Chapter 13.5. Touring the Town

Car racing games are great fun, and don't need complicated 3D rendering to look good. In this chapter, I start with a simple 2D car touring application called TourLand, which is shown on the left of Figure 1. I'll extend the code in two steps, finishing with a version that seems to use 3D, but is actually "fakin' it" with a 2D perspective trick (see the right hand picture in Figure 1).

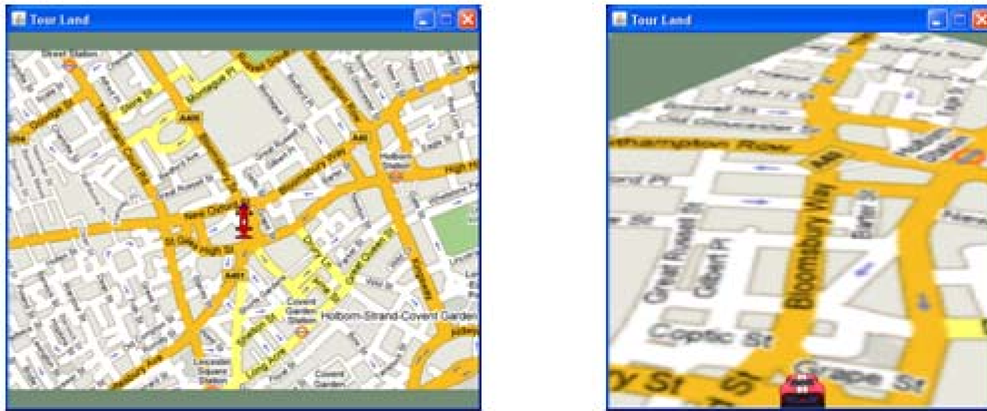


Figure 1. The 2D and Fake 3D Versions of TourLand.

The 2D TourLand displays a red racing car at the center of the window (it's quite hard to see in Figure 1), which moves and turns in response to user key presses. The 3D version draws the back of the racing car on the bottom edge of the window in the middle. Key presses don't affect the car image – instead the view of the map changes.

Perhaps surprisingly, much of the code in the 2D versions of TourLand can be reused in the fake 3D version. In particular, the 2D classes let me test:

- the use of *two* coordinate systems: one utilizing the car's position relative to the map, and the other for the car and map's position relative to the JPanel;
- how to specify the rotation of the car picture, the car's location, and the map with Java 2D affine operations.

The fake 3D version utilizes JAI (the Java Advanced Imaging library from https://jai.dev.java.net/) to perform fast, hardware-supported, perspective warping transformations on the map.

© Andrew Davison 2009

1. TourLand Overview

The top-level TourLand application is a simple JFrame which, depending on user input from the command line, loads a different JPanel for rendering the car and map.

Figure 2 shows the class diagrams for TourLand, with only the class names and public methods shown.

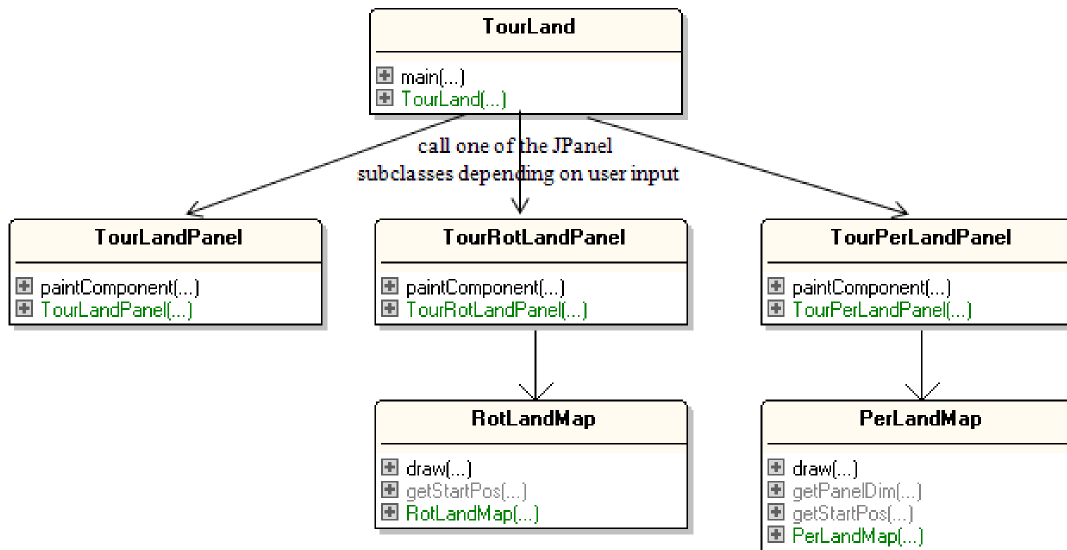


Figure 2. Class Diagrams for TourLand.

There are three different JPanel subclasses:

1. *TourLandPanel*: This panel lets me drive a car over a 2D map. The car can move forwards, backwards, and rotate around its center, but can't drive off the map. Car 'movement' is actually implemented by moving the map while keeping the vehicle stationary, but the car does rotate.
2. *TourRotLandPanel*: I can drive a car over a 2D map with this panel in a similar way to TourLandPanel. The main difference is that the map is now translated *and* rotated, and the car never moves or turns. This change may seem a bit trivial but is a useful intermediate coding step before moving to the fake 3D view. The map image manipulation is complex enough to be carried out in its own RotLandMap class.
3. *TourPerLandPanel*: With this panel, I can drive a car over a fake 3D map. The map is warped to give it a perspective projection, and moves and rotates while the car stays on the bottom edge of the window (as shown in Figure 1). The manipulation of the map is done by the PerLandMap class, with warping support from JAI.

The rest of the chapter describes the three JPanels in more detail, starting with the simple TourLandPanel, and working up to TourPerLandPanel.

© Andrew Davison 2009

2. TourLandPanel: Driving Down the Highway (Version 1)

As we maneuver our sporty red racing car through the mean London streets, TourLandPanel actually leaves the car standing at the center of the window, and translates the map instead. However, the car does rotate around it's center when it is turned, which is somewhat simpler to implement than rotating the map.

Figure 3 shows the car at an edge of the map, having turned to the south west from its starting orientation pointing north.

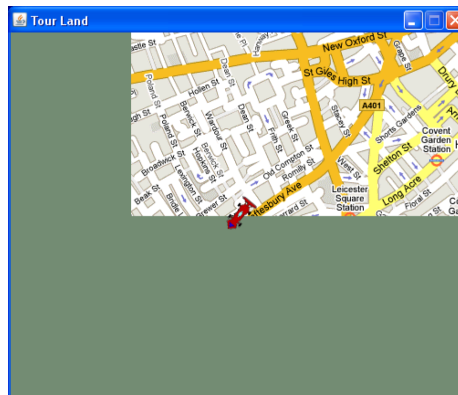


Figure 3. Driving using TourLandPanel

TourLandPanel initially defines its dimensions, loads the map and car images, and sets up a listener for processing the user's key presses.

```
// globals
// panel dimensions
private static final int PWIDTH = 500;
private static final int PHEIGHT = 400;

// background colour (gray green)
private static final Color LAND_BG_COLOUR =  new Color(115,140,115);


public TourLandPanel()
{
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));
  setBackground( LAND_BG_COLOUR );

  initImages();

  setFocusable(true);
  requestFocus();    // the JPanel now receives key events
  addKeyListener( new KeyAdapter() {
    public void keyPressed(KeyEvent e)
    { processKey(e);  }
  });
} // end of TourLandPanel()
```

initImages() loads the map and car images, and calculates *two* (x, y) coordinates for the car.

```
// globals
```

