Chapter 1. An Animation Framework

A core technology for a good game is an animation algorithm that produces reliably fast game play across various OSes (e.g. flavours of Windows, Linux, the Macintosh), and in different kinds of Java programs (e.g. applets, windowed and full-screen applications).

We distinguish between windowed and full-screen applications since J2SE v1.4 introduced *full-screen exclusive mode* (FSEM). It suspends the normal windowing environment, and allows an application to more directly access the underlying graphics hardware. It permits techniques such as page flipping and provides control over the screen's resolution and image depth. The principal aim of FSEM is to speed up graphics-intensive applications, such as games.

The animation algorithm developed through most of this chapter is embedded in a JPanel subclass (called GamePanel), which acts as a canvas for drawing 2D graphics (e.g. lines, circles, text, images). The animation is managed by a thread which ensures that it progresses at a consistent rate, as independent of the vagaries of the hardware and OS as possible. The rate is measured in terms of frames per second (FPS), where a frame corresponds to a single rendering of the application (game) to the canvas.

GamePanel is gradually refined and expanded through the chapter, introducing notions such as:

- the {update, render, sleep} animation loop;
- starting and terminating an animation;
- double buffering;
- user interaction;
- active rendering;
- animation control based on a user's requested FPS;
- the management of inaccuracies in the timer and sleep operations;
- combining FPS and UPS (game state updates per second);
- game pausing and resumption.

We also examine two other ways of coding animation, using the Swing timer and the 'utility' timer in java.util timer.

In chapters 2 and 3, we develop applet, windowed and full-screen applications for a WormChase game using the final version of GamePanel (with minor variations). As a side-effect of the game play, statistics are gathered, including the average FPS and UPS, to show that GamePanel supports consistently high-speed animation.

1. Animation as a Threaded Canvas

A JPanel is employed as a drawing surface, and an animation loop is embedded inside a thread local to the panel. The loop consists of three stages: game update, rendering, and a short sleep.

The following code shows the main elements of GamePanel, including the run() method containing the animation loop. As the chapter progresses, additional methods

and global variables will be added to GamePanel, and some of the existing methods (especially run()) will be changed and extended.

```
public class GamePanel extends JPanel implements Runnable
{
  private static final int PWIDTH = 500;   // size of panel
  private static final int PHEIGHT = 400;

  private Thread animator;            // for the animation
  private boolean running = false;    // stops the animation

  private boolean gameOver = false;   // for game termination

  // more variables, explained later
        :

  public GamePanel()
  {
    setBackground(Color.white);    // white background
    setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

    // create game components
    ...
  }  // end of GamePanel()


  public void addNotify()
  /* Wait for the JPanel to be added to the
     JFrame/JApplet before starting. */
  {
    super.addNotify();   // creates the peer
    startGame();         // start the thread
  }


  private void startGame()
  // initialise and start the thread
  {
    if (animator == null || !running) {
      animator = new Thread(this);
      animator.start();
    }
  } // end of startGame()


  public void stopGame()
  // called by the user to stop execution
  {  running = false;   }


  public void run()
  /* Repeatedly update, render, sleep */
  {
    running = true;
    while(running) {
      gameUpdate();   // game state is updated
      gameRender();   // render to a buffer
      repaint();      // paint with the buffer

      try {
```



```
        Thread.sleep(20);  // sleep a bit
      }
      catch(InterruptedException ex){}
    }
    System.exit(0);   // so enclosing JFrame/JApplet exits
  } // end of run()


  private void gameUpdate()
  { if (!gameOver)
      // update game state ...
  }

  // more methods, explained later...

}  // end of GamePanel class
```

GamePanel is a white canvas with fixed dimensions. A GamePanel object will be added to a JFrame in an application, and a JApplet in an applet. Examples can be found in chapter 2.

For full-screen applications, the coding choices are larger, and chapter 3 describes three different approaches. The full-screen exclusive mode (FSEM) version requires the largest number of changes to GamePanel, but the animation loop stays essentially the same.

addNotify() is called automatically as GamePanel is being added to its enclosing GUI component (e.g. a JFrame or JApplet), and so is a good place to initiate the animation thread (animator).

stopGame() will be called from the enclosing JFrame/JApplet when the user wants the program to terminate; it sets a global boolean, running, to false. Some authors suggest using Thread's stop() method, a technique deprecated by Sun. stop() causes a thread to terminate immediately, perhaps while it is changing data structures or manipulating external resources, causing them to be left in an inconsistent state. The running boolean is a better solution since it allows the programmer to decide how the animation loop should finish. The drawback is that the code must include tests to detect the termination flag.

1.1. Synchronization Concerns

The executing GamePanel object has two main threads: the animator thread for game updates and rendering, and a GUI event processing thread, which responds to such things as key presses and mouse movements. When the user presses a key to stop the game, this *event dispatch thread* will execute stopGame(). It will set running to false at the same time as the animation thread is executing.

Once a program contains two or more threads utilizing a shared variable, data structure, or resource, then thorny *synchronization problems* may appear. For example, what will happen if a shared item is changed by one thread at the same moment that the other one reads it?

The running flag is changed from true to false by stopGame() – a fast, single assignment. The animation thread only examines the boolean at the start of its while loop in run(), and only to test if it is true. If by some slim chance it examines the

variable at the same moment as the assignment occurs then the value may be undefined, and so cause the loop to terminate, which is the desired aim anyway.

In practice, assignments to booleans are completed so quickly, that the possibility of a synchronization problem arising can be ignored. This is not the case for changes to more complex data structures, which we consider in chapter 2.

1.2. Application and Game Termination

A common pitfall is to use a boolean, such as running, to denote application termination *and* game termination. The end of a game occurs when the player wins (or loses), but this is not typically the same as stopping the application. For instance, the end of the game may be followed by the user entering details into a high scores table, or by the user being given the option to play again. Consequently, we represent game ending by a separate boolean, gameOver. It can be seen in gameUpdate(), controlling the game state change.

1.3. Why Sleep?

The animation loop includes an arbitrary 20ms of sleep time:

```
while(running) {
  gameUpdate();   // game state is updated
  gameRender();   // render to a buffer
  repaint();      // paint with the buffer

  try {
    Thread.sleep(20);  // sleep a bit
  }
  catch(InterruptedException ex){}
}
```

Why is this necessary? There are three main reasons.

