Chapter 28.85 Waving a Magic Wand

Would you like to shoot blasts of awesome cosmic energy from a magic wand? This chapter looks at how to replace boring key presses and mouse clicks by graceful wand waving, and there's no need to attend *Hogwarts*, or even Miss Cackle's *Academy for Witches*, to do it.

Figure 1 shows the MagicWand application in action.

Figure 1. Blasting Away with a Wand.

Figure 2 reveals the input technology: a web camera focused on my real-world 'magic wand'. As I wave the wand around, the on-screen wand moves in a similar way.

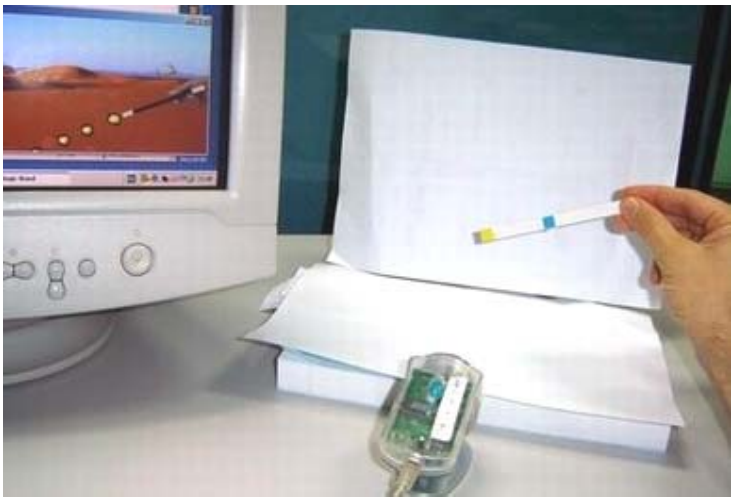

Figure 2. The Webcam and Wand.

Blasts are emitted from the wand at regular intervals, and follow a path set by the current direction of the wand. The blasts don't do anything, but it wouldn't be hard to extend the sprite code to blow something up, or knock something down.

The wand isn't an expensive item from *Ollivanders*; I created it out of coloured card and sticky tape (see Figure 3).

Figure 3. The Real-World Wand.

The yellow and blue squares represent the 'head' and 'tail' of the wand.

The class diagrams for MagicWand are shown in Figure 4. The application has two main areas of functionality: image capturing and processing (inside ImageAnalyzer, with the help of the JMFCapture, BlobsLine, BlobsManager, and Blob), and 2D rendering (in MWPanel, BlastsManager, and BlastSprite).