 © Andrew Davison 2009

```
private static final String LAND_IM = "londonMap.gif";
private static final String PLAYER_IM = "player.gif";
                            // a racing car, viewed from above

// land image info
private BufferedImage landIm;
private int landWidth, landHeight;

// player info
private BufferedImage playerIm;
private int xDrawPos, yDrawPos;
    // drawing coords for player (always in panel center)

private int xPlayer, yPlayer;  // position of player on the map
private int angle;    // angle to the vertical; positive is clockwise


private void initImages()
// load land and player images, and calculate their positions/sizes
{
  landIm = loadImage(LAND_IM);   // load land image (the map)
  landWidth = landIm.getWidth();
  landHeight = landIm.getHeight();

  playerIm = loadImage(PLAYER_IM);  // load player image (the car)

  // player's drawing position in the JPanel
  //    - fixed at the center of the window
  xDrawPos = PWIDTH/2 - playerIm.getWidth()/2;
  yDrawPos = PHEIGHT/2 - playerIm.getHeight()/2;

  // the player starts in the center of the land
  xPlayer = landWidth/2;
  yPlayer = landHeight/2;
  angle = 0;    // player faces north
}  // end of initImages()
```

One of the player's coordinates is (xDrawPos, yDrawPos), its position relative to the panel's coordinate system, which never changes since the car doesn't move from its center point. Figure 4 shows how the coordinate is calculated using the panel and car dimensions.

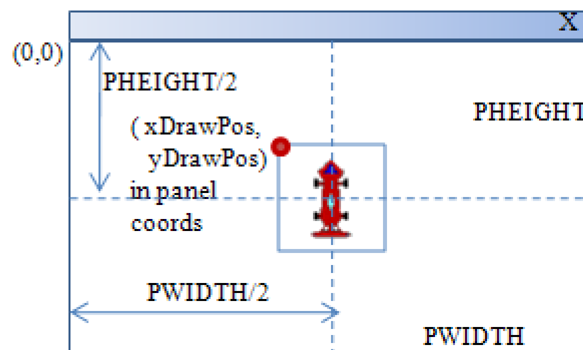


Figure 4. The Car's Position in the JPanel.

 © Andrew Davison 2009

The other car coordinate, (xPlayer, yPlayer), is its position relative to the land, which *does* change over time. The car starts in the middle of the map, pointing north:

```
xPlayer = landWidth/2;
yPlayer = landHeight/2;
angle = 0;   // player faces north
```

These values change when the user presses keys, and are used to translate the land and rotate the car.

2.1. User Input

processKey() deals with keys which terminate the application, and keys that affect the car's land position (xPlayer, yPlayer) or angle.

```
// direction constants
private static final int FWD = 0;
private static final int BACK = 1;
private static final int TURN_ANGLE = 10;  // in degrees


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

  // termination keys
  // listen for esc, q, end, ctrl-c
  if ((keyCode == KeyEvent.VK_ESCAPE) ||
      (keyCode == KeyEvent.VK_Q) ||
      (keyCode == KeyEvent.VK_END) ||
      ((keyCode == KeyEvent.VK_C) && e.isControlDown()) )
    System.exit(0);

  // game-play keys
  if (keyCode == KeyEvent.VK_LEFT)
    turn(-TURN_ANGLE);   // counter clockwise
  else if (keyCode == KeyEvent.VK_RIGHT)
    turn(TURN_ANGLE);   // clockwise
  else if (keyCode == KeyEvent.VK_UP)
    move(FWD);
  else if (keyCode == KeyEvent.VK_DOWN)
    move(BACK);
  }  // end of processKey()
```

turn() changes the angle, and a repaint is requested to redraw the scene

```
private void turn(int ang)
{ angle = (angle + ang) % 360;
  repaint();
}
```

move() moves the player 'forwards' or 'backwards' by a predefined step, with the direction depending on the current angle.

 © Andrew Davison 2009

```
// global
private static final double STEP = 4.0;  // distance to move forward

private void move(int dir)
{
  double rad = Math.toRadians(angle);
  double xStep = STEP * Math.sin(rad);
  double yStep = STEP * Math.cos(rad);

  if (dir == FWD) {    // forwards
    xPlayer += xStep;
    yPlayer -= yStep;
  }
  else if (dir == BACK) {   // backwards
    xPlayer -= xStep;
    yPlayer += yStep;
  }
  else
    System.out.println("Unknown direction: " + dir);

  // keep player on the land
  if (xPlayer < 0)
    xPlayer = 0;
  else if (xPlayer > landWidth-1)
    xPlayer = landWidth-1;

  if (yPlayer < 0)
    yPlayer = 0;
  else if (yPlayer > landHeight-1)
    yPlayer = landHeight-1;

  repaint();
}  // end of move()
```

The calculation of the new (xPlayer, yPlayer) coordinate is illustrated by Figure 5.

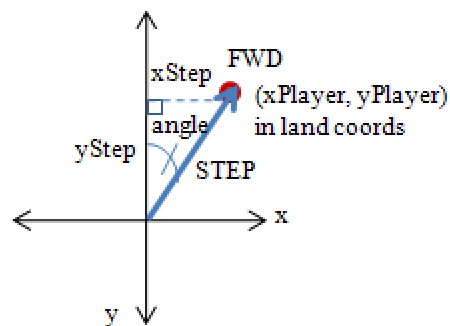


Figure 5. Changing the Player's Coordinate.

move() also ensures that the new position is within the land's boundaries. Then a repaint is requested.

 © Andrew Davison 2009

2.2. Drawing the Scene

After move()'s or turn()'s repaint request, paintComponent() will eventually be called, and the land and player are drawn using (xDrawPos, yDrawPos), (xPlayer, yPlayer) and angle.

```
public void paintComponent(Graphics g)
// draw the land then the (rotated) player
{
  super.paintComponent(g);

  /* (xLand,yland) is the top-left drawing position on
     the land image in the JPanel such that
     (xPlayer, yPlayer) is at the center of the JPanel */
  int xLand = (int)(PWIDTH/2 - xPlayer);
  int yLand = (int)(PHEIGHT/2 - yPlayer);
  g.drawImage(landIm, xLand, yLand, null);     // draw the land

  // draw the (rotated) player
  BufferedImage rotIm = rotatePlayer(playerIm);
  g.drawImage(rotIm, xDrawPos, yDrawPos, null);
} // end of paintComponent()
```

The drawing of the land requires the calculation of the image's top-left corner (xLand, yLand) such that (xPlayer, yPlayer) corresponds to the center of the JPanel (see Figure 6).

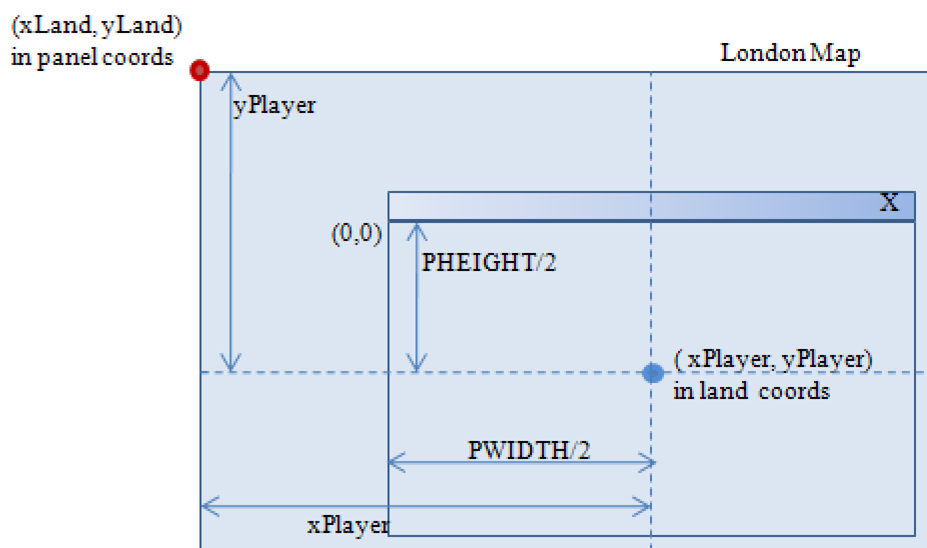


Figure 6. Positioning the Land.

