Chapter 13.9. Perspective Transformations

Java supports the well-known *affine* operations – translation, scaling, and rotation, so named because parallel lines in the image stay parallel after a transformation, although lengths and angles may change. But there are occasions when affine transformations aren't enough, especially in many computer vision applications where a *perspective* view of a scene needs to be manipulated. The perspective transformation is known by various names, including perspective warping, projectivity, collineation, and homography. A simple example is shown in Figure 1.

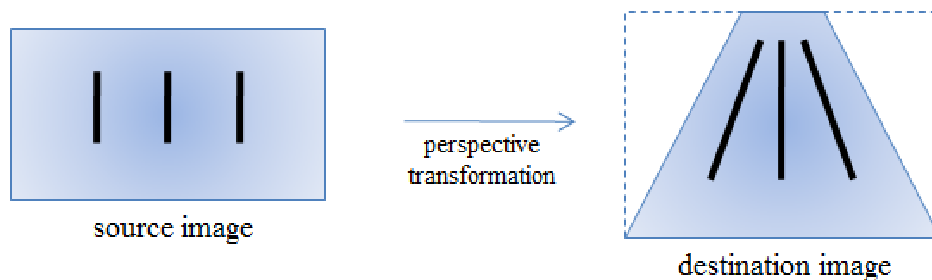


Figure 1. A Perspective Transformation Example.

A computer vision related way of understanding the transformation is in terms of adjusting the view of a 3D scene. In Figure 1, the viewpoint in the source image is facing into the scene, while the destination image has moved the viewpoint so it is 'looking' at the scene from an angle.

A common use for perspective transformation is to 'correct' an angled view so the viewpoint is pointing directly into the scene (as in Figure 2).

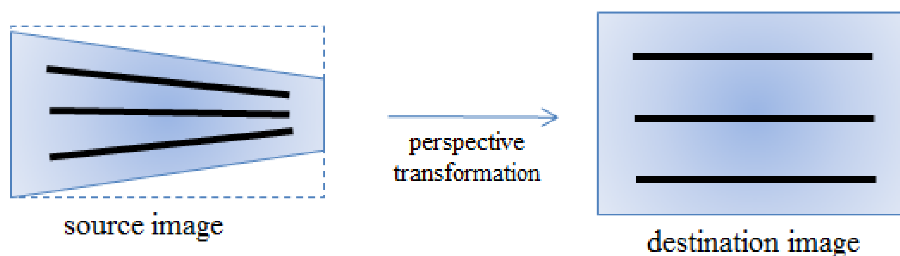


Figure 2. Another Perspective Transformation Example.

I've implemented this kind of transformation in my Warper application (see Figure 3). The user selects four points in the left-hand image, and the quadrilateral is transformed into a rectangle (or square) displayed in the right-hand panel.

© Andrew Davison 2014

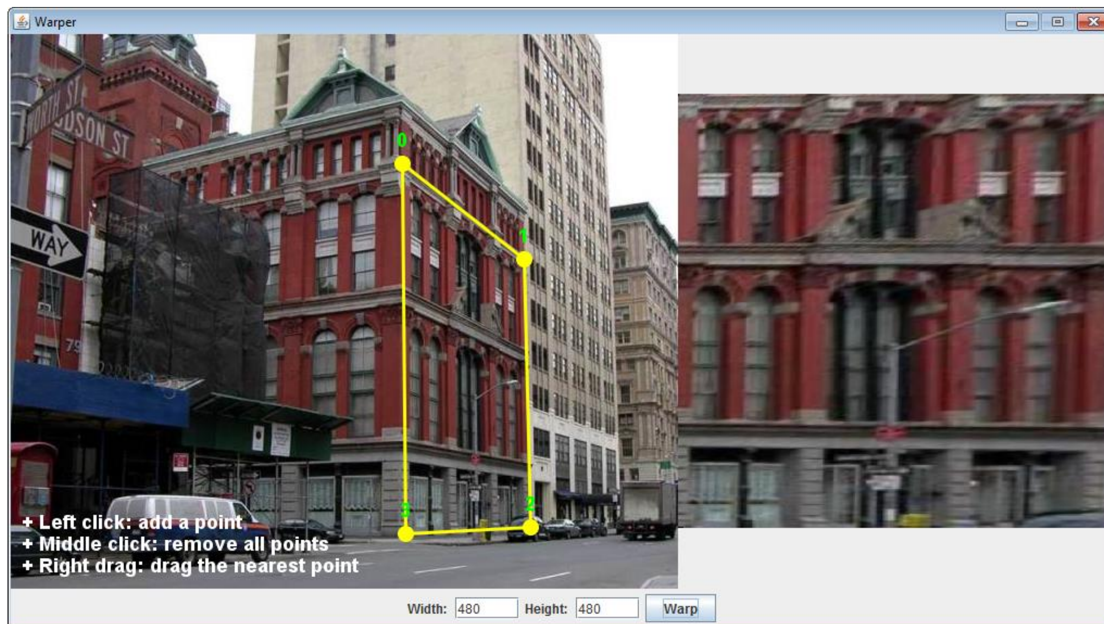


Figure 3. The Warper Application.

