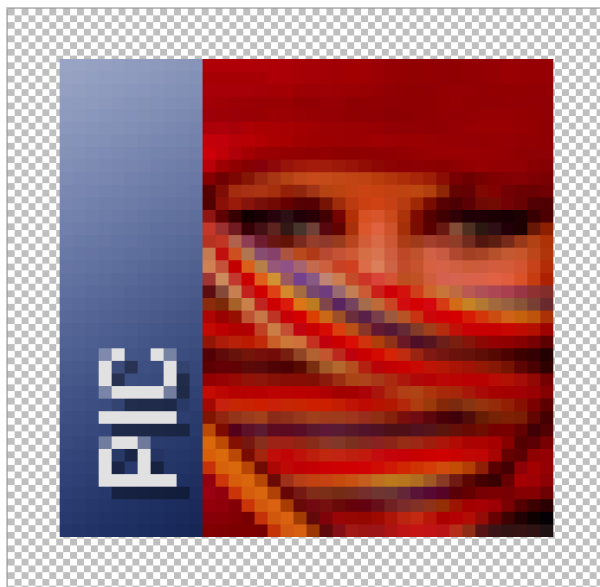


Drawing Glow Masks with Chain Code

It was just a few weeks ago that I started developing the “manual method” to draw a glow mask using a paint program. I sought the advice of the icon master Ken Lester to find out exactly how to make a convincingly good glow border. In order to explain the process a few definitions are needed. There is a difference between a “glow mask” and a “glow border”. The term glow border refers in this sense to the 32 bit ARGB Glow Border we are most familiar with. The term glow mask refers to the RGB image used to produce the glow border. The Blur Effect which is just a Gaussian Blur with Radius 2 is applied to the glow mask which produces the 32 bit glow border.

The manual method was needed so that I would know how to proceed to write a computer code that would draw a glow mask for an icon image automatically. Using a paint program the manual method follows some pre-defined rules for drawing a glow mask. Again, a few definitions will be required to help explain. We start by preparing the original image (second image) to receive a glow border. In many cases the second image is just a copy of the first image that is darkened. For Ken’s Icons the second image is darkened by 30%. That seems like a lot but it’s just perfect.

To prepare the second image we remove the background by erasing it with the paint program. Essentially it makes all the background pixels white with an alpha value of zero. This makes all the background pixels completely transparent. While the RGB Image in the middle has alpha values of 255 which make them completely opaque. This makes the Magic Wand selection of the RGB Image much easier. Later, in the computer program, it also makes things much easier.



We start at the Top Left at the very first pixel of the RGB Image in the middle. We can disregard the transparent background for now. So the Start Point is the first colored pixel that is closest to the top and left of the image. This just seems convenient. Also we draw in a clockwise direction. We start by using White (255,255,255 & alpha 255) to draw the inner band of the glow mask. It is just a simple parallel offset from the outer perimeter of the image. After the white band comes yellow which I call “gold” then orange. There are only three color bands for an actual glow mask.

```

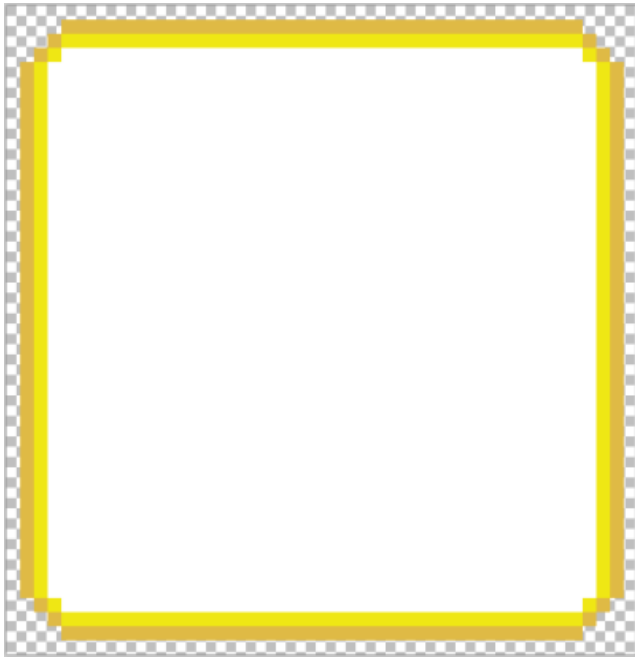
if(stricmp(color, "white")==0)
{
    pixel32[0] = 0;
    pixel32[1] = 255;
    pixel32[2] = 255;
    pixel32[3] = 255;
}
else if(stricmp(color, "gold")==0)
{
    pixel32[0] = 0;
    pixel32[1] = 239;
    pixel32[2] = 231;
    pixel32[3] = 20;
}
else if(stricmp(color, "orange")==0)
{
    pixel32[0] = 0;
    pixel32[1] = 223;
    pixel32[2] = 186;
    pixel32[3] = 69;
}

