Chapter 3. Full-Screen Worms

A popular aim for games is an 'immersive experience', where the player becomes so enthralled with the game that he/she forgets everyday trivia such as eating and visiting the bathroom. One simple way of encouraging immersion is to make the game window the size of the desktop: a full-screen display effectively hides tiresome text editors, spreadsheets, or database applications requiring urgent attention.

We'll look at three approaches to creating full-screen games:

- an almost full-screen JFrame (we'll call this AFS);
- an undecorated full-screen JFrame (UFS);
- full-screen exclusive mode (FSEM).

FSEM is getting a lot of attention since its introduction in J2SE 1.4, because it offers greatly increased frame rates over traditional gaming approaches using repaint events and paintComponent(). However, timing comparisons between AFS, UFS, and FSEM reveal surprisingly little disparity between them, due to our use of the animation loop developed in chapter 1 with its active rendering and high resolution Java 3D timer. I won't be explaining those techniques again, so chapter 1 should be read before you start here.

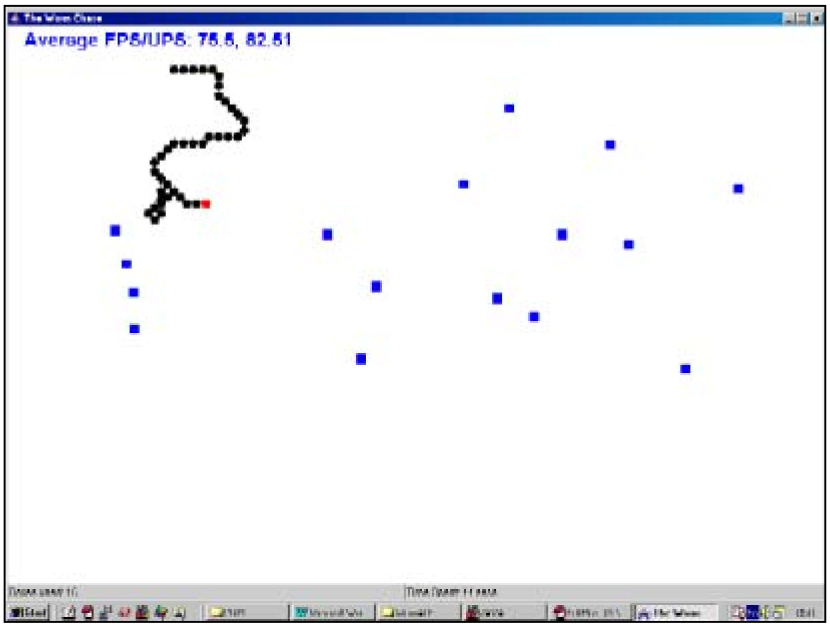
The examples in this chapter will continue using the WormChase game, first introduced in chapter 2 (so you'd better read that chapter as well). By sticking to a single game throughout this chapter, and the last, timing comparisons more accurately reflect differences in the animation frameworks rather than in the game-specific parts.

The objective is to produce frame rates in the range 80-85 FPS, which is near the limit of a typical graphics card's rendering capacity. If the game's frame rate falls short of this, then its updates/second (UPS) should still stay close to 80-85, causing the game to run quickly, but without every update being rendered.

1. An Almost Full-Screen (AFS) Worm

Figure 1 shows the WormChase application running inside a JFrame that almost covers the screen.The JFrame's title bar, including its close box and iconification /deiconfication buttons are visible, and there is a border around the window. The OS desktop controls are visible (in this case MS Windows's task bar at the bottom of the screen).

These JFrame and OS components allow the player to control the game (e.g. pause it by iconification) and to switch to other applications in the usual way, without the need for GUI controls inside the game. Also, very little code has to be modified to change a windowed game into an AFS version, aside from resizing the canvas.



```
Average FPS/UPS: 75.5, 82.51
```

Figure 1. An Almost Full-Screen (AFS) WormChase.

Although the window can be iconified and switched to the background, it cannot be moved. To be more precise, it can be selected and dragged, but as soon as the mouse button is released, the window snaps back to its original position. This is quite a fun effect, as if the window is attached by a rubber band to the top left hand corner of the screen.

Figure 2 gives the UML diagrams for the classes and public methods in the AFS version of WormChase .

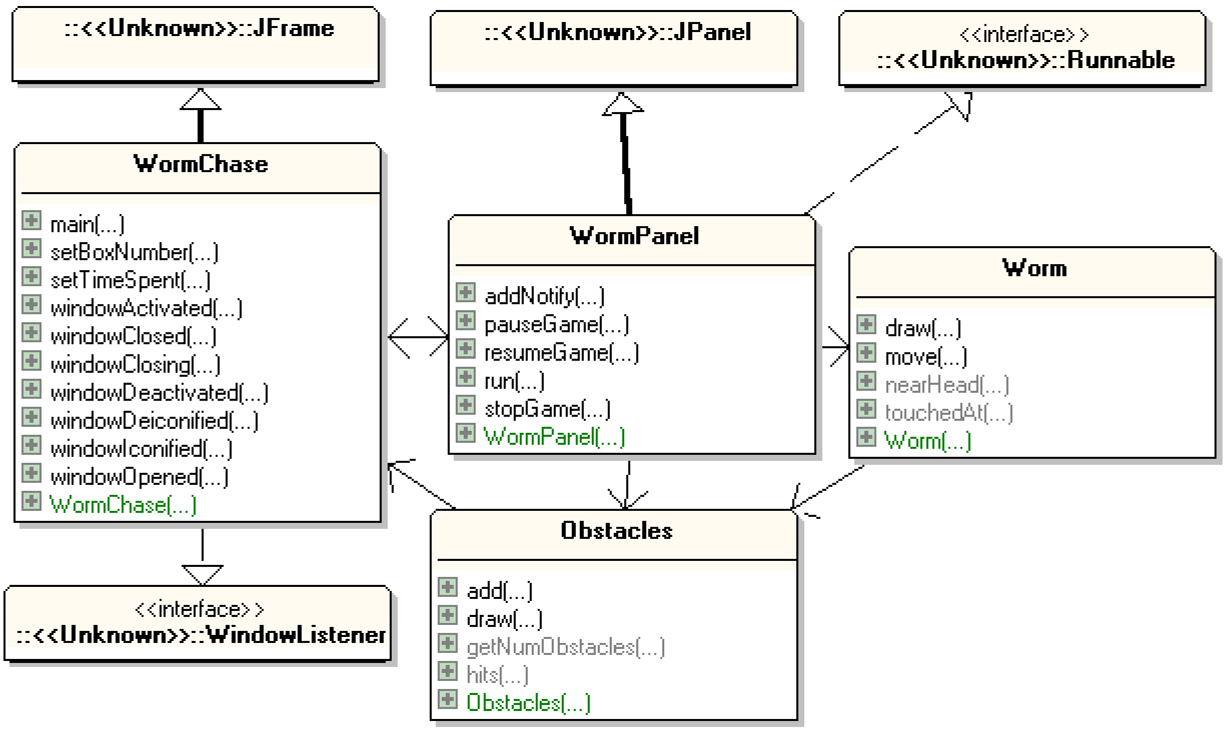


Figure 2. UML Class Diagrams for the AFS Version of WormChase.

The similarity between the AFS approach and the windowed application is highlighted by the fact that the UML diagrams in Figure 2 are identical to those for the windowed application in chapter 2, section 1. The differences are located in private classes and the constructor, where the size of the JFrame is calculated, and listener code is put in place to keep the window from moving.

WormPanel is virtually the same as before, except that WormChase passes it a calculated width and height (in earlier version these were constants in the class).

Worm and Obstacles are unaltered from chapter 2.

The code for the AFS WormChase can be found in /WormAFS in the chapter 3 examples.

1.1. The WormChase Class

The constructor has to work surprisingly hard to obtain correct dimensions for the JPanel. The problem is that the sizes of three distinct kinds of elements must be calculated:

- the JFrame's insets (e.g. the title bar, the borders);
- the desktop's insets (e.g. the taskbar);
- the other Swing components in the window (in this case, two textfields).

The *insets* of a container are the unused areas around its edges (at the top, bottom, left, and right). Typical insets are the container's border lines, and its title bar.

The widths and/or heights of these elements must be subtracted from the screen's dimensions to get WormPanel's width and height.

Figure 3 shows the insets and GUI elements for WormChase.

