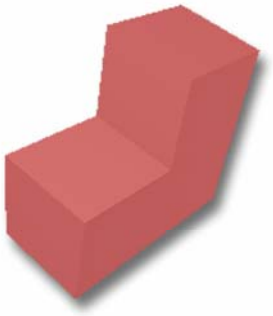


ROBOT VISON ALGORITHMS

By Malcolm Stagg

1. MEAN FILTERING

Mean filtering is used in this project because of its simplicity and effectiveness. The purpose of filtering the image prior to edge detection is to actually decrease sensitivity of the non-maximal suppression. This should lead to straighter, more complete lines.

The method experimented with here is not actually true mean filtering. Normally, the pixels surrounding the current pixels are each weighted equally with the current pixel (Figure 1-1a). However, it was found that the edges were more complete when the current pixel was weighted equally with the sum of the surrounding pixels (Figure 1-1b).

1	1	1
1	1	1
1	1	1

1	1	1
1	8	1
1	1	1

Figure 1-1 a. (left): Most common mean filter kernel. b. (right): This kernel seems to give a better result

This kernel of values passes over the complete image left to right for each row, top to bottom. The process of multiplying this kernel with the entire image is known as convolution. (Figure 1-2) The region around each pixel is multiplied by a set of nine values in the kernel, and the new image is made of the sum of each product.

Three lines of video must be stored in microcontroller RAM at a time for this 3x3 pixel filter to be computed.

```
For each row in the image {
For each column in the row {
image2(x,y) = image(x-1, y-1) * a
+ image(x,y-1) * b
+ image(x+1,y-1) * c
+ image(x-1,y) * d
+ image(x,y) * e
+ image(x+1,y) * f
+ image(x-1,y+1) * g
+ image(x, y+1) * h
+ image(x+1,y+1) * i;
} }
```

Where the kernel is:

a	b	c
d	e	f
g	h	i

Figure 1-2 Formula for convoluting a kernel over the image

2. EDGE DETECTION

a. SOBEL

In my work last year [10] I showed how Sobel is the best simple edge detection method in terms of accuracy and susceptibility to noise. Canny has been shown to have good performance as well, but the complexity and lack of speed made it less appealing.

Similar to the filtering, Sobel operates by convoluting a 3x3 kernel over the image and finding the edge result. Edge detection is often explained as a method which finds areas of high contrast in the image. Basically, it finds areas where the brightness changes, often the edges of simple objects.

The kernel is relatively simple, except you may note from Figure 2-1 that there are two kernels involved. One of these finds horizontal edges

and the other finds vertical edges. This information can be very useful in finding the angle of edges; however, often one just wants to display the actual edges. The equation for obtaining the edges image is shown below (Figures 2-2 and 2-3).

1	0	-1
2	0	-2
1	0	-1

1	2	1
0	0	0
-1	-2	-1

Figure 2-1 Kernels for Sobel edge detection. Note that the left kernel compares the left & right intensities, whereas the right kernel compares the vertical intensities.

$$G_{edge} = \sqrt{G_x^2 + G_y^2}$$

Figure 2-2 Formula for obtaining the final edge result from the horizontal and vertical kernels. This resembles the Pythagorean Theorem.

$$G_{edge} = |G_x| + |G_y|$$

Figure 2-3 Estimation for obtaining the final edge result from the horizontal and vertical kernels. This is easier and much faster than the square root method.

The horizontal and vertical kernel results can be thought of as sides in a triangle, at a 90 degree angle with each other. Trigonometry can be applied to find the angle of the edge, and as shown above, the Pythagorean Theorem can find the final result – the hypotenuse.

While this is the most preferable method, it is quite slow and complicated to take the square root on a microcontroller. The estimation shown in Figure 2-3 was tested on the microcontroller as well as the square root method.

Again, 3 lines of video must be stored at a time for this 3x3 kernel to find and output the edges. The result of this algorithm on a sample image is shown below in Figure 2-4.

Figure 2-4 Sample Sobel edge image of a computer

b. NON-MAXIMAL SUPPRESSION

The only problem with the edges image, as shown in Figure 2-4, is the width of the edges. For vector lines to be computed to represent the input image, most methods require a single-pixel wide line. Thinning can be used to keep only the center of each line. This was experimented with, but it was found that this was extremely slow and often gave poor results. Non-maximal suppression was the obvious alternative.

