Chapter 6.5. Shapely Applications

A lot rests on appearances, and the first thing a user sees is the application window. Java allows the look-and-feel of an application to be varied, but you're still stuck with the same old rectangular frame, with iconify, maximize/minimize and close buttons in a title bar. The ImagePane application shown in Figure 1 is typical of this same-old-look-and-feel.

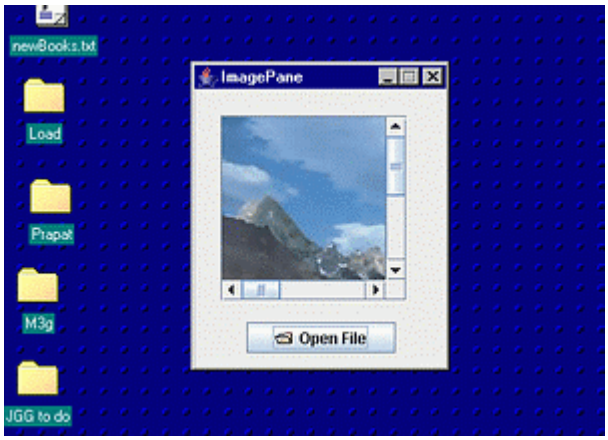

Figure 1. The ImagePane Application.

In this chapter we throw away the rectangular look, recasting an application into any shape you like. For example, Figure 2 shows the "hand held TV" look-and-feel for ImagePane, and Figure 3 uses a "hand mirror".

Figure 2. ImagePane Dressed like a TV.

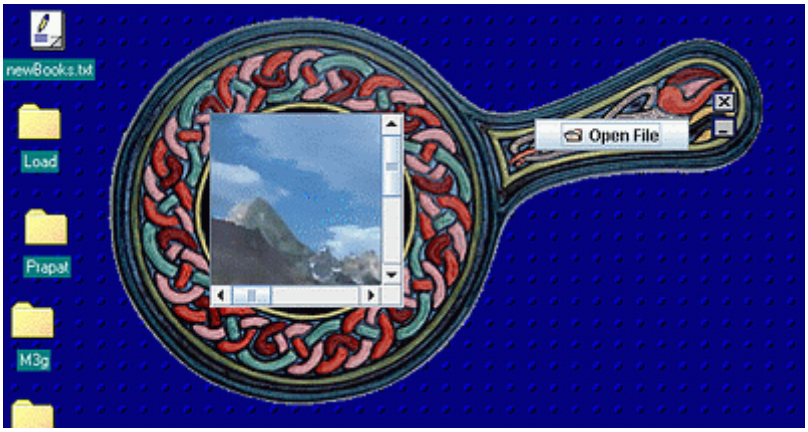

Figure 3. ImagePane Dressed like a Hand Mirror.

We aim to deceive – the application underneath is the same in all three figures. The user clicks on the "Open File" button to load a GIF or JPEG file into the scrollable pane.

A 'shapely' window (my name, and I'm sticking to it) has iconify and close 'icons', but no ability to minimize or maximize. The window can be moved around by the user dragging any part of the visible frame. A shapely window is a non-rectangular window.

This chapter describes the development of the 'hand mirror' version of the image viewer, called ShapelyImagePane. However, the coding approach can easily handle different appearances and applications. For example, the frame's shape can be changed from the hand mirror to the TV by modifying just a few lines of code. Changing the application means changing the GUI elements, which are located in a single JPanel subclass.

1. The Implementation Approach

The application is an undecorated JFrame containing a three-layer JLayeredPane component. In back-to-front order, the layers are:

1. ScreenPanel: this JPanel shows a picture of the screen *behind* the JFrame, which is updated as the JFrame moves, creating the illusion of transparency.
2. BackgroundPanel: this JPanel contains the transparent GIF representing the application's 'frame'. It includes 'hot spots' which are areas where the user can click to quit or iconify the application. Other hot spots can be easily added.
3. GUIPanel: this JPanel contains the GUI components for the application. In this example, the user clicks on a JButton to open a JFileChooser to select an image to display in a JScrollPane object.

The structure of a typical shapely application is shown in Figure 4.

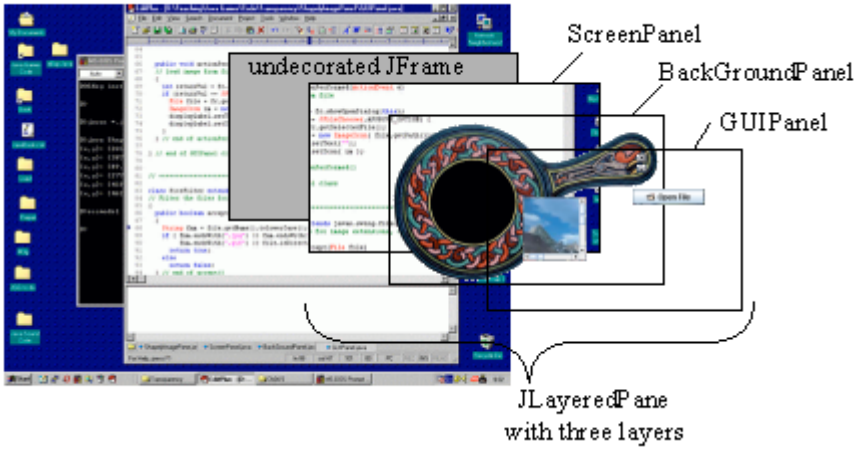

Figure 5. The Components of a Shapely Application.

