Chapter 6.2. A Side Scroller

The player's sprite in a side-scrolling game usually travels left or right through a landscape that extends well beyond the limits of the gaming pane. The landscape scrolls past in the background while the player jumps (or perhaps flies) over various obstacles and bad guys, landing safely on platforms to collect treasure, food, or rescue Princess Peach. Of course, the quintessential side-scroller is *Super Mario* (still available today in many versions, on many platforms).

Most side-scrollers implement their backgrounds using *tile maps*: the tiles can be square, rectangular, or any shape once transparent GIFS are brought in. Tiles can be unchanging blocks, animated, or behave like (clever) sprites.

Backgrounds are often composed from several tile map *layers*, representing various background and foreground details. They may employ *parallax scrolling* – layers 'further' back in the scene scroll past at a slower rate than layers nearer the front.

Tiling is a versatile technique: *Super Mario* (and its numerous relatives) present a side view of the game world, but tiles can offer bird's eye viewpoints looking down on the scene from above, and isometric views, as is *Civilization*, to create a pseudo-3D environment. We'll implement a basic isometric game in the next chapter.

This chapter describes JumpingJack, a side scroller in the *Super Mario* mould, considerably simpler, but illustrating tile maps, layers, parallax scrolling, and a jumping hero called 'Jack' who has to dodge exploding fireballs.

JumpingJack has a few unusual elements: the foreground is a tile map, which Jack scrambles over, but the other layers are large GIFs. The background layers and tiles wrap around the drawing area, so if Jack travels long enough he returns to his starting point. There is an introductory start-up 'screen', which doubles as a help screen, toggled by pressing 'h'.

Two screenshots of JumpingJack are shown in Figure 1.

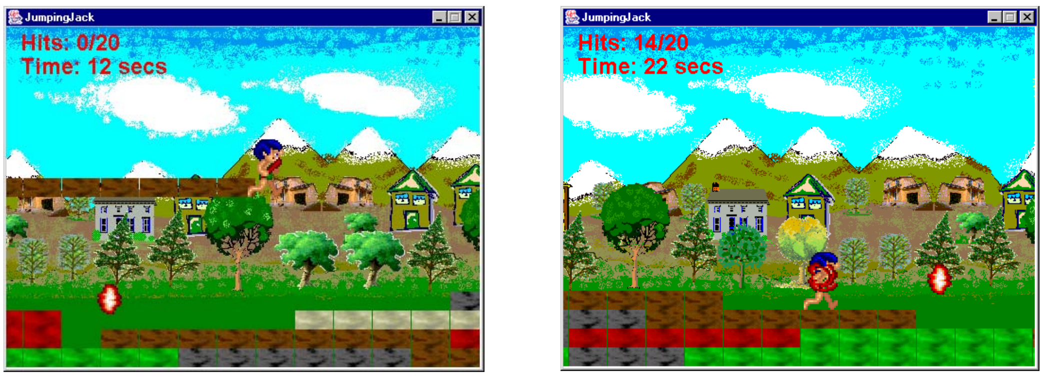

Figure 1. Two JumpingJack Screenshots.

The arrow keys make Jack move left, right, stand still, and jump. Once Jack starts moving (when the user presses the left or right arrow keys), he keeps moving until he hits a brick. To prevent him stopping, the user should press the jump key (up arrow) to make him hop over bricks in his path.

Fireballs shoot out from the right edge of the panel, heading to the left, unaffected by bricks in their way. If a fireball hits Jack, the number of hits reported in the top-left of the panel is incremented; when it reaches 20, the game is over, and a score is reported. As a slight relief, only a single fireball is shot at Jack at a time (it also simplifies our coding).

An instrumental version of "Jumping Jack Flash" by the *Rolling Stones* repeatedly plays in the background, occasionally punctuated by an explosion audio clip when a fireball hits Jack.

1. JumpingJack in Layers

The easiest way of understanding JumpingJack's coding design is to consider the graphical layers making up the on-screen image. Figure 2 shows the various parts, labeled with the classes that represent them.

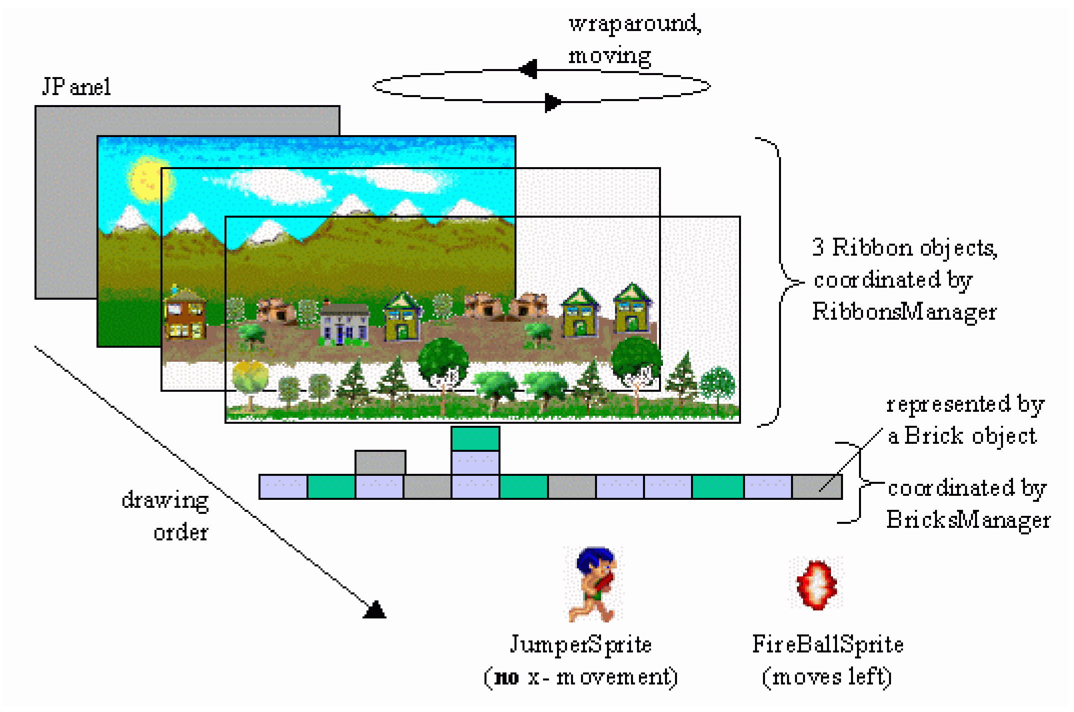
wraparound, moving
JPanel
3 Ribbon objects, coordinated by RibbonsManager
represented by a Brick object
coordinated by BricksManager
drawing order
JumperSprite (no x-movement)
FireBallSprite (moves left)

Figure 2. The Visual Layers in JumpingJack.

The scenic background is made from three GIFs (mountains.gif, houses.gif, and trees.gif in /Images), all wider than the JPanel, and moving at different speeds behind the bricks layer and sprites. The images are drawn to the JPanel in back-to-front order, and easily combined since houses.gif and trees.gif contain large transparent areas.

Each images is maintained by a Ribbon object, which are collectively managed by a RibbonsManager object.

The bricks layer is composed from bricks, positioned on-screen according to a bricks map created by the programmer. Each brick is assigned a GIF, which can be any rectangular shape (other shapes can be 'faked' by using transparency). Each brick is represented by a Brick object, grouped together and managed by BricksManager.

The bricks layer is also wider than the JPanel, and wraps around in a similar way to the Ribbon backgrounds. Jack walks or jumps over the bricks.

A strange feature of side-scrollers, which is hard to believe unless you watch the game very carefully, is that the hero sprite often does not move in the x-direction. The sprite's apparent movement is achieved by shifting the background. For example, when Jack starts going right, he doesn't move at all (aside from his little legs flapping). Instead, the scenery (the GIF ribbons and the bricks layer) move *left*. Similarly, when Jack appears to move left, it is actually the scenery moving right.

When Jack jumps, the sprite moves up and down over the space of 1-2 seconds. However, the jump's arc is an illusion caused by the background moving.

2. UML Diagrams for JumpingJack

Figure 3 shows the UML diagrams for all the classes in the JumpingJack application. Only the class names are shown.

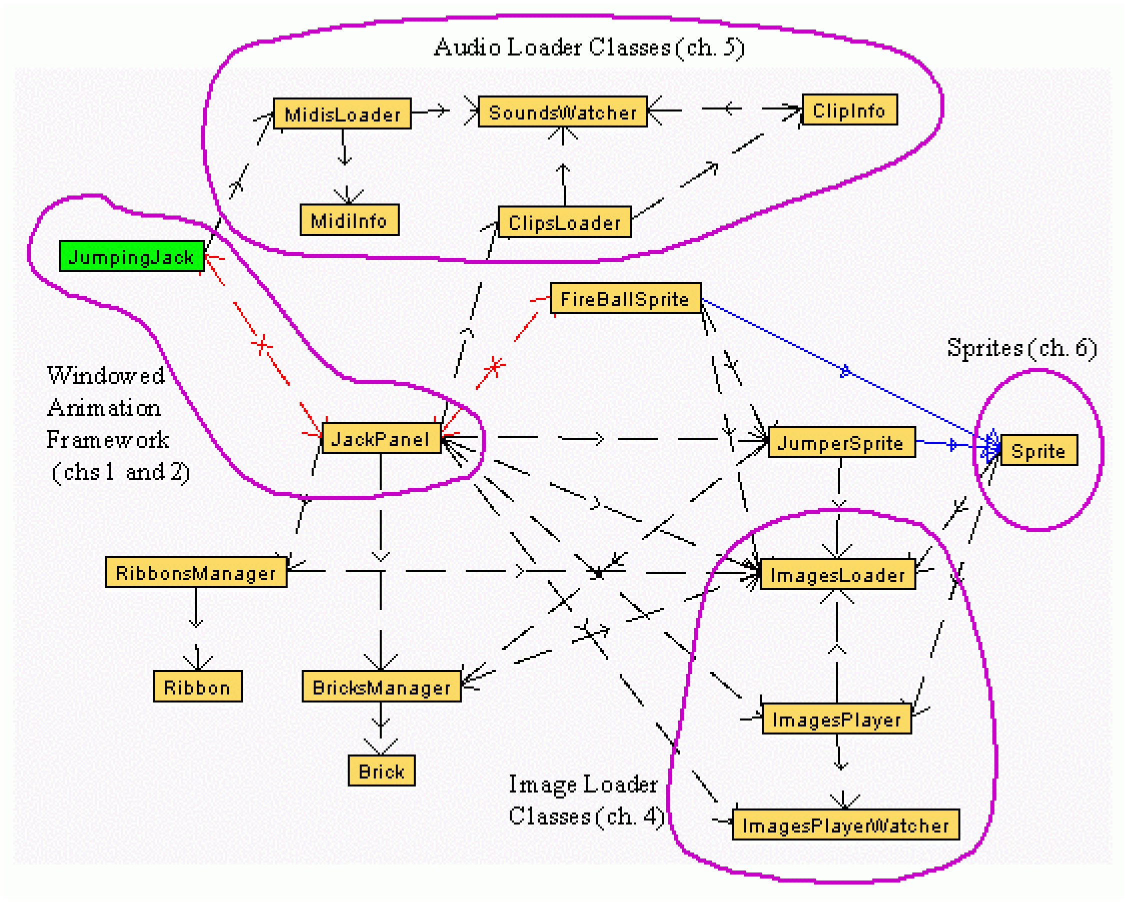

Figure 3. All the UML Class Diagrams for JumpingJack.

The large number of classes is daunting, but many of them can be ignored since they're unchanged from earlier chapters.

The image loaders reads in the GIFs used by the Ribbon objects, and the tile and sprite images. ImagesPlayer animates Jack's legs and the fireball explosion.

The audio loaders play the “Jumping Jack Flash” MIDI sequence, and the explosion and applause clips (always applaud the user, even when they lose).

The JumperSprite object handles Jack, while FireBallSprite is for the fireball; both are subclasses of the Sprite class which we introduced in the last chapter.

The JumpingJack JFrame and the JackPanel JPanel implement the windowed animation framework of chapters 1 and 2; BugRunner of chapter 6 uses the same technique.

If we strip away the unchanged classes from earlier chapters, we're left with the more manageable collection of UML class diagrams shown in Figure 4. The public methods, and any public or protected data, are shown.