The maths is somewhat confusing because it converts a land coordinate (xPlayer, yPlayer) into a panel coordinate (xLand, yLand), which may be negative (as in the Figure 6 example).

 © Andrew Davison 2009

Rotating the car is straight forward, as long as the image file has certain properties, such as the one in Figure 7.

Figure 7. The player.gif Image.

The image must have a transparent background, and the visible parts of the car must all lie within a circle defined by the car's center point, and the shortest distance from that point to the image's edge. This second requirement is illustrated in Figure 8, when the car is rotated by angle degrees.

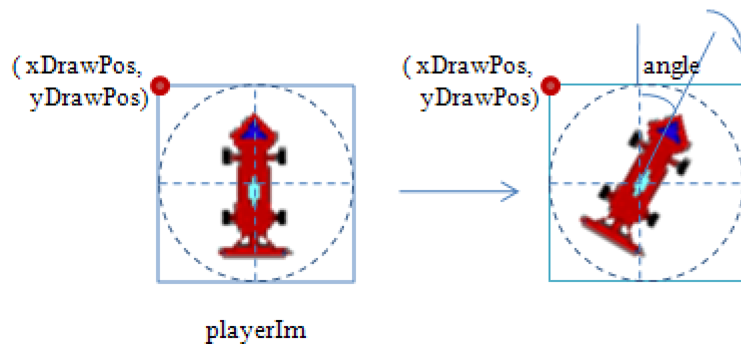


Figure 8. Rotating the Car Image.

If the image extends beyond the dotted circle then there is a risk of that part of the picture will be lost when it is rotated.

The simplest way of meeting this requirement is to enlarge the image in a graphics editor (e.g. The GIMP, http://www.gimp.org/), adding extra transparent background pixels.

The image is rotated using a Java 2D AffineTransform operation.

```
private BufferedImage rotatePlayer(BufferedImage im)
{
  BufferedImage rotIm = im;
  if (angle != 0) {  // only do this if the image needs rotating
    // rotate by angle degrees around the center of the image
    AffineTransform at = AffineTransform.getRotateInstance(
                                   Math.toRadians(angle),
                                   playerIm.getWidth()/2,
                                   playerIm.getHeight()/2);

    AffineTransformOp rotateOp =
        new AffineTransformOp(at, AffineTransformOp.TYPE_BILINEAR);
    rotIm = rotateOp.filter(im, null);
  }
  return rotIm;
}  // end of rotatePlayer()
```

The AffineTransformOp uses bilinear interpolation to improve the quality of the resulting rotated image.

 © Andrew Davison 2009

3. TourRotLandPanel: Driving Down the Highway (Version 2)

The second version of the JPanel, TourRotLandPanel, lets the user press the same keys to move and turn the racing car as previously. But in this version, all the translations and rotations are applied to the land; the car stays fixed in the center of the window, always facing north.

Figure 9 shows the racing car near an edge of the map, having 'turned' through nearly 180 degrees. Of course, the car hasn't turned at all, the land underneath has rotated.

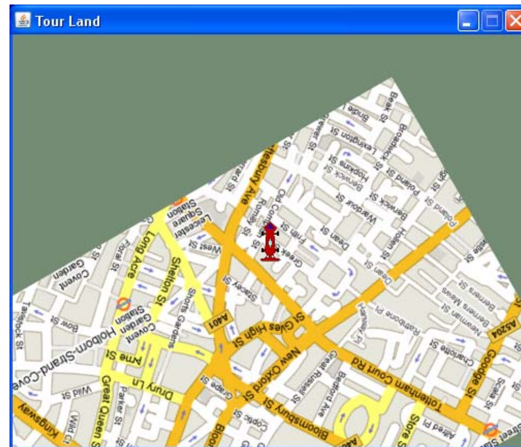


Figure 9. Driving using TourRotLandPanel

3.1. Loading the Images

initImages() in TourRotLandPanel is called to load the land and player images, and set the player coordinates:

```
// globals
private static final int PWIDTH = 500;    // panel dimensions
private static final int PHEIGHT = 400;

private static final String LAND_IM = "londonMap2.gif";
private static final String PLAYER_IM = "player.gif";

private RotLandMap landMap;     // land info in a separate class

private BufferedImage playerIm;    // player info
private int xDrawPos, yDrawPos;    // position in the panel (fixed)
private double xPlayer, yPlayer;   // position on the land
private int angle;


private void initImages()
{
  landMap = new RotLandMap(LAND_IM, PWIDTH, PHEIGHT);

  playerIm = loadImage(PLAYER_IM);   // load player image

  // player's drawing position in JPanel (fixed)
  //     - at the center of the window
  xDrawPos = PWIDTH/2 - playerIm.getWidth()/2;
  yDrawPos = PHEIGHT/2 - playerIm.getHeight()/2;
```

 © Andrew Davison 2009

```
  // set the player's starting position on the land
  Point2D.Double startPt = landMap.getStartPos();
  xPlayer = startPt.x;
  yPlayer = startPt.y;

  angle = 0;   // player faces north
}  // end of initImages()
```

Most of the code related to the player – the loading of its image, and the initialization of its drawing position is unchanged from initImages() in TourLandPanel. However, the extra manipulation of the land image (i.e. translations *and* rotations) is sufficiently complicated to justify its own class, RotLandMap. RotLandMap.getStartPos() is called to obtain the player's starting position relative to the land.

3.2. Moving and Turning

TourRotLandPanel uses the same approach as TourLandPanel for moving the player: it calls processKey() whenever a key is pressed, move() updates the (xPlayer, yPlayer) coordinate, and turn() changes the player's angle.

move() is slightly modified from the one in TourLandPanel; instead of stopping the car from traveling beyond the edges of the land, it uses hardwired (MIN_X, MIN_Y) and (MAX_X, MAX_Y) coordinates:

```
// globals
// land limits (determined for londonMap2.gif)
private static final int MIN_X = 59;
private static final int MAX_X = 610;
private static final int MIN_Y = 157;
private static final int MAX_Y = 518;


private void move(int dir)
{
  double rad = Math.toRadians(angle);
  double xStep = STEP * Math.sin(rad);
  double yStep = STEP * Math.cos(rad);

  if (dir == FWD) {   // forwards
    xPlayer += xStep;
    yPlayer -= yStep;
  }
  else if (dir == BACK) {  // backwards
    xPlayer -= xStep;
    yPlayer += yStep;
  }
  else
    System.out.println("Unknown direction: " + dir);

  // keep player on the land (changed from TourLandPanel)
  if (xPlayer < MIN_X)
    xPlayer = MIN_X;
  else if (xPlayer > MAX_X)
    xPlayer = MAX_X;

  if (yPlayer < MIN_Y)
```

 © Andrew Davison 2009

```
    yPlayer = MIN_Y;
  else if (yPlayer > MAX_Y)
    yPlayer = MAX_Y;

  repaint();
}  // end of move()
```

The reason for these modified limits will become clearer when I describe how the land is rotated in RotLandMap.

3.3. Drawing the Scene

paintComponent() is simpler than the one in TourLandPanel since the details of rotating and translating the land are hidden inside RotLandMap. Also, the player no longer needs to be rotated.

```
public void paintComponent(Graphics g)
// draw the land and then the player
{
  super.paintComponent(g);

  landMap.draw(g, xPlayer, yPlayer, angle);     // draw the land
  g.drawImage(playerIm, xDrawPos, yDrawPos, null);  // draw player
}  // end of paintComponent()
```

3.4. Land Image Requirements

The land rotation code in RotLandMap is simplified by having the image follow similar rules as those used for the rotatable player image in TourLandPanel.

All the map details should occur within a circle defined by the image's center point, and the shortest distance from that point to the image's edge. This requirement is illustrated in Figure 10 when the image is rotated by angle degrees.

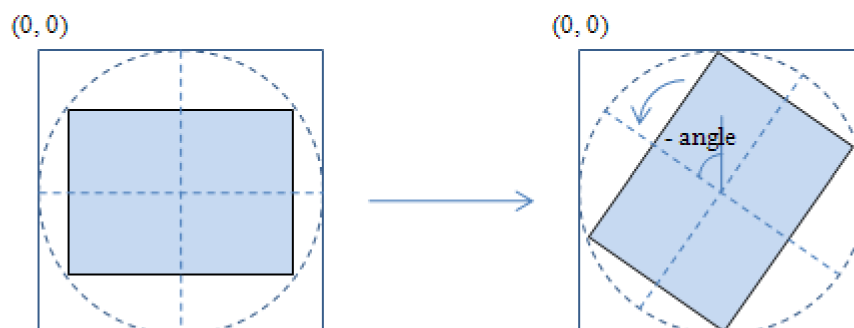


Figure 10. Rotating the Land Image.

If the image extends beyond the dotted circle then there is a risk that a part of the picture will be lost when it is rotated.

 © Andrew Davison 2009

As a consequence, the London map image used by TourRotLandPanel is somewhat different from the one employed in TourLandPanel: there's extra 'padding' around the map so all the details fall inside the rotation circle (see Figure 11).

Figure 11. The Modified London Map

The map's edges no longer coincide with the edges of the file, as they did in TourLandPanel. Now the top-left and bottom-right corners of the map are located at the (MIN_X, MIN_Y) and (MAX_X, MAX_Y) coordinates in the image, which explains their use in the revised version of move() to constrain car movement.

3.5. Initializing the Land Map

RotLandMap begins by storing a copy of the panel dimensions, and loading the land image.

```
// globals
private int pWidth, pHeight;        // panel dimensions

// land image info
private BufferedImage landIm;
private int landWidth, landHeight;


public RotLandMap(String imFnm, int pW, int pH)
{
  pWidth = pW;
  pHeight = pH;

  landIm = loadImage(imFnm);    // load land image
  landWidth = landIm.getWidth();
  landHeight = landIm.getHeight();
} // end of RotLandMap()
```

 © Andrew Davison 2009

3.6. Drawing the Land

When the player moves, it's really the land that shifts underneath the stationary player. The image is rotated and translated by RotLandMap.draw():

```
public void draw(Graphics g, double xPlayer, double yPlayer,
                                                    int angle)
{ // rotation around image center
  AffineTransform at = AffineTransform.getRotateInstance(
                            Math.toRadians(-angle), // -ve rotation
                            landWidth/2, landHeight/2);

  BufferedImage rotIm = rotateImage(landIm, at, angle);
                  // rotate the image
  Point rotCoord = calcRotatedPosn(at, xPlayer, yPlayer, angle);
                  // also apply the rotation to (xPlayer,yPlayer)

  /* (xIm, yIm) is the top-left drawing position on
     the land image in the JPanel such that
     (rotCoord.x, rotCoord.y) is at the center of the JPanel */
  int xIm = pWidth/2 - rotCoord.x;
  int yIm = pHeight/2 - rotCoord.y;

  g.drawImage(rotIm, xIm, yIm, null);
        // draw the rotated land, translated to (xIm, yIm)
}  // end of draw()
```

The affine transformation defines a negative rotation around the image's center because the land must rotate in the *opposite* direction to the car's supposed turn.

rotateImage() applies the transformation to the image:

```
private BufferedImage rotateImage(BufferedImage im,
                                  AffineTransform at, int angle)
{ BufferedImage rotIm = im;
  if (angle != 0) {  // only do this if the image needs rotating
    AffineTransformOp rotateOp =
        new AffineTransformOp(at, AffineTransformOp.TYPE_BILINEAR);
    rotIm = rotateOp.filter(im, null);
  }
  return rotIm;
}  // end of rotateImage()
```

The rotation must be followed by an image translation such that the player's (xPlayer, yPlayer) land position coincides with the car's drawing coordinate at the center of the JPanel.

 © Andrew Davison 2009

The problem is that (xPlayer, yPlayer) will have changed when the image was rotated, and so it's necessary to calculate it's new value after the rotation. This rotation is illustrated by the red dot in Figure 12.

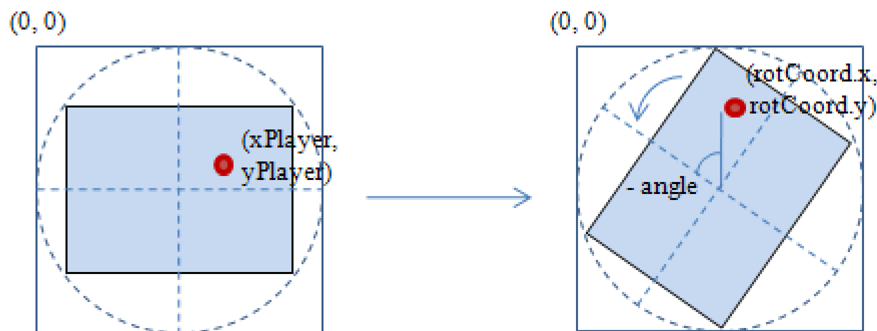


Figure 12. The Land Image Before and After a Rotation.

In Figure 12, the player coordinate has rotated to a position denoted by (rotCoord.x, rotCoord.y). It is quite easy to calculate this point, by applying the affine transformation used on the image to (xPlayer, yPlayer), as implemented in calcRotatedPosn():

```
private Point calcRotatedPosn(AffineTransform at,
                   double xPlayer, double yPlayer, int angle)
{
  Point rotCoord = new Point();

  if (angle != 0) {  // only do this if the position needs rotating
    // rotate the player coordinate
    Point2D.Double rotPt =
        (Point2D.Double) at.transform(
             (Point2D)( new Point2D.Double(xPlayer, yPlayer)), null);
    rotCoord.x = (int) rotPt.x;
    rotCoord.y = (int) rotPt.y;
  }
  else {
    rotCoord.x = (int) xPlayer;
    rotCoord.y = (int) yPlayer;
  }
  return rotCoord;
}  // end of calcRotatedPosn()
```

The rotated coordinate is used to calculate the translation required for the image so that the player is drawn at the center of the JPanel. The relevant code in draw() is:

```
int xIm = pWidth/2 - rotCoord.x;
int yIm = pHeight/2 - rotCoord.y;
g.drawImage(rotIm, xIm, yIm, null);
```

The relationship between the rotated land coordinate and the JPanel is shown in Figure 13.

 © Andrew Davison 2009

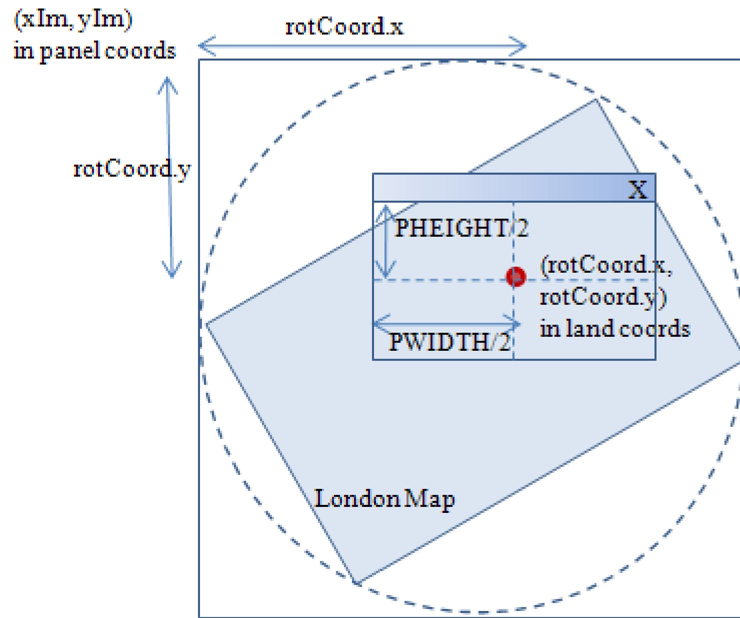
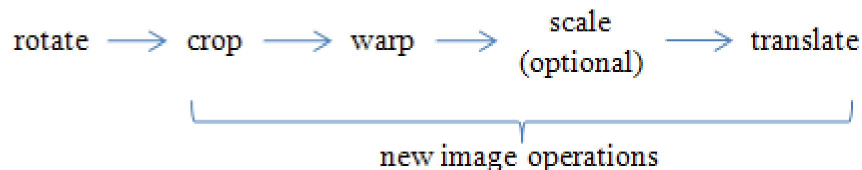


Figure 13. Positioning the Rotated Land Image.

4. TourPerLandPanel: Driving Down the Highway (Version 3)

I'm entering the last lap my touring code. In TourRotLandPanel, the image was rotated; in this last version I'll add *four* more operations:



Warping isn't available in Java 2D, so I utilize the Java Advanced Imaging (JAI) library for that operation. Scaling is optional since its effect could be achieved through more specialized warping, but I've found it better to separate the two to make them easier to understand and 'tweak'.

Before going into the details of the four new operations, I'll briefly summarize JAI.

4.1. JAI Overview

JAI (https://jai.dev.java.net/) offers numerous image processing capabilities beyond those found in Java 2D . For example, its geometric operations including shearing, transposition, and warping. Its pixel-based functions utilize lookup tables and rescaling equations, and can be applied to multiple sources, and be combined into a single outcome. Modifications can be restricted to regions in the source, statistical operations are available (e.g. mean and median), along with frequency domains.

 © Andrew Davison 2009

The current stable JAI version of is 1.1.3 (as of January 2009), but daily early access builds of 1.1.4 are available. I downloaded jai-1_1_3-lib-windows-i586-jdk.exe for Windows, which installs libraries into the JDK that offer native acceleration for many operations.

The JAI website (https://jai.dev.java.net/) includes pointers to demos and tutorials, including an online book "Programming in Java Advanced Imaging" (dating from 1999) at http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc/JAITOC.fm.html.

The JAI API documentation can be found at http://java.sun.com/products/java-media/jai/forDevelopers/jai-apidocs/,or it can be downloaded from http://download.java.net/media/jai/builds/release/1_1_3/jai-1_1-mr-doc.zip.

There's an active JAI forum at http://forums.java.net/jive/forum.jspa?forumID=75, and a good collection of examples at https://jaistuff.dev.java.net/, including a 30 page tutorial at https://jaistuff.dev.java.net/docs/jaitutorial.pdf.

The JAIStuff site is no longer being maintained, but its author, Rafael Santos, has been writing an image processing online book at http://www.lac.inpe.br/~rafael.santos/JIPCookbook/, which uses JAI in many of its examples.

A fairly recent introductory article on JAI can be found at http://java.sun.com/developer/JDCTechTips/2005/tt0601.html, which examines a few basic image transformations. There is a chapter on JAI in *Java Media APIs: Cross-Platform Imaging, Media and Visualization* by Alejandro Terrazas, John Ostuni, Michael Barlow, Sams 2002.

JAI Image I/O Tools (https://jai-imageio.dev.java.net/) is a related library which provides readers, writers, and stream plug-ins for JAI, including support for the BMP, JPEG,, PNG, PNM, Raw, TIFF, and WBMP formats. I didn't need it for this example.

Using JAI

JAI is a complicated library, but much of its coding follows the same three steps:

1. Set up a ParameterBlock consisting of image input sources, and assorted operation parameters.
2. Call JAI.create() with the operation name and ParameterBlock.
3. Save the result as a new image, or as a source for a subsequent image operation.

For example, loading an image named test.jpg requires:

```
ParameterBlock pb = new ParameterBlock();
pb.add("test.jpg");
PlanarImage image = JAI.create("fileload", pb);
```

PlanarImage is the basic JAI class for storing images.

Once the image is available, another ParameterBlock can be employed to scale it by a factor of 2 in the x- and y- directions:

 © Andrew Davison 2009

```
pb = new ParameterBlock();    // reuse pb
pb.addSource(image);
pb.add(2.0f);  // scale by 2 along x-axis
pb.add(2.0f);  // scale by 2 along y-axis
PlanarImage bigImage = JAI.create("scale", pb);
```

4.2. JAI Warping

Java 2D affine transformations, such as translation, scaling, and rotation, share the property that parallel lines in the image stay parallel, although lengths and angles may change. Warping is more general since it can arbitrarily stretch an image about defined points. Another, more descriptive, name for warping is *rubber sheet deformation*.

JAI supports several warping functions, including polynomial warping, grid warping, and perspective warping (which I'll be using). Perspective (or projective) warping applies a perspective-type distortion to an image, which can, for example, reduce the scale of objects that appear at the top of an image (see Figure 14).

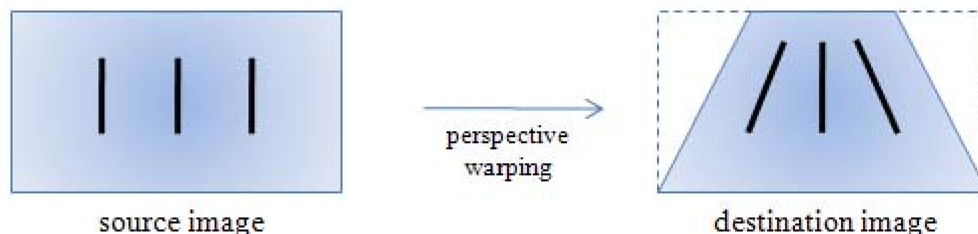


Figure 14. Perspective Warping Example.

This type of distortion can be used to mimic the look of objects in a 3D scene receding into the distance (see the right hand image of Figure 1 or Figure 15).

The JAI WarpPerspective class supports perspective warping. Rather confusingly, the operation is defined in terms of a mapping *from* the destination image *to* the source, but this is quite easy to specify using methods from the JAI PerspectiveTransform class.

4.3. The Touring Panel Again

The third version of the Touring Panel (called TourPerLandPanel; see Figure 2) is very similar to the second version (TourRotLandPanel) since the majority of the changes to the map manipulation are hidden inside the PerLandMap class.

TourPerLandPanel.initImages() initializes the PerLandMap object, and sets the (xDrawPos, yDrawPos) and (xPlayer, yPlayer) coordinates:

```
// globals
private static final String LAND_IM = "londonMap2.gif";
private static final String PLAYER_IM = "smallRacer.png";
            // the back of a racing car
```

 © Andrew Davison 2009

```
private PerLandMap landMap;    // land info

// player info
private BufferedImage playerIm;
private int xDrawPos, yDrawPos;  // drawing coord for player (fixed)

private double xPlayer, yPlayer;  // player position on the land
private int angle;                // angle to the vertical


private void initImages()
// initialize land and player images, and player's coordinates
{
  landMap = new PerLandMap(LAND_IM);

  Dimension panelDim = landMap.getPanelDim(); // set panel size
  setPreferredSize(panelDim);

  playerIm = loadImage(PLAYER_IM);  // load player image

  // calculate fixed drawing position for player
  //    - resting on the bottom edge of panel, in the middle
  xDrawPos = panelDim.width/2 - playerIm.getWidth()/2;
  yDrawPos = panelDim.height - playerIm.getHeight();

  // the player's starting position on the land
  Point2D.Double startPt = landMap.getStartPos();
  xPlayer = startPt.x;
  yPlayer = startPt.y;

  angle = 0;   // player faces north
}  // end of initImages()
```

The complex series of operations carried out by PerLandMap change the size of the land image quite drastically, so the class includes PerLandMap.getPanelDim() to return the panel dimensions based on the new image size. Also, PerLandMap.getStartPos() returns the starting coordinate for the player on the map.

A visible change in TourPerLandPanel is the new car image, and the car's new position resting on the bottom edge of the panel in the middle (see Figure 15).

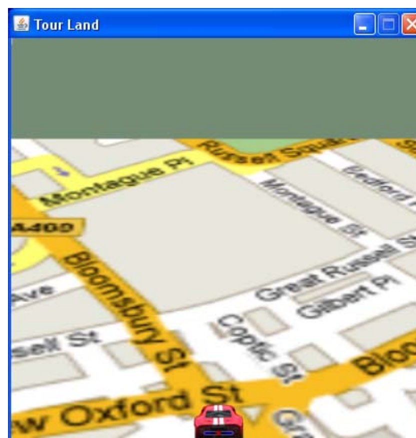


Figure 15. Starting Position for the Player in TourPerLandPanel.

 © Andrew Davison 2009

(xPlayer, yPlayer) and angle are changed as before – via calls to move() and turn() triggered by key presses. However, paintComponent() now calls PerLandMap.draw():

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;

  landMap.draw(g2d, xPlayer, yPlayer, angle);     // draw land
  g2d.drawImage(playerIm, xDrawPos, yDrawPos, null);  // draw player
}
```

The graphics object is cast to a Graphics2D instance since draw() uses rendering hints and an AffineTransform.

4.4. Initializing the Land Map

PerLandMap calculates the panel dimensions, loads the map, and sets up the warp operation.

```
// globals
// crop dimensions
private static final int CROP_WIDTH = 200;
private static final int CROP_HEIGHT = 300;

// warp inset along x-axis of top edge
private static final int INSET = 50;  // 2*INSET must be < CROP_WIDTH

// scale factors
private static final float SCALE_WIDTH = 4.0f;
private static final float SCALE_HEIGHT = 1.3f;


// panel dimensions
private int pWidth, pHeight;

// land image info
private BufferedImage landIm;
private int landWidth, landHeight;


public PerLandMap(String imFnm)
{
  // calculate panel dimensions
  pWidth = (int)((CROP_WIDTH - 2*INSET) * SCALE_WIDTH);
  pHeight = (int)(CROP_HEIGHT * SCALE_HEIGHT);

  landIm = loadImage(imFnm);   // load land image
  landWidth = landIm.getWidth();
  landHeight = landIm.getHeight();

  createWarpOp();
} // end of PerLandMap()
```

 © Andrew Davison 2009

The bewildering set of constants for cropping, scaling, and warp insets will be explained below, as will the calculation of the panel width and height at the start of PerLandMap(). The values for the constants were arrived at by a process of trial and error, until the resulting perspective view of the map looked most like a 3D view.

createWarpOp() instantiates a WarpPerspective object that will be used each time the image is drawn:

```
// globals
private static final int INSET = 50;
                 // 2*INSET must be < CROP_WIDTH

private WarpPerspective warpOp;   // for warping the land image


private void createWarpOp()
{
  // create a perspective transformation
  float w = CROP_WIDTH;
  float h = CROP_HEIGHT;
  PerspectiveTransform perspTrans =
              PerspectiveTransform.getQuadToQuad(
                   0,0,     w,0,        w,h, 0,h,    // source to
                   INSET,0, w-INSET,0,  w,h, 0,h  ); // dest

  // warp the top edge of the image inwards along the x-axis
  warpOp = null;
  try {
    warpOp = new WarpPerspective( perspTrans.createInverse() );
                            // invert the transform
  }
  catch(Exception e)
  {  System.out.println(e);
     System.exit(1);
  }
}  // end of createWarpOp()
```

The perspective transformation is defined in terms of how the four corners of a source quadrilateral are mapped to a resulting quadrilateral, as illustrated by Figure 16.

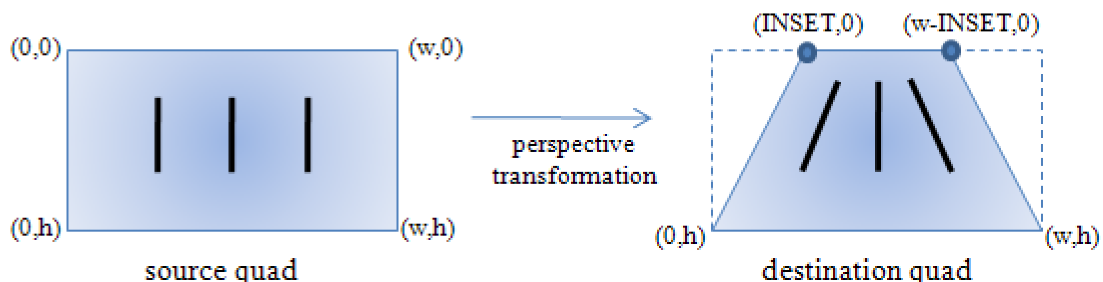


Figure 16. A Perspective Transformation.

The WarpPerspective constructor requires a destination-to-source mapping, but this is easily obtained by calling PerspectiveTransform.createInverse().

 © Andrew Davison 2009

4.5. Drawing the Scene

The draw() method in PerLandMap applies five operations to the land image:

rotate → crop → warp → scale → translate

The rotation code is unchanged from RotLandMap.draw().

```
public void draw(Graphics2D g2d, double xPlayer, double yPlayer,
                                  int angle)
{ // smooth any output
  g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
              RenderingHints.VALUE_INTERPOLATION_BILINEAR);

  // rotation around image center
  AffineTransform at = AffineTransform.getRotateInstance(
                          Math.toRadians(-angle),    // -ve rot
                          landWidth/2, landHeight/2);

  BufferedImage rotIm = rotateImage(landIm, at, angle);
  Point rotCoord = calcRotatedPosn(at, xPlayer, yPlayer, angle);

  BufferedImage cropIm = cropImage(rotIm, rotCoord); // crop
  BufferedImage warpIm = warpImage(cropIm);          // warp

