Chapter 10. Ghostly, Shapely Windows

It's time to give your favorite application a makeover, perking up its tired, overly familiar GUI.

A *ghostly* window is one that that can be translucent, and *shapely* means that the window may be non-rectangular. Translucency utilizes the Win32 API's *layered window* support for alpha-blending (see Hack 10.1), while Win32 *regions* (rectangles, rounded rectangles, polygons, ellipses, and combinations) can be employed to mask portions of a window to change its shape (Hack 10.2).

Regions are easiest to work with when the window's size is fixed, but Hack 10.3 shows an efficient way of handling resizing.

A shapely window implemented with regions can look a bit ugly since parts of the title bar, borders, and components are cut away. Hack 10.4 suggests a few tricks for fixing these problems, based around an undecorated JFrame and a specialized JPanel.

The undecorated JFrame and specialized panel makes an appearance in Hack 10.5 as well, where we implement shapely windows using a transparent pixel color. This approach allows a window to be almost any shape you wish.

It's useful to divide translucency into two types: *global translucency*, applied to an entire window and all its components (as seen in Hack 10.1), and a more versatile form called *per-pixel translucency*. It allows different parts of a window to exhibit different degrees of translucency, and is often used to soften the edges of curved windows to give them a smoother look. Hack 10.6 demonstrates how to employ a per-pixel translucent image to shape a JFrame.

Hack 10.7. utilizes global translucency and regions to animate a window's closing behavior. We describe two examples: a window that closes by fading away, and a window that is gradually cut down to nothing.

A great way of improving a window's appearance is to change its pluggable look and feel (L&F). Hack 10.8 shows how to use Java's standard L&Fs, and we also experiment with some third-party L&Fs (Substance and Napkin).

The Consumer JRE

The Consumer JRE (also known as *Java SE 6 Update N* and *Java SE 6 Update 10*) is a separate upgrade path for Java SE 6, distinct from the main version 6 releases (which are up to Java SE 6 Update 5 as we write). The Consumer JRE's focus is on the user's need for speed – one aim is to reduce the startup time for Java applications and applets, and another is to lower the time a JRE takes to install and launch when the user wants to run applets and Web Start applications on a machine without Java. There are several other interesting features, including a new L&F called Nimbus, a deployment toolkit, and graphics enhancements for Windows. A slightly old overview article by Chet Hasse can be found at http://java.sun.com/developer/technicalArticles/javase/consumerjre/, and the current Consumer JRE build is available from https://jdk6.dev.java.net/6uNea.html.

 © Gayathri Singh and Andrew Davison 2008

The Consumer JRE's relevance for this chapter is it's support for translucent and shaped windows , added in build 10 (February 2008), making it possible to code our hacks without utilizing J/Invoke. A good summary article by Kirill Grouchnikov about these capabilities can be found at http://today.java.net/pub/a/today/2008/03/18/translucent-and-shaped-swing-windows.html.

The obvious question is why bother with J/Invoke when we can utilize the Consumer JRE? The main reason is *code portability*: the Consumer JRE will only become an official part of Java with the release of Java SE 7, which is slated for early 2009 at the moment. In previous versions of Java, the J/Invoke approach is a good alternative.

In the hacks that follow, we'll explain J/Invoke code first, and a comparable Consumer JRE version, when applicable.

Do your Windows Lack Pizzazz?

Before we start liposuction, and the application of anti-aging cream and hair implants, we need a sad-sack window to work on. Figure 10-1 shows the UsualWin application, a JFrame sporting a button and a text field, as it appears in Windows XP and Vista.

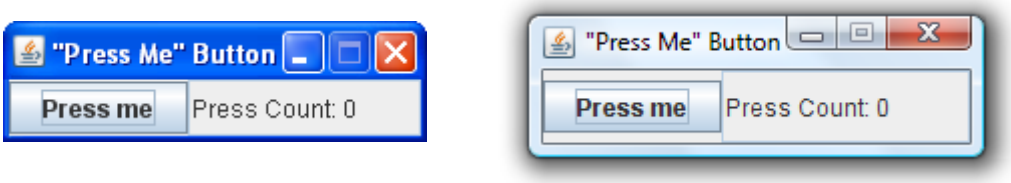


Figure 10-1 The UsualWin GUI (XP and Vista).

When the user presses the button, the counter in the text field is incremented.

There's nothing wrong with this program, apart from its visual over-familiarity. Our efforts in this chapter are purely cosmetic; we won't be switching to different GUI components or modifying the event handling.

We'll spare you most of the UsualWin code, since it's (intentionally) quite standard. The constructor gives a good overview of the class:

```
public UsualWin()
{
  super("\"Press Me\" Button");

  makeGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();

  setResizable(false);
  setLocationRelativeTo(null);
  setVisible(true);
} // end of UsualWin()
```

 © Gayathri Singh and Andrew Davison 2008

makeGUI() adds a JButton and JTextField to a horizontal BoxLayout, and an anonymous ActionListener is attached to the JButton to increment the counter.

Hack 10.1 A Translucent Window (medium level)

Create a window with a specified level of translucency.

TranslucentWin is a translucent version of the UsualWin class, with the translucency value optionally set on the command line when the class is invoked. The value can range from 0 (fully transparent) to 255 (opaque). The default is 128, which is shown applied to the window in Figure 10-2.

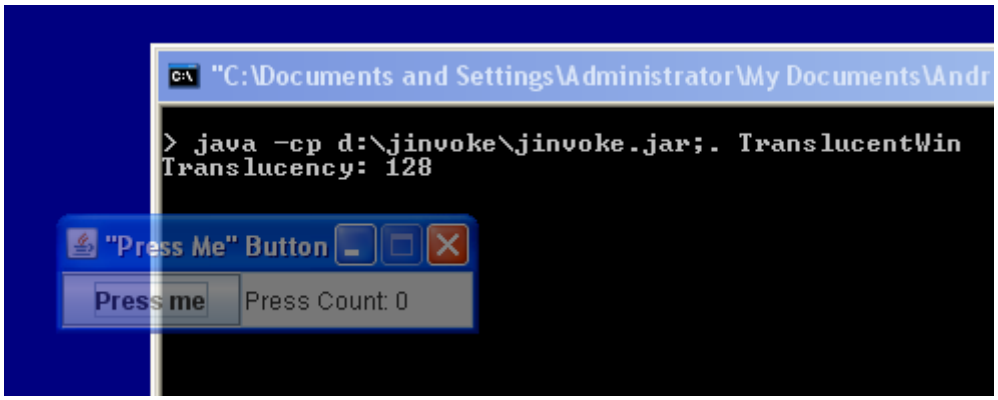


Figure 10-2. A Translucent Window.

For the effect to work smoothly, the use of DirectDraw should be switched off, forcing Java to rely on OpenGL. This must be done before the window is created, and a convenient place is at the start of main():

```
System.setProperty("sun.java2d.noddraw", "true");
```

Most of the code is unchanged from UsualWin, as the TranslucentWin constructor() shows:

```
public TranslucentWin(int translucency)
{
  super("\"Press Me\" Button");

  JInvoke.initialize();

  makeGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();
  setResizable(false);

  fadeWin(translucency);   // apply translucency to the window

  setLocationRelativeTo(null);
  setVisible(true);
} // end of TranslucentWin()
```

 © Gayathri Singh and Andrew Davison 2008

fadeWin() adds layered window support so the window can be made translucent by alpha-blending.

```
// globals
public static final int DEFAULT_TRANSLUCENCY = 128;

@NativeImport(library="user32")
public static native boolean SetLayeredWindowAttributes(
          int hwnd, int colorRef, byte bAlpha, int dwFlags);

private static final int WS_EX_LAYERED = 0x80000;
private static final int LWA_ALPHA = 2;    // 0x2


private void fadeWin(int translucency)
// make the 'title' window translucent by the specified amount
{
  if ((translucency < 0) || (translucency > 255)) {
    System.out.println("Transparency out of range (0-255)");
    translucency = DEFAULT_TRANSLUCENCY;   // use default value
  }

  int hwnd = Util.getWindowHandle(this);

  /* add layered window support to the window so it can be made
     translucent (alpha-blended) */
  int winStyleInfo = User32.GetWindowLong(hwnd,
                                  WinConstants.GWL_EXSTYLE);
  winStyleInfo |= WS_EX_LAYERED;   // 'add' layered support
  User32.SetWindowLong(hwnd, WinConstants.GWL_EXSTYLE, winStyleInfo);

  SetLayeredWindowAttributes(hwnd, 0, (byte)translucency, LWA_ALPHA);
               // set the translucency value of the layered window
} // end of fadeWin()
```

The window's handle is obtained using J/Invoke's utility method (Util.getWindowHandle(Component). The window must have been initialized, but it doesn't need to be visible. As a consequence, fadeWin() should be called after the window have been created with a call to pack() or setSize().

User32.GetWindowLong() is utilized with the GWL_EXSTYLE flag to retrieve the window's extended styles/properties (stored in an integer), and layering is added to them using a bitwise-or operation. Layering lets a window render complex shapes, animation, and alpha blending more efficiently (see http://msdn2.microsoft.com/en-us/library/ms632599(VS.85).aspx). The WS_EX_LAYERED property is described at http://msdn2.microsoft.com/en-us/library/ms632680.aspx, as part of the documentation for the CreateWindowEx() function.

The Win32 API function SetLayeredWindowAttributes() isn't included in the version of J/Invoke we're using, so we defined it ourselves, basing the definition on the MSDN documentation for the function at http://msdn2.microsoft.com/en-us/library/ms633540.aspx, and on J/Invoke's mapping between Java and Win32 C types. The prototype of the MSDN C function is:

```
BOOL SetLayeredWindowAttributes(
```

 © Gayathri Singh and Andrew Davison 2008

```
        HWND hwnd, COLORREF crKey, BYTE bAlpha, DWORD dwFlags);
```

The dwFlags argument can be LWA_COLORKEY, which causes the crKey argument to be used to specify a pixel color that represents transparency. However if dwFlags is LWA_ALPHA (as in fadeWin()) then the bAlpha argument is employed to set the translucency level for the entire window. we'll make use of LWA_COLORKEY in Hack 10.5.

A Consumer JRE Translucent Window

A Consumer JRE version of the application only requires a call to setWindowOpacity() in the com.sun.AWTUtilities. This method replaces our fadeWin() method of the previous example, resulting in the following constructor:

```
public class TranslucentWindow extends JFrame
{
  private JTextField jtf;
  private int pressCount = 1;


  public TranslucentWindow()
  {
    super("Translucent window");

    makeGUI();
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();

    setResizable(false);
    setLocationRelativeTo(null);

    AWTUtilities.setWindowOpacity(this, 0.5f);  // 50 % translucency

    setVisible(true);
  }  // end of TranslucentWindow()
```

The first argument of setWindowOpacity() is a reference to the window that's being made translucent, and the second value is the translucency, expressed as a float between 0 and 1.

There's also no need to switch off the use of DirectDraw by setting the "sun.java2d.noddraw" property.

Compiling and running this example requires the download of the latest version of the Consumer JRE from https://jdk6.dev.java.net/6uNea.html. On our test machine, it was installed at C:\Program Files\Java\jdk1.6.0_10, and javac and java calls must use the tools in its bin\ subdirectory. For example:

```
> "%ProgramFiles%\Java\jdk1.6.0_10\bin\javac" TranslucentWindow.java
> "%ProgramFiles%\Java\jdk1.6.0_10\bin\java" TranslucentWindow
```

The location of the AWTUtilities class in the Java package hierarchy will change in Java SE 7, as these tools are integrated into the rest of AWT. It's also likely that

 © Gayathri Singh and Andrew Davison 2008

method names and arguments may be modified as well, so check the current documentation.

Hack 10.2 A Shapely Window (medium level)

Create various kinds of non-rectangular windows.

ShapedWin is a shapely version of the UsualWin class, illustrating five non-rectangular forms:

1. An elliptical window;
2. A rounded rectangular window;
3. A polygonal window; in this case, an arrow;
4. Two elliptical regions combined to make an 'eyes'-shaped window
5. An elliptical region subtracted from a larger ellipse to make a 'doughnut'-shaped window.

The choice of shape is specified on the command line when ShapedWin is called.

All the examples use MS Windows' region mechanism A region is an ellipse, rectangle, polygon, or combinations of those shapes, which can be used to specify the shape of a window (see http://msdn2.microsoft.com/en-us/library/ms536684(VS.85).aspx for more details). Examples 4 and 5 employ more complex regions, created by combining simple regions using set-like operations (e.g. union and difference)"

The ShapedWin code is simplified by making the window a fixed size. Resizable shaped windows are considered in Hack 10.3.

The ShapedWin() constructor is very similar to the constructor for TranslucentWin (Hack 10.1), except for its call to shapeWin():

```
public ShapedWin(int shapeVal)
{
  super("\"Press Me\" Button");

  JInvoke.initialize();

  makeGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();
  setResizable(false);

  shapeWin(shapeVal);

  setLocationRelativeTo(null);
  setVisible(true);
} // end of ShapedWin()
```

The shapeVal variable contains an integer read from the command line which represents the desired shape.