The first is that sleep() causes the animation thread to stop executing, and so frees up the CPU for other tasks, such as garbage collection by the JVM. Without a period of sleep, the GamePanel thread could hog all the CPU time. However, the 20ms sleep time is somewhat excessive, especially when the loop is executing 50 or 100 times per second.

The second reason for the sleep() call is to give the preceding repaint() time to be processed. The call to repaint() places a *repaint request* in the JVM's event queue, and then returns. Exactly how long the request will be held in the queue before triggering a repaint is beyond our control. The sleep() call makes the thread wait before starting the next update/rendering cycle, to give the JVM time to act. The repaint request will eventually be processed, percolating down through the components of the application until GamePanel's paintComponent() is called. An obvious question is whether 20ms is sufficient time for the request to be carried out? Perhaps it is overly generous?

The sleep() call reduces the chance of *event coalescence*: if the JVM is overloaded by repaint requests it may choose to combine requests. This means that some of the rendering request will be skipped, causing the animation to 'jump' as frames are lost.

2. Double Buffering Drawing

gameRender() draws into its own Graphics object (dbg), which represents an image the same size as the screen (dbImage).

```
// global variables for off-screen rendering
private Graphics dbg;
private Image dbImage = null;
      :

private void gameRender()
// draw the current frame to an image buffer
{
  if (dbImage == null){  // create the buffer
    dbImage = createImage(PWIDTH, PHEIGHT);
    if (dbImage == null) {
      System.out.println("dbImage is null");
      return;
    }
    else
      dbg = dbImage.getGraphics();
  }

  // clear the background
  dbg.setColor(Color.white);
  dbg.fillRect (0, 0, PWIDTH, PHEIGHT);

  // draw game elements
  ...

  if (gameOver)
    gameOverMessage(dbg);
}  // end of gameRender()


private void gameOverMessage(Graphics g)
// center the game-over message
{ ...
  g.drawString(msg, x, y);
}  // end of gameOverMessage()
```

This technique is known as *double buffering*, since the (usually complex) drawing operations required for rendering are not applied directly to the screen, but to a secondary image.

The dbImage image is placed on screen by paintComponent() as a result of the repaint request in the run() loop. This call is only made after the rendering step has been completed.

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  if (dbImage != null)
    g.drawImage(dbImage, 0, 0, null);
}
```


The principle advantage of double buffering is to reduce on-screen flicker. If extensive drawing is done directly to the screen, the process may take long enough to become noticeable by the user. The call to drawImage() in paintComponent() is fast enough that the change from one frame to the next is perceived as instantaneous.

Another reason for keeping paintComponent() simple is that it may be called by the JVM independently of the animation thread. For example, this will occur when the application (or applet) window has been obscured by another window, and then brought back to the front.

The placing of game behaviour inside paintComponent() is a common mistake. This will mean that the animation will be driven forward by its animation loop *and* by the JVM repainting the window.

3. Adding User Interaction

In full-screen applications, there will be no additional GUI elements such as text fields or Swing buttons. Even in applets or windowed applications, the user will probably want to interact directly with the game canvas as far as possible. This means that GamePanel must be able to monitor key presses and mouse activity.

GamePanel utilizes key presses to set the running boolean to false, which terminates the animation loop and application. Mouse presses are processed by testPress(), using the cursor's (x,y) location in various ways (details are given in later chapters).

The GamePanel() constructor is modified to set up the key and mouse listeners:

```
public GamePanel()
{
  setBackground(Color.white);
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  setFocusable(true);
  requestFocus();    // JPanel now receives key events
  readyForTermination();

  // create game components
  ...

  // listen for mouse presses
  addMouseListener( new MouseAdapter() {
    public void mousePressed(MouseEvent e)
    { testPress(e.getX(), e.getY()); }
  });
}  // end of GamePanel()
```

readyForTermination() watches for key presses that signal termination, and sets running to false. testPress() does something with the cursor's (x,y) coordinate, but only if the game has not yet finished.

```
private void readyForTermination()
  {
    addKeyListener( new KeyAdapter() {
```



```
    // listen for esc, q, end, ctrl-c
       public void keyPressed(KeyEvent e)
       { int keyCode = e.getKeyCode();
         if ((keyCode == KeyEvent.VK_ESCAPE) ||
             (keyCode == KeyEvent.VK_Q) ||
             (keyCode == KeyEvent.VK_END) ||
             ((keyCode == KeyEvent.VK_C) && e.isControlDown()) ) {
           running = false;
         }
       }
     });
  }  // end of readyForTermination()


  private void testPress(int x, int y)
  // is (x,y) important to the game?
  {
    if (!gameOver) {
      // do something
    }
  }
```

4. Active Rendering

Since a call to repaint() is only a request, it is difficult to know when the repaint has actually been completed. This means that the sleep time in the animation loop is little more than a guess: if the specified delay is too long then the animation speed is impaired for no reason. If the delay is too short then repaint requests may be queued by the JVM, and skipped if the load becomes too large.

In fact, no single sleep time is satisfactory since the time taken to update and render a frame will vary depending on the activity taking place in the game. The sleep time must be calculated afresh each time round the loop after measuring the iteration's update and rendering periods. Unfortunately, the repaint() part of the rendering is done by the JVM so cannot be easily measured.

As a first step to dealing with these issues, we switch to *active rendering*, shown below as modifications to run():

```
  public void run()
  /* Repeatedly update, render, sleep */
  {
    running = true;
    while(running) {
      gameUpdate();   // game state is updated
      gameRender();   // render to a buffer
      paintScreen();  // draw buffer to screen

      try {
        Thread.sleep(20);  // sleep a bit
      }
      catch(InterruptedException ex){}
    }
    System.exit(0);
  } // end of run()
```