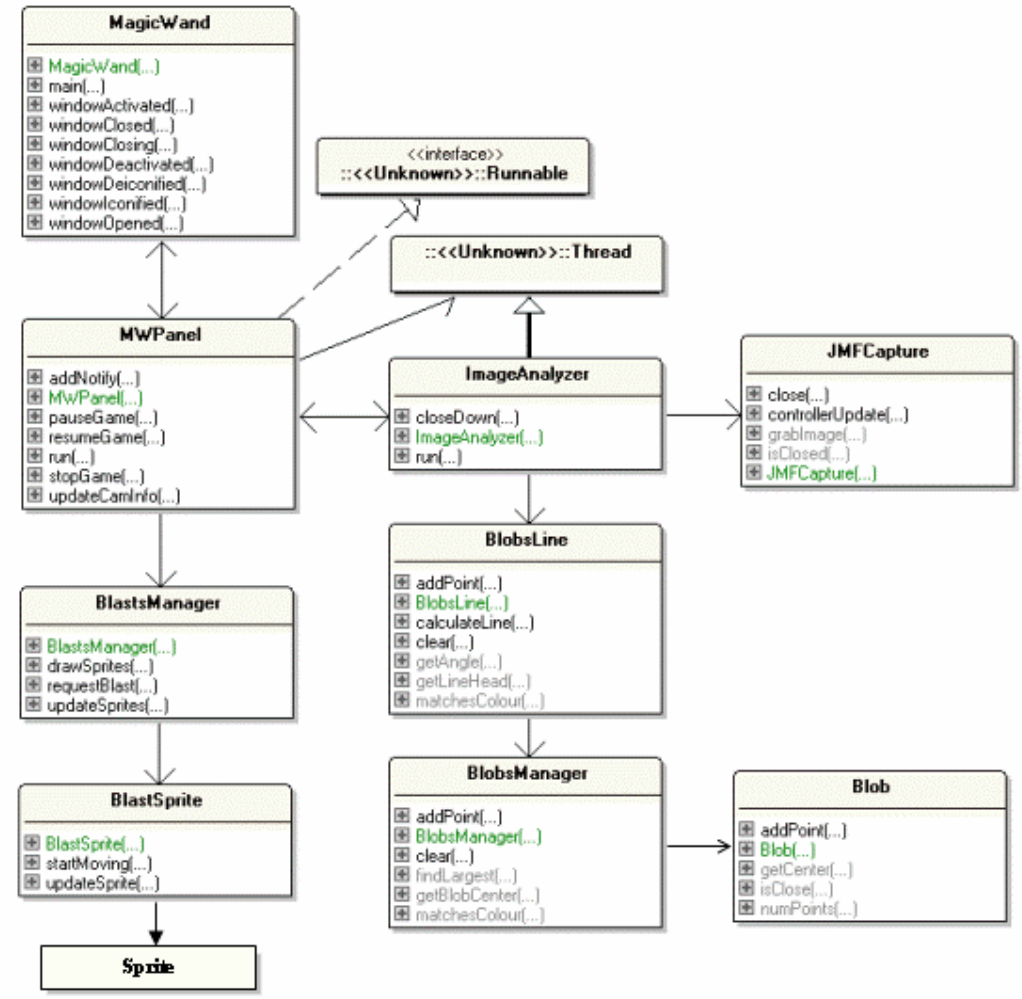

Figure 4. Class Diagrams for MagicWand.

I've left out a few superclasses in Figure 4 to simplify things (e.g. MagicWand extends JFrame, and MWPanel is a JPanel). The support classes for loading GIFs and JPEGs aren't shown either (they're used to load the background desert image, wand image, and the sprite pictures). Only the public methods are listed for each class.

The top-level class, MagicWand, is a JFrame. It creates the game panel, and monitors window events. MWPanel draws the game canvas using the threaded, active rendering approach explained in chapter 2.

The ImageAnalyzer thread is similar to the BandsAnalyser class from chapter 28.8. It grabs a picture from the web camera every 0.1 second, using JMFCapture to interface with the Java Media Framework (JMF).

The image is scanned for *blobs* (groups of pixels with the same colour), which are collected together in two BlobsManagers. Only yellow and blue blobs are stored, since the aim is to detect the head and tail of the wand in the webcam image.

When the picture has been completely scanned, a BlobsLine object calculates the line between the center of the largest yellow blob and the largest blue one. This line should hopefully match the wand's position. The coordinates of the wand head (the yellow blob's center) and the wand's angle to the horizontal are passed to MWPanel.

The blasts shooting from the wand head are BlastSprite objects, which subclass the Sprite class from chapter 11. They're managed by a BlastsManager object.

I won't go through all the details of the reused classes in this chapter, so you may want to read up on:

- the active rendering framework (chapters 2 and 3)
- 2D sprites (chapter 11)
- image processing with blobs (chapter 28.8)

An important issue is MagicWand's use of multiple threads sharing data. A quick look at Figure 4 shows that MWPanel and ImageAnalyzer are both threaded. ImageAnalyzer periodically updates data structures holding wand details and the webcam snap, and these are rendered by MWPanel.

1. Webcam Image Manipulation

The ImageAnalyzer constructor initializes a BlobsLine object with two Color objects, HEAD_COLOUR (yellow) and TAIL_COLOUR (blue), which match those used on the 'real' wand (see Figure 3).

```
// globals
// head and tail colours (yellow, blue) for the wand
private static final Color HEAD_COLOUR = new Color(200, 200, 85);
private static final Color TAIL_COLOUR = new Color(50, 150, 200);

private int imSize;               // size of captured image
private MWPanel mwPanel;
private BlobsLine blobsLine;


public ImageAnalyzer(MWPanel lp, int sz)
{ mwPanel = lp;
  imSize = sz;
  blobsLine = new BlobsLine(HEAD_COLOUR, TAIL_COLOUR);
}  // end of ImageAnalyzer()
```

I extracted suitable RGB values for the Color objects from a webcam snap of the wand. The image's pixels were examined with the freeware version of Ultimate Paint (http://www.ultimatepaint.com/).

The quality of the image analysis is much improved if the wand colours are quite different from the rest of the colours in the scene. This is helped by MagicWand using the white background shown in Figure 2. Red is a poor wand colour since there's a lot of red tone in my hand, which usually appears on the right of a snap. It's also a good idea to use a bright light source, positioned to reduce shadows in the picture.

1.1. Threaded Tasks

The ImageAnalyzer thread does four things repeatedly:

1. it grabs an image from the web camera (with the help of JMFCapture);
2. it analyses the image, by asking two BlobsManagers to store yellow and blue blobs;
3. once the image has been fully scanned, BlobsLine generates line details by examining the largest yellow and blue blobs;
4. it passes the line information, and the webcam image, over to MWPanel, where they're rendered.

