Cs1022 – Assignment #2

2-Dimensional Arrays – Image Manipulation

Part 1 – Brightness and Contrast

In this stage I was required to design, write and test an ARM Assembly language subroutine that will adjust the brightness and contrast of the TCD crest image stored in memory.

<u>Method</u>:

I went about this by first breaking down the larger problem into a handful of smaller problems. These smaller problems included; loading a pixel at row (i, j) in memory, storing a pixel at row (i, j) in memory, getting the red component of a pixel, getting the green component of a pixel and getting the blue component of a pixel. So, the basic method for adjusting the brightness and contrast of any given pixel is to load a pixel from row (i, j) in memory, get it's RGB components, apply the formula to each component, add the components back together and store that pixel back in memory at row (i, j). The subroutines that I used can be seen below.

<u>getpixel subroutine:</u>

```
getpixel() {
R3  = pixelAddr;
pixelAddr = row * picWidth;
pixelAddr += column;
pixelAddr *= 4;
pixelAddr += picAddr;
memory.get[picAddr]
}
```

<u>Interface:</u>

```
; getpixel subroutine
; gets a single pixel from a picture
; Parameters: R1: Row
;    R2: Column
;    R4: Start address of picture
;    R6: Picture Width
; Returns: R0: Pixel
```

As can be seen from the code to the left, I simply found the address of the pixel I was looking for, and loaded it into R3. This was done by multiplying the row by the length of the row, adding the column and multiplying by 4. The sendpixel subroutine was done in the same manner only I was storing the pixel to R0 instead of loading it into R3.

<u>getred subroutine</u>:

```
getred() {
pixel = pixel AND NOT(0xFFFFFF00);
redComponent = pixel;
}
```

<u>Interface:</u>

```
; getred subroutine
; Gets the red components of a pixel.
; Parameters: R1: Pixel value
; Returns: R0: Red component.
```

As can be seen from the code above I simply used a mask with the AND operation to isolate the component I wanted. The same was done for the green and blue components only I shifted right by 8 and 16 respectively. This was done so that I could apply the formula to each component.

The formula was easily applied to each pixel as seen below, with R1 being the brightness and R2 being the contrast.

```
redComponent *= contrast;

redComponent /= 16;

redComponent += brightness;
```

After multiplying by the contrast, I also checked to see if the component of the pixel was less than 0, and if it was I let it equal 0. After I added the brightness I also checked to see if the component of the pixel was less than 0 or greater than 255. If it was less than 0 I let it equal 0 and If it was greater than 255 I let it equal 255.

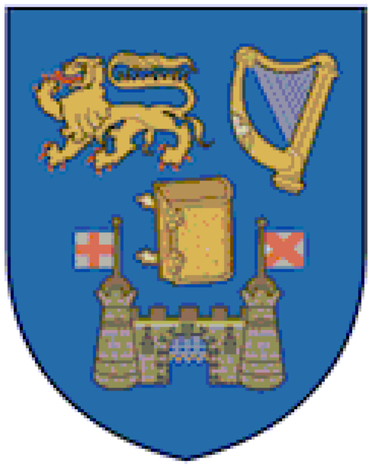
The combination of these subroutines gave me the desired result.

<u>Sample inputs:</u>

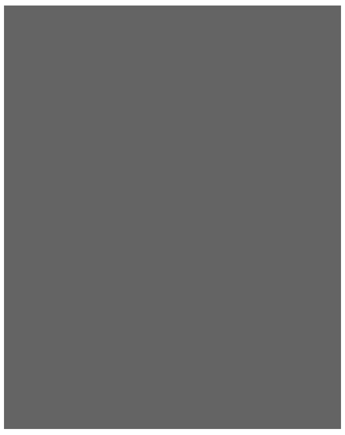
(Pictures can be seen below)

	Brightness	Contrast	Current Pixel	Next Pixel
1)	0	30	0x003692FF	0x0065FFFF
2)	100	-10	0x00FFFFFF	0x00646464
3)	-100	50	0x002010FF	0x000000FF
4)	200	16	0x006020F0	0x00FFE8FF
5)	-150	100	0x0082F456	0x00FFFFFF