Sometimes non-maximal suppression is over-complicated by finding the precise edge angle to determine if this edge is the local maxima that can represent the line. In fact, all you need to find is whether the edge direction found a more horizontal or vertical, if the current pixel a maxima with it's horizontal neighbors, and if it's a maxima with it's vertical neighbors (Figure 2-5). This method of non-maximal suppression was discussed in a report on image compression by Desai et al. [2].

When the 3 factors mentioned above have been determined, then the suppression can be computed. It is an edge (not suppressed) if any of the equations in Figure 2-5 are true. Otherwise, the pixel is suppressed to zero.

Is $|G_x|$ | 1 | 0 | -1 |
2 | 0 | -2 |
1 | 0 | -1 | $> |G_y|$ | 1 | 2 | 1 |
0 | 0 | 0 |
-1 | -2 | -1 | ?

$V_{max} = a < b \geq c$

a
b
c

$H_{max} = a < b \geq c$

a	b	c

Figure 2-5 Information needed by the non-maximal suppression formula.

1. $(G_x > G_y)$ & H_{max}
2. $(G_y \geq G_x)$ & V_{max}
3. H_{max} & V_{max}

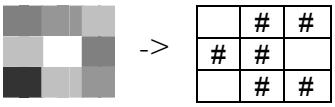
Figure 2-6 Formula for determining actual edges. Pixel is suppressed if all of these are not true.

This is programmed for the microcontroller by storing 4 lines of video, creating two arrays for horizontal/vertical edge, an array for horizontal maximum, and an array for vertical maximum. Non maximal suppression gives the result shown in Figure 2-7.

Figure 2-7 Non-maximal suppression image of the same computer

3. SIMPLE THRESHOLDING

The thresholding used here is very basic, but it uses up really no extra time, and it works for a lot of simple objects and scenes. A threshold is set which each pixel is compared to. If the pixel is greater than or equal to this threshold, it is outputted as a 1 (see Figure 3-1). Otherwise it is outputted as a 0. The result of this on the non-maximal suppression image is shown in Figure 3-2.



	#	#
#	#	
	#	#

Figure 3-1 Example of thresholding a group of pixels based on their gray value

Figure 3-2 Image from Figure 2-7 with threshold = 64/255

4. BINARY FILTERS

a. NOISE FILTER

This noise filter is very simple, yet effective for removing most tiny areas of noise which would slow the vectorization down significantly.

The method described here would pass a digital kernel – similar in some ways to the analog mean and Sobel kernels – over every pixel and the surrounding neighborhood to eliminate noise.

The two kernels for this are shown in Figure 4-1. The first one is good for detecting larger areas of noise, and the second is better for detecting single pixels of noise.

For the kernel in Figure 4-1a, the 16 outside cells are tested to check if they all equal 0. If that is true, and the middle pixel is equal to 1, then the center 3x3 pixels are set to 0. The kernel in Figure 4-1b does almost the same thing. It checks to see if the 8 pixels surrounding the center are all 0, and the center is 1. If that is true, the center is set to 0.

0	0	0	0	0
0				0
0		1		0
0				0
0	0	0	0	0

	0	0	0	
	0	1	0	
	0	0	0	

	0	0	0	
	0	0	0	
	0	0	0	

Figure 4-1 Kernels for noise filter. If either is true, the neighborhood will be set to the values on the right.

After this filter, a real-world result, as shown in Figure 4-2, would be quite neat and tidy. This allows the vectorization to work better, and not slow down when it finds noise.

Figure 3-4 Image from Figure 3-2 with noise filter

b. EIGHT TO FOUR CONNECTIVITY FILTER

This filter has gone through a number of revisions. Its purpose is to find all pixels which are connected diagonally to one another (see Figure 4-3), and add pixels to make them connected up, down, left, and right.

#	
	#

	#
#	

Figure 4-3 These 8-connected pixels are connected to each other diagonally.

#	#
#	#

Figure 4-4 Prevent these 2x2 blobs of pixels at all cost

This started out as simply a filter which would find two diagonally connected pixels, and add another pixel between them to connect them in the 4 directions. However, this sometimes created groups of 4 pixels (Figure 4-4) which would certainly slow down or even stop the vectorization completely.