In terms of viewpoints, the camera position in the source image is off to the left while in the destination image the viewpoint is facing the building.

This viewpoint explanation of the perspective transformation is only approximate since changing the view is a 3D operation, while the transformation is only being applied to a 2D image. For example, a truly 3D viewpoint change may reveal previously obscured parts of the scene, but a perspective transformation is only 'distorting' the image, so no new scenery features can be revealed. This difference is illustrated by Figure 4, where a viewpoint change would reveal more of the man to the right of the box, but the perspective transformation shows nothing new.

 © Andrew Davison 2014

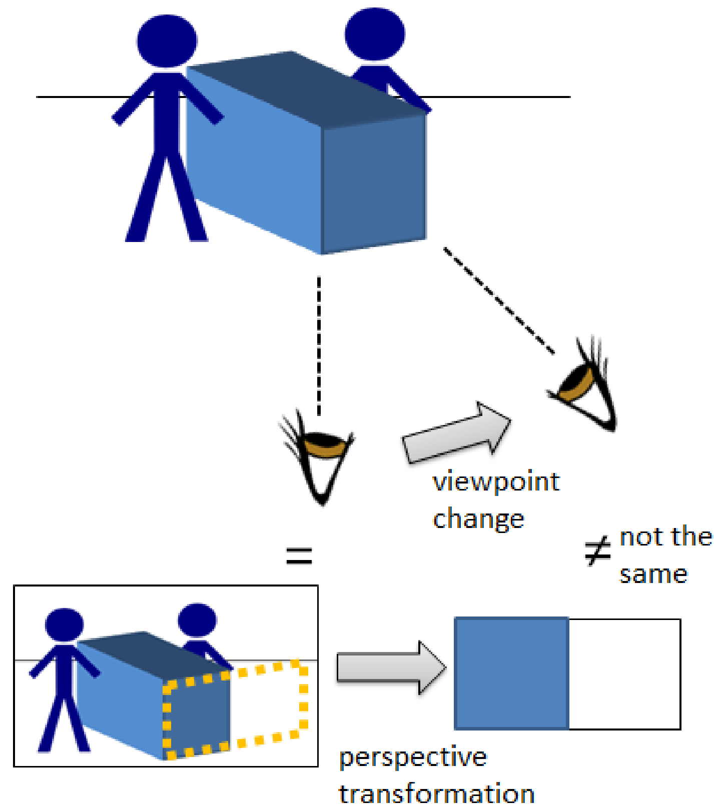


Figure 4. Viewpoint Change and Perspective Transformation.

Nevertheless, perspective transformation is a useful computer vision tool, especially if the scene doesn't have too many obscured regions and/or the transformation only changes the viewpoint by a small amount. For instance, the perspective change applied in Figure 3 is to the face of a building, which is mostly flat, and so the new 'viewpoint' looks mostly correct.

Perspective transformation isn't available in Java, so I'll be employing the Java Advanced Imaging (JAI) library (where it's known as perspective *warping*). JAI supports several interesting kinds of warping, including polynomial and grid warping, which allows for more flexible distortion effects to be applied to an image.

An alternative approach to the JAI WarpPerspective class is the PerspectiveTransform class in JavaFX, but this would require me to utilize a JavaFX scene graph. JAI seems better suited for my image processing needs.

Before going into the details of the Warper application, I'll briefly summarize the features of JAI.

1. JAI: Not Dead Yet

JAI (http://www.oracle.com/technetwork/java/current-142188.html) 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 result. Modifications can be restricted to

 © Andrew Davison 2014

regions in the source, and statistical operations are available (e.g. mean and median), along with frequency domains.

The current JAI version is 1.1.3 (released in January 2006), and the libraries for different platforms are located at http://download.java.net/media/jai/builds/release/1_1_3/. I downloaded jai-1_1_3-lib-windows-i586-jdk.exe for Windows, which offer native acceleration for many operations. The API documentation (only current up to v1.1) can be found at the same place, as the file jai-1_1-mr-doc.zip.

The JAI website includes links to demos and tutorials, but many of those links have 'died'. A very useful resource that's still around is the book "Programming in Java Advanced Imaging" (dating from 1999) at http://docs.oracle.com/cd/E19957-01/806-5413-10/806-5413-10.pdf.

