# Chapter 2. Worms in Windows and Applets

In this chapter, we test the threaded animation loop of chapter 1 inside a windowed application and an applet. To simplify comparisons between the approaches, the programs are all variants of the same WormChase game.

In the next chapter, we continue the comparisons, concentrating on several kinds of full-screen applications.

Figure 1 shows the windowed WormChase application on the left, the applet version on the right.

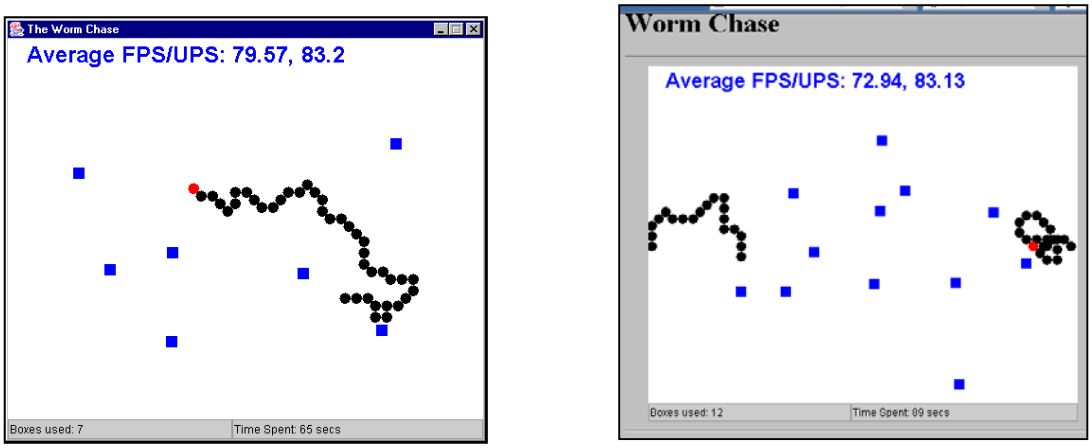


Figure 1. WormChase JFrame and JApplet.

The aim of the game is to click the cursor on the red head of the rapidly moving worm. If the player misses the worm's head then a blue box is added to the canvas (unless the worm's black body was clicked upon).

The worm must go around the boxes in its path, so they *may* make the worm easier to catch. When the worm moves off the top edge of the window it appears at the bottom, and vice versa. When it travels past the left/right edge, it appears at the opposite side.

The worm gradually gets longer until it reaches a maximum length which it keeps from then on.

A score is displayed in the center of the window at the end, calculated from the number of boxes used and the time taken to catch the worm. Less boxes and less time will produce a higher score.

The current time and the number of boxes are displayed below the game canvas in two text fields.

The main drawback of the animation loop of chapter 1 is the need to install Java 3D so that its timer is available. Consequently, two versions of the windowed WormChase application are investigated here, one using the Java 3D timer, the other using the System timer. A comparison of the two will show when the Java 3D timer is beneficial.


© Andrew Davison 2004

All the WormChase versions in this chapter, and the next, use the same game-specific classes (i.e., Worm and Obstacles; see below). They also employ a very similar WormPanel class, which corresponds to the GamePanel animation class of chapter 1.

The main differences between the programs lie in their top-level classes. For example, in this chapter, the windowed applications uses a subclass of JFrame while the applet utilizes JApplet. This requires changes to how game pausing and resumption are triggered, and the way of specifying the required FPS.

Testing is done via the gathering of statistics using a version of the reportStats() method detailed in section 9 of chapter 1. The main change is that an average UPS is calculated alongside the average FPS.

The overall aim of the testing is to see if the animation loop can deliver 80-85 FPS. Failing this, the programs should produce 80-85 updates/second without an excessive number of frames being skipped.

## 1. UML Diagrams for the WormChase Application

Figure 2 shows the UML diagrams for all the classes in the WormChase application. The class names and public methods are shown.

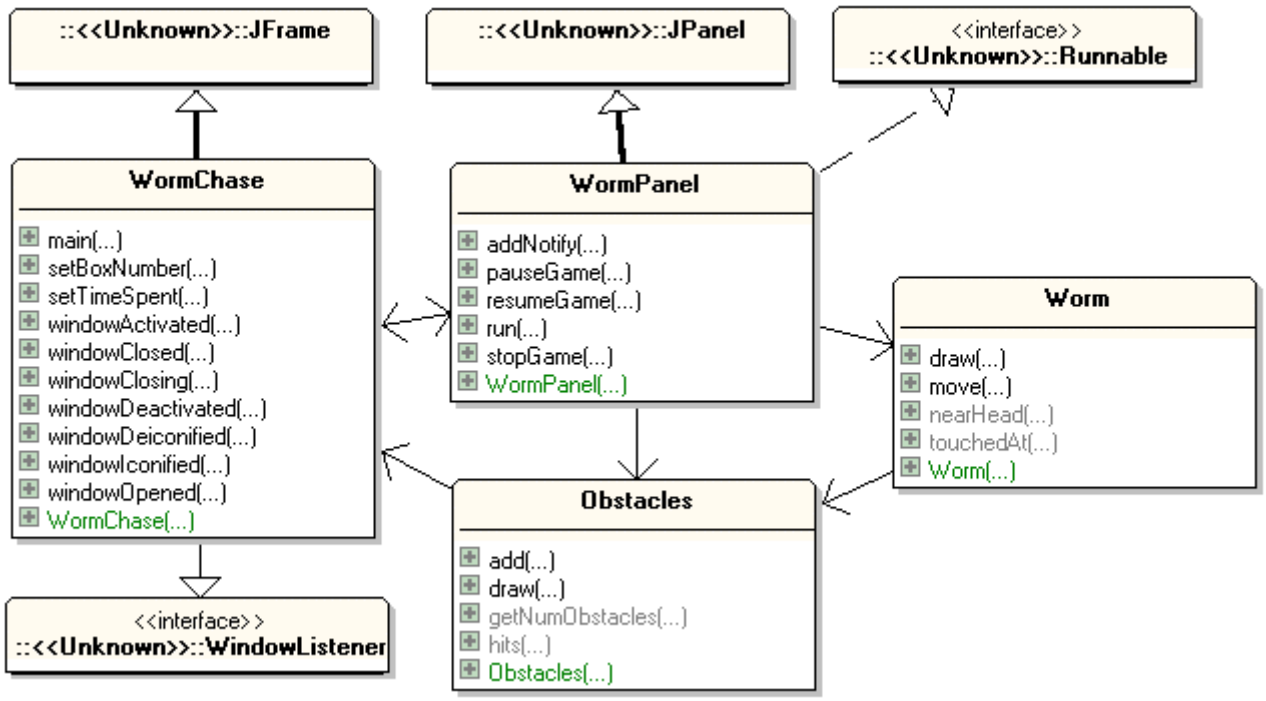


Figure 2. UML Class Diagrams for the WormChase Application.

