under the cow, and playing repeatedly.

The user can travel through the scene by moving their glove forwards, backwards, left, and right. As the camera's position changes in relation to the cow, the music varies in intensity, and the speaker mix is adjusted.

If the music-lover clenches his fist, the music is paused until he clenches it again. The music can also be toggled on and off by pressing the "A" button on the glove.

Figure 2 shows the user navigating around the scene.

Figure 11 gives the class diagrams for HandView3D, showing only the public methods.

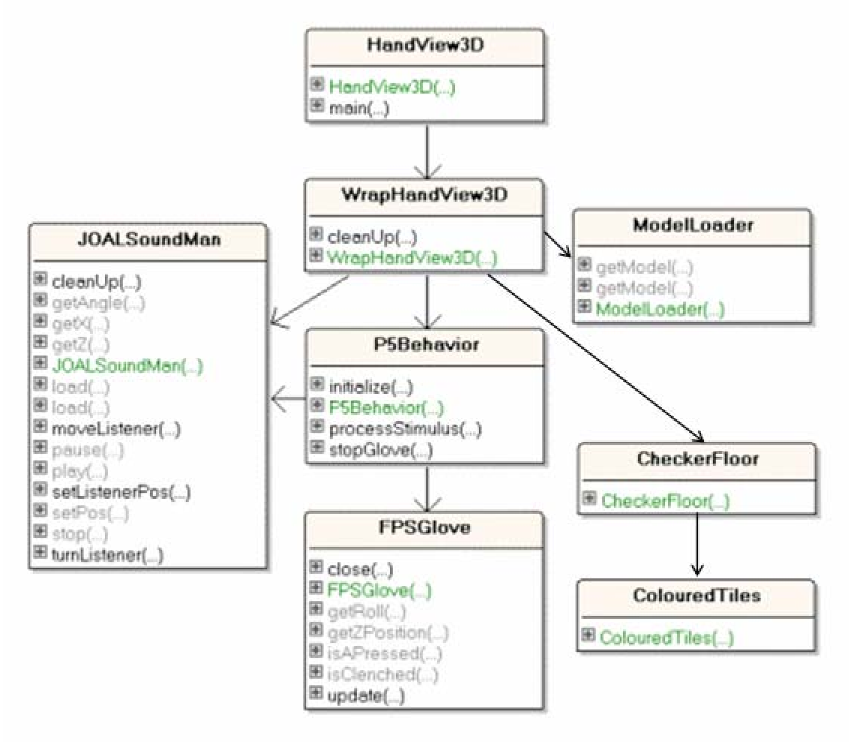


Figure 11. Class Diagrams for HandView3D.

HandView3D creates a JFrame holding a WrapHandView3D JPanel where the scene is rendered. HandView3D andWrapHandView3D are very similar to the ObjView3D and WrapObjView3D classes in chapter 7 ??. The ModelLoader, CheckFloor, and ColouredTiles classes are unchanged.

JOALSoundMan (from chapter 13 ??) manages the 3D sound; FPSGlove is from the previous section.

All that's really new is P5Behavior, and several methods in WrapHandView3D which create the behaviour, utilize JOALSoundMan, and closes down the glove and sound production at termination time. I'll describe those in detail now.

7.1. The Glove Behaviour

P5Behavior is a time-driven behavior, triggered every DELAY ms.

```
public class P5Behavior extends ViewPlatformBehavior
{
  private static final int DELAY = 75;   // ms  (polling interval)
  // more constants go here

  private WakeupCondition wakeUpCond;
  private FPSGlove glove;

  // sound-related
  private JOALSoundMan soundMan;
  private String soundNm;    // name of the JOAL source

  // more globals go here

  public GamePadBehavior(JOALSoundMan sm, String nm)
  {
    soundMan = sm;
    soundNm = nm;

    glove = new FPSGlove();
    wakeUpCond = new WakeupOnElapsedTime(DELAY);
  } // end of GamePadBehavior()

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

  // more methods go here
} // end of P5Behavior
```

P5Behavior extends ViewPlatformBehavior so the scene graph's ViewPlatform's transform group, targetTG, is available. P5Behavior adjusts the player's camera position by applying translations and rotations to targetTG.

A reference to JOALSoundManager is required so the listener can be translated and rotated in unison with the camera. JOALSoundManager also switches on and off the cow’s music, which requires its name, soundNm.

Polling the Glove

When processStimulus() is triggered, it updates the glove's state with a call to FPSGlove.update(), then reads the current data to possibly translate and turn the camera, and toggle the cow’s music on/off.

```
public void processStimulus(Enumeration criteria)
{
  glove.update();  // update the glove settings

  // possibly translate, turn, and toggle the sound
  translate();
  turn();
  toggleSound();

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

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

Translating the Camera

The z-axis position can be either FPSGlove.NEAR, FPSGlove.FAR, or FPSGlove.MIDDLE. The first two make the camera move forwards or backwards.