As you may be starting to realize, JAI is another example of a Java media library that time (or at least Oracle) has forgotten. It's a real shame since it contains some very useful capabilities. I'm not the only person who thinks this, since the JAI forum (at https://www.java.net/forums/javadesktop/jai) is very active.

The no-longer maintained JAIStuff site at https://java.net/projects/jaistuff contains a good collection of examples, although they're buried deep inside the Subversion code manager.

Rafael Santos has been writing an image processing book online at http://www.lac.inpe.br/JIPCookbook/, which uses JAI for many of its examples. He's also produced an excellent 30 page tutorial on JAI, which can be downloaded from http://www.lac.inpe.br/JIPCookbook/Resources/Docs/jaitutorial.pdf.

There's 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 (available with the rest of JAI at http://www.oracle.com/technetwork/java/current-142188.html) provides readers, writers, and stream plug-ins, including support for the BMP, JPEG, PNG, PNM, Raw, TIFF, and WBMP formats. I didn't need it for my Warper application.

As I mentioned above, the Windows version of JAI offers native acceleration, but there are some catches. The media libraries (DLLs) that implement these capabilities are 32-bit, which means they cannot be called by a 64-bit Java run-time. This doesn't 'kill' JAI, but makes it fall back to a pure Java implementation of the graphic operations, which is typically 10-15% slower.

You're notified about this, by the printing of the message:

```
Error: Could not load mediaLib accelerator wrapper classes.
Continuing in pure Java mode.
```

For more details, see http://stackoverflow.com/questions/5207145/jai-and-imageio-for-64-bit-windows

2. Using JAI

JAI is a complicated library, but most JAI coding follows the same three steps:

 © Andrew Davison 2014

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 instance, 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:

```
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);
```

The JAILoader.java example given below loads a specified image and displays it in a panel, as shown in Figure 5.

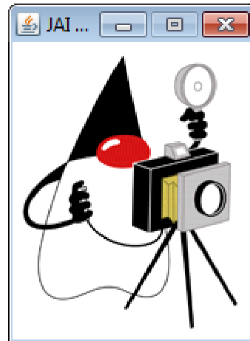


Figure 5. The JAILoader Application.

The constructor for the JAILoader class (which is a subclass of JFrame):

```
public JAILoader(String fnm)
{
  super("JAI Loader: " + fnm);

  // load the image
  ParameterBlock pb = new ParameterBlock();
  pb.add(fnm);
  PlanarImage plim = JAI.create("fileload", pb);

  // print information about the image
  System.out.println("Dimensions: " + plim.getWidth()+ "x" +
                                      plim.getHeight() +
```

```
                                    "; Bands:" + plim.getNumBands());

  Container c = getContentPane();
  c.add( new DisplayJAI(plim) );    // display image in a panel

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setResizable(false);
  pack();
  setVisible(true);
}  // end of JAILoader()
```

The application uses the JAI DisplayJAI class: a JPanel that can render PlanarImage objects.

JAI has specialized methods for many operations, which allow the programmer to employ less ParameterBlock objects. For instance, it's possible to load an image with a single line of code:

```
      PlanarImage plim = JAI.create("fileload", fnm);
```

instead of the three lines used in the method above.

3. Perspective Transformations with Warper

The Warper application requires the user to specify an image at run-time, which is displayed on the left-hand side of the window.

The JAI WarpPerspective class defines a perspective transformation in terms of source and destination quadrilaterals, as in Figure 6.

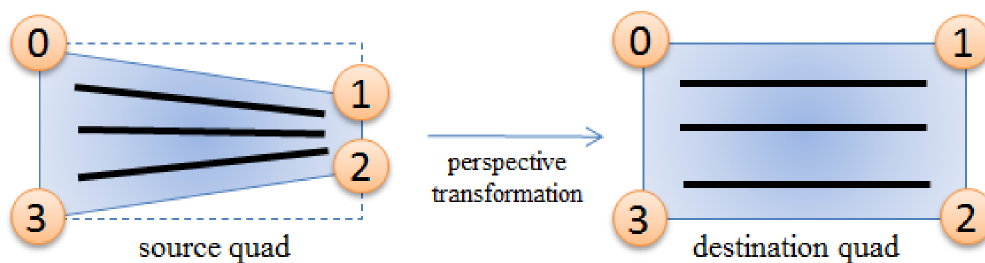


Figure 6. Perspective Transformation in terms of Quadrilaterals.

Fortunately, since Warper always generates a rectangle (or square), its possible to reduce the user's workload. The user has to select four points for the source quadrilateral, but only supply a width and height for the destination quad (Warper assumes that the quad's top-left point is at (0,0)). This simplification is shown in Figure 7.

© Andrew Davison 2014

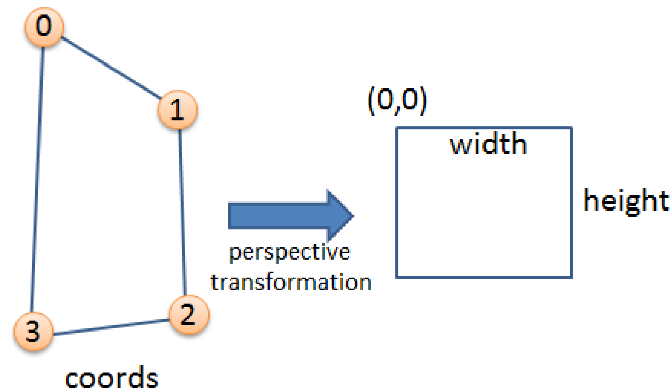


Figure 7. Perspective Transformation Details Used by Warper.

Figure 8 shows another example of Warper being used.

Figure 8. Another Warper Example.

The user selects four points in the left-hand image, and perhaps adjusts the default width and height text field values before pressing the "Warp" button. It's possible to move the points, or delete them, and start again.

The UML class diagrams for Warper in Figure 9 shows its GUI structure.

© Andrew Davison 2014

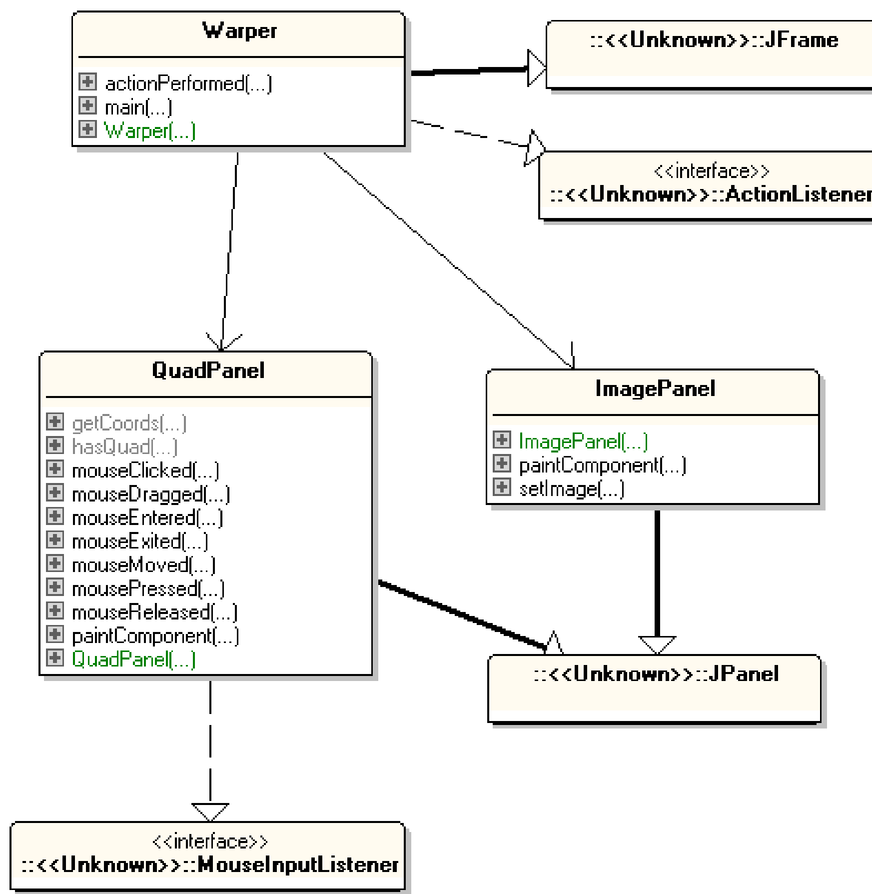


Figure 9. Class Diagrams for Warper.

Warper utilizes two JPanel subclasses: QuadPanel for the left-hand image and ImagePanel for the transformed right-hand image. Both classes employ standard Java code, with QuadPanel being the more interesting since it deals with mouse input:

- a left mouse button click adds a point, with the assumption that the points define a quadrilateral in clockwise-order starting from the top-left;
- a middle mouse button (wheel) click removes all the points;
- a right mouse button drag moves the nearest point, so the quadrilateral can be adjusted.

I won't explain the details of the coding since it doesn't have much to do with perspective transformation; the classes are documented if you want more information.