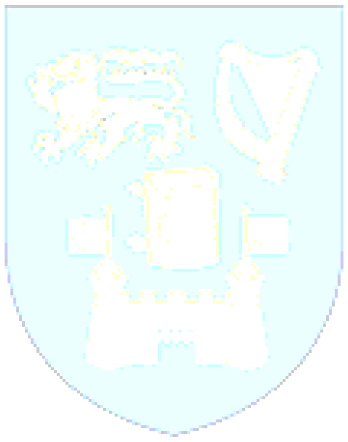
Original Image:



2)



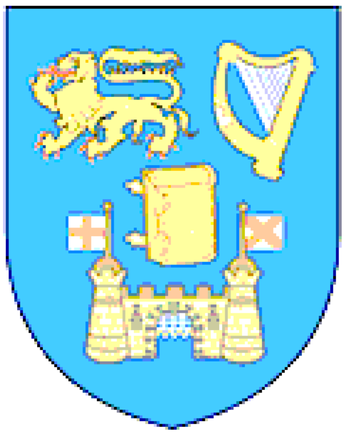
4)

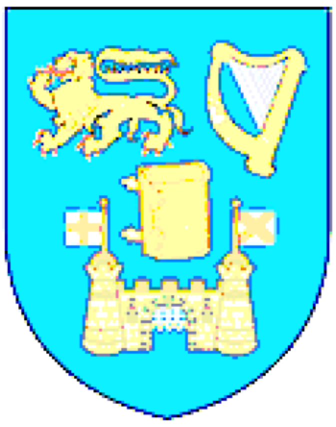


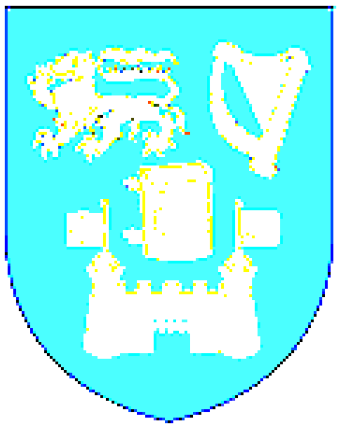
3)

5)

1)







Part 2 – Motion Blur

In this stage I was asked to design, write and test an ARM Assembly Language subroutine that will apply a motion blur effect parallel to a diagonal line from the top-left to the bottom-right of the image.

Method:

I went about this again by splitting a big problem into smaller problems. The steps I used to blur the image are as follows. I went through every pixel in the image. For every pixel in the image I took the average colours of the diagonal pixels including the centre pixel and let the pixel equal to that colour. To do that I had to get the RGB components of each individual pixel, add them to a running total, and then divide them to get the average.

I could use all the same subroutines from Part 1 – Brightness and Contrast, and only had to create a divide and getAverage subroutine. The divide subroutine can be seen below

Divide Subroutine

```
divide() {
  while(num <= divisor)
    {
      num -= divisor;
      quotient ++;
    }
    R0 = quotient;
}
```

Interface

```
; divide subroutine
; takes two numbers and divides them
; parameters: R1: number to be divided
;     R2: divisor
; returns: R0: Quotient
```

To get a pixel diagonal to the centre pixel I simply had to add or subtract 1 to the row and column of the centre pixel. Below is the code I used to get the total RGB components of the pixels to the top-left of the centre pixel.

```
while(radius != 0) {
  radius--;
  row--;
  R1 = row;
  column--;
  R2 = column;
  getpixel();
  R1 = pixel;
  getred();
  totalRedComponent += redComponent;
  getgreen();
  totalGreenComponent += greenComponent;
  getblue();
  totalBlueComponent += blueComponent;
}
```

I then performed the same operation, only adding 1 to the row and column instead of taking 1 away. I then had to find the average of these values, Which can be seen below.

```
R2 = divisor;
R1 = totalRedComponent;
Divide();
R3 = redComponent;
R1 = totalGreenComponent;
Divide();
R4 = greenComponent;
R1 = blueComponent;
Divide();
R5 = blueComponent;
greenComponent *= 0x100;
blueComponent *= 0x10000;
pixel = redComponent + greenComponent;
pixel += blueComponent;
R0 = pixel;
```

The combination of these methods allowed me to find the average colour of a diagonal line of pixels.

