Computing Assignment 3:

Default pic

Part1-Brightness and Contrast

Logic: The logic behind the programming for this section was relatively straightforward. Each pixel is stored as a word in memory with the first channel being empty, ie. the color 0x00FFFFFF has FF-255 brightness for the red, green and blue components of the pixel.

Changing brightness is simply adding a constant value to each pixel channel. If we don't add it to the channels separately it may cause overflow and end up with the wrong result. When any channel goes over 255 we set it to 255 to solve overflow. However, this approach does eventually cause the colors to discolour and turn to white, but I felt it made sense that if you brighten too much it turns white and regardless of the change in color ratios the absolute luminance does still increase. Negative brightness does the opposite and we simply need a check to revert the value back to zero once it goes under

Pic with only brightness at 20

Extra credit: Made sure no overflow and worked for negative brightness

Pseudocode:

```
For(pixel p:pixelarray)
{
Int a = getPixRed(p)+brightness;
Int b = getPixGreen(p)+brightness;
Int c = getPixBlue(p)+brightness;
If(a>255)
        A=255
Else if (a<0)
        A=0
…repeat for all three channels
compositeColors(a,b,c);
```

}

Changing contrast using the method specified was similarly easy. This approach actually increases contrast by emphasizing the relative difference by changing the absolute amount. This ends up actually making the picture brighter overall. A smarter approach would have been to use unsharp masking with a threshold of 0, however this required far greater effort and I rathered to spent my time improving the extra credit of my choice. Implemented a check to ensure the alpha value wasn't too high as to cause overflow.

Pic with only contrast at 20

Extra credit: Dealt with overflow due to large alpha values

Pseudocode:

Same as brightness but + replaced with * and brightness replaced by constrasrt values and each channel /16 at end.

Part2-Motion Blur

Logic: This technique was more complicated, particularly as it dealt with diagonals and the values of pixels other than itself. This causes problems with image depth, a common problem in picture modification with no set solution. Also as it required the average of a number of values I felt I had to implement a better division algorithm.

This program allows you to specify a radius in main and it gets the average of each pixel down right diagonally that is >= radius pixels away. This means a radius of 0 does nothing.

Extra credit: Made proper motion blur by copying to alternate memory, dealt with out of bounds exceptions, works for almost any radius, better division for faster performance

Pseudocode:

For each pixel

{

Sum+=pixel channel value(for each channel different sum)

Count++

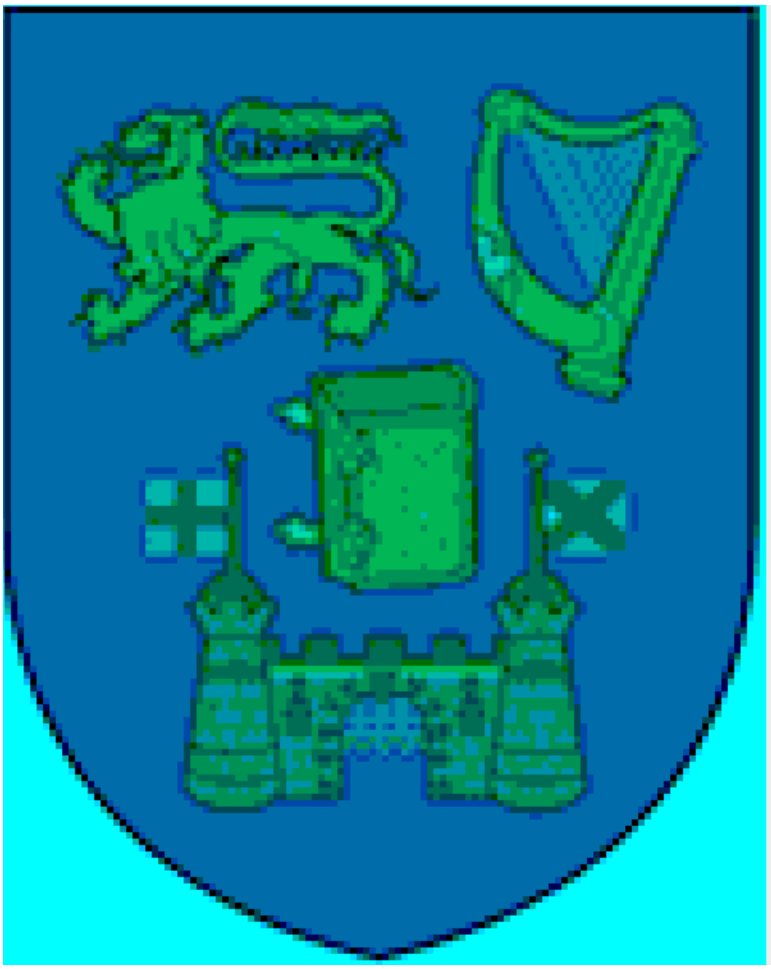
```
Tempcount =0
If(upleft==pixel && tempcount<radius)
{
        Tempcount++
        Sum+=pixel channel value(for each channel different sum)
}
Count+=tempcount;
Repreat in bottomright direction
Divide(sum,count)(for each channel)
compositeColors(resultred,result,green,result,blue);
store pixel in copy address
}
Transfer from copy address
```