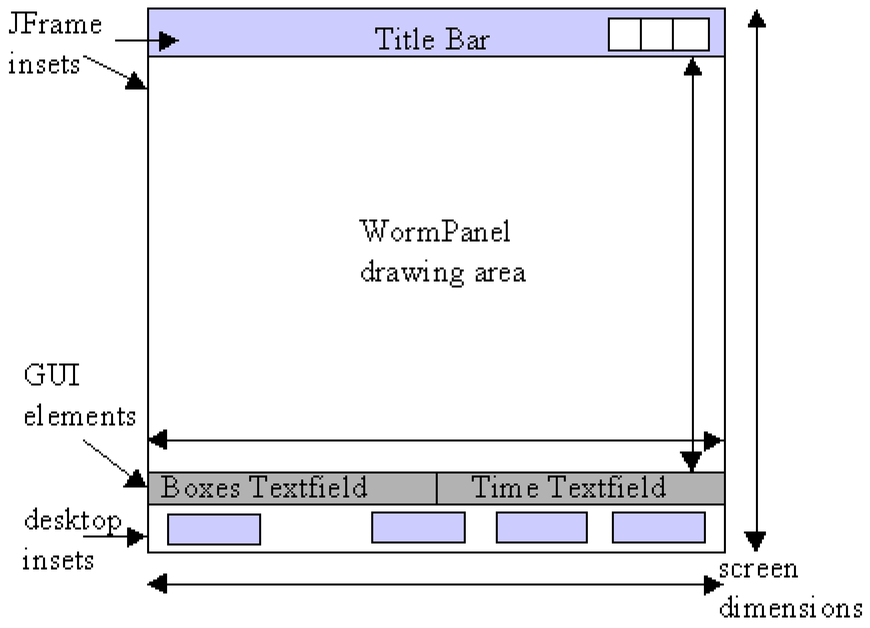


Figure 3. Dimensions in the AFS WormChase.

The subtraction of the desktop and JFrame inset dimensions from the screen size is standard, but the calculation involving the on-screen positions of the GUI elements depends on the game design. For WormChase, only the heights of the textfields affect WormPanel's size.

A subtle problem is that the dimensions of the JFrame insets and GUI elements will be unavailable until the game window has been constructed. In that case, how can the panel's dimensions be calculated if the application has to be created first?

The answer is that the application must be constructed in *stages*: first the JFrame and other pieces needed for the size calculations are put together. This fixes their sizes, so the drawing panel's area can be determined. The sized JPanel is then added to the window to complete it, and the window is made visible.

The WormChase constructor utilizes these stages:

```
public WormChase(long period)
{ super("The Worm Chase");

  makeGUI();

  pack();    // first pack (the GUI doesn't include the JPanel yet)
  setResizable(false);  //so sizes are for non-resizable GUI elems
  calcSizes();
  setResizable(true);    // so panel can be added
```

```
    Container c = getContentPane();
    wp = new WormPanel(this, period, pWidth, pHeight);
    c.add(wp, "Center");
    pack();      // second pack, after JPanel added

    addWindowListener( this );

    addComponentListener( new ComponentAdapter() {
      public void componentMoved(ComponentEvent e)
      {  setLocation(0,0);  }
    });

    setResizable(false);
    show();
  }  // end of WormChase() constructor
```

makeGUI() builds the GUI without a drawing area, and the call to pack() makes the JFrame displayable and calculates the component's sizes. Resizing is turned off since some platforms render insets differently (i.e. with different sizes) when their enclosing window cannot be resized.

calcSizes() initializes two globals, pWidth and pHeight, which are later passed to the WormPanel constructor as the panel's width and height.

```
  private void calcSizes()
  {
    GraphicsConfiguration gc = getGraphicsConfiguration();
    Rectangle screenRect = gc.getBounds();  // screen dimensions

    Toolkit tk = Toolkit.getDefaultToolkit();
    Insets desktopInsets = tk.getScreenInsets(gc);

    Insets frameInsets = getInsets();     // only works after pack()

    Dimension tfDim = jtfBox.getPreferredSize();  // textfield size

    pWidth = screenRect.width
               - (desktopInsets.left + desktopInsets.right)
               - (frameInsets.left + frameInsets.right);

    pHeight = screenRect.height
                - (desktopInsets.top + desktopInsets.bottom)
                - (frameInsets.top + frameInsets.bottom)
                - tfDim.height;
  }
```

If the JFrame's insets (stored in frameInsets) are requested before a call to pack() then they will have zero size.

An Insets object has four public variables, top, bottom, left, and right, which hold the thickness of its container's edges.

Only the dimensions for the boxes textfield (jtfBox) is retrieved, since its height will be the same as the time-used textfield.

Back in WormChase(), resizing is switched back on so that the correctly sized JPanel can be added to the JFrame. Finally, resizing is switched off again (this time permanently), and the application is made visible with a call to show().

1.2. Stopping Window Movement

Unfortunately, there is no simple way of preventing an application's window from being dragged around the screen. The best we can do is move it back to its starting position as soon as the user releases the mouse.

The WormChase constructor sets up a component listener with a componentMoved() handler. This method is called whenever a move is completed.

```
addComponentListener( new ComponentAdapter() {
  public void componentMoved(ComponentEvent e)
  {  setLocation(0,0);  }
});
```

setLocation() positions the JFrame so its top-left corner is at the top-left of the screen.

1.3. Timings for AFS

Timing results for the AFS WormChase are given in Table 1.

Requested FPS	***20***	***50***	***80***	***100***
Windows 98	20/20	49/50	75/83	86/100
Windows 2000	20/20	20/50	20/83	20/100
Windows XP (1)	20/20	50/50	82/83	87/100
Windows XP (2)	20/20	50/50	75/83	75/100

Table 1. Average FPS/UPSs for the AFS WormChase.

WormChase on the slow Windows 2000 machine is the worst performer again, as seen in chapter 2, although its slowness is barely noticeable due to the update rate remaining high.

The Windows 98 and XP boxes produce good frame rates when 80 FPS is requested, close to or inside our desired range (80-85 FPS). The numbers start to flatten out as the FPS request goes higher, indicating that the frames cannot be rendered any faster.

The timing tests for Windows XP were run on two machines, to highlight the variation in WormChase's performance at higher requested FPSs.

2. An Undecorated Full-Screen (UFS) Worm

Figure 4 shows the UFS version of WormChase, a truly full-screen JFrame without a title bar or borders.

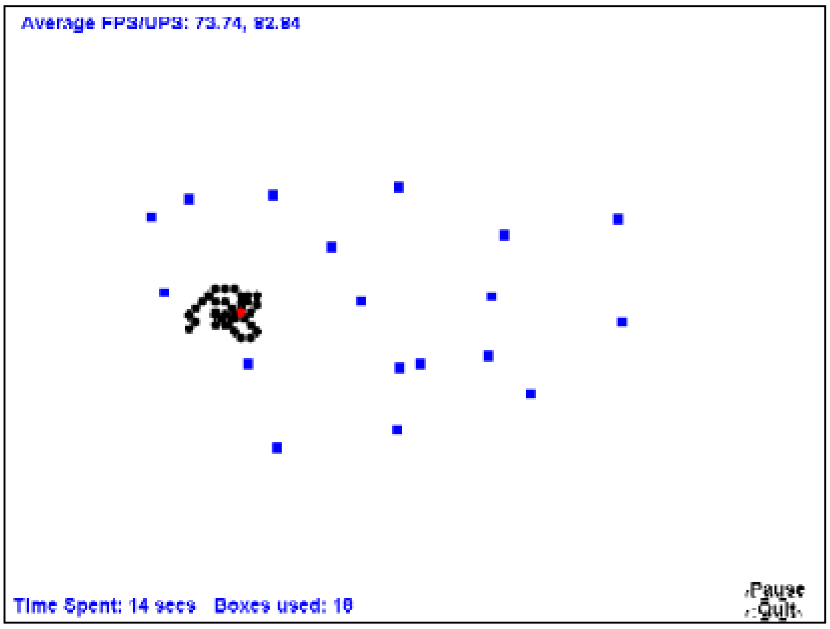


Figure 4. The UFS Worm.

The absence of a title bar means that we must rethink how the application can be paused and resumed (previously done by minimizing/maximizing the window), and how to terminate the game. The solution is to draw pause and quit 'buttons' on the canvas, at the bottom right corner.

Aside from using the quit button, it is still possible to end the game by typing the escape key, ctrl-C , 'q', or the end key.

Data which was previously displayed in textfields is written to the canvas at the lower left corner.

Figure 5 gives the UML class diagrams and public methods for the UFS version of WormChase.

