CSCI 574 Spring 2014 Project 1 Report

Divya Ramesh(dramesh@usc.edu)

February 26, 2014

1. Prove the depth-disparity equation in detail.

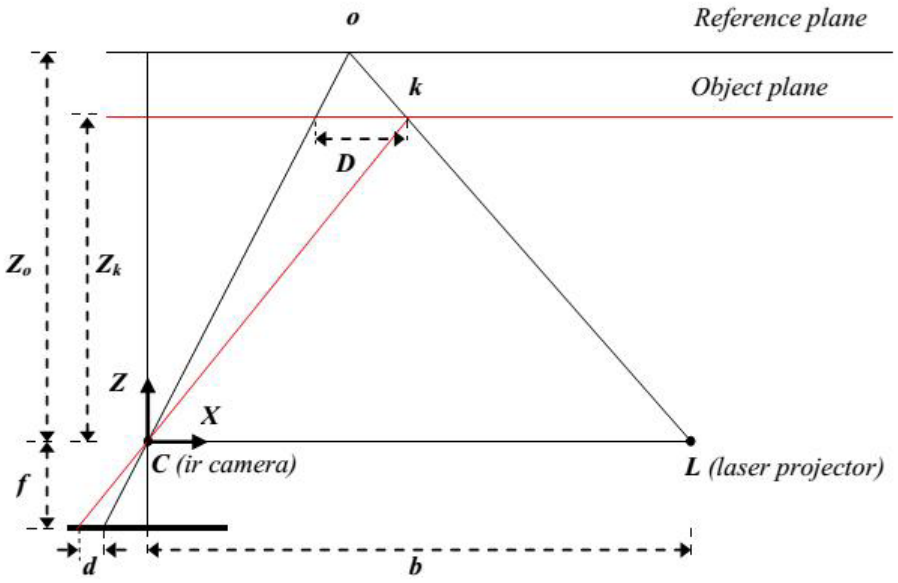


Figure 1: Computational Model.

Consider the triangle formed between the reference and object planes as shown in 1. This is similar to the triangle oCL. By the property of similar triangles, we have,

$$\frac{D}{b} = \frac{Z_0 - Z_k}{Z_0} \tag{1}$$

Considering the triangles formed by D and d with C as similar triangles, we have,

$$\frac{D}{d} = \frac{Z_k}{f} \tag{2}$$

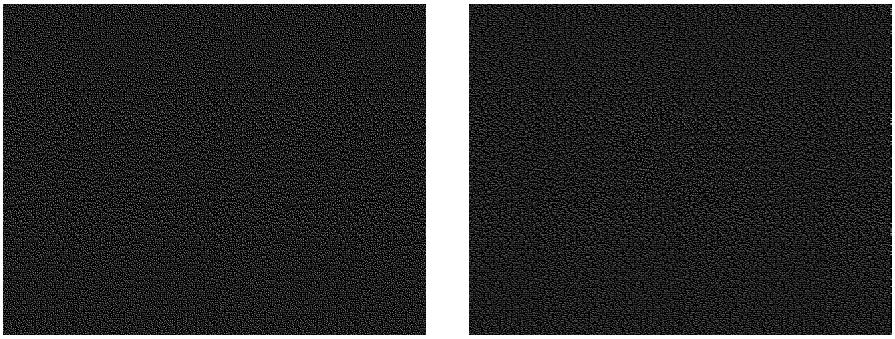
$$D = \frac{Z_k}{f} d \tag{3}$$

Substituting 3 into 1, we get

$$\frac{Z_0 d}{fb} = \frac{Z_0 - Z_k}{Z_0} \tag{4}$$

$$\frac{Z_0 d}{fb} = 1 - \frac{Z_k}{Z_0} \tag{5}$$

$$\therefore Z_k = \frac{Z_0}{1 + \frac{Z_0}{fb} d} \tag{6}$$



(a) Pattern Image (b) IR Image

Figure 2: Input Images.