```
private void paintScreen()
// actively render the buffer image to the screen
{
  Graphics g;
  try {
    g = this.getGraphics();  // get the panel's graphic context
    if ((g != null) && (dbImage != null))
      g.drawImage(dbImage, 0, 0, null);
    g.dispose();
  }
  catch (Exception e)
  { System.out.println("Graphics context error: " + e);  }
} // end of paintScreen()
```

The call to repaint() is gone, as is the overriding of paintComponent(); its role has been taken by paintScreen().

Active rendering puts the task of rendering the buffer image to the screen into our hands. This means that the rendering time can be accurately measured, and concerns about repaint requests being delayed or skipped by the JVM disappear.

The panel's graphics context may be changed by the JVM, typically when the canvas is resized or when it becomes the front window after being behind others. Also, the context may disappear if the application or applet exits while the animation thread is running.

For these reasons, the graphics context must be freshly obtained each time it is needed (by calling getGraphics()). Also, its use must be surrounded by a try-catch block to capture any failure due to its disappearance.

In practice, if the program has a fixed window size, then the most likely time for an exception is when a game *applet* is terminated by the user closing its surrounding Web page.

5. FPS and Sleeping for Varying Times

A weakness of the animation loop is that its execution speed is unconstrained. On a slow machine, it may loop 20 times per second; the same code on a fast machine may loop 80 times, making the game progress 4 times faster, and perhaps become unplayable. The loop's execution speed should be about the same on all platforms.

A popular measure of how fast an animation progresses is the number of frames shown per second (FPS). For GamePanel, a frame corresponds to a single pass through the update-render-sleep loop inside run().

Therefore, a desired 100 FPS implies that each iteration of the loop should take 1000/100 == 10ms. This iteration time is stored in the `period` variable in GamePanel.

The use of active rendering makes it possible to time the update and render stages of each iteration. Subtracting this value from period gives the sleep time required to maintain the desired FPS. For instance, 100 FPS means a period of 10ms, and if the update/render steps take 6ms, then sleep() should be called for 4ms.

The following modified run() method includes timing code and the sleep time calculation:

```
public void run()
/* Repeatedly: update, render, sleep so loop takes close
   to period ms */
{
  long beforeTime, timeDiff, sleepTime;

  beforeTime = System.currentTimeMillis();

  running = true;
  while(running) {
    gameUpdate();
    gameRender();
    paintScreen();

    timeDiff = System.currentTimeMillis() - beforeTime;
    sleepTime = period - timeDiff;   // time left in this loop

    if (sleepTime <= 0)  // update/render took longer than period
      sleepTime = 5;    // sleep a bit anyway

    try {
      Thread.sleep(sleepTime);  // in ms
    }
    catch(InterruptedException ex){}

    beforeTime = System.currentTimeMillis();
  }

  System.exit(0);
} // end of run()
```

timeDiff holds the execution time for the update and render steps, which becomes part of the sleep time calculation.

One problem with this approach is if the update and drawing take longer than the specified period: the sleep time becomes negative! The simple answer is to set the time to some small value to make the thread sleep a bit. This permits other threads, and the JVM, to execute if they wish. Obviously this solution is problematic: why 5ms, and not 2 or 20?

A more subtle issue is the resolution and accuracy of the timer and sleep operations (currentTimeMillis() and sleep()). If they return poor values, then the resulting FPS will be affected.

5.1. Timer Resolution

Timer resolution, or *granularity*, is the amount of time that must separate two timer calls so that different values are returned. For instance, what is the value of diff in the code fragment below?

```
long t1 = System.currentTimeMillis();
long t2 = System.currentTimeMillis();
long diff = t2 - t1;  // in ms
```


The value depends on the resolution of currentTimeMillis(), which unfortunately depends on the OS (to be more precise, the resolution of the standard clock interrupt).

In Windows 95/98, the resolution is 55ms, which means that repeated calls to currentTimeMillis() will only return different values roughly every 55ms.

In the animation loop, the overall effect of poor resolution is to cause the animation to run slower than intended, reducing the FPS. This is due to the timeDiff value, which will be set to 0 if the game update and rendering time is less than 55ms. This causes the sleep time to be assigned the iteration period value, rather than a smaller amount, causing each iteration to sleep longer than necessary.

To try to combat this, the minimum iteration period in GamePanel should be greater than 55ms, indicating an upper limit of about 18 FPS. This frame rate is widely considered inadequate for games, since the slow screen refresh appears as excessive flicker.

On Windows 2000, NT, and XP, currentTimeMillis() has a resolution of 10-15ms, making it possible to obtain 67-100 FPS. This is considered acceptable-to-good for games. The Mac OS X and Linux have timer resolutions of 1ms, which is excellent.

5.2. What is a good FPS?

It's worth taking a brief diversion to consider what FPS value makes for a good game.

A lower bound is dictated by the human eye, and the *critical flicker frequency* (CFF), the rate at which a flickering light appears to be continuous. This occurs somewhere between 10-50Hz, depending on the intensity of the light (which translates into 10-50 FPS). For larger images, the position of the user relative to the image affects the perceived flicker, as well as the colour contrasts and amount of detail in the picture.

Movies are shown at 24 FPS, but this number is somewhat misleading since each frame is projected onto the screen twice (or perhaps 3 times) by the rapid opening and closing of the projector's shutter. Thus, the viewer is actually receiving 48 or 72 image flashes per second.

An upper bound for a good FPS value is the monitor refresh rate. This is typically 70-90Hz (i.e. 70-90 FPS). There is no reason for a program to send more frames per second to the graphics card since they will not be displayed. In fact, an excessive FPS rate consumes needless CPU time and over-stretches the display card.

My monitor refreshes at 85Hz, making 80-85 FPS the goal of the code here.

5.3. Are We Done Yet?

Since the aim is about 85 FPS, then is the current animation loop sufficient for the job? Do we have to complicate it any further? For modern versions of Windows (e.g. NT, 2000, XP), the Mac, and Linux, the average/good timer resolutions mean that the current code is probably just about adequate.

The main problem is the resolution of the Windows 98 timer (55 ms; 18.2 FPS). Google Zeitgeist, a Web site which reports interesting search patterns and trends taken from the Google search engine (http://www.google.com/press/zeitgeist.html), lists OSes used to access Google. Windows 98 usage is currently at about 29%

(September 2003), having dropped from 42% the previous September. The winner has mostly been XP, gaining ground from 20% to 38% in the same interval.

If we are prepared to extrapolate OS popularity from these search engine figures, then it looks like Windows 98 is on its way out. By the time you read this in book form, sometime towards the end of 2004, Windows 98's share of the OS market will probably be below 20%. It may be acceptable to just *ignore* the slowness of its timer since fewer people will be using it.

5.4. Improved Standard Java Timers and Counters

J2SE 1.4.2 has a microsecond accurate timer, hidden in the undocumented class sun.misc.Perf. The diff calculation can be expressed as:

```
Perf perf = Perf.getPerf();
long countFreq = perf.highResFrequency();

long count1 = perf.highResCounter();
long count2 = perf.highResCounter();
long diff =  (count2 - count1) * 1000000000L / countFreq ;
                  // in nanoseconds
```

Perf is not a timer but a high resolution counter, and so is suitable for measuring time *intervals*. highResCounter() returns the current counter value, and highResFrequency() the number of counts made per second. Perf's typical resolution is a few microseconds (2-6 microseconds on different versions of Windows).

Our timer problems will be solved with the release of J2SE 1.5 (codenamed 'Tiger') due early in 2004. It will have System.nanoTime() method, which appears to be counter-based like Perf's highResCounter(). Also, the new java.util.concurrent package for concurrent programming, will include a TimingUnit class that can measure down to the nanosecond level.

5.5. High Performance Counters

Another alternative to currentTimeMillis() is to employ a higher resolution timer from one of Java's extensions. The Java Media Framework (JMF) timer is a possibility, but since the majority of this book is about Java 3D, we'll use the J3DTimer class.

The diff calculation recoded using the Java 3D timer becomes:

```
long t1 = J3DTimer.getValue();
long t2 = J3DTimer.getValue();
long diff =  t2 - t1 ;   // in nanoseconds
```

getValue() returns a time in nanoseconds. On Windows 98, the Java 3D timer has a resolution of about 900 nsecs, which improves to under 300 nsecs on our test XP box.

The principal drawback of using Java 3D is the need to install it in addition to J2SE. Details about how to do this are given in chapter 8 (not yet ??). A set of Powerpoint slides introducing Java 3D is available at http://fivedots.coe.psu.ac.th/~ad/jg/labs/. Sun's top-level Web page for Java 3D is http://java.sun.com/products/java-media/3D/.

One alternative is to extract only the timer-related files from the installation. The J3DTimer class is located in `<JAVA_HOME>\jre\lib\ext\j3dUtils.jar`. In Windows, the J3DTimer class is a thin layer surrounding native method calls carried out by `j3DUtils.dll` (located in `<JAVA_HOME>\jre\bin`).

`j3DUtils.dll` utilizes Window's QueryPerformanceCounter() and QueryPerformanceFrequency() functions. QueryPerformanceCounter() returns the current value of the counter. QueryPerformanceFrequency() returns the number of counts generated per second, stored as a 64 bit integer. For instance, if QueryPerformanceFrequency() returns 3,000,000, then it takes 3,000,000 counter ticks for 1 second to pass, 3000 ticks for 1 ms. The time (in seconds) that has passed between two successive calls to QueryPerformanceCounter() is obtained by dividing the difference by QueryPerformanceFrequency().

Another approach is to use a timer package from one of the game engines on the net. My favourite is *Meat Fighter* by Michael Birken (http://www.meatfighter.com). The animation loop in Meat Fighter had a major influence on the code described here. The StopWatchSource class provides a static method getStopWatch() which uses the best resolution timer available in your system; it considers currentTimeMillis(), and the JMF and Java 3D timers, if present. On Windows, Meat Fighter includes a 40K DLL containing a high resolution timer using QueryPerformanceCounter() and QueryPerformanceFrequency().

5.6. Measuring Timer Resolution

The TimerRes class offers a simple way to discover the resolution of the System, Perf and Java 3D timers on your machine. Of course, Perf is only available in J2SE 1.4.2, and Java 3D must be installed in order for J3DTimer.getResolution() to work.

```
import com.sun.j3d.utils.timer.J3DTimer;

public class TimerRes
{
  public static void main(String args[])
  { j3dTimeResolution();
    sysTimeResolution();
    perfTimeResolution();
  }

  private static void j3dTimeResolution()
  { System.out.println("Java 3D Timer Resolution: " +
                       J3DTimer.getResolution() + " nsecs");
  }

  private static void sysTimeResolution()
  {
    long total, count1, count2;

    count1 = System.currentTimeMillis();
    count2 = System.currentTimeMillis();
    while(count1 == count2)
      count2 = System.currentTimeMillis();
    total = 1000L * (count2 - count1);
```



```
    count1 = System.currentTimeMillis();
    count2 = System.currentTimeMillis();
    while(count1 == count2)
      count2 = System.currentTimeMillis();
    total += 1000L * (count2 - count1);

    count1 = System.currentTimeMillis();
    count2 = System.currentTimeMillis();
    while(count1 == count2)
      count2 = System.currentTimeMillis();
    total += 1000L * (count2 - count1);

    count1 = System.currentTimeMillis();
    count2 = System.currentTimeMillis();
    while(count1 == count2)
      count2 = System.currentTimeMillis();
    total += 1000L * (count2 - count1);

    System.out.println("System Time resolution: " +
                           total/4 + " microsecs");
  } // end of sysTimeResolution()


  private static void perfTimeResolution()
  {
    StopWatch sw = new StopWatch();
    System.out.println("Perf Resolution: " +
                   sw.getResolution() + " nsecs");

    sw.start();
    long time = sw.stop();
    System.out.println("Perf Time " + time  + " nsecs");
  }

} // end of TimerRes class
```

StopWatch is our own class, and wraps up the Perf counter to make it easier to use as a kind of stopwatch. There is also a getResolution() method.

```
import sun.misc.Perf;    // only on J2SE 1.4.2

public class StopWatch
{
  private Perf hiResTimer;
  private long freq;
  private long startTime;

  public StopWatch()
  { hiResTimer = Perf.getPerf();
    freq = hiResTimer.highResFrequency();
  }

  public void start()
  {  startTime = hiResTimer.highResCounter(); }


  public long stop()
  //  return the elapsed time in nanoseconds
  {  return (hiResTimer.highResCounter() -
                           startTime)*1000000000L/freq;  }
```



```
  public long getResolution()
  // return counter resolution in nanoseconds
  {
    long diff, count1, count2;

    count1 = hiResTimer.highResCounter();
    count2 = hiResTimer.highResCounter();
    while(count1 == count2)
      count2 = hiResTimer.highResCounter();
    diff = (count2 - count1);

    count1 = hiResTimer.highResCounter();
    count2 = hiResTimer.highResCounter();
    while(count1 == count2)
      count2 = hiResTimer.highResCounter();
    diff += (count2 - count1);

    count1 = hiResTimer.highResCounter();
    count2 = hiResTimer.highResCounter();
    while(count1 == count2)
      count2 = hiResTimer.highResCounter();
    diff += (count2 - count1);

    count1 = hiResTimer.highResCounter();
    count2 = hiResTimer.highResCounter();
    while(count1 == count2)
      count2 = hiResTimer.highResCounter();
    diff += (count2 - count1);

    return (diff*1000000000L)/(4*freq);
  } // end of getResolution()

} // end of StopWatch class
```

The start() and stop() interface adds a small overhead to the counter, as illustrated in the perfTimeResolution() method in TimerRes. The smallest time that can be obtained is around 10-40 microsecs, compared to the resolution of around 2-6 microsecs.

6. Sleep Accuracy

The accuracy of the sleep() call in the animation loop is important for maintaining the required period. The SleepAcc class calls sleep() with increasingly small values, and measures the actual sleep time using the Java 3D timer.