These jobs are carried out inside a thread since they're computationally expensive and repeated often. Game rendering progresses independently in its own thread in MWPanel.

I've highlighted the four tasks in the run() method:

```
// globals
private static final int SNAP_INTERVAL = 120;  // ms
```

```
private JMFCapture camera;
private BufferedImage camImage = null;
private boolean running;


public void run()
{
  camera = new JMFCapture(imSize);   // initialize the webcam
  System.out.println("**Camera Ready**");

  long duration;
  BufferedImage im = null;
  running = true;

  while (running) {
    long startTime = System.currentTimeMillis();
    im = camera.grabImage();  // take a snap (task 1)
    if (im == null) {
      System.out.println("Problem loading image");
      duration = System.currentTimeMillis() - startTime;
    }
    else {
      analyzeImage(im);   // task 2
      blobsLine.calculateLine();  // task 3
                  // calculate head coords and angle

      duration = System.currentTimeMillis() - startTime;
      // System.out.println("Duration: " + duration + " ms");

      // send snapped image and line info to MWPanel for rendering
      mwPanel.updateCamInfo(im, blobsLine.getLineHead(),
                                blobsLine.getAngle() ); // task 4
    }

    if (duration < SNAP_INTERVAL) {
      try {
        Thread.sleep(SNAP_INTERVAL-duration);
        // wait until interval has passed
      }
      catch (Exception ex) {}
    }
  }
  camera.close();    // close down the camera
}  // end of run()
```

The tasks are repeated every SNAP_INTERVAL ms (120 ms). I arrived at that value by examining the execution times of tasks 1-3. On average, image capture takes around 30ms, and the analysis a further 90ms.

1.2. Analyzing the Image

analyzeImage() converts a snapped image into a pixel array, and then compares each pixel to the wand colours (yellow and blue). If a pixel's colour is near to one of them then it's coordinate is added to a blob for that colour.

```
// globals
private static final int CLEAR = 0xffffff;
  /* black pixel, which will be rendered transparently
```

```
     since the image is ARGB */


private void analyzeImage(BufferedImage im)
{
  if (im == null) {
    System.out.println("Input image is null");
    return;
  }

  blobsLine.clear();   // start with no blobs

  // extract pixels from the image into an array
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int [] pixels = new int[imWidth * imHeight];
  im.getRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);

  int xc = 0;    // hold the current pixel coordinates
  int yc = 0;

  // examine the pixels
  int i = 0;
  int colourID = BlobsLine.UNMATCHED_COLOUR;
  while(i < pixels.length) {
    colourID = blobsLine.matchesColour(pixels[i]);
    if (colourID != BlobsLine.UNMATCHED_COLOUR)  // has wand colour
      blobsLine.addPoint(xc, yc, colourID);   // store point in blob
    else      // no match
      pixels[i] = CLEAR;   // make the pixel transparent

    // move to next coordinate in image
    xc++;
    if (xc == imWidth) {    // at end of row
      xc = 0;   // start next row
      yc++;
    }
    i++;
  }

  // update the image with the modified pixels data
  im.setRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);
}  // end of analyzeImage()
```

analyzeImage() uses the BlobsLine object to test a pixel's colour, and to add it's coordinate to a blob.

If a pixel isn't close to a wand colour then it's made transparent, which is possible since JMFCapture adds an alpha channel to the snapped image.

The transparency changes mean that when the image is rendered by MWPanel only the blobs appear on screen. This gives the user some feedback on the image analysis – if no blobs are visible, then the wand should be moved around until some appear.

Around 20ms can be trimmed from the analysis time (90ms) if the modified pixel data isn't stored back in the BufferedImage object at the end of analyzeImage(). But then the webcam snap displayed by MWPanel wouldn't be partially transparent.

2. Capturing the Image

JMFCapture grabs an image using the Java Media Framework (JMF), an approach explained at length in chapter 28.7. Aside from requiring the JMF API to be installed on your machine, the capture device (i.e. the webcam) should also be registered with JMF via its JMF Registry application.

The version of JMFCapture used here differs in two ways from the one in chapter 28.7. Firstly the image is scaled so that its height is equal to a value passed to the JMFCapture constructor and, secondly, the returned image is given an alpha channel.

