

DRAWING GLOW MASKS: DETECT CONTOUR & CHAINCODE

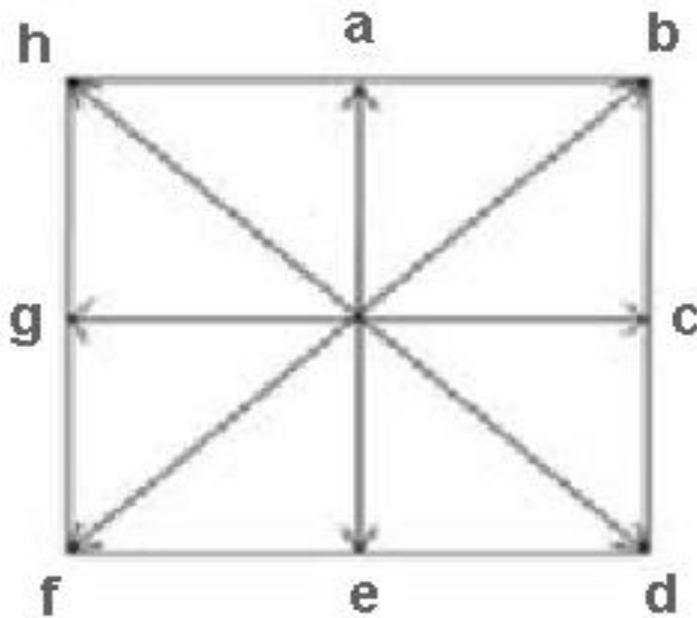
THE STARTING POINT:

It is February 22, 2026. About two weeks ago I set out to write an algorithm to detect the contour of an icon image to build the chaincode to write to a text file for drawing glow masks. It's much easier and faster for the computer to generate the chaincode to draw a convincingly good and accurate glow mask. By hand it would be very tedious and time consuming to use a paint program to draw a glow mask. So, we have a computer do it.

Two weeks ago I barely had any idea about how to achieve that goal. Previously, I had written a function to import chaincode from a text file to use the chaincode to draw a glow mask. The function used pixel search patterns and x-y transitions. So, I wanted to do something similar when writing code to detect contours of the icon image. where to begin writing the code? In my experience the best way to go about it is to just start writing and compiling and testing the code. Then revise the code as needed along the way.

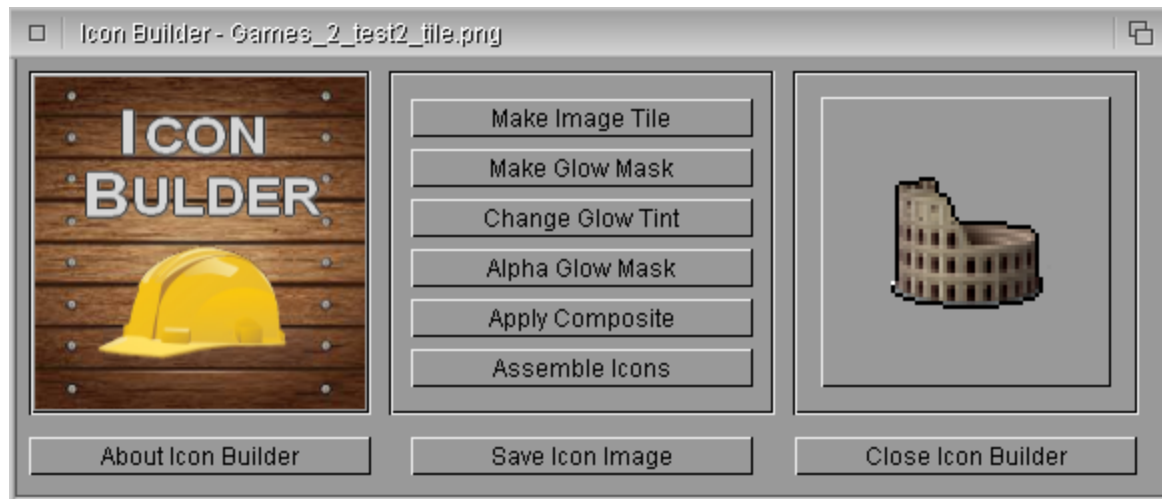
THE BASIC CONCEPTS:

For Detecting Contours of Icon Images I chose to use the 8-connected chaincode as shown in the diagram below which is based on Freeman Chain Code, information about which is available on the internet. Freeman Chain Code is defined as a lossless method representing 2D object boundaries in digital images by tracing the contour and encoding the direction of each movement between pixels. It uses a sequence of directional numbers (0–3 for 4-connectivity or 0–7 for 8-connectivity) to efficiently store and analyze shapes, often used in handwriting analysis, character recognition, and with image compression. Instead of using numbers for the principal directions I chose to use ascii characters to make it easier to understand what is going on in the computer code, rather than using numbers only for the directions and pixel counts which can become very confusing.



In order to better understand the 8-connected chaincode diagram think of it as the directions on a compass that indicate the direction of travel. The rectangular directions which represent 90 degree angles consist of directions a,c,e,g. As points on a compass they correspond to North, East, South and West. The oblique directions which are 45 degree angles consist of directions b,d,f,h. The corresponding compass directions are North East, South East, North West and South West.

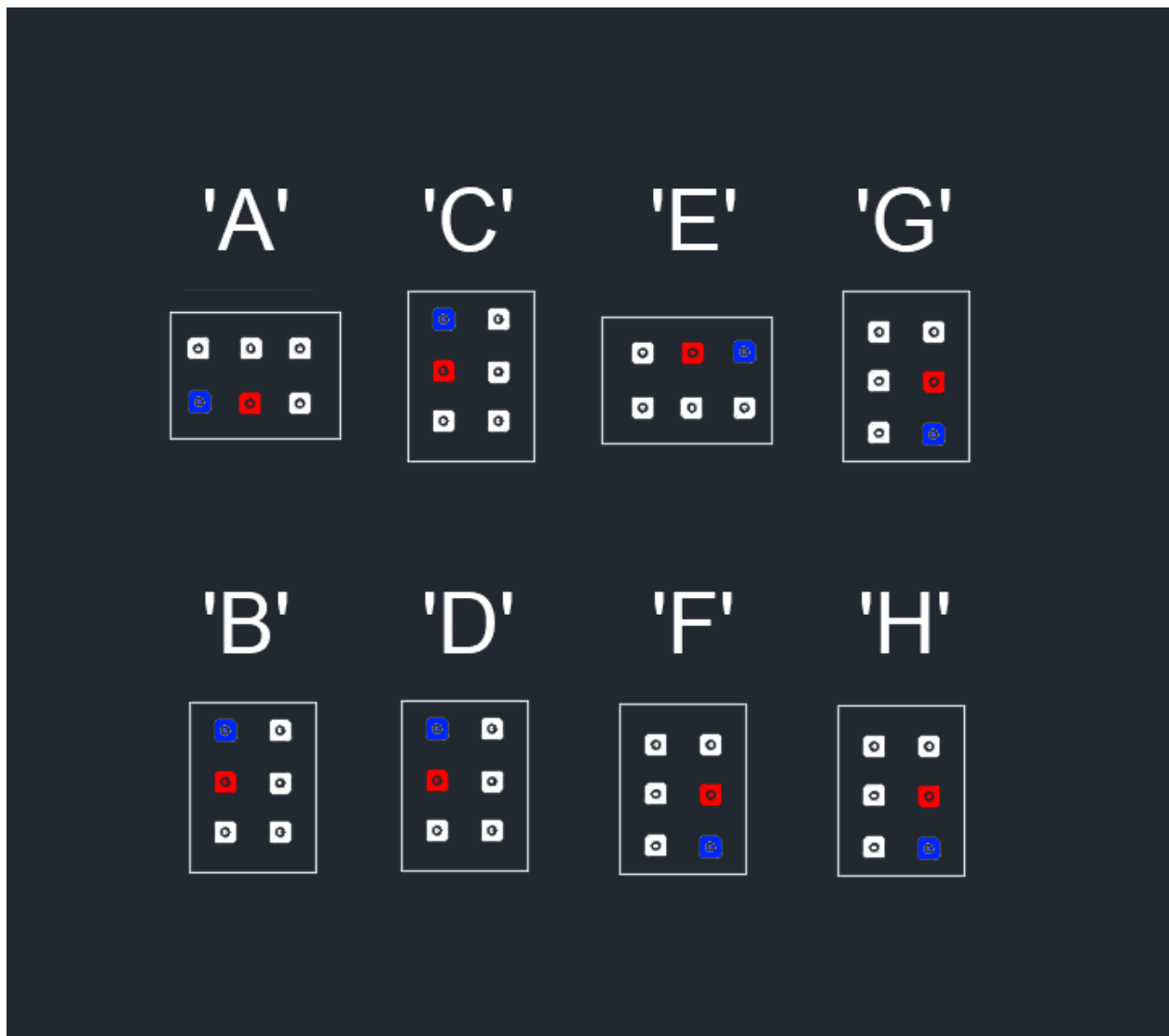
The chaincode always has a start point (control point) that is the first opaque pixel with an alpha value of 255 that is nearest the top and left of the image. For the chaincode to work we always move from the start point in a clockwise manner, processing all the perimeter pixels and storing them in the chaincode structure. The contour shape is complete only when we return to the chaincode start point. In the screenshot of the Games Icon Image the start point on the left side is the white pixel. The perimeter pixels that have been processed are the black pixels.