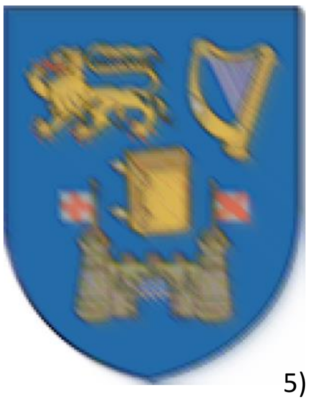
<u>Sample Inputs</u>

(Pictures can be seen below)

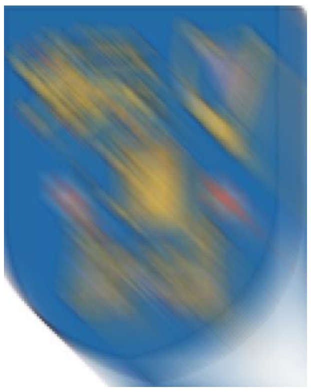
	Radius	First Pixel	Average
1)	0	0x00DADAFF	0x00DADAFF
2)	5	0x00DADAFF	0x00606D8D
3)	20	0x00DADAFF	0x00346DA2
4)	-10	0x00DADAFF	0x00DADAFF
5)	100	0x00DADAFF	0x00697976

1)

2)

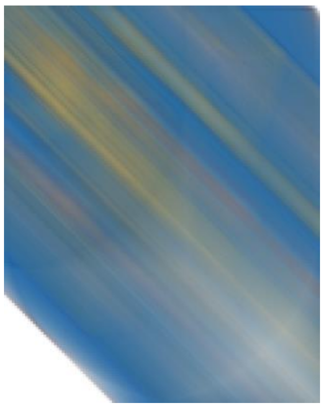


3)



4)

5)



Part 3 – Bonus Effect

In this part I was asked to design, write and test a subroutine that will apply an effect of my choice to the TCD crest image stored in memory. For this I decided to design a subroutine that would apply a convolution matrix to each pixel. I also wanted my program to be a bit more dynamic, so I stored multiple matrices in memory, and picked one based on user input. The matrices I stored are convolution matrices for edge detection, emboss, sharpening, vertical edge detection and horizontal edge detection.

<u>Method</u>:

To start off I first had to find out how to apply a convolution matrix to a pixel. I think the process is best explained based on the example below.

Piece of image in memory:

72	50	32
0	36	23
92	0	82

Calculation for centre pixel:

72(-1) + 50(-1) + 32(-1) + 0(-1) + 36(8) + 23(-1) + 92(-1) + 0(-1) + 82(-1) = -63

Since -63 is less than 0 we let it equal to 0.

This process is repeated for every other pixel.

Edge Detection Convolution Matrix:

-1	-1	-1
-1	8	-1
-1	-1	-1

Result:

255	237	147
0	0	0
255	0	255

I achieved this by using the majority of subroutines used in parts 1 and 2. I had to create the following subroutines, applyMatrix() and clonePicture(). The purpose of cloning the picture was so I could apply the convolution matrix to each pixel of the picture, and store the resultant colour in the original image.

My approach for cloning the picture can be seen below:

```
R6 = multiplier;
R3 = picAddr;
R7 = pixelAmount;
clonePicAddr = multiplier * pixelAmount;
clonePicAddr += pixelAmount;
clonePicAddr += padding;
R0 = clonePicAddr;
R4 = count;

while(count < pixelAmount) {
   R5 = memory.get[pixAddr];
   Memory.get[clonePicAddr] = R5;
   picAddr += 4;
   clonePicAddr += 4;
   count++;
}
```

```
; clonepicture subroutine

; takes an image and makes a copy of it in  memory

; parameters: R1 = start address of original copy

; R2 = size of original copy

; returns: R0: start address of clone
```

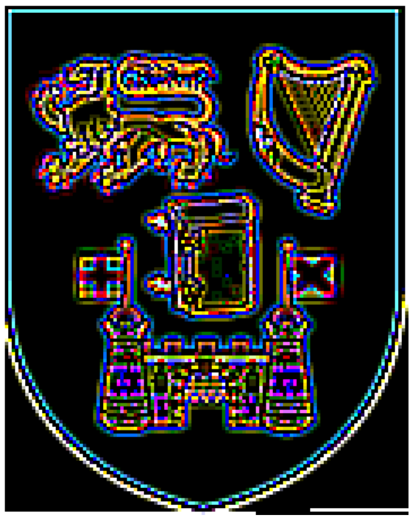
After cloning the picture I then applied the matrix. I first got the start address from the relative pixel for where I would be working. I made sure that if the address of the pixel I was trying to get was out of bounds, I simply added a black pixel. If the address of the pixel was in bounds, I got the pixel, split it up into it's RGB components, multiplied the RGB components by the associated matrix index and then added them to a running total of RGB components. I then checked to see if all of the components where in the range of 0-255, if any were over 255, I set them to 255, if any were below 0, I set them to 0. I then stitched the components back together and stored the pixel in the relative position in the original copy. A summary of this in pseudo code can be seen below.

```
While(rowCount <= matrixDimension) {
   R1 = row;
   R2 = column;
   R6 = picWidth;
   getPixel();
   R1 = pixel;
   R4 = memory.get[matrixAddr];
   getred();
   redComponent *= R4;
   totalRedComponent += redComponent;
   greenComponent *= R4;
   totalGreenComponent += greenComponent;
   blueComponent *= R4;
   totalBlueComponent += blueComponent;
   matrixAddr += 4;
   column++;
   columnCount++;
   if(columnCount >= matrixDimension) {
      row++;
      rowCount++;
```