Then I developed another group of filters – 22 in number – that would find areas where this would likely happen (i.e. a straight line with a bite out of it), and place the new pixels in such a way that these groups of 4 pixels would not be created. The problem with this idea was that passing 22 filters over an image was slow and cumbersome, and would not work well in a microcontroller.

Thinking about this some more, I realized that if I went back to the original simple filters, and checked to make sure a cluster of 4 pixels would not be made, it would be much easier.

This is shown in a mathematical format in Figure 4-5, as a kernel format would be difficult to show correctly.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

```
If (F & K & !G & !J) {
   If (!(C & D & H) & !(H & L) & !(B & C)) {
      G=-1;
   }Else If (!(O & N) & !(M & N & I) & !(I & E)) {
      J=-1;
   }
} Else If (!F & !K & G & J) {
   If (!(B & C) & !(I & E) & !(E & A & B)) {
      F=-1;
   }Else If (!(H & L) & !(L & P & O) & !(O & N) {
      K=-1;
   }
}
```

Figure 4-5 Math that checks to make sure output will be valid before making a pixel 4-connected

c. BROKEN LINES FILTER

This filter is one of the more complex ones I designed, because it can fix lines that are quite broken. How it works is by finding a pixel which is set (part of a line), and one-connected (part of a broken line). It finds another one-connected pixel which is set in the vicinity of the current pixel, and connects the two with a line.

A sample C++ program to demonstrate how this works is shown in Figure 4-7. Notice how it goes through every pixel in the image to see if it is set and 1 connected. If this is true then if searches 11x11 pixels around the current one to find similar pixel. If one can be found it uses

the line formula in Figure 4-8. This is very similar to the line formula in Vectorization.

```
For (y=0; y<239; y++) {
   For (x=0; x<319; x++) {
      If (connected8(x,y) == 1) {
         For (y2=y-5; y2<y+5; y2++) {
            For (x2=x-5; x2<x+5; x2++) {
               If (connected8(x2,y2) != 2) {
                  Drawline(x,y,x2,y2);
               }
            }
         }
      }
   }
}
```

Figure 4-7 Example code to find where to draw new lines to fix broken lines

Assuming the line travels further horizontally than vertically:

```
For (x3=x; x3<x2; x3++) {
   y3=y+x3*(y2-y)/(x2-x)
   PlacePoint(x, y);
}
```

Figure 4-8 This is how a line is drawn between the points (x, y) and (x2, y2)

5. VECTORIZATION

Vectorization is used to look at the input image, and generate a series of vector lines which more-or-less represent the image. The basis of the method used here has been described by Dr. J. R. Parker in [7], however, many changes were made to increase accuracy of the output, and allow an input image from the real world - not a perfect, computer-generated line image.

The method he described would first find a starting point on the image, and then continue adding pixels to the line, and checking if the line still represented the points well. This was done, as illustrated in Figure 5-1, by generating an imaginary line between the first point and the point just added (See Figure 5-2 for the equation). The minimum distance between the pixel and the line horizontally or vertically was calculated, and this was compared to a threshold he set as 1. If it goes above the threshold, then you can add no more to your line, so you can start another until there are no more points to add. As it adds points, it sets them to 0 in the image, so every point can only be added once.

Figure 5-1 The vectorization places an imaginary line between the start and end pixels of an edge segment

$$x = (y_{current} - y_{end}) * (x_{start} - x_{end}) / (y_{start} - y_{end}) + x_{start}$$

$$y = (x_{current} - x_{end}) * (y_{start} - y_{end}) / (x_{start} - x_{end}) + y_{start}$$

Figure 5-2 Vectorization formula for calculating the location of a point between the start and end pixels of a line

The method I propose is similar in many ways. It too continues adding pixels and checking if a line represents them well. One difference though, is that rather than seeing if any one pixel is too far from the line, it finds if the mean distance from the line is too great. When this occurs, it adds a corner at the point furthest from the line. It also adds a corner at the point where it starts the first line. When it can add no more points, it then looks at an identical copy of the image. It starts at a corner connected to some pixels, and follows along until it is close to another corner. As it travels from corner to corner, it adds lines between the points. This collection of lines will accurately represent the image.