```
import java.text.DecimalFormat;
import com.sun.j3d.utils.timer.J3DTimer;

public class SleepAcc
{
  private static DecimalFormat df;

  public static void main(String args[])
  {
    df = new DecimalFormat("0.##");  // 2 dp
```



```
    // test various sleep values
    sleepTest(1000);
    sleepTest(500);
    sleepTest(200);
    sleepTest(100);
    sleepTest(50);
    sleepTest(20);
    sleepTest(10);
    sleepTest(5);
    sleepTest(1);
  } // end of main()


  private static void sleepTest(int delay)
  {
    long timeStart = J3DTimer.getValue();

    try {
      Thread.sleep(delay);
    }
    catch(InterruptedException e) {}

    double timeDiff =
       ((double)(J3DTimer.getValue() - timeStart))/(1000000L);
    double err = ((delay - timeDiff)/timeDiff) * 100;

    System.out.println("Slept: " + delay + " ms  J3D: " +
                          df.format(timeDiff) + " ms  err: " +
                          df.format(err) + " %" );
  }  // end of sleepTest()

} // end of SleepAcc class
```

The difference between the requested and actual sleep delay is negligible for times of 50ms or more, then gradually increases to a +/-10-20% error at 5ms. The % variation between different 1ms tests is enormous, sometimes amounting to +/-100-200%.

The reason for this inaccuracy is probably due to the complexity of the operation, involving the suspension of a thread, and context switching with other activities. Also, even after the sleep time has finished, a thread still has to wait to be selected for execution by the thread scheduler. How long it has to wait depends on the overall load of the JVM (and OS) at that moment.

sleep()'s implementation varies between OSes and different versions of Java, making analysis difficult. Under Windows 98 and J2SE 1.4.2, sleep() utilizes a large native function (located in jvm.dll) which employs the Windows kernel sleep() function (which has a reported accuracy of 1ms).

The conclusion is that we should extend the animation loop to combat sleep()'s inaccuracies.

6.1. Handling Sleep Inaccuracies

This version of run() in this section has been revised in three main ways:

1) it uses the Java 3D timer;

2) sleep()'s execution time is measured, and the error (stored in overSleepTime) is used to adjust the sleeping period in the next iteration;
3) Thread.yield() is utilized to give other threads a chance to execute if the animation loop has not slept for a while.

```
private static final int NO_DELAYS_PER_YIELD = 16;
  /* Number of frames with a delay of 0 ms before the
     animation thread yields to other running threads. */
          :

  public void run()
  /* Repeatedly update, render, sleep so loop takes close
     to period nsecs. Sleep inaccuracies are handled.
     The timing calculation use the Java 3D timer.
  */
  {
    long beforeTime, afterTime, timeDiff, sleepTime;
    long overSleepTime = 0L;
    int noDelays = 0;

    beforeTime = J3DTimer.getValue();

    running = true;
    while(running) {
      gameUpdate();
      gameRender();
      paintScreen();

      afterTime = J3DTimer.getValue();
      timeDiff = afterTime - beforeTime;
      sleepTime = (period - timeDiff) - overSleepTime;

      if (sleepTime > 0) {   // some time left in this cycle
        try {
          Thread.sleep(sleepTime/1000000L);  // nano -> ms
        }
        catch(InterruptedException ex){}
        overSleepTime =
              (J3DTimer.getValue() - afterTime) - sleepTime;
      }
      else {    // sleepTime <= 0; frame took longer than the period
        overSleepTime = 0L;

        if (++noDelays >= NO_DELAYS_PER_YIELD) {
          Thread.yield();   // give another thread a chance to run
          noDelays = 0;
        }
      }

      beforeTime = J3DTimer.getValue();
    }

    System.exit(0);
  } // end of run()
```


If the sleep() call sleeps for 12ms instead of the desired 10ms, then overSleepTime will be assigned 2ms. On the next iteration of the loop, this value will be deducted from the sleep time, reducing it by 2ms. In this way, sleep inaccuracies are corrected.

If the game update and rendering steps take longer than the iteration period, then sleepTime will have a negative value, and this iteration will not include a sleep stage. This causes the noDelays counter to be incremented, and when it reaches `NO_DELAYS_PER_YIELD`, yield() will be called. This allows other threads to execute if they need to, and avoids the use of an arbitrary sleep period in run().

The switch to the Java 3D timer is mostly a matter of changing the calls to System.currentTimeMillis() to J3DTimer.getValue(). Time values change from milliseconds to nanoseconds, which motivates the change to long variables. Also, the sleep time must be converted from nsecs to msecs before calling sleep() (or we'll be waiting a long time for the game to wake up).

7. FPS and UPS

Apart from FPS, there is another useful measure of animation speed: updates per second (UPS). The current animation loop carries out one update and one render in each iteration, but this correspondence isn't necessary. The loop could carry out two updates per each rendering, as illustrated by the code fragment below:

```
public void run()
// Repeatedly update, render, sleep
{ ...
  running = true;
  while(running) {
    gameUpdate();   // game state is updated
    gameUpdate();   // game state is updated again

    gameRender();   // render to a buffer
    paintScreen();  // paint with the buffer

    // sleep a bit
  }
  System.exit(0);
} // end of run()
```

If the game offers 50 FPS (i.e. 50 iterations of the animation loop/second), then it is doing 100 updates/sec.

This coding style causes the game to advance more quickly since the game state is changing twice as fast, but at the cost of skipping the rendering of those extra states. However, this may not be noticeable, especially if the FPS value is high.

7.1. Separating Updates from Rendering

One limitation on high FPS rates is the amount of time that the update and render steps require. Satisfying a period of 5ms (1000/5 == 200 FPS) is impossible if these

steps take more than 5ms to accomplish. One point to note is that most of this execution time is usually consumed by the rendering stage.

In this situation, the way to increase game speed is to increase the number of updates/second (UPS). In programming terms, this translates into calling gameUpdate () more than once during each iteration. However, too many additional calls will cause the game to 'flicker' as too many successive states are not rendered. Also, each update adds to the execution time, which will further reduce the maximum achievable FPS value.

The new run() is given below:

```
private static int MAX_FRAME_SKIPS = 5;
  // no. of frames that can be skipped in any one animation loop
  // i.e the games state is updated but not rendered
         :


public void run()
/* Repeatedly update, render, sleep so loop takes close
   to period nsecs. Sleep inaccuracies are handled.
   The timing calculation use the Java 3D timer.