Working with the chaincode requires some definitions. The Image Tile that contains the icon image to be processed has to be prepared before contour detection can begin. It's called an Image Tile because the background has been removed. All the black background pixels have alpha values set to 0 while the opaque image (RGB Image) in the middle have alpha values set to 255. And some semi-transparent pixels around the edge of the opaque image nearest the background have alpha values that are greater than 0 but less than 255. The fully opaque pixels are called "full pixels". The fully transparent pixels are called "non-pixels" and the semi transparent pixels around the edges of the RGB Image are called "half pixels". What is a Perimeter Pixel? The definition of a perimeter pixel is a full pixel or half pixel (considered a full pixel) that is contiguous to the current perimeter pixel which is in turn adjacent to at least one non-pixel that is part of the background of the icon image. That's the basis of the various search patterns that are used to find the perimeter pixels.

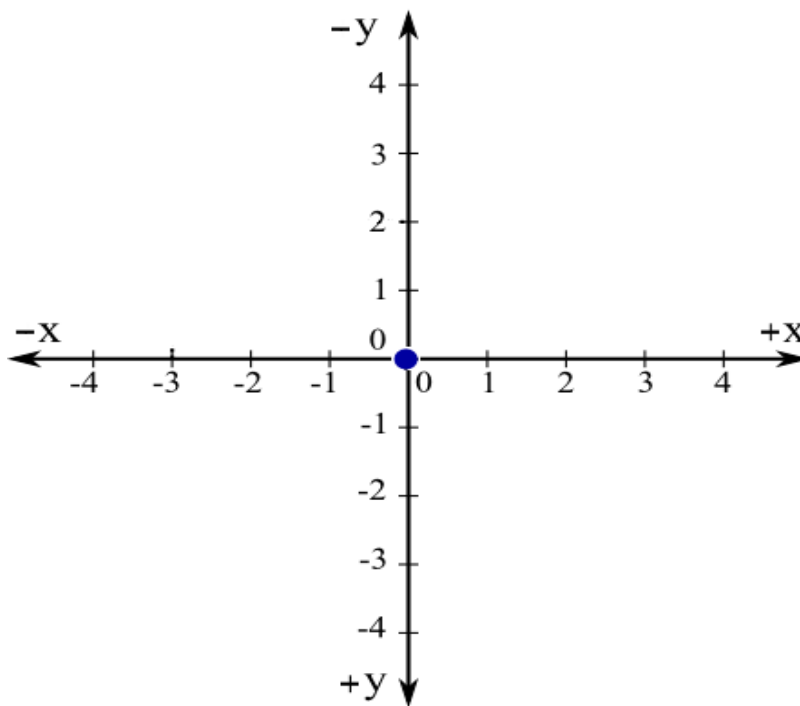
The pixel search patterns that are used to detect the contour of the icon image are arranged around the current pixel according to the direction of travel along the outside of the RGB image. Each direction has an associated search pattern. In the pixel search diagram the central red dot is the current pixel and the white dots are part of the search pattern while the blue dot is the start point of the search pattern. The search patterns "ride the edge" of the RGB Image trying to find the non-pixels that are adjacent to full pixels that are touching the current pixel. Then the new perimeter pixel becomes the current pixel and the process continues, collecting all the perimeter pixels around the outside of the RGB icon image.