```
// globals
private static final double MOVE_STEP = 0.2;
```

```
// hardwired movement vectors
private static final Vector3d FWD = new Vector3d(0,0,-MOVE_STEP);
private static final Vector3d BACK = new Vector3d(0,0,MOVE_STEP);


private void translate()
{
  int handPos = glove.getZPosition();
  switch(handPos) {
    case FPSGlove.NEAR: doMove(FWD); break;
    case FPSGlove.FAR: doMove(BACK); break;
    case FPSGlove.MIDDLE: break;  // do nothing
    default: System.out.println("pos?"); break;  // shouldn't happen
  }
}  // end of translate()
```

doMove() applies a translation to the targetTG TransformGroup, and to the JOAL listener.

```
// globals for repeated calcs
private Transform3D t3d = new Transform3D();
private Transform3D toMove = new Transform3D();
private Vector3d trans = new Vector3d();


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

  // reposition the JOAL listener
  t3d.get(trans);
  soundMan.setListenerPos((float)trans.x, (float)trans.z);
} // end of doMove()
```

P5Behavior doesn't support translations left, right, up, or down. They wouldn't be hard to add, by mapping horizontal and vertical glove movements to camera moves.

Turning the Camera

The roll constant can be either FPSGlove.ROLL_LEFT, FPSGlove.ROLL_RIGHT, or FPSGlove.LEVEL. The first two trigger a camera rotation around the y-axis.

```
// global
private static final double ROT_AMT = Math.PI / 36.0;   // 5 degrees


private void turn()
{
  int handOrient = glove.getRoll();
  switch(handOrient) {
    case FPSGlove.ROLL_LEFT: rotateY(ROT_AMT); break;   // turn left
    case FPSGlove.ROLL_RIGHT: rotateY(-ROT_AMT); break; // turn right
    case FPSGlove.LEVEL: break;  // do nothing
    default: System.out.println("rot?"); break;  // shouldn't happen
  }
}  // end of turn()
```

rotateY() applies a rotation to the targetTG TransformGroup, and to the JOAL listener.

```
// global for repeated calcs
private Transform3D toRot = new Transform3D();

private void rotateY(double radians)
{
  targetTG.getTransform(t3d);
  toRot.rotY(radians);
  t3d.mul(toRot);
  targetTG.setTransform(t3d);

  // rotate the JOAL listener
  soundMan.turnListener((int) Math.toDegrees(radians));
} // end of rotateY()
```

Toggling the Sound

When the user clenches their fist, or presses the “A” button, the cow music is paused (if it’s currently playing), or resumed (if it’s paused).

The simple implementation (and the **wrong** one) would be to call FPSGlove.isClenched() and FPSGlove.isAPressed() and check whether either was true in order to toggle the sound. A code fragment that does this:

```
// global
private boolean soundPlaying = true;


private void toggleSound()
// this is WRONG
{
  if (glove.isClenched()||glove.isAPressed()) {
    if (soundPlaying)   // play --> paused
      soundMan.pause(soundNm);
    else  // paused --> resumed
      soundMan.play(soundNm);
    soundPlaying = !soundPlaying;
  }
}  // end of toggleSound() -- WRONG
```

The code is incorrect because toggleSound() is called every time that processStimulus() is called (every 75 ms). Unfortunately, a user can’t clench and unclench their fist in that short an interval, or easily press and release a button. This means that a playing sound would be paused on one call to processStimulus(), but the next call (75 ms later) would very likely detect that the hand was still closed (or the button still pressed), and resume the music. In other words, the music would resume just 75 ms after being paused.

The solution is to impose a *minimum time interval* between soundPlaying changes. For example, a user can probably clench and unclench their hand in a second.

Therefore, a modification to the soundPlaying state shouldn't be allowed until at least a second after it was last changed.

A counter is the simplest way of coding the time interval. It's started after a state change, and incremented each time that processStimulus() is woken up. Another state change is only permitted after the counter has reached a certain value. This approach is used in the correct version of toggleSound():

```
// globals
private static final int TOGGLE_MAX = 14;
      // toggle count at which soundPlaying can be toggled

private boolean soundPlaying = true;
private int toggleCounter = 0;


private void toggleSound()
// this is CORRECT
{
  if (toggleCounter < TOGGLE_MAX)
    toggleCounter++;

  if ((glove.isClenched()||glove.isAPressed()) &&
      (toggleCounter == TOGGLE_MAX)) {
    toggleCounter = 0;  // reset

    if (soundPlaying)   // play --> paused
      soundMan.pause(soundNm);
    else  // paused --> resumed
      soundMan.play(soundNm);
    soundPlaying = !soundPlaying;
  }
}  // end of toggleSound() -- CORRECT
```

The music is toggled on/off if the glove is clenched or button A is pressed, but only if toggleCounter has reached TOGGLE_MAX. This imposes an interval between state changes equal to TOGGLE_MAX*DELAY, which is about 1 second (14*75 ms).

7.2. Adding a Musical Cow

WrapHandView3D adds a cow to the scene with the help of ModelLoader from chapter 7 ??. JOALSoundMan positions a sound at the same spot, and starts it playing.