2. Describe your implementation with key code sentences.

There are two main parts to generate the depth data from Kinect. They are:

(a) **Disparity Estimation**: The disparity between the IR image as in Fig. 2(b) and the pattern image as in Fig. 2(a) is estimated upto $\frac{1}{8}^{th}$ of a pixel. Hence, it has a sub pixel accuracy of $\frac{1}{8}$. To obtain this accuracy, the pattern image is shifted to the left by one pixel and 7 weighted images of a combination of the original pattern and shifted pattern images are generated. The code below shows how this is done.

```
for (int i=0; i<h; i++){
for (int j=0; j<w-1; j++){
patternShift.at<unsigned char>(i,j)=pattern.at<unsigned char>(i,j+1);
}
}

int j=w-1;
for(int i=0; i<h; i++){
patternShift.at<unsigned char>(i,j)=0;
}


for(int i=0; i<8; i++){
patternList[i] = ((S-i) * pattern + i * patternShift)/S;
}
```

Once the pattern images are generated, the disparity of the pixels is calculated using the following procedure:

i. Initialize a 9×9 window in the IR image.

ii. Initialize a 9×9 window in the pattern image along the corresponding row in the pattern image such that the difference between the column number of the pixel in the IR image and the pixel in pattern image is ∈ [-7, 80].

iii. Compute the Sum of the Absolute Differences (SAD) between corresponding pixels of the two windows. If the SAD is the minimum computed thus far, record the displacement along the row. Also not the pattern number of the image.

iv. Move the window in the pattern image such that $d \in$ [-7,80].

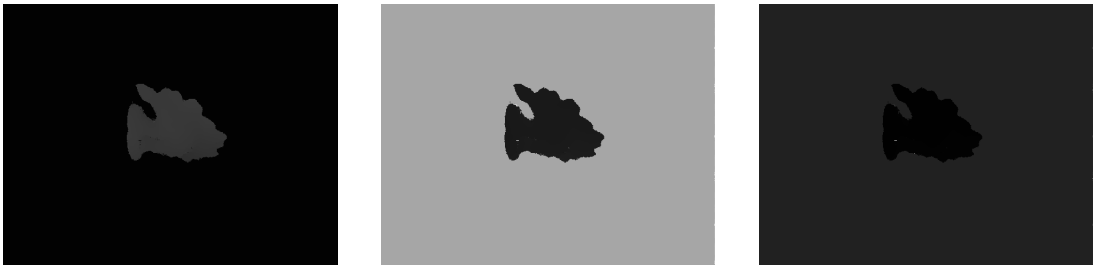
v. Repeat the procedure for all 8 pattern images in patternList.

vi. On exiting this loop, the value of d that is recorded contains the disparity for the center pixel of the window in the IR image.

vii. The above steps are repeated for all the pixels in the image to generate a disparity map. The code snippet below shows this. The variables start and end are explained in the note.

```
// For each subpixel value
for (int sub = 0; sub < S; sub++){

for ( int k=start; k<=end; k++){
```



(a) Disparity Image (b) Depth Image (c) Normalized Depth Image

Figure 3: Results.

```
float SAD=0.0;
for (int p=-margin; p<=margin; p++){
for (int q=-margin; q<=margin; q++){
SAD=SAD+abs(ir.at<unsigned char>(i+p, j+q)-patternList[sub].at<unsigned char>(i+p,k+q));
}
}

if (SAD<=minDiff){
minDiff=SAD;
bestDisparityInt=k-j;
bestDisparityFrac=sub;
}
}
}
disparity.at<float>(i-margin,j-margin) = bestDisparityInt + bestDisparityFrac/S;
}
}
```

Points to note: The value of d is clipped at the margins. This is done by using the following code statements:

```
int start = ((j+min)>=margin) ? j+min : margin;
int end = ((j+max)>w-margin-1) ? w-margin-1: (j+max);
```

(b) **Depth Estimation**: The depth is estimated using Eq. 6 and is as shown below:

```
for(int i=0; i<disparity.rows; i++){
for(int j=0; j<disparity.cols; j++){
depth.at<float>(i,j)=z0/(1+z0*disparity.at<float>(i,j)/(f*b));
}
}
```

This depth matrix is finally normalized and displayed as an image as shown in Figs. 3(a), 3(b) and 3(c).

3. What is the range of the computed depth value?

The depth values are in the range of 189.34 to 11636.36 mm. The Fig. 4 shows this.

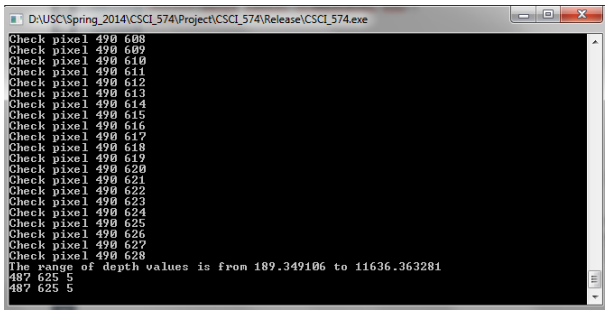


Figure 4: Snapshot of the Command Window.