The height passed to the constructor is used to calculate a scale factor in hasBufferToImage(). A code fragment:

```
Buffer buf = fg.grabFrame();             // take a snap
VideoFormat vf = (VideoFormat) buf.getFormat();
int height = vf.getSize().height;        // the image's height
scaleFactor = ((double) size) / height;  // scale uses height
```

The ARGB version of the image requires a single change inside JMFCapture's makeBIM() method:

```
private BufferedImage makeBIM(Image im)
{
  BufferedImage copy = new BufferedImage(size, size,
                                  BufferedImage.TYPE_INT_ARGB);
  // create a graphics context
  Graphics2D g2d = copy.createGraphics();

  // image --> resized BufferedImage
  g2d.scale(scaleFactor, scaleFactor);  // apply scale factor
  g2d.drawImage(im,0,0,null);
  g2d.dispose();
  return copy;
}  // end of makeBIM()
```

The original version of makeBIM() specified BufferedImage.TYPE_INT_RGB as the image type. makeBIM() also applies the scale factor to the image.

3. Getting a Line on the Blobs

BlobsLine has several duties, as shown in Figure 5.

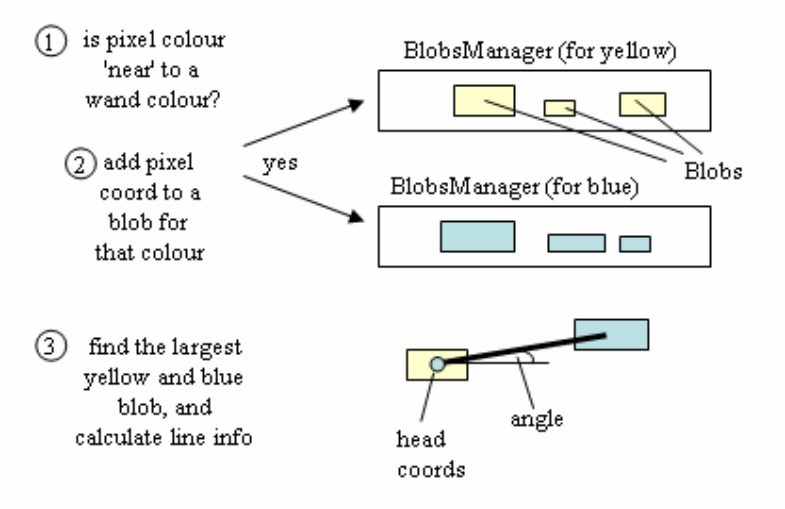

Figure 5. BlobsLine Tasks.

BlobsLine creates two BlobsManager objects to manage the wand-coloured blobs (yellow for the head, blue for its tail).

```
// globals
private BlobsManager blobManHead, blobManTail;

public BlobsLine(Color headColour, Color tailColour)
{ blobManHead = new BlobsManager(headColour);  // for yellow blobs
  blobManTail = new BlobsManager(tailColour);  // for blue blobs
}
```

3.1. Matching Against a Colour

matchesColours() checks if a pixel colour is close to one of the wand colours.

```
// globals
// IDs for the possible line colours
public static final int UNMATCHED_COLOUR = 0;
private static final int HEAD_COLOUR = 1;
private static final int TAIL_COLOUR = 2;

public int matchesColour(int pixel)
{
  if (blobManHead.matchesColour(pixel))   // close to head colour
    return HEAD_COLOUR;
  else if (blobManTail.matchesColour(pixel))  // close to tail colour
    return TAIL_COLOUR;
  else
    return UNMATCHED_COLOUR;
} // end of matchesColour()
```

The pixel is tested against each BlobsManager. The result is an integer constant, which will be UNMATCHED_COLOUR if the pixel isn't yellow or blue.

UNMATCHED_COLOUR is a public constant since it's used back in analyzeImage() in the ImageAnalyzer class.

3.2. Adding a Pixel Coordinate

If the pixel is yellow or blue then analyzeImage() calls addPoint() in BlobsLine to add the pixel's (x, y) coordinate to a blob. The colour ID previously returned by matchesColour() directs the coordinate to the right BlobsManager.

```
public void addPoint(int x, int y, int colourID)
{
  if (colourID == HEAD_COLOUR)
    blobManHead.addPoint(x,y);
  else if (colourID == TAIL_COLOUR)
    blobManTail.addPoint(x,y);
  else
    System.out.println("Could not add point for colour " + colourID);
}  // end of addPoint()
```

3.3. Calculating Line Information

ImageAnalyzer calls BlobsLine's calculateLine() to calculate details about the line connecting the largest blobs of the two wand colours. The hope is that the largest yellow blob is centered on the wand's head, the blue blob on it's tail, and the line matches the wand's position.

```
// globals for the line-related data
private Point lineHead = null;
private double lineAngle = 0;

public void calculateLine()
{
  // get the biggest blob for the head colour
  int headBlobIdx = blobManHead.findLargest();
  if (headBlobIdx == -1)   // no blob found
    return;   // don't change head or angle values

  // get the biggest blob for the tail colour
  int tailBlobIdx = blobManTail.findLargest();
  if (tailBlobIdx == -1)   // no blob found
    return;    // don't change head or angle values

  // found two largest blobs; so get their centers
  lineHead = blobManHead.getBlobCenter( headBlobIdx );
  Point tailPt = blobManTail.getBlobCenter( tailBlobIdx );

  // get angle of head of line to the horizontal
  int xDist = tailPt.x - lineHead.x;
  int yDist = tailPt.y - lineHead.y;
  if (xDist == 0)
    xDist = 1;  // avoid division by zero

  lineAngle = Math.atan( ((double)yDist)/xDist );
}  // end of calculateLine()
```