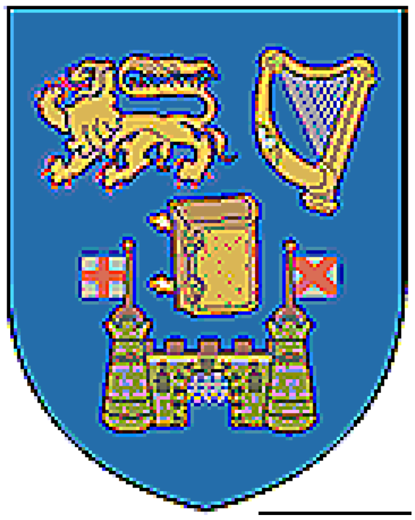
```
      Column -= 3;
      columnCount = 0;
   }
}
If(totalRedComponent <= 0) {
   totalRedComponent = 0;
}
If(totalRedComponent >= 255) {
   totalRedComponent = 255;
}
Same for green…
Same for blue…
greenComponent *= 0x100;
blueComponent *= 0x10000;
pixel = redComponent + greenComponent;
pixel += blueComponent

(The full pseudo code can be seen in the .s
file.)
```

Sample Outputs:

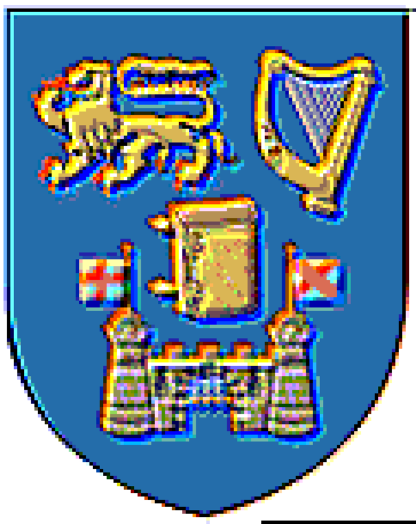
Edge Detection:



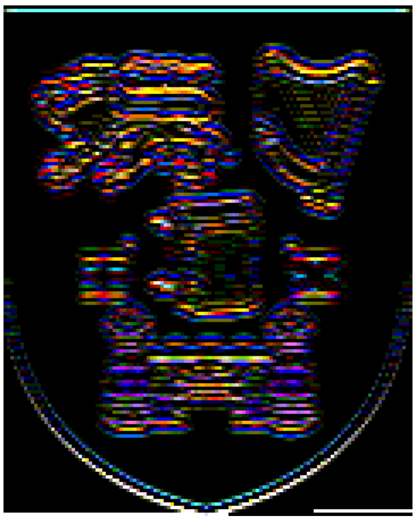
Sharpen:



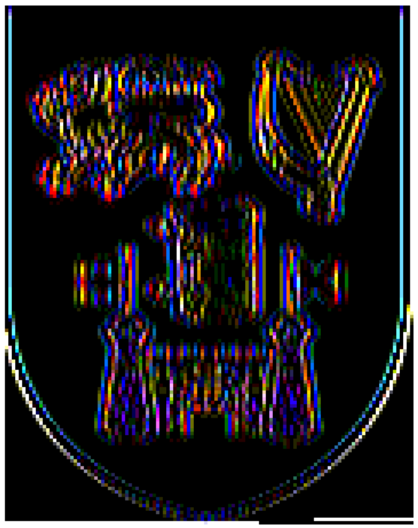
Emboss:



Horizontal Edge Detection:



Vertical Edge Detection:



45 Edge Detection: