Active Content in Java

by Prithvi Rao

<prithvi+@ux4.sp.cs.cmu.edu>

Prithvi Rao is the co-founder of KiwiLabs, which specializes in software engineering methodology and Java/CORBA training. He has also worked on the development of the MACH OS and a real-time version of MACH. He is an adjunct faculty at Carnegie Mellon and teaches in the Heinz School of Public Policy and Management.

Drawing and animation are integral parts of writing Java applications. Applets frequently contain various AWT (Abstract Windowing Toolkit) Components that are part of a graphics context. When they need to be redrawn, the AWT starts with the top Component in the hierarchy and works its way down to the bottom.

The purpose of this article is to expose the reader to various capabilities that are part of the "Active Content" in Java. I'll focus on issues related to the drawing model, drawing shapes, the graphics context, drawing text, measuring text images, and loading them both synchronously and asynchronously. I'll also touch briefly on eliminating flashing and animating images. This information provides for a richer knowledge base from which to venture into writing more interesting Java programs with active content.

The Drawing Model

Redrawing is performed when the AWT starts with the topmost `Component`in the hierarchy and works its way down to the bottom `Component`. Given that containers are Components as well, this applies equally to applets and panels.

Each `Component` draws itself before it draws any of the C`Component` that it contains. This ensures that a Panel's background is visible only where it isn't covered by one of the Components embedded in it.

Programs draw only when the AWT tells them to do so. For instance, this can happen when a `Component` becomes uncovered by the user. Another example is when there is a change in the data being reflected by the`Component`.

The AWT requests that a `Component` draw itself by invoking the`Component`'s `update()` method. The default `Component` implementation of`update()` clears its clipping area in the current background color and then calls the `paint()` method. The default implementation of `paint()` does nothing.

Another way to look at this is that a programmer causes a refresh that's due to some change in state of the control by calling `repaint()`, which in turn calls `update()`. It is usually the case that the `paint()` method is overridden to provide component-specific drawing. It is also possible to override`update()` for more sophisticated results. However, it is possible to call`paint()` directly without calling `update()`, so `paint()` must always be implemented.

Note that `update()` sets the clipping region, and so if the object being drawn changes between drawings, it is possible to end up with half of one and half of the other.

As can be anticipated, the `paint()` and `update()` methods must execute very quickly to ensure acceptable performance. (Note: This is where it becomes advantageous to use threads with applets to achieve better performance. My December 1999 "Using Java" article, "Using Threads Within Applets," discusses this topic.)

The Graphics Context

The argument to the `paint()` and `update()` methods is a Graphics object that represents the context in which the Component can perform its drawing. The Graphics class provides for drawing and filling rectangles, arcs, lines, ovals and polygons, text and images.

It is also possible to "set" and "get" the current color, font, or clipping area, and to set the paint mode.

The `Color` class encapsulates an "r.g.b" triplet, and color is set by:

```
Graphics.setColor(Color color);
```

How text is drawn is defined by the Font class that encapsulates a font (e.g., Times Roman or Helvetica). It also supports point size and a style, although currently only bold, italic, and plain are supported. An example of setting the font is:

```
Graphics.setFont(Font font);
```

Drawing Shapes and Text

The Graphics class has a multitude of methods that permit the drawing of various shapes, including:

`Graphics.drawRect(int x, int y, int width, int height);`
for drawing rectangles
`Graphics.fillRect(int x, int y, int width, int height);`
for filling a rectangle
`Graphics.drawRoundRect(int x, int y, int width, int height, int arcwidth, int archeight);`
`Graphics.fillRoundRect(int x, int y, int width, int height, int arcwidth, int archeight);`

When drawing text it is better to first consider whether it is possible to use a text-oriented Component such as the `Label`, `TextField`, or `TextArea` class. The alternative is to use `drawBytes()`, `drawChars()`, or `drawString()`. For example you can "draw" a string as follows:

```
Graphics.drawString("This is an example of drawing a string", x, y);
```

where x and y specify the coordinates for "drawing" the text.

Measuring Text

In order to determine if text can fit inside a certain area, it is necessary to query the characteristics of the Font using `getFontMetrics()`, which returns a `FontMetrics` object corresponding to a given Font:

```
FontMetrics foo = Graphics.getFontMetrics(Font f);
```

Some of the most commonly used methods that are part of the `FontMetrics`class are `charWidth()`, `getHeight()`, `getAscent()`, `getDescent()`, and`stringWidth()`.

Images and Loading Images Asynchronously

All images are encapsulated by the image class. There are two ways to load images, `Applet.getImage(URL)` and `Toolkit.getImage(filename/URL)`. Both of these load an image, and return an Image object. The method`getImage()` returns immediately without checking whether the image data exists or whether it's been successfully loaded. This is done to improve performance, since it is not necessary to wait for an image to be loaded before going on to perform other functions in the application. Some examples of loading images are:

```
Image image = Applet.getImage(getDocumentBase(), "Image.jpeg");
Image image = Toolkit.getDefaultToolkit().getImage( new/URL
   ("http://www.sample.com/images/sample.gif"));
```

In order to draw the image it is necessary to use one of the Graphics.drawImage() variants. In other words, `drawImage()` allows you to specify the image to draw and the positioning and scaling of the image. It also allows the specification of the color to draw underneath the image. This is useful if the image contains transparent pixels.

The `ImageObserver` parameter specifies an object that implements the`ImageObserver` interface. This object will be notified whenever new information regarding the image becomes available. The `Component` class implements the `ImageObserver` interface to invoke the `repaint()` method as the image data is loaded. The method `drawImage()` returns immediately even if the image data has not been completely loaded; this results in the image being displayed partially or incrementally. The easiest way to track the loading of images and to make sure that `drawImage()` draws only complete images is to use the `MediaTracker` class. The sequence is:

1. Create a MediaTracker instance.
2. Tell it to track one or more images.
3. Ask the MediaTracker the status of images.

The following code examples elucidates this point:

```
tracker = new MediaTracker(this);
images = new Image(num_images);
for (int i=0; i
 images[i] = getImage(this.getDocumentBase()."image" + i);
tracker.addImage(images[i],i);
}
```

and then later in the program

```
for (int i=0; i
  this.showStatus("Loading image: "+i);
 tracker.waitForID(i);
 if(tracker.isErrorID(i)) {
   showStatus("Error loading image"+i);
   return;
 }
}
showStatus("Loading images done.");
```

Animation and Thread Management

Animation is perceived motion accomplished by sequencing rapidly through frames. Frames can be incrementally different images or graphics operations. A good rule of thumb is that animation should run on a separate thread in order not to adversely affect event handling. The typical sequence is:

1. Implement the `run()` method to increment a frame.
2. Perform a `repaint()` operation.
3. Perform a `sleep()` operation to delay the frame.

Incorporating threading with animation permits the suspension and resumption of animation with button or mouse events.

To suspend or resume animation in an applet when the user leaves the page, it is necessary to reimplement the applet's `stop()` and `start()` methods. The following code example shows how to handle a button event:

```
public boolean handleEvent(Event e) {
 if (e.id == Event.ACTION_EVENT) {
   if (e.target == start)
    start();
  else
```

```
    if (target == stop)
      stop();
     }
     return super.handleEvent(e);
  }
  private Thread thread = null;
  public void start() {
   if (thread == null) {
     thread = new Thread(this);
    start.disable();
    stop.enable();
    }
  }
  public void stop() {
    if ((thread != null) && thread.isAlive())
      thread.stop();
    thread = null;
    start.enable();
    stop.disable();
  }
```

A common problem with animation is "flashing," which manifests itself as animation with jitter. One solution to this is to use "double buffering." This involves performing multiple graphics operations on an undisplayed graphics buffer and then displaying the completed image on the screen. Besides preventing incomplete images from being drawn to the screen, double buffering improves drawing performance. This is because drawing to an offscreen image is more efficient than drawing to the screen.

Animating Images

It is likely that loading images will take a long time regardless of whether`MediaTracker` is used or not. In other words, whenever you load an image using a URL, it will be time-consuming. Most of the time is taken up by initiating HTTP connections. Each image file requires a separate HTTP connection, and each connection can take several seconds to initiate.

One way to avoid this performance degradation is to include multiple images in a single file. Performance can be further improved by using some sort of compression scheme, especially one that is designed for moving images. A simple way to do this is to create an image strip, which is a file that contains several images in a row. To draw an image from the strip it is necessary to first set the size of one image, then perform a draw operation on the image strip shifted to the left so that only the image desired appears within the clipping area.

Conclusion

Writing Java programs that contain active content can be very frustrating but also very rewarding. While it may be simple to rapidly prototype an applet with animation and images, fine-tuning it to meet stringent performance requirements requires more than a cursory knowledge of Java.

The JDK provides a versatile and powerful collection of packages to facilitate writing Java programs with active content. The common theme is always to work with the current graphics context and to use classes such as the `FontMetrics` class to determine sizing of text regions.