The method gets the indices of the largest blobs held by the BlobsManagers. If suitable indices are returned by both managers, then the coordinates of their centers are retrieved. The line angle is obtained by a simple piece of trigonometry, illustrated by Figure 6.

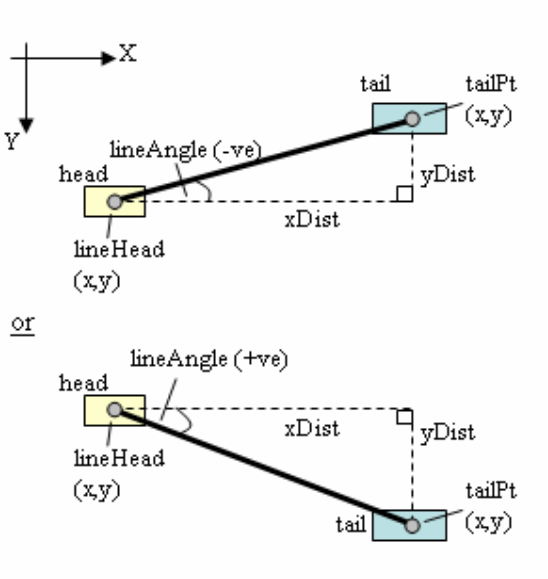

Figure 6. Calculating the Line Angle.

The line angle will be negative if the head is pointing downwards, and positive if it's pointing up.

Aside from the two cases shown in Figure 6, what if the user is pointing the wand to the right? calculateLine() still generates an angle, but when MWPanel uses that angle, it assumes the wand is pointing to the left, and sends sprite blasts to the left. If I wanted to aim blasts in any direction, I'd have to generate a wand direction value (e.g. TO_LEFT, TO_RIGHT) in addition to the angle, so the blasting code could correctly aim the sprites.

Another subtlety is what happens when one of the BlobsManagers doesn't supply an index. In that case, calculateLine() returns without updating lineHead and lineAngle. This means that if the snapped image doesn't have a large enough yellow or blue blob, then MWPanel will keep using the old wand head position and angle. This explains why the on-screen wand stays where it is when the user removes the real-world wand from in front of the webcam.

Another approach would be to set lineHead to null and lineAngle to 0 when no large blobs are found. However, this can have a nasty effect on the rendering code in MWPanel, as discussed below.

The head coordinates and angle are retrieved by ImageAnalyzer calling their get methods:

```
public Point getLineHead()
{ return lineHead; }
```

```
public double getAngle()
{ return lineAngle; }
```

4. Managing the Blobs

A BlobsManager manages all the blobs of a specified colour.

The class bears a striking resemblance to BandManager in chapter 28.8. The matchesColour() method is identical, and addPoint() very similar. The Blob class used by BlobsManager is virtually unchanged, so I won't bother describing it again.

BlobsManager's constructor initializes an ArrayList to store its blobs.

```
// globals
// RGB components for this manager's colours
private int redCol, greenCol, blueCol;

private ArrayList<Blob> blobs;
private int currBlobIdx;     // index of last blob that was updated


public BlobsManager(Color c)
{
  redCol = c.getRed();
  greenCol = c.getGreen();
  blueCol = c.getBlue();

  blobs = new ArrayList<Blob>();
  currBlobIdx = -1;
}  // end of BlobsManager()
```

J2SE 5 generics are utilized for the ArrayList.

4.1. Adding a Pixel Coordinate

addPoint() adds a pixel coordinate to a blob. The coordinate either joins an existing blob if it's close enough, or a new blob is created.

```
public void addPoint(int x, int y)
{
  int blobIndex = findCloseBlob(x,y);
  if (blobIndex != -1) {   // found a blob close to (x,y)
    Blob b = (Blob) blobs.get(blobIndex);
    b.addPoint(x,y);
    currBlobIdx = blobIndex;
  }
  else {   // no close blob, so create a new one
    Blob b = new Blob();
    b.addPoint(x,y);
    blobs.add(b);
    currBlobIdx = blobs.size() - 1;
  }
}  // end of addPoint()
```

findCloseBlob() returns the index of a blob that's close to (x,y). If no suitable blob is found then it returns -1. The code is identical to the same-named method in BandManager.

