Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us,
http://www2.austin.cc.tx.us/baldwin/

The AWT Package, Graphics - Animation and Double Buffering

Java Programming, Lecture Notes # 172, Revised 02/19/98.

Preface

Students in Prof. Baldwin's **Advanced Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

The material in this lesson is extremely important. However, there is simply too much material to be covered in detail during lecture periods. Therefore, students in Prof. Baldwin's **Advanced Java Programming** classes at ACC will be responsible for studying this material on their own, and bringing any questions regarding the material to class for discussion.

This lesson was originally written on November 30, 1997 using the software and documentation in the JDK 1.1.3 download package.

Introduction

A previous lesson introduced you to many aspects of working with images in Java. In this lesson, we will extend what you know into a classical graphics problem - *animation*. In order to improve the illusion of of motion in our animation, we will also introduce you to the use of *double buffering* in Java.

Animation normally consists of the rendering of many images in succession with each image differing incrementally from the previous image. If the images are rendered fast enough, and the incremental changes are small enough, we are fooled into believing we are seeing true motion.

If not done properly, we see flashing and flicker that destroys the illusion.

This lesson will develop an animation program that can be viewed either with or without double buffering so that you can easily see the difference that double buffering makes to the animation process.

We will introduce and discuss a number of new concepts as we view interesting code fragments from the program. A complete listing of the program is provided at the end of the lesson.

Sample Program

This program illustrates animation with or without double buffering. It requires access to an image file named **"logomain.gif"** in the current directory on the hard disk.

Just about any image file should do, but it needs to be large enough that you can see some of the details when it is displayed at its normal size. An image file that covers about one-eighth to one-fourth of your screen should be about right.

Java JDK 1.1.3 only supports image files in the **gif** or **jpg** formats. You should be able to find hundreds of **gif** or **jpg** files in the cache directory of your browser. You will need to rename the file to the name given above and copy it into the current directory on your hard disk.

For simplicity, the name of the graphics file is hard-coded into the program. You could modify the source code to accommodate a different file name, or you could provide additional code in **main** which requests a file name from the standard input device and uses that name for the image file.

If the program is unable to load the image file within ten seconds, it will abort with an error message. If your file is large, or is being loaded from a web server, it may be necessary for you to modify the source code and increase the timeout interval.

To run the program in double-buffer mode, enter the following at the command line:

```
java image04
```

To run the program with double buffering disabled, enter the following at the command line (note the command-line parameter **x**):

```
java image04 x
```

Actually, entering any command-line argument disables double-buffer mode and causes the program to draw directly to the screen.

The animation is performed in a **Frame** object as the container. The size of the **Frame** is automatically adjusted to accommodate the size of the image.

Once started, the program will continue to animate the image until you press the close button on the **Frame** object.

This program causes a series of images of successively increasing or decreasing size to be scaled and rendered on the screen.

The series of images are scaled in such a way as to make it appear that the base image is moving into and out of the screen like a yo-yo. The image is pinned at its upper left corner, so all size changes are relative to that point.

This causes the image to appear to enter the **Frame** in the upper left corner and move toward you with a slight left-to-right angle. When the size of the animated image is almost as large as the **Frame** object, it reverses direction and retreats back into the upper left-hand corner. Then it starts the process all over again.

In order to simulate a more realistic animation task, scaling is performed as the image is being drawn either onto the screen or onto the graphics context of the buffer area. (In other words, prescaled images are not used.) The scaling is performed on-the-fly in the **paint()** method.

The **paint()** method is invoked approximately 20 times per second which is sufficiently fast to provide a good illusion of animation. Once the image becomes large, however, it appears that the repetition rate slows down noticeably due to the time required to render the larger images (on a 133mhz Pentium processor).

Actually, the **paint()** method is invoked again approximately 50 milliseconds after it exits. Thus, the time interval between repetitions is 50 milliseconds plus the time required to render the image in the **paint()** method. A better approach would be to develop a clock signal that is independent of the time required to render the images and use that signal to invoke the **paint()** method each time. This would eliminate (or reduce) the apparent slowdown for large images until the point where the actual rendering consumes the full interval between invocations. However, repetition control is a subject for a different lesson. This lesson is primarily concerned with the graphics aspects of animation.

The screen image is modified once each time that the **paint()** method is invoked.

When in double-buffer mode, the contents of the buffer are transferred to the screen upon entry into the **paint()** method. Then the next version of the image is drawn into the buffer area (to be transferred to the screen at the beginning of the next repetition) and the **paint()** method exits.