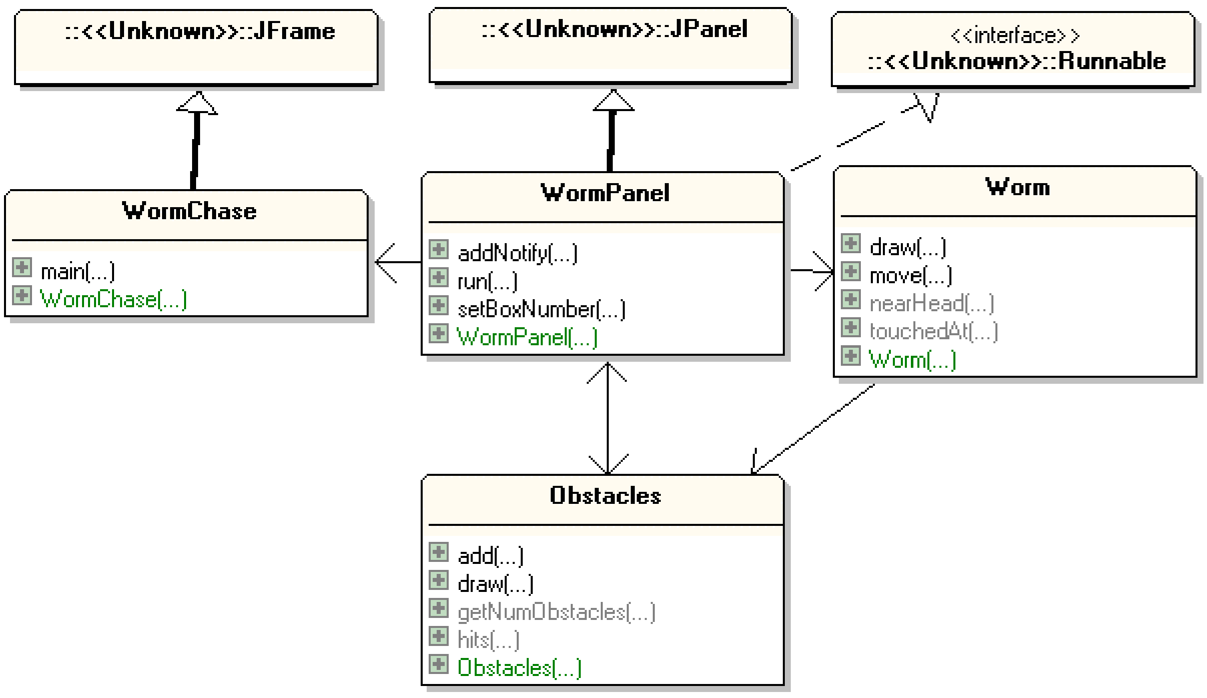


Figure 5. UML Class Diagrams for the UFS Version of WormChase.

A comparison with the AFS diagrams in Figure 2 shows a considerable simplification of WormChase, and fewer methods in WormPanel.

The WormChase class no longer has to be a WindowListener, and so does not include implementations for window handler methods, such as windowClosing(). Those handlers called pauseGame(), resumeGame(), or stopGame() in WormPanel, which are also not required anymore.

The Worm class is unchanged, and the Obstacles class is altered only so that it can call setBoxNumber() in WormPanel; this method was formerly in WormChase to write to a textfield.

The code for the UFS WormChase can be found in /WormUFS in the chapter 3 examples.

2.1. The WormChase Class

With the removal of the WindowListener methods, WormChase hardly does anything. It reads the requested FPS value from the command line, and its constructor creates the WormPanel object.

```
public WormChase(long period)
{ super("The Worm Chase");

  Container c = getContentPane();
  c.setLayout( new BorderLayout() );
```

```
    WormPanel wp = new WormPanel(this, period);
    c.add(wp, "Center");

    setUndecorated(true);   // no borders or title bar
    setIgnoreRepaint(true);
       // turn off paint events since doing active rendering
    pack();
    setResizable(false);
    show();
  }  // end of WormChase() constructor
```

The title bars and other insets are switched off by calling setUndecorated(). setIgnoreRepaint() is utilised since there are no GUI components requiring paint events; WormPanel uses active rendering so doesn't need paint events.

The simplicity of WormChase is an indication that there is no longer any need to employ a separate JPanel as a drawing canvas. It is quite straightforward to move WormPanel's functionality into WormChase, and we explore that approach as part of the FSEM version of WormChase in section 3.

2.2. The WormPanel Class

WormPanel's constructor sets its size to that of the screen, and stores the dimensions in the globals pWidth and pHeight.

```
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension scrDim = tk.getScreenSize();
  setPreferredSize(scrDim);   // set JPanel size

  pWidth = scrDim.width;      // store dimensions for later
  pHeight = scrDim.height;
```

The constructor creates two rectangles, called pauseArea and quitArea, which represent the screen areas for the pause and quit 'buttons'. The drawing of these buttons is left to gameRender(), described below.

```
  private Rectangle pauseArea, quitArea;  // globals

  // in WormPanel()
  // specify screen areas for the buttons
  pauseArea = new Rectangle(pWidth-100, pHeight-45, 70, 15);
  quitArea  = new Rectangle(pWidth-100, pHeight-20, 70, 15);
```

2.3. Button Behaviour

In common with many games, the pause and quit buttons are highlighted when the mouse moves over them. This transition is shown in Figure 6 when the mouse moves over the pause button.

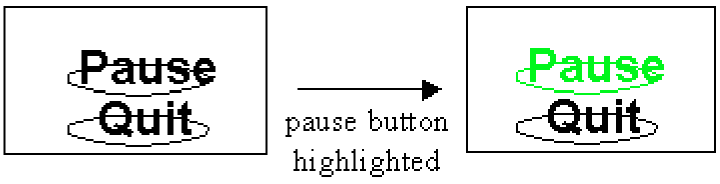


Figure 6. Highlighting the Pause Button.

Another useful kind of feedback is to indicate that the game is paused by changing the wording on the pause button to "Paused", as in Figure 7.

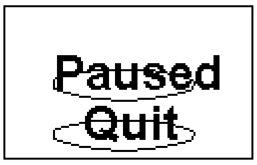


Figure 7. The Pause Button when the Game is Paused.

When the mouse moves over the "Paused" button it will turn green.

The first step to implementing these behaviours is to record when the cursor is inside the pause or quit screen area. This is done by monitoring mouse movements, started in the constructor for WormPanel:

```
addMouseMotionListener( new MouseMotionAdapter() {
  public void mouseMoved(MouseEvent e)
  { testMove(e.getX(), e.getY()); }
});
```

testMove() sets two global booleans (isOverPauseButton, isOverQuitButton) depending on whether the cursor is inside the pause or quit area.

```
private void testMove(int x, int y)
// is (x,y) over the pause or quit button?
{
  if (running) {   // stops problems with a rapid move
                   // after pressing 'quit'
    isOverPauseButton = pauseArea.contains(x,y) ? true : false;
    isOverQuitButton = quitArea.contains(x,y) ? true : false;
  }
}
```

The test of the running boolean prevents button highlight changes after the player has pressed quit but before the application exits.

The other aspect of button behaviour is to deal with a mouse press on top of a button. This is handled by extending testPress(), which previously only dealt with clicks on or near the worm.

```
// in the WormPanel constructor
addMouseListener( new MouseAdapter() {
  public void mousePressed(MouseEvent e)
  { testPress(e.getX(), e.getY()); }
});
   :


private void testPress(int x, int y)
{
  if (isOverPauseButton)
    isPaused = !isPaused;     // toggle pausing
  else if (isOverQuitButton)
    running = false;
  else {
    if (!isPaused && !gameOver) {
      // was mouse pressed on or near the worm?
      . . .
    }
  }
}
```

The highlighted lines in testPress() replace the functionality supported by resumeGame(), pauseGame() and stopGame() in the earlier windowed versions of WormChase.

2.4. Drawing the Game Canvas

The WormPanel canvas contain two elements not present in previous examples:

- the time used and boxes information, drawn at the bottom left corner;
- the pause and quit buttons, drawn at the bottom right.

The buttons are drawn in a different way if the cursor is over them, and the wording on the pause button changes depending on whether the game is actually paused.

