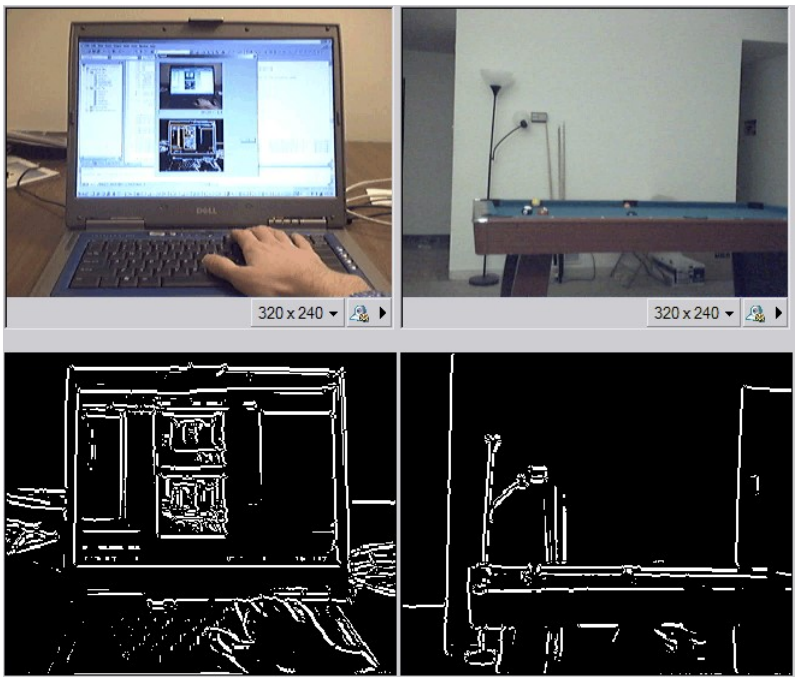
Canny Tutorial

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

Keywords: Canny, TRIPOD, Edge Finding



These photos depict the results of this implementation of the Canny edge detection algorithm. Edge detection is important in image processing programs because it allows object separation and shape detection. This tutorial will teach you how to implement the Canny edge detection algorithm using the TRIPOD framework. It is based on the theory outlined in Bill Green's Canny tutorial: Canny Edge Detection Tutorial The reader should read that tutorial first to fully understand what this code is doing. The basic steps of Canny edge detection are as follows: First, a Gaussian blur is applied to the image to reduce noise. Thresholding can also be used if the objects of interest are significantly contrasted from the background and detailed textures are irrelevant. Next, the edge direction and gradient strength of each pixel in the image are found using Sobel masks. Then edges are traced using that information. Finally, non-maximum edges are suppressed by finding parallel edges and eliminating those with weaker gradient strengths.

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 a Gaussian blur

First necessary variables are declared and some are initialized. Then a Gaussian blur is applied. To do this a 5x5 mask is passed over the image. Each pixel is redefined as the sum of the pixel values in its 5x5 neighborhood times the corresponding Gaussian weight, divided by the total weight of the whole mask.