  // draw a scaled and translated version of the land
  drawScaleTrans(g2d, warpIm);
} // end of draw()
```

The rotation of the image by rotateImage(), and the calculation of the rotated (xPlayer, yPlayer) position by calcRotatedPosn() are the same as in RotLandMap.

The outcome is an image in rotIm something like that shown in Figure 17.

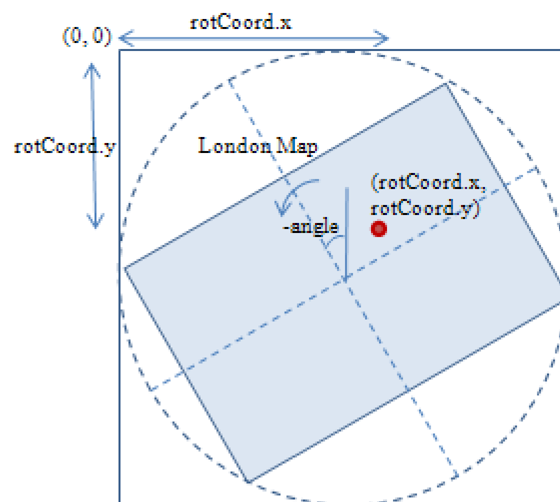


Figure 17. The Rotated Map.

© Andrew Davison 2009

The perspective warping is meant to mimic a 3D view of the scene from the player's position at the rotCoord coordinate. This requires the image to be cropped so that rotCoord is located on the bottom edge, centrally located as in Figure 18.

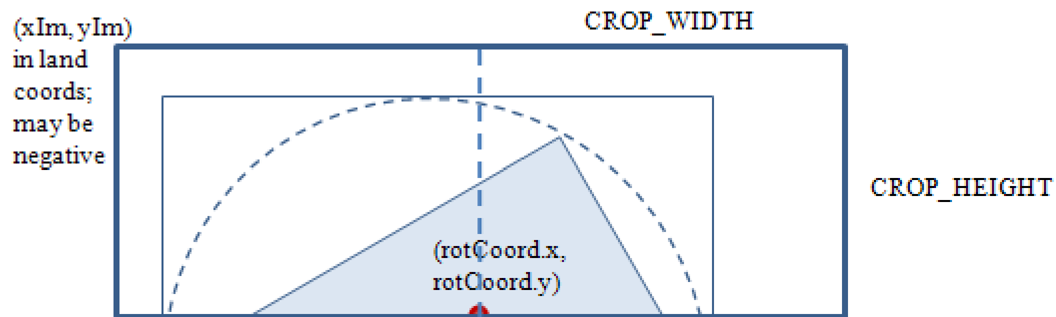


Figure 18. The Cropped Map.

The code for calculating (xIm, yIm) in Figure 18 is in cropImage():

```
// global crop dimensions
private static final int CROP_WIDTH = 200;
private static final int CROP_HEIGHT = 300;


private BufferedImage cropImage(BufferedImage im, Point rotCoord)
/* Clip the image to as close to the required crop dimensions
   as possible, then 'fill in' the clipped image edges so it
   has the required crop dimensions. */
{
  int xIm = rotCoord.x - CROP_WIDTH/2;
  int yIm = rotCoord.y - CROP_HEIGHT;

  BufferedImage clipIm = clipImage(im, xIm, yIm);
  return fillEdges(clipIm, xIm, yIm);
}  // end of cropImage()
```

The required CROP_WIDTH and CROP_HEIGHT dimensions may well be bigger than the land image, causing (xIm, yIm) to be negative. In that case, clipImage() returns the largest possible image clip, and fillEdges() draws the under-sized clip into a new BufferedImage of the required crop dimensions, filling in any empty parts with the grey green background colour.

 © Andrew Davison 2009

For example, clipImage() may return an image something like the one in Figure 19 which is smaller than the requested crop dimensions.

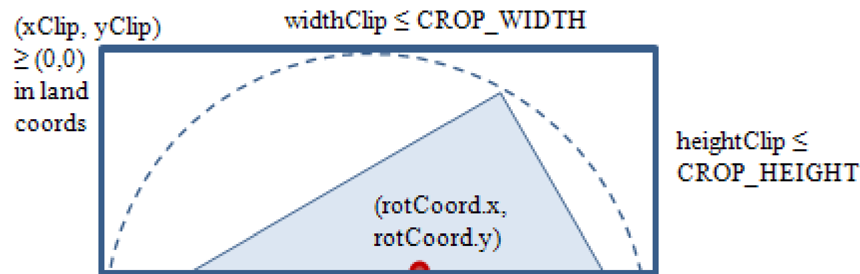


Figure 19. A Clipped Image, near to the Required Crop Dimensions.

The code for clipImage():

```
private BufferedImage clipImage(BufferedImage im, int xIm, int yIm)
/* Extract the part of the land image that falls within the
   crop boundaries. This may be CROP_WIDTH x CROP_HEIGHT in size,
   or perhaps smaller. */
{
  // calculate the top-left coord of the subimage
  int xClip = (xIm < 0) ? 0 : xIm;     // tests to prevent -ve values
  int yClip = (yIm < 0) ? 0 : yIm;

  // get the width and height of the subimage
  int widthClip, heightClip;
  if (xClip + CROP_WIDTH > im.getWidth())
    widthClip = im.getWidth() - xClip;
        // subimage width cannot go beyond image width
  else
    widthClip = CROP_WIDTH;

  if (yClip + CROP_HEIGHT > im.getHeight())
    heightClip = im.getHeight() - yClip;
        // subimage height cannot go beyond image height
  else
    heightClip = CROP_HEIGHT;

  return im.getSubimage(xClip, yClip, widthClip, heightClip);
}  // end of clipImage
```

fillEdges() pads out the clipped image, so rotCoord is drawn at the center of the bottom edge of the image, and the resulting image has the necessary CROP_WIDTH by CROP_HEIGHT dimensions (e.g. as in Figure 20).

 © Andrew Davison 2009

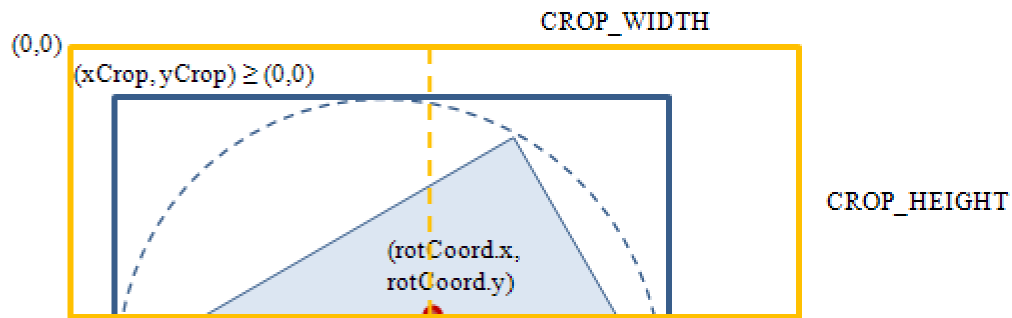


Figure 20. The 'Padded Out' Clipped Image.

The fillEdges() method is:

```
// global
// background colour of the land
private static final Color LAND_BG_COLOUR =  new Color(115,140,115);


private BufferedImage fillEdges(BufferedImage clipIm,
                                           int xIm, int yIm)
/* Draw the clip image into a new BufferedImage of the required
   crop dimensions, filling in any blank parts with the
   background colour, and positioning the image correctly. */
{
  // store clipIm into a CROP_WIDTH x CROP_HEIGHT size image
  BufferedImage cropped = new BufferedImage(CROP_WIDTH, CROP_HEIGHT,
                                  BufferedImage.TYPE_INT_ARGB);
  Graphics g = cropped.getGraphics();

  // fill the image with the background colour
  g.setColor( LAND_BG_COLOUR );
  g.fillRect(0, 0, CROP_WIDTH, CROP_HEIGHT);