These new features are implemented in gameRender():

```
private void gameRender()
{
  // as before: create the image buffer initially
  // set the background to white
  ...

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

  // report time used and boxes used at bottom left
  dbg.drawString("Time Spent: " + timeSpentInGame + " secs",
                                          10, pHeight-15);
  dbg.drawString("Boxes used: " + boxesUsed, 260, pHeight-15);
```

```
    // draw the pause and quit 'buttons'
    drawButtons(dbg);

    dbg.setColor(Color.black);

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

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


  private void drawButtons(Graphics g)
  {
    g.setColor(Color.black);

    // draw the pause 'button'
    if (isOverPauseButton)
      g.setColor(Color.green);

    g.drawOval( pauseArea.x, pauseArea.y,
                        pauseArea.width, pauseArea.height);
    if (isPaused)
      g.drawString("Paused", pauseArea.x, pauseArea.y+10);
    else
      g.drawString("Pause", pauseArea.x+5, pauseArea.y+10);

    if (isOverPauseButton)
      g.setColor(Color.black);

    // draw the quit 'button'
    if (isOverQuitButton)
      g.setColor(Color.green);

    g.drawOval(quitArea.x, quitArea.y,
                      quitArea.width, quitArea.height);
    g.drawString("Quit", quitArea.x+15, quitArea.y+10);

    if (isOverQuitButton)
      g.setColor(Color.black);
  }  // drawButtons()
```

Each button is an oval and a string. Highlighting triggers a change in the foreground colour, using setColor(). Depending on the value of the isPaused boolean, either "Paused" or "Pause" is drawn.

2.5. Exiting the Game

The primary means for terminating the game remains the same: when the running boolean is true then the animation loop will terminate. Before run() returns, the finishOff() method is called.

```
private void finishOff()
{ if (!finishedOff) {
    finishedOff = true;
    printStats();
    System.exit(0);
  }
}
```

The finishedOff boolean is employed to stop a second call to finishOff() printing the statistics information again.

The other way of calling finishOff() is from a shutdown hook (handler) set up when the JPanel is created:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
  public void run()
  { running = false;
    System.out.println("Shutdown hook executed");
    finishOff();
  }
});
```

This code is normally called just before the application exits, and is superfluous since finishOff() will already have been executed. Its real benefit comes if the program terminates unexpectedly. The shutdown hook ensures that the statistics details are still reported.

This kind of defensive programming is often very useful. For example, if the game state *must* be saved to an external file before the program terminates, or if critical resources, such as files or sockets, must be properly closed.

2.6. Timings for UFS

Timing results for the UFS WormChase are given in Table 2.

Requested FPS	*20*	*50*	*80*	*100*
Windows 98	20/20	48/50	70/83	70/100
Windows 2000	18/20	19/50	18/83	18/100
Windows XP (1)	20/20	50/50	77/83	73/100
Windows XP (2)	20/20	50/50	68/83	69/100

Table 2. Average FPS/UPSs for the UFS WormChase.

WormChase on the Windows 2000 machine is the slowest as usual, with marginally slower FPS values than the AFS version (it produces about 20 FPS). However, the poor perfomance is effectively hidden by the high number of updates/second.

The Windows 98 and XP boxes produce reasonable/good frame rates when the requested FPS is 80, but are unable to go much faster. UFS frame rates are about 10% slower than the AFS values at 80 FPS, which may be due to the larger rendering area. UPS figures are unaffected.

3. A Full-Screen Exclusive Mode (FSEM) Worm

Full-screen exclusive mode (FSEM) suspends most of Java's windowing environment, bypassing the Swing and AWT graphics layers to offer almost direct access to the screen. It allows graphics card features, such as page flipping and stereo buffering, to be exploited, and permits the screen's resolution and bit depth to be adjusted.

The graphics hardware acceleration used by FSEM has a disadvantage: it utilizes video memory (VRAM) which may be 'grabbed back' by the OS when, for example, it needs to draw another window, display a screensaver, or change the screen's resolution. The application's image buffer, which is stored in the VRAM, will then have to be reconstructed from scratch. A related issue is that VRAM is not an infinite resource, and trying to place too many images there may cause the OS to start swapping them in and out of memory, causing a slowdown in the rendering.

Aside from FSEM, J2SE 1.4 includes a VolatileImage class to allow images to take advantage of VRAM. Only opaque images and those with transparent areas are accelerated; translucent images are not, at least until J2SE 1.5 arrives. Many forms of image manipulation can cause the acceleration to be lost.

In practice, direct use of VolatileImage is not often required since most graphical applications, such as those written with Swing, attempt to employ hardware acceleration implicitly. For instance, Swing uses VolatileImage for its double buffering, visuals loaded with getImage() are accelerated if possible, as are images used by the Java 2D API (e.g. those built using createImage()). However, more complex rendering features, such as diagonal lines, curved shapes, and anti-aliasing, utilize software rendering at the JVM level.

Another issue with hardware acceleration is that it is principally a MS Windows feature, since DirectDraw is employed by the JVM to access the VRAM. Neither Solaris or Linux provide a way to directly contact VRAM.

There is a Sun tutorial on FSEM (see http://java.sun.com/docs/books/tutorial/extra/fullscreen/), and the rationale behind the VolatileImage class is described in http://java.sun.com/j2se/1.4/pdf/VolatileImage.pdf.

Figure 5 shows a screenshot of the FSEM version of WormChase, which is identical to the UFS interface of section 2, Figure 4.

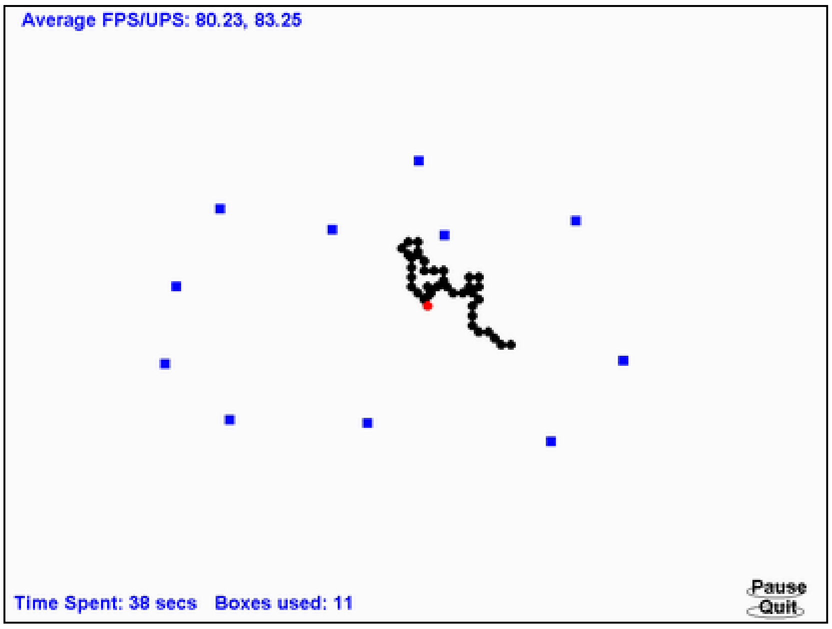


Figure 5. The FSEM WormChase.

The UML class diagrams and public methods for this WormChase are shown in Figure 6.

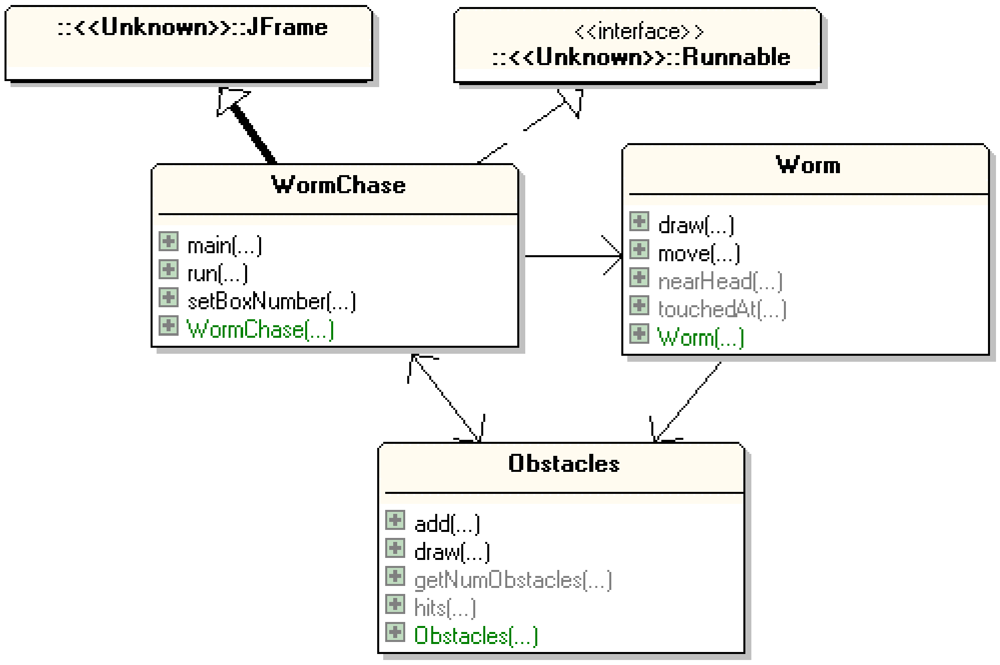


Figure 6. UML Class Diagrams for the FSEM Version of WormChase.

The WormChase and WormPanel classes have been combined into a single WormChase class: it now contains the animation loop, which explains its use of the

Runnable interface. This approach could also be used in the UFS version of WormChase.

The Worm and Obstacles classes are unchanged.

The code for the FSEM WormChase can be found in /WormFSEM in the chapter 3 examples.

3.1. The WormChase Class

The constructor for WormChase is very similar to the constructors for the WormPanel classes of previous sections.