$$\frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Gaussian mask

To be compiled with Microsoft Visual C++

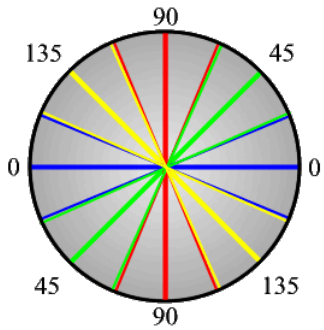
Note: download tripodDlg.cpp rather than cutting and pasting from below.

```
void CTripodDlg::doMyImageProcessing(LPBITMAPINFOHEADER lpThisBitmapInfoHeader)
{
        // doMyImageProcessing:  This is where you'd write your own image processing code
        // Task: Read a pixel's grayscale value and process accordingly

        unsigned int W, H;              // Width and Height of current frame [pixels]
        unsigned int row, col;          // Pixel's row and col positions
        unsigned long i;                // Dummy variable for row-column vector
        int upperThreshold = 60;        // Gradient strength nessicary to start edge
        int lowerThreshold = 30;        // Minimum gradient strength to continue edge
        unsigned long iOffset;          // Variable to offset row-column vector during sobel mask
        int rowOffset;                  // Row offset from the current pixel
        int colOffset;                  // Col offset from the current pixel
        int rowTotal = 0;               // Row position of offset pixel
        int colTotal = 0;               // Col position of offset pixel
        int Gx;                         // Sum of Sobel mask products values in the x direction
        int Gy;                         // Sum of Sobel mask products values in the y direction
        float thisAngle;                // Gradient direction based on Gx and Gy
        int newAngle;                   // Approximation of the gradient direction
        bool edgeEnd;                   // Stores whether or not the edge is at the edge of the possible image
        int GxMask[3][3];               // Sobel mask in the x direction
        int GyMask[3][3];               // Sobel mask in the y direction
        int newPixel;                   // Sum pixel values for gaussian
        int gaussianMask[5][5];         // Gaussian mask

        W = lpThisBitmapInfoHeader->biWidth;  // biWidth: number of columns
        H = lpThisBitmapInfoHeader->biHeight; // biHeight: number of rows

        for (row = 0; row < H; row++) {
                for (col = 0; col < W; col++) {
                        edgeDir[row][col] = 0;
                }
        }

        /* Declare Sobel masks */
        GxMask[0][0] = -1; GxMask[0][1] = 0; GxMask[0][2] = 1;
        GxMask[1][0] = -2; GxMask[1][1] = 0; GxMask[1][2] = 2;
        GxMask[2][0] = -1; GxMask[2][1] = 0; GxMask[2][2] = 1;

        GyMask[0][0] =  1; GyMask[0][1] =  2; GyMask[0][2] =  1;
        GyMask[1][0] =  0; GyMask[1][1] =  0; GyMask[1][2] =  0;
        GyMask[2][0] = -1; GyMask[2][1] = -2; GyMask[2][2] = -1;

        /* Declare Gaussian mask */
        gaussianMask[0][0] = 2;         gaussianMask[0][1] = 4;         gaussianMask[0][2] = 5;         gaussianMask[0][3] = 4;         gau
        gaussianMask[1][0] = 4;         gaussianMask[1][1] = 9;         gaussianMask[1][2] = 12;        gaussianMask[1][3] = 9;         gau
        gaussianMask[2][0] = 5;         gaussianMask[2][1] = 12;        gaussianMask[2][2] = 15;        gaussianMask[2][3] = 12;        gau
        gaussianMask[3][0] = 4;         gaussianMask[3][1] = 9;         gaussianMask[3][2] = 12;        gaussianMask[3][3] = 9;         gau
        gaussianMask[4][0] = 2;         gaussianMask[4][1] = 4;         gaussianMask[4][2] = 5;         gaussianMask[4][3] = 4;         gau


        /* Gaussian Blur */
        for (row = 2; row < H-2; row++) {
                for (col = 2; col < W-2; col++) {
                        newPixel = 0;
                        for (rowOffset=-2; rowOffset<=2; rowOffset++) {
                                for (colOffset=-2; colOffset<=2; colOffset++) {
                                        rowTotal = row + rowOffset;
                                        colTotal = col + colOffset;
                                        iOffset = (unsigned long)(rowTotal*3*W + colTotal*3);
                                        newPixel += (*(m_destinationBmp + iOffset)) * gaussianMask[2 + rowOffset][2 + colOffset];
                                }
                        }
                        i = (unsigned long)(row*3*W + col*3);
                        *(m_destinationBmp + i) = newPixel / 159;
                }
        }
```

Step 2: Find edge gradient strength and direction

The next step is to use Sobel masks to find the edge gradient strength and direction for each pixel. First the Sobel masks are applied to the 3x3 pixel neighborhood of the current pixel, in both the x and y directions. Then the sum of each mask value times the corresponding pixel is computed as the Gx and Gy values, respectively. The square root of Gx squared plus Gy squared equals the edge strength. The inverse tangent of Gx / Gy yields the edge direction. The edge direction is then approximated to one of four possible values that make up the possible directions an edge could be in an image made up of a square pixel grid. This edge direction is then stored in the array edgeDir[row][col] and the gradient strength is stored in the array gradient[row][col].