```

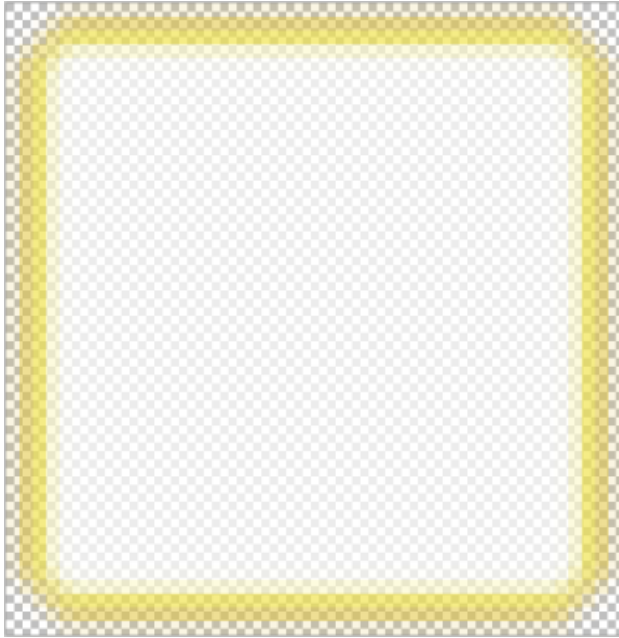
After we draw the white band in a clockwise direction all the way around till we reach the start point again then we draw the gold band in the same way. Then, finally, we draw the orange band as the outer color band of the glow mask. This is a sample of the completed glow mask. Notice how there are additional colored pixels to fill the gaps at the end of each line segment.



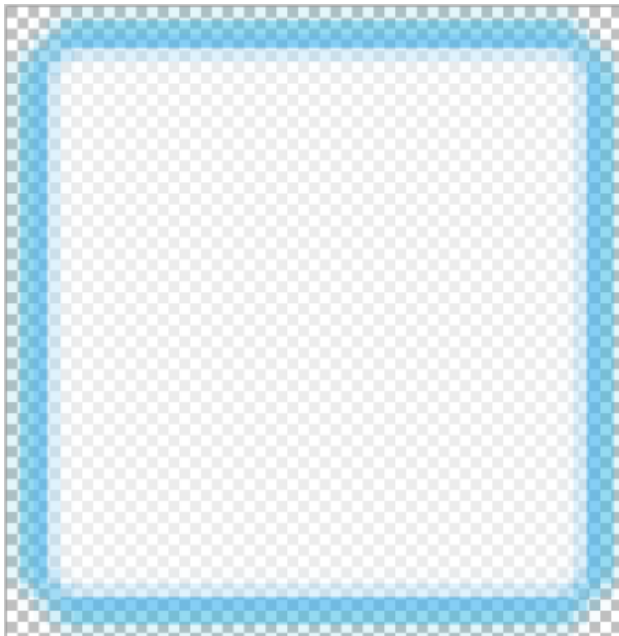
The next step for the manual method is to “Clean” the Glow Mask by removing the RGB Image in the middle to be replaced by White Pixels. The Glow Mask needs to have white pixels in the middle for a specific reason. In Color Theory the color white is a “Tint” while the color black or various grey colors are a “Shade”. By adding a Tint it increases the overall brightness. While adding a Shade decreases the overall brightness. When the Gaussian Blur Effect is applied to the Glow Mask it averages the pixels using weighted averages to produce the blur. It also uses the alpha values to average them and disperse them in a visible pattern from the center toward the outside of the glow border. If we leave the RGB Image in the middle a resulting glow border will be much too dark. So we paint the inside of the Glow Mask using the color white instead.