 © Gayathri Singh and Andrew Davison 2008

shapeWin() changes the window shape by applying a region to it. The choice between the five possible shapes is handled by a multi-way if-statement. In each branch, a region is assigned to the regionHandle variable, and applied to the window by a call to User32.SetWindowRgn().

```
// global
public static final int DEFAULT_SHAPE = 3;   // the arrow shape


private void shapeWin(int shapeVal)
{
  if ((shapeVal < 1) || (shapeVal > 5)) {
    System.out.println("Shape value out of range (1-5)");
    shapeVal = DEFAULT_SHAPE;   // use default value
  }

  int hwnd = Util.getWindowHandle(this);
  int w = getWidth();    // width and height of window
  int h = getHeight();

  // ------------ region creation ----------------

  int regionHandle = Gdi32.CreateRectRgn(0, 0, w, h);
              // a default initial shape, a plain rectangle

  if (shapeVal == 1)         // an ellipse
    regionHandle = Gdi32.CreateEllipticRgn(0, 0, w, h);
  else if (shapeVal == 2)    // a rounded rectangle
    regionHandle = Gdi32.CreateRoundRectRgn( 0, 0, w, h, 50, 50);
  else if (shapeVal == 3) {  // a polygon (an arrow)
    com.jinvoke.win32.structs.Point[] pts = makeArrowPolygon(w, h);
    regionHandle = Gdi32.CreatePolygonRgn( pts, pts.length,
                                      WinConstants.ALTERNATE);
  }
  else if (shapeVal == 4) {  // eyes shape
    int leftHandle = Gdi32.CreateEllipticRgn(0, 0, w/2, h);
    int rightHandle = Gdi32.CreateEllipticRgn(w/2, 0, w, h);
    regionHandle = Gdi32.CreateRectRgn(0, 0, 0, 0);  // dummy values
    Gdi32.CombineRgn(regionHandle, leftHandle, rightHandle,
                                   WinConstants.RGN_OR); // union
    Gdi32.DeleteObject(leftHandle);
    Gdi32.DeleteObject(rightHandle);
  }
  else if (shapeVal == 5)  {  // a doughnut shape
    int outerHandle = Gdi32.CreateEllipticRgn(0, 0, w, h);
    int innerHandle = Gdi32.CreateEllipticRgn(w/4, h/4, 3*w/4,3*h/4);
    regionHandle = Gdi32.CreateRectRgn(0, 0, 0, 0);  // dummy values
    Gdi32.CombineRgn(regionHandle, outerHandle, innerHandle,
                           WinConstants.RGN_DIFF); // set difference
    Gdi32.DeleteObject(outerHandle);
    Gdi32.DeleteObject(innerHandle);
  }
  else   // should not occur
    System.out.println("Did not recognize shape value: " + shapeVal);

  // ------------- end of region creation --------------

  User32.SetWindowRgn(hwnd, regionHandle, true);

  Gdi32.DeleteObject(regionHandle);
```

 © Gayathri Singh and Andrew Davison 2008

```
} // end of shapeWin()
```

The function checks that shapeVal is within a specified range, then obtains the window's handle and dimensions. Util.getWindowHandle() can obtain a window handle once the JFrame is initialized. (This is the same criteria as in TranslucentWin in Hack 10.1)

shapeWin() is only called once, at object initialization, and uses the window's initial width and height (w and h) in its region calculations. This approach is possible since the window is a fixed size.

A complete list of region functions can be found at http://msdn2.microsoft.com/en-us/library/ms536680(VS.85).aspx.

Shape 1: An Ellipse

Figure 10-3 shows the window as an ellipse.



Figure 10-3. An Elliptical ShapedWin.

The application still works as before: pressing the button will increment the counter shown in the text field. The window can be dragged, iconified and closed as usual, but the area cut away by the region makes it a little tricky to select the close box. Also, the shape doesn't have a well defined border that can be selected and dragged to resize the window. This illustrates a general problem with shapely windows –the unusual shape may make the GUI harder to use. We address this issue in Hack 10.4.

The relevant call in shapeWin() is:

```
regionHandle = Gdi32.CreateEllipticRgn(0, 0, w, h);
```

The arguments specify the top-left and lower-right corners of the ellipse's bounding rectangle.

Another issue with regions is that they disable visual styles when applied to a window. If you look closely at Figure 10-3, comparing it with the window in Figure 10-1, its visual style has changed to "Windows Classic" from "Windows XP". The same change can be seen in the other figures in this section (Figures 10-4 to 10-8).

Shape 2: A Rounded Rectangle

Figure 10-4 shows the window as a rounded rectangle.

 © Gayathri Singh and Andrew Davison 2008



Figure 10-4. A Rounded Rectangular ShapedWin.

The relevant call in shapeWin() is:

```
regionHandle = Gdi32.CreateRoundRectRgn( 0, 0, w, h, 50, 50);
```

The first four arguments specify the top-left and lower-right corners of the shape's bounding rectangle; the fifth and sixth parameters are the width and height of the ellipse used for the rounded corners.

Shape 3: A Polygon

Figure 10-5 shows a right facing arrow, a simple polygon.

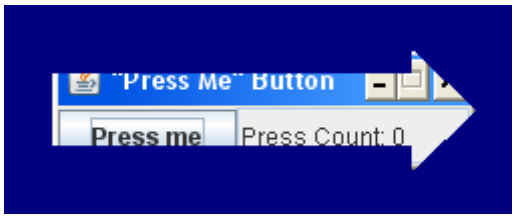


Figure 10-5. A Polygonal ShapedWin.

The relevant code in shapeWin() is:

```
com.jinvoke.win32.structs.Point[] pts = makeArrowPolygon(w, h);
regionHandle = Gdi32.CreatePolygonRgn( pts, pts.length,
                                       WinConstants.ALTERNATE);
```

Gdi32.CreatePolgonRgn() requires an array of points, the number of points in the array, and a polygon-filling mode. The points define the vertices of a closed polygon, so each vertex only needs to be specified once.

The two filling modes are ALTERNATE and WINDING, which only differ in their behavior when a complex, overlapping polygon has to be filled (see the SetPolyFillMode() documentation at http://msdn2.microsoft.com/en-us/library/ms536642(VS.85).aspx for an example).

The series of points for the arrow are shown in Figure 10-6.

 © Gayathri Singh and Andrew Davison 2008

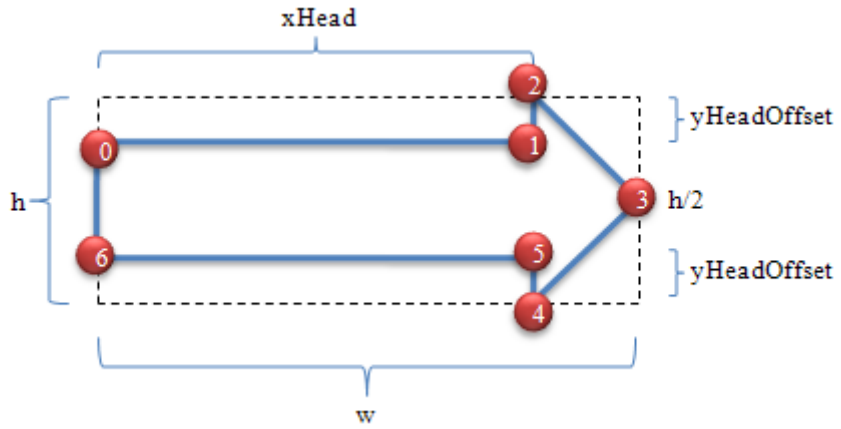


Figure 10-6. The Points in the Arrow.

The array starts with point 0, progresses clockwise, and finishes with point 6. w and h are the width and height of the underlying window.

makeArrowPolygon() creates the points and returns an array:

```
private com.jinvoke.win32.structs.Point[] makeArrowPolygon(
                                                    int w, int h)
// create a right facing arrow shape for the window
{
  int numPoints = 7;
  com.jinvoke.win32.structs.Point[] pts =
                new com.jinvoke.win32.structs.Point[numPoints];
  for (int i=0; i < numPoints; i++)
    pts[i] = new com.jinvoke.win32.structs.Point();

  int xHead = (85 * w)/100;     // x coord of arrow head's back edge
  int yHeadOffset = (20 * h)/100; // y offset of head's back edge

  pts[0].x =  0;     pts[0].y =  yHeadOffset;
  pts[1].x = xHead;  pts[1].y =  yHeadOffset;
  pts[2].x = xHead;  pts[2].y =  0;
  pts[3].x = w;      pts[3].y =  h/2;
  pts[4].x = xHead;  pts[4].y =  h;
  pts[5].x = xHead;  pts[5].y =  h-yHeadOffset;
  pts[6].x =  0;     pts[6].y =  h-yHeadOffset;

  return pts;
}  // end of makeArrowPolygon()
```

The fully qualified name for the Win32 Point class is used to differentiate it from Java's Point class.

Shape 4: The Eyes Have It

Figure 10-7 shows the window as a pair of eyes.

 © Gayathri Singh and Andrew Davison 2008

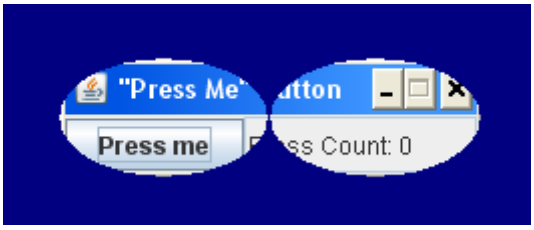


Figure 10-7. An Eyes-shaped ShapedWin.

The relevant code fragment in shapeWin() is:

```
int leftHandle = Gdi32.CreateEllipticRgn(0, 0, w/2, h);
int rightHandle = Gdi32.CreateEllipticRgn(w/2, 0, w, h);

regionHandle = Gdi32.CreateRectRgn(0, 0, 0, 0);  // dummy values
Gdi32.CombineRgn(regionHandle, leftHandle, rightHandle,
                                    WinConstants.RGN_OR); // union
Gdi32.DeleteObject(leftHandle);
Gdi32.DeleteObject(rightHandle);
```

The eyes are two ellipses combined with Gdi32.CombineRgn(). The second and third region arguments of the function are merged according to the flag given as its fourth argument, and the result stored in the first argument. The flag values are set-based, and include:

- RGN_AND: Create the intersection of the two input regions.
- RGN_OR: Create the union of the two input regions.
- RGN_XOR: Create the union of the two input regions except for any overlapping areas.
- RGN_DIFF: Retain the parts of the first region (the second argument) that aren't in the other region (the third argument).

More details can be found at http://msdn2.microsoft.com/en-us/library/ms536688(VS.85).aspx.

Shape 5. A Doughnut.

Figure 10-8 shows the window as a doughnut.



Figure 10-8. A Doughnut-shaped ShapedWin.

The relevant code fragment in shapeWin() is:

 © Gayathri Singh and Andrew Davison 2008

```
int outerHandle = Gdi32.CreateEllipticRgn(0, 0, w, h);
int innerHandle = Gdi32.CreateEllipticRgn(w/4, h/4, 3*w/4, 3*h/4);

regionHandle = Gdi32.CreateRectRgn(0, 0, 0, 0);  // dummy values
Gdi32.CombineRgn(regionHandle, outerHandle, innerHandle,
                           WinConstants.RGN_DIFF); // set difference
Gdi32.DeleteObject(outerHandle);
Gdi32.DeleteObject(innerHandle);
```

The trick is to combine two ellipses once again, but use the RGN_DIFF mode to subtract the smaller ellipse's region from the larger one.

A Consumer JRE Shaped Window

A Consumer JRE version of the application can employ AWTUtilities.setWindowShape() to shape the window. The elliptical window in Figure 10-3 can be coded as shown below:

```
public ShapedWindow()
{
  super("Oval window");

  makeGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();

  setResizable(false);
  setLocationRelativeTo(null);

  AWTUtilities.setWindowShape(this,
    new Ellipse2D.Double(0, 0, getWidth(), getHeight()));
                          // oval shaped
  setVisible(true);
 }  // end of ShapedWindow()
```

The call to setWindowShape() replaces our fadeWin() method, with the second argument being any subclass of java.awt.Shape (in this case Ellipse2D.Double).

It's also necessary to set the L&F to the default, via a call to JFrame.setDefaultLookAndFeelDecorated(true) in main() before the window is created. This results in Figure 10-9.



Figure 10-9. An Elliptical Window using AWTUtilities

We'll talk more about Java L&F in Hack 10.8.

 © Gayathri Singh and Andrew Davison 2008

The Win32 API approach and AWTUtilities.setWindowShape() can create complex shapes in similar ways. Window regions are constructed using set operations (such as union and intersection) while setWindowShape() can employ the Area subclass of java.awt.Shape. An Area object can be built from other Area objects by using operations such as area addition, subtraction, intersection, and exclusive-or. A shape may also be defined using java.awt.geom.GeneralPath to construct it from straight lines, and curves. Some simple examples of Area and GeneralPath can be found at http://www.java2s.com/Code/Java/2D-Graphics-GUI/AreaCalculationAddSubstractXOR.htm.

A Pure Java Approach to Shapely Windows

It's possible to implement shapely windows without resorting to the Win32 API or the Consumer JRE. The technique relies on taking a screenshot before the application is made visible, and using a clipped portion of that screenshot as the application's background. The image will match up with the desktop behind, thereby creating the illusion of transparency.

The details are explained in *Swing Hacks*, Marinacci and Adamson, O'Reilly 2005 (Hack #41, which is online at http://www.onjava.com/pub/a/onjava/excerpt/swinghks_hack41/), and in the article "Shapely (non-rectangular) Applications" available at http://fivedots.coe.psu.ac.th/~ad/jg/ch045/.

The drawback is the overhead of updating the screenshot when the application is moved. Also, it's just about impossible for a pure Java program to detect when the desktop behind the application changes, which should trigger another screenshot.

The answer may seem to be to update the screenshot frequently, but this causes the application to flicker excessively. Each update requires the momentary hiding of the window while the screenshot is taken, followed by a recalculation of the clipped area for the background. Also, an appropriate update frequency isn't easy to determine.

Hack 10.3 A Resizable Shapely Window (medium/high level)

An efficient way to resize a non-rectangular window.

ShapedWin in Hack 10.2 relies on the width and height of the window being fixed. This hack illustrates how a rounded rectangular window can be resized at run-time. The technique can be used to make any shapely window resizable.

There are two issues:

1. The rounded rectangle region must be recalculated whenever the window is resized. This can be done by a component listener, acting in response to window size changes.
2. The component listener has the drawback that it's only triggered when the window has *finished* being resized. This means that the rounded rectangle won't be modified while the user is dragging the window's frame.

   This problem could be fixed by doing frequent shape recalculations during mouse dragging. This is both inefficient and visually ugly. Instead we'll

 © Gayathri Singh and Andrew Davison 2008

temporarily change MS Windows' behavior to show only a window's *outline* while it's being dragged (or moved). Since there's only an outline on display during resizing, there's no need to recalculate the shape until the user releases the mouse and finishes the resize.

The switching on and off of window outlining is managed by a Java window focus listener which monitors when the application gains focus and loses it.