The width and height text fields and the "Warp" button are implemented in the Warper class. When the user presses the button, Warper's actionPerformed() method is called. It retrieves the quadrilateral corners from QuadPanel and the width and height values, and carries out the perspective transformation. The details are a little messy (as we'll see), but there are three main steps, as shown in Figure 10.

 © Andrew Davison 2014

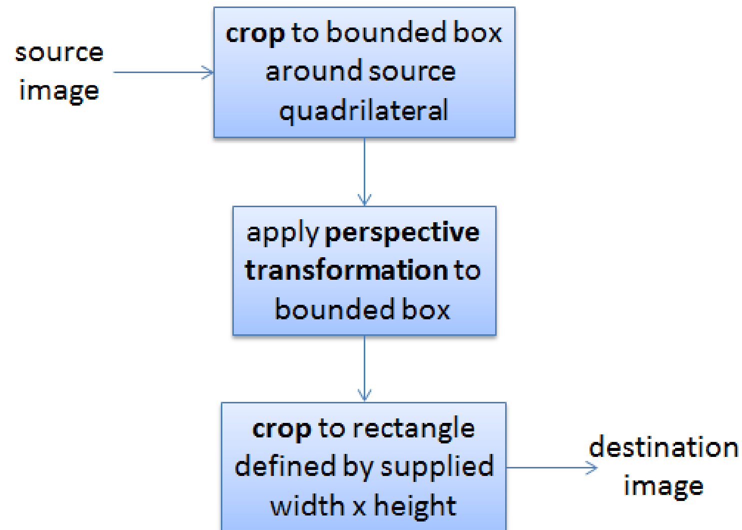


Figure 10. Applying a Perspective Transformation.

The perspective transformation is preceded and followed by cropping operations.

The first crop is essential otherwise there's a good chance that the perspective transformation may run out of memory and crash. Why?

Figure 3 (and Figure 8) shows the source quadrilateral in yellow, inside the larger source image. The perspective transformation implemented by JAI's WarpPerspective class does **not** treat this quad as a clipping region. Warping is applied to the *entire* image, not just to the part inside the quadrilateral. To increase the processing speed, and to reduce the possibility of memory overload, the source image should be cropped to be as small as possible. The result is a bounded box like the one drawn with dashed green lines on the left of Figure 11.

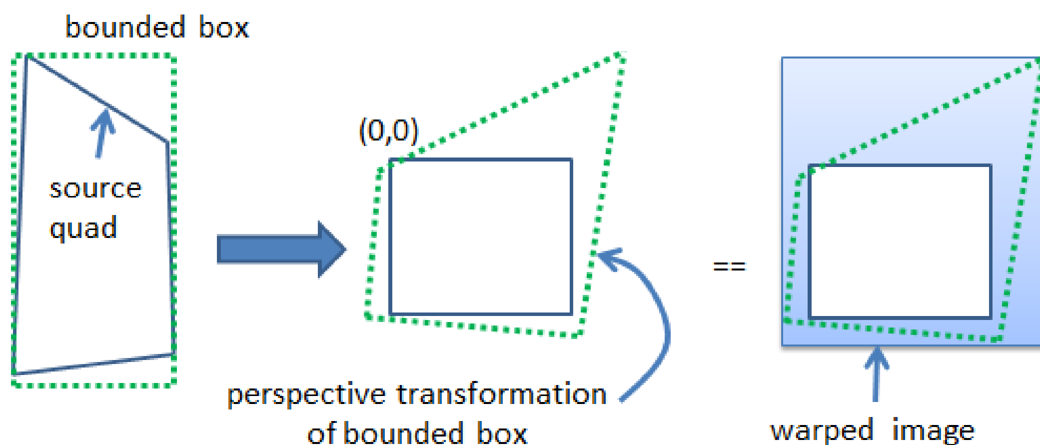


Figure 11. Cropping and Perspective Transformation.

When the perspective transformation is applied to the source quadrilateral, the result is something like the middle image of Figure 11. Not only is the quadrilateral transformed, but so is the bounded box, usually resulting in a large quadrilateral.

 © Andrew Davison 2014

The memory needed for the large quadrilateral is bigger than the middle picture suggests since the image is stored as a rectangle (i.e. as the blue rectangle in the right-hand image of Figure 11). If no bounded box was created as the first step, this warped image could be very large, perhaps exceeding Java's default memory space.

Figure 12 shows an example bounded box and the resulting warped image.

