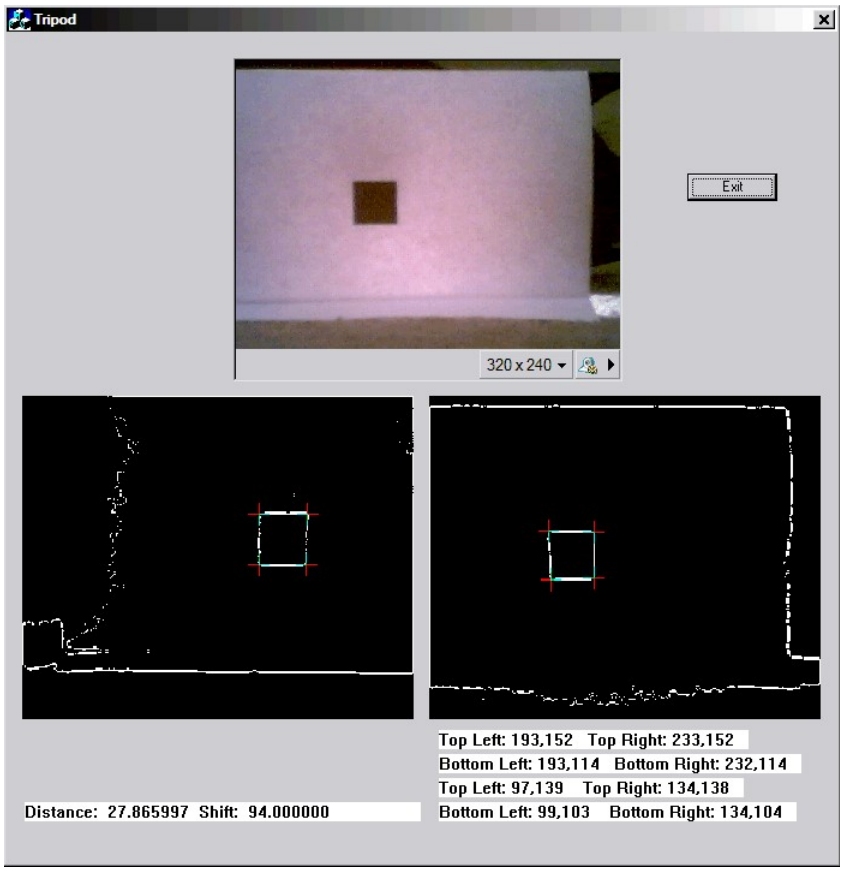
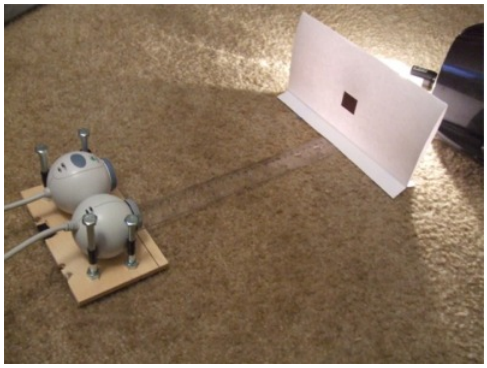
Distance Tutorial

Author: Noah Kuntz (2006)
Contact: nk752@drexel.edu

Keywords: Distance, TRIPOD, Binocular Vision, Corner Finding



This image shows the result of this simple distance finding program. Using two webcams the distance from the cameras to a black square can be found. With code to recognize similar vertical lines in an image, the distance to any object could be found. For simplicity's sake this instead finds 90 degree corners on a square that is sharply contrasted from its surroundings. The first step in writing this program was adapting TRIPOD to use two cameras simultaneously. The focus of this tutorial is on the image processing nessicary to find distance with binocular vision. As far as the interface modifications you should download the entire project and examine it for more details of that change.



The camera setup I used for finding distance.

The image processing in this tutorial can be divided into four steps. First, the image is converted into edges using the Canny edge detection algorythm. Next, masks are loaded that represent ideal corners. Third, those masks are used to find square corners. Finally, the distance is found based on the horizontal shift of the square in the two camera's views.

This tutorial assumes the reader:
(1) Knows how to use TRIPOD
(2) Has a basic knowledge of Visual C++

(3) Has read Bill Green's Canny Edge Detection Tutorial

The rest of the tutorial is presented as follows:

Step 1: Apply Canny Edge Detection

First the image must be converted into edges. Refer to my canny edge detection tutorial for an explanation of that process.

Step 2: Load Masks from Bitmaps

Before corners can be found you need to load the masks to compare them to. Using some heavily modified code from one of Bill Green's tutorials, bitmaps are converted to mask arrays with the appropriate weights. These two functions are called 4 times on initilization, they simply load each of the four bitmaps one at a time and cycle through the pixel bytes. Each pixel's blue value is converted into the mask weight it represents and stored in the right place in the mask array.