```
public WormChase(long period)
{
  super("Worm Chase");

  this.period = period;
  initFullScreen();   // switch to FSEM

  readyForTermination();

  // create game components
  obs = new Obstacles(this);
  fred = new Worm(pWidth, pHeight, obs);

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

  addMouseMotionListener( new MouseMotionAdapter() {
    public void mouseMoved(MouseEvent e)
    { testMove(e.getX(), e.getY()); }
  });

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

  // specify screen areas for the buttons
  pauseArea = new Rectangle(pWidth-100, pHeight-45, 70, 15);
  quitArea = new Rectangle(pWidth-100, pHeight-20, 70, 15);

  // 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;
  }

  gameStart();  // replaces addNotify()
}  // end of WormChase()
```

WormChase() ends with a call to gameStart(), which contains the code formerly in addNotify(). As you may recall, addNotify() is called automatically as its component (e.g. a JPanel) is added to its container (e.g. a JFrame). Since we are no longer using a JPanel, the game is started from WormChase's constructor.

3.2. Setting up Full-Screen Exclusive Mode

The steps necessary to switch the JFrame to FSEM are contained in initFullScreen().

```
// globals used for FSEM tasks
private GraphicsDevice gd;
private Graphics gScr;
private BufferStrategy bufferStrategy;
     :

private void initFullScreen()
{
  GraphicsEnvironment ge =
     GraphicsEnvironment.getLocalGraphicsEnvironment();
  gd = ge.getDefaultScreenDevice();

  setUndecorated(true);    // no menu bar, borders, etc.
  setIgnoreRepaint(true);
          // turn off paint events since doing active rendering
  setResizable(false);

  if (!gd.isFullScreenSupported()) {
    System.out.println("Full-screen exclusive mode not supported");
    System.exit(0);
  }
  gd.setFullScreenWindow(this); // switch on FSEM

  // we can now adjust the display modes, if we wish
  showCurrentMode();   // show the current display mode

  // setDisplayMode(800, 600, 8);   // or try 16 bits
  // setDisplayMode(1280, 1024, 32);

  reportCapabilities();

  pWidth = getBounds().width;
  pHeight = getBounds().height;

  setBufferStrategy();
}  // end of initFullScreen()
```

The graphics card is accessible via a GraphicsDevice object, gd. It is tested with isFullScreenSupported() to see if FSEM is available. Ideally, if the method returns false, the code should switch to using AFS or UFS, but we simply give up.

Once FSEM has been turned on by calling setFullScreenWindow(), it is possible to modify display parameters, such as screen resolution and bit depth. Details on how this can be done are explained below. In the current version of the program, WormChase only reports the current settings by calling showCurrentMode(); the call to setDisplayMode() is commented out.

initFullScreen() switches off window decoration and resizing, which otherwise tend to interact badly with FSEM. Paint events are also not required since we are continuing to use active rendering, albeit a FSEM version (explained below).

After setting the display characteristics, the width and height of the drawing area are stored in pWidth and pHeight.

Once in FSEM, a buffer strategy for updating the screen is specified by calling setBufferStrategy().

```
private void setBufferStrategy()
{ try {
    EventQueue.invokeAndWait( new Runnable() {
      public void run()
      { createBufferStrategy(NUM_BUFFERS);  }
    });
  }
  catch (Exception e) {
    System.out.println("Error while creating buffer strategy");
    System.exit(0);
  }

  try {  // sleep to give time for buffer strategy to be done
    Thread.sleep(500);  // 0.5 sec
  }
  catch(InterruptedException ex){}

  bufferStrategy = getBufferStrategy();  // store for later
}
```

createBufferStrategy() is called with NUM_BUFFERS (2) so *page flipping* with a primary surface and one back buffer will be utilised. Page flipping is explained in the next section.

invokeAndWait() is employed to avoid a possible deadlock between the createBufferStrategy() call and the event dispatcher thread (this issue should be fixed in J2SE 1.5). The thread holding the createBufferStrategy() call is added to the dispatcher queue, and executed when earlier pending events have been processed. When createBufferStrategy() returns, so will invokeAndWait().

However, createBufferStrategy() is an asynchronous operation, so the sleep() call delays execution a little time so that the getBufferStrategy() call will get the correct details.

The asynchronous nature of many of the FSEM methods is a weakness of the API, since it makes it difficult to know when operations have been completed. Adding arbitrary sleep() calls is inelegant, and may slow down execution unnecessarily. Other asynchronous methods include setDisplayMode(), show(), and setFullScreenWindow ().

3.3. Double Buffering, Page Flipping, and More

All of our earlier versions of WormChase have drawn to an off-screen buffer (sometimes called a *back buffer*), which is copied to the screen by a call to drawImage (). The idea is illustrated in Figure 7.

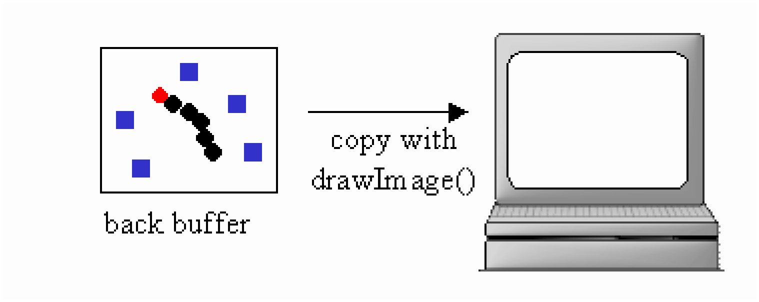


Figure 7. Double Buffering Rendering.

The amount of copying required to display even a single frame is substantial. For example, a display of 1024 by 768 pixels, with 32 bit depth, will need a 3MB sized copy (1024*768*4 bytes), perhaps occuring 80 times per second! This is the principal reason for modifying the display mode: switching to 800 by 600 pixels and 16 bits reduces the copy size to about 940K (800*600*2).

Page flipping avoids these overheads by using a video pointer (if one is available). The video pointer tells the graphics card where to look in VRAM for the image to be displayed during the next refresh. Page flipping involves two buffers, which are used alternatively as the primary surface for the screen. While the video pointer is pointing at one buffer, the other is updated. When the next refresh cycle comes around, the pointer is changed to refer to the second buffer, and the first is updated.

This approach is illustrated by Figures 8 and 9.

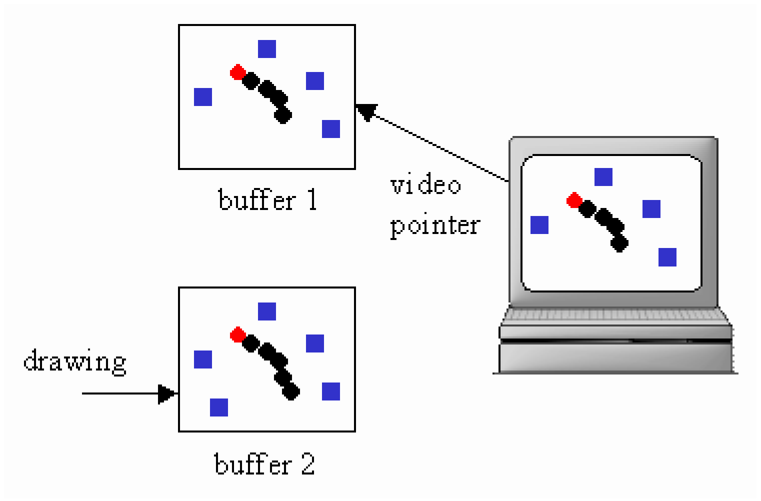


Figure 8. Page Flipping (1). Point to Buffer 1, Update Buffer 2.

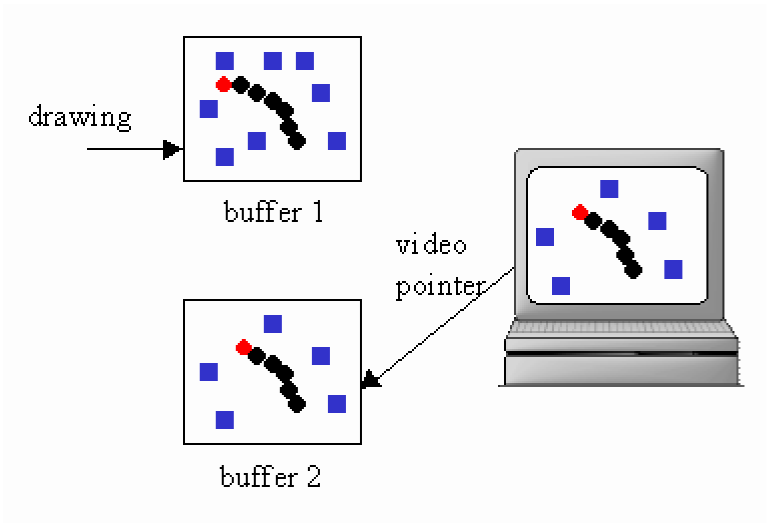


Figure 9. Page Flipping (2). Update Buffer 1, Point to Buffer 2.

The great advantage of this technique is that only pointer manipulation is required, with no need for copying.

The next step is to use more than two buffers, creating a *flip chain*. The video pointer cycles through the buffers while rendering is carried out to the other buffers in the chain.

In initFullScreen(), createBufferStrategy() sets up the buffering for the window, based on the number specified (which should be 2 or more). A page flipping strategy with a video pointer is attempted first, then copying using hardware acceleration. If these are unavailable, an unaccelerated copying strategy is used.

3.4. Information on Graphics Capabilities

initFullScreen() calls reportCapabilities() which illustrates some of the ways that the graphics card's capabilities can be examined.