For the next step Ken Lester shared with me his technique for producing the beautiful 32 bit glow borders for his 32 bit icons. He uses PhotoShop to apply the Blur Effect to the Glow Mask. Start by opening the Glow Mask itself and it looks just like the one pictured above. Then set the overall Opacity to 70% which actually just multiplies all the alpha values by 0.70 to give them all a lower value. This prepares the glow mask to receive the blur effect. The alpha values need to be preset because they are averaged just like the other RGB values for each pixel in the mask. He applies the Blur Effect twice to get the correct dispersal of all the alpha values in the image.



At this point it is important to mention one more thing. By changing the colors of the Glow Mask to either blue or green, or some other color combination, before applying the blur effect we can now produce some nice neon blue or neon green glow borders for icon images. Ken reminded me that the colored glow borders weren't used for the original icon images. He painted his icon images and made his glow borders over 20 years ago when color wasn't really appreciated as "a thing". So all his original icon image only have Yellow Glow Borders. But in a sense, at least they are all consistent and they are however of a very high quality. They are all quite beautiful.



The last step to produce a high quality glow border is to assemble the “Composite” Image by selecting the RGB Image from the Tile Image using the Magic Wand selection tool. This is then copied directly over the new glow border image. Since both images are the exact same size the RGB Image is copied in exactly the same location in the Composite Image as it was in the original Tile Image that we saw at the very beginning of this document. The def_Picture image.



Writing the Computer Code

So now that we fully understand the manual method for producing a high quality glow border we can start to write the computer code with the various functions needed to automate the process. At first I really didn't know where to begin writing the code. How do we make a function to do a parallel offset of a string of pixels to draw a glow mask? Drawing the glow mask itself seemed to be the most difficult part of the process to produce a colored glow border. So I started with that.

The whole thing started with a lot of hours spent doing on-line research about how to offset the parallel lines of the glow border. I didn't find much information about that but I did find one small tidbit that turned out to be the key to the entire drawing process. During my research phase my journey to discover the secrets of drawing a glow mask ended up in a peculiar place. I found a library that had a function to Detect Contours of an image. It was called CV Library. I didn't really look at the code because it would not apply to the specific task of drawing an icon glow mask. But it was interesting that documentation said the result was returned in Freeman Chain Code.

So I focused my research on that because that seemed to be perfect for the type of program that I wanted to write to draw glow masks. After a short time I found some information related to the Chain Code and how to use it to describe all the pixels around the perimeter of an image. Actually, it is more of a roadmap that describes how to get from the start point to every pixel

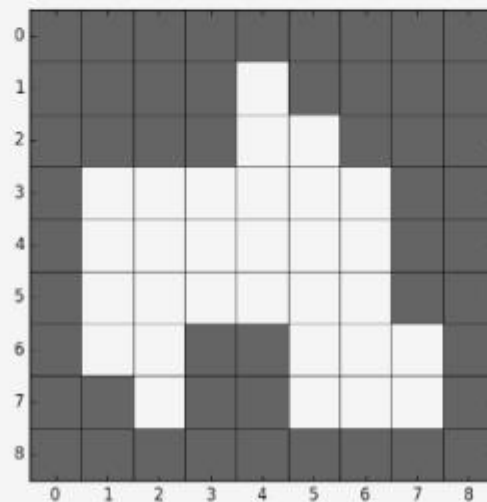
around the perimeter of the image till we arrive back at the start point which becomes the end point of the chain code. It seemed like an elegant solution to a very complex problem. Here is some information about Chain Code:

https://ojskrede.github.io/inf4300/notes/week_04/

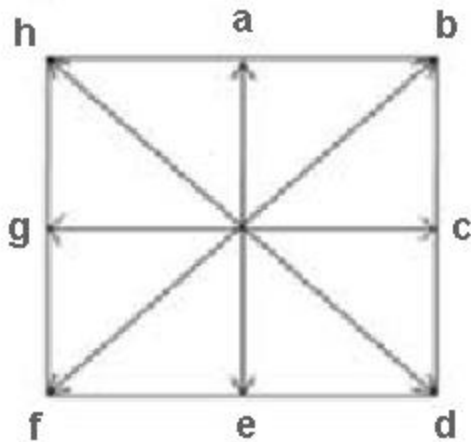


Figure 1: 4-connected and 8-connected neighbourhoods.

I will explain the different terms through examples, and the example figure we will use is a simple binary 9×9 image, where white is the color of the object which boundary we shall describe.



In the Chain Code Diagram we can see the upper two schematics. The one on the left is for the 4-connected chain code. That is the one I started with. But during the process of writing the code I quickly discovered that it was not adequate to describe the oblique angles for some of the icon images. So I switched to using the next schematic called the 8-connected chain code which uses ascii characters to denote the direction of travel along the perimeter of the image.



You may think of the 8-connected chain code as a secret numeric code that represents the direction of travel. We start at the Control Point (start point) whose x, y coordinates are known. The Control Point for the Chain Code is usually the point (pixel) closest to the Top and Left of the image. It's similar to the manual method in that sense. We need a known start point to begin the journey around the perimeter of the RGB Image. We transition from pixel to pixel along the perimeter in a clockwise direction. When we arrive at each of the pixel locations we perform a search pattern to look for and identify background pixels. Now for some more definitions to help explain the chain code method. For a pixel to be a "Perimeter Pixel" it must be adjacent to at least one boundary pixel which is part of the fully transparent background of the icon image. If the current pixel is a vertex pixel at the intersection of two angles or two line segments, then it could be adjacent to more than one background pixel. So the perimeter pixels are called "Whole Pixels" or just "Pixels". The background pixels are called "Non-Pixels". That's the terminology.

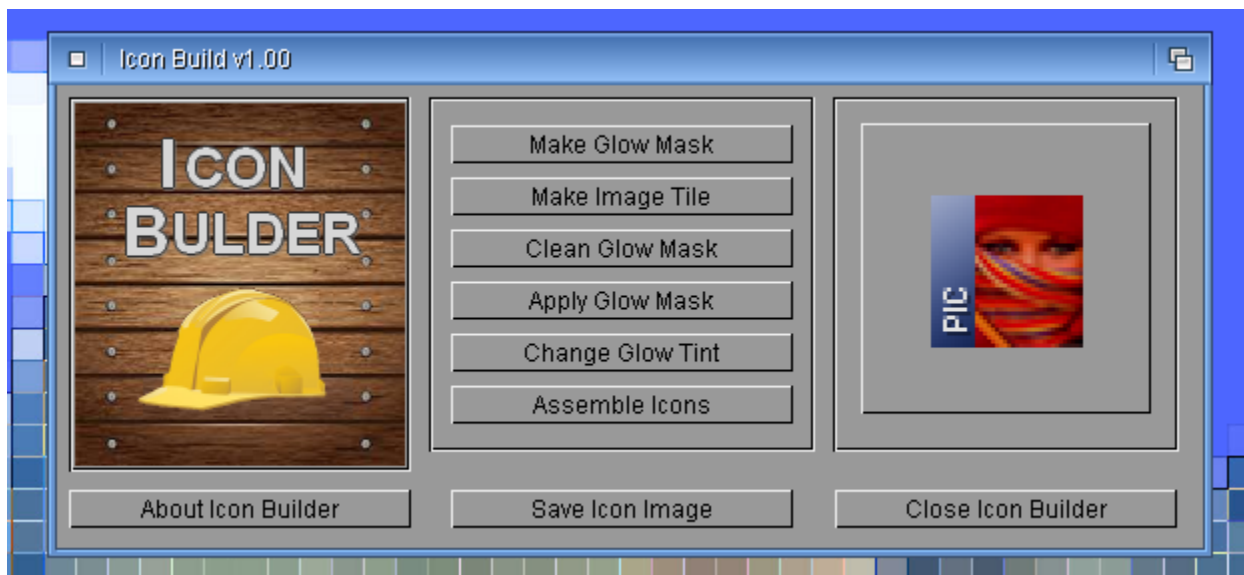
So let's start writing some code. I chose to use a modified chain code to make things easier. It's easier for me that is. The computer is indifferent to the method I used. Instead of using numeric values for 8-connected chain code to indicate the direction of travel I instead used ASCII code. We start at the top with a letter code rather than zero. If we think of it as locations on a clock maybe that will be easier to visualize. At 12 o'clock noon we have the letter 'a'. At 2 o'clock is the letter 'b' which represents a 45 degree angle. At 3 o'clock in the chain code diagram we have the letter 'c'. And that is followed by "d", 'e', 'f', 'g', 'h' and back again to 'a'. So now the chain code to describe the perimeter of the shape for the def_Picture icon image sample.