4. In fact, the implemented algorithm is impractical. What are its weaknesses? Do you have any suggestions to improve the program?

   Some weaknesses and suggestions to improve the algorithm are as suggested below:

   (a) The window approach unnecessarily calculates the Sum of Absolute Differences (SAD) values at each and every pixel of the pattern shifted image. This can be avoided by calculating the SAD value of the current block if and only if the SAD of the previous block was less than the global minimum.

   (b) Further, instead of searching the entire disparity range in all the 8 images, we can search the entire disparity range in the original image and then look for the lowest SAD value corresponding to the same pixel (point of minimum in the original pattern image) in the remaining 7 images.

   (c) Since the exact speckle pattern is unknown, the disparity is searched in the range of [-7, 80]. But in reality, this speckle pattern may be known and hence we can limit the range of the search for the minimum value of Sum of Absolute Differences.

CSCI 574 Computer Vision Project 2

Divya Ramesh - dramesh@usc.edu

Description of the Implementation

1. Prepare Reference 3D Coordinates

This is done through the functions -

```
void calcInnerCorner(Size boardSize, float squareSize, vector<Point3f>& inCorners3D, Mat
&matInCorners3D)
{
	// TODO I.1.a: Compute 3-D coordinates of the checker inner corners in {B}
	float p, q;
	p=-(boardSize.height-1)/2.0;
	for (int i=0; i<boardSize.height; i++) {
		q=-(boardSize.width-1)/2.0;
		for (int j=0; j<boardSize.width; j++) {
			inCorners3D.push_back(cv::Point3f(q*squareSize, p*squareSize, 0.0f));
			q++;
		}
		p++;
	}

	convertPointsToHomogeneous(inCorners3D, matInCorners3D);
	return;
}
```

Here, the inner corners are calculated assuming the origin to be the center of the board, and with the square sizes as reference.

```
void calcOuterCorner(Rect_<float> boardRegion, vector<Point3f>& boardCorners3D, Mat
&matBoardCorners3D)
{
	// TODO I.1.b: Compute 3-D coordinates of the board 4 outer corners in {B}

		boardCorners3D.push_back(cv::Point3f(-boardRegion.width/2.0, -boardRegion.height/
2.0, 0.0f));
		boardCorners3D.push_back(cv::Point3f( boardRegion.width/2.0, -boardRegion.height/
2.0, 0.0f));
		boardCorners3D.push_back(cv::Point3f(-boardRegion.width/2.0, boardRegion.height/
2.0, 0.0f));
		boardCorners3D.push_back(cv::Point3f( boardRegion.width/2.0, boardRegion.height/
2.0, 0.0f));

		convertPointsToHomogeneous(boardCorners3D, matBoardCorners3D);
}
```

2. Collect Data to Calibrate

1. **Extract the pattern inner corners**

Here we use the findChessboardCorners function of openCV to locate the inner corners of the chessboard. The chessboard corners are detected from top left corner to bottom right corner. Sometimes, the chessboard corners may also be detected from right bottom to top left. In this case, care is taken to ensure that they are stored in the right order, although it does not matter much.

findChessboardCorners(extMat, boardSize, extPointBuf);

findChessboardCorners(rgbMat, boardSize, rgbPointBuf);

This is done for both the RGB and external image and the circles are then plotted.

2. **Estimate the board pose**

We estimate the pose of the board using the SolvePnPRansac() function. From this function, we get the values of the rotation and translation matrices which are tVec and rVec.

3. **Detection of Board Corners**

The board corners are detected by segmenting the board. For the segmentation, these are the steps that I carried out:

a. **Thresholding:** I first segment the depth image using a fixed form of thresholding, in such a way that the foreground is mostly the chessboard region, while everything else falls into the background. This thresholding may not be perfect and hence a few more steps are carried out.

b. **Distance Transform and normalization:** The distance transform does a skeletonizing like function, where it tries to estimate the distance of the pixels in the foreground from the background object. this results in a white horizontal line at the center of the chessboard while the corners are towards gray. This image is thresholded again to remove the wrongly classified background and foreground pixels from the first iteration.

c. **Watershed Segmentation and Markers:** A marker image is created in the areas of the foreground and then passed to the watershed segmentation function. This watershed segmentation fills the areas of the foreground with a color and the background with another color thereby segmenting it. The resultant image is then passed on to canny for edge detection.
It is a general notion that watershed segmentation yields a better result. Quite contrary to this fact, the resultant image of the thresholding when passed on to canny gave the best results, and hence ultimately the watershed segmentation step was skipped.