6. POST PROCESSING

After vectorization, the image is often far from perfect. There is a common problem of excess vertices, sometimes stray lines, and also lines from noise in the image. A lot of these can be corrected to help the 3D algorithm make the most accurate match.

One of these problems may be tiny lines from two close vertices. If these occur, found most easily by a small dx and small dy where $dx = |x_1 - x_2|$ and $dy = |y_1 - y_2|$, then the line should be eliminated and the two vertices joined together (see Figure 6-1). Ideally, a new vertex should be placed at the mean location

between the two vertices, but by simply deleting one of the vertices, results would still be pretty good.

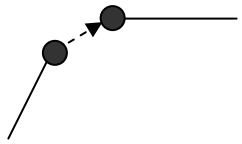


Figure 6-1 When 2 vertices are very close together, they can probably be joined

Another very common problem is extra vertices which aren't really needed to represent the line. These can be found by comparing slopes the lines from one endpoint to the center point, and from the center point to the other end point. If these are close enough, the extra center vertex is deleted, and a line created between the two endpoints (see Figure 6-2).

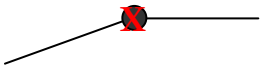


Figure 6-2 When there is an extra vertex not really needed to represent the line, it can be removed

After all this, if there are any vertices only connected to one line, the easiest approach would be to simply delete the line. However, a more accurate plan would be to follow the slope of the line until it comes close to another vertex. However, this requires a lot of computation, and it may make more sense to simply delete the line (see Figure 6-3).

Figure 6-3 When there is a 1-connected line, it is easiest to simply remove it

7. 3D CONVERSION

a. POINT MATCHING

There are two completely different methods of point matching proposed in this project. The first, designed by my uncle, Dr. Peter Stagg, is good for matching points in 2D objects because of its simplicity and computational speed.

The equation in 7-1 shows how a 2D inverse perspective transform will remove perspective from the image. After this, a bubble sort routine will order the transformed y coordinates of the left and right cameras from least to greatest. Every point with the closest y coordinate is matched. If the y coordinates are similar, they are matched according to the x coordinates.

$$x' = (f * x) / (f * \cos(\Theta) + x * \sin(\Theta)$$

$$y' = (f * y * \cos(\Theta)) / (f * \cos(\Theta) + x * \sin(\Theta)$$

Figure 7-1 2D Perspective calculation

$$x' = (f * x) / y$$

$$y' = (f * z) / y$$

Figure 7-2 3D Perspective calculation where f is the focal length

This method seems to work best when the vertical positions of vertices vary a lot. It sometimes may give incorrect results when some vertical positions are quite similar, especially if the cameras are not perfectly aligned.

The second method has been designed by myself. It is excellent for 3D objects because it checks accuracy using the 3D calculations themselves. That will almost guarantee that the points are matched correctly. The only drawback is, if there are a lot of points, every combination must be tested, so it may take quite a while to match the points.

The equation in Figure 7-1 for 2D inverse perspective transform is still used. Then two points go through the 2D to 3D transform described in 7b, giving the x, y, and z coordinates. After this, a 3D to 2D transform with a 3D perspective transform is used (see Figure 7-2). These coordinates are compared to the original coordinates before the 2D perspective transform. If these are close enough, the points are considered to be matched together correctly.

b. STEREOSCOPIC DISPARITY

This is basically how the 3D calculation works. By finding the locations of matching points from two cameras with known angles, the distances to the points in 3D space can be generated (see Figure 7-3).

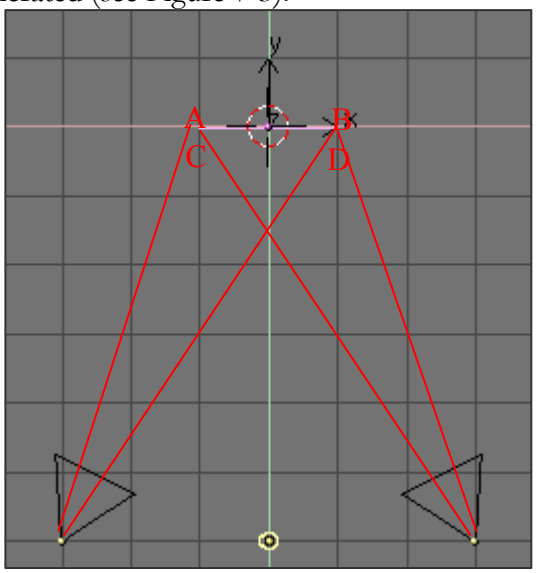


Figure 7-3 Finding the locations of points from two cameras

This equation (Figure 7-4) takes into consideration the camera angles (Θ_1 and Θ_2), the x and y coordinates (x_1, y_1, x_2, and y_2), the focal length (f), the distance between the each camera and a center point (s), the distance from the lens to rotational axis (d), and the height of the lens (y_c).

a = COS(Θ1) + (x1 / f) * SIN(Θ1)
b = COS(Θ2) + (x2 / f) * SIN(Θ2)
c = d * (COS(Θ2) - COS(Θ1))
e = SIN(Θ1) - (x1 / f) * COS(Θ1)
g = SIN(Θ2) - (x2 / f) * COS(Θ2)
h = d * (SIN(Θ2) - SIN(Θ1))
n = (c * e + a * (2 * s - h)) / (a * g - b * e)
x' = -g * n - d * SIN(Θ2) + s
y' = (y2 / f) * n + yc
z' = b * n + d * COS(Θ2)

Figure 7-4 3D Calculations – simplified into variables

When this information is correctly provided, there should be a nice output of x, y, and z values in cm.

c. ANGLE & AREA CALCULATIONS FOR 2D OBJECTS

While looking at a single face of an object, it is possible to find the angles it needs to be rotated to face the camera, and therefore the amount it has been rotated on the z axis, and x axis (tilt). The equations for doing this are shown in Figure 7-5.

$x_{21} = x_2 - x_1$
$x_{31} = x_3 - x_1$
$y_{21} = y_2 - y_1$
$y_{31} = y_3 - y_1$
$z_{21} = z_2 - z_1$
$z_{31} = z_3 - z_1$
$\Theta = \tan^{-1}(-(y_{31} * z_{21} - y_{21} * z_{31}) / (x_{31} * z_{21} - x_{21} * z_{31}))$
$\Phi = \tan^{-1}((x_{21} * \sin(\Theta) + y_{21} * \cos(\Theta)) / z_{21})$

Figure 7-5 Angle calculations for a face in 3D space. Θ is the angle of rotation, and Φ is the angle of tilt.

It is also not too hard to calculate side lengths – which means area as well – from the information of which points are connected. Assuming they are connected in a clockwise order, the code will calculate the total area of an object by dividing it into triangles with a single shared point.

8. OUTPUT STORAGE & DISPLAY

I have designed a file format excellent for storing and displaying 3D objects. There are several similarities to the VRML (Virtual Reality Modeling Language) language, however, VRML is often hard to decode and display, so a simpler alternative has been created.

The file starts with the text "Malcolm File Format 2.0 3D". There was also a version 1.0 which I designed, but version 2.0 allows vertices to be made and connected in many combinations.

After the header, the vertices are defined. This begins with the text "Vertices" followed by the number of vertices. Then each of the following lines defines a vertex in the format x, y, z. After this, a new line begins with the text "Lines", the number of lines, then a definition of the

line in the format (vertex 1, vertex 2). Then faces are defined with the text "Face", the number of faces, and then each face is defined with the number of lines in the face, then each line number.

An example of a file in this format is shown in **Listing 10**. This is code for a triangle in 3D space. If multiple faces are not considered, the faces section can be set to zero and left blank.

Prior to designing this 3D format, I also designed a similar 2D format shown in **Listing 11**. Not only does this include all the vertex locations in 2D space, but also all the information required for a 2D to 3D conversion (**see Listing 4**).