The code for this version of WormChase is in the subdirectory /WormP in the chapter 2 examples.

WormChase is the top-level JFrame, managing the GUI, and processing window events. WormPanel is the game panel holding the threaded animation loop.

 © Andrew Davison 2004

The Worm class maintains the data structures and methods for the on-screen worm. The Obstacles class handles the blue boxes. Both Worm and Obstacles have their own draw() method which is called by WormPanel to render the worm and boxes.

## 2. The WormChase Class

The main() function in WormChase reads the requested FPS from the command line, converting it to a delay in *nanoseconds*, which is past to the WormChase() constructor.

```
public static void main(String args[])
{
  int fps = DEFAULT_FPS;
  if (args.length != 0)
    fps = Integer.parseInt(args[0]);

  long period = (long) 1000.0/fps;
  System.out.println("fps: " + fps + "; period: " +period+ " ms");

  new WormChase(period*1000000L);    // ms --> nanosecs
}
```

The WormChase() constructor creates the WormPanel canvas, and two textfields for displaying the number of boxes added to the scene (jtfBox) and the current time (jtfTime). These text fields can be updated via two public methods:

```
public void setBoxNumber(int no)
{  jtfBox.setText("Boxes used: " + no);  }

public void setTimeSpent(long t)
{  jtfTime.setText("Time Spent: " + t + " secs"); }
```

setBoxNumber() is called from the Obstacles object when a new box (obstacle) is created. setTimeSpent() is called from WormPanel.

The pausing, resumption, and termination of the game is managed through window listener methods (WormChase implements WindowListener). Pausing is triggered by window deactivation or iconification, the application resumes when the window is activated or deiconified, and the clicking of the window close box causes termination.

```
public void windowActivated(WindowEvent e)
{ wp.resumeGame();  }

public void windowDeactivated(WindowEvent e)
{  wp.pauseGame();  }

public void windowDeiconified(WindowEvent e)
{  wp.resumeGame();  }

public void windowIconified(WindowEvent e)
{  wp.pauseGame(); }
```

 **© Andrew Davison 2004**

```
public void windowClosing(WindowEvent e)
{  wp.stopGame();  }
```

wp refers to the WormPanel object.

## 3. The WormPanel Class

The WormPanel class is very similar to the GamePanel class developed in chapter 1, with some additional methods for drawing the game scene.

WormPanel contains an extended version of the reportStats() method used for timing the Swing and utility timers in chapter 1. The principal extension is to report an average UPS (updates/second) in addition to the average FPS.

The WormPanel() constructor sets up the game components and initializes timing elements:

```
public WormPanel(WormChase wc, long period)
{
  wcTop = wc;
  this.period = period;

  setBackground(Color.white);
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  setFocusable(true);
  requestFocus();    // now has focus, so receives key events
  readyForTermination();

  // create game components
  obs = new Obstacles(wcTop);
  fred = new Worm(PWIDTH, PHEIGHT, obs);

  addMouseListener( new MouseAdapter() {
    public void mousePressed(MouseEvent e)
    { testPress(e.getX(), e.getY()); }
  });

  // set up message font
  font = new Font("SansSerif", Font.BOLD, 24);
  metrics = this.getFontMetrics(font);

  // initialise timing elements
  fpsStore = new double[NUM_FPS];
  upsStore = new double[NUM_FPS];
  for (int i=0; i < NUM_FPS; i++) {
    fpsStore[i] = 0.0;
    upsStore[i] = 0.0;
  }
}  // end of WormPanel()
```

The time period intended for each frame (in nanoseconds) is passed to WormPanel from WormChase and stored in a global variable.

 © Andrew Davison 2004

readyForTermination() is the same as in chapter 1: a KeyListener monitors the input for termination characters (e.g. <ctrl>-C), then sets the running boolean to false.

The message font is used to report the score when the game ends, as shown in Figure 4 below.

fpsStore[] and upsStore[] are global arrays holding the previous ten FPS and UPS values calculated by the statistics code.

### 3.1. User Input

The testPress() method handles mouse presses on the canvas, which will be aimed at the worm's red head. If the press is sufficiently near to the head, then the game is won. If the press touches the worm's body (the black circles) then nothing occurs, otherwise an obstacle is added to the scene at that (x,y) location.

```
private void testPress(int x, int y)
// is (x,y) near the head or should an obstacle be added?
{
  if (!isPaused && !gameOver) {
    if (fred.nearHead(x,y)) {   // was mouse press near the head?
      gameOver = true;
      score = (40 - timeSpentInGame) + 40 - obs.getNumObstacles());
          // hack together a score
    }
    else {   // add an obstacle if possible
      if (!fred.touchedAt(x,y))   // was worm's body not touched?
        obs.add(x,y);
    }
  }
}  // end of testPress()
```

testPress() starts by testing isPaused and gameOver. If isPaused is true then the game is paused and mouse presses should be ignored. Similarly, if the game is already over (gameOver == true) then the input is disregarded.

WormChase's WindowListener methods respond to window events by calling the following methods in WormPanel to affect the isPaused and running flags:

```
public void resumeGame()
// called when the JFrame is activated / deiconified
{  isPaused = false;  }

public void pauseGame()
// called when the JFrame is deactivated / iconified
{ isPaused = true;   }

public void stopGame()
// called when the JFrame is closing
{  running = false;   }
```

As discussed in chapter 1, pausing and resumption do not utilize the Thread wait() and notify() methods to affect the animation thread.