When not in double-buffer mode, the image is drawn directly to the screen upon entry into the **paint()** method and then the **paint()** method exits.

In Jave, we actually invoke the **repaint()** method to cause the screen to be painted. Perhaps you have wondered what happens when you invoke the **repaint()** method. One thing that happens is that the **update()** method is invoked which in turn invokes the **paint()** method. The **update()**

method normally erases the entire screen each time **repaint()** is invoked. After erasing the screen, it invokes the **paint()** method.

Normally, we don't override the **update()** method except in special circumstances. This is one of those special circumstances. In this program, the **update()** method is overridden to eliminate unnecessary erasing of the screen because unnecessary erasing of the screen causes an undesirable flashing in an animation program.

Because the **update()** method no longer erases the screen, each time **paint()** is entered (when not in double-buffer mode) it is necessary to erase a small area of the screen before drawing a new image in order to eliminate any residue from the previous image.

It is not necessary to erase the screen in double-buffer mode because an entire screen image is copied at relatively high speed from the buffer area to the screen each time the **paint()** method is invoked. However, it is necessary to erase a small portion of the buffer area each time before drawing a new image in it so as to eliminate any residue from the previous image drawn into the buffer area.

When this program is run in non-buffer mode, there is a lot of flicker and the animation illusion isn't very good. This flicker is apparently the result of drawing each successive image in full view of the user.

In other words, even though the drawing is accomplished very quickly, the viewer can still see each image being drawn on the screen after its area of the screen has been erased and this results in a flicker effect. (Weren't movies called flickers or flicks in the early days?)

The flicker in this program is eliminated in double-buffer mode and the illusion of motion is pretty good. In the double-buffer case, the actual drawing of the individual images takes place out of sight of the viewer, and once drawn, the entire image is blasted to the screen very quickly.

Be aware, however, that even in double-buffer mode, the scaling of the images may be less than perfect and you may see fuzz growing on your image as it changes in size from small to large and back to small again. This is because the pixel representation of the image is not particularly good at any size, and the pixel representation at one size is different from the pixel representation at a slightly different size. This is especially noticible in areas with a high degree of detail such as in areas containing text.

This program was tested using JDK 1.1.3 under Win95.

Interesting Code Fragments

The first interesting code fragment contains the set of instance variables used to implement the behavior of the program. Of particular interest is the reference variable of type **Image** named **rawImage** that points to the raw image loaded from the disk file.

Also of interest are the **Image** variable named **offScreenImage** and the **Graphics** variable named **offScreenContext**. The first is a reference to an object of type **Image** that is used to contain the offscreen version of the image during the buffering process. The second is the graphics context for that object. (Recall that you cannot draw on an **Image** object. Rather, you must obtain the graphics context for the **Image** object and draw on the graphics context.)

```
Image rawImage;//ref to raw image file fetched from disk
int rawWidth;
int rawHeight;

//Width and height values for a particular
// animation frame
int newWidth;
int newHeight;

//Inset values for the container object
int inTop;
int inLeft;

//References to objects used for double buffering
Image offScreenImage;
Graphics offScreenContext;

//To double buffer or not double buffer
boolean doubleBuf = true;
```

We will see numerous references to these instance variables throughout this discussion.

The next interesting code fragment is the code in **main** that checks to see if the user entered any command-line arguments, and if so sets a flag that causes the program to operated in a non-buffered mode (buffered mode is the default).

Code in the **paint()** method checks the state of the **boolean** instance variable named **doubleBuf** to determine whether to implement double buffering or not.

```
  if(args.length == 0){
    obj.doubleBuf = true;
    System.out.println("Double buffer mode");
  }//end if
  else{
    obj.doubleBuf = false;
    System.out.println("Not double buffer mode");
  }//end else
```

This brings us to the actual animation loop that is implemented inside **main**. The code in **main** enters an endless loop which includes a 50-millisecond sleep interval. Once each iteration, it performs some calculations to establish the size for the next rendering of the image and then invokes the **repaint()** method.

According to Just Java 1.1 and Beyond by Peter van der Linden, (page 446) this is what happens when you invoke **repaint()**.

```
repaint() calls update() which calls clear() and then calls paint()
```

In fact, van der Linden tells us (page 469) that the **update()** method looks like this:

```
public void update(Graphics g){
  g.setColor(getBackground());
  g.fillRect(0,0,width,height);
  g.setColor(getForeground());
  paint(g);
}
```

The effect of **update()** (as far as we are concerned here) is to redraw the image in the solid background color (on the graphics context) and then invoke the **paint()** method to render that graphics context to the screen.. Later in this program, we will override the **update()** method to eliminate the first three statements so that it will simply cause **repaint()** to invoke **paint()**.