Part3-Extra Credit

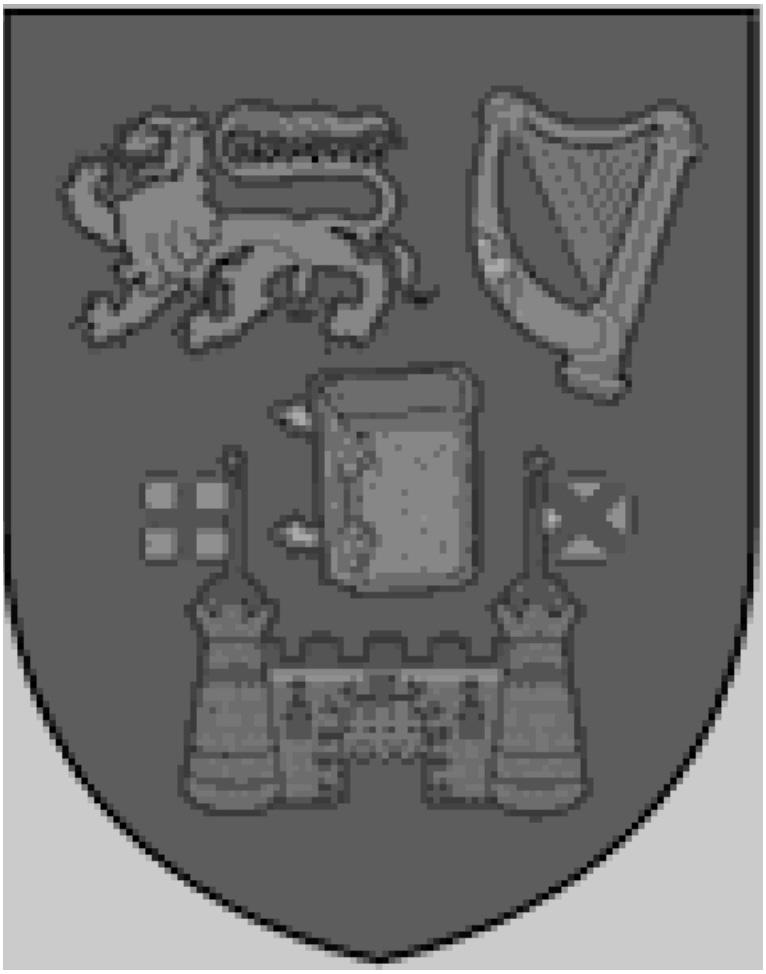
This is a multi-stepped process to apply edge detection to the crest.

Pre-processing:

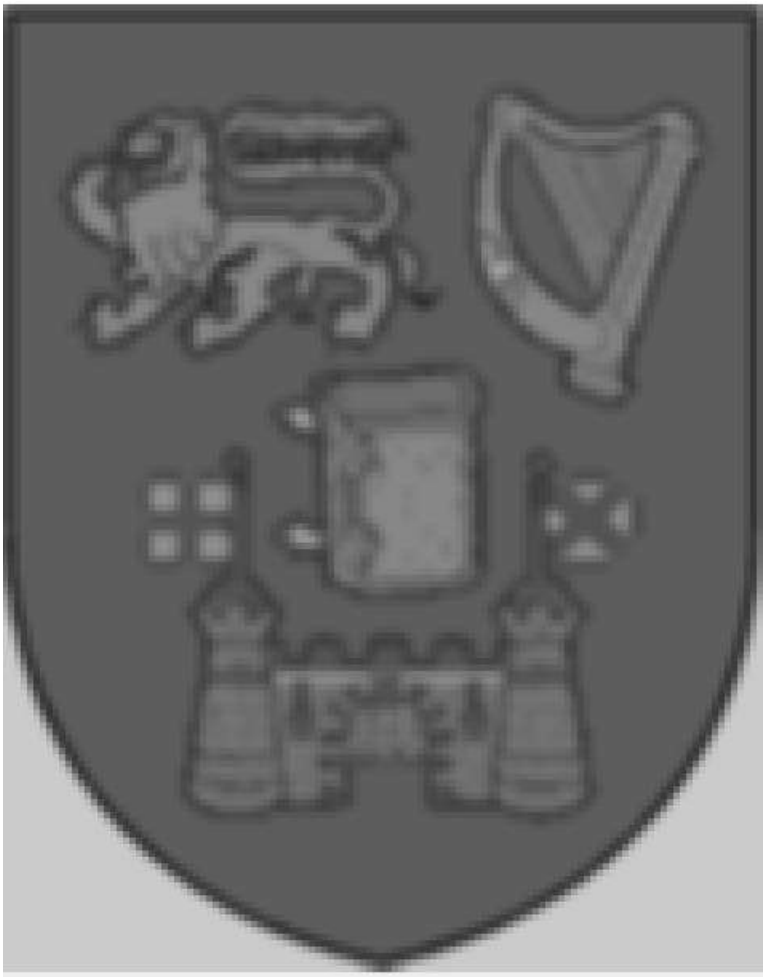
Filter-Allowed the user to use a filter to filter out specific channels or dampen them. This is usefull if there's a lot of noise in one specific channel or in this case where the red channel was used to shade the picture. Removing it allowed me to get more defined edges.


