Chapter NUI-13. Depth Processing

One of the claims to fame of the Kinect sensor is its depth processing capabilities, including the generation of depth maps. It's possible to implement similar functionality on a PC with two ordinary webcams, (after they've been calibrated). Figure 1 shows the left and right images from the cameras being rectified, using the calibration information to undistort and align the pictures. Those images are then transformed into a grayscale disparity map, 3D point cloud, and an anaglyph picture.

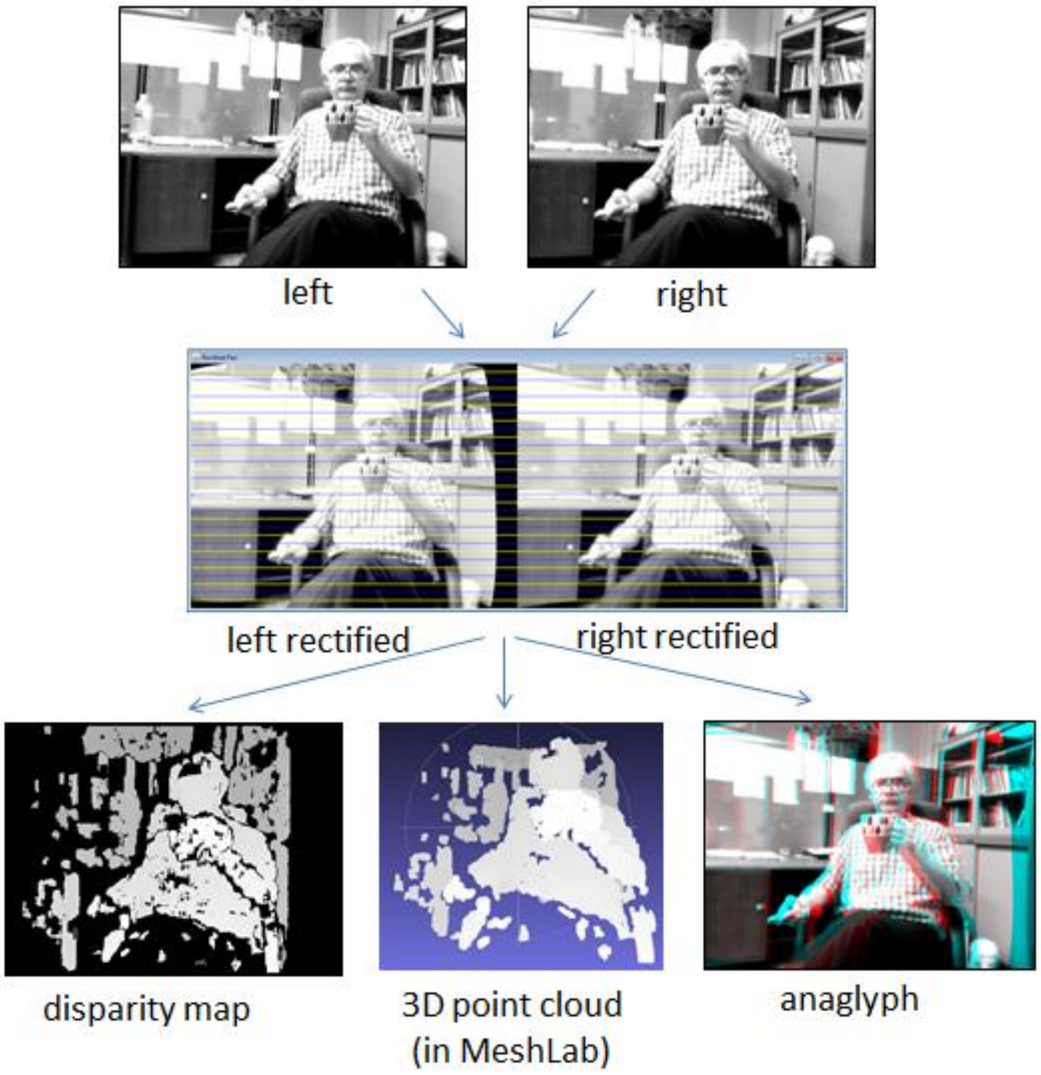


Figure 1. Extracting Depth from Images.

The disparity map indicates that the user's coffee cup is closest to the cameras since it's colored white, the user is a light gray and so a bit further away, and the background is a darker gray.

Figure 2 shows a modified version of the disparity map, adjusted using the application's GUI controls. The right hand camera image is included for comparison.

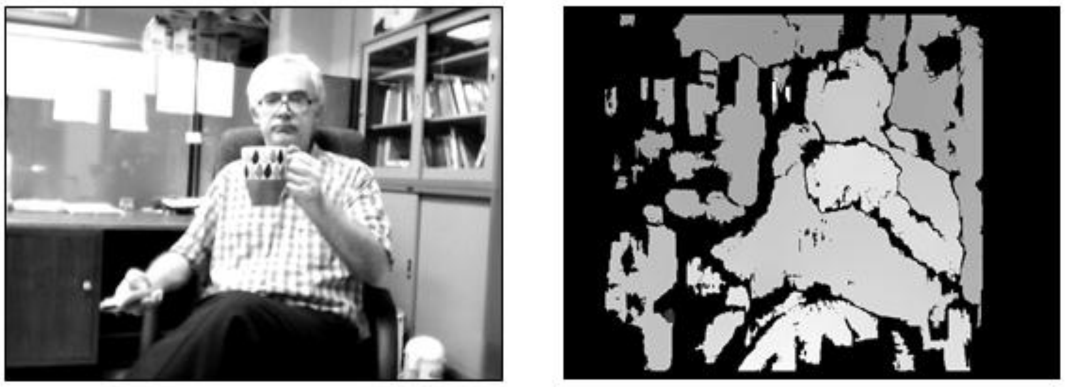
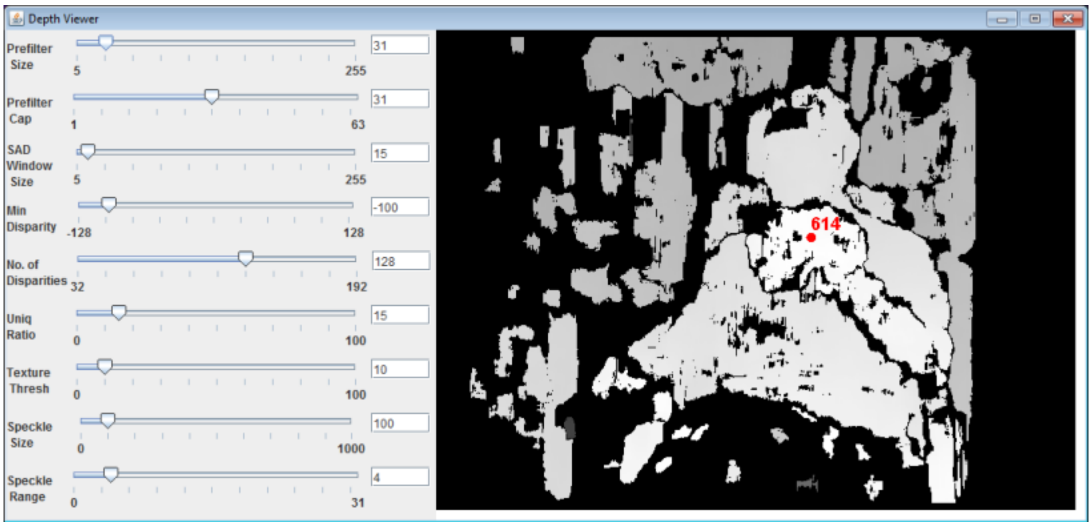


Figure 2. Right-hand Image and Modified Disparity Map.

The user's leg is highlighted in white, and the background is better outlined.

It's possible to click on the disparity map, to retrieve depth information (in millimeters. Figure 3 shows the complete DepthViewer GUI, with a red dot and number marked on the map stating that the coffee cup is 614 mm away from the camera.



Depth Viewer

Prefilter Size	5	255	31
Prefilter Cap	1	63	31
SAD Window Size	5	255	15
Min Disparity	-128	128	-100
No. of Disparities	32	192	128
Uniq Ratio	0	100	15
Texture Thresh	0	100	10
Speckle Size	0	1000	100
Speckle Range	0	31	4

Figure 3. Accessing Depth Information via the GUI.

Unfortunately, this information isn't particularly accurate (the actual distance is nearer 900 mm) due to reasons explained later.

A point cloud is a 3D representation of the depth information, stored in the popular PLY data format (http://en.wikipedia.org/wiki/PLY_(file_format)), which allows it to be loaded (and manipulated) by various 3D tools. Figure 4 shows two screenshots of the point cloud of Figure 1 loaded into MeshLab (http://meshlab.sourceforge.net/).

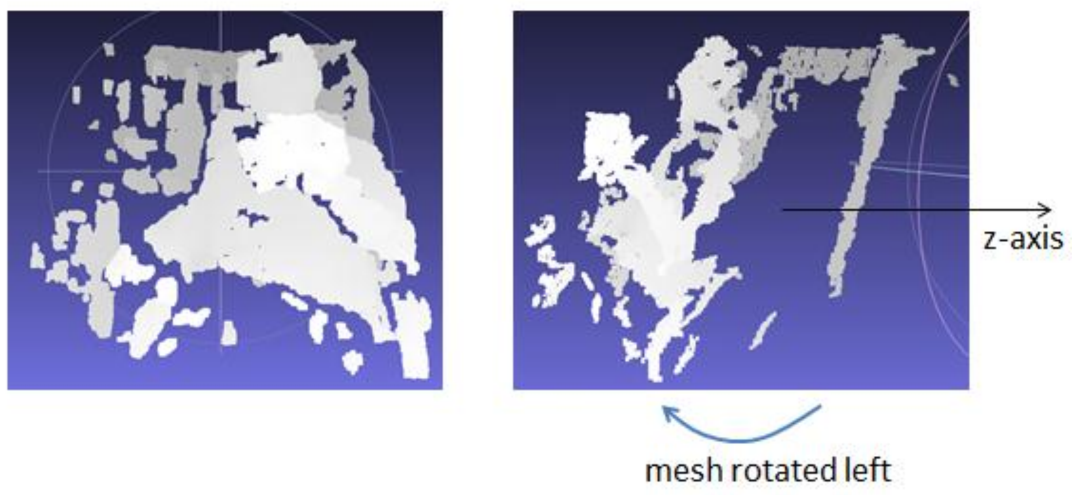


Figure 4. Point Cloud Loaded into MeshLab, and then Rotated.

The image on the right of Figure 4 shows the point cloud rotated to the left so that the z-axis (the depth information) is more visible.

The anaglyph in Figure 1 is created by encoding the left and right rectified images using red and cyan filters and merging them into a single picture. The 3D effect becomes clear when the image is viewed through color-coded anaglyph glasses. An enlarged version of the anaglyph appears in Figure 5, along with an example of suitable glasses.

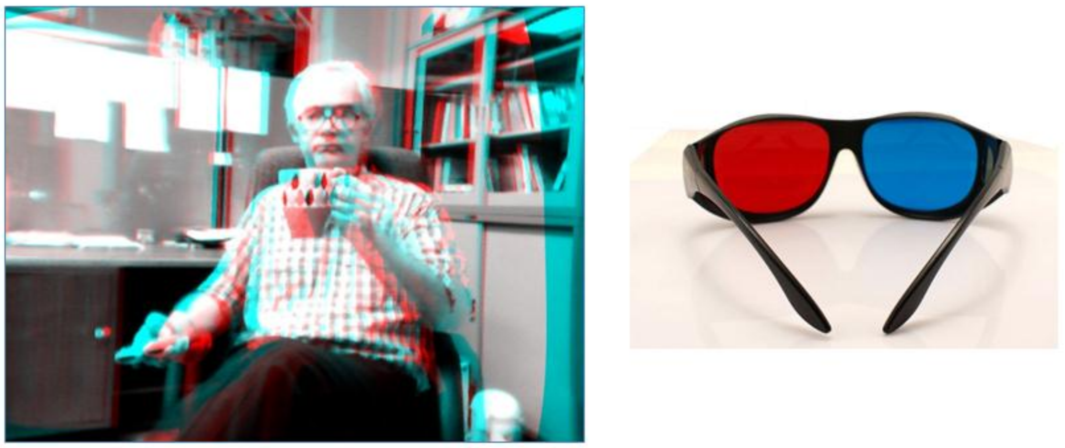


Figure 5. The Anaglyph and Glasses.

The quality of the disparity map, point cloud, and anaglyph depend on the undistortion and rectification mapping carried out on the left and right input images. This mapping is generated during an earlier calibration phase, when a large series of paired images are processed by DepthViewer. These image pairs are collected using a separate application, called SnapPics, that deals with the two webcams independently of the complex tasks involved in depth processing.

The calibration technique supported by OpenCV requires the user to hold a chessboard picture. Figure 6 shows one of the calibration image pairs.

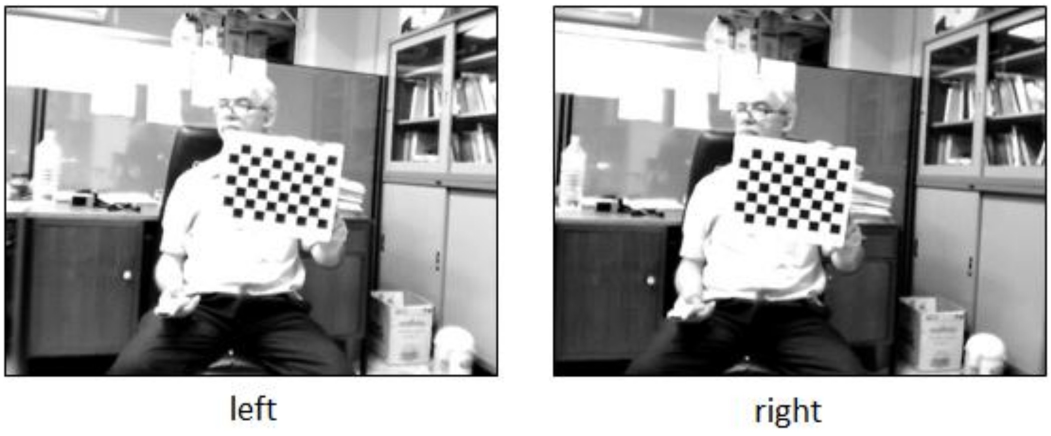


Figure 6. A Calibration Image Pair (User with a Chessboard).