```
private void reportCapabilities()
{
  GraphicsConfiguration gc = gd.getDefaultConfiguration();

  // image capabilities
  ImageCapabilities imageCaps = gc.getImageCapabilities();
  System.out.println("Image Caps. isAccelerated: " +
                              imageCaps.isAccelerated() );
  System.out.println("Image Caps. isTrueVolatile: " +
                              imageCaps.isTrueVolatile());

  // buffer capabilities
  BufferCapabilities bufferCaps = gc.getBufferCapabilities();
  System.out.println("Buffer Caps. isPageFlipping: " +
                               bufferCaps.isPageFlipping());
  System.out.println("Buffer Caps. Flip Contents: " +
                getFlipText( bufferCaps.getFlipContents() ));
  System.out.println("Buffer Caps. Full-screen Required: " +
                bufferCaps.isFullScreenRequired());
  System.out.println("Buffer Caps. MultiBuffers: " +
                bufferCaps.isMultiBufferAvailable());
} // end of reportCapabilities()


private String getFlipText(BufferCapabilities.FlipContents flip)
{
  if (flip == null)
    return "false";
  else if (flip == BufferCapabilities.FlipContents.UNDEFINED)
    return "Undefined";
  else if (flip == BufferCapabilities.FlipContents.BACKGROUND)
    return "Background";
  else if (flip == BufferCapabilities.FlipContents.PRIOR)
    return "Prior";
  else // if (flip == BufferCapabilities.FlipContents.COPIED)
    return "Copied";
} // end of getFlipTest()
```

reportCapabilities() gets a reference to the graphics card's default configuration. In general, there can be many GraphicsConfiguration objects associated with a single graphics device, representing different drawing modes or capabilities.

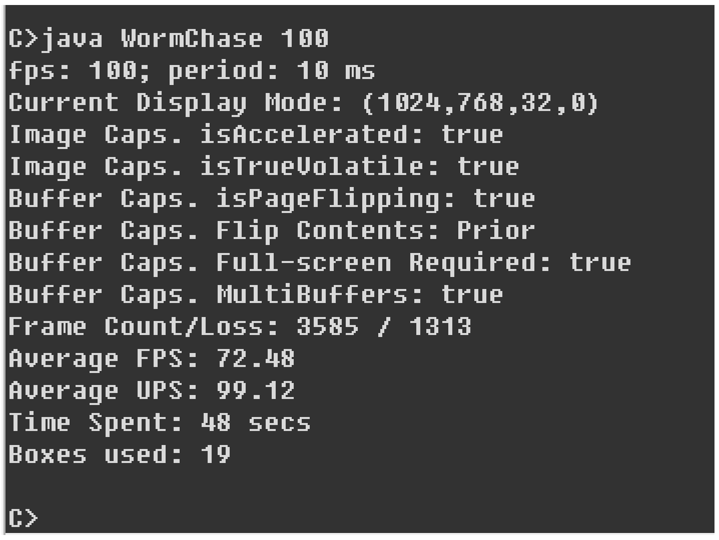
The ImageCapabilities object, imageCaps, permits hardware acceleration details to be examined. isAccelerated() reports whether hardware acceleration is available or not. On MS Windows machines, acceleration means that the buffer image is probably located in VRAM. isTrueVolatile() reports on whether the acceleration is "volatile", meaning that the buffer contents can be lost at any time. On Windows, this method will always return true if isAccelerated() is true.

It is possible to get an indication of the buffer strategies available on the graphics card by accessing the BufferCapabilities object for the graphics configuration. isPageFlipping() reports whether hardware page flipping is available. isFullScreenRequired() returns true if FSEM is required before page flipping can be attempted. isMultiBufferAvailable() returns true if more than two buffers are supported by the hardware, which will permit a flip chain to be employed.

getFlipContents() returns a *hint* of the technique used to do page flipping, in the form of a constant. Possible values include:

- FlipContents.COPIED. Getting back this value probably means that the buffer contents are copied to the screen.
- FlipContents.PRIOR. Usually this value indicates that page flipping with a video pointer is utilized, though this is not guaranteed.

A knowledge of these capabilities may allow the program to optimize its rendering cycle. Our code does not use this information. Figure 10 shows the details printed to standard output by WormChase.



```
C>java WormChase 100
fps: 100; period: 10 ms
Current Display Mode: (1024,768,32,0)
Image Caps. isAccelerated: true
Image Caps. isTrueVolatile: true
Buffer Caps. isPageFlipping: true
Buffer Caps. Flip Contents: Prior
Buffer Caps. Full-screen Required: true
Buffer Caps. MultiBuffers: true
Frame Count/Loss: 3585 / 1313
Average FPS: 72.48
Average UPS: 99.12
Time Spent: 48 secs
Boxes used: 19

C>
```

Figure 10. Standard Output for WormChase

The two “Image Caps.” and four “Buffer Caps.” lines are printed by reportCapabilities(). They suggest that page flipping using VRAM and a video pointer is probably being used on this machine.

More extensive tests can be performed, which are illustrated in the CapabilitiesTest.java example in Sun's FSEM tutorial (http://java.sun.com/docs/books/tutorial/extra/fullscreen/example.html).

3.5. Rendering the Game

At the core of run() are the game update and rendering steps, represented by two method calls:

```
public void run()
{    :
  while(running) {
    gameUpdate();
    screenUpdate();
    // sleep a while
    // maybe do extra gameUpdate()'s
  }
  :
}
```

gameUpdate() is unchanged from before – it updates the worm's state. screenUpdate() still performs active rendering but with the FSEM buffer strategy created in initFullScreen().

```
  private void screenUpdate()
  { try {
      gScr = bufferStrategy.getDrawGraphics();
      gameRender(gScr);
      gScr.dispose();
      if (!bufferStrategy.contentsLost())
        bufferStrategy.show();
      else
        System.out.println("Contents Lost");
    }
    catch (Exception e)
    { e.printStackTrace();
      running = false;
    }
  }  // end of screenUpdate()
```

screenUpdate() utilizes the bufferStrategy reference to get a graphics context (gScr) for drawing.

The try-catch block around the rendering operations means that their failure causes the running boolean to be set true, which will terminate the animation loop.

gameRender() writes to the graphic context in the same way that the gameRender() methods in earlier versions of WormChase write to their off-screen buffer.