Sobel Masks (Image courtesy of Bill Green)



Any edge angle within 11.25 degrees of one of the possible angles is changed to that value.

To be compiled with Microsoft Visual C++

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

```
*NOTE*  These two lines must be added to the file tripodDlg.h, the rest continues
in tripodDlg.cpp's "doMyImageProcessing" function
        int edgeDir[240][320];                  // Stores the edge direction of each pixel
        float gradient[240][320];               // Stores the gradient strength of each pixel

        void CTripodDlg::findEdge(int rowShift, int colShift, int row, int col, int dir, int lowerThreshold);
        void suppressNonMax(int rowShift, int colShift, int row, int col, int dir, int lowerThreshold);
*End NOTE*

/* Determine edge directions and gradient strengths */
for (row = 1; row < H-1; row++) {
        for (col = 1; col < W-1; col++) {
                i = (unsigned long)(row*3*W + 3*col);
                Gx = 0;
                Gy = 0;
                /* Calculate the sum of the Sobel mask times the nine surrounding pixels in the x and y direction */
                for (rowOffset=-1; rowOffset<=1; rowOffset++) {
                        for (colOffset=-1; colOffset<=1; colOffset++) {
                                rowTotal = row + rowOffset;
                                colTotal = col + colOffset;
                                iOffset = (unsigned long)(rowTotal*3*W + colTotal*3);
                                Gx = Gx + (*(m_destinationBmp + iOffset) * GxMask[rowOffset + 1][colOffset + 1]);
                                Gy = Gy + (*(m_destinationBmp + iOffset) * GyMask[rowOffset + 1][colOffset + 1]);
                        }
                }

                gradient[row][col] = sqrt(pow(Gx,2.0) + pow(Gy,2.0));   // Calculate gradient strength
                thisAngle = (atan2(Gx,Gy)/3.14159) * 180.0;             // Calculate actual direction of edge

                /* Convert actual edge direction to approximate value */
                if ( ( (thisAngle < 22.5) && (thisAngle > -22.5) ) || (thisAngle > 157.5) || (thisAngle < -157.5) )
                        newAngle = 0;
                if ( ( (thisAngle > 22.5) && (thisAngle < 67.5) ) || ( (thisAngle < -112.5) && (thisAngle > -157.5) ) )
                        newAngle = 45;
                if ( ( (thisAngle > 67.5) && (thisAngle < 112.5) ) || ( (thisAngle < -67.5) && (thisAngle > -112.5) ) )
                        newAngle = 90;
                if ( ( (thisAngle > 112.5) && (thisAngle < 157.5) ) || ( (thisAngle < -22.5) && (thisAngle > -67.5) ) )
                        newAngle = 135;

                edgeDir[row][col] = newAngle;           // Store the approximate edge direction of each pixel in one array
        }
}
```

Step 3: Trace along the edges

The next step is to actually trace along the edges based on the previously calculated gradient strengths and edge directions. Each pixel is cycled through using two nested for loops. If the current pixel has a gradient strength greater than the defined upperThreshold, then a switch is executed. The switch is determined by the edge direction of the current pixel. It stores the row and column of the next possible pixel in that direction and then tests the edge direction and gradient strength of that pixel. If it has the same edge direction and a gradient strength greater than the lowerThreshold, that pixel is set to white and the next pixel along that edge is tested. In this manner any significantly sharp edge is detected and set to white while all other pixels are set to black.