Figure 10-10 shows the rounded rectangle being resized in outline by mouse dragging.

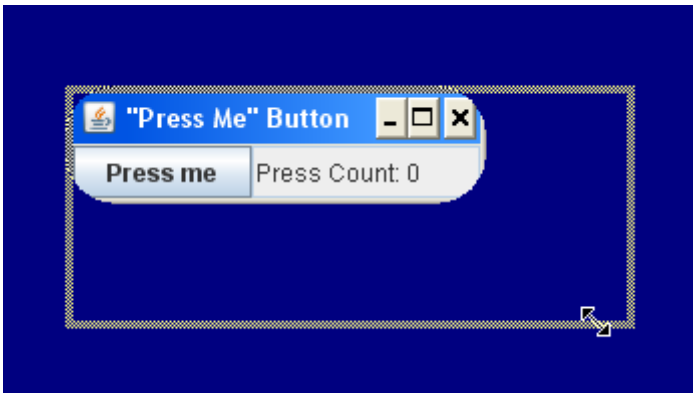


Figure 10-10. Resizing the Window by Dragging.

Figure 10-11 shows the final appearance of the window when the user releases the mouse.

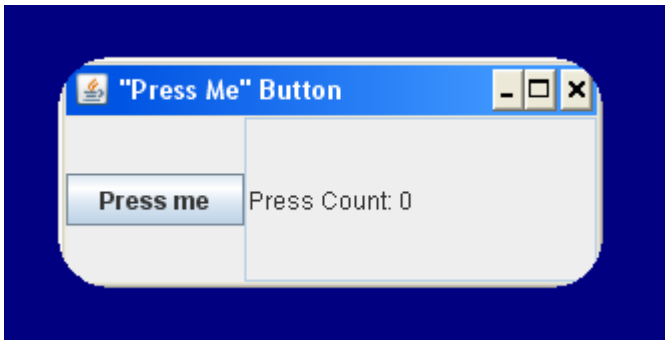


Figure 10-11. The Final Size of the Window.

The ShapedResizeWin constructor sets up the component and window listeners, and manages the close box.

```
// global
private int hwnd;    // window handle

public ShapedResizeWin()
{
  super("\"Press Me\" Button");

  JInvoke.initialize();
```

 © Gayathri Singh and Andrew Davison 2008

```
  makeGUI();
  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { resetWindowResizing();
      System.exit(0);
    }
  });

  pack();
  hwnd = Util.getWindowHandle(this);

  addComponentListener(this);
  addWindowFocusListener(this);

  setLocationRelativeTo(null);
  setVisible(true);
} // end of ShapedResizeWin()
```

The outlining behavior is an OS-wide modification, so when the application finishes it's good manners to switch the behavior back to how it was initially. This is done by resetWindowResizing() before the program exits.

The Component Listener

After the window has been resized, the componentResized() listener method is called to recalculate the rounded rectangle's dimensions, and apply the new region to the window.

```
public void componentResized(ComponentEvent e)
{
  int w = getWidth();    // new dimensions of the window
  int h = getHeight();

  int regionHandle = Gdi32.CreateRoundRectRgn( 0, 0, w, h, 50, 50);
  User32.SetWindowRgn(hwnd, regionHandle, true);

  Gdi32.DeleteObject(regionHandle);
} // end of componentResized()
```

The Window Focus Listener

When the window gains focus, outlined dragging should be switched on, and when focus is lost the original behavior should be restored. This code is placed inside windowGainedFocus() and windowLostFocus().

The relevant Win32 function is SystemParametersInfo(), along with the action parameter for dragging windows (see http://msdn2.microsoft.com/en-us/library/ms724947(VS.85).aspx for the complete, long list of possible parameters for SystemParametersInfo()).

Outlined window dragging is a Windows Control Panel setting, found in Windows XP at Control Panel > Display > Appearance tab > Effects > Show window contents while dragging checkbox (see Figure 10-12).

 © Gayathri Singh and Andrew Davison 2008

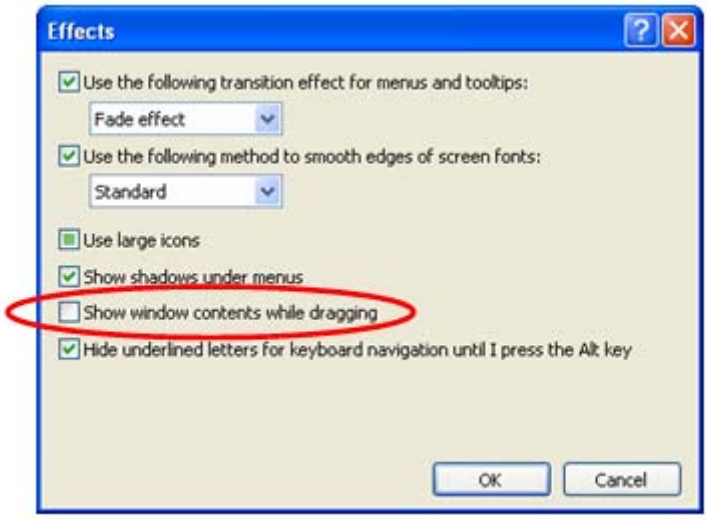


Figure 10-12. The Outlined Dragging Control Panel Setting in Windows XP.

On one of our test machines, the checkbox is unselected (as shown in Figure 10-12), which means that windows are dragged in outline. On another test machine, the box is selected which causes the windows contents to be shown.

On Vista, this setting can be found under: Control Panel > Appearance and Personalization > Customize > Open classic appearance properties or more color options > Advanced.

Since the feature is a control panel setting, SystemParametersInfo() must include flags to access and update the WIN.INI file, and broadcast the changes. The current setting is obtained with SystemParametersInfo() and SPI_GETDRAGFULLWINDOWS:

```
int[] setting = new int[1];   //to hold the retrieved setting
boolean res = User32.SystemParametersInfo(
                 WinConstants.SPI_GETDRAGFULLWINDOWS,
                 0, setting, WinConstants.SPIF_UPDATEINIFILE |
                             WinConstants.SPIF_SENDWININICHANGE);
System.out.println("Dragging setting: " + setting[0]);
```

If the checkbox is currently selected, setting[0] will contain 1, otherwise it will hold 0.

Changing the window behavior to outlined dragging requires the setting to be switched to 0. The code for that employs SystemParametersInfo() and SPI_SETDRAGFULLWINDOWS:

```
res = User32.SystemParametersInfo(
             WinConstants.SPI_SETDRAGFULLWINDOWS,
             0, setting, WinConstants.SPIF_UPDATEINIFILE |
                         WinConstants.SPIF_SENDWININICHANGE);
```

The second parameter of SystemParametersInfo() is 0 to switch to outlined dragging (1 would change it to window contents dragging). The setting[] array isn't used when the feature is being changed.

 © Gayathri Singh and Andrew Davison 2008

More examples of using SystemParametersInfo() to affect system-wide settings can be found at http://support.microsoft.com/kb/97142.

windowGainedFocus() changes the setting (if necessary), and records the fact in a boolean so the original setting can be restored later.

```
// global
private boolean switchedToWinOutline = false;
  /* indicates whether this application changed the Win setting
     so that only an outline is shown while a window is being
     dragged */


public void windowGainedFocus(WindowEvent e)
{
  // retrieve the window dragging setting
  int[] setting = new int[1];
  boolean res = User32.SystemParametersInfo(
                  WinConstants.SPI_GETDRAGFULLWINDOWS,
                  0, setting, WinConstants.SPIF_UPDATEINIFILE |
                              WinConstants.SPIF_SENDWININICHANGE);

  if (res && (setting[0] == 1)) {    // window contents are shown
    res = User32.SystemParametersInfo(       // switch to outline
                  WinConstants.SPI_SETDRAGFULLWINDOWS,
                  0, setting, WinConstants.SPIF_UPDATEINIFILE |
                              WinConstants.SPIF_SENDWININICHANGE);
    if (res)  // if dragging was changed
      switchedToWinOutline = true;
  }
}  // end of windowGainedFocus()
```

windowLostFocus() must reset the window dragging behavior, so it calls resetWindowResizing() (which is also used by windowClosing()).

```
public void windowLostFocus(WindowEvent e)
{  resetWindowResizing();  }


private void resetWindowResizing()
/* called when focus is lost or when the window is closed in order to
   switch window dragging behavior back to how it was initially */
{
  if (switchedToWinOutline) { // dragging behavior was changed
    int[] setting = new int[1];   // an array is needed but not used
    boolean res = User32.SystemParametersInfo(
                    WinConstants.SPI_SETDRAGFULLWINDOWS,
                    1, setting, WinConstants.SPIF_UPDATEINIFILE |
                                WinConstants.SPIF_SENDWININICHANGE);
    if (res)
      switchedToWinOutline = false;
  }
}  // end of resetWindowResizing()
```

 © Gayathri Singh and Andrew Davison 2008

If switchedToWinOutline is true, then SystemParametersInfo() is called with SPI_SETDRAGFULLWINDOWS and a second parameter value of 1 to switch back to window contents dragging.

Hack 10.4 A Decorated Shapely Window (medium level)

Use a painted JPanel and undecorated JFrame to make a shapely window look more beautiful.

A limitation on a window shape is that it mustn't completely obscure the title bar, otherwise it becomes impossible to move the window, minimize/maximize, or close it.

A drastic solution is to do away with the title bar altogether, and use an undecorated JFrame. This removes any restrictions on the shape of the window. Instead window movement, iconification, and termination are managed by mouse listeners

The underlying application in this hack is the same as previously, but the button and text are placed inside a PaintedJPanel instance. This specialized JPanel puts an empty border around the components to offset them from the edges of the JFrame, and supports background images and gradient paints to make the application look better.

The GUI is a little different from previously – the JTextField for showing the press count has been replaced by a JLabel since it looks better against the new backgrounds.

To keep things simple, the window is a fixed size.

Figure 10-13 shows that the application utilizes translucency (using code copied from Hack 10.1), and an arrow shape borrowed from Hack 10.2

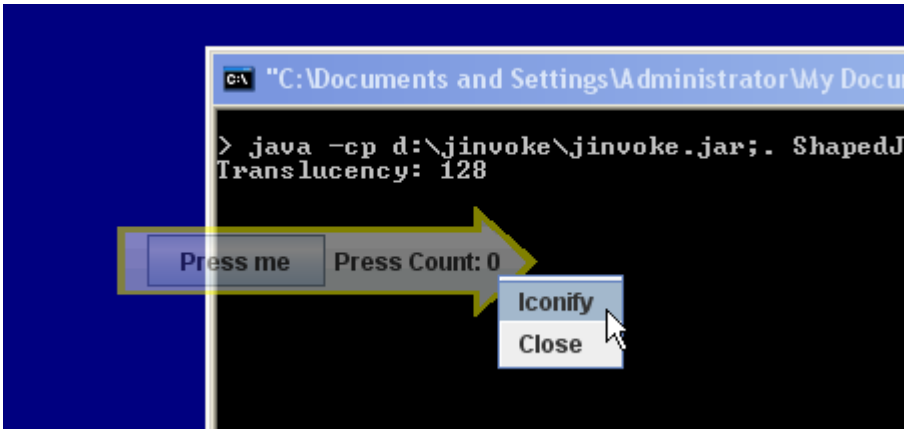


Figure 10-13. A Decorated Arrow Window.

The user has just right-clicked on the JPanel in Figure 10-13, causing a popup menu to appear. The menu allows the user to iconify and close the application. The application can be moved by dragging the mouse anywhere inside the shape.

The arrow shape's decoration (the yellow edges, and gray interior) come from the image shown in Figure 10-14.

 © Gayathri Singh and Andrew Davison 2008

Figure 10-14. The Arrow Background Image.

The image is loaded by PaintedJPanel and painted onto its surface.

Figure 10-15 shows the arrow shape with a blue steel image tiled over the JPanel instead of the arrow image.



Figure 10-15. The Arrow Window with a Blue Steel Background.

Figure 10-16 shows the use of a Java 2D paint gradient in PaintedJPanel rather than an image.



Figure 10-16. The Arrow Window with a Painted Gradient Background.

The full range of Java 2D graphics can be utilized for decorating the shape via the JPanel.

The Shaped Window

The ShapedJFrame() constructor utilizes translucency and shape code from Hacks 10.1 and 10.2.

```
public ShapedJFrame(String title, int translucency)
{
  super(title);

  JInvoke.initialize();

  // listeners for dealing with iconification, closing, and dragging
  addMouseListener(this);
  addMouseMotionListener(this);

  setUndecorated(true);
  makeGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();

  shapeWin(title);
  fadeWin(title, translucency);

  setResizable(false);
```

 © Gayathri Singh and Andrew Davison 2008

```
  setLocationRelativeTo(null);
  setVisible(true);
}  // end of ShapedJFrame()
```

shapeWin() generates an arrow-shaped region using an array of points, while fadeWin() sets the translucency according to a value supplied on the command line.

The only new code is the addition of the mouse listener and mouse motion listener.

Mouse Events

Mouse presses and releases are monitored by ShapedJFrame.

```
// global
private Point startPos;   // cursor position in JFrame


public void mousePressed(MouseEvent e)
{  startPos = e.getPoint();  }


public void mouseReleased(MouseEvent e)
{ if (e.isPopupTrigger())   // the right button is pressed
    showPopup(e);
}
```

A mouse press triggers the storage of the current mouse position, which becomes useful if the user starts dragging the mouse.

A mouse release causes a test of the popup trigger (the right button in MS Windows), and if the button is being held down, the popup menu is displayed.

```
private void showPopup(MouseEvent e)
// The popup supports iconification and closing
{
  JPopupMenu popup = new JPopupMenu();

  popup.add( new AbstractAction("Iconify") {
    public void actionPerformed(ActionEvent e)
    {  setState(JFrame.ICONIFIED); }   // iconify the window
  });

  popup.add( new AbstractAction("Close") {
    public void actionPerformed(ActionEvent e)
    { System.exit(0); }
  });

  popup.pack();
  popup.show( e.getComponent(), e.getX(), e.getY());
} // end of showPopup()
```

The menu's actions are simple enough to include as anonymous classes, and further menu items and functionality could easily be added.

 © Gayathri Singh and Andrew Davison 2008

Mouse Dragging

As the user drags the mouse, the distance that the cursor has moved inside the JFrame relative to its starting position is used to move the entire window.

```
public void mouseDragged(MouseEvent e)
{
  if (!SwingUtilities.isLeftMouseButton(e))
    return;   // only respond to the left button

  Point distMoved = new Point(e.getX() - startPos.x,
                              e.getY() - startPos.y);
      // calculate current cursor posn - start posn in the JFrame

  if ((distMoved.x != 0) || (distMoved.y != 0)) {  // if cursor moved
    Point loc = getLocationOnScreen();   // get current loc of JFrame
    setLocation(loc.x + distMoved.x,  loc.y + distMoved.y);
                  // move JFrame by adding the cursor's movement
  }
}  // end of mouseDragged()
```

The call to setLocation() at the end of mouseDragged() moves the window under the cursor, so the cursor ends up back in its starting position relative to the JFrame.

The Paintable JPanel

The PaintedJPanel constructor loads a specified image, and sets up an invisible border.

```
// globals
private static final int BORDER = 15;

// image used to cover panel
private static final String BACK_IM = "miniSteel.jpg";
// private static final String BACK_IM = "arrow.png";

private BufferedImage bImage;


public PaintedJPanel()
{
  super();

  bImage = loadImage(BACK_IM);
  setLayout( new BoxLayout(this, BoxLayout.LINE_AXIS));
  setBorder(new EmptyBorder(BORDER, BORDER, BORDER, BORDER));

  setBackground(null);   // use same colour as surrounding JFrame
  setOpaque(false);
}  // end of PaintedJPanel()
```

The border indents the panel's components from its edges, which allows a greater range of window shapes to 'cut into' the panel space without hiding the components.

 © Gayathri Singh and Andrew Davison 2008

The JPanel's paintComponent() draws the loaded image onto the panel, tiling it if the image is smaller than the panel area (which is the case for the steel image used in Figure 10-15).

```
public void paintComponent(Graphics g)
{
  if(isOpaque())
    super.paintComponent(g);

  int panelWidth = getWidth();
  int panelHeight = getHeight();

  // display an image
  if (bImage != null) {
    int imWidth = bImage.getWidth(null);
    int imHeight = bImage.getHeight(null);
    // tile the image over the panel if necessary
    for (int x = 0; x < panelWidth; x += imWidth)
       for (int y = 0; y < panelHeight; y += imHeight)
         g.drawImage(bImage, x, y, this);
  }
}  // end of paintComponent()
```

The gradient shown in Figure 10-16 is achieved by the following code fragment:

```
GradientPaint gp =
  new GradientPaint(0, 0, Color.DARK_GRAY, 0, panelHeight/2,
                                          Color.LIGHT_GRAY, true);
Graphics2D g2d = (Graphics2D)g;
g2d.setPaint(gp);
g2d.fillRect(0, 0, panelWidth, panelHeight);
```

An important point to remember is that whatever is painted onto the panel will be clipped if it extends beyond the window region.

Hack 10.5 Windows of Any Shape (medium level)

Create a shapely window of any shape, not just a combination of ellipses, rectangles, and polygons.

Hack 10-2 is a little limiting since the resulting shape must be a combination of rectangles, rounded rectangles, ellipses, and polygons. A different way of creating a window shape is described in this hack, based on specifying a pixel transparency color. Anything drawn in that color will be rendered transparent.

The code uses SetLayeredWindowAttributes(), first seen in Hack 10.1 where its LWA_ALPHA flag was used to define a translucency level. This time we'll be using the LWA_COLORKEY flag which sets a transparency color.

 © Gayathri Singh and Andrew Davison 2008

Figure 10-17 shows the BlobJFrame application, which uses a transparency color to achieve its 'blob' like shape, and also supports translucency.

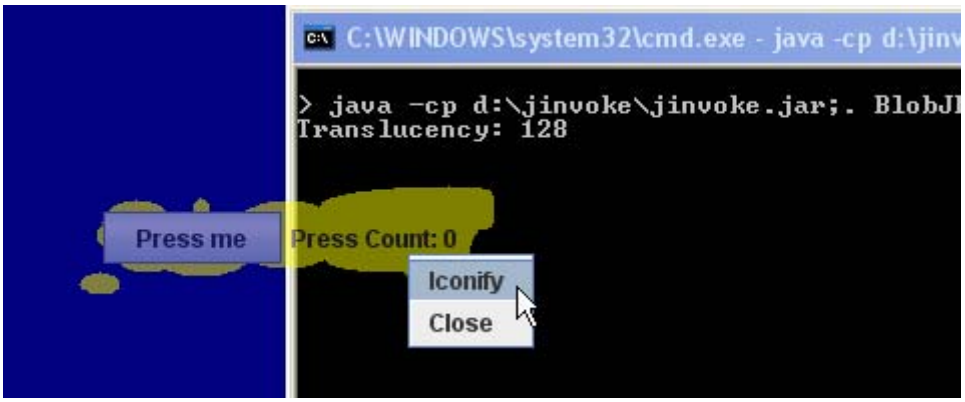


Figure 10-17. A Shapely and Translucent BlobJFrame.

The user has just right-clicked on the JPanel, causing a popup menu to appear. The menu lets the user iconify and close the application. The application can be moved by dragging the mouse anywhere inside the shape.

The code is quite similar to the example in Hack 10.4: BlobJFrame is a subclass of a JFrame using mouse listeners in the same way as ShapedJFrame to implement a popup menu and dragging. We also reuse PaintedJPanel from Hack 10.4, almost unchanged, except that it paints the image shown in Figure 10-18.



Figure 10-18. The Blob Image.

The image is loaded by PaintedJPanel and painted on its surface. The transparency color set by BlobJFrame is red, and so the red parts of the image aren't rendered.

The Blob Window

Much of BlobJFrame is identical to ShapedJFrame, in particular the code for listening to mouse presses, releases, and dragging, so we won't describe it again. The important difference occurs in the setup of the window in BlobJFrame().

```
public BlobJFrame(int translucency)
{
  super("\"Press Me\" Button")

  JInvoke.initialize();

  // add listeners for iconification, closing, and dragging
  addMouseListener(this);
  addMouseMotionListener(this);

  setUndecorated(true);
  makeGUI();
```

 © Gayathri Singh and Andrew Davison 2008

```
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();

  transparentWin(translucency);

  setResizable(false);
  setLocationRelativeTo(null);
  setVisible(true);
}  // end of BlobJFrame()
```

transparentWin() carries out two tasks – the setting of the translucency level (which we've seen several times) and the setting of the pixel transparency color to red.

```
// globals
@NativeImport(library="user32")
public static native boolean SetLayeredWindowAttributes(
           int hwnd, int colorRef, byte bAlpha, int dwFlags);
   // see http://msdn2.microsoft.com/en-us/library/ms633540.aspx

public static final int DEFAULT_TRANSLUCENCY = 128;

private static final int WS_EX_LAYERED = 0x80000;
private static final int LWA_COLORKEY = 1;  // 0x1
private static final int LWA_ALPHA = 2;     // 0x2


private void transparentWin(int translucency)
{
  if ((translucency < 0) || (translucency > 255)) {
    System.out.println("Supplied translucency out of range (0-255)");
    translucency = DEFAULT_TRANSLUCENCY;   // use default value
  }
  System.out.println("Translucency: " + translucency);

  int hwnd = Util.getWindowHandle(this);

  /* add layered window support to the window so it can be made
     translucent (alpha-blended) and transparent */
  int winStyleInfo = User32.GetWindowLong( hwnd,
                                WinConstants.GWL_EXSTYLE);
  winStyleInfo |= WS_EX_LAYERED;
  User32.SetWindowLong(hwnd, WinConstants.GWL_EXSTYLE, winStyleInfo);

  SetLayeredWindowAttributes(hwnd, 0x0000ff, (byte)translucency,
                                          LWA_COLORKEY|LWA_ALPHA);
} // end of transparentWin()
```

The crucial line is the call to SetLayeredWindowAttributes() with both the LWA_COLORKEY and LWA_ALPHA flags set. The transparency color is specified in the second argument of SetLayeredWindowAttributes() as an integer with the hexadecimal format 0x00bbggrr (00 for alpha, bb for blue, gg for green, and rr for red). The value used in the example is 0x0000ff, a color consisting of no blue, no green, and full-on red.

If only the transparency color is required, with no translucency, then the call would become:

```
SetLayeredWindowAttributes(hwnd, 0x0000ff, (byte)0, LWA_COLORKEY);
```

 © Gayathri Singh and Andrew Davison 2008

Hack 10.6 A Window Using Per-pixel Translucency (medium level)

Create a shaped window using a per-pixel translucent image.

Figure 10-19 shows the GUIJWindow application positioned in front of some other windows.

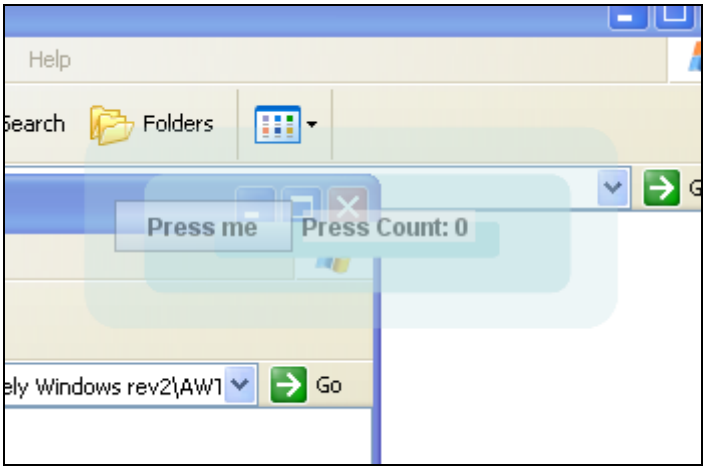


Figure 10-19. The Per-Pixel Translucent Example.

A close look at the window reveals a nested series of increasingly opaque rounded rectangles, with the now-familiar button and text label at their center. This shape is supplied to the application in the form of a 32 bpp (bit per pixel) BMP file, shown in Figure 10-20 loaded into the GIMP 2.4 image processing software (available from http://www.gimp.org/).

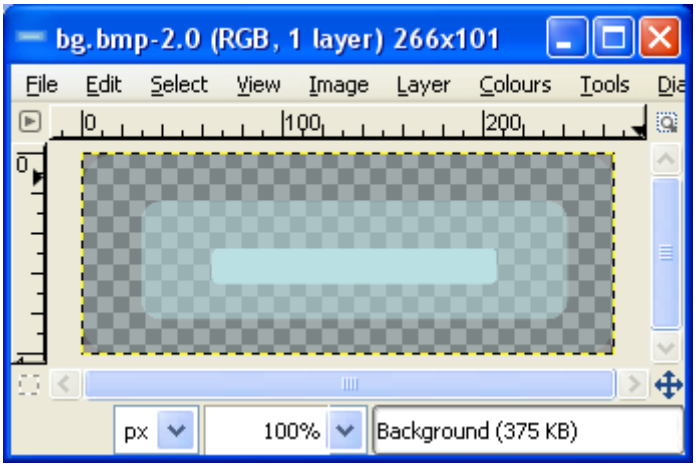


Figure 10-20 The Translucent Bitmap.

The check board background in Figure 10-20 indicates that the rounded rectangles are translucent (apart from the central one).

This application is our most complicated, involving three classes and a new J/Invoke data structure, as shown in the class diagrams in Figure 10-21.

 © Gayathri Singh and Andrew Davison 2008

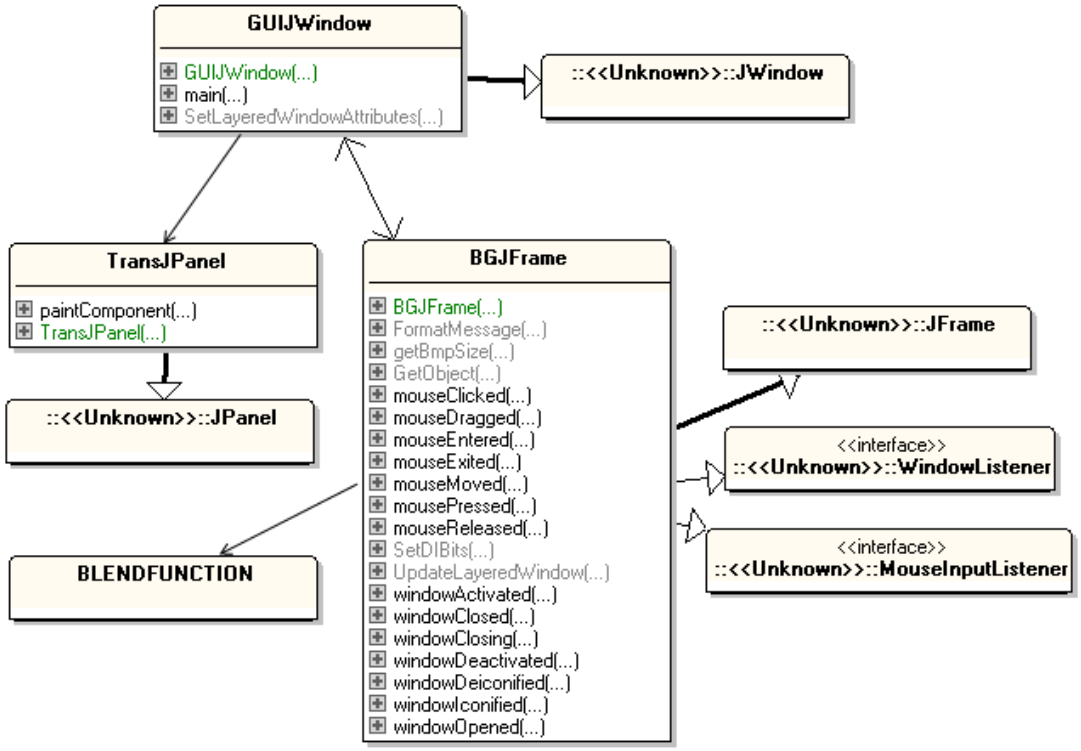


Figure 10-21 Class Diagrams for the GUIJWindow Application.