4.2. Finding the Largest Blob, Maybe

findLargest() performs a linear search through the blobs ArrayList, looking for the blob with the most points, but with at least MIN_POINTS points. This minimum requirement reduces the possibility of mistaking a small blob made from background pixels for the wand's head or tail.

```
// globals
private static final int MIN_POINTS = 100;

public int findLargest()
/* Search through the blobs, and return the index
   of the largest, or -1 if none is found */
{
  int largestIdx = -1;
  int maxPts = MIN_POINTS;    // blob must be at least this big

  int numPts;
  Blob blob;
  for(int i=0; i < blobs.size(); i++) {
    blob = blobs.get(i);
    numPts = blob.numPoints();
    if (numPts > maxPts) {
      maxPts = numPts;
      largestIdx = i;
    }
  }
  return largestIdx;
}  // end of findLargest()
```

4.3. A Blob's Center

getBlobCenter() returns the center of the specified blob:

```
public Point getBlobCenter(int idx)
{
  Point p = null;
  if ((idx < 0) || (idx >= blobs.size()))
    System.out.println("No blob with that index: " + idx);
  else {
    Blob blob = blobs.get(idx);
    p = blob.getCenter();
  }
  return p;
}
```

5. The Game Canvas

The MWPanel constructor sets up active rendering (see chapter 2 for an explanation). In addition, it loads the necessary MagicWand images with the help of ImagesLoader (from chapter 6), initializes the BlastsManager, and starts the ImageAnalyzer thread:

```
// globals
private static final int PWIDTH = 700;   // size of panel
private static final int PHEIGHT = 256;

// image loader information files
private static final String IMS_INFO = "imsInfo.txt";

private BlastsManager blastsMan;     // manages the wand blasts
private ImageAnalyzer imAnalyzer;    // analyzes the webcam images

private BufferedImage bgImage = null;    // the background image
private BufferedImage wandImage = null;  // a picture of the wand


// in the MWPanel constructor...
// load the background and wand images
ImagesLoader imsLoader = new ImagesLoader(IMS_INFO);
bgImage = imsLoader.getImage("desert");
wandImage = imsLoader.getImage("wand");

// create manager for blast sprites
blastsMan = new BlastsManager(PWIDTH, PHEIGHT, imsLoader,
                              (int)(period/1000000L) ); // in ms

// start webcam snapping and analysis
imAnalyzer = new ImageAnalyzer(this, PHEIGHT);
   // the captured image is scaled to be the same height as the panel
imAnalyzer.start();
```

The wand image (see Figure 7) has a transparent background, with the wand head located at (3,22), coordinates used in later code to position the blast sprites.

Figure 7. The Wand Image.

The animation period passed to BlastsManager (period/1000000) is in milliseconds, and controls the frequency of the blasts coming from the wand.

The panel height (PHEIGHT) argument of the ImageAnalyzer object is used to scale the snapped image. When the image is drawn, it'll occupy the full height of the panel.

5.1. The Animation Loop

The active rendering animation loop in run() is quite complex, but can be summarized as:

```
while(running) {
  gameUpdate();
  gameRender();
  paintScreen();
  // maybe sleep a while
}
imAnalyzer.closeDown();    // stop snapping pictures
```

paintScreen() is unchanged from previous active rendering examples, so won't be described further, but I will explain how the game is updated and rendered.

5.2. Updating the Game

Game elements are modified in *two* places. gameUpdate() handles the blast sprites by contacting the BlastsManager:

```
private void gameUpdate()
{ if (!isPaused)
    blastsMan.updateSprites();
}
```

The less obvious place where things are changed is in updateCamInfo(), a method called by the ImageAnalyzer thread after it has analyzed each webcam snap.

```
// global data changed periodically by ImageAnalyzer
private BufferedImage camImage = null;  // the current webcam image
private Point wandHeadPt = null;        // the wand's head coords
private double wandAngle = 0;          // angle of wand to horizontal


public void updateCamInfo(BufferedImage im,
                          Point hdPt, double angle)
{ if (!isPaused) {
    camImage = im;
    wandHeadPt = hdPt;
    wandAngle = angle;
  }
}  // end of updateCamInfo()
```

updateCamInfo() is the point of contact between the ImageAnalyzer thread and the MWPanel rendering thread. This method makes it possible for ImageAnalyzer to change camImage, wandHeadPt, and wandAngle while MWPanel is using them.

A common solution to this kind of sharing conflict is to synchronize the methods that manipulate and use the data with the `synchronized` keyword. This prevents a method that updates the data from executing while the data is being accessed.