The chain code is stored in the program as a simple byte array which consists mostly of several two-byte direction-value pairs. For the rectangular glow mask for the def_Picture the chain code only consists of four two-byte direction-value pairs. The byte pairs for def_Picture are as follows:

'c', 37, 'e', 37, 'g', 37, 'a', 37

The chain code also has two other critical components other than the byte codes. The int chaincount and int control_x, control_y values are also used. The chaincount in this case is the integer value 4 that denotes 4 pairs of direction-value bytes. Each of the letters represents the direction of travel from the Control Point in a clockwise direction. For the computer code we will need several smaller functions to accomplish the task of drawing the glow mask. We can draw a

glow mask in about 20-30 minutes in the paint program using manual methods. But the benefit is that the computer can generate the same glow mask in a fraction of that time. Although using Graphics Library to draw each colored pixel directly to the RasterPort is a really slow process in computer terms, it's really fast in human terms. It may take about 200 ms to draw each color band for the glow mask which takes about 600 ms to complete the entire glow mask. It is so slow in computer terms that you can actually see it drawing the pixels to the RasterPort. It is similar to the old Star Trek cartoons when the captain says "Shields Up" and on the screen we see little dots slowly appearing around the perimeter of the spaceship. We can see the pixels being drawn all the way around the perimeter of the icon image, one color band at a time. But in about a second the glow mask has been completed. It's much faster than a human could draw.



This is the Icon Builder Application which I am developing to help me to produce the glow borders for icon images. This is a screenshot of the "before" condition which shows the Tile Image which is displayed as a 2x enlargement of the original image in the display area on the right. The button at the top "Make Glow Mask" calls the `Make_Glow_Mask` function in the code. That starts the process of drawing the Glow Mask using the three colors: white, gold, orange. The "after" screenshot shows the resulting glow mask that was drawn around the icon image. The reason the image is displayed at 2x (2 times larger) is that it makes it easier to see the individual pixels of the glow border itself, although you may need to squint a little bit still.



As for the Icon Builder Application it consists of a logo image on the left, a series of 'action' buttons in the middle for icon image production as well as three buttons along the bottom. Located on the right side is a 'preview' area where the current icon image is displayed. At first, when I was designing the user interface I noticed that the icon image when shown at full size appeared too small to see the outside border. From a previous project I had code that could increase the size of the image by 2x or 3x if needed using a 'bitmap_to_region' function that mapped each pixel in the original image to a region of 4x4, 6x6 or 8x8 pixels in the rasterport.

Being able to enlarge the image was only part of the issue. I needed a way to display an image in such a way that it was crisp and clear and easy to see. The original 'bitmap_to_region' used an optional 'grid' to display icon image which was really just an extra space between the pixel groups. I chose not to use the grid but instead to display a clear image at 2x magnification. Lastly, I need a way to draw the image to the rasterport of the window. In the past I had used CyberGraphX Library to draw to a window using WritePixelFormat and WritePixelFormatAlpha. However, Icon Builder presented an interesting opportunity to test what I call 'direct draw' method which consists of using Graphics Library and Pens to draw directly to the rasterport. Consequently this method allows complete control of the drawing procedure, although using Graphics Library calls to draw to the window isn't recommended for larger images. It's slow!!

Remember from the manual method the next step is to Clean the Glow Mask to remove the RGB Image to be replaced by the white pixels in the middle. That is what the third button does. The second button prepares the second icon image by making all the background pixels the same so that they are completely transparent. That makes selection of the RGB Image easier. Some graphics programs when detecting contour edges use a Binary Mask which is just Black and White pixels. White pixels are the background. Black pixels are the Image in the middle.