   Overruns in update/renders will cause extra updates
   to be carried out so UPS ~== requested FPS
*/
{
  long beforeTime, afterTime, timeDiff, sleepTime;
  long overSleepTime = 0L;
  int noDelays = 0;
  long excess = 0L;

  beforeTime = J3DTimer.getValue();

  running = true;
  while(running) {
    gameUpdate();
    gameRender();
    paintScreen();

    afterTime = J3DTimer.getValue();
    timeDiff = afterTime - beforeTime;
    sleepTime = (period - timeDiff) - overSleepTime;

    if (sleepTime > 0) {   // some time left in this cycle
      try {
        Thread.sleep(sleepTime/1000000L);  // nano -> ms
      }
      catch(InterruptedException ex){}
      overSleepTime =
            (J3DTimer.getValue() - afterTime) - sleepTime;
    }
    else {    // sleepTime <= 0; frame took longer than the period
      excess -= sleepTime;  // store excess time value
      overSleepTime = 0L;

      if (++noDelays >= NO_DELAYS_PER_YIELD) {
        Thread.yield();   // give another thread a chance to run
        noDelays = 0;
      }
```



```
      }

      beforeTime = J3DTimer.getValue();

      /* If frame animation is taking too long, update the game state
         without rendering it, to get the updates/sec nearer to
         the required FPS. */
      int skips = 0;
      while((excess > period) && (skips < MAX_FRAME_SKIPS)) {
        excess -= period;
        gameUpdate();      // update state but don't render
        skips++;
      }
    }

    System.exit(0);
  } // end of run()
```

If the update-render step takes 12ms, and the required period is 10ms, then sleepTime will be –2ms (perhaps even smaller after overSleepTime has been deducted). This excessive execution time is added to the excess variable, which acts as a total of all the overruns by the update-render calls.

When excess exceeds the iteration period, the equivalent of one frame has been lost. A while loop is entered which updates the game for each period amount lost, up to a maximum of `MAX_FRAME_SKIPS` (5 updates). The remaining time overrun is stored for use in a later iteration.

The outcome is that when a game cannot update and render fast enough to match the desired FPS then additional calls will be made to gameUpdate(). This changes the state without rendering it, which the user sees as the game moving 'faster', even though the number of rendered frames remains the same.

8. Pausing and Resuming

How ever exciting the game, there comes a time when the user wants to pause it (and resume later).

One, discredited, coding approach is to use Thread.suspend() and resume(). They are deprecated for a similar reason to Thread.stop() – suspend() can cause an applet/application to suspend at any point in its execution. This can easily lead to deadlock if the thread is holding a resource, since it will not be released until the thread resumes.

The Java documentation for the Thread class recommends using wait() and notify() to implement pause and resume functionality. The idea is to suspend the animation thread, but the event dispatcher thread will still be able to respond to GUI activity.

The approach introduces an isPaused boolean which is set true via pauseGame():

```
// global variable
private boolean isPaused = false;
        :

public void pauseGame()
```



```
{ isPaused = true;   }


public void run()
// Repeatedly (possibly pause) update, render, sleep
// Our code does not use this approach.
{ ...
  running = true;
  while(running) {
    try {
      if (isPaused) {
        synchronized(this) {
          while (isPaused && running)
            wait();
        }
      }
    } // of try block
    catch (InterruptedException e){}

    gameUpdate();   // game state is updated
    gameRender();   // render to a buffer
    paintScreen();  // paint with the buffer

    // sleep a bit
  }
  System.exit(0);
} // end of run()
```

The isPaused flag is detected in run(), and triggers a wait() call to suspend the animation thread.

The thread is resumed by resumeGame() or stopGame(), which both call notify(). These methods must be synchronized so that the animation thread does not miss the notification, and remain suspended indefinitely.

```
public synchronized void resumeGame()
{  isPaused = false;   // we do not do this
   notify();
}


public synchronized void stopGame()
{  running = false;    // we do not do this
   notify();
}
```

This coding style can be criticised for combining two notions: game pausing/resuming *and* program pausing/resuming. This is the main reason why we do *not* use it.

Although the elements of the game seen by the user can pause, it is often very useful for the other parts to continue executing. For example, in a network game, it may be necessary to keep monitoring sockets for messages coming from other players.

The drawback of keeping the application running is the cost of executing the animation thread when the user is not actually playing.

Our approach also uses an isPaused boolean, which is set with pauseGame().


```
// this is our approach
private boolean isPaused = false;
        :

public void pauseGame()
{ isPaused = true;   }
```

However, isPaused is not monitored in run(), since the animation thread does not suspend. isPaused is only used to switch off testPress() and gameUpdate().

```
private void testPress(int x, int y)
// is (x,y) important to the game?
{
  if (!isPaused && !gameOver) {
    // do something
  }
}


private void gameUpdate()
{ if (!isPaused && !gameOver)
    // update game state ...
}
```

Key presses are still handled by the KeyListener method since it must be possible to quit even in the paused state.

isPaused is set to false with resumeGame():

