VidWorks Entertainment New Developer Training Program
Tutorial #5

Images & Animation

Images

There is a class in the javax.imageio package called ImageIO. The ImageIO class is a powerful, complicated class that can do a lot with images. However, for our purposes, we only need one method:

```
static BufferedImage read(File input) throws IOException
```

BufferedImage is a subclass of the abstract class Image, and provides most of the functionality. However, since we usually don't need that extra functionality, we'll just work with the objects of static type Image. (It saves us all that time we'd be spending writing "Buffered", anyway.) However, this is more a matter of style and less a matter of programming technique, so it's up to you.

And for those of you who are not aware, the File class represents a file, and it has several constructors. The most useful and simplest one is:

```
File(String pathname)
```

Thus, the following snippet of code can easily load images for us:

```
Image img;
try {
      img = ImageIO.read(new File("[filename]"));
} catch (IOException e) {
      // handle the error
}
```

The Graphics object we should be familiar with in painting has the following *six* methods for drawing images:

`boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)` - draws the image with the given coordinates, filling in the background with the given color for images with transparencies

`boolean drawImage(Image img, int x, int y, ImageObserver observer)` - draws the image with the given coordinates, but objects underneath still show

`boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)` - draws the image with the given coordinates, given dimensions (stretching or shrinking if necessary), and with the given background color

`boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)` - draws the image with the given coordinates, given dimensions, and allows objects underneath to show

`boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)` - draws a portion of the image with corners at (sx1, sy1) and (sx2, sy2) and draws it in a rectangle with corners at (dx1, dy1) and (dx2, dy2), with the given background color

`boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)` - draws a portion of the image with corners at (sx1, sy1) and (sx2, sy2) and draws it in a rectangle with corners at (dx1, dy1) and (dx2, dy2), but allows objects underneath to show

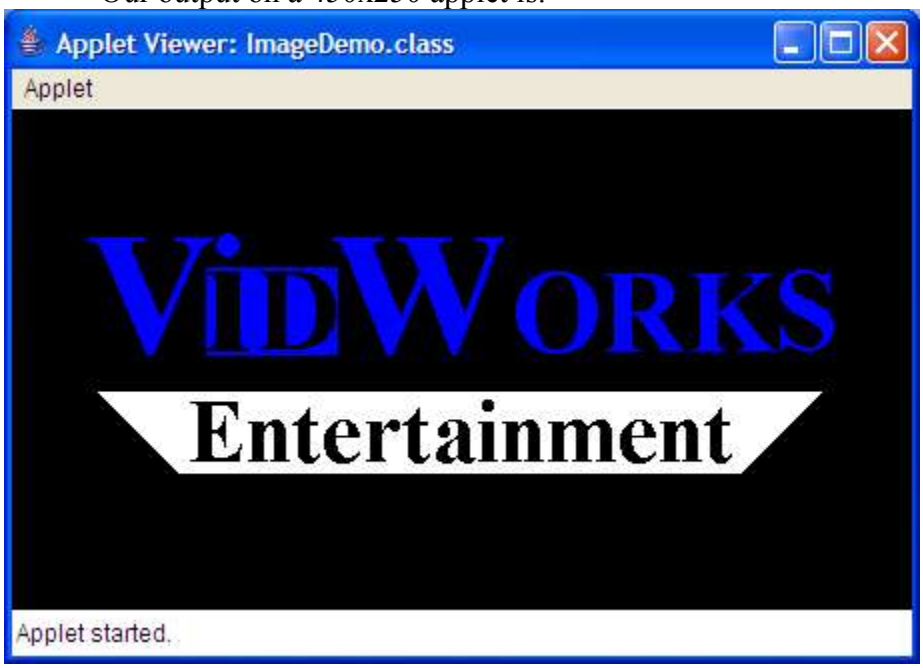
All these methods return true if the image is done loading and false otherwise. Note that this is only applicable if you use older methods that would load images asynchronously (well, they still are loaded asynchronously, technically speaking). However, since `ImageIO.read()` doesn't return until the image has been fully loaded, for our intents and purposes, all the `drawImage()` methods return true.

In addition, ImageObserver is an interface that allows us to receive updates about an Image loading. Again, this is obsolete for our purposes, so we can usually pass in null values for the ImageObserver arguments.

And now, we have enough information for an example. Don't forget to download the images!

```
/* ImageDemo.java */

import java.awt.*;
import java.applet.*;
import javax.imageio.*;
import java.io.*;

public class ImageDemo extends Applet {
	Image img;

	public void init() {
		try {
			img = ImageIO.read(new File("vidlogo.gif"));
		} catch (IOException e) {
			System.out.println("Error loading image!");
		}

		setBackground(Color.black);
	}

	public void paint(Graphics g) {
		g.drawImage(img, 25, 25, null);
	}
}
```

Our output on a 450x250 applet is:



Yes, I am aware that this by itself would be a very stupid applet.

Animation

The basic concept behind animation is that we have one thread controlling the whole process, periodically calling the `repaint()` method on an applet. Then, at each call to `repaint()`, we change what we display slightly so that it appears to be animated.