  /* Position the clipped image to deal with the chance that
     (xIm, yIm) are negative. In that case, the clip should be
     drawn moved to the right and down to include empty
     background space at the top left. */
  int xCrop = (xIm < 0) ? -xIm : 0;
  int yCrop = (yIm < 0) ? -yIm : 0;
  g.drawImage(clipIm, xCrop, yCrop, null);    // draw clip

  return cropped;
}  // end of fillEdges()
```

All this work results in an image whose dimensions are guaranteed to be CROP_WIDTH by CROP_HEIGHT, with the rotCoord coordinate positioned at the center of the bottom edge of the image.

The standard dimensions makes it possible to reuse the same WarpPerspective operation every time the map is warped – a handy speed optimization for the code.

 © Andrew Davison 2009

The image is perspective warped in the vertical direction, with its top edge squeezed in by INSET pixels on the left and right, as shown in Figure 21.

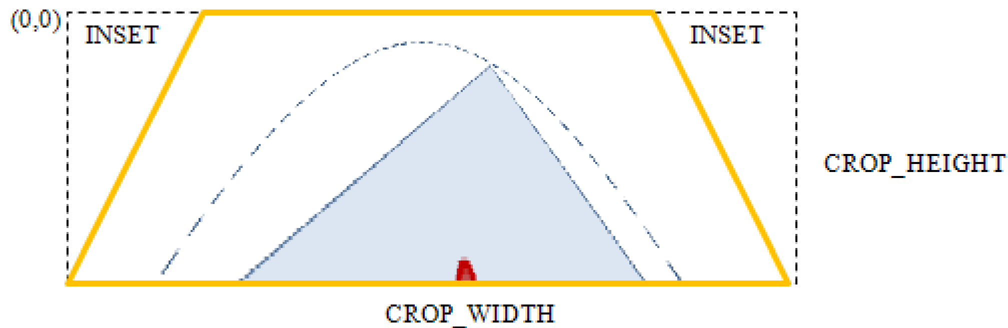


Figure 21. The Warped Image.

warpImage() sets up a ParameterBlock using the existing WarpPerspective operation, and applies it to the image.

```
// global
private WarpPerspective warpOp;


private BufferedImage warpImage(BufferedImage im)
/* warp the top of the image inwards by INSET on the
   top-left and top-right corners */
{
  ParameterBlock pb = new ParameterBlock();
  pb.addSource( PlanarImage.wrapRenderedImage(im) );
                              // BufferedImage --> PlanarImage
  pb.add( warpOp );
  pb.add( new InterpolationBilinear() );   // smooth
  PlanarImage warpedImage = JAI.create("warp", pb);

  BufferedImage warpIm = null;
  try {       // PlanarImage --> BufferedImage
    warpIm = warpedImage.getAsBufferedImage();
  }
  catch (Exception e)
  { System.out.println(e); }

  return warpIm;
}  // end of warpImage()
```

An inconvenience managed by warpImage() is the need to convert the map's BufferedImage into a PlanarImage before warping, and change the resulting PlanarImage into a BufferedImage afterwards.

The quality of the image is noticeably improved by specifying bilinear interpolation smoothing as part of the operation rather than relying on the default nearest pixel interpolation.

The image's scaling step could be incorporated into the PerspectiveTransform, but it's more useful to separate it out so that different scale values can be tried during code

 © Andrew Davison 2009

development. The most convincing 3D-like view seems to require a stretching of the x-axis, while the y-axis is left pretty much unchanged. The hard-wired scale factors (called SCALE_WIDTH and SCALE_HEIGHT) result in an image like the one in Figure 22.

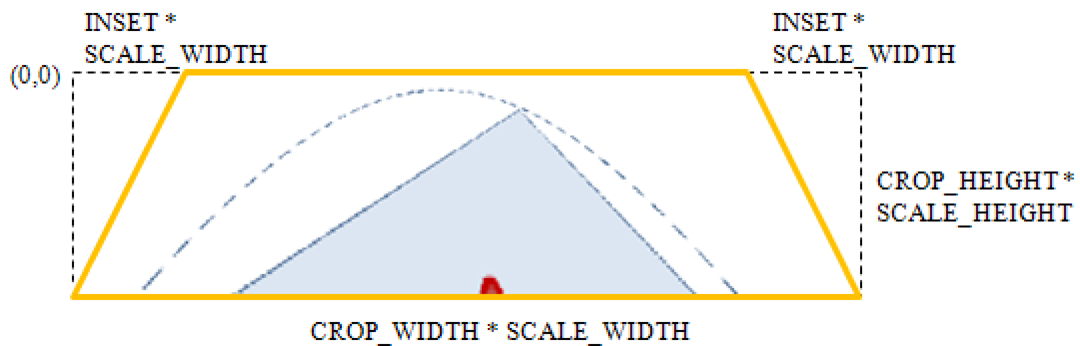


Figure 22. The Scaled Image.

When the image is rendered into the panel, its angled sides shouldn't be drawn, which is simply achieved by drawing the image translated to the left by INSET*SCALE_WIDTH pixels, and by specifying the panel width as (CROP_WIDTH - 2*INSET) * SCALE_WIDTH (see Figure 23).

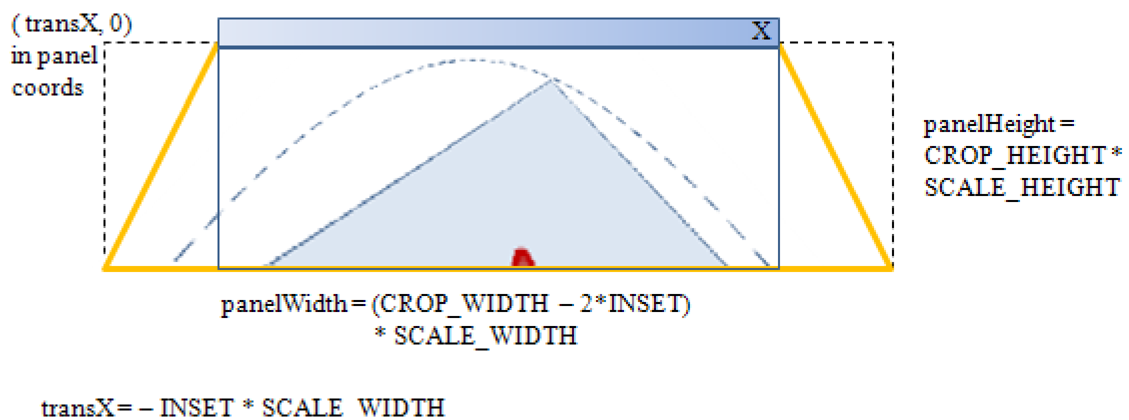


Figure 23

. The Translated Image.

The scaling and translation are carried out together in drawScaleTrans():

```
// globals
// scale factors
private static final float SCALE_WIDTH = 4.0f;
private static final float SCALE_HEIGHT = 1.3f;

private void drawScaleTrans(Graphics2D g2d, BufferedImage im)
// draw a scaled and translated version of the image
{
  int newW = (int)(CROP_WIDTH * SCALE_WIDTH);
```

 © Andrew Davison 2009

```
  int newH = (int)(CROP_HEIGHT * SCALE_HEIGHT);

  int transX = (int)(-INSET * SCALE_WIDTH);
     /* draw image starting off to the left of the panel,
        so slanted sides are invisible */

  g2d.drawImage(im, transX, 0, newW, newH, null);
}  // end of drawScaleTrans()
```

This transformation finally explains the reasoning behind the initialization of the panel height and width values in the PerLandMap constructor():

```
// calculate panel dimensions in PerLandMap()
pWidth = (int)((CROP_WIDTH - 2*INSET) * SCALE_WIDTH);
pHeight = (int)(CROP_HEIGHT * SCALE_HEIGHT);
```

These values are accessed via PerLandMap.getPanelDim(), which is called from TourPerLandPanel to set the panel's dimensions:

```
public Dimension getPanelDim()
// return the panel dimensions
{  return new Dimension(pWidth, pHeight);  }
```

 © Andrew Davison 2009