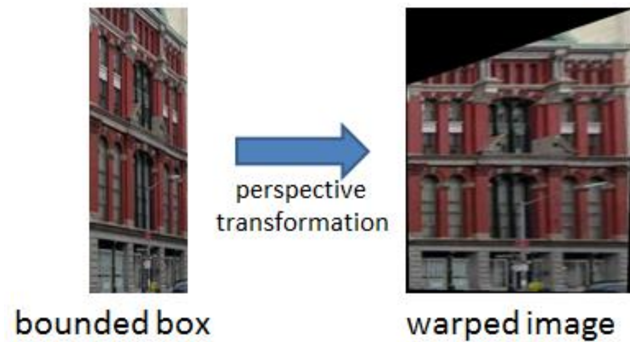
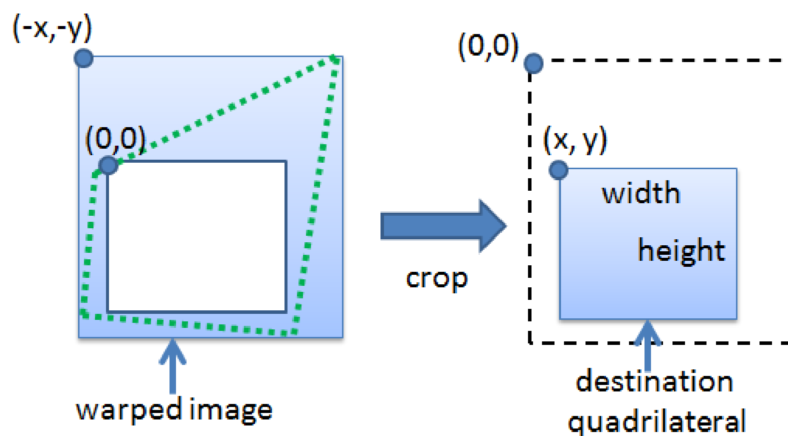


Figure 12. Bounded Box and Warped Image Example.

Note that the parts of the warped image outside the transformed bounded box are represented by black pixels. These pixels correspond to the shaded blue parts of the warped image in Figure 11 that occur outside the dashed green lines of the transformed bounded box.

Figure 11 corresponds to the first two operations in Figure 10. The second crop operation is required to cut away the excess parts of the warped image, leaving only the destination quadrilateral. The tricky part is determining the top-left coordinate of the quadrilateral, as suggested by Figure 13.



99

Figure 13. Cropping the Warped Image.

 © Andrew Davison 2014

The perspective transformation is set-up to place the top-left corner of the destination quadrilateral at (0, 0), which means that the top-left corner of the enclosing warped image is actually a negative coordinate (shown as (-x, -y) in Figure 13). In that case, when the cropping operation is defined for the image, the top-left corner of the quadrilateral will be at (x, y) relative to the image.

Figure 14 shows how this cropping affects the warped image from Figure 12.

Figure 14. Cropping the Warped Image Example.

If the original source image is not cropped then a very large warped image can be generated. Even when this image does not cause a memory crash, it seems that JAI's WarpPerspective class sometimes miscalculates the top-left corner of the warped image, producing a large positive coordinate. This causes the second crop to generate an incorrect destination quadrilateral.

It appears that underflow is occurring when very large negative numbers are generated for the (-x, -y) coordinate, and the large negative numbers become large positive values as a result. This problem is avoided by cropping the source image so that the warped image is as small as possible.

4. Implementing the Perspective Transformation

All the operations depicted in Figure 10 (crop, transformation, crop) are performed inside Warper's actionPerformed() method:

```
// globals
private static final String WARP_FNM = "warpOut.png";
                 // name of warped, cropped image file

private static final int NUM_CORNERS = 4;

private BufferedImage sourceIm;  // input image

// GUI elements
private QuadPanel quadPanel;   // left-hand panel
private ImagePanel imPanel;    // right-hand panel
private JTextField widthTF, heightTF;
```

 © Andrew Davison 2014

```
private JButton warpButton;


public void actionPerformed(ActionEvent e)
{
  // get source and destination quad info from the GUI
  int warpWidth = toInt(widthTF.getText(), WARP_WIDTH);
  int warpHeight = toInt(heightTF.getText(), WARP_HEIGHT);

  if (!quadPanel.hasQuad()) {
    System.out.println("No quadrilateral defined");
    return;
  }
  Point[] coords = quadPanel.getCoords();

  // crop the image so it tightly bounds the source quad
  Rectangle bounds = rectangleBounds(coords);
  BufferedImage boundedIm = cropImage(sourceIm, bounds.x, bounds.y,
                                  bounds.width, bounds.height);

  // save then read in again!
  saveImage("bounded", boundedIm, "bounded.png");
  boundedIm = loadImage("bounded.png");

  // adjust source quad coordinates to apply to bounded image
  for (int i=0; i < NUM_CORNERS; i++) {
    coords[i].x -= bounds.x;
    coords[i].y -= bounds.y;
  }

  // convert the source quad into a JAI perspective transform
  PerspectiveTransform persTF = makeWarpTransform(coords,
                                    warpWidth, warpHeight);

  // apply perspective transformation to image
  BufferedImage warpIm = warpImage(boundedIm, persTF);
  if (warpIm == null)
    return;

  // find top-left coordinate of warped image
  Point topLeft = findWarpTopLeft(boundedIm, persTF);
  if (topLeft == null)
    return;

  // crop warped image to destination quad
  BufferedImage cropIm = cropWarp(warpIm, topLeft,
                                  warpWidth, warpHeight);

  imPanel.setImage(cropIm);   // display in ImagePanel
  saveImage("warped and cropped", cropIm, WARP_FNM);
}  // end of actionPerformed()
```