```
public void resumeGame()
{  isPaused = false;  }
```

The animation loop is not suspended when isPaused is set true, so rendering will continue. A program using active rendering cannot rely on the JVM calling repaint(), since the GamePanel's paintComponent() method is empty.

The situations which trigger pausing and resuming vary between the different types of Java programs.

In an applet, the animation should pause when the applet is stopped, and resume when the applet is restarted by the browser. A stop occurs when the user leaves the page – for example, to go to another page. When the user returns to the page, the applet starts itself again. The same sequence should be triggered when the user iconifies the applet's page and later reopens it.

In an application, pausing should be initiated when the window is iconified or deactivated, and execution resume when the window is enlarged or activated. A window is deactivated when it is obscured, and activated when brought back to the front.

In a full-screen application, pausing and resumption will be controlled by buttons on the canvas since the user interface lacks a title bar, and the OS taskbar is hidden.

Examples of these approaches can be found in chapters 2 and 3.

9. Swing Timer Animation

The Swing timer (in javax.swing.Timer) is used as the basis of animation examples in many Java textbooks.

The essential coding technique is to set a Timer object to 'tick' every few milliseconds. Each tick sends an event to a specified ActionEvent listener, triggering a call to actionPerformed(). actionPerformed() calls repaint() to send a repaint request to the JVM. Eventually, repainting reaches the paintComponent() method for the JPanel, which redraws the animation canvas.

These stages are shown in Figure 1, which represents the test code in SwingTimerTest.java.

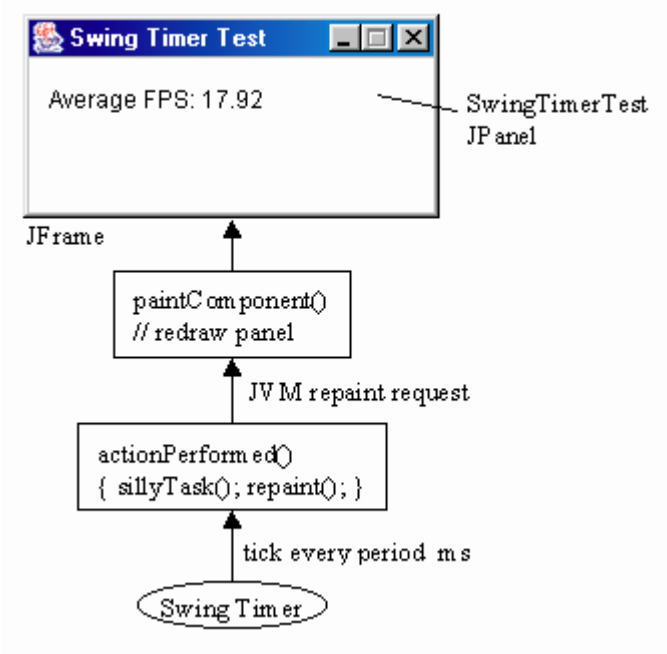

Figure 1. Swing Timer Animation.

The SwingTimerTest class uses the Swing timer to repeatedly draw the current average FPS value into a JPanel. The period for the timer is obtained from the requested FPS given on the command line. The average FPS is calculated every second, based on FPS values collected over the previous 10 seconds.

main() reads in the user's required FPS and converts it to a period. It creates a JFrame, and puts the SwingTimerPanel inside it.

The SwingTimerTest() constructor creates the timer, and sends its 'ticks' to itself:

```
new Timer(period, this).start();
```

actionPerformed() wastes some time by calling a sillyTask() method that does a lot of looping, then requests a repaint:


```
 public void actionPerformed(ActionEvent e)
 { sillyTask();
   repaint();
 }
```

paintComponent() updates the JPanel, and records statistics:

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

  // clear the background
  g.setColor(Color.white);
  g.fillRect (0, 0, PWIDTH, PHEIGHT);

  // report average FPS
  g.setColor(Color.black);
  g.drawString("Average FPS: " + df.format(averageFPS), 10, 25);

  reportStats();   // record/report statistics
} // end of paintComponent()
```

The most complicated part of this example is the statistics gathering done by reportStats(). It is worth looking at the code since it appears again in chapters 2 and 3.

reportStats() prints a line of statistics every second, as shown in figure 2.

```
C>java SwingTimerTest 50
fps: 50; period: 20 ms
1 3.319s 231.9% 50c 15.06 15.06 afps
1 2.8592s 185.92% 100c 16.19 15.63 afps
1 2.7492s 174.92% 150c 16.8 16.02 afps
1 2.8592s 185.92% 200c 16.97 16.26 afps
1 2.7457s 174.57% 250c 17.2 16.44 afps
1 2.7986s 179.86% 300c 17.31 16.59 afps
1 2.8077s 180.77% 350c 17.38 16.7 afps
1 2.84s 184% 400c 17.41 16.79 afps
1 2.8643s 186.43% 450c 17.41 16.86 afps
1 2.7507s 175.07% 500c 17.49 16.92 afps
1 2.8428s 184.28% 550c 17.5 17.17 afps
```

Figure 2. Statistics Output by SwingTimerTest.

The first line of the output lists the requested FPS and the corresponding period used by the timer. It is followed by multiple statistic lines, with a new line generated when the accumulated timer period reaches 1 second since the last line was printed.

Each statistics line presents 6 numbers. The first three relate to the execution time. The first number is the accumulated timer period since the last output, which will be close to 1 second. The second number is the actual elapsed time, measured with the Java 3D timer, and the third value is the percentage error between the two numbers.

The fourth number is the total number of calls to paintComponent() since the program began, which should increase by the requested FPS value each second.

The fifth number is the current FPS, calculated by dividing the total number of calls by the total elapsed time since the program began. The sixth number is an average of the last ten FPS numbers (or less if ten have not yet been calculated).

The reportStats() method, and its associated global variables:

```
private static long MAX_STATS_INTERVAL = 1000L;
  // record stats every 1 second (roughly)

private static int NUM_FPS = 10;
   // number of FPS values stored to get an average

// used for gathering statistics
private long statsInterval = 0L;    // in ms
private long prevStatsTime;
private long totalElapsedTime = 0L;

private long frameCount = 0;
private double fpsStore[];
private long statsCount = 0;
private double averageFPS = 0.0;


private DecimalFormat df = new DecimalFormat("0.##");  // 2 dp
private DecimalFormat timedf = new DecimalFormat("0.####");  //4 dp

private int period;       // period between drawing in ms
            :


private void reportStats()
{
  frameCount++;
  statsInterval += period;

  if (statsInterval >= MAX_STATS_INTERVAL) {
    long timeNow = J3DTimer.getValue();

    long realElapsedTime = timeNow - prevStatsTime;
           // time since last stats collection
    totalElapsedTime += realElapsedTime;

    long sInterval = (long)statsInterval*1000000L;  // ms --> ns
    double timingError =
       ((double)(realElapsedTime - sInterval)) / sInterval * 100.0;

    double actualFPS = 0;     // calculate the latest FPS
    if (totalElapsedTime > 0)
      actualFPS = (((double)frameCount / totalElapsedTime) *
                                                  1000000000L);
    // store the latest FPS
    fpsStore[ (int)statsCount%NUM_FPS ] = actualFPS;
    statsCount = statsCount+1;

    double totalFPS = 0.0;     // total the stored FPSs
    for (int i=0; i < NUM_FPS; i++)
      totalFPS += fpsStore[i];

    if (statsCount < NUM_FPS)  // obtain the average FPS
      averageFPS = totalFPS/statsCount;
```



```
      else
        averageFPS = totalFPS/NUM_FPS;

      System.out.println(
         timedf.format( (double) statsInterval/1000) + " " +
         timedf.format((double) realElapsedTime/1000000000L) + "s " +
         df.format(timingError) + "% " +
         frameCount + "c " +
         df.format(actualFPS) + " " +
         df.format(averageFPS) + " afps"   );

      prevStatsTime = timeNow;
      statsInterval = 0L;   // reset
    }
  }  // end of reportStats()
```

reportStats() is called in paintComponent() after the timer has 'ticked'. This is recognized by incrementing frameCount and adding the period amount to statsInterval.