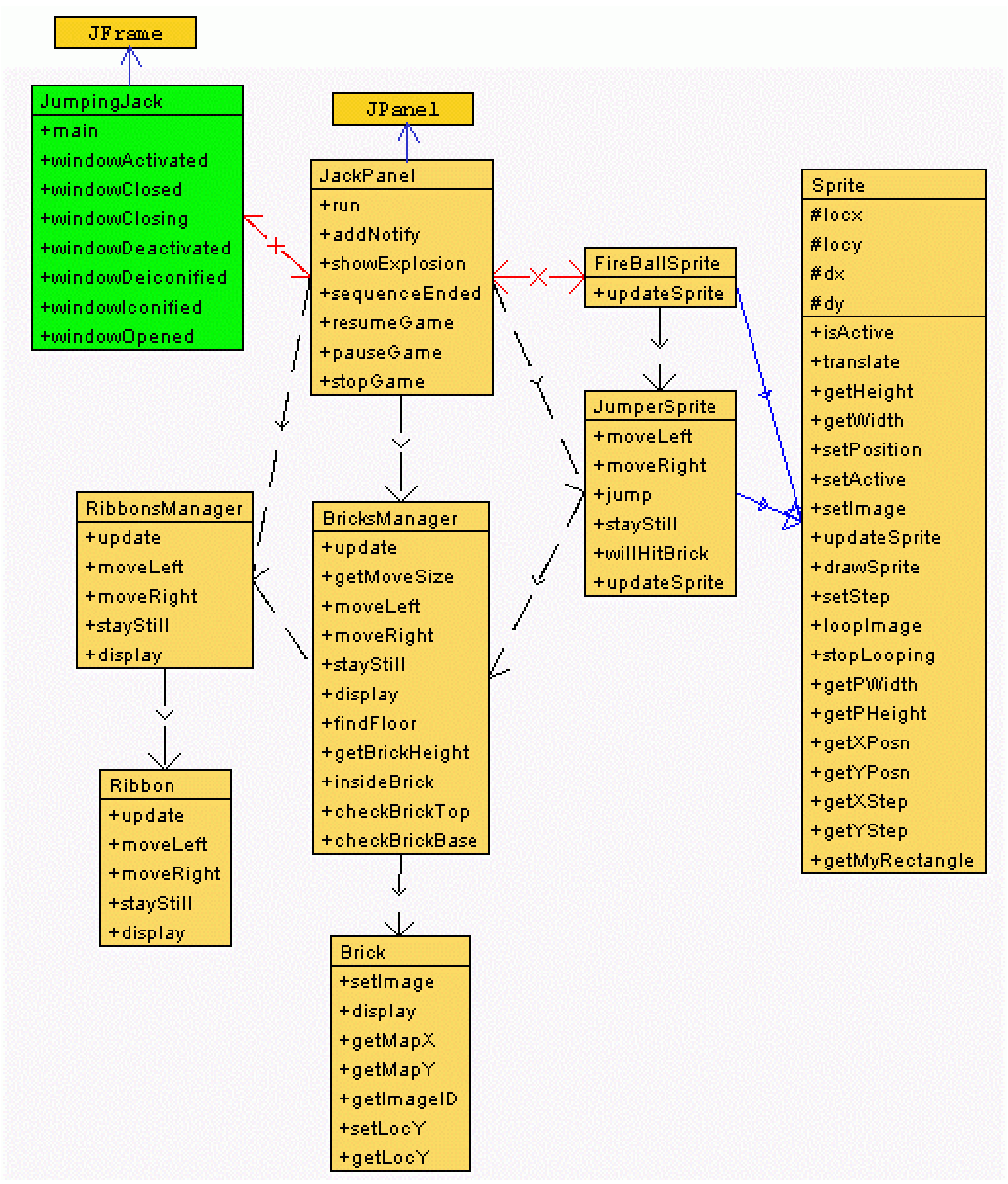

Figure 4. The Core Classes of JumpingJack.

The Sprite class is included since JumperSprite and FireBallSprite use many of its methods, but it's unchanged from chapter 6. We won't bother explaining it again.

3. The JumpingJack Class

JumpingJack fixes the frame rate at only 30 FPS; anything faster makes it almost impossible to control Jack. The illusion of speed is governed by how fast the bricks and image ribbons move, which is controlled by a single moveSize variable in the BricksManager class. moveSize specifies the distance that the bricks layer should be shifted in each update of the animation loop.

JumpingJack sets up window listener methods for pausing and resuming the game, in a similar way to the BugRunner application in chapter 6.

4. The JackPanel Class

JackPanel is a subclass of JPanel, and implements the animation framework described in chapter 1 and 2; JackPanel also resembles the BugPanel class of chapter 6.

The JackPanel() constructor creates the game entities: the RibbonsManager, BricksManager, JumperSprite, and FireBallSprite objects. It also prepares the explosion animation and the title/help screen.

```
// some of the globals
private JumpingJack jackTop;
private JumperSprite jack;          // the sprites
private FireBallSprite fireball;
private RibbonsManager ribsMan;     // the ribbons manager
private BricksManager bricksMan;    // the bricks manager

// to display the title/help screen
private boolean showHelp;
private BufferedImage helpIm;

// explosion-related
private ImagesPlayer explosionPlayer = null;
private boolean showExplosion = false;
private int explWidth, explHeight;   // image dimensions
private int xExpl, yExpl;   // coords where image is drawn
           :

public JackPanel(JumpingJack jj, long period)
{
  jackTop = jj;
  this.period = period;

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

  setFocusable(true);
  requestFocus();    // so receives key events

  addKeyListener( new KeyAdapter() {
    public void keyPressed(KeyEvent e)
    { processKey(e); }
  });

  // initialise the loaders
  ImagesLoader imsLoader = new ImagesLoader(IMS_INFO);
  clipsLoader = new ClipsLoader(SNDS_FILE);

  // initialise the game entities
  bricksMan = new BricksManager(PWIDTH, PHEIGHT,
                                  BRICKS_INFO, imsLoader);
  int brickMoveSize = bricksMan.getMoveSize();

  ribsMan = new RibbonsManager(PWIDTH, PHEIGHT,
                                     brickMoveSize, imsLoader);

  jack = new JumperSprite(PWIDTH, PHEIGHT, brickMoveSize,
           bricksMan, imsLoader, (int)(period/1000000L) ); // in ms

  fireball = new FireBallSprite(PWIDTH, PHEIGHT,
                                       imsLoader, this, jack);

  // prepare the explosion animation
  explosionPlayer =  new ImagesPlayer("explosion",
                    (int)(period/1000000L), 0.5, false, imsLoader);
  BufferedImage explosionIm = imsLoader.getImage("explosion");
  explWidth = explosionIm.getWidth();
  explHeight = explosionIm.getHeight();
  explosionPlayer.setWatcher(this)     // report anim end back here

  // prepare title/help screen
  helpIm = imsLoader.getImage("title");
  showHelp = true;    // show at start-up
  isPaused = true;

  // set up message font
  msgsFont = new Font("SansSerif", Font.BOLD, 24);
  metrics = this.getFontMetrics(msgsFont);
}  // end of JackPanel()
```

We create the BricksManager object first, so that a brickMoveSize variable can be initialised. This will contain the number of pixels the bricks map is shifted when the sprite appears to make a move. brickMoveSize is used as the basis for the move increments employed by the Ribbon objects managed in RibbonsManager, and is also used by the JumperSprite. However, the fireball travels at its own rate, independent of the background, so doesn't require the move size.

JackPanel is in charge of a fireball's animated explosion and its associated audio, rather than FireBallSprite. The explosion animation in explosion.gif is loaded into an ImagesPlayer (see Figure 5 for its contents), and the dimensions of its first image are recorded. When the sequence is finished, ImagesPlayer will call sequenceEnded() back in JackPanel.

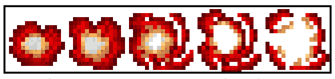
Figure 5. The Images Strip in explosion.gif.

The title/help image (in title.gif, see Figure 6) is loaded into the global helpIm, and the booleans showHelp and isPaused are set. isPaused causes the game's execution to pause, and was introduced in the basic game animation framework; showHelp is a new boolean, examined by gameRender() to decide whether to draw the image.

gameRender() displays the image centered in the JPanel, so the image should not be too large or its borders may be beyond the edges of the panel.

If the image is the same size as the JPanel, it will totally obscure the game window, and so look more like a 'screen' rather than an image drawn on the game surface.

Clever use can be made of transparency to make the image an interesting shape, although it is still a rectangle as far as drawImage() is concerned.

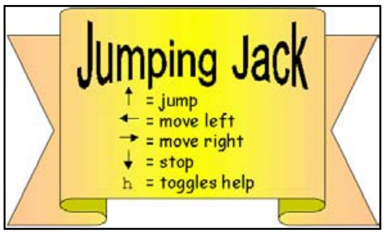

Figure 6. title.gif: the title/help screen in JumpingJack

The switching on of isPaused while the help image is visible requires a small change to the resumeGame() method.

```
public void resumeGame()
{ if (!showHelp)     // CHANGED
    isPaused = false;
}
```

This method is called from the enclosing JumpingJack JFrame when the frame is activated (deiconified). Previously, resumeGame() always set isPaused to false, but now this occurs only when the help screen is not being displayed.

If the game design requires distinct title and help screens, then two images and two booleans will be needed. For example, showHelp for the help image, and showTitle for the titles, which will be examined in gameRender(). Initially, showTitle would be set true, showHelp assigned false. When either the titles or the help is on-screen, isPaused should be set to true.

4.1. Dealing with Input

Only keyboard input is supported in JumpingJack. A key press triggers a call to processKey(), which handles three kinds of input: termination keys, help controls, and game play keys.

```
private void processKey(KeyEvent e)
{
  int keyCode = e.getKeyCode();

  // termination keys
  // listen for esc, q, end, ctrl-c on the canvas to
  // allow a convenient exit from the full screen configuration
  if ((keyCode==KeyEvent.VK_ESCAPE) || (keyCode==KeyEvent.VK_Q) ||
      (keyCode == KeyEvent.VK_END) ||
      ((keyCode == KeyEvent.VK_C) && e.isControlDown()) )
    running = false;
  // help controls
  if (keyCode == KeyEvent.VK_H) {
    if (showHelp) {  // help being shown
      showHelp = false;  // switch off
      isPaused = false;
    }
    else {  // help not being shown
     showHelp = true;    // show it
     isPaused = true;
    }
  }

  // game-play keys
  if (!isPaused && !gameOver) {
    // move the sprite and ribbons based on the arrow key pressed
    if (keyCode == KeyEvent.VK_LEFT) {
      jack.moveLeft();
      bricksMan.moveRight();   // bricks and ribbons move other way
      ribsMan.moveRight();
    }
    else if (keyCode == KeyEvent.VK_RIGHT) {
      jack.moveRight();
      bricksMan.moveLeft();
      ribsMan.moveLeft();
    }
    else if (keyCode == KeyEvent.VK_UP)
      jack.jump();    // jumping has no effect on bricks/ribbons
    else if (keyCode == KeyEvent.VK_DOWN) {
      jack.stayStill();
      bricksMan.stayStill();
      ribsMan.stayStill();
    }
  }
}  // end of processKey()
```

The termination keys are utilized in the same way as in earlier examples.

The help key (‘h’) toggles the showHelp and isPaused booleans on and off.

The arrow keys are assigned to be the game play keys. When the left or right arrow keys are pressed, the scenery (the bricks and ribbons) is moved in the opposite direction from Jack. We will see later that the calls to moveLeft() and moveRight() in Jack do not actually cause the sprite to move at all.

4.2. Multiple Key Presses/Actions

A common requirement in many games is to process multiple key presses together. For example, it should be possible for Jack to jump and move left/right at the same time. There are two parts to this feature: implementing key capture code to handle simultaneous key presses, and implementing simultaneous behaviours in the sprite (or other game entity).

JumpingJack already has the ability to jump and move left/right at the same time: it was wired into the JumperSprite class at the design stage, as we'll see later. If Jack is currently moving left or right, then an up arrow press will make him jump. A related ‘trick’ is to start him jumping from a stationary position, causing him to rise and fall over 1-2 seconds. During that interval, the left or right arrow keys can be pressed to get him moving horizontally through the air, or to change his direction in mid flight!

Although Jack can jump and move at the same time, this behaviour is triggered by distinct key presses. First the left/right arrow key is pressed to start him moving, and then the up arrow key makes him jump. Alternatively the up arrow key can be pressed first, followed by the left or right arrow keys. If we want to capture multiple key presses at the same time, then modifications are needed to the key listener code.

The main change is to use keyPressed() *and* keyReleased(), and to introduce new booleans to indicate when keys are being pressed. The basic coding strategy is shown below:

```
// global booleans, true when a key is being pressed
private boolean leftKeyPressed = false;
private boolean rightKeyPressed = false;
private boolean upKeyPressed = false;
             :

  // inside JackPanel()
  addKeyListener( new KeyAdapter() {
     public void keyPressed(KeyEvent e)
     { processKeyPress(e);  }
     public void keyReleased(KeyEvent e)
     { processKeyRelease(e); }
  });
                  :

private void processKeyPress(KeyEvent e)
{
  int keyCode = e.getKeyCode();

  // record the key press in a boolean
  if (keyCode == KeyEvent.VK_LEFT)
    leftKeyPressed = true;
  else if (keyCode == KeyEvent.VK_RIGHT)
    rightKeyPressed = true;
  else if (keyCode == KeyEvent.VK_UP)
    upKeyPressed = true;

  // use the combined key presses
  if (leftKeyPressed && upKeyPressed)
    // do a combined left and up action
  else if (rightKeyPressed && upKeyPressed)
    // do a combined right and up action
                :
} // end of processKeyPress()


private void processKeyRelease(KeyEvent e)
{
  int keyCode = e.getKeyCode();

  // record the key release in a boolean
  if (keyCode == KeyEvent.VK_LEFT)
    leftKeyPressed = false;
  else if (keyCode == KeyEvent.VK_RIGHT)
    rightKeyPressed = false;
  else if (keyCode == KeyEvent.VK_UP)
```

```
      upKeyPressed = false;
  } // end of processKeyRelease()
```

Key presses cause the relevant booleans to be set, and these remain set until the user releases the keys at some future time. The combination of key presses can be detected by testing the booleans in processKeyPress().