GUIJWindow initializes the GUI components (the button and label), and places them inside a TransJPanel instance. TransJPanel's main task is to make the JWindow completely transparent by painting the panel in a transparency color. GUIJWindow uses User32.SetLayeredWindowAttributes() and the LWA_COLORKEY flag to specify the color, a technique first employed back in Hack 10.6.

BGJFrame loads a translucent bitmap, and uses the Win32 API's layered windows and User32.UpdateLayeredWindow() to draw it onto the frame, retaining the per-pixel translucency of the bitmap.

The frame is undecorated, so movement, iconification, and termination are managed by mouse listeners, and iconification and exiting are offered via a popup menu. (This is the same approach as seen in Hacks 10.4 and 10.6.) BGJFrame is also a WindowListener so it can detect when it is iconified and deiconified.

This application is unusual because it consists of two windows: a JFrame (BGJFrame) and JWindow (GUIJWindow). BGJFrame retains a reference to the GUIJWindow instance, so the JWindow can be moved when the JFrame is moved, and be made invisible/visible when the JFrame is iconified/deiconified.

The reason for this two-window approach is a 'feature' of the Win32 API User32.UpdateLayeredWindow() function used by GUIJFrame. When the function draws a translucent bitmap to the screen, any children of the JFrame (e.g. GUI components) are rendered invisible! This means that if we want the translucent window to have visible components, they must be managed by a separate window, which is the role played by GUIJWindow.

This may all seem rather complicated, but the separation of tasks between the three classes (GUIJWindow, TransJPanel, and BGJFrame) makes it relatively easy to reuse

 © Gayathri Singh and Andrew Davison 2008

this design for other translucent, shaped applications. Changing the translucent background only requires the BMP file to be altered. Changing the GUI components only requires modifications to the GUI code inside GUIJWindow.

Managing the GUI Components

GUIJWindow creates the GUI controls in the usual way, but places them inside a TransJPanel instance; it also creates the translucent BGJFrame.

```
// global
private BGJFrame bgFrame;


public GUIJWindow()
{
  bgFrame = new BGJFrame("Translucency Example", this);

  JInvoke.initialize();
  makeGUI( bgFrame.getBmpSize() );
  pack();

  transWin();
  setLocation( bgFrame.getLocation() );
  setVisible(true);
}  // end of GUIJWindow()
```

The makeGUI() method is almost the same as previously, except that the controls are added to a TransJPanel instance, created using:

```
// inside makeGUI()
TransJPanel p = new TransJPanel(bmpSize, TRANSPARENT_COLOR);
```

The panel's dimensions are specified using bmpSize, and it paints over itself using the color defined in TRANSPARENT_COLOR:

```
// global in GUIJWindow
private static final int TRANSPARENT_COLOR = 0x0000ff;
                                  // red (format is 0xbbggrr)
```

The color is specified using the Win32 color reference format. TransJPanel converts this into a Java Color object before using it for painting. Red is employed as the transparency color since it isn't used by any of the GUI components, and so they won't be affected by the transparency. (This is not quite true, as discussed in the "A Problem with Per-Pixel Transparency" section below.)

The bmpSize value comes from BGJFrame, and is the size of the its loaded bitmap. Sharing the same dimensions ensures that the JFrame, JWindow, and JPanel are all the same size.

The location of GUI JWindow is set to be the same as the BGJFrame, with the line:

```
setLocation( bgFrame.getLocation() );
```

 © Gayathri Singh and Andrew Davison 2008

Setting Translucency and Transparency

GUIJWindow calls transWin() to set the pixel transparency colour, and also to apply a global 50% translucency effect to the components. This reduces their visibility, making them blend more naturally with the per-pixel translucent background drawn by BGJFrame.

```
// globals
@NativeImport(library="user32")
public static native boolean SetLayeredWindowAttributes(int hwnd,
                      int colorRef, byte bAlpha, int dwFlags);
   // see http://msdn2.microsoft.com/en-us/library/ms633540.aspx

private static final byte TRANSLUCENCY = (byte) 128;
                                            // 50% translucency

private static final int WS_EX_LAYERED = 0x80000;
private static final int LWA_COLORKEY = 1;    // 0x1
private static final int LWA_ALPHA = 2;    // 0x2


private void transWin()
{
  int hwnd =  Util.getWindowHandle(this);

  /* add layered window support to the window so it can be made
     translucent (alpha-blended) and use pixel transparency  */
  int winStyleInfo =
         User32.GetWindowLong(hwnd, WinConstants.GWL_EXSTYLE);
  winStyleInfo |= WS_EX_LAYERED;
  User32.SetWindowLong(hwnd, WinConstants.GWL_EXSTYLE, winStyleInfo);

  SetLayeredWindowAttributes(hwnd, TRANSPARENT_COLOR, TRANSLUCENCY,
                                       LWA_COLORKEY|LWA_ALPHA);
} // end of transWin()
```

This code may be familiar, since we used the same approach in Hack 10.5.

Positioning Components and Handling Transparency

BGJWindow places its GUI components in TransJPanel, which puts them into a horizontal line with a border to indent them from the shaped edges of the application. TransJPanel is set to be the same size as the translucency BGJFrame.

```
// globals
private static final int BORDER = 15;

private BufferedImage bImage;   // image used to cover the panel


public TransJPanel(Dimension bmpSize, int tColor)
{
  super();
  setPreferredSize(bmpSize);

  Color transColor = convertTransColor(tColor);
  bImage = buildImage(bmpSize, transColor);
```

 © Gayathri Singh and Andrew Davison 2008

```
  setLayout( new BoxLayout(this, BoxLayout.LINE_AXIS));
  setBorder(new EmptyBorder(BORDER, BORDER, BORDER, BORDER));
  setOpaque(false);
}  // end of TransJPanel()
```

The transparency color (tColor) is converted into a Color object, and used to generate an image the same size as the panel.

```
private Color convertTransColor(int tColor)
// convert tColor (format is 0xbbggrr) into a Color object
{
  int blue = (tColor >> 16) & 0xFF;
  int green = (tColor >> 8) & 0xFF;
  int red = tColor & 0xFF;

  Color transColor = new Color(red, green, blue);
  return transColor;
}  // end of convertTransColor()


private BufferedImage buildImage(Dimension bmpSize, Color transColor)
// generate a rectangular image of size bmpSize and color transColor
{
  BufferedImage im =
          new BufferedImage(bmpSize.width, bmpSize.height,
                                  BufferedImage.TYPE_INT_ARGB);
  Graphics2D g2d = im.createGraphics();
  g2d.setColor(transColor);
  g2d.fillRect(0, 0, bmpSize.width, bmpSize.height);
  g2d.dispose();
  return im;
}  // end of buildImage()
```

The image is drawn over the panel by its paintComponent() method:

```
public void paintComponent(Graphics g)
{  g.drawImage(bImage, 0, 0, this); }
```

The image is a solid red rectangle, but since red is the transparency color, this rectangle will ‘erase’ the background.

A simpler strategy is to utilize the transparency color as the background color of the panel:

```
setBackground(transColor);
```

This erases the background but also makes the panel ‘solid’ in the sense that mouse clicks and drags will be sent to TransJPanel instead of down to BGJFrame. We want BGJFrame to handle the mouse, and so paint an image instead.

A Problem with Per-Pixel Transparency

The window’s background is made invisible by painting a red rectangle behind all the components in TransJPanel. It works, as Figure 10-19 illustrates, but there’s an irritating problem related to how some components are drawn.

 © Gayathri Singh and Andrew Davison 2008

The "Press Count: " text is drawn in a JLabel, which normally has an invisible background. Java uses anti-aliasing to smooth the text in the label, which basically blends the text with the background color. This causes some of the red background pixels to be darkened so they are no longer made invisible, as shown in Figure 10-22.



Figure 10-22 Text with a Red Tinge

Our solution is to make the JLabel opaque, causing its text to be smoothed against its own background:

```
// global
private JLabel pressInfo;

// in makeGUI() in GUIJFrame
pressInfo = new JLabel("Press Count: 0 ");
pressInfo.setOpaque(true);
```

Dealing with Translucency and Mouse Actions

BGJFrame loads a translucent bitmap, and draws it onto the frame with the User32.UpdateLayeredWindow() function.

As with the frames used in Hacks 10.4 and 10.6, BGJFrame is undecorated, so movement, iconification, and termination are managed by mouse listeners, and iconification and exiting are offered via a popup menu. BGJFrame is also a WindowListener so it can detect when it is iconified and deiconified.

The listener code is slightly more complicated than in the previous hacks since it must also move and control the visibility of GUIJWindow. In addition, the listener-generated popup menu must be positioned carefully or a ‘feature’ of UpdateLayeredWindow() will cause it to be rendered invisible.

Manipulating the Bitmap

BGJFrame simply loads and draw a bitmap. Unfortunately, the details are a little tricky, requiring several Win32 API functions. The code is divided between several methods, called from BGJFrame’s constructor:

```
// in BGJframe()
int hwnd =  com.jinvoke.Util.getWindowHandle(this);
layerWin(hwnd);
int hwdBmp = loadBitmap(IM_FNM);

bmpSize = getImageSize(hwdBmp);
```

 © Gayathri Singh and Andrew Davison 2008

```
setSize(bmpSize);  // change JFrame's size to match bitmap's size

modifyBitmap(hwdBmp, bmpSize);
drawBitmap(hwdBmp, bmpSize, hwnd);
```

We'll consider each of the highlighted methods in turn.

Loading the Bitmap

User32.LoadImage() is a general purpose function that can load icons, cursors, or bitmaps. This is reflected by the large number of flags available for the various arguments, which are detailed at http://msdn2.microsoft.com/en-us/library/ms648045.aspx. Fortunately, loading a bitmap file is fairly straightforward, as shown in the loadBitmap() method from BGJFrame.

```
// global
private static final String IMAGES_DIR = "images/";


private int loadBitmap(String fnm)
// load the image from fnm as a bitmap
{
  System.out.println("Loading background bitmap from " +
                                              IMAGES_DIR + fnm);
  int hwdBmp = User32.LoadImage(WinConstants.NULL, IMAGES_DIR + fnm,
                          WinConstants.IMAGE_BITMAP, 0, 0,
                          WinConstants.LR_LOADFROMFILE );
  if (hwdBmp == WinConstants.NULL)
    System.out.println("Image not loaded");
  return hwdBmp;
}  // end of loadBitmap()
```

What Size is the Bitmap?

Surprisingly, it's somewhat harder to extract the bitmap's dimensions than to load it. The BITMAP data structure has to be accessed via a pointer, using the Gdi32.GetObject() function. It's defined at http://msdn2.microsoft.com/en-us/library/ms533268.aspx as:

```
int GetObject(
  HGDIOBJ hgdiobj,  // handle to graphics object
  int cbBuffer,     // size of buffer for object information
  LPVOID lpvObject  // buffer for object information
);
```

The type of the pointer to the buffer is LPVOID which reflects the way that GetObject() can be used to return information on a variety of graphical objects: lbvObject can end up pointing to a structs such as BITMAP, DIBSECTION, and EXTLOGPEN.

We're only interested in getting a BITMAP struct, so we redefine GetObject() to take a pointer to a J/Invoke Bitmap object:

 © Gayathri Singh and Andrew Davison 2008

```
@NativeImport(library="Gdi32")
public static native int GetObject(int hnd, int bSize, Bitmap[] ptr);
     // third argument redefined to point to a Bitmap
```

We have to dereference the pointer, and extract the struct's width and height fields. BITMAP is documented at http://msdn2.microsoft.com/en-us/library/ms532298(VS.85).aspx

All this is done by a getImageSize() method in BGJFrame:

```
private Dimension getImageSize(int hwdBmp)
// return the dimensions (width, height) of the bitmap
{
  Dimension d = new Dimension();

  // get size of Bitmap struct
  int bitmapStructSize = Util.getStructSize(Bitmap.class);

  Bitmap[] bmp = new Bitmap[1];
  bmp[0] = new Bitmap();   // create an empty Bitmap

  // fill in the Bitmap struct
  int res = GetObject(hwdBmp, bitmapStructSize, bmp);
  if (res != 0) {  // extract bitmap dimensions
    d.width = (int)bmp[0].bmWidth;
    d.height = (int)bmp[0].bmHeight;
  }
  return d;
} // end of getImageSize()
```

Modifying the Bitmap

The drawing of the bitmap only succeeds if its format is correct. In particular, the alpha blending carried out in drawBitmap() requires the bitmap to be 32 bpp, and the RGB channels of its pixels to be pre-multiplied by their corresponding alphas.