red filter removed red channel

GreyScale-It’s a common step to convert an image to greyscale first before using edge detection as this allows you to detect abrupt changes in luminance rather than hue. Rather than simply computing a simple average of the three channels, I weighted each channel based on their perceived luminance.



GaussianBlur-Noise reduction filters are used before edge detection to remove unwanted noise. While Gaussian blur may not be a particularly good filter for noise, it is relatively computationally inexpensive. Gaussian blur is achieved through convolution with a 3*3 matrix of values(hence each pixel is blurred with respect to it's neighbours) .When a pixel couldn't be found due to being out of bounds a neighbouring pixel was cloned to substitute it. Gaussian blurs smooth out high frequencies meaning it also has the unwanted effect of removing edges.

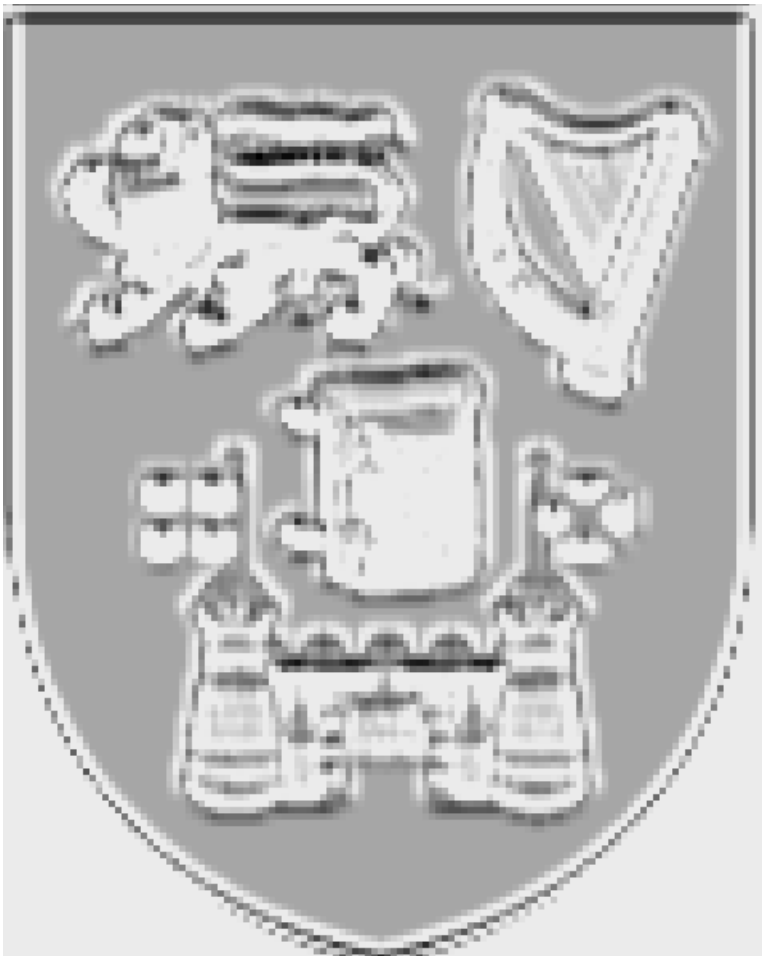


Processing:

Sobel Edge Detection-A simple kernel which when convolved with the pixel gives the relative strength of it's edge. This algorithm calculates the strength of the edge in both x and y dimensions then gets the hypotenuse/magnitude of the vector. However as square root in assembly is computationally time consuming I went with the approximation of adding the absolute value of each.

Post-processing:

Normalising: This step attempts to allow the mapping of luminance to different ranges without changing any relative difference in luminance ie.it only changes the absolute difference between pixels not relative. This is usefull for emphasize edges and give a result that uses entire luminance spectrum. I mapped it to 0-255.



Thresholding and linking: This step allows us to filter out edges to a specified strength and also attempts to link edges using a simple algorithm. The user specifies a lower luminance value and a higher luminance value. If the pixel is below lower threshold it is detected as an edge and if it is above higher it is not an edge. However, if a pixel is between the two values and next to a pixel below the lower value it is also detected as an edge. Due to this design having wider ranges causes thicker borders and higher lower threshold causes more noise being detected. However, this step is very usefull in obtaining thinner more defined edges as well as less edges.

Lower bound of 160 and max of 180