So why don't we just invoke **paint()** in the first place if we don't want **update()** to erase the screen (fill it with the background color)? Because **paint()** requires a parameter that we don't normally have access to: the **Graphics** context of the object for which **paint()** is being overridden. In fact, one of the three ways that we gain access to the **Graphics** context is by overriding either **paint()** or **update()**.

Now let's get back to the fact that the code in **main** enters an endless loop which includes a 50-millisecond sleep interval. Once each iteration, it performs some calculations to establish the size for the next rendering of the image and then invokes the **repaint()** method.

The calculations (and the associated tests) are pretty straightforward. The code simply decides whether the image should be growing or shrinking (or has hit a limit and should change direction) and then uses a scale factor to calculate the new size of the image to be displayed. The horizontal and vertical sizes are assigned to two of the instance variables that we saw at the beginning of the class definition and the scale factor is changed for use in the next iteration..

After invoking **repaint()**, the code goes to sleep for 50 milliseconds, wakes up and executes another iteration. When the user clicks the *close* button on the **Frame** object, a Window event will be processed to terminate the program.

```
    //Loop in animated mode until close button is clicked
    while(true){
      //Reverse direction of motion if necessary
      if(scale >= 0.999) delta = -0.005;//move into screen
      if(scale <= 0.015) delta = 0.005;//move out of screen
```

```
      //Establish width and height for this rendering
      // of the image.
      obj.newWidth = (int)(scale*obj.rawWidth);
      obj.newHeight = (int)(scale*obj.rawHeight);
      obj.repaint();//render the image

      scale += delta;//update scale for next rendering

      //Sleep for awhile. Animate at approximately 20
      // frames per second.
      try{
        Thread.currentThread().sleep(50);
      }catch(InterruptedException e){System.out.println(e);}
    }//end while loop
  }//end main
```

Now we come to the constructor. The first interesting code fragment in the constructor is the statement that "gets" the image contained in the specified file. We have seen statements like this in earlier lessons where the **getImage()** method is invoked on an object of the **Toolkit** class to read a disk file and assign the contents of the file to a reference variable of type **Image**.

```
    rawImage =
      Toolkit.getDefaultToolkit().getImage("logomain.gif");
```

We don't really want to try creating scaled versions of the image until is is fully loaded, so we instantiate and use a **MediaTracker** object to track the loading status of the image file as shown in the next code fragment..

Once we instantiate the object, we invoke the **addImage()** method to add this image to the list of images to be tracked by the object. In the process, we establish the identification value "1" for this image. This identification can be used to request a variety of different types of tracking information about the image from the tracker object.

The tracker object can also be asked to block until the identified image is fully loaded, or until the passage of a specified amount of time.

In this case, we use the **waitForId()** method to ask the tracker object to block until the image is loaded, or ten seconds elapses, whichever comes first. The method returns **true** if the image is successfully loaded during the ten-second interval, and **false** otherwise. If **false**, we display an error message and terminate the program.

```
    //Use a MediaTracker object to block until the image
    // is loaded or ten seconds has elapsed.
    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(rawImage,1);
```

```
    try{
      if(!tracker.waitForID(1,10000)){
        System.out.println("Load error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){System.out.println(e);}
```

You should also note that the **waitForID()** method throws a **InterruptedException** object which is a *checked* exception. Therefore, we must either declare this fact in our method signature, or include the call to **waitForID()** in a *try/catch* block.

We attempted to make this program compatible with a wide variety of image files. As a result, we will see later that the size of the **Frame** object that serves as the container is automatically adjusted to the size of the raw image.

The next interesting code fragment extracts the width and height of the raw image and assigns them to two of the instance variables that we saw at the beginning of the class definition. This makes the information available later for the process of adjusting the size of the **Frame** object to make it match the size of the raw image.

```
    rawWidth = rawImage.getWidth(this);
    rawHeight = rawImage.getHeight(this);
```

Inset information for the **Frame** object is also needed in the automatic sizing of that object to fit the image. The next step is to make the **Frame** object visible and then get the inset information which we assign to two of the instance variables. Note that the insets information is not available until you associate the **Frame** with a peer object. **setVisible()** is one way to do that.

```
    this.setVisible(true);//make the Frame visible

    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;
```

The next code fragment makes use of the information about the image and container gathered thus far and uses that information to establish the size of the **Frame** object to match the size of the raw image.

```
    this.setSize(inLeft+rawWidth,inTop+rawHeight);
```

The next code fragment is completely new to this lesson. You haven't seen it in any of my previous lessons (unless I update some of them after this writing).