/ Perform the distance transform algorithm
cv::Mat dist;

```
cv::distanceTransform(segmentedDepthMat, dist, CV_DIST_L2, 3);

// Normalize the distance image for range = {0.0, 1.0}
// so we can visualize and threshold it
cv::normalize(dist, dist, 0, 1., cv::NORM_MINMAX);
cv::imshow("dist", dist);
cv::waitKey( 20 );

// Threshold to obtain the peaks
// This will be the markers for the foreground objects
cv::threshold(dist, dist, .5, 1., CV_THRESH_BINARY);
cv::imshow("dist2", dist);
cv::waitKey( 20 );

// Create the CV_8U version of the distance image
// It is needed for cv::findContours()
cv::Mat dist_8u;
dist.convertTo(dist_8u, CV_8U);

// Find total markers
std::vector<std::vector<cv::Point> > contours;
cv::findContours(dist_8u, contours, CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE);
int ncomp = contours.size();

// Create the marker image for the watershed algorithm
cv::Mat markers = cv::Mat::zeros(dist.size(), CV_32SC1);

// Draw the foreground markers
for (int i = 0; i < ncomp; i++)
        cv::drawContours(markers, contours, i, cv::Scalar::all(i+1), -1);

// Draw the background marker
cv::circle(markers, cv::Point(5,5), 3, CV_RGB(255,255,255), -1);
cv::imshow("markers", markers*10000);
cv::waitKey(20);

// Perform the watershed algorithm
cv::watershed(depthMat1, markers);
```

d. Canny and Hough Lines

The output of the canny edge detector is given to houghlinesP where in the lines in the image are detected. Unwanted lines are filtered out by taking a reference of the four inner corners which lie along the edges of the board. If in a frame, all four edges are detected, it is passed to the findintersections function, and then the corners detected are sorted from top left to right bottom. The corresponding depth values are also stored.

```
cv::Point2f computeIntersect(cv::Vec4i a, cv::Vec4i b)
{
   int x1 = a[0], y1 = a[1], x2 = a[2], y2 = a[3];
   int x3 = b[0], y3 = b[1], x4 = b[2], y4 = b[3];

   if (float d = ((float)(x1-x2) * (y3-y4)) - ((y1-y2) * (x3-x4)))
   {
      cv::Point2f pt;
```

```
        pt.x = ((x1*y2 - y1*x2) * (x3-x4) - (x1-x2) * (x3*y4 - y3*x4)) / d;
        pt.y = ((x1*y2 - y1*x2) * (y3-y4) - (y1-y2) * (x3*y4 - y3*x4)) / d;
        return pt;
    }
    else
        return cv::Point2f(-1, -1);
}


void sortCorners(vector<Point2f>& board2DCorners, Point2f center)
{
  vector<Point2f> top, bot;

    for (int i = 0; i < board2DCorners.size(); i++)
    {
        if (board2DCorners[i].y < center.y)
            top.push_back(board2DCorners[i]);
        else
            bot.push_back(board2DCorners[i]);
    }

    cv::Point2f tl = top[0].x > top[1].x ? top[1] : top[0];
    cv::Point2f tr = top[0].x > top[1].x ? top[0] : top[1];
    cv::Point2f bl = bot[0].x > bot[1].x ? bot[1] : bot[0];
    cv::Point2f br = bot[0].x > bot[1].x ? bot[0] : bot[1];

    board2DCorners.clear();
    board2DCorners.push_back(tl);
    board2DCorners.push_back(tr);
    board2DCorners.push_back(bl);
    board2DCorners.push_back(br);
}
```

3. Calibrate Sensors

We calibrate the depth sensor using the equations 4, 5, 6 of the problem statement. To do this, the rvec obtained during the pose estimation phase is first converted to a matrix using the rodriques function. Then, corner points in 3D are estimated. Using these 3D corner points for all the good frames, and a method of least square estimation, we estimate the value of t1.

4. Paint Depth Images

Finally, we warp color images into the depth ones. The algorithm is described as below:

· For every point in the depth image, use equation (4) to compute its 3-D coordinates in {D}.