When there is a change of direction the previous pixel count is stored in the chaincode then the new direction is also stored in the chaincode, waiting for the next direction change to store the current pixel count. At the end when the current pixel is actually at the start point then the contour shape is complete. But since there is no actual change of direction when the start point is reached the last pixel count is stored and the final chaincode pair count is calculated and stored. The chaincode count is needed to accurately setup for loops in the computer code to read the chaincode byte pairs that are the directions and pixel counts.



Now that we have all the basic concepts and definitions out of the way, onward to the computer code to detect contours of icon images using search patterns and x-y transitions

which are based on a modified cartesian coordinate system where the y-axis is flipped. When using Cartesian Coordinates going from right to left is the -x direction. While going from left to right is the +x direction. But since the origin of an icon image is in the upper left corner (0,0) the index values of the pixel data as well as the y coordinates increase as we go from top to bottom of icon images. So, going from top to bottom is +y and going from bottom to top is actually -y. That is the opposite of the y-axis for standard Cartesian Coordinates. Looking at the Cartesian Coordinates Diagram we can see that the y-axis actually has been flipped. So, it's very important to keep the change in coordinates in mind.



THE COMPUTER CODE:

There are two main c code structs being used in the Detect Contour Algorithm. They are struct IconImage *im that contains the pixel data and struct ChainCode *chaincode. The main part of the algorithm consists of two for loops, the upper loop and the lower loop. The main algorithm only directs traffic and coordinates the results. This was done so that it would be easier to edit the code and easier to understand what is going on with the code. There are also another five helper functions that do most of the hard work. The functions are Is_Pixel, GetDirection, GetTransition_X, GetTransition_Y and Write_Chaincode.

```

struct ChainCode
{
    int Count;
    int Ctrl_X;
    int Ctrl_Y;
    unsigned char Code[1000];
};

//struct IconImage
struct IconImage
{
    long          Depth;
    long          NumColors;
    long          Width;
    long          Height;
    unsigned char ImageData[MAXICONSIZE*MAXICONSIZE*MAXDEPTH/8];
    struct IconPalette Palette[MAXCOLORS];
    long          Transparency; /* Transparent color or -1 */
};

```

The beginning of the Detect Contour Algorithm is the upper for loops where there is an X loop for width and a Y loop for height. These for loops use srce_xy which is the coordinate of the alpha values of all the pixels in the byte array such as im->ImageData[srce_xy]. The goal is to find the X-Y coordinates of the control point which is the starting point for the chaincode heading in a clockwise direction around the perimeter of the icon image. The control point is the first pixel with an alpha value of 255 that is closest to the top and left of the icon image. Once the control point is found then count is set to 1 (it starts at 0). Count will be re-used as the index counter for the chaincode values later in the next for loop.

```

/*****
/* FIND CONTROL POINTS X-Y */
*****/

//For-Loops to find control pt
for(x = 0; x < width; ++x)
{
    for(y = 0; y < height; ++y)
    {
        //Read Across Scanlines to Find Control Point (x, Y)
        //First Alpha Value = 255 reading from left to right is Control Point
        srce_xy = (y * width + x) * 4;
        alpha = im->ImageData[srce_xy];

        if(alpha == 255)
        {
            if(count == 0) //If count still = 0 add control x, control y & count++
            {
                chaincode->Ctrl_X = x + 1;
                if(verbose) Printf("Write Chain Code: Control_X: %d\n", x+1);
                chaincode->Ctrl_Y = y + 1;
                chaincode->Code[count] = direction;
                if(verbose) Printf("Write Chain Code: Control_Y: %d\n", y+1);
                dx = x; //set initial dx (change in x)
                dy = y; //set initial dy (change in y)
                count++; //count = 1 //used as chaincode byte counter
            }
        }
    }
}
}