The 32 bpp requirement is satisfied by ensuring that the GIMP graphics application saves the bitmap with an alpha channel. Pre-multiplication requires some processing of the bitmap after it has been loaded, which is carried out by modifyBitmap().

modifyBitmap() calls Gdi32.GetDIBits() twice, manipulated the pixels, then calls Gdi32.SetDIBits() to return the modified pixels to the bitmap. These functions are described at http://msdn2.microsoft.com/en-us/library/ms532334(VS.85).aspx and http://msdn2.microsoft.com/en-us/library/ms532281(VS.85).aspx.

```
// global
@NativeImport(library="Gdi32")
public static native int SetDIBits(int hdc, int hbmp, int uStartScan,
               int cScanLines, int[] lpvBits, BitmapInfo lpbmi,
               int fuColorUse);
     // 8th argument redefined to BitmapInfo from int


private void modifyBitmap(int hwdBmp, Dimension d)
{
```

 © Gayathri Singh and Andrew Davison 2008

```
  BitmapInfo bmi = new BitmapInfo();
  bmi.bmiHeader.biSize = Util.getStructSize(BitmapInfoHeader.class);

  int hDC = Gdi32.CreateCompatibleDC(0);

  // extract bitmap info from the loaded bitmap, storing it in bmi
  int res1 = Gdi32.GetDIBits(hDC, hwdBmp, 0, d.height, null, bmi,
                                      WinConstants.DIB_RGB_COLORS);
  // System.out.println("1st GetDIBits result: " + res1);
                                              // should be 1

  // create an empty array for the bitmap's pixel data
  int[] pixels = new int[bmi.bmiHeader.biWidth *
                         bmi.bmiHeader.biHeight];

  bmi.bmiHeader.biBitCount = 32;  // the bitmap uses ARGB
  bmi.bmiHeader.biCompression = WinConstants.BI_RGB;
                                  // an uncompressed format

  /* extract the pixel data from the loaded bitmap,
     storing it in pixels[] */
  int res2 = Gdi32.GetDIBits(hDC, hwdBmp, 0, d.height,
                    pixels, bmi, WinConstants.DIB_RGB_COLORS);
  if (bmi.bmiHeader.biBitCount != 32)
    System.out.println("Bitmap bit count not 32!");
  // System.out.println("2nd GetDIBits result: " + res2);
            // should be height of bitmap

  // premultiply RGB channels of pixels by their alphas
  for (int i=0; i < pixels.length; i++)
    pixels[i] = preMultiplyPixel(pixels[i]);

  // store changed pixels back in the bitmap
  int res3 = SetDIBits(hDC, hwdBmp, 0, d.height,
                       pixels, bmi, WinConstants.DIB_RGB_COLORS);
  // System.out.println("SetDIBits result: " + res3);
                   // should be bitmap height
} // end of modifyBitmap()
```

The first call to Gdi32.GetDIBits() retrieves a BITMAPINFO struct containing information about the dimensions and color format of the device-independent bitmap (DIB for short). J/Invoke represents this struct using the BitmapInfo and BitmapInfoHeader classes. The second call to GetDIBits() uses the header information (with some additional data) to retrieve the bitmap's pixels as a long array of integers.

modifyBitmap() contains several commented out System.out.println() calls which report the results of the Win32 API function calls. These are very useful for determining if the functions are working correctly.

preMultiplyPixel() pulls apart an integer in the pixels[] array into its component ARGB channels, and multiplies the RGB values by the alpha value mapped from a range of 0 to 255 to between 0 and 1.

```
private int preMultiplyPixel(int pixel)
{
  // extract ARGB channels of the pixel
  int alpha = (pixel >> 24) & 0xFF;
```

 © Gayathri Singh and Andrew Davison 2008

```
  int red = (pixel >> 16) & 0xFF;
  int green = (pixel >> 8) & 0xFF;
  int blue = pixel & 0xFF;

  red = (int)(red * alpha / 255);
  green = (int)(green * alpha / 255);
  blue = (int)(blue * alpha / 255);

  return ((alpha << 24) | (red << 16) | (green << 8) | blue);
                       // new pixel
}  // end of preMultiplyPixel()
```

Drawing the Bitmap

The drawing operation utilizes per-pixel alpha-blending to combine the bitmap with the screen image underneath the window. This requires the initialization of a PIXELBLEND struct, and a call to User32.UpdateLayeredWindow(). Neither the struct nor the function are part of the version of J/Invoke we're using, so must be defined.

The BLENDFUNCTION type is defined at http://msdn2.microsoft.com/en-us/library/ms532306(VS.85).aspx:

```
typedef struct _BLENDFUNCTION {
  BYTE BlendOp;
  BYTE BlendFlags;
  BYTE SourceConstantAlpha;
  BYTE AlphaFormat;
} BLENDFUNCTION;
```

The J/Invoke version is a direct translation:

```
@NativeStruct
public class BLENDFUNCTION {
  public byte BlendOp;
  public byte BlendFlags;
  public byte SourceConstantAlpha;
  public byte AlphaFormat;
}
```

This is stored in BLENDFUNCTION.java.

The struct controls the blending of the source and destination bitmaps, which in our case will be the 32 bpp translucent bitmap and the screen.

User32.UpdateLayeredWindow() only supports the AC_SRC_OVER blend operation (stored in the BlendOp field), and BlendFlags is 0. The SourceConstantAlpha value must be 255 (opaque) when we use per-pixel blending.

The AlphaFormat value should be AC_SRC_ALPHA to denote that the bitmap has an alpha channel. BLENDFUNCTION assumes that the RGB channels of the source image have already been multiplied by the alpha channel value, which we did back in modifyBitmap().

These requirements result in the following code fragment in drawBitmap():

 © Gayathri Singh and Andrew Davison 2008

```
// globals
private static final byte AC_SRC_OVER = 0;  // 0x00
private static final byte AC_SRC_ALPHA = 1;  // 0x01;


// in drawBitmap()
BLENDFUNCTION pixelBlend = new BLENDFUNCTION();
pixelBlend.BlendOp = AC_SRC_OVER;
pixelBlend.BlendFlags = 0;
pixelBlend.SourceConstantAlpha = (byte) 255;
pixelBlend.AlphaFormat = AC_SRC_ALPHA;
```

UpdateLayeredWindow() is defined as:

```
@NativeImport(library="user32")
public static native boolean UpdateLayeredWindow(
      int hwnd, int hdcDst,
      com.jinvoke.win32.structs.Point pptDst, Size psize,
      int hdcSrc, com.jinvoke.win32.structs.Point pptSrc,
      int crKey, BLENDFUNCTION pixelBlend, int dwFlags);
```

The idea is to paint the window (hwnd) by blending the source bitmap (hdcSrc) with the screen (hdcDst) using the blend function in pixelBlend. hdcSrc and hdcDst are device contexts, which are similar to Java graphic contexts.

The device context for the bitmap must be compatible with the desktop since we're painting the image onto the desktop, leading to the following:

```
// in drawBitmap()
int desktopWnd = User32.GetDesktopWindow();
int desktopDC = User32.GetWindowDC(desktopWnd);
int hdcSrc = Gdi32.CreateCompatibleDC(desktopDC);
int res1 = Gdi32.SelectObject(hdcSrc, hwdBmp);
                      // put bitmap into source DC
```

The screen's device context is easier to obtain:

```
int screenDC = User32.GetDC(0);
```

UpdateLayeredWindow() also needs to be given the size of the resulting window (psize), even though we're not changing it, and the blending starting point in the source bitmap ((0,0) in pptSrc). The resulting call is:

```
// global
private static final int ULW_ALPHA = 2; // 0x02


// in drawBitmap()
Size psize= new Size();    // size of the resulting window
psize.cx = d.width; psize.cy = d.height;

com.jinvoke.win32.structs.Point pptSrc =
          new com.jinvoke.win32.structs.Point();
pptSrc.x = 0; pptSrc.y = 0;  // starting point in bitmap
```

 © Gayathri Singh and Andrew Davison 2008

```
boolean res2 = UpdateLayeredWindow(hwnd, screenDC, null, psize,
                       hdcSrc, pptSrc, 0, pixelBlend, ULW_ALPHA);
if (!res2)
  System.out.println("UpdateLayeredWindow failed");
```

The ULW_ALPHA flag indicates that the blend function should be used rather than the crKey argument (which is set to 0).

More details on User32.UpdateLayeredWindow() can be found at http://msdn2.microsoft.com/en-us/library/ms633556.aspx

Listening for Mouse and Window Events

A quick look back at the class diagram for BGJFrame in Figure 10-21 shows that it's both a mouse and window listener. As in Hack 10.4, the mouse listening handles window movement, iconification, and termination, with iconification and exiting available via a popup menu.

The mouse listening code is much the same as in Hack 10.4, except that BGJFrame must ensure that the GUIJWindow stays in front of the JFrame when it's moved, which is done by calling it's Window.setLocation() and Window.toFront() methods.

BGJFrame is a WindowListener so it can detect when it's iconified and deiconified, and then hide (or show) GUIJWindow:

```
// global
private GUIJWindow guiWin;   // window holding the GUI components


public void windowDeiconified(WindowEvent e)
{ guiWin.setVisible(true); }
     // reappearance of JFrame makes GUI window visible


public void windowIconified(WindowEvent e)
{  guiWin.setVisible(false); }
    // disappearance of JFrame makes GUI window invisible
```

The most significant code change is to the positioning of the popup menu, due to an unfortunate feature of User32.UpdateLayeredWindow(). Any child components of the JFrame are rendered invisible by UpdateLayeredWindow(), and this includes JPopupMenu objects.

However, a popup menu is only turned invisible if it pops up entirely over the JFrame; if some part of it extends beyond the JFrame's edge then the menu will be drawn. Using this 'loophole', we modified our showPopup() method to check the popup menu's position, and move it slightly if necessary.

```
// in showPopup() in BGJFrame
java.awt.Point loc = positionMenu( e.getPoint(),
                              popup.getPreferredSize());
popup.show( e.getComponent(), loc.x, loc.y);
```

 © Gayathri Singh and Andrew Davison 2008

positionMenu() check the position of the popup menu relative to the window, and possibly adjusts that position.

```
private java.awt.Point positionMenu(java.awt.Point loc,
                                            Dimension menuSz)
{
  Rectangle popRect = new Rectangle(loc, menuSz);
                                           // popup menu rectangle
  Rectangle bgRect = new Rectangle(0, 0, getWidth(), getHeight());
                                           // JFrame rectangle
  if (bgRect.contains(popRect)) {
    /* menu will not be visible in this case
       so move popup location to over an edge */
    if (loc.x < loc.y)
      loc.x = -1;
    else
      loc.y = -1;
  }
  return loc;
}  // end of positionMenu()
```

A Consumer JRE Per-Pixel Translucent Window

The Consumer JRE hides all the nasty low-level details of per-pixel translucency behind a single method, AWTUtilities.setWindowOpaque(). setWindowOpaque(JFrame, false) also stops the frame's background from being rendered, but any components (JPanel, JButton, etc.) will remain visible.

Per-pixel translucency is utilized by painting operations called from the paintComponent() method of a JPanel inside the JFrame. The panel must be non-opaque and have double buffering turned off.

This coding approach is illustrated by the class diagrams for our example application, PPTransJFrame.java, in Figure 10-23.

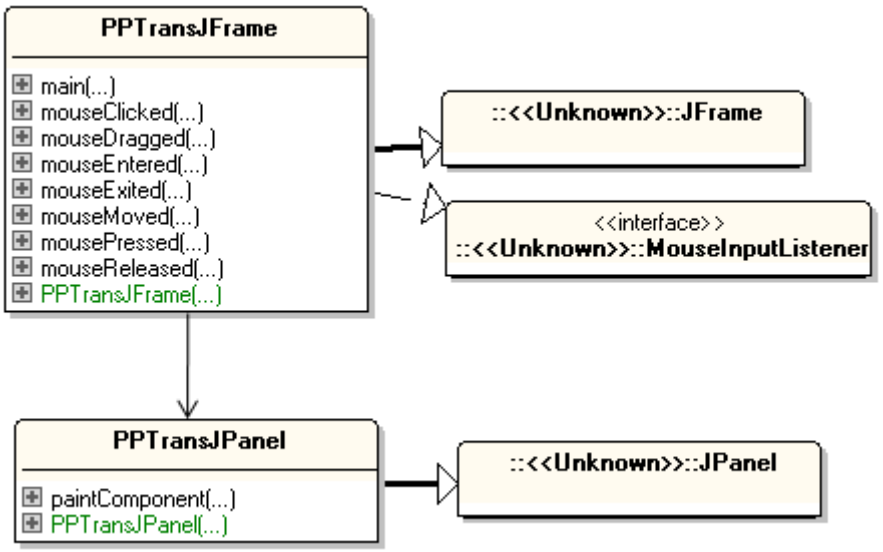


Figure 10-23. Class Diagrams for PPTransJFrame.

 © Gayathri Singh and Andrew Davison 2008

PPTransJFrame is an undecorated JFrame which uses the now-familiar mouse listener approach to respond to mouse dragging, and has a popup menu for iconification and termination. PPTransJFrame calls AWTUtilities.setWindowOpaque() to make itself transparent and turn on per-pixel translucency.

The button and label are placed inside a PPTransJPanel instance which puts an empty border around the components and paints the translucent background. Our PPTransJPanel class illustrates two ways of creating the background: by loading an existing translucent image from a PNG file and drawing it (see Figure 10-24), and by dynamically painting translucent shapes using Graphics2D operations (Figure 10-25).

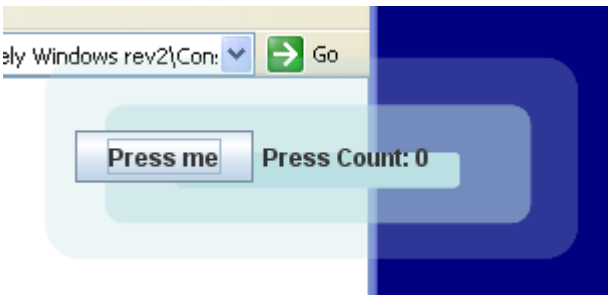


Figure 10-24. Translucency Using a PNG Image

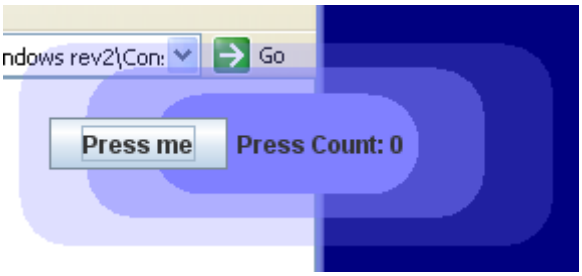


Figure 10-25. Translucency Using Painting Operations.

Creating the Window

The PPTransJFrame() constructor sets up the mouse listeners, creates the GUI, initializes the frame, and calls AWTUtilities.setWindowOpaque(). It's important that the call be made after the window has become displayable, but it doesn't need to be visible.

```
public PPTransJFrame()
{
  super("\"Press Me\" Button");

  // add listeners for iconification, closing, and dragging
  addMouseListener(this);
  addMouseMotionListener(this);

  setUndecorated(true);
  makeGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();
```

 © Gayathri Singh and Andrew Davison 2008

```
  AWTUtilities.setWindowOpaque(this, false);
    // make window transparent, and switch on per-pixel translucency

  setResizable(false);
  setLocationRelativeTo(null);
  setVisible(true);
}  // end of PPTransJFrame()
```

makeGUI() adds the button and label to an instance of PPTransJPanel, created like so:

```
PPTransJPanel p = new PPTransJPanel();
```

The mouse listeners are the versions we met back in Hack 10.4. There's no need for additional code to move a JWindow or to employ a window listener, which we had to include in the GUIJWindow example.

Painting the Panel

If the PPTransJPanel paints a translucent image loaded from a file, then its constructor is as follows:

```
// globals
private static final int BORDER = 15;
private static final String BACK_IM = "bg2.png";
                                    // image drawn onto panel
private BufferedImage bImage;


public PPTransJPanel()
{
  super();

  bImage = loadImage(BACK_IM);
  int imWidth = bImage.getWidth(null);
  int imHeight = bImage.getHeight(null);
  setPreferredSize( new Dimension(imWidth, imHeight));

  setLayout( new BoxLayout(this, BoxLayout.LINE_AXIS));

  setBorder(new EmptyBorder(BORDER, BORDER, BORDER, BORDER));
     //indent the components from the panel's edge.

  setOpaque(false);
  setDoubleBuffered(false);
}  // end of PPTransJPanel()
```

If the panel is going to paint its own shapes then there's no need to call loadImage(), but the dimensions of the panel should still be specified.

The panel must be made non-opaque and have its double buffering turned off.

loadImage() is straightforward, thankfully lacking direct pixel manipulation and blend functions.

```
private BufferedImage loadImage(String fnm)
{
  BufferedImage im = null;
```

 © Gayathri Singh and Andrew Davison 2008

```
  try {
    im = ImageIO.read( new File("images/" + fnm));
  }
  catch(Exception e)
  {  System.out.println("images/" + fnm + " not found");  }

  return im;
}  // end of loadImage()
```

The image is a standard PNG file, with no need to convert it to a 32 bpp bitmap.

The panel's paintComponent() method draws the image:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  g.drawImage(bImage, 0, 0, this);
}
```

If the panel paints its own translucent shapes, all the usual Java 2D shapes and drawing methods are available, with the only trick being to use colors that have alpha components. The following version of paintComponent() draws the rounded rectangles shown in Figure 10-25.

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);

  Graphics2D g2d = (Graphics2D)g;
  int w = getWidth();
  int h = getHeight();

  // draw 3 blueish rounded rectangles with increasing opaqueness
  g2d.setPaint( new Color(128, 128, 255, 64) );
                          // alpha == 64 == 25% translucent
  RoundRectangle2D roundedRect =
              new RoundRectangle2D.Float(0, 0, w, h, 50, 50);
  g2d.fill(roundedRect);

  g2d.setPaint( new Color(128, 128, 255, 128) );
                         // alpha == 128 == 50% translucent
  roundedRect =
        new RoundRectangle2D.Float(w/8, h/8, w*3/4, h*3/4, 50, 50);
  g2d.fill(roundedRect);

  g2d.setPaint( new Color(128, 128, 255) );
                         // alpha == 255 == fully opaque
  roundedRect =
        new RoundRectangle2D.Float(w/4, h/4, w/2, h/2, 50, 50);
  g2d.fill(roundedRect);
}  // end of paintComponent()
```

The simplicity of this per-pixel translucency approach, especially when compared to the direct use of the Win32 API, is a strong reason for upgrading to the Consumer JRE, or to Java SE 7 when it's released.

 © Gayathri Singh and Andrew Davison 2008

Hack 10.7 Windows that Close Gradually (medium level)

Close an application by making it fade away, or by cutting away increasing large pieces of its window.

The effects described in Hack 10.1 and Hack 10.2 can be employed to make window closing more like a graceful departure instead of a sudden disappearance.

For example, when the application's close box is clicked, the window's translucency can be increased over a few seconds, to make it fade away. Alternatively, the window's shape region can be steadily reduced, to cut away ever-increasing portions of the window.

We'll explain how to implement these two effects in this hack.

Effect 1: Fading Away

Once the application's close box is pressed, the window begins to fade away, disappearing after about 5 seconds (as shown in Figure 10-26).



Figure 10-26. Fading when Closed.

One aspect of this effect is that the application remains operational during the fade. This means that the user can keep pressing the "Press Me" button to increment the counter. If this behavior isn't desirable then the button needs to be disabled when the fading commenced (with JComponent.setEnabled(false)).

The constructor for the CloseFadeWin class sets up a window closing listener, and initializes the layered window support for translucency.

```
public CloseFadeWin()
{
  super("\"Press Me\" Button");

  JInvoke.initialize();
  makeGUI();

  setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { animatedClose(); }
  });

  pack();
  setResizable(false);

  setWinLayering();

  setLocationRelativeTo(null);
  setVisible(true);
```

 © Gayathri Singh and Andrew Davison 2008

```
} // end of CloseFadeWin()
```

The JFrame.DO_NOTHING_ON_CLOSE flag overrides the JFrame's default behavior which is to make the window invisible. Instead, animatedClose() will progressively increase the window's translucency and then exit.

setWinLayering() is a version of fadeWin() described in Hack 10.1. It adds layered support to the window so it can be alpha-blended, and sets the initial translucency to 255 (completely opaque).

```
// globals
private int winHandle;


private void setWinLayering()
{
  winHandle = Util.getWindowHandle(this);

  // add layered window support to the window
  int winStyleInfo = User32.GetWindowLong(winHandle,
                              WinConstants.GWL_EXSTYLE);
  winStyleInfo |= WS_EX_LAYERED;
  User32.SetWindowLong(winHandle, WinConstants.GWL_EXSTYLE,
                                              winStyleInfo);
  setTranslucency(255);   // set window to be fully opaque
}  // end of setWinLayering()


private void setTranslucency(int translucency)
{ SetLayeredWindowAttributes(winHandle, 0,
                             (byte)translucency, LWA_ALPHA);
}
```

Animated Closing

The animatedClose() method called from windowClosing() sets up a timer that triggers a task every ANIM_STEP ms until the time reaches ANIM_DURATION. The task adjusts the window's translucency by calling animStep().

```
// globals
private static final int ANIM_DURATION = 5000;  // ms
private static final int ANIM_STEP = 500;       // ms

private boolean isClosing = false;
private double currAnimTime = 0;
private Timer animTimer;


private void animatedClose()
{
  if (isClosing)
    return;

  // task to be executed by the timer
  ActionListener animTask = new ActionListener() {
    public void actionPerformed(ActionEvent e)
```

 © Gayathri Singh and Andrew Davison 2008

```
    {
      currAnimTime += ANIM_STEP;
      if (currAnimTime >= ANIM_DURATION) {   // time to stop
        animTimer.stop();
        System.exit(0);
      }
      else   // make another change
        animStep();
    }
  };

  isClosing = true;
  animTimer = new Timer(ANIM_STEP, animTask);
  animTimer.start();    // start the timer
}  // end of animatedClose()
```

The isClosing() boolean is used to prevent multiple invocations of the timer if the user presses the close box more than once.

animStep() translates the current animation time, which increases from 0 up to ANIM_DURATION, into a translucency value (starting at 255 and decreasing towards 0).

```
private void animStep()
{
  int translucency = 255 - (int)(255*currAnimTime/ANIM_DURATION);
  setTranslucency(translucency);
}
```

Effect 2: Cutting Away the Window

When the application's close box is pressed, the window is progressively 'cut away' until it disappears. The cutting starts from the window's right edge and moves to the left (as shown in Figure 10-27).

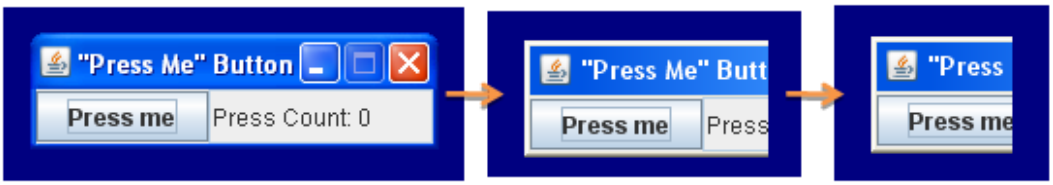


Figure 10-27. Cutting Away when Closed.

Figure 10-27 clearly shows the region issue related to visual style – when the regions start being applied the window's visual style changes from "Windows XP" to "Windows Classic" (most clearly seen in the edges of the title bar).

The constructor for the CloseCutWin class is very similar to the CloseFadeWin constructor: it sets up a window closing listener, and initializes the data needed for region support.

```
public CloseCutWin()
```

 © Gayathri Singh and Andrew Davison 2008

```
{
  super("\"Press Me\" Button");

  JInvoke.initialize();
  makeGUI();

  setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { animatedClose(); }
  });

  pack();
  setResizable(false);

  getRegionDetails();

  setLocationRelativeTo(null);
  setVisible(true);
} // end of CloseCutWin()
```

getRegionDetails() stores the window's handle, dimensions, and a rectangular region for the entire window.

```
// globals
private int winHandle;
private int winWidth, winHeight;
private int winRegionHandle;  // region for the entire window


private void getRegionDetails()
{
  winHandle = Util.getWindowHandle(this);
  winWidth = getWidth();
  winHeight = getHeight();
  winRegionHandle = Gdi32.CreateRectRgn(0, 0, winWidth, winHeight);
}  // end of getRegionDetails()
```

animatedClose() is unchanged from the CloseFadeWin example – the effect-specific code is located in animStep(), which is executed every ANIM_STEP ms for a total of ANIM_DURATION ms.

animStep() cuts away the window on its right side by creating a rectangular 'cut' region for that part of the window, and combining it with the region for the entire window (winRegionHandle) using set difference.

```
private void animStep()
{
  int width = (int)(winWidth*(ANIM_DURATION -
                                  currAnimTime)/ANIM_DURATION);
  int cutHandle = Gdi32.CreateRectRgn(width,0, winWidth,winHeight);
     // shape for the 'cut away' part of the window

  int visibleHandle = Gdi32.CreateRectRgn(0,0,0,0); // dummy value
  Gdi32.CombineRgn(visibleHandle, winRegionHandle, cutHandle,
                                WinConstants.RGN_DIFF); // set diff
  User32.SetWindowRgn(winHandle, visibleHandle, true);
```

 © Gayathri Singh and Andrew Davison 2008

```
  Gdi32.DeleteObject(cutHandle);
  Gdi32.DeleteObject(visibleHandle);
}  // end of animStep()
```

The size of the 'cut' region depends on the width variable, which is obtained by mapping the current animation time, into a value decreasing from winWidth down to 0.

The Animated Window API

Win32 provides an AnimateWindow() function (see http://msdn2.microsoft.com/en-us/library/ms632669(VS.85).aspx) that can provide animated transition effects when a window is shown or hidden. The transitions include: rolling, sliding, window collapsing or expansion, and alpha-blending fade. Why didn't we use one of those?

Unfortunately, AnimateWindow() doesn't work very well with Java, turning the window black before starting a transition. This is probably due to the function's underlying use of Win32's WM_PRINT and WM_PRINTCLIENT messages. It seems likely that Java windows don't support WM_PRINTCLIENT, a problem that has also occurred in Visual Basic programs in the past, as explained at http://www.xtremevbtalk.com/showthread.php?t=204486

Another issue may be threading: AnimateWindow() is supposed to be called from the thread that owns the window. Java manages the window in an AWT Window thread , but our code is invoked from the AWT Event thread.

Hack 10.8 Window Makeovers (low level)

Give your application a makeover with Java's built-in look and feels (L&Fs), or with third-party L&Fs, such as Substance and Napkin. J/Invoke isn't needed in this hack.

You may be feeling (like me) that you've seen rather too much of the "Press Me" Button example. Is there a quick, easy way of enlivening its appearance, without a lot of GUI coding? The answer is *yes*, by using Java's pluggable look and feel (L&F).

Most L&F capabilities are accessed through Java's UIManager class: applications can query which L&Fs are present with UIManager.getInstalledLookAndFeels(), and change the application's L&F with UIManager.setLookAndFeel().

Every platform has a native L&F which closely parallels the appearance of native OS applications. UIManager.getSystemLookAndFeelClassName() returns its name.

There's a cross-platform L&F, a Java-specific look and feel that's the same on every platform. It's name is obtained with UIManager.getCrossPlatformLookAndFeelClassName().

For this hack, LAFWin.java is the name of our favorite "Press Me" Button application. Functionally, it's the same as previously, but it now allows the user to change its L&F. Figure 10-28 show the application sporting the native, window-specific, and Motif L&Fs.

 © Gayathri Singh and Andrew Davison 2008

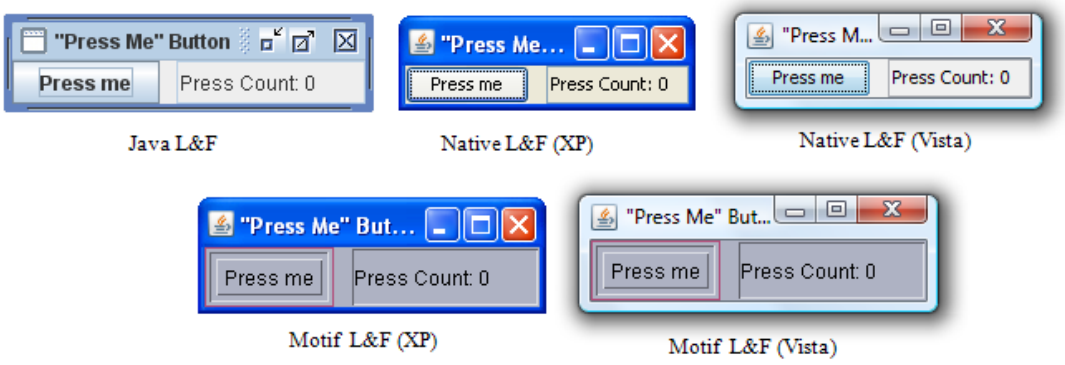


Figure 10-28. Five Different L&Fs for the Same Application.

LAFWin is executed from the command line with a numerical argument which specifies which L&F to employ. For example:

```
>  java LAFWin 1
```

This utilizes the Java L&F for LAFWin.