Getting back to the code now, the Make_Glow_Mask function only has one job. It does some calculations to find the int srce_x, and int srce_y coordinates which is the offset to the upper left of the display area in the RasterPort for the second raised panel where the icon image is shown. The srce_x and srce_y is the origin point for drawing in the the right-most display area. The

point itself is located at the top left of the display area. These x, y coordinates are used to help find the coordinates of the pixels that will be drawn to the RasterPort during the process of drawing the glow mask. So `Make_Glow_Mask` sends the `srce_x`, `srce_y` values to another function `Draw_Glow_Mask`. This is the main function where the chain code is used to help draw the glow mask. At this point in time the chain code is hard-coded into the program which makes it really inconvenient because only the icon image related to that specific code can be used. At some point in the near future I will write the code to load the chain code from a text file that will be converted into the byte array by parsing the text in the file. `Draw_Glow_Mask` has many responsibilities. The chain code such as for `def_Picture` is located in this function. It's just a byte array of two-byte pairs. It defines the graphics path around the perimeter of the icon image.

The way that `Draw_Glow_Mask` works is that it only has four main types of structures to define the shape. There is Linear Structures such as line segments, and from Geometry you may recall that a Single Point which is a transition from one pixel to another in a straight line is also linear. But we also have Oblique Structures which can be composed of many pixels in an angled line or just a Single Oblique Pixel which is really a transition from one pixel to another in an oblique direction which represents a 45 degree angle. So `Draw_Glow_Mask` sets up a for-loop for the number of items in `chaincount` (count of 2-byte pairs). It reads the ascii letter for the first 2-byte pair. The computer returns the value as a number such as '98' or '99' to indicate travel direction. In this case it reads 'c'. Then it reads 37 which is the number of pixels in the line segment minus one. We don't count the first pixel in the line, we only count the transitions from pixel to pixel.

To make the drawing process easier the current color is determined by the value of the letter 'l'. We set up another for-loop such as `for(i=0; i<3; i++)` so that we process the chain code three times, once for each color of the glow border bands. The current color is determined by the conditional `if(i == 0) color = "white"; if(i == 1) color = "gold"; and if(i == 2) color = orange";` The function to draw line segments inside the for-loop will perform a search pattern as it processes each of the pixels on the perimeter of the image described by the chain code. It calls the next function `Check_Pixel` and sends it the current color and the x, y coordinate of the pixel to check. It is checking the alpha value to see if it is 0 or 255. Remember that the Tile Image only has fully transparent pixels (0) or fully opaque pixels (255). When `Check_Pixel` identifies a Non-Pixel with an alpha value of 0 in the byte array for the Tile Image it calls the function `Write_Mask` which only has one responsibility. It sends it the current color, `srce_x`, `srce_y`, x, y values. `Write_Mask` will convert the color into a ULONG value to use `WriteRGBPixel` to draw the glow mask color to the RasterPort at the specified coordinates provided. It draws color bands to the RasterPort.

The search pattern used with `Check_Pixel` will automatically fill in the gaps at the end of each of the line segments with the correct color. For each perimeter pixel we search in four directions. We search the pixel directly above the current pixel, the one directly to the right, directly below, and directly the left of it. This search pattern for `Check_Pixel` guarantees that there is adequate coverage for the background pixels that are adjacent to the perimeter of the icon image. It's ok if we search the same perimeter pixel more than once if there's an overlap. There's no harm in it.

There were some problems with the code that needed to be overcome. For a while I had no way to save the icon image with the nice new glow mask. Because there are so many sub-functions involved in drawing the glow mask I couldn't send a pixel buffer that is a byte array of the original image to all the various functions. It was just too problematic and it wouldn't work. But the method that did work was that `Write_Mask` could draw directly to the RasterPort itself.

At first I didn't have any way of deciding which colored pixels had been drawn to the icon image for each of the preceding color bands. So when we reached the "orange" color it would simply overwrite all the "gold" and "white" pixels that we had just drawn. So I needed a way to find out which colored pixels had already been drawn. The byte array for the original Tile Image won't work for this purpose because it only has the RGB Image and transparent pixels. So I found a way to "Sample the RasterPort" by using ReadRGBPixel which returns a ULONG Pen Color. I used PutUint32 which is a macro to convert the ULONG into a 4-byte pixel32 byte array. Then I could simply compare the values of the original pixel and the RasterPort pixel. If the colors are different is it "white", "gold" or "orange". If so we copy the pixel color directly back to the byte array that represents the Tile Image. In that way we assemble a byte array with the new glow mask colors. So Check_Pixel can Sample the RasterPort to find if a color band pixel has been placed previously so it won't be overwritten. We simply skip the pixels that have been drawn.