In summary, depth processing consists of three stages:

1. Multiple image pairs are snapped using the SnapPics application. At least 20 image pairs are needed, and usually a lot more, in order for the calibration process in stage 2 to produce decent results.
2. The calibration phase performed by DepthViewer analyses the image pairs (all showing the user holding a chessboard in various poses). The result is a collection of undistortion and rectification matrices that are employed in stage 3.
3. The depth processing phase, illustrated by Figure 1, converts a specific image pair into a disparity map, point cloud PLY file, and an anaglyph. At this stage, it's no longer necessary for the user to be holding a chessboard in the images.

I'll explain these stages in more detail during the course of this chapter. For more information on the underlying maths, I recommend chapters 11 and 12 of *Learning OpenCV* by Gary Bradski and Adrian Kaehler, O'Reilly 2008.

1. Preparing the Webcams

The hardware required for collecting image pairs seems fairly simple at first glance: two USB 2.0 webcams plugged into a laptop. But a quick look online shows that multiple webcams have posed a problem for PCs in the past. One reason is the old USB 1.1 protocol which couldn't deal with the bandwidth requirements of simultaneous input from two cameras. This problem has receded with more modern hardware and OSes, although USB 2.0 is still not fast enough for processing data from more than two cameras unless some kind of data compression is utilized (e.g. as in the PS3 eye cameras).

Bandwidth problems can be avoided by plugging the webcams into separate USB controllers on the PC. They should not be attached to an external hub using a single USB port since this raises the chances of it's bandwidth being exceeded. One way to reduce the data load on the USB controller is to switch each camera's resolution to 320x240 pixels, but this makes it harder for the computer vision code in stages 2 and 3 to produce reliable results. Fortunately, my Windows 7 test machine was fast enough to handle resolutions of 640x480 from both cameras.

Windows XP had poor support for multiple webcams, but this was remedied in Windows 7 which has a default USB 2.0 camera driver that supports multiple cameras. Once the webcams are plugged in, the Device Manager will list the drivers under "Imaging Devices" (see Figure 7).

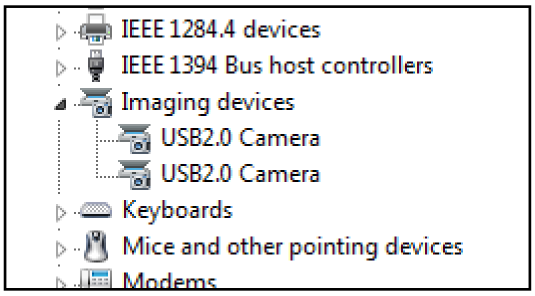


Figure 7 The Windows 7 Webcam Drivers in the Device Manager.

The calibration techniques implemented by OpenCV assume the webcams are aligned parallel to each other (or very nearly parallel), and have similar device characteristics. The easiest way of satisfying this second requirement is to buy two cameras of the same brand. Ideally, the cameras should have the same focal length, so some time should be spent on focussing them on the user's location. If the cameras have auto-focus, switch it off .

Proper alignment is very important for obtaining good results, which I achieved by attaching my cameras to a fixed tripod bar, as in Figure 8

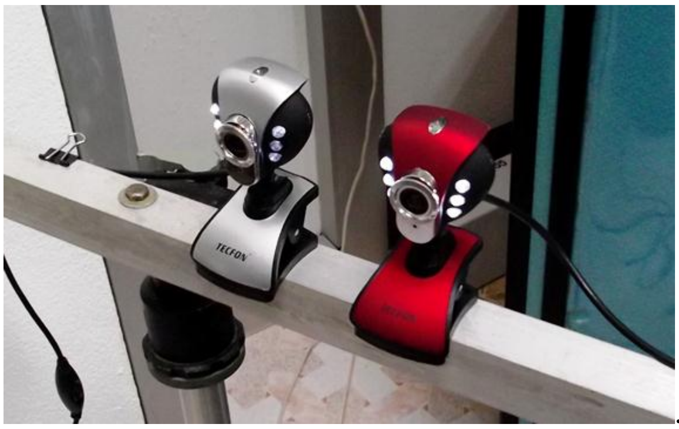
.

Figure 8. Webcams on a Tripod Bar.

The cameras configuration is shown graphically in Figure 9.

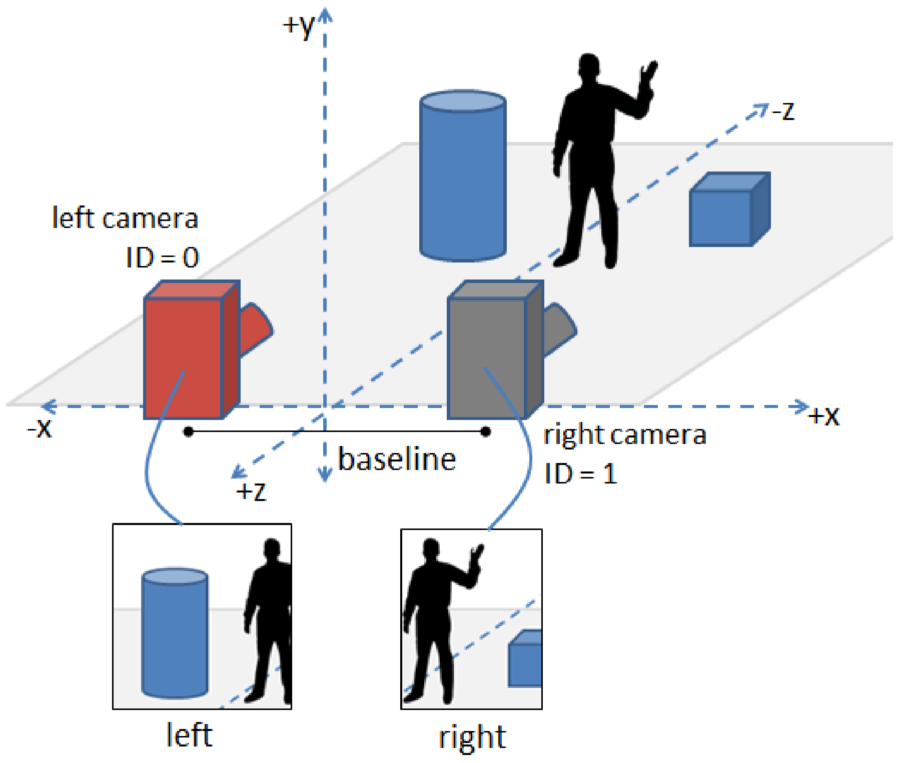


Figure 9. Webcams in a Scene.

The webcams are labeled "left" and "right" relative to the camera's point of view. The camera IDs in Figure 9 are employed by JavaCV's FrameGrabber class – the left camera uses ID 0, and the right camera is number 1.

Also important for later is to note that the depths will be negative, since the scene is spaced out along the negative z-axis.

2. Collecting Webcam Pictures

SnapPics, the webcam image collector, is a variation of my usual 'picture snapping' code which utilizes two JavaCV FrameGrabber instances. A threaded loop inside the JPanel grabs snaps from both cameras on each iteration, and displays them side-by-side in the window. Figure 10 shows SnapPics executing, with the left webcam image displayed on the left side of the panel, and the right webcam on the right.

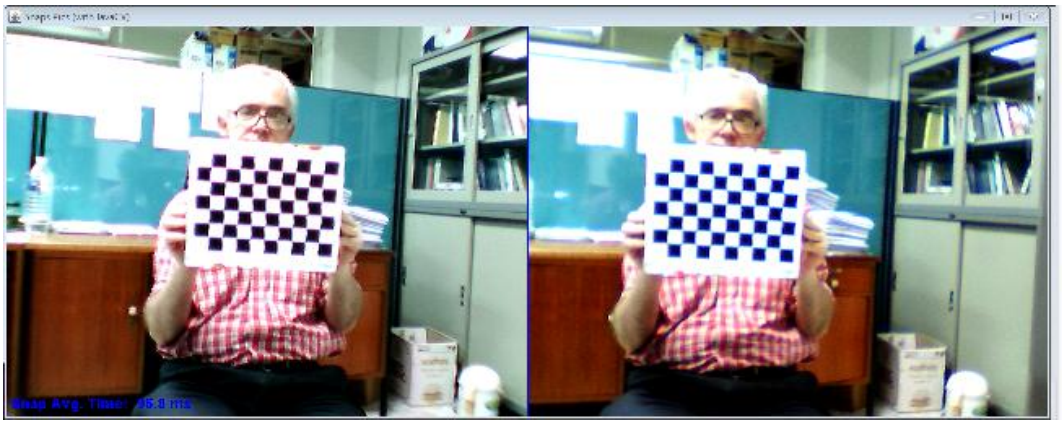


Figure 10. The SnapPics Application.

SnapPics snaps images every 150ms or so, but only saves pictures to files when the user presses the <enter>, space, or numpad-5 key. A surprisingly tricky problem is having the user press keys without moving, which would blur the images. My solution was to use a wireless keyboard device.

The calibration images must include a chessboard, and preferably one which is **not** square. The calibration code has a slightly easier job judging a board's orientation if its edges are not equal. The user needs to supply *at least* 20 image pairs, with the board held in a variety of positions and orientations in each pair. The board shouldn't be obscured or be bent. For that reason, my 'board' (actually an A4 sheet of paper) is stuck to a cardboard box lid (see Figure 11). This makes it both rigid and easier to hold.

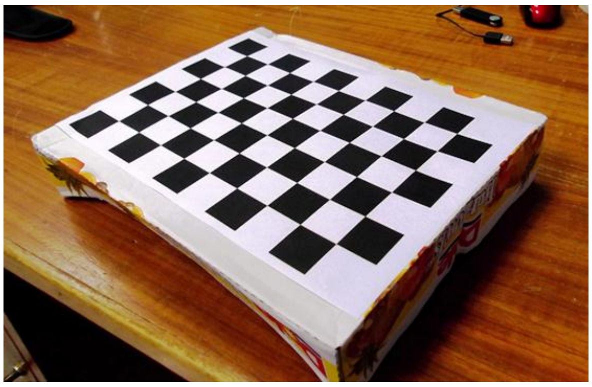


Figure 11. The Chessboard Stuck to a Cardboard Lid.

The OpenCV method for analyzing a chessboard image looks for the **interior** corners in the board, which means that my board has 9x6 corners. The best dimensions make the chessboard grid asymmetrical (as here), and so easier for the software to correctly orientate.

SnapPics uses JavaCV's FrameGrabber class, whose constructor uses a camera ID to identify the webcam of interest. The easiest way of obtaining the IDs is to run ListDevices.java:

```
public class ListDevices
{
  public static void main( String args[] )
  {
    int numDevs = videoInput.listDevices();
    System.out.println("No of video input devices: " + numDevs);
    for (int i = 0; i < numDevs; i++)
      System.out.println(" " + i + ": " +
                              videoInput.getDeviceName(i));
  }  // end of main()

} // end of ListDevices class
```

When both webcams are plugged into my PC, ListDevices prints the following:

```
No of video input devices: 6
 0: USB2.0 Camera
 1: USB2.0 Camera
 2: Kinect Virtual Camera : Depth
 3: Kinect Virtual Camera : Image
 4: Kinect Virtual Camera : SmartCam
 5: Video Blaster WebCam 3/WebCam Plus (VFW)
```

Therefore, I should use camera IDs 0 and 1 inside SnapPics. The simplest way of identifying which camera uses which ID is to unplug one of them, and run ListDevices again:

```
No of video input devices: 5
 0: USB2.0 Camera
 1: Kinect Virtual Camera : Depth
 2: Kinect Virtual Camera : Image
 3: Kinect Virtual Camera : SmartCam
 4: Video Blaster WebCam 3/WebCam Plus (VFW)
```

I obtained this result after unplugging the gray camera (the right hand device in Figure 9), so the left camera must be ID 0, and the right camera is ID 1 (when plugged back in).

The UML class diagrams for the SnapPics application has the familiar structure of a JFrame containing a threaded JPanel (see Figure 12).

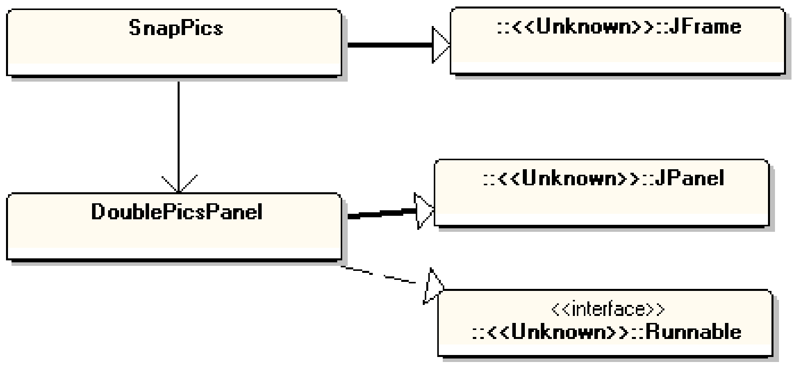


Figure 12. Class Diagrams for the SnapPics Application.