```

After the upper for loops another for loop begins which is the main part of the algorithm. The counter for this loop is arbitrarily set at 200 ($46+46+46+46=184$ + some padding). So, we can collect up to 200 perimeter pixels with this loop. But the loop will end before that because at the bottom of the for loop is a conditional statement that sets a break point such that when we arrive back at the start point we finish up remaining tasks then break. This drops out of the for loop and the algorithm ends. Above the break point is the main functionality of the algorithm and below that is what we do when there is a change in direction. That is when we store the previous pixel count and the new direction in the chaincode structure. Then "change" is set to FALSE otherwise it would be stuck in a continuous change of direction loop and pixel count would always be just 1 which is not the desired result. Just after finding the control points the initial direction is set to 'c' since we are traveling in a clockwise direction from the control point, although the initial direction may not necessarily be 'c' but that's what we start with to get things going.

```

/*****
/* SEARCH PARAMETERS WITH X-Y TRANSITIONS */
*****/

//For-Loop to find contour (perimeter pixel locations)
//for loop based on perimeter of image + (width/2) ??
for(j = 0; j < 200; ++j)
{
    if(verbose) Printf("Current_X: %d\n", dx);
    if(verbose) Printf("Current_Y: %d\n", dy);

    //Write black pixel to current perimeter pixel location
    Write_Pixel("black", srce_x, srce_y, dx, dy, win);
    if(verbose) Printf("Write Chain Code: Direction: %c\n", direction);
    if(verbose) Printf("Write Chain Code: PixelCount: %d\n", pixelcount);

    /*****
    /* MAIN FUNCTIONALITY */
    *****/

    /* Method to use alternate search patterns */
    if(direction == 'b' || direction == 'h')
    {
        dir = GetDirection(im, direction, x, y); //get direction
        dir = GetDirection(im, 'a', x, y); //get alternate
    }
    if(direction == 'f' || direction == 'd')
    {
        dir = GetDirection(im, direction, x, y); //get direction
        dir = GetDirection(im, 'e', x, y); //get alternate
    }
    else
    {
        dir = GetDirection(im, direction, x, y); //get new pixel direction
    }

    dx = GetTransition_X(im, dir, x);
    dy = GetTransition_Y(im, dir, y);
    if(verbose) Printf("Transition_X: %d\n", dx);
    if(verbose) Printf("Transition_Y: %d\n", dy);

    if(dir != direction) change = TRUE;
    pixelcount++;
}

```

The GetDirection function contains all of the search patterns being used for each principal direction of travel. But there are only four search patterns, though there are eight directions

of travel because a,e directions have their own patterns, but b,c,d are grouped together and f,g,h are grouped together as well. While developing the algorithm I noticed that the oblique directions b,d,f,h are each located between two rectangular directions so they can acquire the search pattern for either one. Such as direction 'b' is assigned to use the search pattern for direction 'c' by default but sometimes like when ascending a vertical edge as in the left edge of the Games Icon Image it requires the search pattern for direction 'a' instead. So, to solve this problem of alternate search patterns we simply run the first pattern then the second pattern in sequence and we let the computer decide which result that it will use. After GetDirection and the Alternate Search Patterns comes two helper functions which are GetTransition_X and GetTransition_Y which provide the coordinates of the next perimeter pixel, then it becomes the Current Perimeter pixel and the entire process starts again.

```
int GetTransition_X(struct IconImage *im, unsigned char dir, int x)
{
    int dx;

    //When direction = 'a' //x,y-1
    if(dir == 'a')
    {
        dx = x;
    }
    //When direction = 'b' //x+1,y-1
    if(dir == 'b')
    {
        dx = x+1;
    }
    //When direction = 'c' //x+1,y
    if(dir == 'c')
    {
        dx = x+1;
    }
    //When direction = 'd' //x+1,y+1
    if(dir == 'd')
    {
        dx = x+1;
    }
    //When direction = 'e' //x,y+1
    if(dir == 'e')
    {
        dx = x;
    }
    //When direction = 'f' //x-1,y+1
    if(dir == 'f')
    {
        dx = x-1;
    }
    //When direction = 'g' //x-1,y
    if(dir == 'g')
    {
        dx = x-1;
    }
    //When direction = 'h' //x-1,y-1
    if(dir == 'h')
    {
        dx = x-1;
    }
    //else dx = 0;

    if(verbose) Printf("Transition_X: %d\n", dx);

    return dx;
}
```