 © Andrew Davison 2004

### 3.2. The Animation Loop

For the sake of completeness, we include the run() method from WormPanel. The parts of it which differ from the animation loop in section 7.1 of chapter 1 are marked in bold.

```
public void run()
/* The frames of the animation are drawn inside the while loop. */
{
  long beforeTime, afterTime, timeDiff, sleepTime;
  long overSleepTime = 0L;
  int noDelays = 0;
  long excess = 0L;

  gameStartTime = J3DTimer.getValue();
  prevStatsTime = gameStartTime;
  beforeTime = gameStartTime;

  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++;
    }
    framesSkipped += skips;

    storeStats();
```

 © Andrew Davison 2004

```
      }

    printStats();
    System.exit(0);   // so window disappears
  } // end of run()
```

The global variables, gameStartTime and prevStatsTime, are utilized in the statistics calculations, as is the frameSkipped variable. frameSkipped holds the total number of skipped frames since the last UPS calculation in storeStats(). printStats() reports selected numbers and statistics at program termination time.

### 3.3. Statistics Gathering

storeStats() is a close relative of the reportStats() method of section 9 in chapter 1. Again for completeness, we list the method here and the new global variables which it manipulates in addition to the ones described in chapter 1. The parts of reportStats() which are new (or changed) are marked in bold.

```
// used for gathering statistics
    : // many, see section 9, chapter 1
private long gameStartTime;
private int timeSpentInGame = 0;    // in seconds

private long framesSkipped = 0L;
private long totalFramesSkipped = 0L;
private double upsStore[];
private double averageUPS = 0.0;


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

  if (statsInterval >= MAX_STATS_INTERVAL) {
    long timeNow = J3DTimer.getValue();
    timeSpentInGame =
      (int) ((timeNow - gameStartTime)/1000000000L);  // ns-->secs
    wcTop.setTimeSpent( timeSpentInGame );

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

    double timingError = (double)
        (realElapsedTime-statsInterval) / statsInterval)*100.0;

    totalFramesSkipped += framesSkipped;

    double actualFPS = 0;     // calculate the latest FPS and UPS
    double actualUPS = 0;
    if (totalElapsedTime > 0) {
      actualFPS = (((double)frameCount / totalElapsedTime) *
                                              1000000000L);
      actualUPS = (((double)(frameCount + totalFramesSkipped) /
          totalElapsedTime) * 1000000000L);
    }
```

 © Andrew Davison 2004

```
      // store the latest FPS and UPS
      fpsStore[ (int)statsCount%NUM_FPS ] = actualFPS;
      upsStore[ (int)statsCount%NUM_FPS ] = actualUPS;
      statsCount = statsCount+1;

      double totalFPS = 0.0;     // total the stored FPSs and UPSs
      double totalUPS = 0.0;
      for (int i=0; i < NUM_FPS; i++) {
        totalFPS += fpsStore[i];
        totalUPS += upsStore[i];
      }

      if (statsCount < NUM_FPS) { // obtain the average FPS and UPS
        averageFPS = totalFPS/statsCount;
        averageUPS = totalUPS/statsCount;
      }
      else {
        averageFPS = totalFPS/NUM_FPS;
        averageUPS = totalUPS/NUM_FPS;
      }
/*
     System.out.println(
       timedf.format( (double) statsInterval/1000000000L) + " " +
       timedf.format((double) realElapsedTime/1000000000L)+"s "+
       df.format(timingError) + "% " +
       frameCount + "c " +
       framesSkipped + "/" + totalFramesSkipped + " skip; " +
       df.format(actualFPS) + " " + df.format(averageFPS)+" afps; " +
       df.format(actualUPS) + " " + df.format(averageUPS)+" aups" );
*/
      framesSkipped = 0;
      prevStatsTime = timeNow;
      statsInterval = 0L;   // reset
    }
  }  // end of storeStats()
```

gameStartTime is used to calculate timeSpentInGame, which WormPanel reports to the player by writing to the time textfield in the top-level window.

Just as in chapter 1, the statsInterval value is a sum of the requested periods adding up to `MAX_STATS_INTERVAL`. The difference is that the period is measured in nanoseconds here (due to the use of the Java 3D timer). This means that the timingError calculation does not need to translate the statsInterval value from milliseconds to nanoseconds before using it.

The main additions to storeStats() are the calculation of a UPS value, its storage in the upsStore[] array, and the use of that array to calculate an average UPS. The UPS value comes from the statements:

```
        totalFramesSkipped += framesSkipped;

        actualUPS = (((double)(frameCount + totalFramesSkipped) /
            totalElapsedTime) * 1000000000L);
```

frameCount is the total number of rendered frames in the game so far, which is added to the total number of skipped frames. (A skipped frame is a game state update which was not rendered.) The total is equivalent to the total number of game updates. The

 **© Andrew Davison 2004**

division by the total elapsed time and multiplication by 1,000,000,000 gives the updates per second.

The large println() call in storeStats() produces a line of statistics. It is commented out since it is intended for debugging purposes. Figure 3 shows its output.

```
C>java WormChase 80
fps: 80; period: 12 ms
1.008 1.1392s 13.01% 84c 10/10 skip; 73.74 73.74 afps; 82.52 82.52 aups
1.008 1.0257s 1.75% 168c 2/12 skip; 77.6 75.67 afps; 83.15 82.83 aups
1.008 1.0237s 1.56% 252c 1/13 skip; 79.03 76.79 afps; 83.11 82.92 aups
1.008 1.0637s 5.53% 336c 5/18 skip; 79.02 77.35 afps; 83.25 83.01 aups
1.008 1.0267s 1.86% 420c 1/19 skip; 79.56 77.79 afps; 83.16 83.04 aups
1.008 1.0231s 1.5% 504c 1/20 skip; 79.97 78.15 afps; 83.15 83.06 aups
1.008 1.1629s 15.36% 588c 10/30 skip; 78.77 78.24 afps; 82.79 83.02 aups
Frame Count/Loss: 588 / 30
Average FPS: 78.24
Average UPS: 83.02
Time Spent: 7 secs
Boxes used: 0

C>
```

Figure 3. storeStats() and Other Output.

Each statistics line presents ten 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 run() since the program began, which should increase by the FPS value each second.

The fifth and sixth numbers (separated by a '/') are the frames skipped in this interval, and the total number of frames skipped since the game began. A frame skip is a game update without a corresponding render.

The seventh and eighth numbers are the current UPS and average. The ninth and tenth numbers of the current FPS and its average.