The drawback with this approach in MagicWand is that ImageAnalyzer carries out changes every 0.1 second. The frequent locking caused by synchronization would severely impact the performance of the rendering code.

An alternative is to not to bother with expensive synchronization, but code the rendering methods with care instead, bearing in mind that camImage, wandHeadPt, and wandAngle may change at any time.

A useful feature of ImageAnalyzer is that after the first picture has been processed, the head coordinates object will never again be null. If large blobs cannot be found in the webcam snap then the head coordinates are left unchanged. This means that the rendering code can rely on the wandHeadPt object always having a value after its first assignment.

5.3. Rendering the Scene

The game scene is composed from a background image, a (mostly transparent) webcam image, a picture of a wand, and the blast sprites shot from the wand.

```
// globals for off-screen rendering
private Graphics2D dbg2;
private Image dbImage = null;

private void gameRender()
{
  if (dbImage == null){
    dbImage = createImage(PWIDTH, PHEIGHT);
    if (dbImage == null) {
      System.out.println("dbImage is null");
      return;
    }
    else
      dbg2 = (Graphics2D) dbImage.getGraphics();
  }

  // draw the background
  dbg2.drawImage(bgImage, 0, 0, this);

  if (camImage != null)  {   // if there's a webcam image
    int imXOffset = PWIDTH - camImage.getWidth();
    /* this offset will put the right edge of webcam image
       against the right edge of the panel */

    dbg2.drawImage(camImage, imXOffset, 0, this);
       // draw the webcam image offset to the right

    if (wandHeadPt != null) {
      // copy wand head coords, and make x-value relative to panel
      int xHead = wandHeadPt.x + imXOffset;
      int yHead = wandHeadPt.y;
      double angle = wandAngle;   // copy the wand angle also

      drawWand(dbg2, xHead, yHead, angle);
      fireWand(xHead, yHead, angle);
    }
  }
  blastsMan.drawSprites(dbg2);   // draw the blast sprites
}  // end of gameRender()
```

The imXOffset value ensures that the webcam image, wand, and blasts are positioned over on the right side of the panel.

The advantage of using copies of the wandHeadPt coordinates and wandAngle is that they cannot change while they're being used inside drawWand() and fireWand(). However, copying only *reduces* the chance of conflicts. There's still a slim chance

that the head coordinates and/or angle will be changed by ImageAnalyzer during their copying. This is really not so serious, since it only means that the wand image and a new blast sprite may be incorrectly positioned. The benefits of not being burdened with synchronization code outweigh this slight problem.

5.4. Drawing the Wand

The wand image (see Figure 7) has to be translated to the head position, and rotated by the angle amount. Rather than applying these operations directly to the image, an affine transformation is applied to the coordinates system instead. The payoff is that the call to drawImage() can use (0,0) as the drawing position.

```
private void drawWand(Graphics2D g2, int xHead, int yHead,
                                                    double angle)
{ AffineTransform oldTrans = g2.getTransform();
                           // save current coords system

  AffineTransform posHead = new AffineTransform();
  posHead.translate(xHead-3, yHead-22);
    /* Move the drawing coordinates to the head position.
       The (-3,-22) shifts the coordinates back to the
       top-left of the wand image. */

  // rotate the drawing coordinates by the wand angle amount
  posHead.rotate(angle);

  g2.transform(posHead);  // apply the translation and rotation
  g2.drawImage(wandImage, 0,0, this);
      // draw the image at the modified (0,0) position

  // restore the old coordinates system
  g2.setTransform(oldTrans);
}  // end of drawWand()
```

It's important to backup the original coordinates system at the start of drawWand(), so it can be restored at the end.

The drawing coordinates for the wand image are offset back to the top-left corner of the image by subtracting (3, 22) from the wand head position.

5.5. Firing a Blast from the Wand

fireWand() asks the BlastsManager to fire a blast sprite, with a frequency controlled by a counter.

```
// globals
private static final int BLAST_FREQ = 5;
private int blastCounter = 0;

private void fireWand(int xHead, int yHead, double angle)
{
  blastCounter++;
  if (blastCounter >= BLAST_FREQ) {
    blastsMan.requestBlast(xHead, yHead, angle);
    blastCounter = 0;   // reset counter
```

```
  }
}
```

6. Managing the Blasts

BlastsManager's constructor creates an array of ten inactive BlastSprite objects.

```
// globals
private static final int NUM_BLASTS = 10;
private BlastSprite sprites[];

public BlastsManager(int width, int height,
                     ImagesLoader imsLd, int period)
{ sprites = new BlastSprite[NUM_BLASTS];
  for (int i=0; i < NUM_BLASTS; i++)
    sprites[i] = new BlastSprite(width, height, imsLd, period);
}
```