To be compiled with Microsoft Visual C++
Note: download tripodDlg.cpp rather than cutting and pasting from below.

```
/*------STRUCTURES--------*/
typedef struct {int rows; int cols; unsigned char* data;} sImage;

void CTripodDlg::getMaskBmp(int fileInt)
{
	FILE *bmpInput;
	sImage originalImage;
	unsigned char someChar;
	unsigned char *pChar;
	long fileSize;
	int nColors;
	int r, c;
	int redValue, greenValue, blueValue;
	/*---------INITIALIZE POINTER----------*/
	someChar = '0';
	pChar = &someChar;

	/*----DECLARE INPUT AND OUTPUT FILES----*/
	switch (fileInt){
	case 0:
		bmpInput = fopen("corner1Mask.bmp", "rb");
	break;
	case 1:
		bmpInput = fopen("corner2Mask.bmp", "rb");
	break;
	case 2:
		bmpInput = fopen("corner3Mask.bmp", "rb");
	break;
	case 3:
		bmpInput = fopen("corner4Mask.bmp", "rb");
	break;
	}

	fseek(bmpInput, 0L, SEEK_END);

	/*-----GET BMP DATA-----*/
	originalImage.cols = (int)getImageInfo(bmpInput, 18, 4);
	originalImage.rows = (int)getImageInfo(bmpInput, 22, 4);
	fileSize = getImageInfo(bmpInput, 2, 4);
	nColors = getImageInfo(bmpInput, 46, 4);

	/*----FOR 24-BIT BMP, THERE IS NO COLOR TABLE-----*/
	fseek(bmpInput, 54, SEEK_SET);

	/*-----------READ RASTER DATA-----------*/
	for(r=originalImage.rows-1; r>=0; r--) {
		for(c=0; c < originalImage.cols; c++) {
			/*----READ FIRST BYTE TO GET BLUE VALUE-----*/
			fread(pChar, sizeof(char), 1, bmpInput);
			blueValue = *pChar;

			/*-----READ NEXT BYTE TO GET GREEN VALUE-----*/
			fread(pChar, sizeof(char), 1, bmpInput);
			greenValue = *pChar;

			/*-----READ NEXT BYTE TO GET RED VALUE-----*/
			fread(pChar, sizeof(char), 1, bmpInput);
			redValue = *pChar;

			if(blueValue == 255)
				cornerMask[fileInt][r][c] = 24;
			else if(blueValue == 0)
				cornerMask[fileInt][r][c] = -4;
			else if(blueValue == 128)
				cornerMask[fileInt][r][c] = 4;
			else if(blueValue == 64)
				cornerMask[fileInt][r][c] = 1;
			else
				cornerMask[fileInt][r][c] = 0;
```

```
                }
                fread(pChar, sizeof(char), 1, bmpInput);
        }
        fclose(bmpInput);
}

/*----GET IMAGE INFO SUBPROGRAM------*/

long CTripodDlg::getImageInfo(FILE* inputFile, long offset, int numberOfChars)
{
        unsigned char           *ptrC;
        long                    value=0L;
        int                     i;
        unsigned char           dummy;

        dummy = '0';
        ptrC = &dummy;

        fseek(inputFile, offset, SEEK_SET);

        for(i=1; i<=numberOfChars; i++)
        {
                fread(ptrC, sizeof(char), 1, inputFile);
                /* calculate value based on adding bytes */
                value = (long)(value + (*ptrC)*(pow(256, (i-1))));
        }

        return(value);
}
```

Step 3: Find Corners

Corners are found by comparing the edge pixels with bitmaps of ideal corners.

Mask for top left corner

For each possible corner pixel a 17 x 17 neighborhood is examined. The point of the corner is matched with the point being considered. The brightness value of each pixel around the possible corner is multiplied by the weight of the mask at that point. For black pixels on the mask the weight is -4, any white edge pixels in this area reduce the corner strength value for that corner. For dark gray pixels on the mask the weight is 1, and lighter gray pixels have a weight of 4. Any edge pixels in this area have a moderate effect on increasing the corner strength value. White mask pixels have a weight of 24, this is where the edge pixels really need to be. After the the corner strength is calculated for each of the 4 corners for each possible pixel, the highest value is marked as the max and if its over the threshold and greater than the current best value for that corner then this is set as being one of the 4 corners. After the whole image is processed the best of each corner has been found and marked.