The output after the statistics lines comes from printStats(), which is called as run() is finishing. It gives a briefer summary of the game characteristics.

```
private void printStats()
{
  System.out.println("Frame Count/Loss: " + frameCount +
                          " / " + totalFramesSkipped);
  System.out.println("Average FPS: " + df.format(averageFPS));
  System.out.println("Average UPS: " + df.format(averageUPS));
  System.out.println("Time Spent: " + timeSpentInGame + " secs");
  System.out.println("Boxes used: " + obs.getNumObstacles());
}  // end of printStats()
```

 © Andrew Davison 2004

### 3.4. Game-Specific Behaviour

The behavior specific to the WormChase game originates in two method calls at the start of the animation loop:

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

gameUpdate() changes the game state every frame. For WormChase, this consists in requesting that the worm (called fred) moves:

```
private void gameUpdate()
{ if (!isPaused && !gameOver)
    fred.move();
}
```

The details of the move are left to fred, in the usual object-oriented style. No move is requested if the game is paused or has finished.

gameRender() draws the game elements (e.g. the worm and obstacles) to an image acting as a buffer.

```
private void gameRender()
{
  if (dbImage == null){
    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);

  dbg.setColor(Color.blue);
  dbg.setFont(font);

  // report average FPS and UPS at top left
  dbg.drawString("Average FPS/UPS: " + df.format(averageFPS) +
           ", " + df.format(averageUPS), 20, 25);

  dbg.setColor(Color.black);

  // draw game elements: the obstacles and the worm
  obs.draw(dbg);
  fred.draw(dbg);

  if (gameOver)
```

 © Andrew Davison 2004

```
      gameOverMessage(dbg);
  }  // end of gameRender()
```

gameRender() begins in the way described in chapter 1: the first call to the method causes the image and its graphics context to be created.

A useful convenience for testing is to draw the average FPS and UPS values on the canvas; these operations would normally be commented out when the coding was completed.

The actual game elements are drawn by passing draw requests onto the worm and the obstacles objects:

```
obs.draw(dbg);
fred.draw(dbg);
```

This approach relieves the game panel of drawing work, and moves the drawing activity to the object responsible for the game component's behaviour.

The gameOverMessage() method uses font metrics and the length of the message to place it in the center of the drawing area. Typical output is shown in Figure 4.

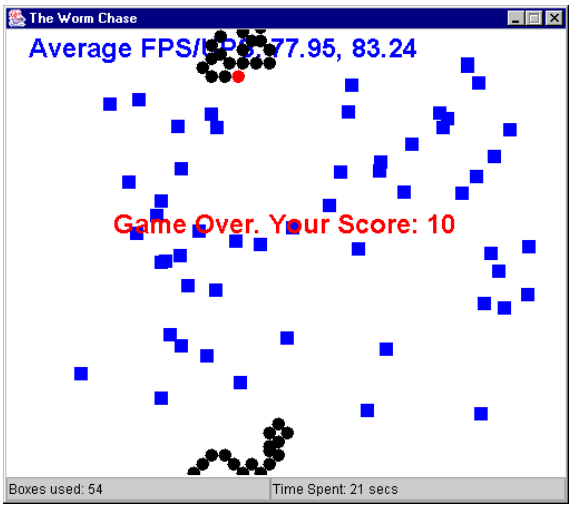


Figure 4. Game Over Message.

As the number of obstacles indicates, a frame rate of 80 FPS makes it very difficult for the player to 'hit' the worm.

paintScreen() actively renders the buffer image to the JPanel canvas, and is unchanged from section 4, chapter 1:

```
private void paintScreen()
// use active rendering to put the buffered image on-screen
{
  Graphics g;
  try {
    g = this.getGraphics();
    if ((g != null) && (dbImage != null))
      g.drawImage(dbImage, 0, 0, null);
```

 © Andrew Davison 2004

```
      g.dispose();
    }
    catch (Exception e)
    { System.out.println("Graphics context error: " + e);  }
  } // end of paintScreen()
```

## 4. The Worm Class

The Worm class stores coordinate information about the worm in a circular buffer. It includes testing methods for checking if the player has clicked near the worm's head or body, and methods for moving and drawing the worm.

The issues which make things more complicated include:

- having the worm grow in length up to a maximum size;
- regulating the worm's movements to be *semi-random* so that it mostly moves in a forward direction;
- getting the worm to go around obstacles in its path.

### 4.1. Growing a Worm

The worm is grown by storing a series of Point objects in a cells[] array. Each point represents the location of one of the black circles of the worm's body (and the red circle for its head). As the worm grows, more points are added to the array until it is full: the worm's maximum extent is equivalent to the array's size.

Movement of the full-size worm is achieved by creating a new head circle at its front, and removing the tail circle (if necessary). This removal frees up a space in the cells[] array where the point for the new head can be stored.

 **© Andrew Davison 2004**

The growing and movement phases are illustrated by Figure 5 which shows how the cells[] array is gradually filled, and reused. The two indices, headPosn and tailPosn, make it simple to modify the head and tail of the worm, and nPoints records the length of the worm.

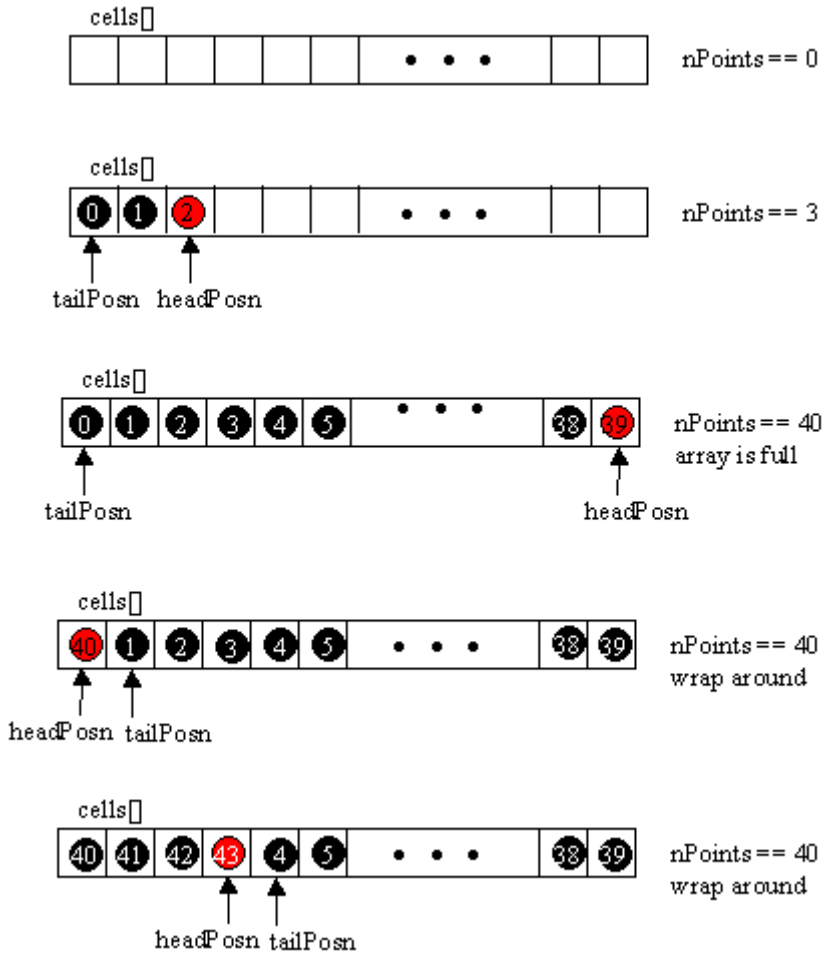


Figure 5. Worm Data Structures during Growth and Movement.