This coding effort is only need for combinations of ‘normal’ keys (e.g. the letters, the numbers, arrow keys). Key combinations involving a standard key and the shift, control, or meta keys can be detected more directly by using the KeyEvent methods isShiftDown(), isControlDown(), and isMetaDown(). This coding style can be seen in the termination keys code in processKey():

```
  if (...||((keyCode==KeyEvent.VK_C) && e.isControlDown())) //ctrl-c
    running = false;
```

4.3. The Animation Loop

The animation loop is located in run(), and is unchanged from earlier examples (e.g. run() in BugRunner of chapter 6). In essence, it is:

```
  public void run()
  { ...
    while (running) {
      gameUpdate();
      gameRender();
      paintScreen();
      // timing correction code
    }
    System.exit(0);
  }
```

gameUpdate() updates the various game elements (the sprites, the bricks layer and Ribbon objects).

```
  private void gameUpdate()
  {
    if (!isPaused && !gameOver) {
      if (jack.willHitBrick()) { // collision checking first
        jack.stayStill();    // stop jack and scenery
        bricksMan.stayStill();
        ribsMan.stayStill();
      }
      ribsMan.update();   // update background and sprites
      bricksMan.update();
      jack.updateSprite();
      fireball.updateSprite();

      if (showExplosion)
        explosionPlayer.updateTick();  // update the animation
    }
  }
```

The new element here is dealing with potential collisions: if Jack *will* hit a brick when the current update is carried out then the update should be cancelled. This requires a testing phase before the update is committed, embodied in willHitBrick() in JumperSprite. If Jack will hit a brick with his next update, it will be due to him moving (we do not have animated tiles in this game), so the collision can be avoided by stopping Jack (and the backgrounds).

The fireball sprite is unaffected by Jack's impending collision: it keeps travelling left regardless of what the JumperSprite is doing.

The showExplosion boolean is set true when the explosion animation is being played by the ImagesPlayer, explosionPlayer, and so updateTick() must be called during each game update.

gameRender() draws the multiple layers making up the game. Their ordering is important: rendering must start with the image furthest back in the scene and work forwards. The order used in JumpingJack is illustrated in Figure 2.

```
private void gameRender()
{
  if (dbImage == null){
    dbImage = createImage(PWIDTH, PHEIGHT);
    if (dbImage == null) {
      System.out.println("dbImage is null");
      return;
    }
    else
      dbg = dbImage.getGraphics();
  }

  // draw a white background
  dbg.setColor(Color.white);
  dbg.fillRect(0, 0, PWIDTH, PHEIGHT);

  // draw the game elements: order is important
  ribsMan.display(dbg);        // the background ribbons
  bricksMan.display(dbg);      // the bricks
  jack.drawSprite(dbg);        // the sprites
  fireball.drawSprite(dbg);

  if (showExplosion)      // draw the explosion (in front of jack)
    dbg.drawImage(explosionPlayer.getCurrentImage(),
                                      xExpl, yExpl, null);
  reportStats(dbg);
  if (gameOver)
    gameOverMessage(dbg);

  if (showHelp)    // draw help at the very front (if switched on)
    dbg.drawImage(helpIm, (PWIDTH-helpIm.getWidth())/2,
                        (PHEIGHT-helpIm.getHeight())/2, null);
}  // end of gameRender()
```

gameRender() relies on the RibbonsManager and BricksManager objects to draw the multiple Ribbon objects and the individual bricks. The code order means that Jack will be drawn behind the fireball if they are at the same spot (i.e. when the fireball hits him). An explosion is drawn in front of the fireball, then the game statistics, the game over message, and finally the help 'screen' is top-most.

4.4. Handling an Explosion

The fireball sprite passes the responsibility of showing the explosion animation and its audio clip to JackPanel, by calling showExplosion().

```
// names of the explosion clips
private static final String[] exploNames =
                           {"explo1", "explo2", "explo3"};
           :

public void showExplosion(int x, int y)
// called by FireBallSprite
{
  if (!showExplosion) {  // only allow a single explosion at a time
    showExplosion = true;
    xExpl = x - explWidth/2;   // (x,y) is center of explosion
    yExpl = y - explHeight/2;

    /* Play an explosion clip, but cycle through them.
       This adds variety, and gets round not being able to
       play multiple instances of a clip at the same time. */
    clipsLoader.play( exploNames[ numHits%exploNames.length ],
                                                       false);
    numHits++;
  }
} // end of showExplosion()
```

The (x,y) coordinate passed to showExplosion() is assumed to be where the center of the explosion should occur, and so the top-left corner of the explosion image is calculated and placed in the globals (xExpl, yExpl). These are used to position the explosion in gameRender().

The use of a single boolean, showExplosion, to determine if an explosion appears on-screen or not, is adequate only if a single explosion animation is shown at a time. This means that if a fireball hits Jack while an explosion sequence is playing, a second animation will not be rendered. This restriction allows us to manage with a single ImagesPlayer object instead of a set containing one ImagesPlayer for each of the current explosions.

play() in ClipsLoader eventually calls start() for the Clip object. A design feature of start() is that when a clip is already playing, further calls to start() are ignored. This makes it impossible to play multiple instances of the same Clip object at the same time.

One answer would be to ignore the problem, and do nothing (as with the multiple explosion animations issue). However, a typical audio clip can last several seconds, which is a long time to wait for the sound to be playable again. Also, the absence of the sound effect seems more noticeable than the lack of the animation.

Therefore, we've gone for a set of explosion clips, listed in exploNames[], and the code cycles through them. A set of three seems enough to deal with even the highest rate of fireball hits to Jack. Since these names represent separate Clips stored in the ClipsLoader, they can be played simultaneously.

Once an explosion animation has finished playing, its ImagesPlayer object calls sequenceEnded() in JackPanel.

```
public void sequenceEnded(String imageName)
// called by ImagesPlayer when the expl. animation finishes
{
  showExplosion = false;
  explosionPlayer.restartAt(0);   // reset animation for next time

  if (numHits >= MAX_HITS) {
    gameOver = true;
    score = (int) ((J3DTimer.getValue() -
                             gameStartTime)/1000000000L);
    clipsLoader.play("applause", false);
  }
}
```

sequenceEnded() resets the animation, so it’s ready to be played next time, and checks the game over condition. If the number of fireball hits equals or exceeds MAX_HITS, then the game over flag is set, causing the game to eventually terminate.

The main question about sequenceEnded() is why it’s being used at all. The reason is to make the game terminate at a natural time – just after an explosion has finished. For instance, if the game over condition was tested at the end of showExplosion(), the game may be terminated while ImagesPlayer was in the middle of displaying the explosion animation. This doesn’t really matter, but may seem a little odd to the player.

5. The RibbonsManager Class

RibbonsManager is mainly a router, sending move method calls, update() and display () calls, on to the multiple Ribbon objects under its charge. Initially, it creates the Ribbon objects, so also acts as a central storage for their GIFs and move factors.

The initialisation phase is carried out in the constructor:

```
// globals
private String ribImages[] = {"mountains", "houses", "trees"};
private double moveFactors[] = {0.1, 0.5, 1.0};
   // applied to moveSize
   // a move factor of 0 would make a ribbon stationary

private Ribbon[] ribbons;
private int numRibbons;
private int moveSize;
   // standard distance for a ribbon to 'move' each tick


public RibbonsManager(int w, int h, int brickMvSz,
                                          ImagesLoader imsLd)
{ moveSize = brickMvSz;
```

```
        // the basic move size is the same as the bricks map

    numRibbons = ribImages.length;
    ribbons = new Ribbon[numRibbons];

    for (int i = 0; i < numRibbons; i++)
       ribbons[i] = new Ribbon(w, h, imsLd.getImage( ribImages[i] ),
                               (int) (moveFactors[i]*moveSize));
  }  // end of RibbonsManager()
```

The choice of GIFs is hardwired into ribImages[], and the constructor loops through the array creating a Ribbon object for each one.

The basic move size is the same as that used by the bricks layer, but multiplied by a fixed moveFactors[] value to get a size suitable for each Ribbon. (A move size is the amount that a background layer moves in each animation period.)

A move factor will usually be less than one, to reduce the move size for a Ribbon in comparison to the bricks layer. This reinforces the illusion that the Ribbons are further back in the scene.

The other methods in RibbonsManager are simple routers. For example, moveRight() and display():

```
  public void moveRight()
  { for (int i=0; i < numRibbons; i++)
      ribbons[i].moveRight();
  }

  public void display(Graphics g)
  /* The display order is important.
     Display ribbons from the back to the front of the scene. */
  { for (int i=0; i < numRibbons; i++)
      ribbons[i].display(g);
  }
```

moveLeft(), stayStill(), and update() are similar.

The calls from display() ensure that the display of the Ribbons is carried out in a back-to-front order, in this case, mountains, houses, then trees.

6. The Ribbon Class

A Ribbon object manages a wraparound, moveable image, which should be wider than the game panel. This width requirement is important for the amount of work needed to draw the image as it wraps around the JPanel. The worst case task is to draw the tail of the image on the left side, followed by its start on the right. If the image was narrower than the panel, then it might be necessary to use three drawImage () calls, or more.

The constructor initialises the graphic, its moveSize value, two movement flags, and a position variable called xImHead.

```
  // globals
  private BufferedImage im;
  private int width;      // the width of the image (>= pWidth)
  private int pWidth, pHeight;    // dimensions of display panel

  private int moveSize;       // size of the image move (in pixels)
  private boolean isMovingRight;  // movement flags
  private boolean isMovingLeft;

  private int xImHead;   // panel position of image’s left side


  public Ribbon(int w, int h, BufferedImage im, int moveSz)
  {
    pWidth = w; pHeight = h;

    this.im = im;
    width = im.getWidth();    // no need to store the height
    if (width < pWidth)
      System.out.println("Ribbon width < panel width");

    moveSize = moveSz;
    isMovingRight = false;   // no movement at start
    isMovingLeft = false;
    xImHead = 0;
  }
```

xImHead holds the x-coordinate in the panel where the left side of the image (its *head*) should be drawn.

The isMovingRight and isMovingLeft flags determine the direction of movement for the Ribbon image (or whether it is stationary) when its JPanel position is updated. The flags are set by the moveRight(), moveLeft() and stayStill() methods. For example:

```
  public void moveRight()
  // move the ribbon image to the right on the next update
  { isMovingRight = true;
    isMovingLeft = false;
  }
```

update() adjusts the xImHead value depending on the movement flags. xImHead can range between -width to width (the width of the image).

```
  public void update()
  { if (isMovingRight)
      xImHead = (xImHead + moveSize) % width;
    else if (isMovingLeft)
      xImHead = (xImHead - moveSize) % width;
  }
```

As xImHead varies, the drawing of the ribbon in the JPanel will usually be a combination of the image's tail followed by its head.

6.1. Drawing the Ribbon's Image

The display() method does the hard work of deciding where various bits of the image should be drawn in the JPanel.

One of the hard aspects of display() is that it utilises two different coordinate systems: JPanel coordinates and image coordinates. This can be seen in the many calls to Graphics' drawImage() method:

```
boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2,
                             int sx1, int sy1, int sx2, int sy2,
                                   ImageObserver observer);
```

Figure 7 shows that the eight integers represent two regions – the destination JPanel and source image.

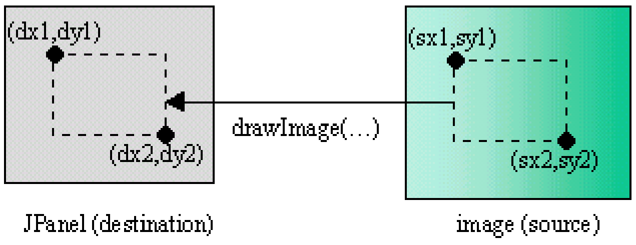

Figure 7. Drawing a Region with drawImage().

Fortunately, in Jumping Jack, the regions are always the same height, starting at the top edge of the JPanel (y == 0) and extending to its bottom (y == pHeight). However, dx1 and dx2 will vary in the JPanel, and sx1 and sx2 vary in the image.

The x-coordinates are derived from the current xImHead value, which ranges between width and –width as the image is shifted right or left across the JPanel.

Also, as the image moves right (or left), there will come a point when it will be necessary to draw both the head and tail of the image in order to cover the JPanel.