Updating and drawing the sprites involves iterating through the array and calling the sprites' updateSprite() and drawSprite() methods. A sprite is only updated and drawn if it's active.

```
public void updateSprites()
{ for (int i=0; i < NUM_BLASTS; i++)
    sprites[i].updateSprite();
}

public void drawSprites(Graphics g)
{ for (int i=0; i < NUM_BLASTS; i++)
    sprites[i].drawSprite(g);
}
```

6.1. Requesting a New Blast

MWPanel calls BlastsManager's requestBlast() to fire a blast from the head of the wand. The sprite then moves a fixed distance on each update (STEP_SIZE pixels), independent of its starting angle. All the sprites move by the same amount, which means they all have the same velocity.

Figure 8 illustrates how the STEP_SIZE distance along the sprite's path is split into x- and y- axis components.

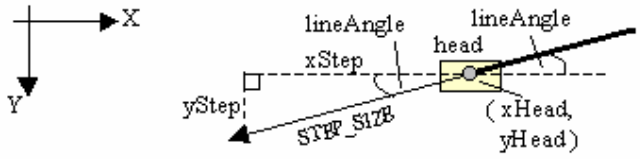

Figure 8. Step Sizes for a Blast Sprite.

```
// globals
private static final double STEP_SIZE = 10.0;
```

```
public void requestBlast(int xHead, int yHead, double lineAngle)
{
  // calculate sprite's steps in the x- and y- directions
  int xStep = (int) Math.round( -STEP_SIZE * Math.cos(lineAngle) );
  int yStep = (int) Math.round( -STEP_SIZE * Math.sin(lineAngle) );
    /* the negative step makes the direction opposite to the
       direction defined by the wand head-to-tail */

  // fire an inactive blast sprite (if there's one available)
  for (int i=0; i < NUM_BLASTS; i++)
    if (!sprites[i].isActive()) {
      sprites[i].startMoving(xHead, yHead, xStep, yStep);
      break;
    }
}  // end of requestBlast()
```

The negative step size used in the step expressions makes the sprite move in the opposite direction from that specified by the wand's head-to-tail orientation.

The head position (xHead, yHead) and direction vector (xStep, yStep) are assigned to the first inactive sprite found in the sprites[] array. If all the sprites are active (i.e. moving on-screen) then the request isn't carried out

7. The Blast Sprite

A blast sprite is an animated sprite, which inherits most of its functionality from the Sprite class described in chapter 11.

The constructor creates an inactive sprite, visually represented by the image strip stored in blast.gif.

```
// globals
private int period;   // in ms
  /* The game's animation period used by the image cycling. */


public BlastSprite(int width, int height, ImagesLoader imsLd, int p)
{
  super(0, 0, width, height, imsLd, "blast");
  period = p;
  setStep(0,0);  // no movement
  setActive(false);
} // end of BlastSprite()
```

blast.gif contains 6 images, shown in Figure 9. I grabbed them from the excellent freeware sprite library, SpriteLib GPL, by Ari Feldman, available at http://www.flyingyogi.com/fun/spritelib.html.

Figure 9. The blast.gif Image Strip.

7.1. Making the Sprite Move

startMoving() sets the sprite's starting position, its step increments in the x- and y-directions (xStep, yStep), and the duration of its animation loop.

```
// globals
private static final double DURATION = 1.0;  // secs
  // total time to cycle through all the images


public void startMoving(int xHead, int yHead, int xStep, int yStep)
{
  // center the sprite at the wand head position
  setPosition(xHead - (getWidth()/2), yHead - (getHeight()/2) );
  setStep(xStep, yStep);         // movement direction
  loopImage(period, DURATION);   // cycle through the images
  setActive(true);
} // end of moveLeft()
```

setPosition() requires the sprite's top-left corner, which is calculated so its center is located at the wand head position (xHead, yHead).

The call to loopImage() specifies that the 'blast' image strip is cycled through every second.

7.2. Updating the Sprite

The blast sprite utilizes the inherited updateSprite() behavior except when it leaves the panel. In that case, the sprite is made inactive, so it can be reused as a 'new' blast later on.

```
public void updateSprite()
{
  if (isActive()) {
    if ( ((locx+getWidth() < 0) && (dx < 0)) ||     // gone off lhs
         ((locx > getPWidth()) && (dx > 0)) ||      // gone off rhs
         ((locy+getHeight() < 0) && (dy < 0)) ||    // gone off top
         ((locy > getPHeight()) && (dy > 0)) ) {   // gone off bottom
      setStep(0,0);  // no movement
      setActive(false);
    }
    super.updateSprite();
  }
} // end of updateSprite()
```