The FPS values are stored in the fpsStore[] array. When the array is full, new values overwrite the old ones by cycling around the array. The average FPS smooths over variations in the application's execution time.

Table 1 shows the reported average FPSs on different versions of Windows, when the requested FPSs were 20, 50, 80, and 100.

Requested FPS	*20*	*50*	*80*	*100*
Windows 98	18	18	18	18
Windows 2000	19	49	49	98
Windows XP	16	32	64	64

Table 1. Reported Average FPSs for SwingTimerTest.

Each test was run three times on a lightly loaded machine, running for a few minutes. The results show a wide variation in the accuracy of the timer, but the results for the 80 FPS request are poor/awful in all cases. The Swing timer cannot be recommended for high frame rate games.

The timer is designed for repeatedly triggering actions after a fixed period. However, the actual action frequency can drift because of extra delays introduced by the garbage collector or long-running game updates and rendering. It may be possible to code round this by dynamically adjusting the timer's period using setDelay().

The timer uses currentTimeMillis() internally, with its attendant resolution problems.

The official Java tutorial contains more information about the Swing timer and animation, located in the Swing trail in “Performing Animations” (http://java.sun.com/docs/books/tutorial/uiswing/painting/animation.html).

10. The Utility Timer

A timer is also available in the java.util.Timer class. Instead of scheduling calls to actionPerformed(), the run() method of a TimerTask object is invoked.

The utility timer provides more flexibility over scheduling than the Swing timer: tasks can run at a fixed rate, or a fixed period after a previous task. The latter approach is similar to the Swing timer, and means that the timing of the calls can drift.

In fixed-rate scheduling, each task is scheduled relative to the scheduled execution time of the initial task. If a task is delayed for any reason (such as garbage collection), two or more tasks will occur in rapid succession to catch up.

The most important difference between javax.Swing.Timer and java.util.Timer is that the latter does not run its tasks in the event-dispatching thread. Consequently, the test code employs three classes: one for the timer, consisting of little more than a main() function, a subclass of TimerTask for the repeated task, and a subclass of JPanel as a canvas.

These components are shown in Figure 3, which represents the test code in UtilTimerTest.java.

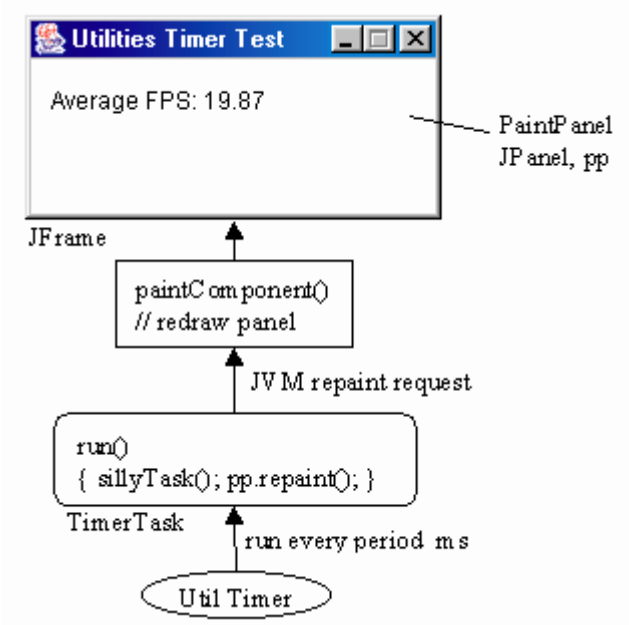

Figure 3. Utility Timer Animation.

The timer schedules the TimerTask at a fixed rate.

```
MyTimerTask task = new MyTimerTask(...);
Timer t = new Timer();
t.scheduleAtFixedRate(task, 0, period);
```

The TimerTask run() method does some time-wasting looping in sillyTask(), and then repaints its JPanel:

```
class MyTimerTask extends TimerTask
{
```



```
    :
  public void run()
  { sillyTask();
    pp.repaint();
  }
    :
} // end of MyTimerTask
```

The JPanel is subclassed to paint the current average FPS value onto the canvas, and call reportStats() to record timing information. Its paintComponent() and reportStats() are the same as in SwingTimerTest.

Table 2 shows the reported average FPSs on different versions of Windows, when the requested FPSs are 20, 50, 80, and 100.

Requested FPS	*20*	*50*	*80*	*100*
Windows 98	20	47	81	94
Windows 2000	20	50	83	99
Windows XP	20	50	83	95

Table 2. Reported Average FPSs for UtilTimerTest.

The average FPSs are excellent, which is somewhat surprising since currentTimeMillis() is employed in the timer's scheduler. The average hides the fact that it takes 1-2 minutes for the frame rate to rise towards the average. Also, JVM garbage collection reduces the FPS for a few seconds each time it occurs.

The average FPS for a requested 80 FPS is often near to 83 due to a quirk of my coding. The frame rate is converted to an *integer* period using (int) 1000/80 == 12 ms. Later this is converted back to a frame rate of 1000/12 == 83.333.

The drawback of the utility timer is that the details of the timer and sleeping operations are mostly out of reach of the programmer, and so not easily modified, unlike the threaded animation loop.

The Java tutorial contains information about the utility timer and TimerTasks in the threads trail under the heading “Using the Timer and TimerTask Classes” (http://java.sun.com/docs/books/tutorial/essential/threads/timer.html).

11. What’s Next?

The threaded animation loop developed in this chapter will be used throughout the rest of the 2D chapters. Chapters 2 and 3 develop a WormChase game in applet, windowed application, and full-screen forms, in order to test whether a frame rate of 80-85 FPS is possible with this approach.