To be compiled with Microsoft Visual C++

Note: download tripodDlg.cpp rather than cutting and pasting from below.

```
        // X , Y , Corner Strength
        cornersLeft[0][0] = -1;
        cornersLeft[1][0] = -1;
        cornersLeft[2][0] = -1;
        cornersLeft[3][0] = -1;
        cornersLeft[0][1] = -1;
        cornersLeft[1][1] = -1;
        cornersLeft[2][1] = -1;
        cornersLeft[3][1] = -1;
        cornersLeft[0][2] = 0;
        cornersLeft[1][2] = 0;
        cornersLeft[2][2] = 0;
        cornersLeft[3][2] = 0;

        // Find Corners
        for (row = halfMask + 21; row < H - halfMask - 21; row++) {
                for (col = halfMask + 21; col < W - halfMask - 21; col++) {
                        for (count = 0; count < 8; count++) {
                                cornerMaskVal[count] = 0;
                        }
                                for (rowOffset=-halfMask; rowOffset<=halfMask; rowOffset++) {
                                        for (colOffset=-halfMask; colOffset<=halfMask; colOffset++) {
                                                rowTotal = row + rowOffset;
                                                colTotal = col + colOffset;
                                                iOffset = (unsigned long)(rowTotal*3*W + colTotal*3);
                                                curPixel = *(m_destinationBmp + iOffset);
                                                for (count = 0; count < 8; count++) {
                                                        cornerMaskVal[count] += (cornerMask[count][rowOffset + halfMask][colOffset
                                                }
                                        }
                                }

                        orientation = 10;
                        max = 0;

                        for (count = 0; count < 4; count++) {
                                if((cornerMaskVal[count] > max) && (cornerMaskVal[count] > 70000)) {
                                        max = cornerMaskVal[count];
                                        orientation = count;
                                }
```

```
                        }

                        // Limit to square corners
                        if      ((orientation != 10) && (max > cornersLeft[orientation][2])) {
                                cornersLeft[orientation][0] = col;
                                cornersLeft[orientation][1] = row;
                                cornersLeft[orientation][2] = max;
                        }

                }
        }

        // Mark Corners
        for (count = 0; count < 4; count++) {
                if(cornersLeft[count][2] != 0) {
                for (rowOffset=-halfMask; rowOffset<=halfMask; rowOffset++) {
                        rowTotal = cornersLeft[count][1] + rowOffset;
                        iOffset = (unsigned long)(rowTotal*3*W + cornersLeft[count][0]*3);
                        if(cornerMask[count][rowOffset + halfMask][8] == center) {
                                *(m_destinationBmp + iOffset + 1) = 255;
                                *(m_destinationBmp + iOffset + 2) = 0;
                        } else {
                                *(m_destinationBmp + iOffset + 1) = 0;
                                *(m_destinationBmp + iOffset + 2) = 255;
                        }
                }

                for (colOffset=-halfMask; colOffset<=halfMask; colOffset++) {
                        colTotal = cornersLeft[count][0] + colOffset;
                        iOffset = (unsigned long)(cornersLeft[count][1]*3*W + colTotal*3);
                        if(cornerMask[count][8][colOffset + halfMask] == center) {
                                *(m_destinationBmp + iOffset + 1) = 255;
                                *(m_destinationBmp + iOffset + 2) = 0;
                        } else {
                                *(m_destinationBmp + iOffset + 1) = 0;
                                *(m_destinationBmp + iOffset + 2) = 255;
                        }
                }
                }
        }
```

Step 4: Calculate Distance

To calculate the distance there must be good corners for both the top and bottom corner on either the right or left side of the square in both images. If statements check to see if these good corners exist and if each set are close enough together in the horizontal direction to be part of the same square. If they are then the shift between either the right or left sides is calculated. Then the distance can be calculated using a formula based on the focal length of the lenses used, the distance between the cameras, and the shift: distance = (914.0 * 2.9) / shift. The distance formula can also be synthesized, sometimes more accurately, by finding the shift at various differences and applying a curve fit. With my setup, cameras, and a 2.9 cm shift the best formula was distance = 2946.5 * pow(shift, -1.0259).

To be compiled with Microsoft Visual C++

Note: download tripodDlg.cpp rather than cutting and pasting from below.

```
if ( (cornersRight[0][2] != 0) && (cornersRight[2][2] != 0) && (cornersLeft[0][2] != 0) && (cornersLeft[2][2] != 0) ) {
        if ( ( abs(cornersRight[0][0] - cornersRight[2][0]) < 6 ) && ( abs(cornersLeft[0][0] - cornersLeft[2][0]) < 6 ) ) {
                shift = abs((cornersRight[0][0] + cornersRight[2][0])/2 - (cornersLeft[0][0] + cornersLeft[2][0])/2);
                //distance = 2946.5 * pow(shift, -1.0259);
                distance = (914.0 * 2.9) / shift;
        }
}
else if ( (cornersRight[1][2] != 0) && (cornersRight[3][2] != 0) && (cornersLeft[1][2] != 0) && (cornersLeft[3][2] != 0) ) {
        if ( ( abs(cornersRight[1][0] - cornersRight[3][0]) < 6 ) && ( abs(cornersLeft[1][0] - cornersLeft[3][0]) < 6 ) ) {
                shift = abs((cornersRight[1][0] + cornersRight[3][0])/2 - (cornersLeft[1][0] + cornersLeft[3][0]) / 2);
                //distance = 2946.5 * pow(shift, -1.0259);
                distance = (914.0 * 2.9) / shift;
        }
}
```

Final Words

This tutorial's objective was to show how simple distance finding can be implemented with a modified version of TRIPOD. If instead of recognizing corners you matched vertical lines between images, the distance to more complex objects could be found. This is the first step towards understanding a scene in three dimensions.

Click here to email me.
Click here to return to my Tutorials page.