The numbered black dots (and red dot) represent the Point objects which store the (x,y) coordinates of the worm's parts. The numbers are included in the figure to indicate the order in which the array is filled and over-written; they are not part of the actual data structure, which is defined like so:

```
private static final int MAXPOINTS = 40;

private Point cells[];
private int nPoints;
private int tailPosn, headPosn;   // tail and head of buffer
          :

cells = new Point[MAXPOINTS];   // initialise buffer
nPoints = 0;
headPosn = -1;  tailPosn = -1;
```

 © Andrew Davison 2004

The other important Worm data structure is its current bearing, which can be in one of eight predefined compass directions: N = north, NE = north east, and so on, around to NW = north west. The choices are shown in Figure 6.

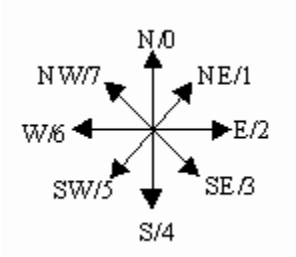


Figure 6. Compass Directions and Integers.

Each compass direction is represented by an integer, which label the bearings in clockwise order. The relevant constants and variable are:

```
// compass direction/bearing constants
private static final int NUM_DIRS = 8;
private static final int N = 0;  // north, etc going clockwise
private static final int NE = 1;
private static final int E = 2;
private static final int SE = 3;
private static final int S = 4;
private static final int SW = 5;
private static final int W = 6;
private static final int NW = 7;

private int currCompass;  // the current compass dir/bearing
```

Limiting the possible directions that a worm can move in, allows the movement steps to be predefined. This reduces the computation at run time, speeding up the worm.

When a new head is made for the worm, it is positioned in one of the eight compass directions, offset by one 'unit' from the current head. This is illustrated in Figure 7.

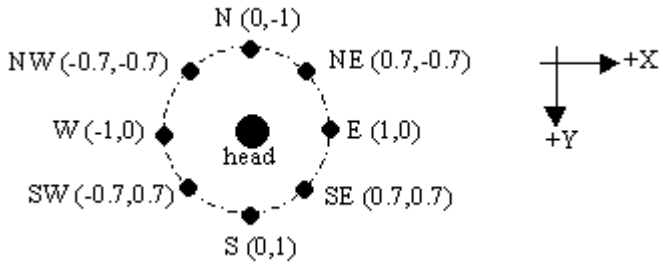


Figure 7. Offsets from the Current Head Position.

The offsets are defined as Point2D.Double objects (a kind of Point class that can hold doubles). They are stored in an incrs[] array, created at Worm construction time:

```
Point2D.Double incrs[];
     :

incrs = new Point2D.Double[NUM_DIRS];
```

 © Andrew Davison 2004

```
    incrs[N] = new Point2D.Double(0.0, -1.0);
    incrs[NE] = new Point2D.Double(0.7, -0.7);
    incrs[E] = new Point2D.Double(1.0, 0.0);
    incrs[SE] = new Point2D.Double(0.7, 0.7);
    incrs[S] = new Point2D.Double(0.0, 1.0);
    incrs[SW] = new Point2D.Double(-0.7, 0.7);
    incrs[W] = new Point2D.Double(-1.0, 0.0);
    incrs[NW] = new Point2D.Double(-0.7, -0.7);
```

### 4.2. Calculating a New Head Point

nextPoint() employs the index position in cells[] of the current head (called prevPosn) and the chosen bearing (e.g., N, SE) to calculate a Point for the new head.

The method is complicated by the need to deal with wraparound positioning top-to-bottom and left-to-right. For example, if the new head is placed off the top of the canvas, it should be repositioned to just above the bottom.

```
private Point nextPoint(int prevPosn, int bearing)
{
  // get the increment for the compass bearing
  Point2D.Double incr = incrs[bearing];

  int newX = cells[prevPosn].x + (int)(DOTSIZE * incr.x);
  int newY = cells[prevPosn].y + (int)(DOTSIZE * incr.y);

  // modify newX/newY if < 0, or > pWidth/pHeight; use wraparound
  if (newX+DOTSIZE < 0)     // is circle off left edge of canvas?
    newX = newX + pWidth;
  else  if (newX > pWidth)  // is circle off right edge of canvas?
    newX = newX - pWidth;

  if (newY+DOTSIZE < 0)     // is circle off top of canvas?
    newY = newY + pHeight;
  else  if (newY > pHeight) // is circle off bottom of canvas?
    newY = newY - pHeight;
  return new Point(newX,newY);
}  // end of nextPoint()
```

The code uses the constant `DOTSIZE` (12), which is the pixel length and height of the circle representing a part of the worm. The new coordinate (newX,newY) is obtained by looking up the offset in incr[] for the given bearing, and adding it to the current head position.

 **© Andrew Davison 2004**

Each circle is defined by its (x,y) coordinate and its `DOTSIZE` length. The (x,y) value is not the centre of the circle, but its top-left corner, as used in drawing operations such as fillOval() (see Figure 8).

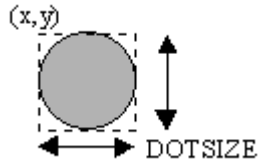


Figure 8. The Coordinates of a Worm Circle.

This explains the wraparound calculations which check if the circle is positioned off the left, right, top or bottom edges of the canvas. The panel dimensions, pWidth and pHeight, are passed to the Worm object by WormPanel at construction time.

### 4.3. Choosing a Bearing

The compass bearing used in nextPoint() comes from varyBearing():

```
int newBearing = varyBearing();
Point newPt = nextPoint(prevPosn, newBearing);
```

varyBearing() is defined as:

```
private int varyBearing()
// vary the compass bearing semi-randomly
{ int newOffset =
        probsForOffset[ (int)( Math.random()*NUM_PROBS )];
  return calcBearing(newOffset);
}
```

The probsForOffset[] array is randomly accessed and returns a new offset.

```
int[] probsForOffset = new int[NUM_PROBS];
probsForOffset[0] = 0;  probsForOffset[1] = 0;
probsForOffset[2] = 0;  probsForOffset[3] = 1;
probsForOffset[4] = 1;  probsForOffset[5] = 2;
probsForOffset[6] = -1;  probsForOffset[7] = -1;
probsForOffset[8] = -2;
```

The distribution of values in the array means that the new offset is most likely to be 0, which keeps the worm moving in the same direction. Less likely is 1 or –1 which will cause the worm to turn slightly left or right. The least likely is 2 or –2, which triggers a larger turn.

calcBearing() adds the offset to the old compass bearing (stored in currCompass), modulo the compass setting ranges North – North West (0 to 7).

```
private int calcBearing(int offset)
// Use the offset to calculate a new compass bearing based
// on the current compass direction.
```

 **© Andrew Davison 2004**

```
{
  int turn = currCompass + offset;
  // ensure that turn is between N to NW (0 to 7)
  if (turn >= NUM_DIRS)
    turn = turn - NUM_DIRS;
  else if (turn < 0)
    turn = NUM_DIRS + turn;
  return turn;
}  // end of calcBearing()
```

### 4.4. Dealing With Obstacles

newHead() generates a new head using varyBearing() and nextPoint(), and updates the cell[] array and compass setting.

```
private void newHead(int prevPosn)  // not finished yet
{
  int newBearing = varyBearing();
  Point newPt = nextPoint(prevPosn, newBearing );

  // what about obstacles?
  ...

  cells[headPosn] = newPt;     // new head position
  currCompass = newBearing;    // new compass direction
}
```

Unfortunately, as the comment above suggests, this code is insufficient for dealing with obstacles: what will happen when the new head is placed at the same spot as an obstacle?

The new point must be tested against the obstacles to make sure it isn't touching any of them. If it is touching, then a new compass bearing and point must be generated. We try three possible moves: turn left by 90 degrees, turn right be 90 degrees and, failing those, turn round and have the worm go back the way it came.

These moves are defined as offsets in the fixedOffs[] array in newHead():

```
private void newHead(int prevPosn)
{
  int fixedOffs[] = {-2, 2, -4};  // offsets to avoid an obstacle

  int newBearing = varyBearing();
  Point newPt = nextPoint(prevPosn, newBearing );

  if (obs.hits(newPt, DOTSIZE)) {
    for (int i=0; i < fixedOffs.length; i++) {
      newBearing = calcBearing(fixedOffs[i]);
      newPt = nextPoint(prevPosn, newBearing);
      if (!obs.hits(newPt, DOTSIZE))
        break;     // one of the fixed offsets will work
    }
  }
  cells[headPosn] = newPt;     // new head position
  currCompass = newBearing;    // new compass direction
```

 © Andrew Davison 2004

```
}  // end of newHead()
```

Key to this strategy is the assumption that the worm can always turn around. This is possible since the player cannot *easily* add obstacles behind the worm since a box cannot be put on a spot currently occupied by the worm's body.

### 4.5. Moving the Worm

The public method move() initiates the worm's movement, utilizing newHead() to obtain a new head position and compass bearing.

The cells[] array, tailPosn and headPosn indices, and the number of points in cells[] are updated in slightly different ways depending on the current stage in the worm's development. These stages are:

- when the worm is first created;
- when the worm is growing, but the cells[] array is not full;
- when the cells[] array is full, and so the addition of a new head must be balanced by the removal of a tail circle.

```
public void move()
{
  int prevPosn = headPosn;
            // save old head posn while creating new one
  headPosn = (headPosn + 1) % MAXPOINTS;

  if (nPoints == 0) {   // empty array at start
    tailPosn = headPosn;
    currCompass = (int)( Math.random()*NUM_DIRS );  // random dir.
    cells[headPosn] = new Point(pWidth/2, pHeight/2); //center pt
    nPoints++;
  }
  else if (nPoints == MAXPOINTS) {    // array is full
    tailPosn = (tailPosn + 1) % MAXPOINTS;    // forget last tail
    newHead(prevPosn);
  }
  else {     // still room in cells[]
    newHead(prevPosn);
    nPoints++;
  }
}  // end of move()
```

### 4.6. Drawing the Worm

WormPanel calls Worm's draw() method to render the worm into the graphics context g. The rendering starts with the point in cell[tailPosn] and moves through the array until cell[headPosn] is reached. The iteration from the tailPosn position to headPosn may involve jumping from the end of the array back to the start.

```
public void draw(Graphics g)
// draw a black worm with a red head
{
```

 **© Andrew Davison 2004**

```
    if (nPoints > 0) {
      g.setColor(Color.black);
      int i = tailPosn;
      while (i != headPosn) {
        g.fillOval(cells[i].x, cells[i].y, DOTSIZE, DOTSIZE);
        i = (i+1) % MAXPOINTS;
      }
      g.setColor(Color.red);
      g.fillOval( cells[headPosn].x, cells[headPosn].y,
                                          DOTSIZE, DOTSIZE);
    }
  }  // end of draw()
```

### 4.7. Testing the Worm

nearHead() and touchedAt() are boolean methods used by WormPanel. nearHead() decides if a given (x,y) coordinate is near the worm's head, while touchedAt() examines its body.

```
public boolean nearHead(int x, int y)
// is (x,y) near the worm's head?
{ if (nPoints > 0) {
    if( (Math.abs( cells[headPosn].x + RADIUS - x) <= DOTSIZE) &&
        (Math.abs( cells[headPosn].y + RADIUS - y) <= DOTSIZE) )
      return true;
  }
  return false;
} // end of nearHead()


public boolean touchedAt(int x, int y)
// is (x,y) near any part of the worm's body?
{
  int i = tailPosn;
  while (i != headPosn) {
    if( (Math.abs( cells[i].x + RADIUS - x) <= RADIUS) &&
        (Math.abs( cells[i].y + RADIUS - y) <= RADIUS) )
      return true;
    i = (i+1) % MAXPOINTS;
  }
  return false;
}  // end of touchedAt()
```

The `RADIUS` constant is half the `DOTSIZE` value. The test in nearHead() allows the (x,y) coordinate to be within *two* radii of the center of the worm's head; any less makes hitting the head almost impossible at 80+ FPS. touchedAt() only checks for an intersection within a single radius of the center.

The addition of `RADIUS` to the (x,y) coordinate in cells[] offsets it from the top-left corner of the circle (see Figure 8) to its center.

 © Andrew Davison 2004

## 5. The Obstacles Class

The Obstacles object maintains an arraylist of Rectangle objects called boxes. Each object contains the top-left hand coordinate of a box and the length of its square sides.

The public methods in Obstacles are synchronized since it is possible for the event thread of the game to add a box to the obstacles list (via a call to add()) at the same time that the animation thread is examining or drawing the list.

add() is defined as:

```
synchronized public void add(int x, int y)
{
  boxes.add( new Rectangle(x,y, BOX_LENGTH, BOX_LENGTH));
  wcTop.setBoxNumber( boxes.size() );   // report new no. of boxes
}
```

The method updates the boxes textfield at the top-level of the game, by calling setBoxNumber().

WormPanel delegates the task of drawing the obstacles to the Obstacles object itself, by calling draw():

```
synchronized public void draw(Graphics g)
// draw a series of blue boxes
{
  Rectangle box;
  g.setColor(Color.blue);
  for(int i=0; i < boxes.size(); i++) {
    box = (Rectangle) boxes.get(i);
    g.fillRect( box.x, box.y, box.width, box.height);
  }
}  // end of draw()
```

Worm communicates with Obstacles to determine if its new head (a Point object, p) intersects with any of the boxes.

```
synchronized public boolean hits(Point p, int size)
{
  Rectangle r = new Rectangle(p.x, p.y, size, size);
  Rectangle box;
  for(int i=0; i < boxes.size(); i++) {
    box = (Rectangle) boxes.get(i);
    if (box.intersects(r))
      return true;
  }
  return false;
}  // end of hits()
```

 **© Andrew Davison 2004**

## 6. Time for Testing

This version of WormChase is a windowed application, with an animation loop driven by the Java 3D timer. Can it support frame rates of 80-85 FPS?

We also consider average UPS, which gives an indication of the 'speed' of the game.

| *Requested FPS* | *20* | *50* | *80* | *100* |
|---|---|---|---|---|
| ***Windows 98*** | 20/20 | 48/50 | 81/83 | 96/100 |
| ***Windows 2000*** | 20/20 | 43/50 | 59/83 | 58/100 |
| ***Windows XP*** | 20/20 | 50/50 | 83/83 | 100/100 |

Table 1. Reported Average FPS/UPSs for the Windowed WormChase using the Java 3D Timer.

Each test was run three times on a lightly loaded machine, executing for a few minutes.

The numbers are very good for the machines hosting Windows 98 and XP, but the frame rates on the Windows 2000 machine plateau at about 60. This behaviour is probably due to the extreme age of the machine: a Pentium 2 with a paltry 64MB of RAM. On a more modern CPU, the frame rates are similar to the XP row of table 1.

A requested frame rate of 80 is changed to 83.333 inside the program, explaining why the 80's column shows numbers close to 83 in most cases. The frame rate is divided into 1000 using integer division, so that 1000/80 becomes 12. Later this period value is converted back to a frame rate using doubles, so that 1000.0/12 becomes 83.3333.

The Windows 2000 figures show that slow hardware is an issue. The processing power of the machine may not be able to deliver the requested frame rate due to excessive time spent in modifying the game state and rendering. Fortunately, the game play on the Windows 2000 machine does not appear to be slow, since the UPS stays near to the request FPS.

Close to 41% of the frames are skipped ((83-59)/83), meaning that almost every second game update is not rendered. Surprisingly, this is not apparent when playing the game. This shows the great benefit of decoupling game updates from rendering, so that the update rate can out perform a poor frame rate.

 **© Andrew Davison 2004**

## 7. WormChase Using currentTimeMillis()

It is interesting to examine the performance of a version of WormChase using System.currentTimeMillis() rather than the Java 3D timer.

The WormChase class and its associated main() function must be modified to represent the period value in milliseconds rather than nanoseconds.

In WormPanel, the calls to J3DTimer.getValue() in run() and storeStats() must be replaced by System.currentTimeMillis(). The sleep() call in run() no longer needs to change sleepTime to milliseconds:

```
Thread.sleep(sleepTime);  // already in ms
```

storeStats() must also be edited to take account of the millisecond units.

The code for this version of WormChase is in /WormPMillis.

The timing results are given in table 2.

| *Requested FPS* | *20* | *50* | *80* | *100* |
|---|---|---|---|---|
| ***Windows 98*** | 19/20 | 43/50 | 54/83 | 57/100 |
| ***Windows 2000*** | 20/20 | 50/50 | 57/83 | 58/100 |
| ***Windows XP*** | 20/20 | 50/50 | 83/83 | 100/100 |

Table 2. Reported Average FPS/UPSs for the Windowed WormChase using the System Timer.

The Windows 98 row shows the effect of the System timer's poor resolution: it causes the animation loop to sleep too much at the end of each update and render, leading to a reduction in the realized frame rate. However, the updates/second are unaffected, making the game advance quickly.

The Windows 2000 row illustrates the slowness of the host machine. The figures are comparable to the version of WormChase using the Java 3D timer. The Windows XP row shows that the System timer's performance is essentially equivalent to the Java 3D timer. The System timer's resolution on Windows 2000 and XP is 10-15 milliseconds (67 –100 FPS)

 **© Andrew Davison 2004**

## 8. WormChase as an Applet

Figure 1 at the start of this chapter shows the WormChase game as an applet.

It has the same GUI interface as the windowed version: a large canvas with two textfields at the bottom used to report the number of boxes added to the scene, and the time.

UML class diagrams showing the public methods are given in Figure 9. A comparison with the diagrams for the windowed version in Figure 2 show that the classes stay mainly the same. The only substantial change is to replace JFrame by JApplet at the top-level.

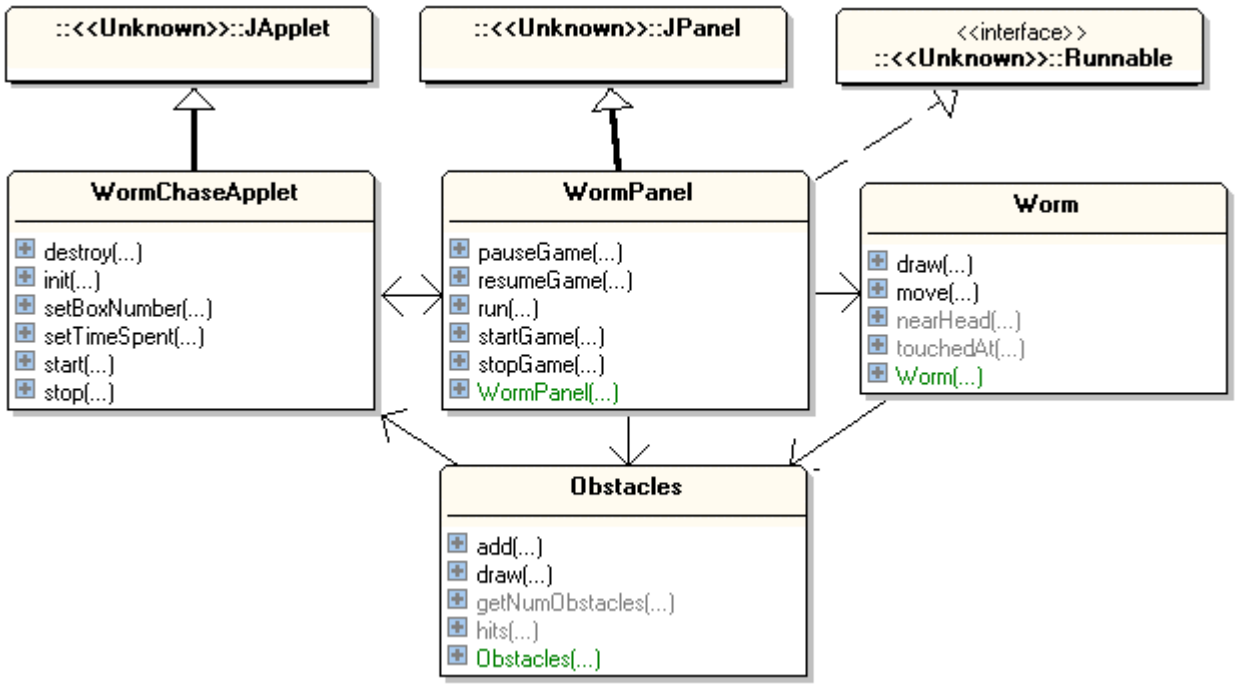


Figure 9. UML Class Diagrams for the WormChase Applet.

The code for this version of WormChase is in the subdirectory /WormApplet in the chapter 2 examples.

The Worm class is unchanged from earlier. The Obstacles class now calls setBoxNumber() in WormChaseApplet rather than WormChase.

WormPanel reports its termination statistics in a slightly different way, but the animation loop and statistics gathering is unchanged.

WormChaseApplet handles pausing, resumption, and termination by tying them to events in the applet life cycle (by comparison, WormChase utilizes Window events).

The applet's Web page passes the requested frame rate to it as a parameter:

```
<applet code="WormChaseApplet.class" width="500" height="415">
   <param name="fps" value="80">
</applet>
```

 © Andrew Davison 2004

### 8.1. The WormChaseApplet Class

The applet's init() method reads the FPS value from the Web page, sets up the GUI, and starts the game.

```
public void init()
{
  String str = getParameter("fps");
  int fps = (str != null) ? Integer.parseInt(str) : DEFAULT_FPS;

  long period = (long) 1000.0/fps;
  System.out.println("fps: " + fps + "; period: "+period+" ms");

  makeGUI(period);
  wp.startGame();
}
```

makeGUI() is the same as the one in the JFrame version. The call to startGame() replaces the use of addNotify() in the JPanel.

The applet life-cycle methods, start(), stop(), and destroy() contain calls to WormPanel to resume, pause, and terminate the game.

```
public void start()
{ wp.resumeGame(); }

public void stop()
{ wp.pauseGame(); }

public void destroy()
{ wp.stopGame(); }
```

A browser calls destroy() just prior to deleting the Web page (and its applet), or perhaps as the browser itself is closed. The browser will wait for the destroy() call to return, before exiting.

### 8.2. The WormPanel Class

The only major change to WormPanel is how printStats() is called. The stopGame() method is modified to call finishOff(), which calls printStats():

```
public void stopGame()
{ running = false;
  finishOff();  // new bit, different from the application
}

private void finishOff()
{ if (!finishedOff) {
    finishedOff = true;
    printStats();
  }
} // end of finishedOff()
```

 © Andrew Davison 2004

finishOff() checks a global finishedOff boolean to decide whether to report the statistics. finishedOff starts with the value false.

finishOff() is also called at the end of run() as the animation loop finishes. The first call to finishOff() will pass the if-test, set the flag to true, and print the data. The flag will then prevent a second call from repeating the output. (Actually, there is a slim chance of a *race condition*, with two simultaneous calls to finishOff() getting past the if-test at the same time, but it's not serious or likely, so we ignore it.)

In the windowed application, stopGame(), only sets running to false before returning; there is no call to finishOff(). The threaded animation loop may then execute for a short time before checking the flag, stopping, and calling printStats().

This approach is fine in an application where the animation thread will be allowed to finish before the application terminates. Unfortunately, as soon as an applet's destroy () method returns, then the applet, or the browser itself, can exit. The animation thread may not have time to reach its printStats() call.

To ensure that the statistics are printed, finishOff() is called in the applet's stopGame (). The other call to finishOff() at the end of run() is a catch-all in case we modify the game so that it can terminate the animation loop without passing through stopGame().

### 8.3. Timing Results

The timing results are given in table 3.

| *Requested FPS* | *20* | *50* | *80* | *100* |
|---|---|---|---|---|
| ***Windows 98*** | 20/20 | 50/50 | 82/83 | 97/100 |
| ***Windows 2000*** | 20/20 | 46/50 | 63/83 | 61/100 |
| ***Windows XP*** | 20/20 | 50/50 | 83/83 | 100/100 |

Table 3. Reported Average FPS/UPSs for the Applet Version of WormChase.

The poor showing for the frame rate on the Windows 2000 machine is to be expected, but the applet performs very well on more modern hardware.

 **© Andrew Davison 2004**