To be compiled with Microsoft Visual C++

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

```
	/* Trace along all the edges in the image */
	for (row = 1; row < H - 1; row++) {
		for (col = 1; col < W - 1; col++) {
			edgeEnd = false;
			if (gradient[row][col] > upperThreshold) {		// Check to see if current pixel has a high enough gradient
				/* Switch based on current pixel's edge direction */
				switch (edgeDir[row][col]){
					case 0:
						findEdge(0, 1, row, col, 0, lowerThreshold);
						break;
					case 45:
						findEdge(1, 1, row, col, 45, lowerThreshold);
						break;
					case 90:
						findEdge(1, 0, row, col, 90, lowerThreshold);
						break;
					case 135:
						findEdge(1, -1, row, col, 135, lowerThreshold);
						break;
					default :
						i = (unsigned long)(row*3*W + 3*col);
						*(m_destinationBmp + i) =
						*(m_destinationBmp + i + 1) =
						*(m_destinationBmp + i + 2) = 0;
						break;
					}
				}
			else {
				i = (unsigned long)(row*3*W + 3*col);
					*(m_destinationBmp + i) =
					*(m_destinationBmp + i + 1) =
					*(m_destinationBmp + i + 2) = 0;
			}
		}
	}

void CTripodDlg::findEdge(int rowShift, int colShift, int row, int col, int dir, int lowerThreshold)
{
	int W = 320;
	int H = 240;
	int newRow;
	int newCol;
	unsigned long i;
	bool edgeEnd = false;

	/* Find the row and column values for the next possible pixel on the edge */
	if (colShift < 0) {
		if (col > 0)
			newCol = col + colShift;
		else
			edgeEnd = true;
	} else if (col < W - 1) {
		newCol = col + colShift;
	} else
		edgeEnd = true;		// If the next pixel would be off image, don't do the while loop
	if (rowShift < 0) {
		if (row > 0)
			newRow = row + rowShift;
		else
			edgeEnd = true;
	} else if (row < H - 1) {
		newRow = row + rowShift;
	} else
		edgeEnd = true;

	/* Determine edge directions and gradient strengths */
	while ( (edgeDir[newRow][newCol]==dir) && !edgeEnd && (gradient[newRow][newCol] > lowerThreshold) ) {
		/* Set the new pixel as white to show it is an edge */
		i = (unsigned long)(newRow*3*W + 3*newCol);
		*(m_destinationBmp + i) =
		*(m_destinationBmp + i + 1) =
		*(m_destinationBmp + i + 2) = 255;
		if (colShift < 0) {
			if (newCol > 0)
				newCol = newCol + colShift;
			else
				edgeEnd = true;
		} else if (newCol < W - 1) {
			newCol = newCol + colShift;
		} else
			edgeEnd = true;
		if (rowShift < 0) {
			if (newRow > 0)
				newRow = newRow + rowShift;
			else
				edgeEnd = true;
		} else if (newRow < H - 1) {
			newRow = newRow + rowShift;
		} else
			edgeEnd = true;
	}
}
```

Step 4: Suppress non-maximum edges

The last step is to find weak edges that are parallel to strong edges and eliminate them. This is accomplished by examining the pixels perpendicular to a particular edge pixel, and eliminating the non-maximum edges. The code used is very similar to the edge tracing code.