These considerations lead to display() consisting of five cases; we consider each one below.

```
public void display(Graphics g)
{
  if (xImHead == 0)   // draw im start at (0,0)
    draw(g, im, 0, pWidth, 0, pWidth);
  else if ((xImHead > 0) && (xImHead < pWidth)) {
     // draw im tail at (0,0) and im start at (xImHead,0)
    draw(g, im, 0, xImHead, width-xImHead, width);   // im tail
    draw(g, im, xImHead, pWidth, 0, pWidth-xImHead);  // im start
  }
  else if (xImHead >= pWidth)   // only draw im tail at (0,0)
    draw(g, im, 0, pWidth,
                width-xImHead, width-xImHead+pWidth);  // im tail
  else if ((xImHead < 0) && (xImHead >= pWidth-width))
    draw(g, im, 0, pWidth, -xImHead, pWidth-xImHead);  // im body
  else if (xImHead < pWidth-width) {
     // draw im tail at (0,0) and im start at (width+xImHead,0)
    draw(g, im, 0, width+xImHead, -xImHead, width);  // im tail
    draw(g, im, width+xImHead, pWidth,
                0, pWidth-width-xImHead);  // im start
  }
} // end of display()


private void draw(Graphics g, BufferedImage im,
                      int scrX1, int scrX2, int imX1, int imX2)
/* The y-coords of the image always starts at 0 and ends at
   pHeight (the height of the panel), so are hardwired. */
{ g.drawImage(im, scrX1, 0, scrX2, pHeight,
                   imX1, 0,  imX2, pHeight, null);
}
```

6.1.1. Case 1. Draw the Image at JPanel (0,0)

The relevant code snippet from display():

```
  if (xImHead == 0)   // draw im start at (0,0)
    draw(g, im, 0, pWidth, 0, pWidth);
```

Figure 8 illustrates the drawing operation:

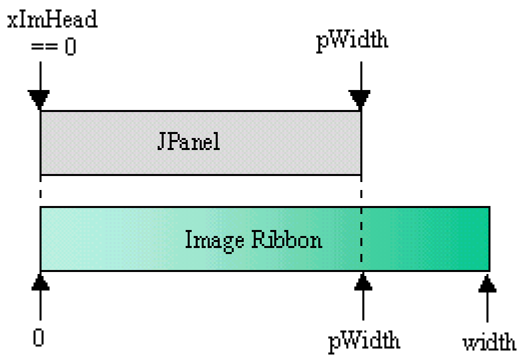

Figure 8. Case 1 in Ribbon's display().

Case 1 occurs at start-up time, when the scene is first drawn, and reoccurs when Jack has run around the width of the image, and xImHead is back at 0.

draw() is a simplified interface to drawImage(), hiding the fixed y-coordinates (0 to pHeight). Its third and fourth arguments are the x-coordinates in the JPanel (the positions pointed to in the gray box in Figure 8). The fifth and sixth arguments are the positions pointed to in the image ribbon (the green box).

6.1.2. Case 2. Image Moving Right, xImHead < pWidth

The code fragment from display():

```
  if ((xImHead > 0) && (xImHead < pWidth)) {
       // draw im tail at (0,0) and im head at (xImHead,0)
      draw(g, im, 0, xImHead, width-xImHead, width);   // im tail
      draw(g, im, xImHead, pWidth, 0, pWidth-xImHead);  // im head
    }
```

Figure 9 illustrates the drawing operations.

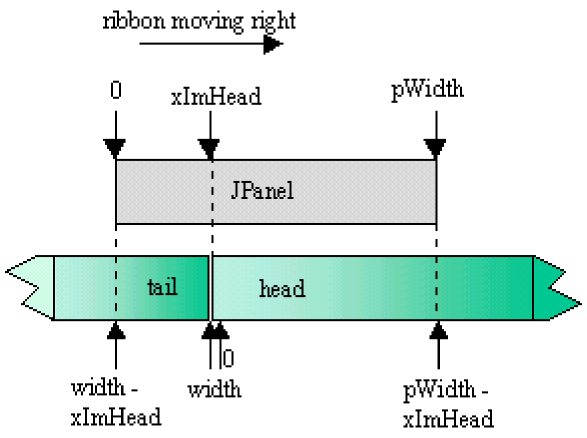

Figure 9. Case 2 in Ribbon's display().

When the image moves right (caused by the sprite apparently moving left), the JPanel drawing will require two drawImage() calls: one for the tail of the image, the other for the head (which still begins at xImHead in the JPanel).

The tricky part is calculating the x-coordinate of the start of the image's tail, and the x-coordinate of the end of the head.

6.1.3. Case 3. Image Moving Right, xImHead >= pWidth

The relevant piece of code:

```
    if (xImHead >= pWidth)   // only draw im tail at (0,0)
      draw(g, im, 0, pWidth, width-xImHead, width-xImHead+pWidth);
```

Figure 10 shows the drawing operation.

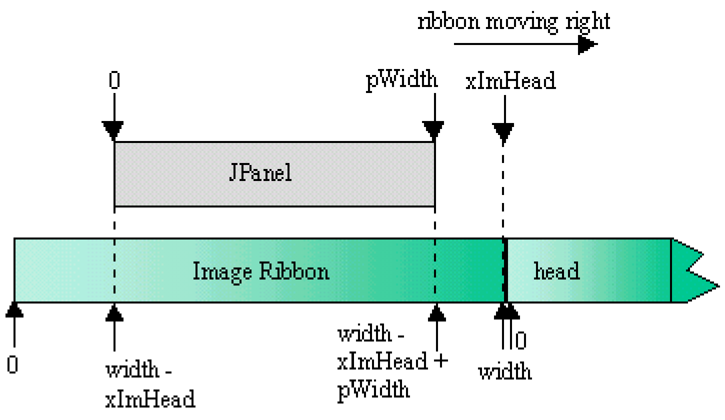

Figure 10. Case 3 in Ribbon's display().

Case 3 happens after case 2, as the image moves even further to the right, and xImHead travels beyond the right edge of the JPanel. This means that only a single drawImage() call is necessary, to draw the middle part of the image into the JPanel. The tricky x-coordinates are the start and end points for the image's middle.

6.1.4. Case 4. Image Moving Left, xImHead >= pWidth-width

The code snippet:

```
  if ((xImHead < 0) && (xImHead >= pWidth-width))
      draw(g, im, 0, pWidth, -xImHead, pWidth-xImHead);  // im body
```

Figure 11 illustrates the drawing operation.

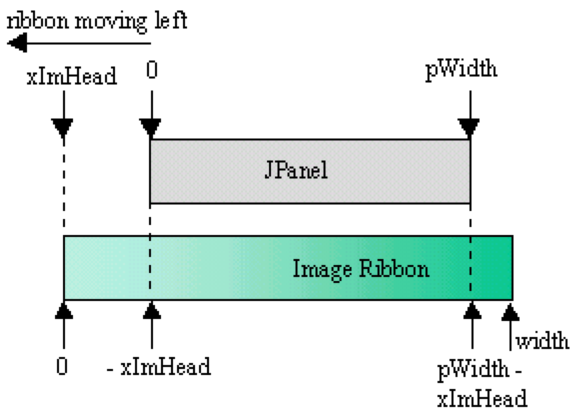

Figure 11. Case 4 in Ribbon's display().

Case 4 occurs when the image is moving left, which happens when the sprite apparently travels to the right. xImHead will become negative, since it is to the left of JPanel's origin. While it is still greater than pWidth-width, only a single drawImage() is needed, for the middle of the image.

6.1.5. Case 5. Image Moving Left, xImHead < pWidth-width

The code:

```
if (xImHead < pWidth-width) {
   // draw im tail at (0,0) and im head at (width+xImHead,0)
  draw(g, im, 0, width+xImHead, -xImHead, width);  // im tail
  draw(g, im, width+xImHead, pWidth,
              0, pWidth-width-xImHead);  // im head
}
```

Figure 12 shows the drawing operations.

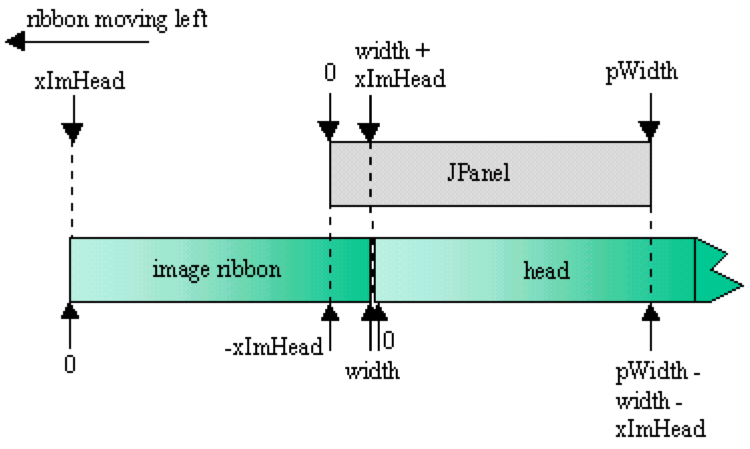

Figure 12. Case 5 in Ribbon's display().

Case 5 occurs after case 4, when the image has moved further to the left, and xImHead is smaller than pWidth-width. This distance marks the point at which two drawImage() calls are required, one for the tail of the image, the other for its head.

7. The BricksManager Class

BricksManager is separated into five broad groups of methods for:

- loading bricks information;
- initialising the bricks data structures;
- moving the bricks map;
- drawing the bricks;
- JumperSprite-related tasks. These are mostly various forms of collision detection between the sprite and the bricks.

BricksManager reads a bricks map, and creates a Brick object for each brick. The data structure holding the Brick objects is optimised so that drawing and collision detection can be carried out quickly.

Moving and drawing the bricks map is analogous to the moving and drawing of an image by a Ribbon object. However, the drawing process is complicated by the ribbon being made up of multiple bricks instead of a single GIF.

Jack, the JumperSprite object, uses BricksManager methods to determine if its planned moves will cause it to collide with a brick.

7.1. Loading Bricks Information

BricksManager calls loadBricksFile() to load a bricks map; in our code, the map is assumed to be in bricksInfo.txt from /Images. The map starts with the name of an image strip which holds the images referred to in the map. The bricks map follows: each line continues a mix of single digits or spaces. Up to a maximum of MAX_BRICKS_LINES lines are processed.

The map file may contain empty lines and comment lines (those starting with //), which are ignored.

bricksInfo.txt is:

```
// bricks information

s tiles.gif 5

// -----------
44444
                                  222222222
                                111
                                2222
                             11111
                          444
                          444
        22222          444                      111
     1111112222222     23333  2     33      44444444
00 0001113333330000002222222333333  333 2222222222333301
00000000011100000000002220000000003300000111111222222234
// -----------
```

The images strip in tiles.gif is shown in Figure 13.

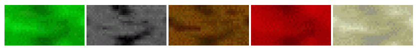
Figure 13. The Images Strip in tiles.gif.

The images strip is loaded with an ImagesLoader object, and the array of BufferedImages are stored in a global array called brickImages[].

There are several drawbacks with this approach. One is the reliance on single digits to index into the images strip. This makes it impossible to utilize strips with more than 10 images (0-9), which is inadequate for a real map. The solution probably entails moving to a letter-based scheme (A-Z, a-z), to allow up to 52 tiles.

loadBricksFile() calls storeBricks() to read in a single map line, adding Brick objects to a bricksList ArrayList.

```
private void storeBricks(String line, int lineNo, int numImages)
{
  int imageID;
  for(int x=0; x < line.length(); x++) {
    char ch = line.charAt(x);
    if (ch == ' ')   // ignore a space
      continue;
    if (Character.isDigit(ch)) {
      imageID = ch - '0';    // we assume a digit is 0-9
      if (imageID >= numImages)
        System.out.println("Image ID "+imageID+" out of range");
      else   // make a Brick object
        bricksList.add( new Brick(imageID, x, lineNo) );
    }
    else
      System.out.println("Brick char " + ch + " is not a digit");
  }
}
```

A Brick object is initialised with its image ID (a number in the range 0-9); a reference to the actual image is added later. The brick is also passed its map indices (x, lineNo). lineNo starts at 0 when the first map line is read, and incremented with each new line.

Figure 14 shows some of the important variables associated with a map, including example map indices.