The transparency around the irregular-shaped window is really a screenshot fragment displayed by ScreenPanel. ScreenPanel takes a screenshot of the entire desktop, and displays the part of it corresponding to the region behind the application. The rectangle is regenerated every time the application moves.

Each time a new screenshot is taken, there's a discernable flicker (which I'll explain in detail later), and the screen-sized image takes a short, but just about discernable, amount of time to create. Consequently, we want to minimize the picture taking frequency. We've chosen to only take a new screenshot when the application is de-iconified, or brought to the front after being behind another application. In other words, the desktop image is only refreshed when the application gains focus from a different application.

This 'transparency' approach has a number of drawbacks:

1. As a user drags the shapely application, ScreenPanel must extract a rectangular region from the screenshot, and render it. This extraction and rendering is somewhat slow (depending on the rectangle's size), causing a short 'time lag' in the updating of the transparent areas.
2. If the desktop is changing behind the shapely application, then the changes aren't shown in its 'transparent' parts. Desktop changes include text moving in another application's window, or a folder or icon being opened. Since these changes are carried out by the OS or other applications, our shapely application knows nothing about them, and its screenshot remains the one from when it last gained focus.
3. You can't click on something visible through the 'transparent' part of the shapely application (such as a desktop icon). The click is really going to ScreenPanel, so will cause the shapely application to come into focus.
4. If you want to download a shapely application using Java Web Start, then you'll have to sign it. The reasoning behind this is that an application which takes screenshots could be used for spying! My thanks to oNyx for pointing this out at http://www.javagaming.org/cgi-bin/JGNetForums/YaBB.cgi?board=2D;action=display;num=1101719607. Details on Java Web Start can be found in Appendix B.

Problem (2) suggests that the screenshot should be updated more frequently. The simplest way of achieving this is to take a new screen picture at regular, fixed intervals, governed by a TimerTask. This raises the question of the best frequency for the updates, since there may be long periods when the desktop isn't changing. Also, more frequent screen updates increase the flicker and image-generation overheads.

Our view is that the transparent parts behind the shapely application will usually not be in front of some rapidly changing desktop activity, and so there's no need to schedule periodic updates.

2. Class Diagrams for ShapelyImagePane

Figure 6 shows the class diagrams for all the classes in the ShapelyImagePane application. Only the public and protected methods are shown.

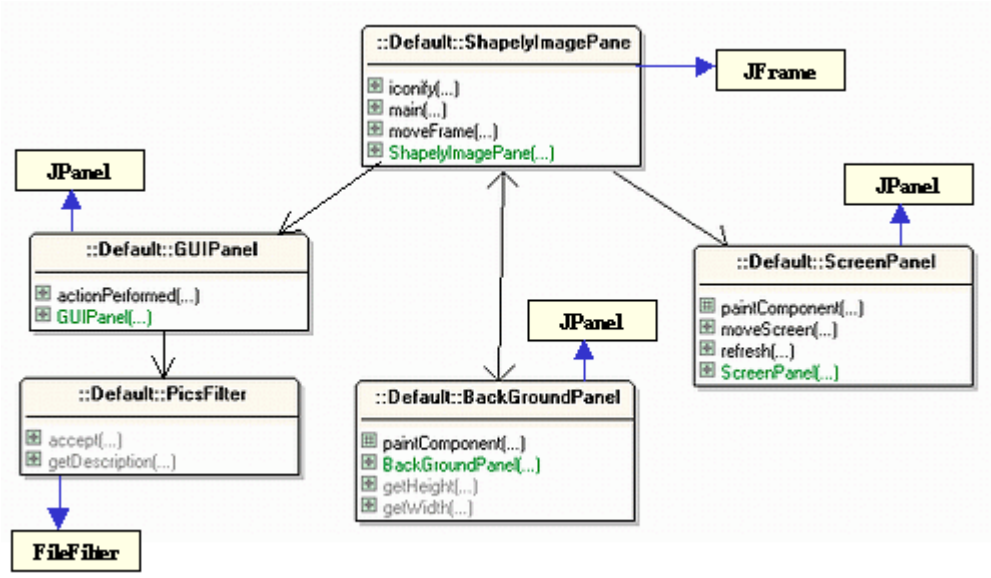

Figure 6. Class Diagrams for ShapelyImagePane.

ShapelyImagePane is the top-level undecorated JFrame, which sets up a JLayeredPane object containing the ScreenPanel, BackgroundPanel, and GUIPanel objects. GUIPanel using a PicsFilter object to limit the choice of files in its JFileChooser to those with image extensions.

3. The ShapelyImagePane Class

The JFrame is undecorated (i.e. no title bar or borders), and we position it in the middle of the screen. The sizes of the JFrame and all its panels are set to be the same as the dimensions of the transparent GIF used by the BackgroundPanel object. These

dimension will remain fixed throughout execution, and so the frame's minimization and maximization capabilities are switched off.

```
// Global constants and variables
 private static final String BACK_FNM = "handMirror.gif";
            // the transparent GIF used by BackgroundPanel

private int xScr, yScr;            // window's location on screen
private boolean justInitialised;

private ScreenPanel scrPanel;        // for drawing the screen
private BackGroundPanel backPanel;   // for the background image


public ShapelyImagePane()
{
  super("ShapelyImagePane");

  setUndecorated(true);  // no title bar, borders
  setResizable(false);   // cannot be minimized/maximized

  // create the background panel
  BackGroundPanel backPanel = new BackGroundPanel(BACK_FNM, this);
  int bgWidth = backPanel.getWidth();
  int bgHeight = backPanel.getHeight();

  // center the JFrame on screen, and set its size
  Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
  xScr = (d.width - bgWidth)/2;
  yScr = (d.height - bgHeight)/2;
  setLocation(xScr, yScr);
  setSize(bgWidth, bgHeight);    // same size as background image

  // initialise screen panel
  scrPanel = new ScreenPanel(bgWidth, bgHeight, xScr, yScr);

  // initialise GUI panel
  GUIPanel guiPanel = new GUIPanel(bgWidth, bgHeight);

  // put the parts together in layers
  JLayeredPane lPane = new JLayeredPane();
  lPane.add(scrPanel, new Integer(1));
  lPane.add(backPanel, new Integer(2));
  lPane.add(guiPanel, new Integer(3));

  // add the layers to the JFrame
  Container c = getContentPane();
  c.setLayout( new BorderLayout() );
  c.add(lPane, BorderLayout.CENTER);

  justInitialised = true;
  setFocusListening();

  setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  setVisible(true);
} // end of ShapelyImagePane()
```


3.1. Focus Control

A new screenshot is obtained when the focus changes from *another* application to this one. Focus changes within the application are ignored, such as when the user clicks on the background after pressing the "Open File" button. There's no need to regenerate the screen image in those cases since the focus change has not affect on the desktop image.

setFocusListening() sets up the focus listener:

```
// global constants and variables
private static final int FOCUS_INTERVAL = 500;  // 0.5 second

private long focusGainedTime = 0;


private void setFocusListening()
{
  addWindowFocusListener( new WindowAdapter()
  {
    public void windowGainedFocus(WindowEvent evt)
    {
      if (justInitialised) {      // allows us to ignore the focus
        justInitialised = false;  // gained at the start
        return;
      }
      long focusTime = System.currentTimeMillis();
      if ((focusTime - focusGainedTime) > FOCUS_INTERVAL) {
        focusGainedTime = focusTime;
        Window oppWin = evt.getOppositeWindow();
        if (oppWin == null)   // if focus came from another app
          refresh();  // get a new screenshot
      }
    }
  });
} // end of setFocusListening
```

The WindowEvent object, evt, is examined with getOppositeWindow() which returns a reference to the window that had the focus previously. This reference will be null if the window was an application outside the control of the JVM.

The justInitialised boolean is used to disable a call to refresh() when the application is first started, since the ScreenPanel object will already have a screenshot.

The checking of focus times is used to avoid a nasty problem with the code. refresh(), which we'll look at in a moment, triggers a new application-changed focus event as part of its processing. Without the timing test, this new event would cause refresh() to be called again. An infinite calling sequence would ensue: refresh() would trigger a call to refresh(), would trigger a call to refresh(), and so on. The timing code ignores a focus event which arrives quickly after the previous one, and so the infinite chain of calls never gets going.

refresh() asks the ScreenPanel object to create a new screenshot, and then repaints the application at the current screen position (xScr, yScr).

```
private void refresh()
```



```
{
  setVisible(false);   // hide ourselves

  try { Thread.sleep(30); }         //this short delay ensures that
  catch(InterruptedException ex) {}  // we've disappeared before...

  scrPanel.refresh(xScr, yScr);   // take a new snap

  setVisible(true);    // show ourselves
  setLocation(xScr, yScr);
  repaint();   // this updates the screenPanel, and anything else
}  // end of refresh()
```

The catch with taking a screenshot is that it shouldn't include the shapely application. Rectangles will be cut from the screenshot as the application is moved about, and we don't want the application to appear in them.