A DoublePicsPanel instance is passed the IDs of the two cameras:

```
// in the SnapPics class
DoublePicsPanel pp = new DoublePicsPanel(0, 1);
```

These IDs are used to create JavaCV FrameGrabber instances at the beginning of the DoublePicsPanel run() method:

```
// in the DoublePicsPanel class
// globals
private static final int DELAY = 150;  // ms

// directory and filenames used to save images
private static final String SAVE_DIR = "pics/";
private static final String LEFT_FNM = "left";
private static final String RIGHT_FNM = "right";

private volatile boolean isRunning;
private long totalTime = 0;
private int imageCount = 0;

private IplImage leftImage = null;
private IplImage rightImage = null;
private int leftID, rightID;    // IDs of the FrameGrabber objects
private volatile boolean takeSnaps = false;


public void run()
{
  FrameGrabber leftGrabber = initGrabber(leftID);
  FrameGrabber rightGrabber = initGrabber(rightID);
  if ((leftGrabber == null) || (rightGrabber == null))
    return;

  long duration;
  int snapCount = 0;
  isRunning = true;

  while (isRunning) {
    long startTime = System.currentTimeMillis();

    leftImage = picGrab(leftGrabber, leftID);
    rightImage = picGrab(rightGrabber, rightID);

    if (takeSnaps) {   // save the current images
      saveImage(leftImage, LEFT_FNM, snapCount);
      saveImage(rightImage, RIGHT_FNM, snapCount);
      snapCount++;
      takeSnaps = false;
    }

    imageCount++;
    repaint();

    duration = System.currentTimeMillis() - startTime;
    totalTime += duration;
    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);
        // wait until DELAY time has passed
      }
      catch (Exception ex) {}
    }
  }
  closeGrabber(leftGrabber, leftID);
  closeGrabber(rightGrabber, rightID);
}  // end of run()
```

The FrameGrabbers take snapshots roughly every DELAY(150) milliseconds until the isRunning boolean is set to false. If the takeSnaps boolean is true (which occurs via a call to DoublePicsPanel.takeSnaps() when the user presses the relevant key) then the current snapshots are saved in a local subdirectory. Each image is labeled "left" and "right", together with the snapCount number.

initGrabber() initializes the FrameGrabber instance to use DirectShow and return a 640x480 size image:

```
private FrameGrabber initGrabber(int ID)
{
  FrameGrabber grabber = null;
  System.out.println("Initializing grabber for " +
               videoInput.getDeviceName(ID) + " ...");
  try {
    grabber = FrameGrabber.createDefault(ID);
    grabber.setFormat("dshow");       // using DirectShow
    grabber.setImageWidth(WIDTH);     // change from 320x240
    grabber.setImageHeight(HEIGHT);
    grabber.start();
  }
  catch(Exception e)
  {  System.out.println("Could not start grabber");
     System.out.println(e);
     System.exit(1);
  }
  return grabber;
}  // end of initGrabber()
```

picGrab() wraps up a call to FrameGrabber.grab():

```
private IplImage picGrab(FrameGrabber grabber, int ID)
{
  IplImage im = null;
  try {
    im = grabber.grab();  // take a snap
  }
  catch(Exception e)
  {  System.out.println("Problem grabbing camera " + ID);  }
  return im;
}  // end of picGrab()
```

closeGrabber(), called twice at the end of run(), shuts down a FrameGrabber:

```
private void closeGrabber(FrameGrabber grabber, int ID)
{
  try {
    grabber.stop();
    grabber.release();
  }
  catch(Exception e)
  {  System.out.println("Problem stopping camera " + ID);  }
}  // end of closeGrabber()
```

By the time SnapPics finishes, a sequence of numbered JPG files labeled "left" and "right" have been collected. These can be used as the image pairs input to DepthViewer.

3. An Overview of the DepthViewer

Figure 13 shows the class diagrams for the DepthViewer application.

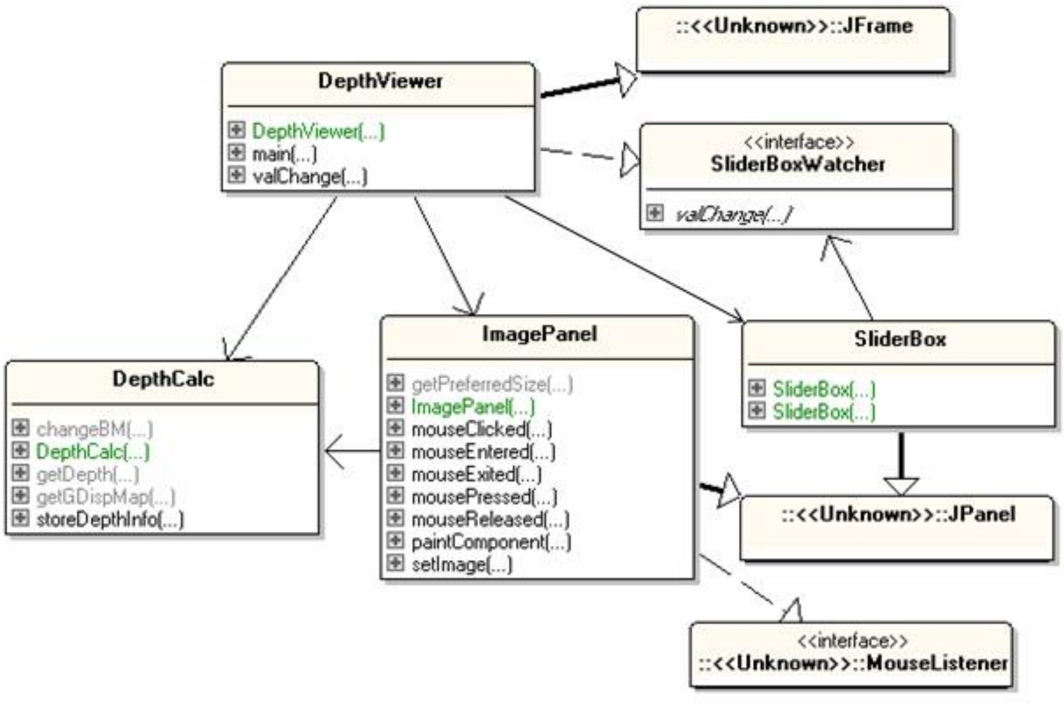


Figure 13. Class Diagrams for the DepthViewer Application.

The collection of sliders on the left side of the GUI in Figure 3 are SliderBox objects, which communicate slider adjustments to DepthViewer via calls to the SliderBoxWatcher.valChange() method implemented in DepthViewer. The disparity map on the right-hand side of the GUI is managed by an ImagePanel instance, which calls getGDispMap() in DepthCalc to get the current image, and getDepth() to get the depth at a particular (x, y) coordinate on the image.

I'll spend most of this chapter talking about DepthCalc, which is where the interesting computer vision code is located.

4. Calibrating the Depth Viewer

The DepthViewer application can be started in one of two modes, either for calibration or for depth processing. An example command line call that starts the calibration process is:

```
>  run DepthViewer -n 40
```

This causes DepthViewer to load 40 image pairs from a calibration subdirectory, The images are assumed to be labeled "left" and "right" and be numbered from 0 up to 39. These pairs are used to create a variety of calibration matrices, which are saved back to the local directory for use during depth processing. The main calibration steps are illustrated in Figure 14.

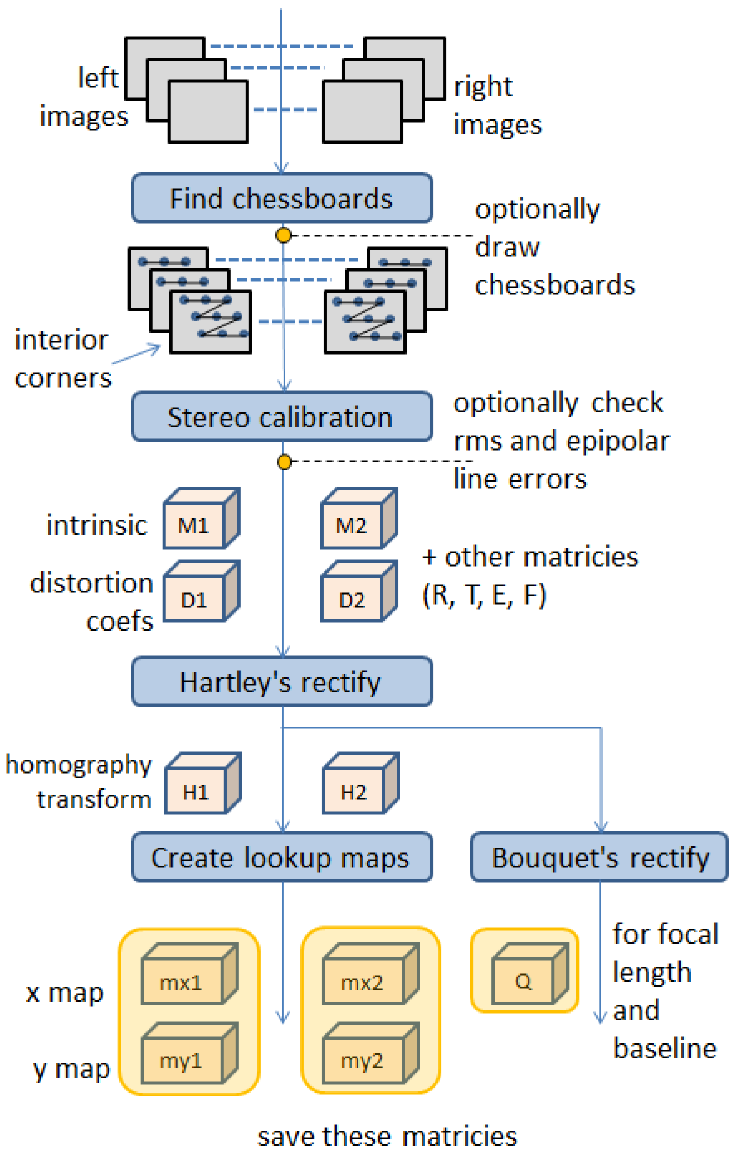


Figure 14. The Calibration Stages in DepthViewer.

The steps shown in Figure 14 are carried out by DepthCalc, starting in its calibrateCams() method.

```
// globals
// number of INTERIOR corners in board along its rows and columns
private static final int CORNERS_ROWS = 6;
private static final int CORNERS_COLS = 9;
private static final int NUM_CORNS = CORNERS_ROWS * CORNERS_COLS;

private int totPoints;


private void calibrateCams(int maxPairs)
{
  totPoints = maxPairs * NUM_CORNS;

  CvMat objPts = CvMat.create(1, totPoints, CV_32F, 3);
  FloatBuffer objPtsBuf = objPts.getFloatBuffer();
      // corner coords for an image, repeated maxPairs times

  CvMat nPts = CvMat.create(1, maxPairs, CV_32S, 1);
  IntBuffer nPtsBuf = nPts.getIntBuffer();
      // number of corners in an image, repeated maxPairs times

  CvMat imPts1 = CvMat.create(1, totPoints, CV_32F, 2);
  FloatBuffer imPts1Buf  = imPts1.getFloatBuffer();

  CvMat imPts2 = CvMat.create(1, totPoints, CV_32F, 2);
  FloatBuffer imPts2Buf  = imPts2.getFloatBuffer();
      // holds the pixel coordinates of corners in each image
      // in the same order as the corners in objPts

  loadPairs(objPtsBuf, nPtsBuf, imPts1Buf, imPts2Buf, maxPairs);
  calibrateWithPairs(objPts, nPts, imPts1, imPts2);
}  // end of calibrateCams()
```

calibrateCams()'s main job is the creation of four matrices and corresponding Buffer objects which allows Java to fill the matrices efficiently. The four matrices are:

- **objPts**: this matrix specifies the order of the interior chessboard corners for an image, repeated once for each image pair. A possible ordering for the points in one of my images is shown in Figure 15. The corners are ordered row-by-row from 0 up to 53 since there are 9x6 interior corners.

  Since my calibration uses 40 image pairs, then objPts will repeat the corner points ordering 40 times. This means that the matrix must be big enough to hold a total of 40x9x6 (2160) elements. This value is store in the global variable totPoints in the calibrateCams() method.

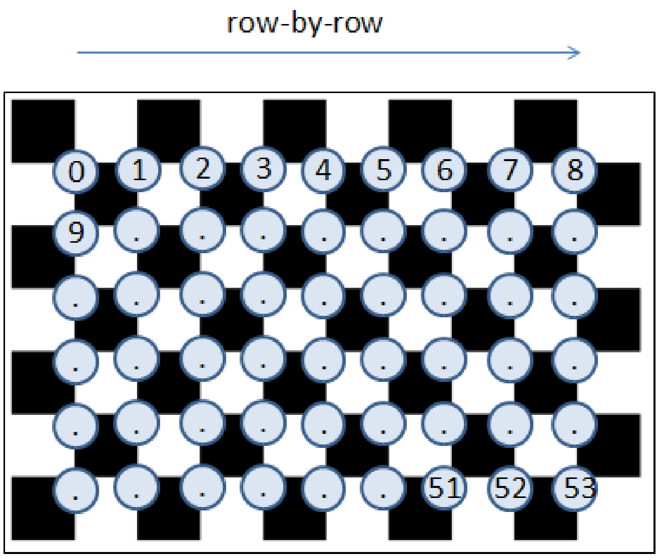


Figure 15. A Board with Numbered Interior Points.

- **nPts**: this matrix specifies the number of interior corners in a chessboard, and is repeated for each image pair. Thus, in my code, nPts is a 1-by-40 element matrix, with each element containing the value 54 (the number of corners).
- **imPts1** and **imPts2**: these hold the corner points that are actually detected by the calibration process in the image pairs. imPts1 holds the corner points for all the left images, and imPts2 all the right image corners.

  In my code, 9x6 corners should be detected in a left image and in a right image, and this success should be repeated 40 times for all the pairs. Therefore imPts1 and imPts2 should be big enough to each store a total of 40x9x6 corners.

  To keep things simple (!), if my calibration code doesn't find this total number of corners, then the program exits after alerting the user about which image pairs have too few points.