It's best to show you what I mean by jumping directly into a self-explanatory example:

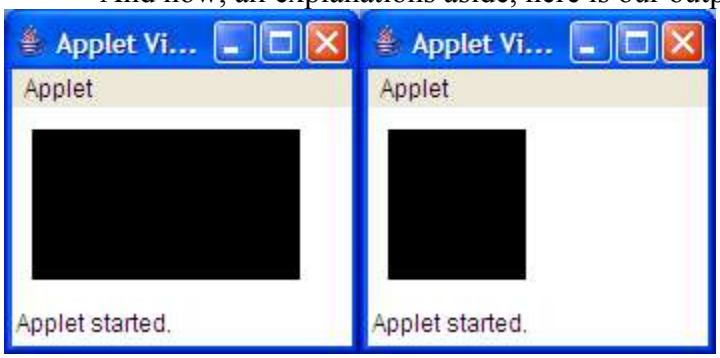
```
/* AnimationDemo.java */

import java.applet.*;
import java.awt.*;

public class AnimationDemo extends Applet implements Runnable {
	static final int FRAMETIME = 40;
	static final double D_THETA = 0.05;

	boolean running = true;
	double theta = 0;

	public void init() {
		new Thread(this).start();
	}

	public void paint(Graphics g) {
		theta += D_THETA;
```

```
            int width = (int) Math.ceil(100 + 50 * Math.sin(theta));
            g.fillRect(10, 10, width, 75);
      }

      public void run() {
            while (running) {
                  try {
                        Thread.sleep(FRAMETIME);
                  } catch (InterruptedException e) { }
                  repaint();
            }
      }

      public void stop() {
            running = false;
      }
}
```

A few notes: First, all applets have a `repaint()` method, which sets up an applet to repaint itself by first getting the Graphics object, then calling `update()`, which calls `paint()`. We call this method to repaint our applet instead of `paint()`, mainly because we don't have a Graphics object to pass into `paint()`.

The second note is the method `stop()`. All applets have a `stop()` method which is called when the user leaves the website containing the applet, closes the browser, or otherwise forces an applet to stop running. In our case, we use the `stop()` method to set a flag which tells our animation thread to finish.

And now, all explanations aside, here is our output at 170x95:



Double Buffering

You've probably noticed by now that there's a nasty flicker in our little animation. What? You can't see it? Then maybe your computer is fast enough to avoid the flicker. (After all, our applet runs at a pathetic 25 frames a second.) Decrease the frame time (which is the same as increasing the frame rate), and you'll notice the flickering gets worse and worse. (For a more complicated drawing, like an animated Sword of Viltris, the flicking would be even more horrible.)

The cause of this flicker is twofold: First of all, the `repaint()` method always calls `update()`, and `update()` clears the applet by paints over the entire background with a solid white (or whatever the background color is). Since painting is done in real-time, there is a very small moment of time between clearing the screen and drawing the new rectangle on it. When the frame rate is high enough, the white background appears often enough that it becomes visible. This is what causes the flickering in our applet.

The second reason applies mainly toward applets whose animations consists of more then one draw command, for example, if an applet drew two images in sequence. Again, since drawing is done in real-time, there will be a small amount of time during each frame when the first image is visible, but the second image hasn't appeared yet, which is another source of flickering.

How do we solve our flickering problems? Basically, we need to make sure *everything* gets drawn simultaneously, so that there will be no portion of time where we see the flicker. This is done using a technique called double-buffering, also known as block line transfer or BLTing (pronounced "blitting", not to be confused with the sandwich). We draw everything into a *back buffer* that's not visible to the screen. In Java, this is usually an Image object. Then, using `drawImage()`, we draw the entire thing *in one step*, onto our applet.

Note that we can create a blank Image to use as a back buffer by calling:

```
Image createImage(int width, int height)
```

This method is a member of the Component class, which is a super-class to Applets and Frames as well as all UI Components we have discussed.

To draw on our back buffer, we need its Graphics object. Thankfully, every Image has the following method:

```
Graphics getGraphics()
```

Unfortunately, this doesn't fix our problem of `update()` clearing the screen for us, as there still is a bit of time between `update()` clearing the screen and us drawing our back buffer to the screen. We can solve this by overriding `update()` so that all it does is call `paint()`. The following code snippet should do the trick:

```
public void update(Graphics g) {
      paint(g);
}
```

Now, let's de-flicker our animated applet:

```
/* Animation2.java */

import java.applet.*;
import java.awt.*;

public class Animation2 extends Applet implements Runnable {
      static final int FRAMETIME = 10;
      static final double D_THETA = 0.05;

      boolean running = true;
      double theta = 0;
      Image buffer;

      public void init() {
            buffer = createImage(170, 95);

            new Thread(this).start();
      }

      public void update(Graphics g) {
            paint(g);
      }

      public void paint(Graphics g) {
            theta += D_THETA;
            int width = (int) Math.ceil(100 + 50 * Math.sin(theta));

            Graphics bgfx = buffer.getGraphics();
            bgfx.setColor(getBackground());
            bgfx.fillRect(0, 0, 170, 95);
            bgfx.setColor(getForeground());
            bgfx.fillRect(10, 10, width, 75);

            g.drawImage(buffer, 0, 0, null);
      }

      public void run() {
            while (running) {
                  try {
                        Thread.sleep(FRAMETIME);
                  } catch (InterruptedException e) { }
                  repaint();
            }
      }

      public void stop() {
            running = false;
      }
}
```

(Note, no screenshot is provided, since it would be identical to the ones before.)

Much better, isn't it? Granted, the flicker isn't *totally* eliminated (there's a bit at the moving edge if you look carefully), but even at 100 frames per second, it looks better than our original 25 frames per second version.

Conclusion

Now, you've learned how to read from images and how to make animations on screen. This is more than enough to make your own games. (Well, you could have made your own games as early as Tutorial #2, but now you can animate applets, you can make your own *action* games.)

However, there's still a lot more we can teach you, if you're willing to learn. Next time: We will give you a hodgepodge of our developers' favorite graphics & animation techniques (well, favorite at the time of this writing)!

Homework Assignment!

You provide the unifying example! Make an applet (or an application if you want) that has an animated image! (No, animated GIFs don't count.) Be sure to use double-buffering!