In this code fragment, we use the **createImage()** method to get an **Image** object of the specified size that we can use for our offscreen buffer area. Recall, however that you cannot draw directly on an **Image** object. Rather, you must get a graphics context for the object and actually draw on the graphics context.

We use the **getGraphics()** method to get that graphics context. This is one of the three ways that you can gain access to a graphics context of type **Graphics**. The other two ways are to overload the **update()** and **paint()** methods which receive a graphics context as a parameter.

```
    offScreenImage = this.createImage(rawWidth,rawHeight);
    offScreenContext = offScreenImage.getGraphics();
```

That concludes the discussion of code fragments in the constructor. The next interesting code fragment is the overridden **update()** method. As you may recall, we overrode this method to prevent it from erasing the screen before invoking the **paint()** method as described earlier. As you can see, we have eliminated all but one of the statements in the **update()** method that you saw earlier.

```
  public void update(Graphics g){
    paint(g);
  }//end overridden update() method
```

Next we turn our attention to the overridden **paint()** method where we do all of our drawing. We will discuss this method in two parts.

In the first part we will highlight the code that is executed when the user selects the default double-buffer mode.

In the second part we will highlight the code that is executed when the user selects the non-buffered mode.

Recall that this method receives the graphics context for the **Frame** object (the class for which it is being overridden). This graphics context is known locally as **g**. The method also has access to the graphics context for the offscreen image which is named **offScreenContext**.

Each time this method is invoked in the double-buffer mode, it draws a new image and leaves it in the offscreen graphics context.

We begin the method by accessing the image from the offscreen graphics context and rendering it to the screen. We use the **drawImage()** method of the **Frame's** graphics context to do this, passing the **offScreenImage** as a parameter. This causes the **offScreenImage** to be drawn on the screen at the location specified by the second and third parameters. Note that this is a non-scaling version of the **drawImage()** method which executes very rapidly.

```
      g.drawImage(offScreenImage,inLeft,inTop,this);

      offScreenContext.clearRect(
                                 0,0,rawWidth,rawHeight);
      offScreenContext.drawImage(
                       rawImage,0,0,newWidth,newHeight,this);
```

Then we need to draw a new offscreen image based on the values of the **newWidth** and **newHeight** instance variables.

First, however, we use **clearRect()** to clear a rectangular area in the offscreen graphics context the size of the raw image, beginning at the upper left hand corner.

Then we use a scaling version of the **drawImage()** method to draw our new image to the offscreen graphics context and scale it to the new desired size in the process. The scaling version executes much more slowly than the non-scaling version. However, this isn't a problem because we are doing this drawing out of sight of the user. (Of course, if it is too slow, we will never be able to keep up with the overall animation objective.)

Then we exit the **paint()** method, leaving the new image to be rendered to the screen the next time the method in invoked.

That concludes our discussion for the double-buffered mode. The other possibility is that the user has selected the non-buffered mode. This is illustrated in the next code fragment.

First we invoke the **clearRect()** method on the screen's graphics context to clear a rectangular area the size of the raw image.

Then we use a scaling version of the **drawImage()** method to draw the new image on the screen, scaling it in the process.

As mentioned earlier, this scaling and drawing are performed in full view of the user which results in an annoying flicker, and in some cases renders the process useless.

```
      g.clearRect(inLeft,inTop,rawWidth,rawHeight);
      g.drawImage(
            rawImage,inLeft,inTop,newWidth,newHeight,this);
```

Note that in both the buffered and the non-buffered cases, we used the overloaded **drawImage()** method to actually draw the image on the screen. The major difference, however, is that in the non-buffered case, we used a version of the method that is required to scale the image as it is being drawn. This causes it to be significantly slower than the other version that simply transfers a previously scaled image directly to the screen with no significant processing along the way.