DepthCalc.loadPairs() loads all the image pairs inside a loop, calling findCorners() to find each pair's interior corners:

```
// globals
private static final int LEFT = 0;
private static final int RIGHT = 1;


private void loadPairs(FloatBuffer objPtsBuf, IntBuffer nPtsBuf,
                       FloatBuffer imPts1Buf,
                       FloatBuffer imPts2Buf, int maxPairs)
{
  IplImage[] images = new IplImage[2];   // left and right images
  CvPoint2D32f[] cornsPair =  new CvPoint2D32f[2];
                // for corner pts found in left and right images

  // read in image pairs, finding chessboards corners in each
  int numPairs =  0;
  for(int i=0; i < maxPairs; i++) {
    System.out.println("Loading left & right images with ID " + i);
    images[LEFT] = cvLoadImage( makeName("left", i), 0);
    images[RIGHT] = cvLoadImage(makeName("right", i), 0);

    if ((images[LEFT] == null) || (images[RIGHT] == null))
      System.out.println("  One of the images is null;
                                             not adding pair");
    else if (!isRightSize(images[LEFT]) ||
             !isRightSize(images[LEFT]))
        System.out.println("  One of the images is the wrong size;
                                           not adding pair");
    else {
      cornsPair[LEFT] = findCorners("left" + i, images[LEFT]);
      cornsPair[RIGHT] = findCorners("right" + i, images[RIGHT]);

      if ((cornsPair[LEFT] != null) && (cornsPair[RIGHT] != null)) {
        addCornersPair(cornsPair, objPtsBuf, nPtsBuf,
                                  imPts1Buf, imPts2Buf);
        numPairs++;
      }
    }
  }

  System.out.println("No. of valid image pairs: " + numPairs);
  if (numPairs < maxPairs) {
    // give up if there were any errors during the loading
    System.out.println("Please fix " + (maxPairs - numPairs) + "
                                                 invalid pairs");
    System.exit(1);
  }
}  // end of loadPairs()
```

loadPairs() finishes by checking if all the pairs had enough corners. If not then the calibration is terminated. When that occurs, the user should replace the offending image pairs with new pairs where the chessboard is clearer.

4.1. Finding the Chessboards

findCorners() utilizes the OpenCV cvFindChessboardCorners() function for detecting the chessboard corner points in an image. The best source for information of this function (and the other calibration functions we'll encounter) is the Willow Garage documentation on camera calibration and 3D reconstruction at http://opencv.willowgarage.com/documentation/camera_calibration_and_3d_reconstruction.html. A more verbose explanation of many of these functions can be found in chapters 11 and 12 of *Learning OpenCV* by Bradski and Kaehler.

findCorners() uses cvFindChessboardCorners() to scan the image for the 54 corners, and cvFindCornerSubPix() to improve the pixel resolution of the detected coordinates.

```
// globals
private static final int CORNERS_ROWS = 6;
private static final int CORNERS_COLS = 9;
private static final int NUM_CORNS = CORNERS_ROWS * CORNERS_COLS;
private static final CvSize BOARD_SZ =
                           cvSize(CORNERS_ROWS, CORNERS_COLS);
```

```
private CvPoint2D32f findCorners(String fnm, IplImage im)
{
  int[] cornerCount = new int[1];
  cornerCount[0] = 0;
  CvPoint2D32f corners = new CvPoint2D32f(NUM_CORNS);

  // find the chessboards and its corners points
  int result = cvFindChessboardCorners(im, BOARD_SZ,
                        corners, cornerCount,
                        CV_CALIB_CB_ADAPTIVE_THRESH |
                        CV_CALIB_CB_NORMALIZE_IMAGE);

  if (result != 1) {
    System.out.println("Could not find chessboard image in " + fnm);
    return null;
  }

  if (cornerCount[0] != NUM_CORNS) {
    System.out.println("The chessboard image in " +
                   fnm + " has the wrong number of corners");
    return null;
  }

  // improve corner locations by using subpixel interpolation
  cvFindCornerSubPix(im, corners, cornerCount[0],
                    cvSize(11, 11), cvSize(-1,-1),
       cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 30, 0.01));

  if (drawChessboards) {
    // draw chessboard with found corners
    IplImage colImg = cvCreateImage(cvGetSize(im), 8, 3);
    cvCvtColor(im, colImg, CV_GRAY2BGR);
    cvDrawChessboardCorners(colImg, BOARD_SZ, corners,
                                  cornerCount[0], result);
    if (displayFrame == null)
      displayFrame = new CanvasFrame("Chessboard " + fnm);
    else
      displayFrame.setTitle("Chessboard " + fnm);
    displayFrame.showImage(colImg);
    enterPause();
  }

  return corners;
}  // end of findCorners()
```

When the DepthViewer application is first called, a "draw" argument turns on the drawChessboards flag. Inside findCorners(), this allows cvDrawChessboardCorners() to draw each analyzed image in a temporary window, which remains on-screen until the user types <enter>. This slows down the calibration process considerably, but is useful for visually checking if all the corners were detected in the correct row-by-row order. A typical chessboard drawing is shown in Figure 16.

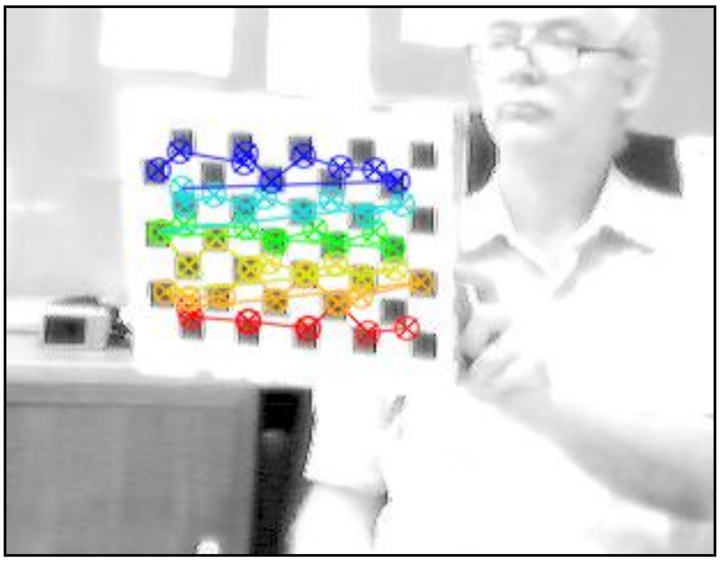


Figure 16. OpenCV Drawing of a Chessboard Corners.

I've cropped and enlarged the image in Figure 16 to show that although 54 corners were found, some of them are in the wrong position. For example, the third point of the top row and the fourth point of the bottom row. This will reduce the final quality of the calibration matrices, so the user should consider replacing image pairs like this one.

Although it's hard to see if you not viewing Figure 16 in color, each row of corners is painted in a different color. If some corners aren't found, then the remaining points are drawn in red.

Each successful call to findCorners() results in loadPairs() adding the detected corners information to the four matrices (objPts, nPts, imPts1, and imPts2). This is done indirectly by having addCornerPairs() write to their Buffer objects.

4.2. Stereo Calibration

The stereo calibration stage shown in Figure 13 begins in DepthCalc.calibrateWithPairs(). The crucial OpenCV method is cvStereoCalibrate() which returns a bewildering amount of information about the two cameras. The method signature is:

```
double cvStereoCalibrate(
        CvMat objPts, CvMat imPts1, CvMat imPts2, CvMat nPts,
        CvMat M1, CvMat D1, CvMat M2, CvMat D2, CvSize imageSize,
        CvMat R, CvMat T, CvMat E, CvMat F,
        CvTermCriteria term_crit, int flags)¶
```

The objPts, imPts1, imPts2, and nPts matrices are inputs, which were filled with chessboard corners information during the previous stage. The output matrices are M1, D1, M2, D2, R, T, E, and F.

Initially cvStereoCalibrate() examines the sequence of left and right images separately to calibrate each camera in terms of an intrinsic matrix (usually called M1 for the left camera, M2 for the right), and a distortion matrix (D1 and D2).

An intrinsic matrix contains information about the focal length and the optical center of a camera relative to the image. The matrix has the form:

$$\mathrm{M} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

f_x and f_y are the focal lengths along the x- and y- axes, and c_x and c_y are the coordinates of the optical center inside the image.

Since the two webcams are identical, and were previously focused on a common spot, then the focal lengths in M1 and M2 should be nearly identical. Also, since the cameras are pointing straight down the z-axis, c_x and c_y for both cameras should be near the center of the image. We can check out these values by printing the M1 and M2 matrices returned by cvStereoCalibrate().

A distortion matrix contains coefficients related to the radial and tangential distortion of the lens. Radial effects make the image bulge, producing a fish-eye shaped picture. Tangential distortion is caused by a misaligned lens which warps the image. Figure 17 shows how radial and tangential distortion can affect an image.

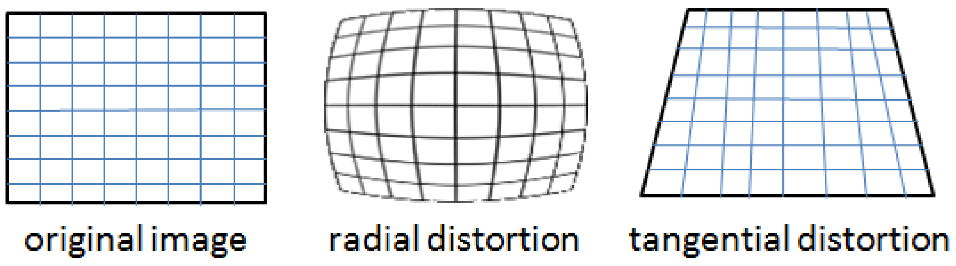


Figure 17. Radial and Tangential Distortion.

Once cvStereoCalibrate() has determined each camera's characteristics, it turns to computing the relationship between the cameras in terms of a rotation matrix R and a translation matrix T, These specify the right camera's projection plane relative to the left camera's plane (see Figure 18)

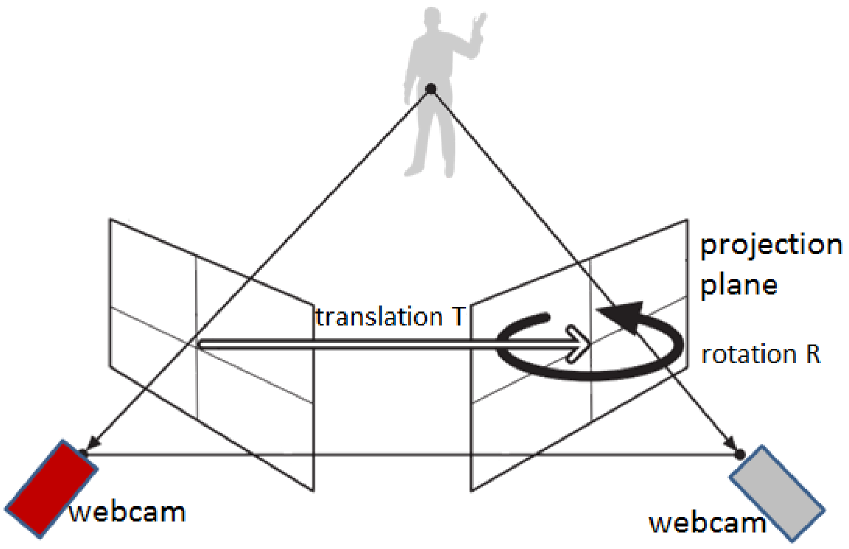


Figure 18. Rotating and Translation the Right Camera Relative to the Left.

Figure 18 exaggerates the angle between the two planes. In practice, the cameras will be nearly parallel to each other, and so the rotation matrix R should be very close to identity (i.e. no rotation is needed). The translation vector (a 3x1 matrix) should also be fairly simple since the two cameras are positioned side-by-side. The only large translation should be in the x-direction, which is called the baseline distance between the webcams. We can check out these values by printing the R and T matrices returned by cvStereoCalibrate().

The other two output matrices are the essential matrix E and the fundamental matrix F. E combines the translation and rotation between the two camera which we can access more easily via the separate T and R matrices. The F matrix combines the information from E with the intrinsic matrices M1 and M2, which means that the rotation and translation are expressed in terms of image coordinates. F is used for the calculation of epipolar lines and the reprojection matrix Q.

DepthCalc.calibrateWithPairs() calls cvStereoCalibrate():

```
private void calibrateWithPairs(CvMat objPts, CvMat nPts,
                                 CvMat imPts1, CvMat imPts2)
{
  System.out.println("\nStarting calibration of two cameras ...");
  long startTime = System.currentTimeMillis();

  // initialize the camera intrinsics matrices
  CvMat M1 = cvCreateMat(3, 3, CV_64F);    // for left camera image
  cvSetIdentity(M1);
  CvMat M2 = cvCreateMat(3, 3, CV_64F);    // right camera
  cvSetIdentity(M2);

  // initialize the distortion coefficients matrices
  CvMat D1 = cvCreateMat(1, 5, CV_64F);
  cvZero(D1);
  CvMat D2 = cvCreateMat(1, 5, CV_64F);
  cvZero(D2);
  CvMat R = cvCreateMat(3, 3, CV_64F);   // the rotation matrix
  CvMat T = cvCreateMat(3, 1, CV_64F);   // the translation vector

  CvMat E = cvCreateMat(3, 3, CV_64F);   // the 'essential' matrix
  CvMat F = cvCreateMat(3, 3, CV_64F);   // the fundamental matrix
  CvSize imSize = cvSize(IM_HEIGHT, IM_WIDTH);

  double rms = cvStereoCalibrate(objPts, imPts1, imPts2, nPts,
                         M1, D1, M2, D2,
                         imSize, R, T, E, F,
       cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5),
                         CV_CALIB_FIX_ASPECT_RATIO);

  System.out.println("Calibration took " +
              (System.currentTimeMillis() - startTime) + " ms");
  System.out.printf( "RMS reprojection error: %.3f\n", rms);

  printMatrix("M1", M1);
  printMatrix("M2", M2);
  printMatrix("D1", D1);
  printMatrix("D2", D2);
  printMatrix("Rotation Matrix R:", R);
  printMatrix("Translation Matrix T:", T);

  System.out.println("Undistorting image points...");
  cvUndistortPoints( imPts1, imPts1, M1, D1, null, M1);
  cvUndistortPoints( imPts2, imPts2, M2, D2, null, M2);
                    // intrinsic matrices also adjusted

  // check calibration quality
  showEpipolarError(imPts1, imPts2, F);

  // rectification using Hartley's method
  System.out.println("Calculating homography matrices...");
  CvMat H1 = cvCreateMat(3, 3, CV_64F);
  CvMat H2 = cvCreateMat(3, 3, CV_64F);
  cvStereoRectifyUncalibrated( imPts1, imPts2, F, imSize, H1, H2, 3);
  calculateLookupMaps(M1, M2, D1, D2, H1, H2);

  calculateQ(M1, M2, D1, D2, imSize, R, T);
}  // end of calibrateWithPairs()
```

The new matrices are created in the first half of calibrateWithPairs().

The cvStereoCalibrate() call includes a cvTermCriteria() value which dictates how accurately the computed matrix parameters should be before the function returns. The CV_CALIB_FIX_ASPECT_RATIO flag lets the calibration process assume that the f_x and f_y focal lengths have a fixed ratio.

Another way of calling cvStereoCalibrate() is to pass it already-computed M1, M2, D1, and D2 matrices, and set flags to indicate that the function should use those for calculating the R, T, E and F matrices. The M and D matrices for a camera can be obtained through a separate calibration process aimed at a single camera (see chapter 11 of *Learning OpenCV* for how this is performed). This approach is useful because it lets cvStereoCalibrate() concentrate on the stereo calibration between cameras.

Where possible, additional flags should be added to the CV_CALIB_FIX_ASPECT_RATIO setting in cvStereoCalibrate() in order to simplify the calibration task. Including CV_CALIB_SAME_FOCAL_LENGTH and

CV_CALIB_ZERO_TANGENT_DIST is quite common because they specify that the f_x and f_y lengths are the same for both cameras and the cameras don't suffer from tangential distortion.

When cvStereoCalibrate() returns, calibrateWithPairs() prints its result (RMS) and some of the generated matrices, so we can check on the quality of the calibration. The following was printed after calibrating using 40 image pairs:

```
RMS reprojection error: 0.468

M1
|  690.698   0.000   332.098 |
|  0.000   690.698   223.154 |
|  0.000     0.000     1.000 |

M2
|  698.122   0.000   304.415 |
|  0.000   698.122   238.133 |
|  0.000     0.000     1.000 |

D1
|  0.212  -0.547   0.007  -0.005  -0.815 |

D2
|  0.321  -1.756   0.003  -0.001   2.628 |

Rotation Matrix R:
|  0.997   0.003  -0.075 |
| -0.004   1.000  -0.011 |
|  0.075   0.012   0.997 |

Translation Matrix T:
| -3.017 |
| -0.024 |
| -0.239 |
```

The RMS (root mean square) reprojection error returned by cvStereoCalibrate() should fall between 0.1 and 1, with a value closer to 0.1 being better; a result of 0.468 is acceptable.

The M1 and M2 matrices show that the focal lengths for the two cameras are almost identical: M1's $f_x = f_y = 690.698$, and M2's $f_x = f_y = 698.122$. This suggests that the calibration might be improved by setting the CV_CALIB_SAME_FOCAL_LENGTH flag.

If the webcams have reasonably undistorted lens, then the optical centers in M1 and M2 should be close to the middle of the image, i.e. at (320, 240) in an 640x480 picture. The reported values are near: M1's (c_x, c_y) = (332, 223) and M2's (c_x, c_y) = (304, 238).

The rotation R should be an identity matrix, and the translation T should be along the x-axis only, and the printed values are both fairly close to those. The x-axis distance can be converted to a real-world measurement by multiplying it by the size of a square in the printed chessboard. The squares on my board are 27x27 mm^2, and so OpenCV believes the offset between the cameras is about -3 x 27 = -81 mm. This can be checked by measuring the distance with a ruler, and my cameras are 80 mm apart.

One confusing aspect of the T vector is that its x-axis offset is negative (-3.017 units). It should be positive, since the right camera's translation is specified relative to the left camera, and so should be a vector pointing along the +x axis.

The two distortion matrices, D1 and D2 arrange their parameters so radial values fill the first, second, and fifth arguments, while the tangential distortions are the third and fourth values. The results above show that the tangential values are very close to 0, and so it might benefit the calibration to add the CV_CALIB_ZERO_TANGENT_DIST flag to the cvStereoCalibrate() call.

4.3. Undistorting the Points

The D1 and D2 matrices are utilized by cvUndistortPoints() to cancel out any distortion effects on the chessboard corner points and intrinsic matrices. cvUndistortPoints() is called twice, for the points and intrinsic of each camera:

```
// inside calibrateWithPairs()
cvUndistortPoints( imPts1, imPts1, M1, D1, null, M1);
cvUndistortPoints( imPts2, imPts2, M2, D2, null, M2);
```

The calibration quality can be checked using these undistorted coordinates by analyzing how close the corner points in one image are to their matching epipolar lines in the other image. I'll explain what 'epipolar lines' means by referring to Figure 19.

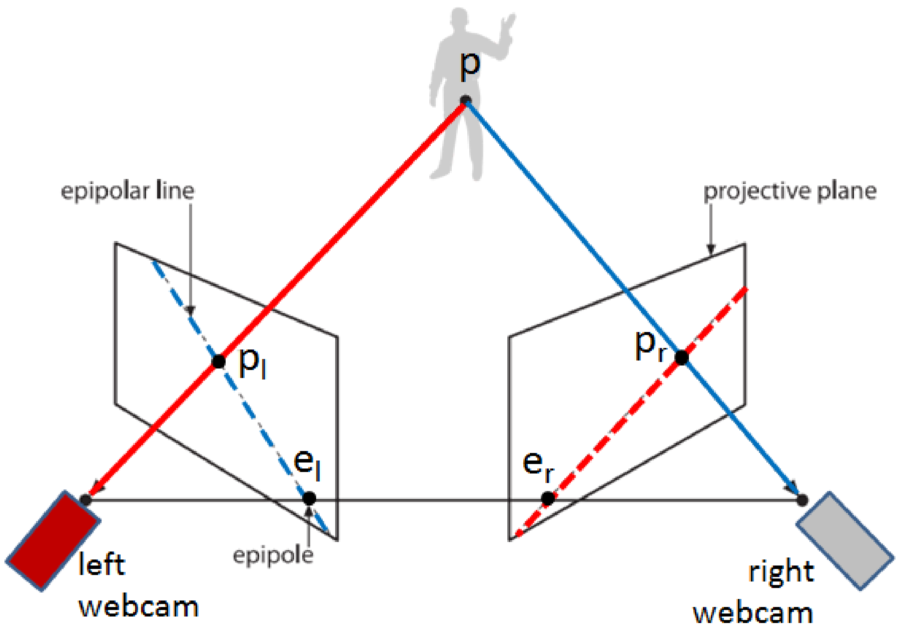


Figure 19. Epipoles and Epipolar Lines.

When the right-hand webcam records the user at P, the point is projected onto the webcam's projective plane, becoming p_r. The user may be located anywhere on the line linking the camera to p_r.

If this line is projected onto the left webcam's projective plane, the result is the epipolar line. This line goes through p_l and the e_l point (called the epipole). e_l is important because it intersects the line linking the two cameras.

The same argument can be applied the opposite way: the left webcam maps P onto p_l, which defines a line for the user's location. This line becomes the epipolar line linking p_r and e_r on the right webcam's projective plane. e_r also falls on the line linking the two cameras.

Why are epipolar lines useful? One answer is that any given point in an image (e.g. p_l in the left webcam) will appear on the corresponding epipolar line on the other image. These epipolar lines can be computed by the OpenCV function cvComputeCorrespondEpilines(), with the help of the fundamental matrix F. If the calibration is good enough, every point in a webcam image should lie on its corresponding epipolar line in the other webcam's image. This matching is carried out by DepthCalc by calling showEpipolarError():

```
private void showEpipolarError(CvMat imPts1, CvMat imPts2, CvMat F)
{
  CvMat L1 = CvMat.create(1, totPoints, CV_32F, 3); //epipolar lines
  CvMat L2 = CvMat.create(1, totPoints, CV_32F, 3);
  cvComputeCorrespondEpilines(imPts1, 1, F, L1);
  cvComputeCorrespondEpilines(imPts2, 2, F, L2);

  double avgErr = 0;
  for(int i = 0; i < totPoints; i++) {
      double err = Math.abs((imPts1.get(0,i,0) * L2.get(0,i,0)) +
                            (imPts1.get(0,i,1) * L2.get(0,i,1)) +
                            L2.get(0,i,2)) +
                   Math.abs((imPts2.get(0,i,0) * L1.get(0,i,0)) +
                            (imPts2.get(0,i,1) * L1.get(0,i,1)) +
                            L1.get(0,i,2));
      avgErr += err;
  }
  System.out.printf("Calibration average error: %.4f\n",
                                       avgErr/totPoints);
}  // end of showEpipolarError()
```

cvComputeCorrespondEpilines() represents each epipolar line as a vector of three parameters (a, b, c) such that the line is defined by the equation:

```
      ax + by + c = 0
```

Inside showEpipolarError()'s for-loop, each (x, y) point in one image is plugged into the corresponding epipolar line equation in the other image, and the result is summed in avgErr. If the calibration is good then the average equation result should be 0; for my image pairs the output was:

```
Calibration average error: 0.6096
```

This confirms that the calibration is satisfactory.

4.4. Hartley's Rectification

Rectification is the adjustment of the cameras' projection planes so they have the same position and orientation. This will mean that the boards inside the paired images will be aligned with each other.

The DepthCalc class uses Hartley's rectification method which begins by calculating homography matrices for the two cameras. A homograph is a mapping which sends a webcam's projective plane to a new orientation and position. The relevant lines in DepthCalc.calibrateWithPairs() are:

```
CvMat H1 = cvCreateMat(3, 3, CV_64F);
CvMat H2 = cvCreateMat(3, 3, CV_64F);
cvStereoRectifyUncalibrated( imPts1, imPts2, F, imSize, H1, H2, 3);
```

The name of the OpenCV cvStereoRectifyUncalibrated() function is a bit misleading since it isn't performing rectification, but generating the homography matrices, H1 and H2, for the webcams. Rectification matricies are computed using these homographs, which are encoded as four lookup maps. All this is done in calculateLookupMaps():

```
private void calculateLookupMaps(CvMat M1, CvMat M2,
                   CvMat D1, CvMat D2, CvMat H1, CvMat H2)
{
  System.out.println("Calculating rectification matrices...");
  CvMat Re1 = cvCreateMat(3, 3, CV_64F);
  CvMat Re2 = cvCreateMat(3, 3, CV_64F);
  CvMat iM = cvCreateMat(3, 3, CV_64F);
  cvInvert(M1, iM);
  cvMatMul(H1, M1, Re1);
  cvMatMul(iM, Re1, Re1);   // Re1 =  iM1 * H1 * M1
  cvInvert(M2, iM);
  cvMatMul(H2, M2, Re2);
  cvMatMul(iM, Re2, Re2);   // Re2 =  iM2 * H2 * M2

  System.out.println("Calculating undistortion/rectification
                                         lookup maps...");
  mx1 = cvCreateMat( IM_HEIGHT, IM_WIDTH, CV_32F);
  my1 = cvCreateMat( IM_HEIGHT, IM_WIDTH, CV_32F);
  mx2 = cvCreateMat( IM_HEIGHT, IM_WIDTH, CV_32F);
  my2 = cvCreateMat( IM_HEIGHT, IM_WIDTH, CV_32F);
  cvInitUndistortRectifyMap(M1, D1, Re1, M1, mx1, my1);   // left
  cvInitUndistortRectifyMap(M2, D2, Re2, M2, mx2, my2);   // right

  System.out.println("Saving maps");
  saveMatrix(STEREO_DIR + "mx1.txt", mx1);
  saveMatrix(STEREO_DIR + "my1.txt", my1);
  saveMatrix(STEREO_DIR + "mx2.txt", mx2);
  saveMatrix(STEREO_DIR + "my2.txt", my2);
}  // end of calculateLookupMaps()
```

The first few lines of calculateLookupMaps() convert the homographs into rectification matrices by incorporating the intrinsic properties of the cameras like so:

$$Re = M^{-1} H M$$

The second half of the method calculates undistortion and rectification transformation for the cameras, as four lookup maps. The use of separate maps for the x- and y-values makes it easier to apply the transformations in the cvRemap() method later. The remap matrices are saved to files, so they can be loaded by subsequent calls to DepthViewer that depth process an image pair.

4.5. Bouquet's Rectification

Figure 14 includes one more task before the calibration is finished: the calculation of the rectification transforms again, this time with Bouquet's method. This may seem a little pointless, and it is possible to skip this stage since the calibration process already has enough information from the other matrices it has generated.

One reason for performing rectification again is that it provides an easy, fast way to double-check the existing results.

calculateQ() generates the reprojection matrix Q, which contains the left camera's focal length and the baseline distance between the cameras. In detail Q is:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/T_x & (c_x - c'_x)/T_x \end{bmatrix}$$

The focal length, *f*, is in cell Q[2][3], and T_x is -1/(Q[3][2]).

c_x and c_y are the optical center of the left camera, while c'_x is the optical x-coordinate of the right image. If the cameras are parallel to the z-axis then c_x and c'_x should be equal.

What are the differences between Hartley's and Bouquet's methods? Hartley's approach relies on matching common points in the image pairs, and so can be utilized in situations where calibration patterns, such as chessboards, aren't available. For example, Hartley's method can be applied to consecutive video frames recorded by a single camera.

The OpenCV function that implements Hartley, cvStereoRectifyUncalibrated(), only needs the fundamental matrix F as input in addition to the points information. Although I didn't do so in this chapter, F can be computed using point matching between two pictures in the scene (with the cvFindFundamentalMat() function). This explains why the word "uncalibrated" appears in the cvStereoRectifyUncalibrated() name.