· Compute its coordinate in {C} using the transformation: T=[I | -t]

· Project it to the color image. Get pixel color and paint on the depth image.

```
projectPoints(_3DPoints, rrvec, -t1, cameraMatrix[RGB_SENSOR], distCoeffs, projectedPoints);

for(int i=0; i<projectedPoints.size(); i++){
```

```
                    if (projectedPoints[i].x>=0 && projectedPoints[i].x<rgbMat0.cols &&
projectedPoints[i].y>=0 && projectedPoints[i].y<rgbMat0.rows){
outTest.at<Vec3b>(pSet[i],qSet[i])=avSubPixelValue8U3(Point2f((projectedPoints[i].x),
(projectedPoints[i].y)),rgbMat0);
        pointCount++;
         }
}
```

External Camera Calibration

1. Calibrate the external camera

In this part, we calibrate the external camera. To simplify the problem, we assume that the

camera is well-manufactured (no distortion, the principal points are centralized).

1.1. Extract the pattern inner corners

First, for each frame, detect the checker inner corners in the external image using

function **findChessboardCorners()** of OpenCV.

1.2. Compute the external camera intrinsic

After collecting necessary information, do calibration using function **calibrateCamera()**
of OpenCV.
In this step, we have to ensure that the object points are the same size as the image points. To do this, we make the image points a vector of vectors and then pass it to the cameraCalibrate function of OpenCV.

```
bool runCalibration(vector<vector<Point3f>> newinCorners3D, Size imageSize, Mat& cameraMatrix,
Mat& distCoeffs,
vector<vector<Point2f> > incorner2DPoints, vector<Mat>& rvecs, vector<Mat>& tvecs)
{
        calibrateCamera(newinCorners3D, incorner2DPoints, imageSize, cameraMatrix, distCoeffs,
rvecs, tvecs, CV_CALIB_FIX_PRINCIPAL_POINT + CV_CALIB_ZERO_TANGENT_DIST +
CV_CALIB_FIX_K1 + CV_CALIB_FIX_K2 + CV_CALIB_FIX_K3 + CV_CALIB_FIX_K4 +
CV_CALIB_FIX_K5 + CV_CALIB_FIX_K6);

return true;
}
```

.Proper flags are used to ensure that image is not warped in the ends.

1.3. Collect correspondences

Push all extracted 2-D corners into vector ***point2D***.

Compute from equation (8) and push them into vector ***point3D***.

```
for(int j=0; j<20; j++){

tempP3D=matR*Mat(newinCorners3D[i][j])+Mat(tvecs[RGB_SENSOR][i])+t1;

point3D.push_back(Point3f(tempP3D));

}
```

The SolvePnPRansac function is called to estimate the R2 and t2 parameters.

Paint Depth Images

In order to paint the depth images, we make use of a z-buffer. First the 3Dpoints are projected to the external image. These coordinates are divided by 4, and if the depth of the corresponding coordinates is less than that stored in the rough depth image, we store this depth, else it is left as it is.
Next, the projected points' depth is checked with the z-buffer and if it is within 120% of that of the z-buffer, the values are retained, and the image is painted from the external rgb image, else it is left as it is.

```
for(int ii=0; ii<pSet.size(); ii++){

zrows=floor(projectedPoints2[ii].y/4);

zcols=floor(projectedPoints2[ii].x/4);

if(zrows>0 && zrows<extDepth.rows && zcols >0 && zcols<extDepth.cols){

if (extDepth.at<float>(zrows, zcols)>_3DPoints[ii].z){

extDepth.at<float>(zrows, zcols)=_3DPoints[ii].z;

}

}

}

for(int ii=0; ii<pSet.size(); ii++){

zrows=floor(projectedPoints2[ii].y/4);

zcols=floor(projectedPoints2[ii].x/4);

if(zrows>0 && zrows<extDepth.rows && zcols >0 && zcols<extDepth.cols){



if (1.2*(extDepth.at<float>(zrows, zcols))>_3DPoints[ii].z){
```

```
outTest2.at<Vec3b>(pSet[ii],
qSet[ii])=avSubPixelValue8U3(Point2f(floor(projectedPoints2[ii].x),floor(projectedPoints2[ii].y)),extMat
0);

}

}

}
```

Creating Own Dataset

The above steps are performed for the own data set that is generated and the calibration is completed.

Q2. The Values of t1, t2 and r2

t1 values are -33.9171, 0.454256, -29.2032
t2 values are -90.1727, 6.39608, 7.81709
r2 values are 0.0803265, 0.0212301, -0.0244855

Q3. If Fc and Fd are unknown, propose a solution

If Fc and Fd values are unknown, we can use the calibrateCamera() function of openCV and perform calibration, as is the case in most real-world scenarios.

Extra Credit:

If the two cameras, do not have the same orientation, we can take more correspondences between the images from the two frames and perform a least squares approximation. For example, in this case we took a total of 10 frames and 4 boardcorners. We may take more number of frames and hence obtain more correspondences and hence solve the problem.