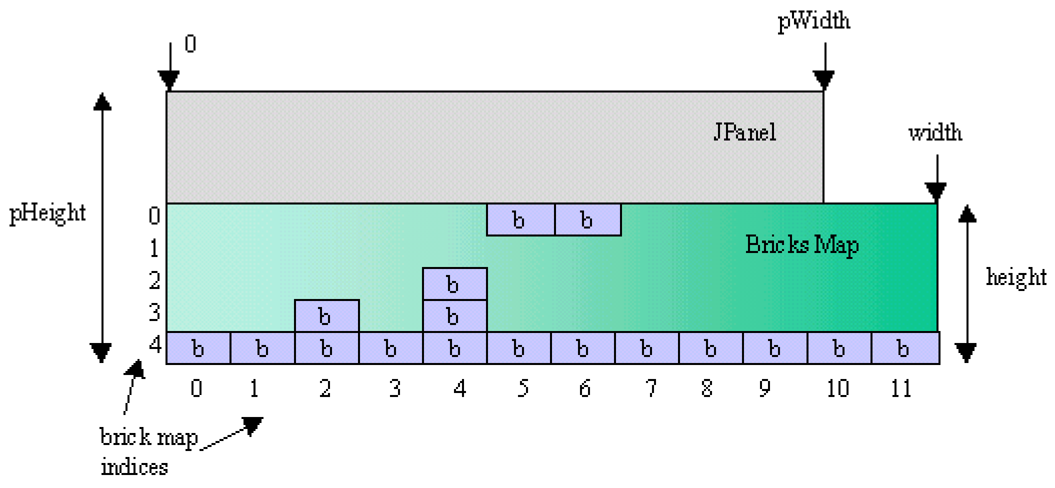

Figure 14. Brick Map Variables.

7.2. Initialising the Bricks Data Structures

Once the bricksList ArrayList has been filled, BricksManager calls initBricksInfo() to extract various global data from the list, and check if certain criteria are met. For instance, the maximum width of the map should be greater than the width of the panel (width >= pWidth).

Another criteria for accepting the map is that its bottom row must not have any gaps. This makes it impossible for Jack to fall down a hole while running about, so simplifying the JumperSprite implementation.

The bricksList ArrayList doesn't store its Brick objects in order, which makes finding a particular Brick very time-consuming. Unfortunately, searching for a brick is a common task – it must be performed every time that Jack is about to move, to prevent it from hitting something.

A more useful way of storing the bricks map is ordered by column, as illustrated in Figure 15.

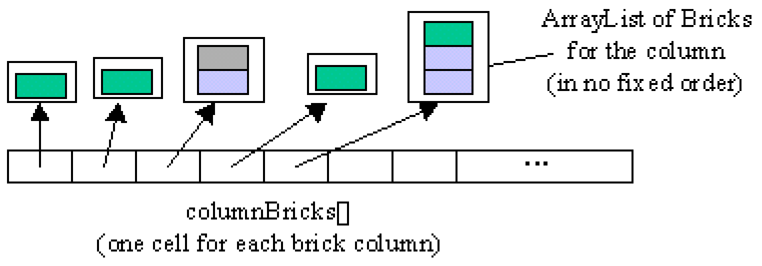
ArrayList of Bricks for the column (in no fixed order)
...
columnBricks[]
(one cell for each brick column)

Figure 15. Bricks Stored by Column.

This data structure is excellent for brick searches where the column of interest is known beforehand, since the array allows constant-time access to a given column.

A column is implemented as an ArrayList of Bricks in no particular order, so a linear search looks for a brick in the selected column. However, a column contains very few bricks compared to the entire map, so the search time is acceptable.

Also, since there are no gaps in the bottom row of the map, each column must contain at least one brick, guaranteeing that none of the column ArrayLists in columnBricks[] is null.

The columnBricks[] array is built by BricksManager calling createColumns().

7.3. Moving the Bricks Map

The BricksManager uses the same approach to moving its bricks map as the Ribbon class does for its GIF.

The isMovingRight and isMovingLeft flags determine the direction of movement for the bricks map (or whether it is stationary) when its JPanel position is updated. The flags are set by the moveRight(), moveLeft() and stayStill() methods. For example:

```
public void moveRight()
{ isMovingRight = true;
  isMovingLeft = false;
}
```

update() increments a xMapHead value depending on the movement flags. xMapHead is the x-coordinate in the panel where the left edge of the bricks map (its head) should be drawn. xMapHead can range between -width to width (the width of the bricks map in pixels).

```
public void update()
{ if (isMovingRight)
    xMapHead = (xMapHead + moveSize) % width;
  else if (isMovingLeft)
    xMapHead = (xMapHead - moveSize) % width;
}
```

7.4. Drawing the Bricks

The display() method does the hard work of deciding where the bricks in the map should be drawn in the JPanel.

As in the Ribbon class, several different coordinate systems are combined: the JPanel coordinates and the bricks map coordinates. The bad news is that the bricks map use two different schemes. One way of locating a brick is by its pixel position in the bricks map, the other is by using its map indices (see Figure 14). This means that there are *three* coordinate systems utilised in display() and its helper method drawBricks().

```
public void display(Graphics g)
{
  int bCoord = (int)(xMapHead/imWidth) * imWidth;
```

```
    // bCoord is the drawing x-coord of the brick containing xMapHead
    int offset;    // offset is distance between bCoord and xMapHead
    if (bCoord >= 0)
      offset = xMapHead - bCoord;   // offset is positive
    else  // negative position
      offset = bCoord - xMapHead;   // offset is positive

    if ((bCoord >= 0) && (bCoord < pWidth)) {
      drawBricks(g, 0-(imWidth-offset), xMapHead,
                        width-bCoord-imWidth);   // bm tail
      drawBricks(g, xMapHead, pWidth, 0);  // bm start
    }
    else if (bCoord >= pWidth)
      drawBricks(g, 0-(imWidth-offset), pWidth,
                       width-bCoord-imWidth);  // bm tail
    else if ((bCoord < 0) && (bCoord >= pWidth-width+imWidth))
      drawBricks(g, 0-offset, pWidth, -bCoord);       // bm tail
    else if (bCoord < pWidth-width+imWidth) {
      drawBricks(g, 0-offset, width+xMapHead, -bCoord);  // bm tail
      drawBricks(g, width+xMapHead, pWidth, 0);     // bm start
    }
  } // end of display()
```

The details of drawBricks() will be explained below. For now, it's enough to know the meaning of its prototype:

```
  void drawBricks(Graphics g, int xStart, int xEnd, int xBrick);
```

drawBricks() draws bricks into the JPanel starting at xStart, ending at xEnd. The bricks are drawn a column at a time. The first column of bricks is the one at the xBrick pixel x-coordinate in the bricks map.

display() starts by calculating a brick coordinate (bCoord) and offset from the xMapHead position. These are used in the calls to drawBricks() to specify where a brick image's left edge should appear. This should hopefully become clearer as we consider the four drawing cases.

7.4.1. Case 1. Bricks Map Moving Right and bCoord < pWidth

The relevant code snippet in display():

```
if ((bCoord >= 0) && (bCoord < pWidth)) {
  drawBricks(g, 0-(imWidth-offset), xMapHead,
                       width-bCoord-imWidth);   // bm tail
  drawBricks(g, xMapHead, pWidth, 0);  // bm start
}  // bm means bricks map
```

Figure 16 illustrates the drawing operations:

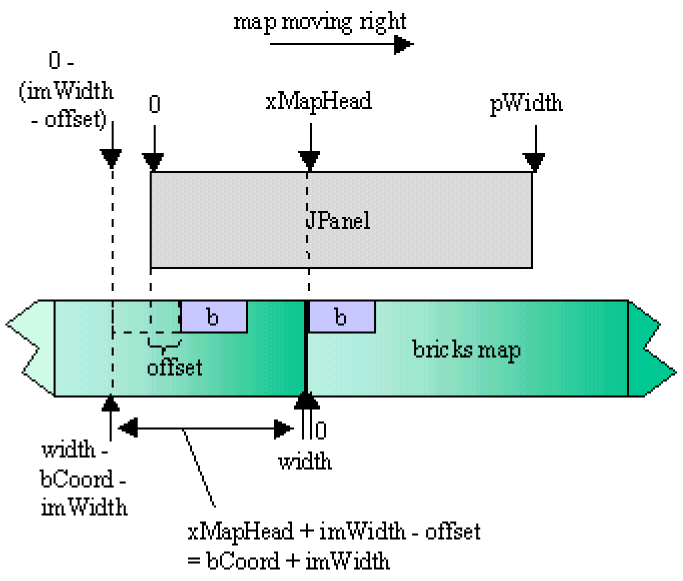

Figure 16. Case 1 in BricksManager's display().

Case 1 occurs as the bricks map moves right, since the sprite is apparently moving left. xMapHead will have a value between 0 and pWidth (the width of the JPanel). Two groups of bricks will need to be drawn, requiring two calls to drawBricks(). The first group starts *near* the left edge of the JPanel, the second starts at the xMapHead position. It is the first group that poses the problem.

A complete column of bricks must be drawn, and this requires the x-coordinate of the column's left edge. Depending on the xMapHead value, the left edge of the column probably does not line up with the left edge of the panel, most likely occuring somewhere to its left, off screen.

The calculation of the column's left edge requires the offset value calculated in display() before the if tests were entered. The resulting coordinate can also be expressed using bCoord.

A drawBricks() call takes three integer arguments. The first two are the start and end coordinates in the JPanel for a set of columns, which correspond to the values pointed to in the gray JPanel box in Figure 16. The third argument is the coordinate of the left edge of the bricks column in the bricks map, which is the value pointed to in the green rectangle in Figure 16.

7.4.2. Case 2. Bricks Map Moving Right and bCoord >= pWidth

The code piece:

```
if (bCoord >= pWidth)
  drawBricks(g, 0-(imWidth-offset), pWidth,
                         width-bCoord-imWidth);  // bm tail
```

Figure 17 shows the operation:

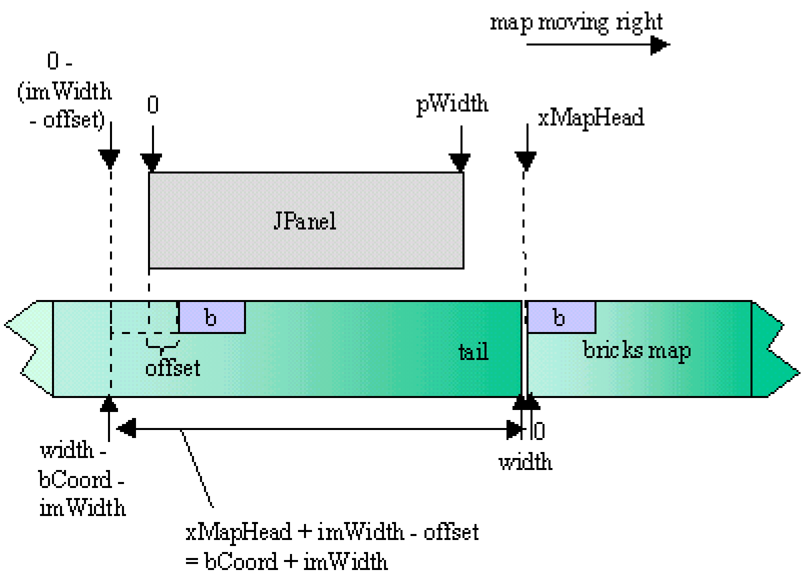

Figure 17. Case 2 in BricksManager's display().

Case 2 happens some time after case 1, when xMapHead has moved further right, beyond the right edge of the JPanel. The drawing task becomes simpler, since only a single call to drawBricks() is required to draw a group of columns taken from the middle of the bricks map.

Case 2 has the same problem as case 1 in determining the position of the first column's left edge, which is solved using the offset and bCoord values.

7.4.3. Case 3. Bricks Map Moving Left and bCoord >= pWidth-width+imWidth

The relevant code fragment:

```
if ((bCoord < 0) && (bCoord >= pWidth-width+imWidth))
  drawBricks(g, 0-offset, pWidth, -bCoord);        // bm tail
```

Figure 18 illustrates the drawing operation:

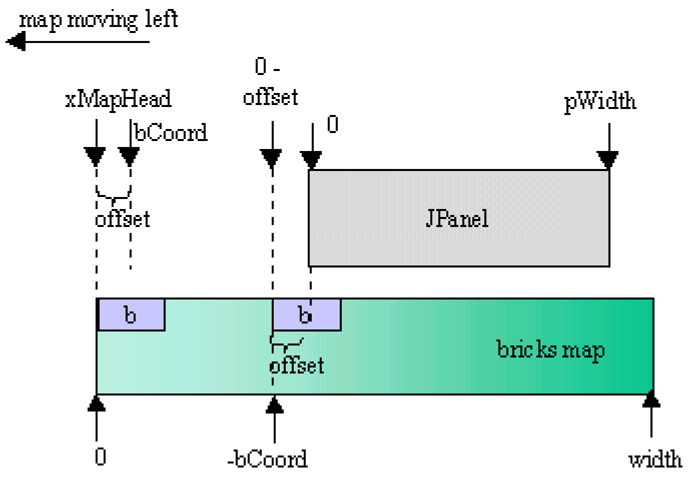

Figure 18. Case 3 in BricksManager's display().

Case 3 applies when the bricks map is moving left, as the sprite is supposedly travelling to the right. xMapHead goes negative, as does bCoord, but the calculated offset is adjusted to be positive.

Until bCoord drops below pWidth-width+imWidth, the bricks map will only require a single drawBricks() call to fill the JPanel.