actionPerformed() begins by obtaining the quadrilaterals information from the GUI – the width and height of the destination quadrilateral, and the source quadrilateral corners (which are stored in the coords[] array).

The rectangleBound() and cropImage() methods calculate a bounded box around the source quad, and use it to clip the source image. cropImage() utilizes the standard Java 2D BufferedImage.getSubimage() method; there's no need to employ JAI yet.

 © Andrew Davison 2014

Unfortunately, JAI does not work well with cropped BufferedImages, and so it's necessary to save the cropped image and then immediately read it back in. The boundedIm variable points to the same image as before but now the subsequent JAI transformation will not crash! This bug was identified at least 6 years ago (see https://community.oracle.com/thread/1272280), but hasn't been fixed yet.

The perspective transformation is performed in two steps, implemented by calls to makeWarpTransform() and warpImage(): makeWarpTransform() creates the operation, and warpImage() applies it.

makeWarpTransform() specifies the transformation in terms of source and destination quadrilaterals, as in Figure 15.

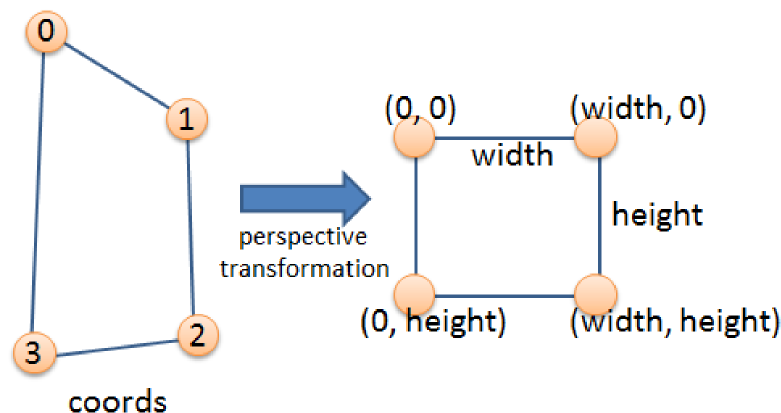


Figure 15. Perspective Transformation in terms of Quadrilaterals.

The makeWarpTransform() code:

```
private PerspectiveTransform makeWarpTransform(Point[] coords,
                                                int w, int h)
{ return PerspectiveTransform.getQuadToQuad(
            coords[0].x, coords[0].y,   coords[1].x, coords[1].y,
            coords[2].x, coords[2].y,   coords[3].x, coords[3].y,
                     0,0,     w,0,         w,h,      0,h  );
}
```

With warpImage(), I finally get to use a JAI ParameterBlock, which brings together three things: a PlanarImage version of the bounded box BufferedImage, the perspective transformation, and a smoothing operation to make the result look better.

The result is another JAI PlanarImage, and this must be converted to a BufferedImage before being returned.

```
private BufferedImage warpImage(BufferedImage boundedIm,
                                PerspectiveTransform persTF)
{ WarpPerspective warpOp = null;
  try {
    warpOp = new WarpPerspective( persTF.createInverse() );
                             // invert the transform
  }
  catch(Exception e)
  {  System.out.println("Unable to create warp operation");  }
```