Now we have a means of saving the icon image with the new glow mask. Save_Icon_Image is the function associated with the middle button on the bottom of Icon Builder. It will read the original Tile Image to get a byte array for the pixels. It will then loop through each pixel and compare it to the corresponding pixel in the RasterPort. Again, if the color is different and if the color is one of the mask colors it copies it back to the pixel buffer to save as a new icon image.

The Make_Glow_Mask function and all the other sub-functions work really well. Depending on the accuracy of the chain code for each image increasingly more complex glow masks can be drawn using the computer code. But there is still much work to be done for the rest of the code to make Glow Borders by applying the Blur Effect (Gaussian Blur radius2) and the Composite.

The sample code from Icon Builder associated with Drawing the Glow Mask are provided for you to see all the different processes needed to draw the glow mask using the mask colors. Most of the code relies on what I refer to as X-Y Transition Matrices. The transition matrix is just an easy way to transition from one pixel to another using its cartesian coordinates. If we are moving from left to right the y value is the same. Only the x value changes for each new pixel. So the transition matrix for that operation is as follows: if(direction == 'c') x += 1; That's it.

Alpha Blending

I thought it was worth mentioning the journey that I took to get the icon images to display in the rasterport as 32bit images. Using the 'direct draw' method requires using ULONG values for pens and ObtainBestPen function to get Pens (colors) to draw with using Graphics Library calls. Two Macros PutUint32 and GetUint32 were used to convert 4-byte arrays into ULONG values and back again for the Pen values. At first I noticed that one of the values were always zero but 32bit RTG screens are supposed to have four bytes not just three. Three bytes represent RGB.

As a result I couldn't quite understand why my 32bit Icon Images were being displayed with a black background and RGB image in the middle. Wasn't the icon background supposed to be transparent? So I dug deeply into the Icon Library code to figure out what was missing but I didn't find any useful information about drawing. Then I thought about using CyberGraphX to draw the images and I had an idea. I looked at the CyberGraphX code then Graphics Library code to try to understand how the drawing to the rasterport worked. Strangely, Graphics calls from Graphics Library ended in the same drawing procedure as for CyberGraphX. The two sets of functions were linked. But, deep in the drawing code I noticed that the Pen values were RGB.

Importantly, the drawing functions in Graphics Library (and CyberGraphX) were using Alpha Blending to blend the icon image with the background color. So I wrote a new Alpha Blending function to do the same thing for my program. It took several attempts at drawing to get it right. Eventually, I achieved success! Now the enlarged icon images were using alpha blending also. Displayed on the left is the original icon image in Magellan at actual size. On the right side is the enlarged (2x) icon image drawn in the rasterport of Icon Builder using Alpha Blending to display.



Future Development

When the program code gets more advanced it may be possible for 'Make_Glow_Mask' to automatically generate the chain code which is used internally to send to 'Draw_Glow_Mask' without reading the chain code byte array in from a text file. That would allow making a glow mask more seamless with less interaction. That makes the Icon Builder more user friendly. Automatically detecting the perimeter of the icon image might be similar to the 'check pixel' method mentioned earlier, deciding which is a 'Pixel' (on the perimeter) and which pixel is a 'Non-Pixel' (on the transparent background). The 'Detect_Contour' function would be useful.

For Change Glow Tint an easy requester will be used with various buttons for various colors. Likewise, Assemble Icons will use an easy requester with three buttons at the bottom of the window: PNG, OS4 and Close. The user must decide which style of icons to assemble from the PNG images in the list. Initially, Assemble Icons will read an IconsList text file where the images are arranged in pairs with the extensions _1.png and _2.png to allow for easier assembly. When the function gets more advanced it would be better to use a list view with buttons at the bottom instead of reading from the text file to get names of icon image pairs. Buttons at the bottom of the List View would be "Add Files", "Add Folder" and "Build Icons". Using the List View in such a way allows the user to add files for two icon images for a single icon or a series of icon image pairs or just simply add an entire folder of icon image pairs.