7.4.4. Case 4. Bricks Map Moving Left and bCoord < pWidth-width+imWidth

The code:

```
if (bCoord < pWidth-width+imWidth) {
  drawBricks(g, 0-offset, width+xMapHead, -bCoord);  // bm tail
  drawBricks(g, width+xMapHead, pWidth, 0);     // bm start
}
```

Figure 19 shows the operations:

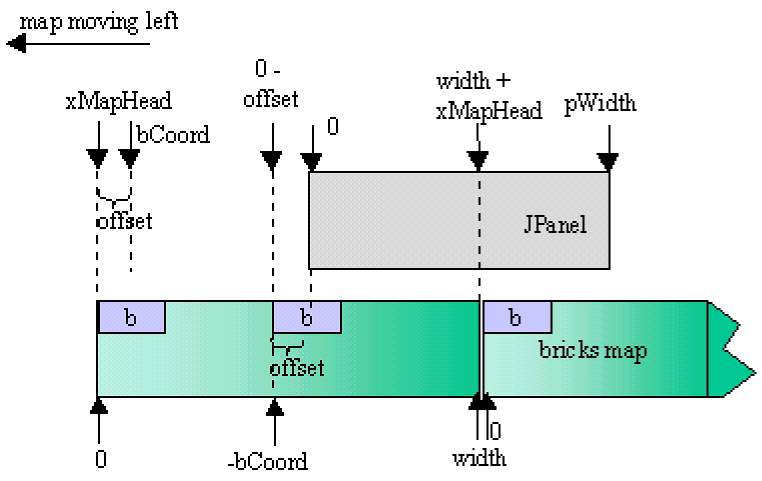

Figure 19. Case 4 in BricksManager's display().

Case 4 occurs after xMapHead has moved to the left of pWidth-width+imWidth. Two drawBricks() calls are needed to render two groups of columns to the JPanel.

7.4.5. The drawBricks() Method

drawBricks() draws bricks into the JPanel between xStart and xEnd. The bricks are drawn a column at a time, separated by imWidth pixels. The first column of bricks drawn is the one at the xBrick pixel x-coordinate in the bricks map.

```
private void drawBricks(Graphics g, int xStart, int xEnd,
                                                int xBrick)
{ int xMap = xBrick/imWidth;    // get column position of the brick
                                // in the bricks map
  ArrayList column;
  Brick b;
  for (int x = xStart; x < xEnd; x += imWidth) {
    column = columnBricks[ xMap ];   // get the current column
    for (int i=0; i < column.size(); i++) {   // draw all bricks
       b = (Brick) column.get(i);
       b.display(g, x);   // draw brick b at JPanel posn x
    }
    xMap++;  // examine the next column of bricks
  }
}
```

drawBricks() converts the xBrick value, a pixel x-coordinate in the bricks map, into a map x index. This index is the column position of the brick, and so the entire column can be accessed immediately in columnBricks[]. The bricks in the column are drawn by calling the display() method for each brick.

Only the JPanel's x-coordinate is passed to display(), the y-coordinate is already stored in the Brick object. This is possible since a brick's y-axis position never changes as the bricks map is moved horizontally over the JPanel.

7.5. JumperSprite-related Methods

The BricksManager has several public methods used by JumperSprite to determine or check its position in the bricks map. The prototypes of the complicated methods are:

- `int findFloor(int xSprite);`
- `boolean insideBrick(int xWorld, int yWorld);`
- `int checkBrickBase(int xWorld, int yWorld, int step);`
- `int checkBrickTop(int xWorld, int yWorld, int step);`

7.5.1. Finding the Floor

When Jack is first added to the scene, his x-coordinate is in the middle of the JPanel, but what should his y-coordinate be? He should be placed on the top-most brick at, or near, the given x-coordinate. findFloor() searches for this brick, returning its y-coordinate.

```
public int findFloor(int xSprite)
{
  int xMap = (int)(xSprite/imWidth);   // x map index

  int locY = pHeight;    // starting y pos (largest possible)
  ArrayList column = columnBricks[ xMap ];
  Brick b;
  for (int i=0; i < column.size(); i++) {
    b = (Brick) column.get(i);
    if (b.getLocY() < locY)
      locY = b.getLocY();   // reduce locY (i.e. move up)
  }
  return locY;
}
```

Matters are simplified by the timing of the call – findFloor() is invoked before the sprite has moved, so before the bricks map has moved. Consequently, the sprite’s x-coordinate in the JPanel (xSprite) is the same x-coordinate in the bricks map.

xSprite is converted to a map x index, to permit the relevant column of bricks to be accessed in columnBricks[].

7.5.2. Testing for Brick Collision

JumperSprite implements collision detection by calculating its new position after a proposed move, and testing whether that point (xWorld, yWorld) is inside a brick. If it is, then the move is aborted, and the sprite stops moving.

The point testing is done by BricksManager's insideBrick(), which uses worldToMap () to convert the sprite’s coordinate to a brick map index tuple.

```
public boolean insideBrick(int xWorld, int yWorld)
// Check if the world coord is inside a brick
{
  Point mapCoord = worldToMap(xWorld, yWorld);
  ArrayList column = columnBricks[ mapCoord.x ];
```

```
    Brick b;
    for (int i=0; i < column.size(); i++) {
      b = (Brick) column.get(i);
      if (mapCoord.y == b.getMapY())
        return true;
    }
    return false;
  }  // end of insideBrick()
```

worldToMap() returns a Point object holding the x and y map indices corresponding to (xWorld,yWorld). The relevant brick column in columnBricks[] can then be searched for a brick at the y map position.

The conversion carried out by worldToMap() can be understood by referring to Figure 14, which we repeat again here as Figure 20.

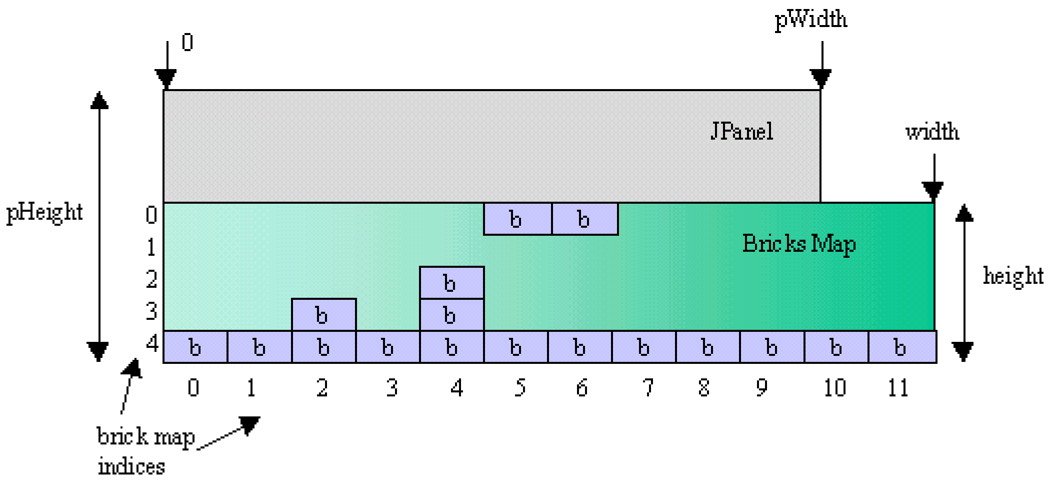

Figure 20. Brick Map Variables.

The code:

```
  private Point worldToMap(int xWorld, int yWorld)
  // convert world coord (x,y) to a map index tuple
  {
    xWorld = xWorld % width;    // limit to range (width to -width)
    if (xWorld < 0)             // make positive
      xWorld += width;
    int mapX = (int) (xWorld/imWidth);    // map x-index

    yWorld = yWorld - (pHeight-height);  // relative to map
    int mapY = (int) (yWorld/imHeight);  // map y-index

    if (yWorld < 0)    // above the top of the bricks
      mapY = mapY-1;  // match to next 'row' up

    return new Point(mapX, mapY);
  }
```

xWorld can be any positive or negative value, so must be restricted to the range 0-width (the extent of the bricks map). The coordinate is then converted to a map a index.

The yWorld value uses the JPanel's coordinate system, so is made relative to the y-origin of the bricks map (some distance down from the top of the JPanel). The conversion to a map y index must take into account the possibility that the sprite's position is above the top of the bricks map. This can occur by having the sprite jump upwards while standing on a platform at the top of the bricks map.

7.5.3. Jumping and Hitting Your Head

When Jack jumps, his progress upwards will be halted if he about to pass through the base of a brick. The idea is shown in Figure 21.

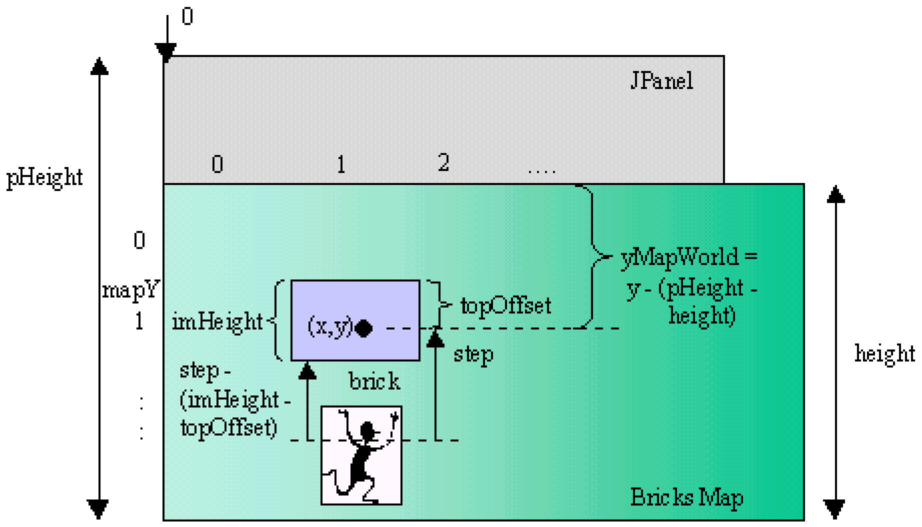

Figure 21. A Rising Sprite Hitting a Brick

The sprite hopes to move upwards by a step amount, but this will cause it to enter the brick. Instead it will travel upwards by a smaller step, step-(imHeight-topOffset), placing its top edge next to the bottom edge of the brick.

checkBrickBase() is supplied with the planned new position (xWorld, yWorld), labelled as (x,y) in Figure 21, and the step. It returns the step distance that the sprite can move without passing into a brick.

```
public int checkBrickBase(int xWorld, int yWorld, int step)
{
  if (insideBrick(xWorld, yWorld)) {
    int yMapWorld = yWorld - (pHeight-height);
    int mapY = (int) (yMapWorld/imHeight);  // map y- index
    int topOffset = yMapWorld - (mapY * imHeight);
    return (step - (imHeight-topOffset));  // a smaller step
  }
  return step;   // no change
}
```

7.5.4. Falling and Sinking into the Ground

As a sprite descends, during a jump or after walking off the edge of a raised platform, it must test its next position to ensure that it doesn't pass through a brick on its way down. When a brick is detected beneath the sprite's feet, the descent is stopped so it lands on top of the brick. Figure 22 illustrates the calculation.

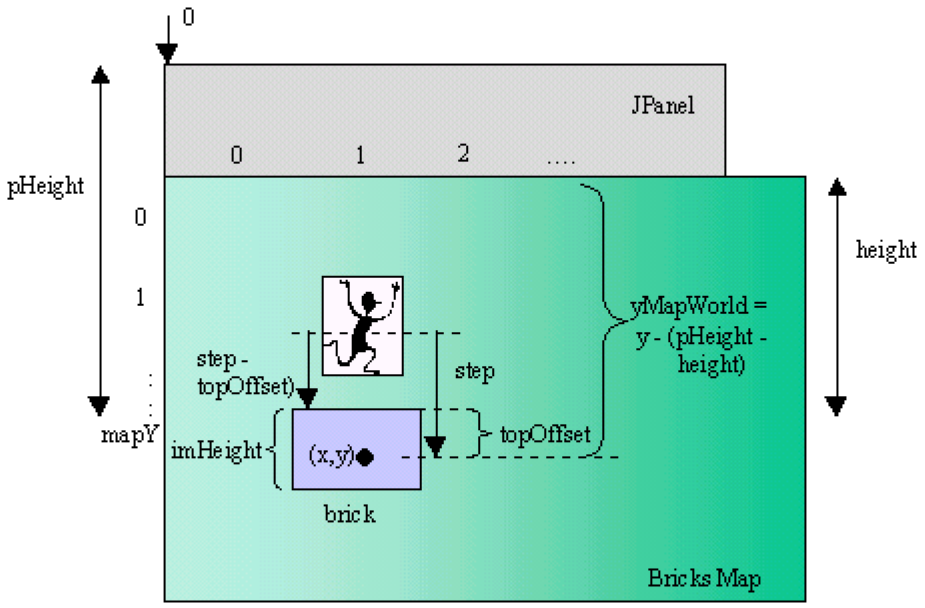

Figure 22. A Falling Sprite Hitting a Brick.

The sprite moves downwards by a step amount on each update, but when a collision is detected, the step size is reduced to step-topOffset, so it comes to rest on top of the brick.

```
public int checkBrickTop(int xWorld, int yWorld, int step)
{
  if (insideBrick(xWorld, yWorld)) {
    int yMapWorld = yWorld - (pHeight-height);
    int mapY = (int) (yMapWorld/imHeight);  // map y- index
    int topOffset = yMapWorld - (mapY * imHeight);
    return (step - topOffset);    // a smaller step
  }
  return step;   // no change
}
```

The intended new position for the sprite (xWorld,yWorld) is passed to checkBrickTop (), along with the step size. The returned value is the step that the sprite should take to avoid sinking into a brick.

8. The Brick Class

The Brick class stores coordinate information for a brick, and a reference to its image.

The coordinate details are the brick's map indices, and its y-axis pixel position inside the map. The x-axis position isn't stored since it changes as the bricks map is moved horizontally.

Brick's display() method is very simple:

```
  public void display(Graphics g, int xScr)
  // called by BricksManager's drawBricks()
  {  g.drawImage(image, xScr, locY, null);  }
```

xScr is the current JPanel x coordinate for the brick.

The capabilities of the Brick class could easily be extended. One common feature in side-scrollers are animated tiles, such as flames and rotating balls. If the animation is local to the tile's allocated map location, then the effect can be coded by adding an ImagesPlayer to Brick. One issue is whether to assign a unique ImagesPlayer to each Brick (costly if there are many bricks), or store a reference to a single ImagesPlayer. The drawback with the reference solution is that all the animated bricks be the same.

Probably the best solution is to create an AnimatedBrick subclass, which will be used rarely, and so can support the overhead of having its own ImagesPlayer.

If tiles can move about in the game world (e.g. a platform that moves up and down), then bricks will need more sprite-like capabilities. This will also complicate BricksManager since a Brick object can no longer be relied on to stay in the same column.

9. The FireBallSprite Class

A fireball starts at the lower right hand side of the panel, and travels across to the left. If it hits Jack, the fireball explodes, and a suitable sound is heard. A fireball that has traveled off the left hand side of the panel, or exploded, is reused.

Only a single fireball is on-screen at a time, so JumpingJack only creates a single FireBallSprite object. It is declared in JackPanel's constructor:

```
  fireball = new FireBallSprite(PWIDTH, PHEIGHT, imsLoader,
                                                 this, jack);
```

The fourth argument is a reference to JackPanel, the fifth argument allows the fireball to communicate with Jack.

As the fireball moves left, it keeps checking whether it has hit Jack. If a collision occurs, JackPanel is asked to display an explosion, while FireBallSprite resets its position.

9.1. Statechart Specification

The statechart in Figure 23 is a useful way of specifying the design needs of FireBallSprite. Statecharts were introduced in chapter 6.

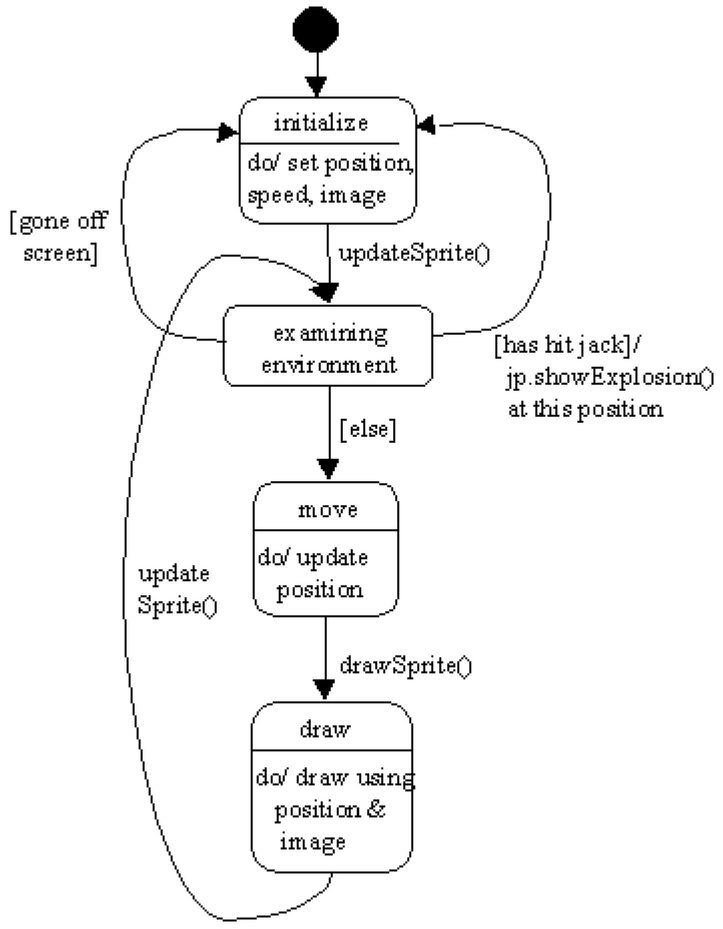

Figure 23. The FireBallSprite Statechart.

The update/draw cycle driven by JackPanel's animation loop is clearly visible. There are two special cases to consider – when the fireball hits Jack, and when it leaves the left side of the panel.

The "examining environment" and "move" states are represented by updateSprite():

```
public void updateSprite()
{ hasHitJack();
  goneOffScreen();
  super.updateSprite();
}


private void hasHitJack()
/* If the ball has hit jack, tell JackPanel (which will
   display an explosion and play a clip), and begin again.
*/
{ Rectangle jackBox = jack.getMyRectangle();
  jackBox.grow(-jackBox.width/3, 0);   // make bounded box thinner
```

```
    if (jackBox.intersects( getMyRectangle() )) {    // collision?
      jp.showExplosion(locx, locy+getHeight()/2);
             // tell JackPanel, supplying it with a hit coordinate
      initPosition();
    }
  } // end of hasHitJack()


  private void goneOffScreen()
  {
    if (((locx+getWidth()) <= 0) && (dx < 0)) // gone off left
      initPosition();   // start the ball in a new position
  }
```

Collision detection (the [has hit jack]) condition in the statechart), is carried out by obtaining Jack's bounded box, and checking if it intersects the bounded box for the fireball. Jack's bounded box width is reduced a little, to trigger a collision only when the fireball is right on top of him.

The move state is dealt with by Sprite's updateSprite(), which is called from FireBallSprite's updateSprite().

The draw state is implemented by Sprite's drawSprite() method.

10. The JumperSprite Class

A JumperSprite object can appear to move left or right, jump, and stand still. In fact, the sprite doesn't move horizontally at all, but the left and right movement requests will affect its internal state. It maintains its current world coordinates in (xWorld, yWorld).

When a sprite starts moving left or right, it will keep travelling in that direction until stopped by a brick. If the sprite runs off a raised platform, it will fall to the ground below, and continue moving forward.

When the sprite jumps, it continues upwards for a certain distance, then falls back to the ground. The upwards trajectory is stopped if the sprite hits a brick.

10.1. Statechart Specification

The JumperSprite statechart is given in Figure 24.

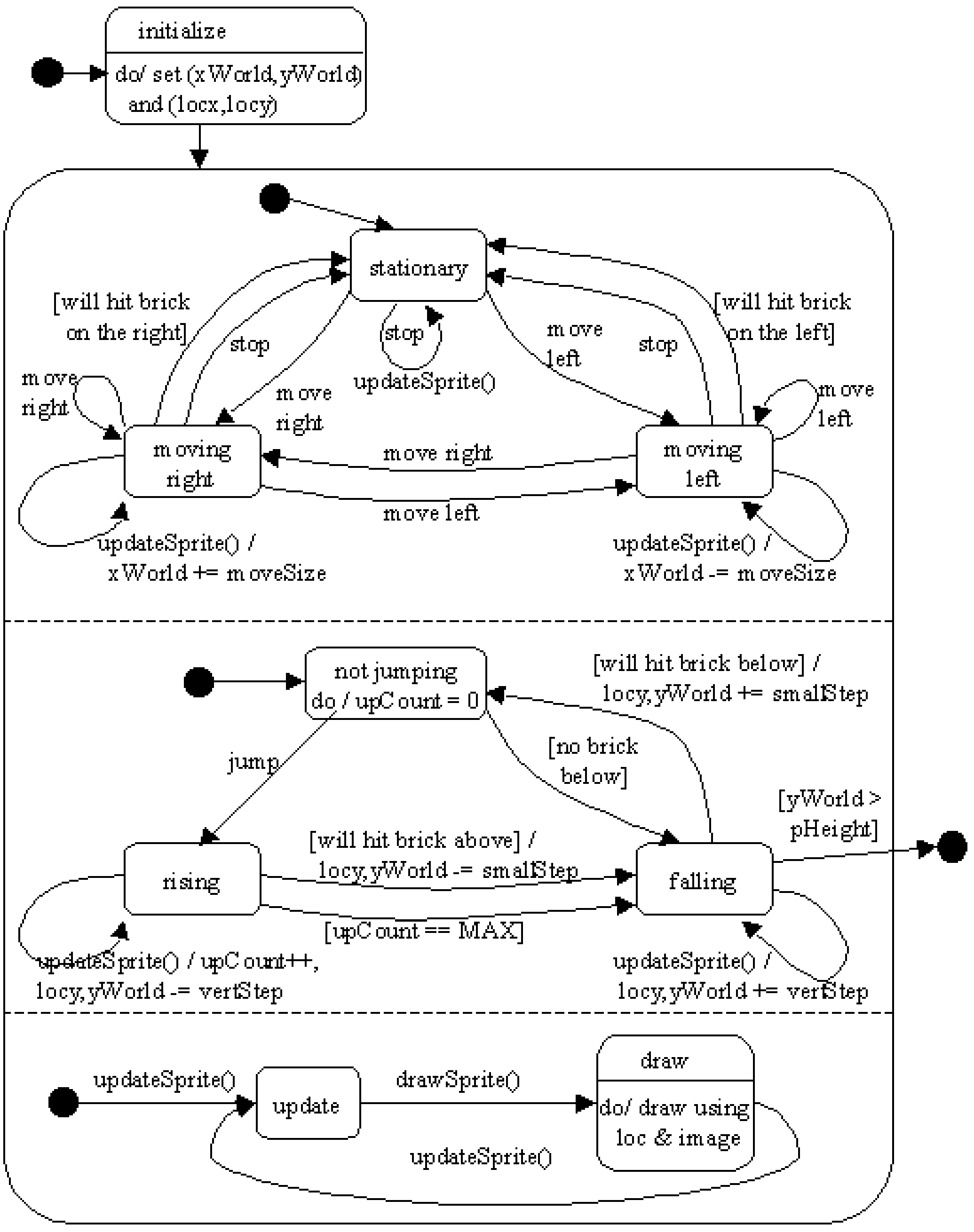
initialize
do/ set (xWorld,yWorld) and (locx,locy)
stationary
[will hit brick on the right]
stop
stop
updateSprite()
move right
move left
stop
[will hit brick on the left]
move right
moving right
move right
moving left
move left
move left
updateSprite() / xWorld += moveSize
updateSprite() / xWorld -= moveSize
not jumping
do / upCount = 0
[will hit brick below] / locy,yWorld += smallStep
[no brick below]
jump
[yWorld > pHeight]
rising
[will hit brick above] / locy,yWorld -= smallStep
falling
[upCount == MAX]
updateSprite() / upCount++, locy,yWorld -= vertStep
updateSprite() / locy,yWorld += vertStep
updateSprite()
update
drawSprite()
draw
do/ draw using loc & image
updateSprite()

Figure 24. The JumperSprite Statechart.

The statechart models JumperSprite as three concurrent activities: its horizontal movement in the top section, its vertical movement in the middle section, and the update/draw cycle in the bottom section.

The effects of an updateSprite() event have been distributed through the diagram, rather than placing them together in an "examining environment" state.

The horizontal movement section shows that a new updateSprite() event does not change the current state, be it "moving right", "moving left", or "stationary".

Movement stops either when the user sends a stop event, or when the sprite hits a brick.

The vertical movement section utilises three states: "not jumping", "rising", and "falling". Rising is controlled by an upCount counter, which limits how long an upward move can last. Rising may also be stopped by the sprite hitting a brick. Falling is triggered when rising finishes, and when there is no brick underneath the sprite. This latter condition becomes true when the sprite moves horizontally off a raised platform.

The "falling" state can lead to termination if the sprite drops below the bottom of the panel [yWorld > pHeight]. In fact, this transition led to a redesign of BricksManager to reject a bricks map with a gap in its floor. Consequently, 'dropping off the panel' cannot occur in JumpingJack.

Although the statechart is very clear, we want to avoid the complexity of multiple threads in JumperSprite. Instead, the concurrent activities are interleaved together in the code, making it somewhat harder to understand.

10.2. Representing the States

The "moving right", "moving left", and "stationary" states are represented indirectly as two booleans, isFacingRight and isStill, which combine to define the current horizontal state. For instance, when isStill is false and isFacingRight is true, then the sprite is moving right.

The "not jumping", "rising", and "falling" states are encoded as constants, assigned to a vertMoveMode variable.

```
private static final int NOT_JUMPING = 0;
private static final int RISING = 1;
private static final int FALLING = 2;

private int vertMoveMode;
  /* can be NOT_JUMPING, RISING, or FALLING */

private boolean isFacingRight, isStill;
```

In J2SE 1.5, vertMoveMode could be defined using an enumerated type.

10.3. Initialization

The "initialize" state is coded in JumperSprite's constructor.

```
// some globals
private int vertStep;   // distance to move vertically in one step
private int upCount;

private int moveSize;   // obtained from BricksManager

private int xWorld, yWorld;
  /* the current position of the sprite in 'world' coordinates.
     The x-values may be negative. The y-values will be between
     0 and pHeight. */
```

```
        :


  public JumperSprite(int w, int h, int brickMvSz, BricksManager bm,
                                      ImagesLoader imsLd, int p)
  {
    super(w/2, h/2, w, h, imsLd, "runningRight");
       // standing center screen, facing right
    moveSize = brickMvSz;
          // the move size is the same as the bricks ribbon

    brickMan = bm;
    period = p;
    setStep(0,0);     // no movement

    isFacingRight = true;
    isStill = true;

    /* Adjust the sprite's y- position so it is
       standing on the brick at its mid x- position. */
    locy = brickMan.findFloor(locx+getWidth()/2)-getHeight();
    xWorld = locx; yWorld = locy;    // store current position

    vertMoveMode = NOT_JUMPING;
    vertStep = brickMan.getBrickHeight()/2;
                // the jump step is half a brick's height
    upCount = 0;
  }  // end of JumperSprite()
```

The (xWorld, yWorld) coordinates are set, as are the sprite's position and speed. The state variables isFacingRight, isStill, and vertMoveMode define a stationary, non-jumping sprite, facing to the right.

Other variables from the statechart (e.g. moveSize, vertStep) are initialized in the constructor.

BricksManager's findFloor() method is used to get a y location for the sprite that lets it stand on top of a brick.

10.4. Key Event Processing

The events "move left", "move right", "stop", and "jump" in the statechart are caught as key presses by the key listener in JackPanel, triggering calls to the JumperSprite methods moveLeft(), moveRight(), stayStill(), and jump().

moveLeft(), moveRight() and stayStill() affect the horizontal state by adjusting the isFacingRight and isStill variables. The animated image associated with the sprite is also changed.

```
  public void moveLeft()
  { setImage("runningLeft");
    loopImage(period, DURATION);   // cycle through the images
    isFacingRight = false;  isStill = false;
  }

  public void moveRight()
  { setImage("runningRight");
    loopImage(period, DURATION);   // cycle through the images
    isFacingRight = true;  isStill = false;
  }

  public void stayStill()
  { stopLooping();
    isStill = true;
  }
```

The jump() method represents the transition from the "not jumping" to the "rising" state in the statechart. This is coded by changing the value stored in vertMoveMode. The sprite's image is also modified.

```
  public void jump()
  { if (vertMoveMode == NOT_JUMPING) {
      vertMoveMode = RISING;
      upCount = 0;
      if (isStill) {    // only change image if the sprite is 'still'
        if (isFacingRight)
          setImage("jumpRight");
        else
          setImage("jumpLeft");
      }
    }
  }
```

10.5. JackPanel Collision Testing

The [will hit brick on the right] and [will it brick on the left] conditional transitions in the statechart are implemented as a public willHitBrick() method, called from JackPanel's gameUpdate() method.

```
  private void gameUpdate()
  {
    if (!isPaused && !gameOver) {
      if (jack.willHitBrick()) { // collision checking first
        jack.stayStill();    // stop everything moving
        bricksMan.stayStill();
        ribsMan.stayStill();
      }
      ribsMan.update();   // update background and sprites
      bricksMan.update();
      jack.updateSprite();
      fireball.updateSprite();
              :
    }
  }
```

The reason for placing the test in JackPanel's hands is so it can coordinate the other game entities when a collision occurs. Not only is the JumperSprite brought to a halt, so are the background layers in the game.

```
  public boolean willHitBrick()
  {
    if (isStill)
```

```
      return false;   // can't hit anything if not moving

    int xTest;   // for testing the new x- position
    if (isFacingRight)   // moving right
      xTest = xWorld + moveSize;
    else // moving left
      xTest = xWorld - moveSize;

    // test a point near the base of the sprite
    int xMid = xTest + getWidth()/2;
    int yMid = yWorld + (int)(getHeight()*0.8);   // use y posn

    return brickMan.insideBrick(xMid,yMid);
  }  // end of willHitBrick()
```

willHitBrick() represents two conditional transitions, so the isFacingRight flag is used to distinguish how xTest should be modified. The proposed new coordinate is generated, then passed to BricksManager's insideBrick() for evaluation.

The vertical collision testing in the middle section of the statechart, [will hit brick below] and [will hit brick above], are carried out by JumperSprite not JackPanel, since a collision only affects the sprite.

10.6. The updateSprite() Method

The statechart distributes the actions of the updateState() event around the statechart: actions are associated with the "moving right", "moving left", "rising", and "falling" states. These actions are implemented in the updateState() method, and the functions it calls:

```
  public void updateSprite()
  {
    if (!isStill) {    // moving
      if (isFacingRight)  // moving right
        xWorld += moveSize;
      else // moving left
        xWorld -= moveSize;
      if (vertMoveMode == NOT_JUMPING)   // if not jumping
        checkIfFalling();   // may have moved out into empty space
    }

    // vertical movement has two components: RISING and FALLING
    if (vertMoveMode == RISING)
      updateRising();
    else if (vertMoveMode == FALLING)
      updateFalling();

    super.updateSprite();
  }  // end of updateSprite()
```

The method updates its horizontal position (xWorld) first, distinguishing between moving right or left by examining isStill and isFacingRight.

After the move, checkIfFalling() decides whether the [no brick below] transition from "not jumping" to "falling" should be applied.

The third stage of the method is to update the vertical states.

Lastly, the call to Sprite's updateSprite() method modifies the sprite's position and image.

updateSprite() illustrates the sort of complications that arise when concurrent activities (horizontal and vertical movement) are combined and sequentialised. The horizontal actions are carried out before the vertical ones.

checkIfFalling() determines whether the "not jumping" state should be changed to "falling".

```
private void checkIfFalling()
{
  // could the sprite move downwards if it wanted to?
  // test its center x-coord, base y-coord
  int yTrans = brickMan.checkBrickTop( xWorld+(getWidth()/2),
                    yWorld+getHeight()+vertStep, vertStep);
  if (yTrans != 0)   // yes it could
    vertMoveMode = FALLING;   // set it to be in falling mode
}
```

The test is carried out by passing the coordinates of the sprite's feet, plus a vertical offset downwards, to checkBrickTop() in BricksManager.

10.7. Vertical Movement

updateRising() deals with the updateSprite() event associated with the "rising" state, and tests the two conditional transitions which leave the state: [upCount == MAX] and [will hit brick above]. Rising will continue until the maximum number of vertical steps is reached, or the sprite hits the base of a brick. The sprite then switches to falling mode. checkBrickBase() in BricksManager carries out the collision detection.

```
private void updateRising()
{ if (upCount == MAX_UP_STEPS) {
    vertMoveMode = FALLING;   // at top, now start falling
    upCount = 0;
  }
  else {
    int yTrans = brickMan.checkBrickBase(xWorld+(getWidth()/2),
                              yWorld-vertStep, vertStep);
    if (yTrans == 0) {   // hit the base of a brick
      vertMoveMode = FALLING;   // start falling
      upCount = 0;
    }
    else {   // can move upwards another step
      translate(0, -yTrans);
      yWorld -= yTrans;   // update position
      upCount++;
    }
  }
}  // end of updateRising()
```

updateFalling() processes the updateSprite() event associated with the "falling" state, and deals with the [will hit brick below] transition going to the "not jumping" state. checkBrickTop() in BricksManager carries out the collision detection.

The other conditional leading to termination is not implemented, since the bricks map cannot contain any holes for the sprite to fall through.

```
  private void updateFalling()
  { int yTrans = brickMan.checkBrickTop(xWorld+(getWidth()/2),
                         yWorld+getHeight()+vertStep, vertStep);
    if (yTrans == 0)   // hit the top of a brick
      finishJumping();
    else {    // can move downwards another step
      translate(0, yTrans);
      yWorld += yTrans;   // update position
    }
  }

  private void finishJumping()
  { vertMoveMode = NOT_JUMPING;
    upCount = 0;
    if (isStill) {    // change to running image, but not looping yet
      if (isFacingRight)
        setImage("runningRight");
      else    // facing left
        setImage("runningLeft");
    }
  }
```

11. Other Side-Scroller Examples

There are many areas where JumpingJack could be improved, including adding multiple levels, more 'bad guys' (enemy sprites), and complex tiles.

A good source of ideas are other side-scrolling games. ArcadePod.com (http://arcadepod.com/java/) lists 64 'scoller' games, although none of the ones I tried came with source code.

The following is a list of side-scrollers which do include source, and were written in the last 2-3 years.

- *MeatFighter: the Wiener Warrior* (http://www.meatfighter.com/). The Web site includes an article about the implementation, which appeared in *Java Developers Journal*, March 2003, Vol. 8, No. 3.
- *Frogma* (http://sourceforge.net/projects/frogma/).
- *VideoToons* (http://sourceforge.net/projects/videotoons/).
- *Mario Platformer* (http://www.paraduck.net/misterbob/Platformer1.1/classes/). Only the compiled classes are available for this applet.

Chapter 5 of David Brackeen's book is about a side-scroller (a 2D platform game):

> *Developing Games in Java*
> David Brackeen, Bret Barker, Laurence Vanhelswue
> New Riders, August 2003

The source can be obtained from http://www.brackeen.com/javagamebook/. He develops a wider range of bad guys than I have, and includes things for the hero to pick up.

A good place for articles about tile-based games is the "Isometric and Tile-based Games" reference section at *GameDev* (http://www.gamedev.net/reference/list.asp?categoryid=44).

12. J2ME and the Game API

MIDP 2.0 for J2ME has a Game API offering an easy-to-use animation loop, and a set of classes for building layers of tiles and sprites. Porting the API to J2SE would be quite straightforward.

Good online overviews of the API:

- Jonathan Knudsen, "Creating 2D Action Games with the Game API", `http://developers.sun.com/techtopics/mobility/midp/articles/game/` Knudsen's example is a tank game with the player's viewpoint looking downwards from directly overhead.
- Mikko Kontio, "MIDP 2.0: The Game API", `http://www.microjava.com/articles/techtalk/game_api?content_id=4271` The example is a blocky side-scroller.
- Carol Hamer, "MIDP 2.0 Games: a Step-by-Step Tutorial with Code Samples", `http://www.microjava.com/articles/techtalk/midp2_games` This lengthy tutorial builds up to a side-scroller involving a cowboy jumping over tumbleweed.

The only textbook that includes a Game API example (a side-scroller) is:

> *Wireless Java: Developing with J2ME*
> Jonathan Knudsen
> APress, 2003, 2nd edition

The source from the book can be obtained from http://www.apress.com/book/bookDisplay.html?bID=138. The side-scroller appears in chapter 11.

The Java Tiny Gfx Library (JTGL) (http://www.jtgl.org) provides a common set of graphics and gaming classes on top of J2ME, J2SE, and several mobile APIs. The gaming classes include Surface, TiledMap, and Sprite.

13. Tiling Software

One of the time-consuming aspects of side-scroller creation is the building of the tile map. A realistic game will require a much larger collection of tiles, including ones for smoothing the transition between one type of tile and another.

Tile map editors let you visually edit tiles, and build a map using drawing tools. Two popular, and free, tools are:

- Tile Studio (http://tilestudio.sourceforge.net/)
- Mappy for PC (http://www.tilemap.co.uk/mappy.php)

Functionally they are quite similar, but Mappy has additional support for creating hexagonal and isometric tiles.

It is possible to customize the way that TileStudio exports its data, by creating a TSD (Tile Studio Definition) defining the output file format.

Tile Studio is used with Java (actually J2ME) in chapter 11 of:

> *J2ME Game Programming*
> Martin Wells
> Premier Press, February 2004

In the example, Tile Studio exports several tile maps to a TSD-defined file, and java is used to read them. This chapter is available online at http://www.courseptr.com/ptr_detail.cfm?isbn=1592001181

Mappy places a lot of emphasis on playback libraries/APIs, allowing its maps to be loaded, manipulated, and displayed. The Mappy Web site offers two Java playback libraries. There is also JavaMappy (http://www.alienfactory.co.uk/javamappy/), an open source Java playback library for Mappy. It includes pluggable renderers for J2ME and J2SE 1.4. The download includes several examples and documentation.