 © Andrew Davison 2014

```
  if (warpOp == null)
    return null;

  // create and execute a parameter block
  ParameterBlock pb = new ParameterBlock();
  pb.addSource( PlanarImage.wrapRenderedImage(boundedIm) );
                        // bounded box BufferedImage --> PlanarImage
  pb.add( warpOp );
  pb.add( Interpolation.getInstance(Interpolation.INTERP_BILINEAR ));
  PlanarImage warpedImage = JAI.create("warp", pb);

  BufferedImage warpIm = null;
  try {
    warpIm = warpedImage.getAsBufferedImage();
                  // PlanarImage --> BufferedImage
  }
  catch (Exception e)
  { System.out.println("Unable to create warped image: " + e); }
  catch(OutOfMemoryError e)
              // quite likely if source image is large
  { System.out.println("Warped image is too large: " + e); }

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

The perspective transformation has to be wrapped inside a WarpPerspective object, which requires a reversed destination-to-source quadrilateral mapping obtained by calling PerspectiveTransform.createInverse().

The possible memory crashes discussed earlier will occur when the JAI PlanarImage is converted into a BufferedImage, but will be caught by an OutOfMemoryError catch block.

Finding the Top-left of the Warped Image

Figure 16 shows the warped image, and the problem of calculating its top-left corner. This will be negative (-x, -y) since the top-left corner of the destination quadrilateral is set to be at (0, 0).

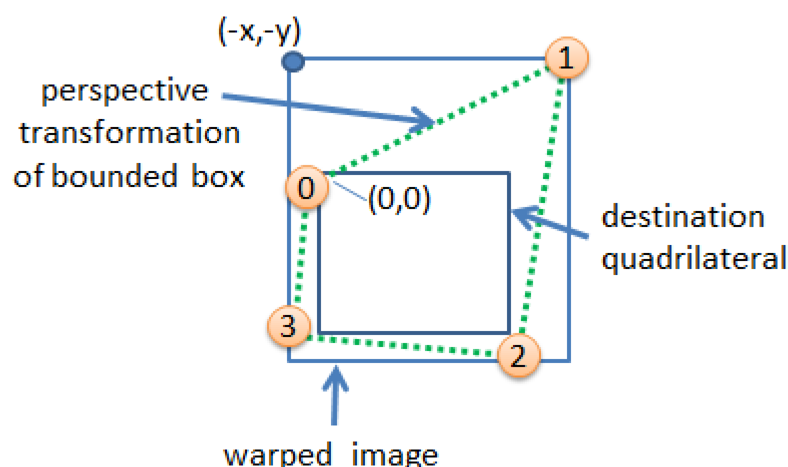


Figure 16. The Warped Image, Destination Quadrilateral, and Transformation.

© Andrew Davison 2014

The trick is to examine the coordinates of the transformed bounded box (shown as orange numbered circles in Figure 16). The minimum x- and y- values of those four coordinates are equivalent to the (-x, -y) coordinate because the warped image is a bounded box around the transformation. This approach utilizes JAI's PerspectiveTransform.transform() method which can apply a transformation to a point. findWarpTopLeft() calls transform() on the four bounded box coordinates to generate the four *transformed* bounded box points shown in Figure 16.

The findWarpTopLeft() code:

```
private Point findWarpTopLeft(BufferedImage boundedIm,
                              PerspectiveTransform persTF)
{
  int imWidth = boundedIm.getWidth();
  int imHeight = boundedIm.getHeight();

  // store the four corner points of the bounded box
  Point[] bbCorners = new Point[NUM_CORNERS];
  bbCorners[0] = new Point(0,0);
  bbCorners[1] = new Point(imWidth,0);
  bbCorners[2] = new Point(imWidth, imHeight);
  bbCorners[3] = new Point(0, imHeight);

  // warp the bounded box corners using the perspective transform
  Point[] corners = new Point[NUM_CORNERS];
  for (int i=0; i < NUM_CORNERS; i++) {
    corners[i] = new Point();
    persTF.transform(bbCorners[i], corners[i]);
  }

  /* find the smallest (x,y) coordinate, which is
     equivalent to the top-left corner of the warped image */
  int xMin = corners[0].x;
  int yMin = corners[0].y;
  for (int i=1; i < NUM_CORNERS; i++) {
    if (corners[i].x < xMin)
      xMin = corners[i].x;
    if (corners[i].y < yMin)
      yMin = corners[i].y;
  }

  if (!((xMin <= 0) && (yMin <= 0))) {     // smallest should be -ve
    System.out.println("Perspective calculation error");
    return null;
  }

  return new Point(xMin, yMin);
}  // end of findWarpTopLeft()
```

The warped bounded box corners (the orange circles in Figure 16) are stored in a corners[] array inside findWarpTopLeft(), and their minimum x- and y- values are returned as a point.

The if-test for the negativity of the point at the end of findWarpTopLeft() detects if the transformation has gone wrong because of a numerical underflow. Underflow will

 © Andrew Davison 2014

cause one, or both, of the values to be greater than 0, which is impossible since the top-left corner must be above and to the left of (0, 0) in the destination quadrilateral.

Back in actionPerformed(), the (-x, -y) coordinate is passed to cropWarp(), which converts it to (x, y) and calls cropImage() to extract the destination quadrilateral. In essence, cropWarp() implements Figure 13 from earlier. No JAI features are used.

 © Andrew Davison 2014