Immediately after GetTransition_X and GetTransition_Y the conditional statement compares dir to direction to see if there is a change in direction. If there is a change in direction then the next block of code will address that by storing the previous pixel count and by storing the new direction of travel.

```

/*****
/*  CHANGE OF DIRECTION  */
*****/

if(change == TRUE)
{
    if(verbose) Printf("****Change Dir: Current Direction: %c\n", direction);
    if(verbose) Printf("****Change Dir: Proposed Direction: %c\n", dir);
    //store new direction //'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h'
    direction = dir;
    //Store pixelcount
    //Don't count the pixel before the direction change //pixelcount - 1
    if(verbose) Printf("Current Pixel Count: %d\n", pixelcount - 1);
    chaincode->Code[count] = pixelcount - 1;
    chaincode->Code[count+1] = dir;
    //But we count the pixel after the direction change //pixelcount = 1
    pixelcount = 1;
    count += 2;
    change = FALSE; //remember to reset change value
}

/*****
/*  FINISH REMAINING TASKS  */
*****/

//Break point to exit for-loop
if((dx == chaincode->Ctrl_X - 1) && (dy == chaincode->Ctrl_Y - 1))
{
    /* Handle End Case when we arrive at the start pt */
    if(verbose) Printf("End Perimeter Pixel Search.\n");
    /* Draw a White Pixel to indicate we're at the Start Pt */
    Write_Pixel("white", srce_x, srce_y, dx, dy, win);

    if(verbose) Printf("****Current Pixel Count: %d\n", pixelcount);
    chaincode->Code[count] = pixelcount; //Store pixelcount in chaincode structure
    count += 1; //What should this be?
    chaincount = count;
    if(verbose) Printf("****Current Chain Count: %d\n", chaincount);
    chaincode->Count = (chaincount/2); //Store chaincount in chaincode structure

    //Write Chain Code to output text file only if we're at the start pt
    char *outputfile = "Ram Disk:Test_Code_CC.txt";
    Write_Chaincode(outputfile, chaincode);
    break;
}
x = dx; y = dy; //Reset x, y based on dx, dy