```
  private void gameRender(Graphics gScr)
  {
    // clear the background
    gScr.setColor(Color.white);
    gScr.fillRect (0, 0, pWidth, pHeight);

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

    // report frame count & average FPS and UPS at top left
    // report time used and boxes used at bottom left
    // draw the pause and quit buttons
    // draw game elements: the obstacles and the worm
    // game over stuff
  } // end of gameRender()
```

The only change is at the start of gameRender(): there is no longer any need to create an off-screen buffer; initFullScreen() does it by calling createBufferStrategy().

Back in screenUpdate(), contentsLost() reports whether the VRAM used by the buffer has been lost since the call to getDrawGraphics(), caused by the OS taking back the memory.

show() makes the buffer visible on screen. This is achieved either by changing the video pointer (flipping), or by copying (blitting).

If contentsLost() returns true, it means that the entire image in the off-screen buffer must be redrawn. In our code, this will happen in any case during the next iteration of the animation loop, when screenUpdate() is called again.

If clipping is being utilized to reduce the amount of drawing into the buffers, it will need to be switched off until all of their images have been fully redrawn.

3.6. Finishing Off

The finishOff() method is called in the same way as in the UFS version of WormChase: either at the end of run() as the animation loop is finishing, or in response to a shutdown event.

```
  private void finishOff()
  {
    if (!finishedOff) {
      finishedOff = true;
      printStats();
      restoreScreen();
      System.exit(0):
    }
  }

  private void restoreScreen()
  { Window w = gd.getFullScreenWindow();
    if (w != null)
      w.dispose();
    gd.setFullScreenWindow(null);
  }
```

The call to restoreScreen() is the only addition to finishOff(). It switches off FSEM by executing setFullScreenWindow(null). This method also restores the display mode to its original state, if it was previously changed with setDisplayMode().

3.7. Displaying the Display Mode

initFullScreen() calls methods for reading and changing the display mode (although the call to setDisplayMode() is commented out). The display mode can only be changed after the application is in full-screen exclusive mode.

```
public void initFullScreen()
{
     :
  gd.setFullScreenWindow(this); // switch on FSEM

  // we can now adjust the display modes, if we wish
  showCurrentMode();

  // setDisplayMode(800, 600, 8);      // 800 by 600, 8 bits, or
  // setDisplayMode(1280, 1024, 32);  // 1280 by 1024, 32 bits

      :
}  // end of initFullScreen()
```

showCurrentMode() prints the display mode details for the graphic card.

```
private void showCurrentMode()
{
  DisplayMode dm = gd.getDisplayMode();
  System.out.println("Current Display Mode: (" +
          dm.getWidth() + "," + dm.getHeight() + "," +
          dm.getBitDepth() + "," + dm.getRefreshRate() + ")  " );
}
```

A display mode is composed of the width and height of the monitor (in pixels), bit depth (the number of bits per pixel), and refresh rate. getBitDepth() returns the integer BIT_DEPTH_MULTI (-1) if multiple bit depths are allowed. getRefreshRate() returns REFRESH_RATE_UNKNOWN (0) if there is no information available on the refresh rate; it also means that the refresh rate cannot be changed.

Figure 10 in section 3.4 shows the output from showCurrentMode(): a screen resolution of 1024 by 768, 32 bit depth, and an unknown (unchangeable) refresh rate.

3.8. Changing the Display Mode

A basic question is why bother changing the display mode, since the current setting is probably the most suitable one for the hardware?

The answer is to increase *performance*. A smaller screen resolution and bit depth reduces the amount of data transferred when the back buffer is copied to the screen. However, this advantage is irrelevant if the rendering is carried out by page flipping with video pointer manipulation.

A game can run more quickly if its images share the same bit depth as the screen. This is easier to do if we fix the bit depth inside the application.

A known screen size may make drawing operations simpler, especially for images which would normally have to be scaled to fit different display sizes.

Our setDisplayMode() method is supplied with a width, height, and bit depth, and attempts to set the display mode accordingly.

```
private void setDisplayMode(int width, int height, int bitDepth)
{
  if (!gd.isDisplayChangeSupported()) {
    System.out.println("Display mode changing not supported");
    return;
  }

  if (!isDisplayModeAvailable(width, height, bitDepth)) {
    System.out.println("Display mode (" + width + "," +
                 height + "," + bitDepth + ") not available");
    return;
  }

  DisplayMode dm = new DisplayMode(width, height, bitDepth,
           DisplayMode.REFRESH_RATE_UNKNOWN);   // any refresh rate
  try {
    gd.setDisplayMode(dm);
    System.out.println("Display mode set to: (" +
             width + "," + height + "," + bitDepth + ")");
  }
  catch (IllegalArgumentException e)
  {  System.out.println("Error setting Display mode (" +
            width + "," + height + "," + bitDepth + ")");  }

  try {  // sleep to give time for the display to be changed
    Thread.sleep(1000);  // 1 sec
  }
  catch(InterruptedException ex){}
}  // end of setDisplayMode()
```

The method checks whether display mode changing is supported (the application must be in FSEM for changes to go ahead), and also if the given mode is available for this graphics device, via a call to our isDisplayModeAvailable() method.

isDisplayModeAvailable() retrieves an array of display modes useable by this device, and cycles through them to see if one matches the requested parameters.

```
private boolean isDisplayModeAvailable(int width, int height,
                                         int bitDepth)
/* Check that a displayMode with this width, height, and
   bit depth is available.
   We don't care about the refresh rate, which is probably
   REFRESH_RATE_UNKNOWN anyway.
*/
{ DisplayMode[] modes = gd.getDisplayModes(); // modes list
  showModes(modes);

  for(int i = 0; i < modes.length; i++) {
    if ( width == modes[i].getWidth() &&
         height == modes[i].getHeight() &&
         bitDepth == modes[i].getBitDepth() )
      return true;
  }
  return false;
  }  // end of isDisplayModeAvailable()
```

showModes() is a pretty printer for the array of DisplayMode objects.

```
  private void showModes(DisplayMode[] modes)
  {
    System.out.println("Modes");
    for(int i = 0; i < modes.length; i++) {
      System.out.print("(" + modes[i].getWidth() + "," +
                             modes[i].getHeight() + "," +
                             modes[i].getBitDepth() + "," +
                             modes[i].getRefreshRate() + ")  ");
      if ((i+1)%4 == 0)
        System.out.println();
    }
    System.out.println();
  }
```

Back in our setDisplayMode(), a new display mode object is created, then set with GraphicDevice's setDisplayMode(), which may raise an exception if any of its arguments are incorrect. setDisplayMode() is asynchronous, and so the subsequent sleep() call delays execution a short time in the hope that the display will actually be changed before the method returns. Some programmers suggest a delay of two seconds.

The setDisplayMode() method in the GraphicsDevice class (i.e. not our setDisplayMode()) is known to be somewhat bug-ridden. However, it has improved in recent versions of J2SE 1.4, and will probably improve further in J2SE 1.5. Our tests across several versions of Windows, using J2SE 1.4.2, sometimes resulted in a JVM crash, occurring after the program had been run successfully a few times. This is one reason why the call to our setDisplayMode() is commented out in initFullScreen().

setDisplayMode() can be employed to set the screen size to 800x600, 8 bit depth, like so:

```
    setDisplayMode(800, 600, 8);
```

The resulting on-screen appearance is shown in Figure 11. The reduced screen resolution means that the various graphical elements (e.g. the text, circles, boxes) are bigger. The reduced bit depth causes a reduction in the number of available colours, but the basic colours used here (blue, black, red, and green) are still present.

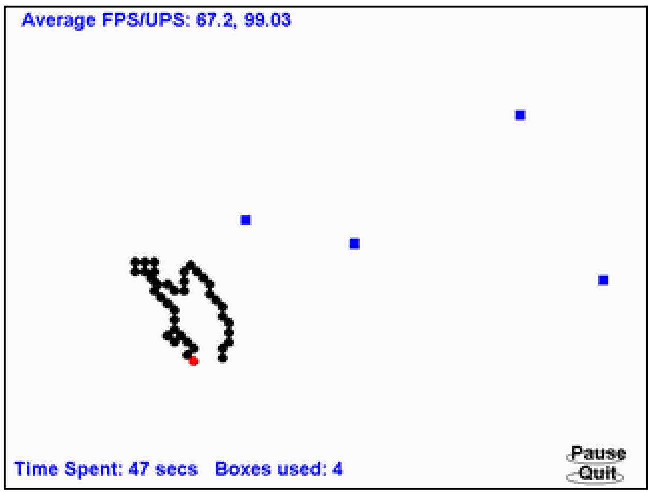
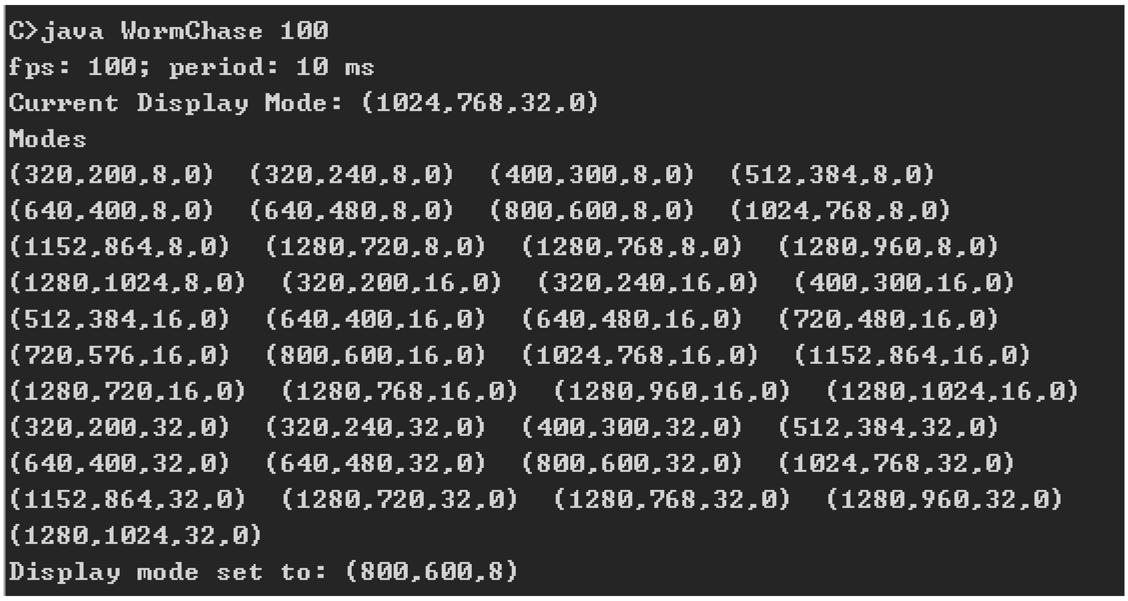


Figure 11. WormChase with a Modified Display Mode.

Figure 12 shows the standard output from WormChase which lists the initial display mode, the range of possible modes, and the new mode.



```
C>java WormChase 100
fps: 100; period: 10 ms
Current Display Mode: (1024,768,32,0)
Modes
(320,200,8,0)  (320,240,8,0)  (400,300,8,0)  (512,384,8,0)
(640,400,8,0)  (640,480,8,0)  (800,600,8,0)  (1024,768,8,0)
(1152,864,8,0)  (1280,720,8,0)  (1280,768,8,0)  (1280,960,8,0)
(1280,1024,8,0)  (320,200,16,0)  (320,240,16,0)  (400,300,16,0)
(512,384,16,0)  (640,400,16,0)  (640,480,16,0)  (720,480,16,0)
(720,576,16,0)  (800,600,16,0)  (1024,768,16,0)  (1152,864,16,0)
(1280,720,16,0)  (1280,768,16,0)  (1280,960,16,0)  (1280,1024,16,0)
(320,200,32,0)  (320,240,32,0)  (400,300,32,0)  (512,384,32,0)
(640,400,32,0)  (640,480,32,0)  (800,600,32,0)  (1024,768,32,0)
(1152,864,32,0)  (1280,720,32,0)  (1280,768,32,0)  (1280,960,32,0)
(1280,1024,32,0)
Display mode set to: (800,600,8)
```

Figure 12. Display Mode Output.

An essential task if the display mode is changed, is to change it back to its original setting at the end of the application. WormChase does this by calling gd.setFullScreenWindow(null) in restoreScreen().

3.9. Timings for FSEM

Timing results for the FSEM WormChase are given in Table 3.

Requested FPS	*20*	*50*	*80*	*100*
Windows 98	20/20	50/50	81/83	84/100
Windows 2000	20/20	50/50	60/83	60/100
Windows XP (1)	20/20	50/50	74/83	76/100
Windows XP (2)	20/20	50/50	83/83	85/100

Table 3. Average FPS/UPSs for the FSEM WormChase.

WormChase on the Windows 2000 machine is the worst performer as usual, but its UPS values are fine. FSEM produces a drastic increase the frame rate: the UFS version only manages 18 FPS when 80 is requested, compared to 60 here.

The Windows 98 and XP boxes produce good/excellent frame rates at 80 FPS, but are unable to go any faster. FSEM improves the frame rates by around 20% compared to UFS, except in the case of the first XP machiine.

It is briefly worth mentioning that frame rates for Windows-based FSEM applications can be collected using the FRAPS utility (http://www.fraps.com). Figure 12 shows WormChase with a FRAPS-generated FPS value in the top right hand corner.

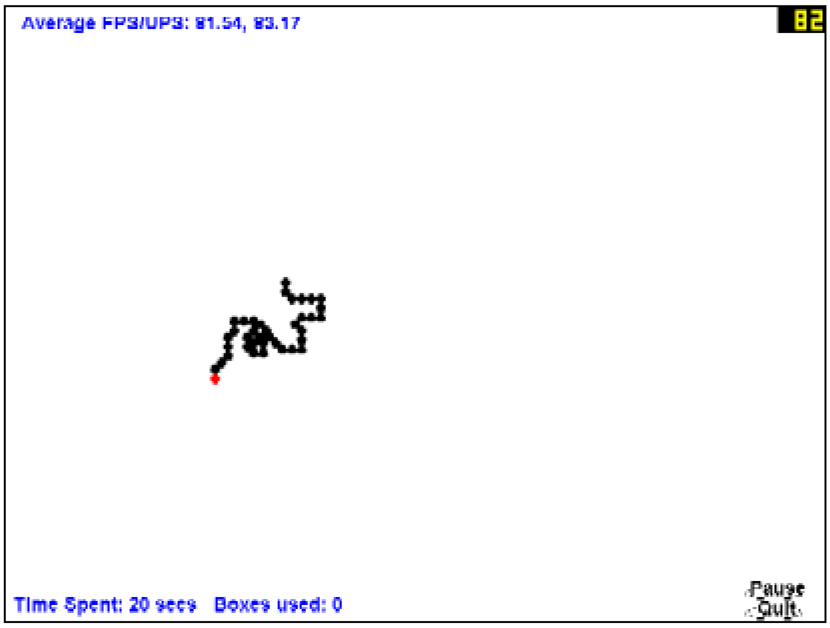


Figure 12. FSEM WormChase with FRAPS Output.

One reason for the flattening out of the frame rate values may be that BufferStrategy's show() method, used in our screenUpdate() to render to the screen, is tied to the frequency of the vertical synchronization (vsync) of the monitor. In FSEM, show() blocks until the next vsync signal.

4. Timings at 80-85 FPS

Table 4 shows UFS, AFS, and FSEM results for different versions of MS Windows when 80 FPS is requested.

Requested 80 FPS	*AFS*	*UFS*	*FSEM*
Windows 98	75/83	70/83	81/83
Windows 2000	20/83	18/83	60/83
Windows XP (1)	82/83	77/83	74/83
Windows XP (2)	75/83	68/83	83/83

Table 3. Average FPS/UPSs for the AFS, UFS, and FSEM Versions of WormChase when 80 FPS is Requested.

The numbers send mixed signals and, in any case, the sample size is too small for strong conclusions to be drawn. Nevertheless, a few observations can be made.

The use of additional state updates to keep the updates/second (UPS) close to the requested FPS is an important technique for giving the appearance of speed even when the rendering rate is sluggish.

FSEM offers better frame rates than UFS, sometimes dramatically better. However, FSEM's benefits rely on MS Window's access to the graphics device via DirectDraw. The improvements on Linux, Solaris, and the Mac OS may not be so striking.

AFS produces higher frame rates than UFS, and may be a good choice if full screen exclusive mode is not available.

All the approaches supply good/excellent frame rates on modern CPUs (the Windows 2000 machine sports a Pentium II). Consequently, the 'best' full-screen technique for a particular game will probably have to be determined by timing the game. Additional optimization techniques, such as clipping, may highlight the benefits of one technique over another.

Thanks to two of my students, Khun Patipol Kulasi and Khun Thana Konglikhit, who helped gather the timing data used in this chapter, and in chapter 2.