Bouquet's method is implemented by cvStereoRectify() which requires several matrices generated during the calibration phase, including the rotation and translation (R and T) relating the right-hand webcam to the left. cvStereoRectify() also employs the webcams' intrinsic and distortion matrices. All this extra information simplifies the rectification task carried out by Bouquet's algorithm, and allows it to produce more accurate results.

The calculateQ() method:

```
// globals
```

```
private CvMat Q;
private double focalLength, baselineDist;
           // parts of the reprojection matrix, Q


private void calculateQ(CvMat M1, CvMat M2, CvMat D1, CvMat D2,
                        CvSize imSize, CvMat R, CvMat T)
{
  // the rectification matrices using Bouquet
  CvMat Reb1 = cvCreateMat(3, 3, CV_64F);
  CvMat Reb2 = cvCreateMat(3, 3, CV_64F);

  // projection matrices in the rectified coordinate system
  CvMat P1 = cvCreateMat(3, 4, CV_64F);
  CvMat P2 = cvCreateMat(3, 4, CV_64F);

  // Q is the 4x4 reprojection matrix
  Q = cvCreateMat(4, 4, CV_64F);
  cvStereoRectify(M1, M2, D1, D2, imSize, R, T, Reb1, Reb2, P1, P2,
                  Q, CV_CALIB_ZERO_DISPARITY, -1,
                  cvSize(0,0), null, null);
  printMatrix("Reprojection Matrix Q:", Q);

  // save focal length and baseline distance in globals
  focalLength = Q.get(2,3););
  baselineDist = -1.0/Q.get(3,2);
  System.out.printf("Focal length: %.4f\n", focalLength);
  System.out.printf("Baseline distance: %.4f\n", baselineDist);

  System.out.println("Saving reprojection matrix");
  saveMatrix(STEREO_DIR + "q.txt", Q);

/*
  CvMat diff = cvCreateMat(3, 3, CV_64F);
  cvSub(Reb1, Re1, diff, null);   // diff = Reb1 - Re1
  printMatrix("Rectification difference for camera 1:", diff);
  cvSub(Reb2, Re2, diff, null);   // diff = Reb2 - Re2
  printMatrix("Rectification difference for camera 2:", diff);
*/
}  // end of calculateQ()
```

cvStereoRectify() computes two rectification matrices, Reb1 and Reb2, which could be compared with the rectifications generated by Hartley's method (Re1 and Re2 in calculateLookupMaps()). Commented-out code for doing this appears at the end of calculateQ().

Q's focal length and baseline values are stored in globals, so they will be available during depth processing later. Q can also be used for generating a point cloud, although I don't use that approach.

When this part of DepthViewer is executed, the following results are printed:

```
Reprojection Matrix Q:
|  1.000   0.000   0.000  -292.396 |
|  0.000   1.000   0.000  -209.151 |
|  0.000   0.000   0.000   690.698 |
|  0.000   0.000   0.330    -0.000 |

Focal length: 690.6976
```

```
Baseline distance: -3.0261
```

The focal length of the left camera has already been calculated, as the f_x and f_y values in the M1 matrix. The baseline was also previously computed in the T matrix, as its x-value. That offset can be converted to real-world units by multiplying it by the size of a chessboard square (27 mm), producing a camera separation distance of about -82 mm, which is very close to the actual distance (80 mm).

Although calculateQ() isn't that useful, I've left it in the program as an example of how to use Bouquet's method. It's presence only adds a small execution overhead to the calibration.

5. Depth Processing

After calculateQ() returns, the DepthCalc constructor switches to the depth processing of the first image pair. It does this to test the quality of the four lookup maps in mx1, my1, mx2, and my2, and the Q matrix.

It's also possible to start DepthCalc directly in depth-processing mode, where it bypasses the calibration steps and attempts to load previously saved lookup maps. The user supplies an image pair ID on the command line, and those images are analyzed. For example:

```
> run DepthViewer –p 7
```

will initiate depth processing on image pair number 7.

Figure 20 shows a flowchart of the main steps involved in depth processing.

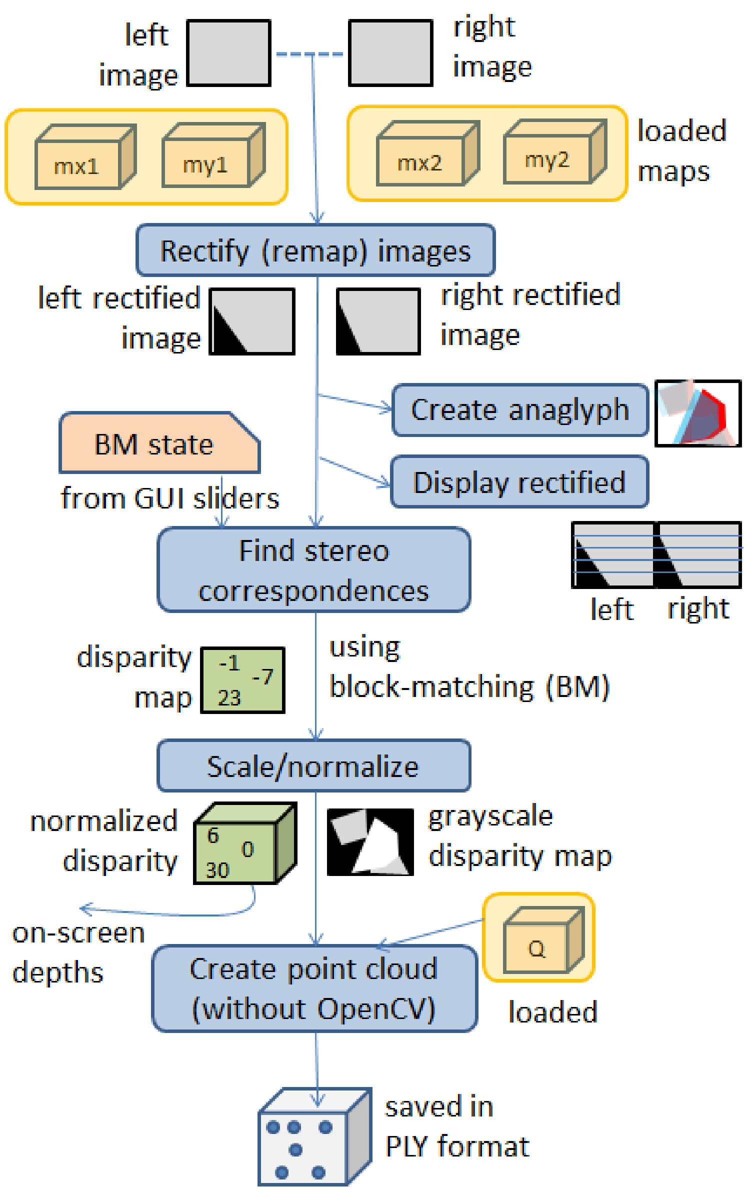


Figure 20. The Depth Processing Steps.

The choice of left and right images depends on their ID, which is passed to the depthProcessing() method:

```
// globals
private IplImage[] imagesRectified = null;
             // rectified image pair being depth processed

private CvMat normalizedDisp;
            // disparity data used for depth calculations

// private CvMat xyzMat;


private void depthProcessing(int ID)
{
  imagesRectified = rectify(ID);
  showRectifiedImages(imagesRectified);
  showAnaglyph(imagesRectified);
  normalizedDisp = createDisparityMaps(imagesRectified);

/* System.out.println("Generating point cloud");
  xyzMat = cvCreateMat(IM_HEIGHT, IM_WIDTH, CV_32FC3);
  cvReprojectImageTo3D(normalizedDisp, xyzMat, Q, 0);
*/
}  // end of depthProcessing()
```

depthProcessing() originally called the OpenCV cvReprojectImageTo3D() function to create the depth data for the point cloud. The code is commented out since I decided to generate my own point cloud, which makes it easier to filter and scale the data, and add color information. In addition, DepthViewer only creates a point cloud when the user clicks on the application's close box. The details are explained in section 6.

5.1. Rectification

rectify() uses the lookup maps to undistort and rectify the image pair.

```
// globals
private CvMat mx1, my1, mx2, my2;
                // lookup maps for the left and right cameras


private IplImage[] rectify(int ID)
{
  System.out.println("\nDepth processing image pair " + ID + "...");

  IplImage leftIm = cvLoadImage(makeName("left", ID), 0);
  IplImage rightIm = cvLoadImage(makeName("right", ID), 0);
  if ((leftIm == null) || (rightIm == null)) {
    System.out.println("Error loading image pair " + ID);
    System.exit(1);
  }

  // undistort and rectify the images
  IplImage[] imagesRectified = new IplImage[2];
  imagesRectified[LEFT] =
       IplImage.create(cvGetSize(leftIm), IPL_DEPTH_8U, 1);
  imagesRectified[RIGHT] =
       IplImage.create(cvGetSize(rightIm), IPL_DEPTH_8U, 1);
  cvRemap(leftIm, imagesRectified[LEFT], mx1, my1,
                  CV_INTER_LINEAR | CV_WARP_FILL_OUTLIERS,
                  CvScalar.ZERO);
  cvRemap(rightIm, imagesRectified[RIGHT], mx2, my2,
                  CV_INTER_LINEAR | CV_WARP_FILL_OUTLIERS,
                  CvScalar.ZERO);
  return imagesRectified;
}  // end of rectify()
```

The images are distorted so the chessboards fall into alignment. Rectification doesn't mean that the chessboards are twisted back into rectangular shape; instead, it aligns the board rows so they're parallel, but perhaps still at an angle relative to the image borders. Any chessboard lines which were radially distorted in the original should be straightened.

The rectified images are displayed side-by-side by showRectifiedImages(). It also draws colored lines across the images to help with the comparison, as in Figure 21.

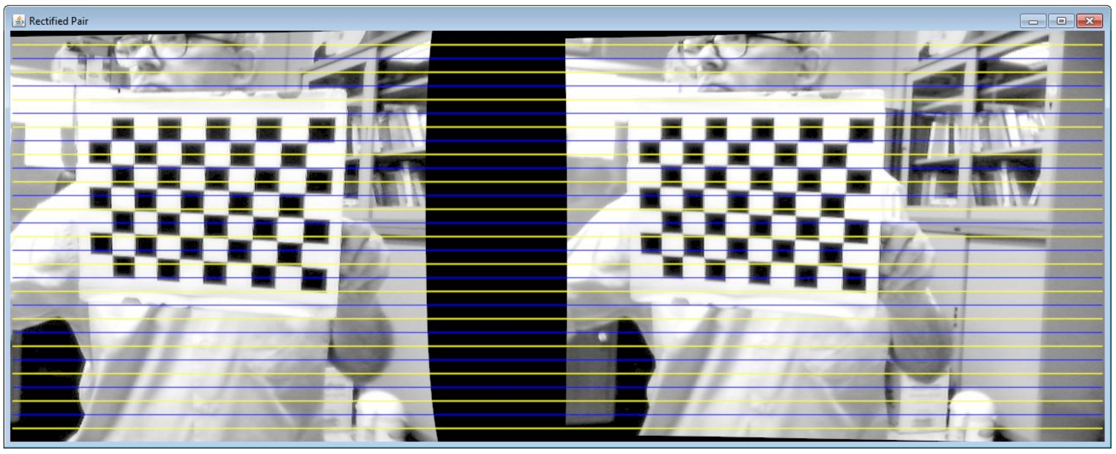


Figure 21. Rectified Images with Lines.

The showRectifiedImages() code:

```
// globals
private static final int IM_WIDTH = 640;
private static final int IM_HEIGHT = 480;

private CanvasFrame displayFrame;
          // used to display chessboard points and rectified images


private void showRectifiedImages(IplImage[] imagesRectified)
{
  IplImage imPair =
       IplImage.create(IM_WIDTH*2, IM_HEIGHT, IPL_DEPTH_8U, 3);
            // a color image big enough for both input images
  IplImage colTemp =
       IplImage.create(IM_WIDTH, IM_HEIGHT, IPL_DEPTH_8U, 3);
            // temporary color image object

  // copy left image to left side
  cvCvtColor(imagesRectified[LEFT], colTemp, CV_GRAY2RGB);
  cvSetImageROI(imPair, cvRect(0, 0, IM_WIDTH, IM_HEIGHT));
  cvCopy(colTemp, imPair);

  // copy right image to right side
  cvCvtColor(imagesRectified[RIGHT], colTemp, CV_GRAY2RGB);
  cvSetImageROI(imPair, cvRect(IM_WIDTH, 0 , IM_WIDTH*2, IM_HEIGHT));
  cvCopy(colTemp, imPair);

  // reset large image's Region Of Interest
  cvResetImageROI(imPair);

  // draw yellow and blue lines across the images
  CvScalar color;
  int count = 0;
  for(int i = 0; i < IM_HEIGHT; i += 16) {
    color = (count%2 == 0) ? CvScalar.BLUE : CvScalar.YELLOW;
    cvLine(imPair, cvPoint(0,i), cvPoint(IM_WIDTH*2, i),
                                       color, 1, CV_AA, 0);
    count++;
  }

  // display the result
  if (displayFrame == null)
    displayFrame = new CanvasFrame("Rectified Pair");
  else {
    displayFrame.setTitle("Rectified Pair");
    displayFrame.setSize(IM_WIDTH*2, IM_HEIGHT);
  }

  displayFrame.showImage(imPair);
}  // end of showRectifiedImages()
```

5.2. Creating an Anaglyph

An anaglyph is made from combining two colored-filtered images for the left and right eye. Typical filters are red and cyan (a greenish-blue), and the resulting image should be viewed through glasses that have a red filter over the left eye and a cyan one over the right (see Figure 22).