```

The main helper function actually does most of the hard work. `GetDirection` contains all the Search Patterns and it returns the new direction from the Current Pixel to the next Pixel along the perimeter of the icon image. As noted previously, there are only four search patterns though there are eight directions of travel in the 8-connected chaincode diagram.

```

/*****
/*  PIXEL SEARCH PATTERNS  */
*****/

//When direction = 'a' //if(direction == 'a')
if(direction == 'a')
{
    //if(Is_Pixel(im, x-1,y) == FALSE) set bit at index 8 to 1
    //Syntax: bitbuffer | (1 << bit_position_to_set) //5 bits 8,7,6,5,4
    //Printf("Ans: %d", bitbuffer | (1 << 5)); //set 5th bit to 1
    //Syntax: number & ~(1 << bit_position_to_clear)
    //Printf("Ans: %d", bitbuffer & ~(1 << 2) );
    //Syntax: number & (1 << bit_position_to_check)
    //Printf("Ans: %d \n", bitbuffer & (1 << 5) ); //check if bit is set

    if(Is_Pixel(im, x-1,y) == FALSE) pix[0] = 'f'; //x-1,y = 'g'
    else pix[0] = 't';
    if(Is_Pixel(im, x-1,y-1) == FALSE) pix[1] = 'f'; //x-1,y-1 = 'h'
    else pix[1] = 't';
    if(Is_Pixel(im, x,y-1) == FALSE) pix[2] = 'f'; //x,y-1 = 'a'
    else pix[2] = 't';
    if(Is_Pixel(im, x+1,y-1) == FALSE) pix[3] = 'f'; //x+1,y-1 = 'b'
    else pix[3] = 't';
    if(Is_Pixel(im, x+1,y) == FALSE) pix[4] = 'f'; //x+1,y = 'c'
    else pix[4] = 't';
    pix[5] = 0; //End of string

    //Mask:  00001111b
    //Value:  01010101b
    //Result: 00000101b

    //uint8_t stuff(...)
    //{
        //uint8_t mask = 0x0f; //00001111b //0000000F
        //uint8_t value = 0x55; //01010101b //00000055
        //return mask & value; //00000101b
    //}

    //Compare strings to get new direction
    //Compare hex values to get new direction
    if(strcmp(pix,"tfttt") == 0) dir = 'g'; //bitbuffer = 10111
    if(strcmp(pix,"ftttt") == 0) dir = 'h'; //bitbuffer = 01111
    if(strcmp(pix,"ffttt") == 0) dir = 'a'; //bitbuffer = 00111
    if(strcmp(pix,"fffft") == 0) dir = 'b'; //bitbuffer = 00011
    if(strcmp(pix,"fffff") == 0) dir = 'c'; //bitbuffer = 00001
}

```

```

//When direction = 'c' //if(direction == 'c')
else if(direction == 'b' || direction == 'c' || direction == 'd')
{
    if(Is_Pixel(im, x,y-1) == FALSE) pix[0] = 'f'; //x,y-1 = 'a'
    else pix[0] = 't';
    if(Is_Pixel(im, x+1,y-1) == FALSE) pix[1] = 'f'; //x+1,y-1 = 'b'
    else pix[1] = 't';
    if(Is_Pixel(im, x+1,y) == FALSE) pix[2] = 'f'; //x+1,y = 'c'
    else pix[2] = 't';
    if(Is_Pixel(im, x+1,y+1) == FALSE) pix[3] = 'f'; //x+1,y+1 = 'd'
    else pix[3] = 't';
    if(Is_Pixel(im, x,y+1) == FALSE) pix[4] = 'f'; //x,y+1 = 'e'
    else pix[4] = 't';
    pix[5] = 0; //End of string

    //Compare strings to get new direction
    if(strcmp(pix,"tfttt") == 0) dir = 'a';
    if(strcmp(pix,"ftttt") == 0) dir = 'b';
    if(strcmp(pix,"ffttt") == 0) dir = 'c';
    if(strcmp(pix,"fffft") == 0) dir = 'd';
    if(strcmp(pix,"ffffff") == 0) dir = 'e';
}

```

Originally GetDirection was using string comparisons but it was eventually changed to use bitmasking instead to make it more compact and more efficient. Search Pattern A has the prototype code for bitmasking.

THE CHAINCODE TEXT FILE:

The function Write_Chaincode writes the chaincode generated by Detect Contour at the end of the algorithm to a text file for later use by Draw_Glow_Mask which reads the text file to reconstitute the chaincode structure to do parallel offsets to draw the color bands of the glow mask around the perimeter of the icon image. The char string for each line of the chaincode text file is exactly 12 bytes long. First use sprintf to add the text portion that is separated by a colon (':') then the chaincode values. When Draw Glow Mask reads the lines of the text file it splits each string at the colon and grabs the values on the right side after the split. Detect Contour will only write the chaincode to a text file if the Boolean 'code' is used such as "Detect_Contour(im, TRUE);". If FALSE it only returns the struct.

```

static void Write_Chaincode(char *outputfile, struct ChainCode *chaincode)
{
    int count;
    int j, k, s;
    int ccsz = 12;
    char ccString[ccsz];

    s = 0;
    count = chaincode->Count;
    /* Check Chaincode pairs for Zero Entries (c,0) */
    if(chaincode->Code[0] == 'c' && chaincode->Code[1] == 0)
    {
        s = 2; //starting byte pair for chaincode byte array
        //count--; //DON'T remove the byte pair from chaincount
        chaincode->Count = count-1; //adjust the chaincode value
    }

    BPTR ccfile = Open(outputfile, MODE_NEWFILE);
    if(ccfile)
    {
        /* Chaincode count */
        for(j=s; j<count*2; j+=2)
        {
            /* Chaincode pairs */
            for(k=0; k<2; k++)
            {
                if(k == 0)
                {
                    //example chaincode[j+k] = 'c'
                    sprintf(ccString, "chain_code:%c", chaincode->Code[j+k]);
                    Fputs(ccfile, ccString);
                    Fputc(ccfile, 0x0D); Fputc(ccfile, 0x0A); /* add CRLF */
                }
                if(k == 1)
                {
                    //example: chaincode[j+k] = '37'
                    sprintf(ccString, "chain_code:%d", chaincode->Code[j+k]);
                    Fputs(ccfile, ccString);
                    Fputc(ccfile, 0x0D); Fputc(ccfile, 0x0A); /* add CRLF */
                }
            }
        }

        /* write control_x string to outputfile */
        sprintf(ccString, "ccontrol_x:%d", chaincode->Ctrl_X);
        Fputs(ccfile, ccString);
        Fputc(ccfile, 0x0D); Fputc(ccfile, 0x0A); /* add CRLF */
        /* write control_y string to outputfile */
        sprintf(ccString, "ccontrol_y:%d", chaincode->Ctrl_Y);
        Fputs(ccfile, ccString);
        Fputc(ccfile, 0x0D); Fputc(ccfile, 0x0A); /* add CRLF */
        /* write chaincount string to outputfile - no CRLF */
        sprintf(ccString, "chaincount:%d", chaincode->Count);
        Fputs(ccfile, ccString);
        Close(ccfile);
        if(verbose) Printf("Success Writing Chain Code.\n");
    }
    else
        if(verbose) Printf("Failed to Write Chain Code! %s\n", outputfile);
}

```

DRAWING THE GLOW MASK:

Now that we have generated the chaincode and we have written it to a text file the Draw Glow Mask Algorithm can import the chaincode text file to rebuild the chaincode struct, allowing it to use the information to draw a beautiful glow mask around the image! The Glow Mask in turn is used by whichever method is preferred to make a Glow Border.



CHAINCODE TO GRAPHICS PATH:

If chaincode is like a road map of the directions of travel around the perimeter of the icon image then a graphics path is like the GPS coordinates of every pixel stop along the way. The chaincode structure can easily be converted to a graphics path structure which consists of the 1-byte count followed by X-Y byte pairs for the coordinates for each of the perimeter pixels. The count can be determined by adding all pixel counts in the chaincode. Then all the X-Y coordinates can be gathered by following the chaincode direction and pixelcount pairs around the perimeter of the icon image and back again to the start point. The dedicated function `Convert_Chaincode` can thus convert chaincode to graphics path.

USES FOR A GRAPHICS PATH:

Having a graphics path is convenient because hit testing can be used to check if a given point (pixel) is located inside, outside or on the graphics path. It's necessary to verify if it is part of the opaque image or the transparent background ? Once a hit test has been developed it's possible to use the graphics path to generate a selection set. All the pixels on and inside the graphics path represent the opaque image in the middle of the icon image. Likewise, all the pixels on the outside of the graphics path represent the background of the icon image. So, it becomes easier to select only the opaque image in the middle of the icon to perform a color change operation such as HSL hue rotation to change the color tint of the image without changing the background. Also, it becomes possible to accurately select only the opaque image so that it can be overlaid on top of a newly made drop shadow or glow border to make a final composite image to complete the process of adding a new drop shadow or glow border to an existing icon image. Or maybe it's just a really good way to do some parallel offsets around the perimeter of an icon image to draw a nice glow mask!