```
// in WrapHandView3D
// globals
private static final String COW_SND = "spacemusic";
private BranchGroup sceneBG;


private void addCow()
{
  ModelLoader ml = new ModelLoader();
  Transform3D t3d = new Transform3D();

  // a cow model
  t3d.setTranslation( new Vector3d(-2,0,-4));   // move
  t3d.setScale(0.7); // shrink
  TransformGroup tg1 = new TransformGroup(t3d);
  tg1.addChild( ml.getModel("cow.obj", 1.3) );
  sceneBG.addChild(tg1);

  if (!soundMan.load(COW_SND, -2, 0, -4, true))
    System.out.println("Could not load " + COW_SND);
  else
    soundMan.play(COW_SND);
}  // end of addCow()
```

Both the model and music are placed at (-2, 0, 4). The music is set to play repeatedly.

7.3. Initialising the Viewpoint

WrapHandView3D positions the camera and JOAL lisener at the same spot, and connects the P5 behavior object to the camera's viewpoint.

```
// globals
private final static double Z_START = 9.0;

private P5Behavior p5Beh;
    // moves/rotates the viewpoint with P5 glove input


private void createUserControls()
{
  ViewingPlatform vp = su.getViewingPlatform();

  // position viewpoint
  TransformGroup targetTG = vp.getViewPlatformTransform();
  Transform3D t3d = new Transform3D();
  targetTG.getTransform(t3d);
  t3d.setTranslation( new Vector3d(0,1,Z_START));
  targetTG.setTransform(t3d);

  // position JOAL listener at same spot as camera
  soundMan.moveListener(0, (float)Z_START);

  // set up P5 glove controls to move the viewpoint
  p5Beh = new P5Behavior(soundMan, COW_SND);
  p5Beh.setSchedulingBounds(bounds);
  vp.setViewPlatformBehavior(p5Beh);
} // end of createUserControls()
```

The viewpoint and listener start at the same (x, z) coordinate, (0, Z_START).

P5Behavior is given a reference to the JOALSoundMan and the name of the cow music, so it can move the listener and viewpoint together, and pause/resume the music.

7.4. Cleaning Up

When the user presses HandView3D’s close box, there’s some cleaning up needed before termination. Good housekeeping dictates that the JOAL sources and buffers (managed by JOALSoundMan) should be deleted, and that the P5 glove link should be closed.

HandView3D calls cleanUp() in WrapHandView3D when a window closing event is detected.

```
// in WrapHandView3D
// globals
private P5Behavior p5Beh;
private JOALSoundMan soundMan;

public void cleanUp()
{ p5Beh.stopGlove();
  soundMan.cleanUp();
}
```

cleanUp() calls stopGlove() in P5Behavior:

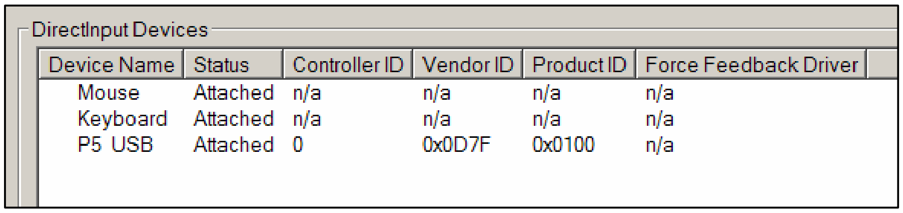
```
// in P5Behavior
// global
private FPSGlove glove;

public void stopGlove()
{ glove.close();
  setEnable(false);
}
```

The call to setEnable() switches off the behavior so it can't be triggered again.

8. The P5 Glove and JInput

If you've been reading these chapters in order, you may be wondering if JInput could be utilized with the P5 glove. (JInput is a Java API for the discovery and polling of input devices, described in chapters 11 and 12 ??)

When the P5 is plugged into the PC's USB port, DirectInput sees it as a USB device, as shown by dxdiag in Figure 12.



DirectInput Devices

Device Name	Status	Controller ID	Vendor ID	Product ID	Force Feedback Driver
Mouse	Attached	n/a	n/a	n/a	n/a
Keyboard	Attached	n/a	n/a	n/a	n/a
P5 USB	Attached	0	0x0D7F	0x0100	n/a

Figure 12. Part of the dxdiag Input Tab.

The ListControllers application from chapter 11 ?? supplies slightly more information:

```
> runJI ListControllers
Executing ListControllers with JInput...
JInput version: 2.0.0-b01
```

```
0. Mouse, Mouse
1. Keyboard, Keyboard
2. P5  USB, Stick
```

The P5 controller is classified as a joystick.

Details about the controller are available by calling ControllerDetails (also from chapter 11 ??):

```
> runJI ControllerDetails 2
Executing ControllerDetails with JInput...
Details for: P5  USB, Stick, Unknown
No Components
No Rumblers
No subcontrollers
```

This is disappointing, and due to the P5 not employing a DirectInput driver. Instead, it relies on generic USB support in Windows.

To be fair to JInput, it is capable of 'seeing' all of the P5's components, but requires someone to write a JInput plug-in for the glove. According to JInput experts, most of the code needed for such a plug-in is already available in Carl Kenner's API.