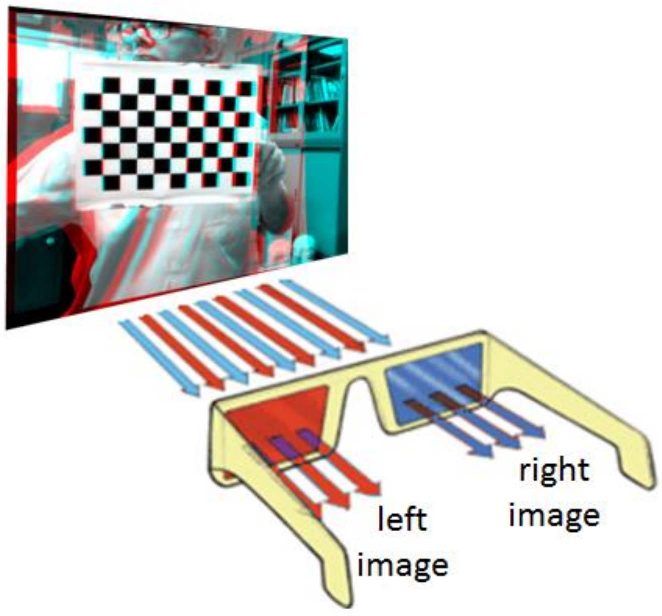


Figure 22. Viewing an Anaglyph.

The filtering means that the red component of the anaglyph will be most visible to the left eye and the cyan component most visible to the right eye. Therefore, to create the stereo effect, the image for the left eye should be encoded using red and the right eye image in cyan. This is easy to implement in OpenCV by using cvMerge() to fill the red, green, and blue channels of a new IplImage object. The only tricky aspect is remembering that IplImage encodes its channels in BGR order (e.g. channel 0 is for blue). The resulting code is in DepthCalc.showAnaglyph():

```
// globals
private static final int LEFT = 0;
private static final int RIGHT = 1;

private static final int IM_WIDTH = 640;
private static final int IM_HEIGHT = 480;

private static final String ANA_FNM = "anaglyph.jpg";


private void showAnaglyph(IplImage[] imagesRectified)
{
  IplImage anaImg =
     IplImage.create(IM_WIDTH, IM_HEIGHT, IPL_DEPTH_8U, 3);
             // color image object with BGR channel ordering

  cvMerge(imagesRectified[RIGHT], imagesRectified[RIGHT],
                         imagesRectified[LEFT], null, anaImg);
     // BGR: (blue, green) = cyan = right image;  red = left image

  // display the result
  CanvasFrame anaFrame = new CanvasFrame("Anaglyph");
  anaFrame.showImage(anaImg);

  System.out.println("Saving anaglyph to " + ANA_FNM);
  cvSaveImage(ANA_FNM, anaImg);
}  // end of showAnaglyph()
```

5.3. Finding Stereo Correspondences

Stereo correspondence is the calculation of disparities (offsets) between common points that appear in both the left and right images. A disparity can easily be converted into a depth measurement since disparity $\propto$ 1/depth.

We use disparity calculations all the time when we judge distances. Objects that are close to us have a large disparity while those further away have a smaller disparity. You can see this by looking at something near to you, first with your left eye, then your right. The object appears to jump to the left (relative to your eye's field of view) when you switch to your right eye. The jump is much less noticeable if you focus on something in the mid-distance, and the disparity drops to 0 for objects far away.

The same technique can be applied to the left and right rectified images in an efficient manner because of their alignment. It means that a point at (x, y) in the left image will be located somewhere to the left of that position in the right image. The idea is illustrated in Figure 23.

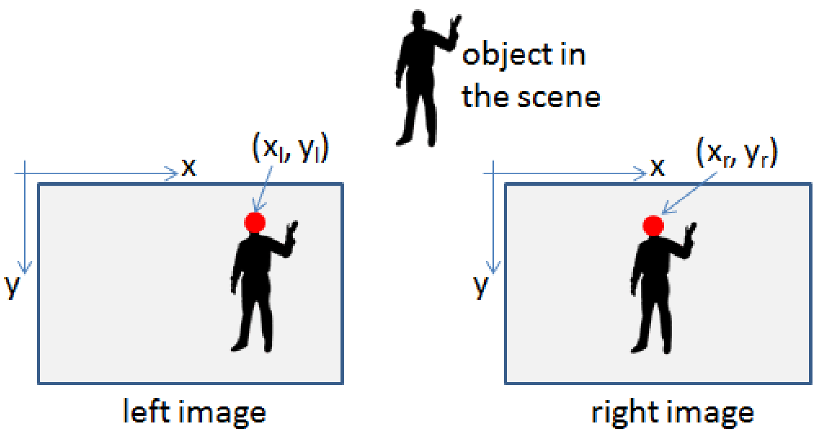


Figure 23. Finding the Disparity for a Point in Two Images.

y_l and y_r will be the same (due to rectification), so the disparity is $(x_l – x_r)$.

When disparities are calculated for all the points in the scene, the largest will occur for objects close to the cameras, and smaller disparities will be apparent for things that are further away, as shown in Figure 24.

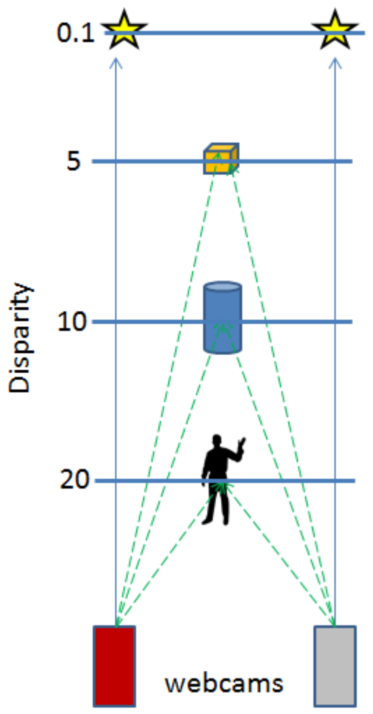


Figure 24. Disparity values Relative to the Cameras.

The inverse relationship between disparities and depths means that large disparities (i.e. objects close to the camera) have small depths, while small disparities (i.e. objects far away) have large depths, as illustrated by Figure 25.

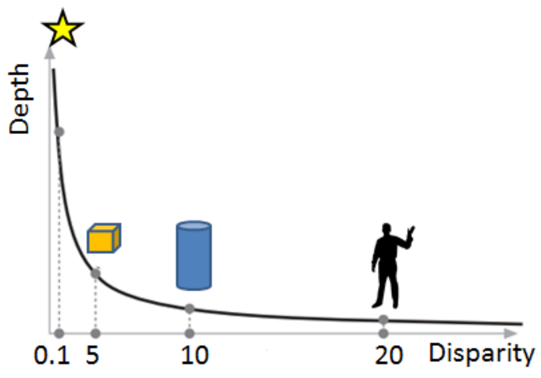


Figure 25. Disparity and Depth Inverse Relationship.

The inverse relationship also means that depths calculated from disparities are only accurate for objects close to the camera. Once an object is a reasonable distance away, a small error in its disparity can introduce a large change in its depth.

OpenCV offers a few functions for calculating stereo correspondences. The most popular is cvFindStereoCorrespondenceBM() (BM stands for block-matching) which is fast, and explained in excellent detail in *Learning OpenCV*.

Another function is cvFindStereoCorrespondenceGC() (GC stands for graph-cut), which is more accurate but too slow for most real-time computer vision applications. It has been replaced in newer versions of OpenCV by a function using semi-global block matching, a fast variant of cvFindStereoCorrespondenceBM(). It is used in the stereo_match.cpp example that come in the OpenCV download.

I've utilized cvFindStereoCorrespondenceBM(), which is fast enough to be adjusted at run-time via the sliders in DepthViewer's GUI (see the left of Figure 3).

The block-matching parameters used by cvFindStereoCorrespondenceBM() are both numerous and fairly confusing since the algorithm passes through three stages:

1. **Prefiltering**. The input images' brightnesses are normalized, and texturing is enhanced using a moving window of a specified pixel block size.

2. **Correspondence search**. The disparities between common points are calculated using a moving SAD (sum of absolute differences) window which looks for matching, points as in Figure 23. To speed things up, SAD matches *blocks* of pixels (hence, the algorithm's name), with larger windows being faster and more resistant to noise but also creating blurrier disparity data.

There are many parameters for controlling where the search begins on the right-hand image's row, and how much of the row to search.

3. **Postfiltering**. Poor matches are removed using a uniqueness ratio, texture thresholding, and speckle size to separate noise from the true texture matches.

The computed disparities are stored as integers after being multiplied by 16 and rounded; actual disparities are retrieved by dividing by 16.

5.4. Producing Disparity Maps

Initial block-matching parameters are passed to the DepthCalc object when it's created in the DepthViewer class:

```
// in the DepthViewer class
int preFilterSize = 31;
int prefilterCap = 31;
int sadSize = 15;

int minDisp = -100;
int numDisp = 128;

int uniqRatio = 15;
int texThresh = 10;
int specSize = 100;
int specRange = 4;

depthCalc = new DepthCalc(val, isCalibrating, drawChessboards,
                        preFilterSize, prefilterCap, sadSize,
                        minDisp, numDisp, uniqRatio,
                        texThresh, specSize, specRange);
```

I based these values on those in the cvFindStereoCorrespondenceBM() example in *Learning OpenCV* (p.451) and other online programs. They are also used to initialize the GUI's sliders.

The DepthCalc constructor creates a CvStereoBMState object which is used by cvFindStereoCorrespondenceBM() later:

```
// globals
private CvStereoBMState bmState;

// in DepthCalc()
bmState = cvCreateStereoBMState(CV_STEREO_BM_BASIC, 0);
bmState.preFilterSize(preFilterSize);    // prefilters
bmState.preFilterCap(prefilterCap);

bmState.SADWindowSize(sadSize);          // SAD-related

bmState.minDisparity(minDisp);
bmState.numberOfDisparities(numDisp);    // postfilters
bmState.textureThreshold(texThresh);
bmState.uniquenessRatio(uniqRatio);
bmState.speckleWindowSize(specSize);
bmState.speckleRange(specRange);
```

DepthCalc.createDisparityMaps() creates a disparity matrix by calling cvFindStereoCorrespondenceBM():

```
// globals
private IplImage gDispMap;         // grayscale disparity image
private CvMat normalizedDisp;
                   // disparities used for depth calculations
```

```
private CvMat createDisparityMaps(IplImage[] imagesRectified)
{
  CvSize imSize = cvGetSize(imagesRectified[LEFT]);

  IplImage disparityMap = IplImage.create(imSize, IPL_DEPTH_16S, 1);
  cvFindStereoCorrespondenceBM(imagesRectified[LEFT],
                               imagesRectified[RIGHT],
                               disparityMap, bmState);

  // convert disparity map to a grayscale disparity image
  gDispMap = IplImage.create(imSize, IPL_DEPTH_8U, 1);
  cvNormalize(disparityMap, gDispMap, 0, 255, CV_MINMAX, null);

  // normalize the disparity map; use this for depth calculations
  CvMat normalizedDisp = cvCreateMat(IM_HEIGHT, IM_WIDTH, CV_32F);
  cvConvertScale(disparityMap, normalizedDisp, 1.0/16, 0);

  if (minVal[0]*maxVal[0] < 0) // is there a sign change?
    cvConvertScale(normalizedDisp, normalizedDisp, 1, -minVal[0]);
                                // move to be all positive
  return normalizedDisp;
}  // end of createDisparityMaps()
```

The disparity matrix created by cvFindStereoCorrespondenceBM() is stored as an IplImage object, but this is just a convenience for the creation of the grayscale disparity image at the next stage of the method. cvNormalize() scales the disparity data so it ranges between 0 and 255, thereby allowing the values to be treated as grayscales. The resulting global grayscale image, gDispMap, can be accessed via the top-level DepthViewer application, and drawn in its right-hand panel, as shown in Figure 3.

A second disparity map, called normalizedDisp, is also created, as the first step in calculating the depth data for the point cloud. As I mentioned above, disparities are stored as integers after being multiplied by 16 and rounded, and so normalizedDisp stores the true disparities by dividing the data by 16.

While I was debugging this code, I included println() calls to report on the minimum and maximum values in the matrices and maps. To my surprise, the disparity maps included negative values. For example, the following code fragment:

```
CvMat normalizedDisp = cvCreateMat(IM_HEIGHT, IM_WIDTH, CV_32F);
cvConvertScale(disparityMap, normalizedDisp, 1.0/16, 0);

// debugging code
double[] minVal = new double[1];
double[] maxVal = new double[1];
cvMinMaxLoc(normalizedDisp, minVal, maxVal);
System.out.println("Normalized disparity map range: " +
                                  minVal[0] + " - " + maxVal[0]);
```

produces the output:

```
Normalized disparity map range: -101.0 - 10.75
```

The usual reason for negative disparities is that the webcams aren't aligned in parallel, instead pointing slightly towards each other. I decided to fix the problem with a hack involving a data shift:

```
if (minVal[0]*maxVal[0] < 0)  // is there a sign change?
  cvConvertScale(normalizedDisp, normalizedDisp, 1, -minVal[0]);

// debugging code
cvMinMaxLoc(normalizedDisp, minVal, maxVal);
System.out.println("Translated norm disparity map range: " +
                                  minVal[0] + " - " + maxVal[0]);
```

The effect of the cvConvertScale() call is to shift the data to the right along the x-axis, making it all positive. The println() in the above code fragment reports:

```
Translated norm disparity map range: 0.0 - 111.75
```

This shift is not ideal since the left-most disparity (e.g. at the old disparity of -101 in the example above) is shifted to 0. This means that the corresponding depth is now at infinity.

5.5. Calculating Depths

As seen in Figure 3, a depth value is drawn on top of the grayscale disparity map when the user clicks on it. The ImagePanel object obtains the number by calling DepthCalc.getDepth() with the (x, y) coordinate where the mouse was pressed:

```
// global
private CvMat normalizedDisp;


public int getDepth(int x, int y)
{
  if (normalizedDisp == null)
    return 0;
  return disparity2Depth( normalizedDisp.get(y,x));
}  // end of getDepth()
```

The normalizedDisp get() call reverses the (x, y) order since OpenCV stores matrices in row-column order.