Program Listing

A complete listing of this program follows. Some of the interesting code fragments are highlighted in **boldface**. A description of the operational features of the program was provided earlier.

```
/*File Image04.java
Copyright 1997, R.G.Baldwin

This program illustrates animation with or without double
buffering. It requires access to an image file named
"logomain.gif" in the current directory on the hard disk.

To run the program in double-buffer mode, enter the
following at the command line:

java image04

To run the program without double buffering, enter the
following at the command line:

java image04 x

The program will continue to animate the image until you
press the close button on the Frame object.

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

class Image04 extends Frame{ //controlling class
  Image rawImage;//ref to raw image file fetched from disk
  int rawWidth;
  int rawHeight;

  //Width and height values for a particular
  // animation frame
  int newWidth;
  int newHeight;

  //Inset values for the container object
  int inTop;
  int inLeft;

  //References to objects used for double buffering
  Image offScreenImage;
  Graphics offScreenContext;

  //To double buffer or not double buffer
  boolean doubleBuf = true;

  //======================================================//

  public static void main(String[] args){
    Image04 obj = new Image04();//instantiate this object
```

```
  //Declare some local variables
  double delta = 0;
  double scale = 0;

  //Set for double buffering or not based on command-
  // line args.  Default is double buffering.  Enter any
  // command-line argument to disable double buffering.
  if(args.length == 0){
    obj.doubleBuf = true;
    System.out.println("Double buffer mode");
  }//end if
  else{
    obj.doubleBuf = false;
    System.out.println("Not double buffer mode");
  }//end else

  //Loop in animated mode until close button is clicked
  while(true){
    //Reverse direction of motion if necessary
    if(scale >= 0.999) delta = -0.005;//move into screen
    if(scale <= 0.015) delta = 0.005;//move out of screen

    //Establish width and height for this rendering
    // of the image.
    obj.newWidth = (int)(scale*obj.rawWidth);
    obj.newHeight = (int)(scale*obj.rawHeight);
    obj.repaint();//render the image

    scale += delta;//update scale for next rendering

    //Sleep for awhile. Animate at approximately 20
    // frames per second.
    try{
      Thread.currentThread().sleep(50);
    }catch(InterruptedException e){System.out.println(e);}
  }//end while loop
 }//end main

//=====================================================//

public Image04(){//constructor
  //Get an image from the specified file in the current
  // directory on the local hard disk.
  rawImage =
    Toolkit.getDefaultToolkit().getImage("logomain.gif");

  //Use a MediaTracker object to block until the image
  // is loaded or ten seconds has elapsed.
  MediaTracker tracker = new MediaTracker(this);
  tracker.addImage(rawImage,1);

  try{
    if(!tracker.waitForID(1,10000)){
      System.out.println("Load error.");
      System.exit(1);
```

```
      }//end if
    }catch(InterruptedException e){System.out.println(e);}

    //Raw image has been loaded.  Establish width and
    // height of the raw image.
    rawWidth = rawImage.getWidth(this);
    rawHeight = rawImage.getHeight(this);

    this.setVisible(true);//make the Frame visible

    //Get and store inset data for the Frame object so
    // that it can be easily avoided.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;

    //Use the insets and the size of the raw image to
    // establish the overall size of the Frame object.
    this.setSize(inLeft+rawWidth,inTop+rawHeight);
    this.setTitle("Copyright 1997, Baldwin");
    this.setBackground(Color.yellow);

    //Get an offscreen image object to draw on.  Then get
    // the graphics context for that offscreen image so
    // that it can be drawn on.
    offScreenImage = this.createImage(rawWidth,rawHeight);
    offScreenContext = offScreenImage.getGraphics();

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor
//=====================================================//

  //Override the update() method to eliminate unnecessary
  // erasing of the screen and the flashing caused by
  // such unnecessary erasing.  This requires screen
  // erasure to be handled in the overridden paint()
  // method.
  public void update(Graphics g){
    paint(g);
  }//end overridden update() method
  //=====================================================//

  //Override the paint method
  public void paint(Graphics g){
    if(doubleBuf){//use double buffering
      //Render (to the screen) the image previously
      // created in the offscreen image
      g.drawImage(offScreenImage,inLeft,inTop,this);

      //Scale and draw the next image in the offscreen
```

```
      // area using the instance variables named newWidth
      // and newHeight to establish the size.  It will be
      // rendered to the screen the next time paint() is
      // invoked.  Note that the maximum drawing size is
      // limited to the size of the raw image.
      offScreenContext.clearRect(
                                0,0,rawWidth,rawHeight);
      offScreenContext.drawImage(
                     rawImage,0,0,newWidth,newHeight,this);
    }//end if
    else{ //don't use double buffering
      g.clearRect(inLeft,inTop,rawWidth,rawHeight);
      g.drawImage(
            rawImage,inLeft,inTop,newWidth,newHeight,this);
    }//end else
  }//end paint()
}//end Image04 class
//=======================================================//
```

-end-