Displaying the output is done by first loading the 3D image and storing all the vertex and line locations, as well as the number of lines. As can be seen in the program in **Listing 5**, each line is read to find the vertices which it connects together, then theses vertex locations are found, and connected with a rotated, scaled, and translated line. The equations for this from Listing 5 are shown in Figure 8-1.

```
x' = x * scx + tx
y' = y * scy + ty
z' = z * scz + tz

cx = Cos(rx * pi / 180)
sx = Sin(rx * pi / 180)

cy = Cos(ry * pi / 180)
sy = Sin(ry * pi / 180)

cz = Cos(rz * pi / 180)
sz = Sin(rz * pi / 180)

x" = (x' * cz - y' * sz)
y" = (x' * sz + y' * cz)

y"' = (y" * cx - z' * sx)
z" = (y" * sx + z' * cx)

x"' = (x" * cy + z" * sy)
z"' = (-x" * sy + z" * cy)
```

Figure 8-1 Rotation, translation, scaling calculations for viewing the object on a computer

9. PC BOARD CONSTRUCTION

I have designed printed circuit board layouts for every part of the circuit, but at this current time, I have only been able to complete the first. Design was done in a program called ExpressPCB. I chose that program because I am new to designing PC boards, and it has a very simple interface.

When designing a board, you must consider placement of parts to make it easy to connect them together with the metal traces. It is also possible to link traces from one side of the board to the other.

Construction of the boards can be done with several chemicals. First, you must have a transparency of the pattern to transfer to the board. Then, exposing the presensitized board and pattern to ultraviolet light for a few minutes will transfer the pattern to the board.

The extra resist which does not belong to the pattern can be removed using Sodium Hydroxide – a strong base. After this, all that is remaining on the copper board is the pattern. Putting the board in Ferric Chloride – a strong acid – for half an hour or so will remove the exposed copper from the board, leaving only the pattern in metal.

Unfortunately, only the first board could not be constructed due to a misaligned laser printer which wasted 2 circuit boards.

10. HOW IT ALL WORKS TOGETHER

I have made a flow chart in Figure 10-1 which shows an overall look at how the design for this system works. Notice how everything is duplicated for the left and right cameras until the 3D conversion is reached.

Because of the low processing power of the microcontrollers, it was necessary to keep the algorithms used as simple and fast as possible. For that reason, no methods like thinning, which use many iterations to filter the image, were used.

However, to get a good result in vectorization, a number of filters had to be designed to fix-up the image before placing lines. This slowed it down quite a bit, but was necessary to obtain reasonable results.

It is also interesting to note, that if a microcontroller were added between two sections of the circuit, it would be possible to divide the output to multiple circuits. In the future, this may allow for finding motion of objects, or looking back at the original grayscale images to determine the shade of a face.

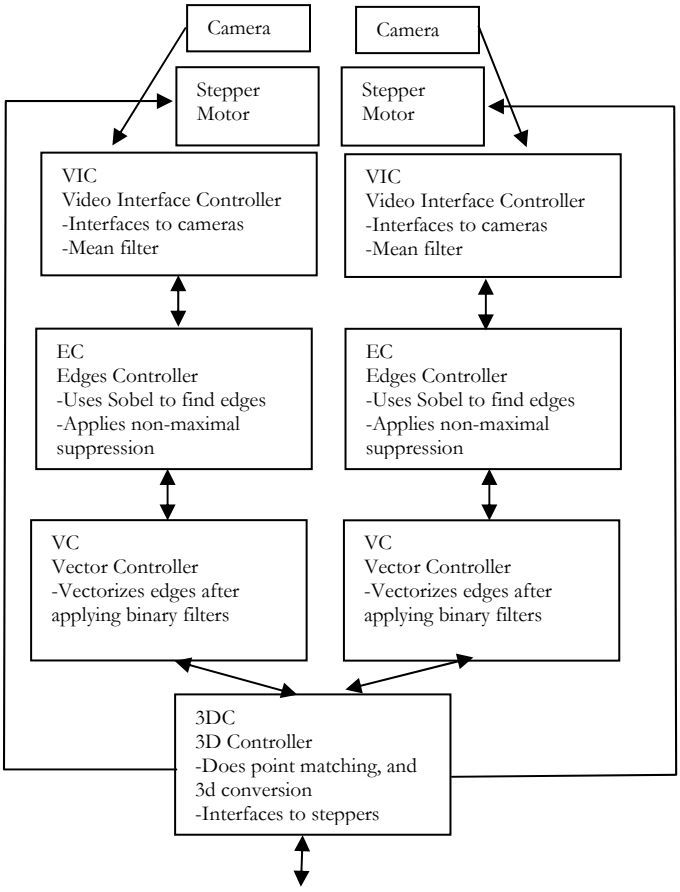


Figure 10-1 Flow chart of basic system layout