A sharp-eyed reader will notice that the original "Press Me" Button application shown back in Figure 10-1 isn't the same as any of the L&Fs in Figure 10-28 (although it's closest to the Java L&F, apart from the title bar). We'll explain why in a moment.

The LAFWin class (its fields, constructor, and methods) are unchanged from the original application. All the modifications lie in the main() function that invokes the LAFWin instance.

```
public static void main( String args[] )
{
  listLookAndFeels();

  if (args.length == 1)
    processLAFArg(args[0]);

  showCurrentLookAndFeel();

  JFrame.setDefaultLookAndFeelDecorated(true);
    // request that window decorations be done by the current L&F

  new LAFWin();
}  // end of main()
```

Applications often manipulate their L&F in main() since the changes must be made before a JFrame is created.

main() calls listLookAndFeels() to print the JVM's available L&Fs to stdout. Then LAFWin's command line argument is passed to processLAFArg() which changes the look and feel. The new setting is reported by showCurrentLookAndFeel().

If LAFWin is called without an argument, the program executes with the default L&F (which is the Java L&F for Sun's JVM).

The call to JFrame.setDefaultLookAndFeelDecorated() requests the JVM to kindly draw the entire window itself using Swing. The default Swing behavior is to leave the

 © Gayathri Singh and Andrew Davison 2008

title bar and window frame to be painted by the OS. This call is only a request, and won't be granted if the L&F doesn't support window decoration.

The call to setDefaultLookAndFeelDecorated() explains the difference between Figure 10-1 and the left-most image in Figure 10-28. Both use the Java L&F, but the original application didn't request window decoration by Swing, so it's title bar is drawn by MS Windows.

When the Java L&F is used with Swing window decoration, the issue with regions changing the visual style to "Windows Classic" (e.g. see Hack 10.2 and Hack 10.7, effect 2) does not occur since all the decoration is done by Java rather than the OS.

There's also a JDialog.setDefaultLookAndFeelDecorated() method which requests that Swing decorates its own dialog boxes, but we've not used it here.

Listing the Look and Feels

listLookAndFeels() utilizes UIManager.getInstalledLookAndFeels(), and several methods from UIManager.LookAndFeelInfo:

```
private static void listLookAndFeels()
{
  UIManager.LookAndFeelInfo[] lafsInfo =
                           UIManager.getInstalledLookAndFeels();
  System.out.println("\nJVM Look and Feels: (" +
                                    lafsInfo.length + ")");
  for(UIManager.LookAndFeelInfo info : lafsInfo)
    System.out.println( info.getName() + ": " + info.getClassName());
  System.out.println();
}  // end of listLookAndFeels()
```

The output on a test machine is:

```
JVM Look and Feels: (4)
Metal: javax.swing.plaf.metal.MetalLookAndFeel
CDE/Motif: com.sun.java.swing.plaf.motif.MotifLookAndFeel
Windows: com.sun.java.swing.plaf.windows.WindowsLookAndFeel
Windows Classic:
         com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel
```

Metal is the name of the Java L&F.

Processing the Look and Feel Choice

The integer entered on the command line should be 0, 1, or 2: 0 corresponds to the native L&F, 1 is for the Java L&F, and 2 for Motif. If the number isn't one of these then the Java L&F is employed by default.

```
private static void processLAFArg(String arg)
// change the command line argument into a L&F setting
{
  int lafVal = 1;    // default is Java L&F
  try {
    lafVal = Integer.parseInt(arg);
```

 © Gayathri Singh and Andrew Davison 2008

```
  }
  catch(Exception e)
  { System.out.println("L&F argument is not an integer");  }

  System.out.println("lafVal: " + lafVal);

  if (lafVal == 0)
    setNativeLookAndFeel();
  else if (lafVal == 1)
    setJavaLookAndFeel();
  else if (lafVal == 2)
    setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  else
    System.out.println("Do not recognize L&F number: " + lafVal);
}  // end of processLAFArg()
```

The set methods for the different L&Fs are quite similar – they call a UIManager method to make the change, and surround that call with try-catch blocks in case of failure.

```
private static void setNativeLookAndFeel()
{
  try {
    UIManager.setLookAndFeel(
                UIManager.getSystemLookAndFeelClassName());
    System.out.println("Using native L&F");
  }
  catch(Exception e)
  {  System.out.println("Error setting native L&F: " + e); }
}  // end of setNativeLookAndFeel()


private static void setJavaLookAndFeel()
{
  try {
    UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
    System.out.println("Using Java L&F");
  }
  catch(Exception e)
  {  System.out.println("Error setting Java L&F: " + e);  }
}  // end of setJavaLookAndFeel()


private static void setLookAndFeel(String className)
{
  try {
    UIManager.setLookAndFeel(className);
    System.out.println("Using " + className + " L&F");
  }
  catch(Exception e)
  {  System.out.println("Error setting " + className + " L&F: "+e);}
}  // end of setLookAndFeel()
```

Note that setLookAndFeel() takes a String argument for the class name; this can be obtained using UIManager.LookAndFeelInfo.getClassName(), as shown in the listLookAndFeels() method above.

 © Gayathri Singh and Andrew Davison 2008

What Do We End up With?

showCurrentLookAndFeel() utilizes UIManager.getLookAndFeel() to get information on the L&F that's going to be used for the application.

```
private static void showCurrentLookAndFeel()
{
  LookAndFeel info = UIManager.getLookAndFeel();
  if (info == null)
    System.out.println("\nNo L&F set");
  else
    System.out.println("\nCurrent L&F is " + info.getName());
}  // end of showCurrentLookAndFeel()
```

Third-Party Look and Feels

It's not at all difficult to utilize a third-party L&F (i.e. one that isn't part of the JRE). A growing list can be found at http://javootoo.com/ (also accessible at http://javootoo.l2fprod.com/). We'll use the skinable Substance L&F and the cool looking Napkin L&F in this hack.

The Substance L&F

Substance is a highly customizable L&F, developed by Kirill Grouchnikov, and freely available from https://substance.dev.java.net/. It offers a wide range of different L&Fs, grouped into skins:

- Toned down skins: https://substance.dev.java.net/docs/skins/toneddown.html
- Dark skins: https://substance.dev.java.net/docs/skins/dark.html
- Saturated skins: https://substance.dev.java.net/docs/skins/saturated.html

The saturated skins are the most fun in our opinion, including "FindingNemoSkin" and "GreenMagicSkin".

For each skin, it's possible to adjust the colour scheme (theme), watermarks, button shapes, borders, the title bar, and many other attributes.

We downloaded the Substance runtime library, substance.jar, version 4.2 (1.8 MB), and saved it in a new directory (d:\lafs\substance\) on a test machine.

The simplest way of trying out different skins is to start the application with the basic Substance L&F called "org.jvnet.substance.SubstanceLookAndFeel". This can be done from the command line by setting the Java "swing.defaultlaf" property:

```
>  java -cp d:\lafs\substance\substance.jar;.
       -Dswing.defaultlaf=org.jvnet.substance.SubstanceLookAndFeel
           LAFWin
```

There's no need to recompile LAFWin.java, but substance.jar must be added to the classpath (which we've done with the -cp option).

 © Gayathri Singh and Andrew Davison 2008

Note that LAFWin is called without a numerical argument, making it use the default L&F, which has just been set to SubstanceLookAndFeel with "swing.defaultlaf". The new LAFWin look is shown in Figure 10-29.



Figure 10-29. LAFWin Using the Substance L&F.

If the application has set the title bar to be drawn by the JVM (using JFrame.setDefaultLookAndFeelDecorated()), then the Java icon on the left of the title bar can be left-clicked to display a popup menu for configuring Substance (see Figure 10-30).

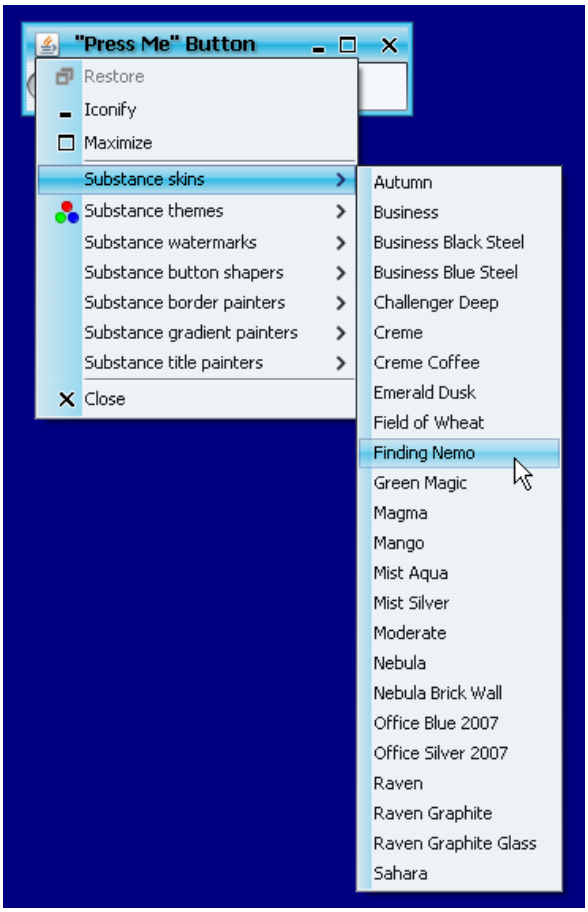


Figure 10-30. The Substance Configuration Popup Menu.

Selecting "FindingNemo" changes the application to look like Figure 10-31.



Figure 10-31. LAFWin Using the "FindingNemo" Substance L&F.

 © Gayathri Singh and Andrew Davison 2008

Once you've chosen a L&F, it can be specified on the java command line at execution time:

```
> java -cp d:\lafs\substance\substance.jar;.
       -Dswing.defaultlaf=org.jvnet.substance.skin.
                                SubstanceOfficeSilver2007LookAndFeel
       -Dsubstancelaf.noExtraElements
            LAFWin
```

This makes LAFWin use the altogether more sensible Office Silver 2007 L&F, as shown in Figure 10-32.

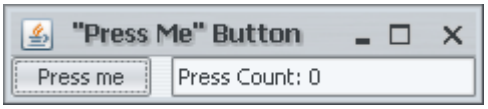


Figure 10-32. LAFWin Using the "OfficeSilver2007" Substance L&F.

The "substancelaf.noExtraElements" property on the command line switches off Substance's popup menu, which probably isn't required once a L&F has been chosen.

The Napkin L&F

Napkin is a unique L&F that make the GUI look like it was quickly scrawled on a napkin. It was developed by Ken Arnold and Alex Lam, and is freely available from http://napkinlaf.sourceforge.net/.

We downloaded napkinlaf-jars-1.0_001.zip (3.05 MB), unzipped it, and saved napkinlaf.jar (only 695 KB) in a new directory (d:\lafs\napkin\) on a test machine.

Most of the download is taken up with a Napkin-enabled version of the SwingSet2 JAR, a standard Sun demo which illustrates many Swing features. Double click on napkinlaf-swingset2.jar to start it. A typical tab looks like Figure 10-33.

 © Gayathri Singh and Andrew Davison 2008

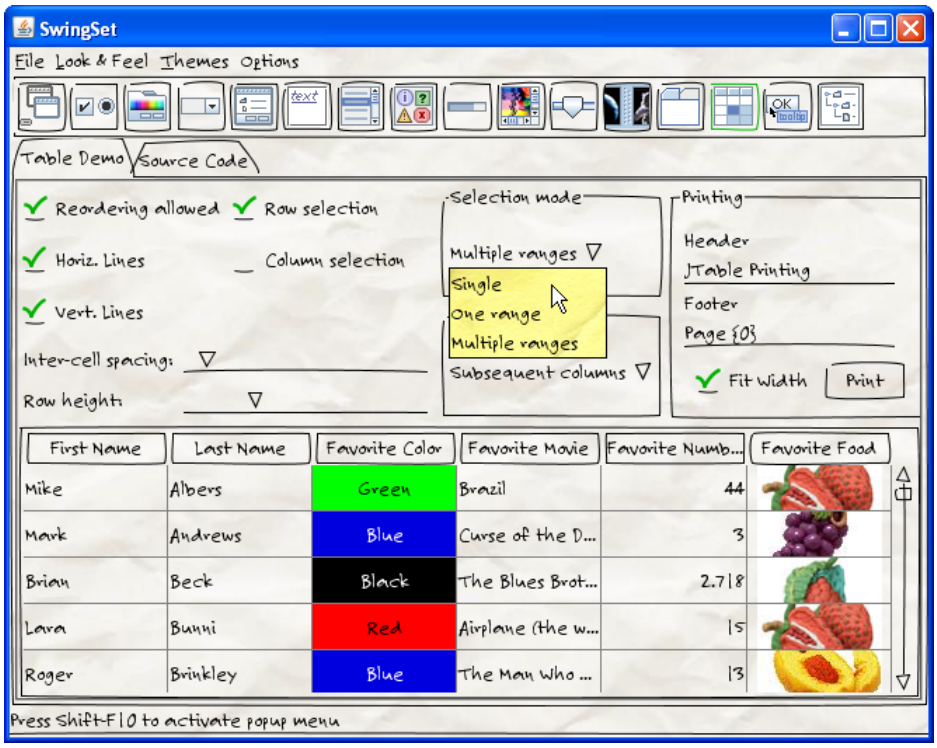


Figure 10-33 SwingSet2 Using the Napkin L&F.

The Napkin L&F can be applied to LAFWin using a similar Java command line as employed with Substance:

```
> java -cp d:\lafs\napkin\napkinlaf.jar;.
       -Dswing.defaultlaf=net.sourceforge.napkinlaf.NapkinLookAndFeel
           LAFWin
```

The result is shown in Figure 10-34.

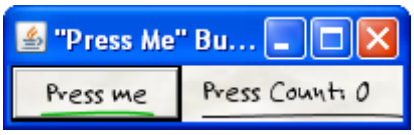


Figure 10-34. LAFWin Using the Napkin L&F.

Note that Napkin doesn't affect the title bar, which detracts from the effect a little.

 © Gayathri Singh and Andrew Davison 2008