The disparity-to-depth conversion is performed by disparity2Depth() which implements the equation Z = (f*T)/d to convert disparity values (d) into depths (Z); f is the focal length and T the baseline distance, which both come from the reproduction matrix Q.

```
// globals
private static final int CHESS_SQ_LENGTH = 27;  // mm

private double focalLength, baselineDist;
        // obtained from the reprojection matrix, Q


private int disparity2Depth(double disp)
{ if (disp == 0)    // an infinite depth
```

```
    return 0;
  else
    return -(int)Math.round((
                focalLength*baselineDist)/disp*CHESS_SQ_LENGTH);
}
```

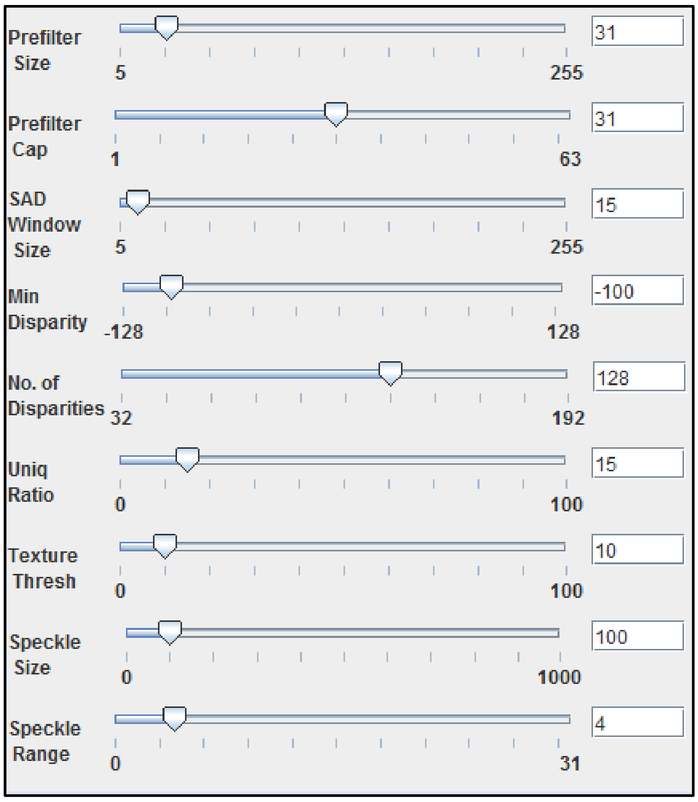
The CHESS_SQ_LENGTH value comes from measuring the chessboard print-out with a ruler, and converts the depth from a value in pixels into millimeters. The depth returned by the equation is negative since depths are positioned along the negative z-axis (see Figure 9). However, it seems less confusing for the user to display the depth as a positive, so the number is negated before being returned. In addition, infinite depths are represented by 0.

At this point, an obvious question is whether these depths bear any relationship to real-world distances? The good news is that two depths can always be compared since a smaller depth does means a point closer to the camera.

The bad news is that the accuracy of a depth depends on the point's proximity to the camera. Points closer to the camera are represented by more disparity values, and so their depths will be more accurate. Points further away will be less accurately defined. Another problem is the hack I utilized in createDisparityMaps() to make all the disparities positive. A side-effect of that numerical shift is the introduction of another error into the disparity data.

5.6. Changing the Stereo Correspondence Settings

Figure 26 shows a close-up of the slider panel in the DepthViewer application.



Prefilter Size 5 255 31
Prefilter Cap 1 63 31
SAD Window Size 5 255 15
Min Disparity -128 128 -100
No. of Disparities 32 192 128
Uniq Ratio 0 100 15
Texture Thresh 0 100 10
Speckle Size 0 1000 100
Speckle Range 0 31 4

Figure 26. The Slider Controls in DepthViewer.

When a slider is moved, the GUI responds by adjusting the corresponding block-matching parameter, and calling DepthCalc.createDisparityMaps() (and hence cvFindStereoCorrespondenceBM()) to recalculate the disparity data.

There are a few issues with this coding approach. One is that some parameters have restrictions on their allowed values. For example, the minimum disparity (i.e. the "Min Disparity" slider) must be a multiple of 16, which isn't enforced by the slider component itself. It's only in the code triggered by the slider change that the value is checked, and possibly corrected. Unfortunately, the GUI doesn't show the corrected value, and the user only knows about the change via a warning message printed to standard output.

Another potential problem is that DepthCalc.createDisparityMaps() is called in Java's GUI thread, which means that the GUI 'freezes' until the block match has been recomputed and createDisparityMaps() returns. This only takes around a second, so the delay isn't too severe, but a slower algorithm, such as graph-cut correspondence, would cause the GUI to hang for too long.

6. Building a Point Cloud

The point cloud isn't computed until the user presses the application's close box. The reasoning behind this is that the user may want to adjust the disparity data with the sliders before generating the final cloud. I'm not really happy with this design since the closing action hangs for 1-2 seconds while the point cloud is calculated and written out to a file.

Cloud creation is triggered by a window listener connected to the top-level DepthViewer JFrame:

```
// code fragment in DepthViewer()
addWindowListener( new WindowAdapter() {
  public void windowClosing(WindowEvent e)
  { try {
      setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
      depthCalc.storeDepthInfo();
    }
    catch(Exception ie) {}
    finally {
      setCursor(Cursor.getDefaultCursor());
    }
    System.exit(0);
  }
});
```

DepthCalc.storeDepthInfo() writes information to three different files – a PLY point cloud is saved, as are the current slider settings and the grayscale disparity image. The surrounding calls to setCursor() are meant to change the cursor to a waiting icon to suggest that something is happening during the closing delay. However, the cursor doesn't change on my test machines.

The interesting part of storeDepthInfo() is the call to savePly() which builds a 2D array of depths, and writes it out in PLY format to a text file:

```
// globals
private IplImage gDispMap;            // grayscale disparity image
private CvMat normalizedDisp;
                     // disparity data used for depth calculations


private void savePly(String fnm, CvMat gDispMat)
{
  int rows = gDispMat.rows();
  int cols = gDispMat.cols();
  int totalVerts = rows*cols;
  System.out.println();

  double[][] pclCoords = new double[cols][rows];    // x then y
  int numZeros = convertDisparities(normalizedDisp,
                                   pclCoords, rows, cols);
  if (numZeros > 0) {
    double percentZeros = 100.0*((double)numZeros)/totalVerts;
    System.out.printf("No. of vertices with 0 depth:
                %d (%.1f%%)\n",numZeros, percentZeros);
  }
  int dataVerts = totalVerts - numZeros;
                        // ignore points with 0 depth

  System.out.println("Saving point cloud coordinates to " + fnm);
  try {
    PrintWriter out = new PrintWriter(new FileWriter(fnm));

    out.println("ply");
    out.println("format ascii 1.0");
    out.println("comment Point Cloud output from DepthCalc");

    // x, y, z coordinate for a point and colors
    out.println("element vertex " + dataVerts);
    out.println("property double x");    // vertex coordinates
    out.println("property double y");
    out.println("property double z");
    out.println("property uchar red");   // vertex colors
    out.println("property uchar green");
    out.println("property uchar blue");
    out.println("end_header");

    for (int x=0; x < cols; x++)
      for (int y=0; y < rows; y++) {
        double depth = (double)pclCoords[x][y];
        int gray = (int) gDispMat.get((rows-1-y),x);   // grayscale
        if (depth != 0)    // do not save vertices with 0 depths
          out.printf("%d  %d  %.3f  %d  %d  %d\n",
                        x, y, depth, gray, gray, gray);
      }
    out.flush();
    out.close();
  }
  catch(IOException ex)
  {  System.out.println("Unable to save");  }
}  // end of savePly()
```

The depth information comes from the normalized disparity map, normalizedDisp, and also employs the grayscale disparity map in gDispMap to add color (gray levels) information

PLY is a popular text-based format for representing point clouds, and is supported by most 3D modeling tools, including the open source MeshLab (http://meshlab.sourceforge.net/). Unlike some 3D applications, MeshLab is relatively simple to use, since it focuses on the editing of triangular meshes. It's easy to add and delete points, reformat the data (e.g. stretch out the z-axis), and move around the point cloud (see Figure 4).

The PLY format is aimed at representing shapes as collections of points and faces (http://paulbourke.net/dataformats/ply/), and it's easy to add additional properties such as color, textures, transparencies, and normal information. Depending on the shape, unnecessary details can be left out – in my case, I don't store face data.

savePly() writes out a PLY header which describes the element types used by the shape. The details include the element name ("vertex"), how many vertices are in the cloud, and a list of a vertex's properties. A point cloud vertex has six properties: a (x, y, z) position and an RGB color.

The header in the resulting PLY file looks something like:

```
ply
format ascii 1.0
comment Point Cloud output from DepthCalc
element vertex 41456
property double x
property double y
property double z
property uchar red
property uchar green
property uchar blue
end_header
```

Lines of data follow the header, with one line for each vertex. For instance, the first few lines may look like:

```
34  201  -870.653  187  187  187
34  206  -878.257  186  186  186
34  207  -878.257  186  186  186
34  208  -879.525  186  186  186
34  209  -880.792  185  185  185
      :
```

The first three numbers on a line are the point's (x, y, z) coordinate, and the last three define its RGB color. The color is encoded by repeating the grayscale value for that point in the grayscale disparity map.

Each coordinate comes from the pclCoords[][] 2D array – the x- and y- values are the subscripts of the array while the z-value is that cell's value. pclCoords' depths are calculated in convertDisparities() using the disparities from the normalized matrix, normalizedDisp.

One advantage of this approach is the opportunity to reduce the size of the generated point cloud. The grayscale disparity map, such as the one in Figure 2, contains a lot of black, which indicates that the disparities at those coordinates couldn't be calculated. There's no benefit to including that information in the point cloud, and removing it will substantially reduce the size of the file.

convertDisparities() returns a depth value of 0 for those "black" points, and savePly() uses this value to skip the storage of those vertices. The method reports the percentage of zero points in the data, which is often very high (75% to 80% is common). The resulting point cloud is much smaller, and only points with actual depths are displayed (see Figure 4).

6.1. From Disparities to Depths

convertDisparities() converts the normalized disparity map matrix into a 2D array of depths called pclCoords. It also performs several additional transformations, which is my reason for implementing convertDisparities() rather than using OpenCV's cvReprojectImageTo3D() conversion function. You may recall that I'd commented out its call at the end of DepthCalc.depthProcessing().

The extra transformations include changing the y-axis so its origin starts at the bottom-left rather than the top-left. Also, the array data is re-organized into the more familiar column-row order (x then y) rather than the matrix row-column order.

The hardest transformation is scaling the data so that different depths are clear inside the point cloud. The reason for the difficulty is the highly skewed nature of the data, with the vast majority of depth measurements clustered near to the camera, and a few very large numbers (which are mostly noise). These values mean that scaling based on the range of depths (minimum to maximum) would lead to most of the fine-level depth differences close to the camera being lost.

The solution is to base the range on percentiles: the depth data is sorted, and the range is limited to span the minimum depth (i.e. the 0th percentile) up to the 98th percentile, causing very large data to be excluded. In addition, any depths which exceed the 98th percent are set to 0, which allows savePly() to treat them as 'black' points that aren't saved in the point cloud file.

The first half of convertDisparities() applies the axis changes, and the second half uses the percentile-based range to scale the depths:

```
private int convertDisparities(CvMat normalizedDisp,
                        double[][] pclCoords, int rows, int cols)
{
  System.out.println("Converting disparity map into point cloud");
  ArrayList<Integer> depths = new ArrayList<Integer>();

  // convert disparities to depths
  int numZeros = 0;
  int z;
  for (int i=0; i < rows; i++) {
    for (int j=0; j < cols; j++) {
      z = disparity2Depth( normalizedDisp.get(i,j));
      if (z == 0)
        numZeros++;
        else {
```

```
        pclCoords[j][rows-1-i] = z;
                // x-axis goes to the right, y-axis runs up screen
        depths.add(z);
      }
    }
  }

  // calculate a scale factor
  Collections.sort(depths);
  int minDepth = depths.get(0);
  int maxDepth = findMaxDepth(depths);
  System.out.println("min - max depth: " +
                       minDepth + " - " + maxDepth);
  double scaleFactor = ((double)cols)/(maxDepth - minDepth);
  System.out.printf("Scale factor: %.3f\n", scaleFactor);

  // scale the depths
  double pz;
  for (int x=0; x < cols; x++) {
    for (int y=0; y < rows; y++) {
      pz = pclCoords[x][y];
      if (pz != 0) {
        if (pz > maxDepth) {    // ignore depths that are too big
          pclCoords[x][y] = 0;
          numZeros++;
        }
        else
          pclCoords[x][y] = -scaleFactor*(pz - (minDepth-10));
                      // scale depths along -z axis
      }
    }
  }

  return numZeros;
}  // end of convertDisparities()
```

One aspect of the scaling is that the depths will be positive, but they need to be converted to negative numbers so they will be correctly ordered along the –z axis.

findMaxDepth() is passed a sorted list of depths, which allows it to calculate a percentile position using getPercentile():

```
private int findMaxDepth(ArrayList<Integer> depths)
{
  int dVal = getPercentile(depths, 0.98);
  int maxMult =  10*depths.get(0);   // large multiple of minimum
  int maxDepth = (maxMult < dVal) ? maxMult : dVal;
                 // use smaller of two
  return maxDepth;
}  // end of findMaxDepth()


private int getPercentile(ArrayList<Integer> list, double percent)
{
  if ((percent < 0) || (percent > 1.0)) {
    System.out.println("% should be between 0 and 1; using 0.5");
    percent = 0.5;
  }
  int pcPosn = (int)Math.round(list.size()-1 * percent);
  return list.get(pcPosn);
}  // end of getPercentile()
```

The choice of 98th percentile was arrived at by testing out multiple disparity maps. As a fall-back position, a large multiple of the minimum depth is also calculated, and the smaller of the two is used as the maximum depth.