This requirement means that the scrPanel.refresh() call, must be bracketed by calls to setVisible(). Unfortunately, setVisible(false) causes the application to lose focus, and the subsequent setVisible(true) brings it back again. This triggers an application-changed focus event, which could potentially cause an infinite series of calls to refresh(), as mentioned above.

Another drawback with setVisible(false) is that the application may not be totally invisible before the screenshot is taken. This manifests itself as a gray border or other artifact on the screenshot, which varies with each screen snap. Our solution is to sleep for a short time before the call to scrPanel.refresh(), to give the application time to disappear.

Unfortunately, the sleep period lengthens the period of invisibility, which appears to the user as an irritating 'flicker'. This is one reason why we chose to regenerate the screenshot during focus changes. The flicker isn't so noticeable when the application is being de-iconified, or coming to the front.

The use of sleep() is a bit hacky, and the actual amount of time required for the application to disappear will vary depending on the hardware. I arrived at 30 ms by testing the code on an old, slow Win 9x machine.

3.2. Application-level Operations

ShapelyImagePane is the home to two application-level operations: moveFrame() moves the frame, while iconify() iconifies it.

The frame moves when the mouse is dragged inside the background image. BackGroundPanel detects the dragging, and sends the new screen coordinates to ShapelyImagePane's moveFrame(). It updates the frame's position, and informs ScreenPanel of the move. ScreenPanel will then have to display a new screen sub-rectangle.

```
public void moveFrame(int x, int y)
{ xScr = x; yScr = y;
  scrPanel.moveScreen(xScr, yScr);
  setLocation(xScr, yScr);
  repaint();    // this updates the screenPanel, and anything else
```



```
}
```

iconify() is called by BackGroundPanel when the user clicks inside its iconify 'hot spot'.

```
public void iconify()
{ int state = getExtendedState();
  state |= Frame.ICONIFIED;      // set the iconified bit
  setExtendedState(state);       // iconify the frame
}
```

4. The ScreenPanel Class

ScreenPanel shows a picture of the screen behind the application's JFrame, which is updated as the application moves. A snap is taken of the entire screen, but only the relevant rectangle is shown.

There are two public methods: refresh() and moveScreen(). refresh() takes a screenshot and uses moveScreen() to show the required rectangle. refresh() is only called when the application gains focus, since image regeneration is slow, and accompanied by a flicker.

The constructor sets up the JPanel's coordinates and dimensions (which are the same as the background image's). It calls refresh() to take the first snapshot.

```
// globals
private int bgWidth, bgHeight;   // of background image

private int scrWidth, scrHeight;  // of screen
private Rectangle scrRect;   // for the screen

private Robot robot = null;


public ScreenPanel(int bgW, int bgH, int xScr, int yScr)
{
  bgWidth = bgW; bgHeight = bgH;
  setBounds(0, 0, bgWidth, bgHeight);

  // get screen dimensions and rectangle
  Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
  scrWidth = d.width;  scrHeight = d.height;
  scrRect = new Rectangle(0,0, scrWidth, scrHeight);

  try {
    robot = new Robot();
  }
  catch(Exception e)
  { System.out.println("Error with robot"); }

  refresh(xScr, yScr);
}
```


The java.awt.Robot class has a screenshot method, used in refresh(). It needs a rectangle specifying the capture area, which is created in the constructor, and stored in scrRect.

```
// global
private BufferedImage scrIm = null;   // stores entire screen image

public void refresh(int xScr, int yScr)
{ scrIm = robot.createScreenCapture(scrRect);    // take a snap
  moveScreen(xScr, yScr);
}  // end of refresh()
```

moveScreen()'s code is a little tricky because of the need to deal with boundary cases, when part of the shapely application is off the edge of the desktop, either at the bottom, top, left, or right sides. The screen rectangle will be different in those situations, since there's no desktop image outside the boundaries of the screen.

The screen rectangle requires four values: a top-left coordinate (x,y), a width (w), and height (h). We also need to calculate a coordinate in the JPanel (xDraw, yDraw), where the top-left corner of the screen rectangle will be placed.

```
// where the image is drawn in this panel
private int xDraw = 0;
private int yDraw = 0;
private BufferedImage subIm = null;   // sub-region being displayed


public void moveScreen(int xScr, int yScr)
{
  xDraw = 0;         // default drawing position in this JPanel
  yDraw = 0;

  int x = xScr;      // default (x,y) for screen sub-image
  int y = yScr;
  int w = bgWidth;   // default width and height of sub-image
  int h = bgHeight;

  if (xScr > (scrWidth-bgWidth))  // panel RHS is off screen
    w = scrWidth - xScr;
  else if (xScr < 0) {   // panel LHS is off screen
    xDraw = -xScr;
    x = 0;
    w = scrWidth - xDraw;
  }

  if (yScr > (scrHeight-bgHeight))  // panel bottom is off screen
    h = scrHeight - yScr;
  else if (yScr < 0) {   // panel top is off screen
    yDraw = -yScr;
    y = 0;
    h = scrHeight - yDraw;
  }

  subIm =  scrIm.getSubimage(x, y, w, h);
}  // end of moveScreen()
```

The four boundary cases are handled by the if-tests, which modify the default settings for (xDraw,yDraw), (x,y), w, and h. They are illustrated by Figures 7 to10 below.

Figure 7 shows the case when the application is partially off the right hand side of the screen.

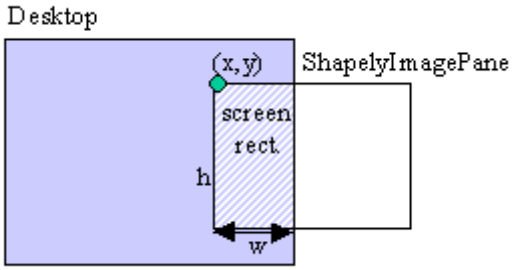

Figure 7. Application Partially off the Right Hand Side.

(xDraw, yDraw) will the origin of the JPanel, (x,y) will be the current screen coordinates (xScr,yScr), h will be the height of the application, and w will be truncated to extend only to the right edge of the screen.

Figure 8 illustrates the situation when the application is partly off the left hand side of the screen.

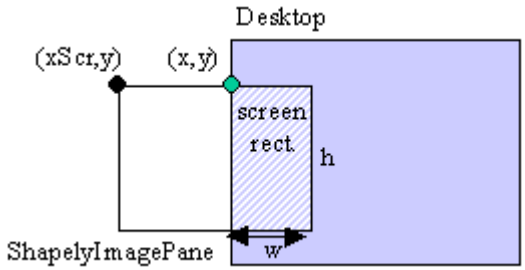

Figure 8. Application Partially off the Left Hand Side.

In this case, xDraw will be -xScr, since the origin of the panel is at the x-coordinate xScr (which will have a negative value). The x-coordinate for the screen image will need to be changed to 0, since that's as far left as the screenshot extends. The width will need to be reduced as well.

Figure 9 shows the application dropping off the bottom of the screen.

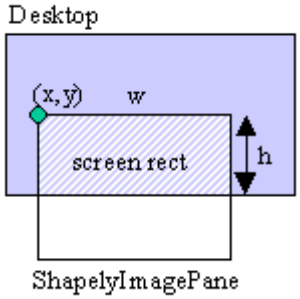

Figure 9. Application Partially off the Bottom of the Screen.

This boundary case only requires an adjustment to the height of the screen rectangle.

Figure 10 has the application moving off the top of the screen.

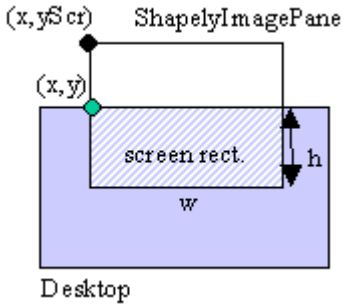

Figure 10. Application Partially off the Top of the Screen.

The yDraw value is set to -yScr, since the origin of the panel is at the y-coordinate yScr (which will have a negative value). The y-coordinate for the screen image will need to be changed to 0, since that's as far upwards as the screenshot extends. The height will need to be reduced as well.

After all these calculations, rendering is just a matter of drawing the screen rectangle stored in subIm at (xDraw,yDraw) in the panel.

```
protected void paintComponent(Graphics g)
{ super.paintComponent(g);
  g.drawImage(subIm, xDraw, yDraw, null);
}
```


5. The BackGroundPanel Class

The BackGroundPanel's main tasks are:

- to load and render the transparent GIF that acts as the application's background;
- to respond to mouse drags by causing the application to move;
- to respond to clicks in 'hot spots' (special areas on the graphic). Typically there are two hot-spots, for the iconify and close icons.

The hand mirror image, which is used by the shapely application in Figure 3, is stored in handMirror.gif as shown in Figure 11.

Figure 11. The Hand Mirror GIF Image.

The grey and white square indicate transparency. The close and iconify icons are at the end of the mirror's handle.

The constructor loads the image, uses it's dimensions to set the panel's size, and sets up the hot spots and mouse listeners.

```
// globals
private ShapelyImagePane top;      // so can move and iconify JFrame
private int scrWidth, scrHeight;   // screen size
private int imWidth, imHeight;     // image size
private boolean usingHotSpots = true;   // are we using hotspots?


public BackGroundPanel(String backFnm, ShapelyImagePane p)
{
  top = p;

  setOpaque(false);    // invisible JPanel
  setFocusable(false);

  // get screen dimensions
  Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
  scrWidth = d.width;  scrHeight = d.height;

  loadImage(backFnm);
  setBounds(0, 0, imWidth, imHeight);   //JPanel size == image size

  if (usingHotSpots)
    initHotSpots();
```



```
    setMouseListening();
    setMouseMotionListening();
  }  // end of BackGroundPanel()
```

The usingHotSpots boolean is set to false when a new application is being developed, a topic I discuss at the end of the chapter.

5.1. Loading and Rendering the Background Image

The loading of the image is done in a slightly non-standard way.

```
// global background image info
private BufferedImage backIm = null;
private int imTransparency;    // transparency characteristics


private void loadImage(String fnm)
// fnm contains the transparent GIF used as the appl.'s background
{
  try {
    backIm =  ImageIO.read(getClass().getResource(fnm));
  }
  catch(IOException e) {
    System.out.println("Load Image error for " + fnm + ":\n" + e);
  }
  imWidth = backIm.getWidth();
  imHeight = backIm.getHeight();
  imTransparency = backIm.getColorModel().getTransparency();
}
```

By loading the image as a BufferedImage object, it allows us to easily check for transparent pixels in isTransparent() (see below), and also do a quick test by examining it's transparent colour model, which will be one of `Transparency.BITMASK`, `Transparency.OPAQUE`, or `Transparency.TRANSLUCENT`. If its value is `OPAQUE`, then there's no need to check any pixels in isTransparent(). The BufferedImage class is discussed in Chapter 4.

Rendering the image is a lot easier than the similar task in ScreenPanel, since the image doesn't change as the application moves.

```
protected void paintComponent(Graphics g)
{ super.paintComponent(g);
  g.drawImage(backIm, 0, 0, null);
}
```

5.2. Listening for Mouse Presses

The listening for mouse presses is set up in setMouseListening(). The processing done by the listener reflects the two ways a press may be used: as a precursor to dragging the application, or as a click in a hot spot.

```
// globals
```



```
private int xScr, yScr;     // application location on screen
private int xPosn, yPosn;   //(x,y) inside appl. when mouse pressed
private boolean isDraggable = false;


private void setMouseListening()
{
  addMouseListener(new MouseAdapter()
  {
    public void mousePressed(MouseEvent e)
    { Point p = getLocationOnScreen();
      xScr = p.x; yScr = p.y;             // store screen coords
      xPosn = e.getX(); yPosn = e.getY();   // get JPanel coords

      // System.out.println("(x,y): (" + xPosn + ", "+yPosn + ")");
      /* You can use this println for deciding where GUI
         components and hot spots should be placed. */

      if (usingHotSpots)
        processHotSpots(xPosn, yPosn);

      isDraggable = !isTransparent(xPosn,yPosn);
       // isDraggable is true only if user clicked in opaque area
    }
   });
}  // end of setMouseListening()
```

Both the screen and the local JPanel coordinates of the key press are stored.

The main problem in the early stages of an application's design is deciding *where* GUI components and hot spots should be located on the background graphic. The println() (once uncommented) allows the user to click on interesting points in the background graphic and note their locations.

isTransparent() decides if the panel coordinate is a transparent pixel in the background image. It means that the user has clicked on an 'transparent' part of the application, and so dragging should be disabled.

```
private boolean isTransparent(int x, int y)
// is (x,y) in a transparent part of the background image?
{
  if (backIm == null)
    return true;   // since no image!

  if (imTransparency == Transparency.OPAQUE)
    return false;   // since none of the image is transparent

  // range checking for (x,y)
  if ((x > imWidth) || (y > imHeight))
    return true;   // since beyond extent of image

  int pixel = backIm.getRGB(x,y);
  int alphaVal = (pixel >> 24) & 255;    //extract alpha from pixel
  if (alphaVal == 0)  // it's transparent
    return true;
  else
    return false;
}  // end of isTransparent()
```


The usefulness of storing the graphic in a BufferedImage object comes into play in isTransparent(), since we can employ BufferedImage.getRGB() to extract the pixel's colour. The colour is stored in ARGB form as a 32 bit integer (if the image has a transparent component), and the alpha byte can be extracted by a simple piece of bit shifting and masking. More details about ARGB can be found in chapter 4.

The use of an isDraggable boolean allows us to disable mouse dragging when the mouse press is on a transparent pixel in the image. Unfortunately, there's no way to stop the mouse press from triggering focus change. This is problem (3) mentioned at the start of the chapter: a user can think they are selecting something on the desktop, but the shapely application will come to the foreground instead.

5.3. Mouse Dragging

The listener for mouse drags is set up in setMouseMotionListening(). The code calculates a new screen coordinate (xScr, yScr) in terms of the offset from the starting mouse press position.

```
private void setMouseMotionListening()
{
  addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e)
    {
      if (isDraggable && !isIconified()) {
        xScr += (e.getX() - xPosn);
              // add change in posn to the screen coords
        yScr += (e.getY() - yPosn);
        top.moveFrame(xScr, yScr);    // move to new screen posn
      }
     }
   });
}
```

The new coordinate is passed to moveFrame() in ShapelyImagePane, where it causes the application to move, and ScreenPanel to draw a new screen rectangle (but not take a new screenshot).

The isIconified() test in mouseDragged() stops an attempt to drag the window when it's iconified. This seemingly impossible case was brought to my attention by Andres Vera: it occurs when the user presses on the iconification hot spot *and* drags the mouse at the same time. The application is iconified, and then the mouse drag event is processed.

isIconified() is defined as:

```
private boolean isIconified()
{ return ((top.getExtendedState() & JFrame.ICONIFIED) ==
                                          JFrame.ICONIFIED);
}
```


5.4. Hot Spots

Hot spots are rectangular areas on the background graphic which trigger processing when the user clicks inside them. The areas are stored as Rectangle objects in an array, set up by initHotSpots().

```
// globals
private static final int NUM_HOT_SPOTS = 2;

private Rectangle[] spotLocs;
private String[] spotNames;

private void initHotSpots()
{
  spotLocs = new Rectangle[NUM_HOT_SPOTS];
  spotLocs[0] = new Rectangle(456, 70, 15, 15);  // quit
  spotLocs[1] = new Rectangle(456, 89, 15, 15);  // iconify

  spotNames = new String[NUM_HOT_SPOTS];
  spotNames[0] = new String("quit");
  spotNames[1] = new String("iconify");
     // spotNames[] is used for debugging purposes
}
```

The numbers for the hot spot rectangles were obtained by clicking on the graphic and noting down the coordinates. This requires the (xPosn, yPosn) println() call in mousePressed() to be uncommented.

Although we only define two hot spots in initHotSpots(), there's no restriction on adding more.

The spotNames[] array is used in processHotSpots() during debugging, when we want to report on which hot spot has been selected.

```
private void processHotSpots(int xPosn, int yPosn)
// determine if the mouse click was in a hot spot, and respond
{
  int i;
  for(i=0; i < NUM_HOT_SPOTS; i++)
    if (spotLocs[i].contains(xPosn, yPosn))
      break;

  // System.out.println("Clicked on hot spot: " + spotNames[i]);
      // this println is useful for debugging the hotspots

  if (i < NUM_HOT_SPOTS) {    // clicked in a hot spot
    switch (i) {
      case 0: System.exit(0); break;
      case 1: top.iconify(); break;
      default: System.out.println("Unknown hot spot: " + i); break;
    }
  }
}
```

Hot spots processing is normally carried out by the top-level JFrame, or perhaps by the GUIPanel. Exiting the application is easy enough to do in processHotSpots(), but

iconification is handled by ShapelyImagePane since it requires changes to the JFrame's state bits.

6. The GUIPanel Class

GUIPanel contains the GUI components which trigger application processing. When we create a new shapely application, the top-level JFrame, ScreenPanel and BackGroundPanel change in quite simple ways (e.g. the definition of new hot spots), but GUIPanel's contents will alter drastically. Nevertheless, there are certain elements that won't change.

For example, GUIPanel always obtains its dimensions through its constructor, since they come from the background image. It will not use a layout manager, since the components will require very specific pixel placement. The panel itself will be transparent, so we can see the background image and 'transparent' edges.

These requirements mean that the GUIPanel constructor will have the following structure:

```
public GUIPanel(int bgImWidth, int bgImHeight)
{
  setLayout(null);                  // no layout manager
  setSize(bgImWidth, bgImHeight);   // same size as background image
  setOpaque(false);                 // the panel is invisible

  // setBorder(BorderFactory.createLineBorder(Color.white));
       // shows a white border which can be useful during debugging

  // create GUI components
  // position and size the components using setBounds()

  // set up listeners for the components

  // add the components to the JPanel using add()
}
```

The setBorder() call puts a white border around the edge of the application, which is useful during the development stages when the background image and GUI elements are still being designed and positioned.

The GUI components in ShapelyImagePane are a JButton linked to a JFileChooser, and a JScrollPane containing a JLabel. The JLabel is used as a convenient way of showing a text message when the application starts, and the loaded image thereafter.

The complete GUIPanel():

```
// globals
private final static String IMAGE_DIR = "images/";

// GUI elements
private JFileChooser fc;
private JLabel displayLabel;
```



```
  public GUIPanel(int bgImWidth, int bgImHeight)
  {
    setLayout(null);                  // no layout manager
    setSize(bgImWidth, bgImHeight);   // same size as background image
    setOpaque(false);                 // the panel is invisible

    setBorder(BorderFactory.createLineBorder(Color.white));
         // shows a white border which can be useful during debugging

    // create file chooser
    fc = new JFileChooser();
    fc.setFileFilter( new PicsFilter() );
    fc.setCurrentDirectory( new File(IMAGE_DIR) );

    // create open-file button
    JButton openButton =
        new JButton("Open File", new ImageIcon("openIcon.gif"));
    openButton.addActionListener(this);
    openButton.setBounds(321, 88, 115, 24);

    // label for showing loaded image
    displayLabel = new JLabel("Image appears here", JLabel.CENTER);

    // place inside a scrollPane
    JScrollPane jsp = new JScrollPane( displayLabel );
    jsp.setPreferredSize( new Dimension(144, 144) );
    jsp.setBounds(80, 84, 144, 144);

    add(openButton);
    add(jsp);
  }  // end of GUIPanel()
```

The setBounds() calls for the JButton and JScrollPane objects require a top-left coordinate for the component, a width, and height. We obtained these by clicking on the background image and noting down suitable values (after the (xPosn,yPosn) println() in mousePressed() in BackGroundPanel had been uncommented).

GUIPanel implements ActionListener, and so has an actionPerformed() method to deal with button presses:

```
  public void actionPerformed(ActionEvent e)
  // load image from file
  {
    int returnVal = fc.showOpenDialog(this);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
      File file = fc.getSelectedFile();
      ImageIcon im = new ImageIcon( file.getPath());
      displayLabel.setText("");
      displayLabel.setIcon( im );
    }
  }
```

The JFileChooser object uses a file filter to restrict the choice of files to those with JPG and GIF extensions

```
class PicsFilter extends javax.swing.filechooser.FileFilter
```



```
// Filter the files for image extensions, and allow sub-directories
{
  public boolean accept(File file)
  {
    String fnm = file.getName().toLowerCase();
    if ( fnm.endsWith(".jpg") || fnm.endsWith(".jpeg") ||
         fnm.endsWith(".gif") || file.isDirectory() )
      return true;
    else
      return false;
  } // end of accept()

  public String getDescription()
  { return "JPG and GIF Images"; }

} // end of PicsFilter class
```

The PicsFilter class can be found in GUIPanel.java.

6.1. Developing GUIPanel

I found it useful to implement the GUI interface and its processing in a more conventional JFrame window, so I didn't have to fiddle around with a background image and hot spot configuration.

A simple, conventional JFrame for GUIPanel:

```
public class ImagePane extends JFrame
{
  public ImagePane()
  {
     super( "ImagePane" );
     setSize(200, 240);    // dimensions for JFrame

     Container c = getContentPane();
     c.add( new GUIPanel( getWidth(), getHeight()) );
                  // reuse frame dimensions

     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     setResizable(false);
     setVisible(true);
  }


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

} // end of ImagePane class
```

The JFrame dimensions can be anything, but the same numbers should be supplied to the GUIPanel constructor. Resizing can be disabled, so the frame has the same fixed size as the top-level of the shapely application.

The GUI components inside GUIPanel must be positioned to fit into the confines of the JFrame. This means modifying the setBounds() calls for the JButton and JScrollPane. For example:


```
openButton.setBounds(40, 180, 115, 24);

jsp.setBounds(20, 20, 144, 144);
```

This creates the ImagePane layout shown in Figure 1.

7. Creating a New Shapely Application

A new shapely application is developed in fairly standard sequence of steps:

1. Develop the new application's GUIPanel inside a conventional JFrame. See the last section for an example JFrame subclass.
2. Create a transparent background image for the shapely application, and change the BACK_FNM constant in ShapelyImagePane to refer to it.
3. Switch off BackGroundPanel's hot spots by setting its usingHotSpots boolean to false. Switch on the printing of (xPosn,yPosn) coordinates by uncommenting the println() in mousePressed().
4. Comment out the addition of the GUIPanel layer to the JLayeredPane object in ShapelyImagePane's constructor. We don't want to involve GUIPanel just yet.
5. Start ShapelyImagePane, and click on the background image, collecting coordinate information (it'll appear on standard output). Use this information to determine the position of the hot spots and GUI components for steps (6) and (7). The application has no close icon, so has to be stopped with a ctrl-C or equivalent.
6. Set up the hotspot areas in initHotSpots() in BackGroundPanel, and change usingHotSpots back to true.
7. Fix the position of the GUI components by adjusting their setBounds() calls in GUIPanel's constructor. Add the GUIPanel object to the JLayeredPane object in ShapelyImagePane's constructor.
8. Test ShapelyImagePane, which will probably include the adjustment of the positions of some of the GUI elements. When you're happy, comment out the (xPosn,yPosn) println() call in mousePressed() in BackGroundPanel.

8. Other Approaches

The IFrame class developed by Michael Abernethy is a replacement for JFrame, which permits a much greater degree of customization (http://www-106.ibm.com/developerworks/java/library/j-iframe/) It allows custom colors, custom borders, custom shapes, custom components, and transparency.

An important advantage of IFrame over the code here, is that its implementation details are hidden behind a collection of classes, and an extensive API. This makes it possible to create JFrame variants without delving into the internals of methods, as we have to do.