To be compiled with Microsoft Visual C++

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

```
        /* Non-maximum Suppression */
        for (row = 1; row < H - 1; row++) {
                for (col = 1; col < W - 1; col++) {
                        i = (unsigned long)(row*3*W + 3*col);
                        if (*(m_destinationBmp + i) == 255) {           // Check to see if current pixel is an edge
                                /* Switch based on current pixel's edge direction */
                                switch (edgeDir[row][col]) {
                                        case 0:
                                                suppressNonMax( 1, 0, row, col, 0, lowerThreshold);
                                                break;
                                        case 45:
                                                suppressNonMax( 1, -1, row, col, 45, lowerThreshold);
                                                break;
                                        case 90:
                                                suppressNonMax( 0, 1, row, col, 90, lowerThreshold);
                                                break;
                                        case 135:
                                                suppressNonMax( 1, 1, row, col, 135, lowerThreshold);
                                                break;
                                        default :
                                                break;
                                }
                        }
                }
        }
void CTripodDlg::suppressNonMax(int rowShift, int colShift, int row, int col, int dir, int lowerThreshold)
{
        int W = 320;
        int H = 240;
        int newRow = 0;
        int newCol = 0;
        unsigned long i;
        bool edgeEnd = false;
        float nonMax[320][3];           // Temporarily stores gradients and positions of pixels in parallel edges
        int pixelCount = 0;             // Stores the number of pixels in parallel edges
        int count;
        int max[3];                     // Maximum point in a wide edge

        if (colShift < 0) {
                if (col > 0)
                        newCol = col + colShift;
                else
                        edgeEnd = true;
        } else if (col < W - 1) {
                newCol = col + colShift;
        } else
                edgeEnd = true;         // If the next pixel would be off image, don't do the while loop
        if (rowShift < 0) {
                if (row > 0)
                        newRow = row + rowShift;
                else
                        edgeEnd = true;
        } else if (row < H - 1) {
                newRow = row + rowShift;
        } else
                edgeEnd = true;
        i = (unsigned long)(newRow*3*W + 3*newCol);
        /* Find non-maximum parallel edges tracing up */
        while ((edgeDir[newRow][newCol] == dir) && !edgeEnd && (*(m_destinationBmp + i) == 255)) {
                if (colShift < 0) {
                        if (newCol > 0)
                                newCol = newCol + colShift;
                        else
                                edgeEnd = true;
                } else if (newCol < W - 1) {
                        newCol = newCol + colShift;
                } else
                        edgeEnd = true;
                if (rowShift < 0) {
                        if (newRow > 0)
                                newRow = newRow + rowShift;
                        else
                                edgeEnd = true;
                } else if (newRow < H - 1) {
                        newRow = newRow + rowShift;
                } else
                        edgeEnd = true;
                nonMax[pixelCount][0] = newRow;
                nonMax[pixelCount][1] = newCol;
                nonMax[pixelCount][2] = gradient[newRow][newCol];
                pixelCount++;
                i = (unsigned long)(newRow*3*W + 3*newCol);
        }

        /* Find non-maximum parallel edges tracing down */
        edgeEnd = false;
        colShift *= -1;
        rowShift *= -1;
        if (colShift < 0) {
```

```
                if (col > 0)
                        newCol = col + colShift;
                else
                        edgeEnd = true;
        } else if (col < W - 1) {
                newCol = col + colShift;
        } else
                edgeEnd = true;
        if (rowShift < 0) {
                if (row > 0)
                        newRow = row + rowShift;
                else
                        edgeEnd = true;
        } else if (row < H - 1) {
                newRow = row + rowShift;
        } else
                edgeEnd = true;
        i = (unsigned long)(newRow*3*W + 3*newCol);
        while ((edgeDir[newRow][newCol] == dir) && !edgeEnd && (*(m_destinationBmp + i) == 255)) {
                if (colShift < 0) {
                        if (newCol > 0)
                                newCol = newCol + colShift;
                        else
                                edgeEnd = true;
                } else if (newCol < W - 1) {
                        newCol = newCol + colShift;
                } else
                        edgeEnd = true;
                if (rowShift < 0) {
                        if (newRow > 0)
                                newRow = newRow + rowShift;
                        else
                                edgeEnd = true;
                } else if (newRow < H - 1) {
                        newRow = newRow + rowShift;
                } else
                        edgeEnd = true;
                nonMax[pixelCount][0] = newRow;
                nonMax[pixelCount][1] = newCol;
                nonMax[pixelCount][2] = gradient[newRow][newCol];
                pixelCount++;
                i = (unsigned long)(newRow*3*W + 3*newCol);
        }

        /* Suppress non-maximum edges */
        max[0] = 0;
        max[1] = 0;
        max[2] = 0;
        for (count = 0; count < pixelCount; count++) {
                if (nonMax[count][2] > max[2]) {
                        max[0] = nonMax[count][0];
                        max[1] = nonMax[count][1];
                        max[2] = nonMax[count][2];
                }
        }
        for (count = 0; count < pixelCount; count++) {
                i = (unsigned long)(nonMax[count][0]*3*W + 3*nonMax[count][1]);
                *(m_destinationBmp + i) =
                *(m_destinationBmp + i + 1) =
                *(m_destinationBmp + i + 2) = 0;
        }
}
```

Final Words

This tutorial's objective was to show how simple edge detection can be implemented in TRIPOD. Using this as a basis many other operations could be performed on the image. Separate objects can be located, counted, and processed to determine their shape.

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