Computing Assignment #1 Image Manipulation

16324334

3/21/2017

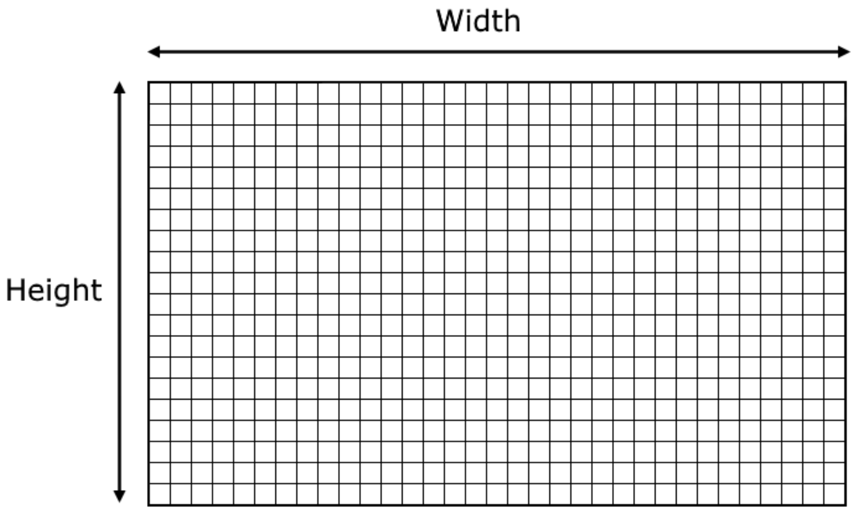
Part one: Brightness and Contrast

For the first part of the assignment, it was instructed we create a subroutine that allowed the brightness and contrast of the image to be altered. To do this, a formula on how to change the brightness and contrast of a pixel was provided.

$$R'_{i,j} = \left(\frac{R_{i,j} \times \alpha}{16}\right) + \beta$$
$$G'_{i,j} = \left(\frac{G_{i,j} \times \alpha}{16}\right) + \beta$$
$$B'_{i,j} = \left(\frac{B_{i,j} \times \alpha}{16}\right) + \beta$$

To follow this formula a pixel had to be multiplied by the contrast value α, then diving by 16, and finally adding the brightness value β.

Before this could be even considered, a way of getting each pixel would be needed. To accomplish this, the width and the height were multiplied by each other and then subsequently multiplied by the four as each pixel corresponds to one byte in memory.



```
for (eachRow in Image)
{
    for (eachPixel in eachRow)
    {
        for (eachColour in eachPixel)
        {
        eachColour *= contrast/16
        eachColour += brightnes
        }
    }
}
```

By multiplying the height and the width the loop can go through the entire image adjusting each pixel as needed.

Inside this loop was another loop to go through each colour (Red, Green and Blue) and adjusted that value of that pixels colour value.

At first it might seem that a division method is needed when applying the formula, however using a logical shift of four accomplishes the same task and uses significantly less time to compute.

The pixel was set again at the end of the most inside loop. A more complex method of doing this is by shifting the R, G and B values across and setting all three pixels at the same time. This way would most likely be more efficient but for part one, storing just the individual colours seemed better for readability.

An issue that came about was when adding high brightness values or changing the contrast by a lot, the pixel would be larger than a byte. To prevent this happening in any case the pixel was compared to the byte value and if it was too large it was just set to the largest possible value, which was 255.

The same was done for values less than zero, by checking if the value was less than it, and if it was the pixel value would then be set to 0.

Another thing that needed to be checked was if the contrast value was less than zero as if it was it would result in the contrast value being set to the lowest amount suitable, which was zero.

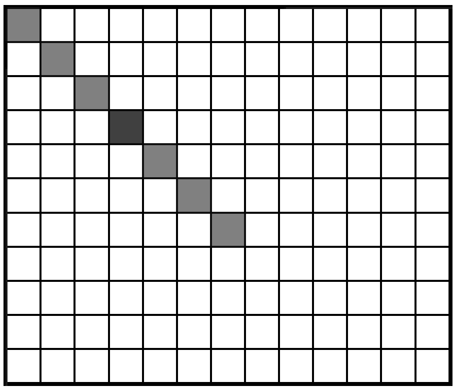
Lastly, if the contrast and the brightness level would not affect the image in any way the entire subroutine would be bypassed saving it having to calculate redundant values.

Part two: Motion Blur

The second part of the assignment is more difficult as it requires multiple pixels to be processed to simulate effect of motion blur.

The way in which a motion blur can be applied is by going through every pixel and then getting the average of those pixels around it.

By doing this you can make an image look as if it's moving a lot, or a little depending on the radius of pixels chosen.

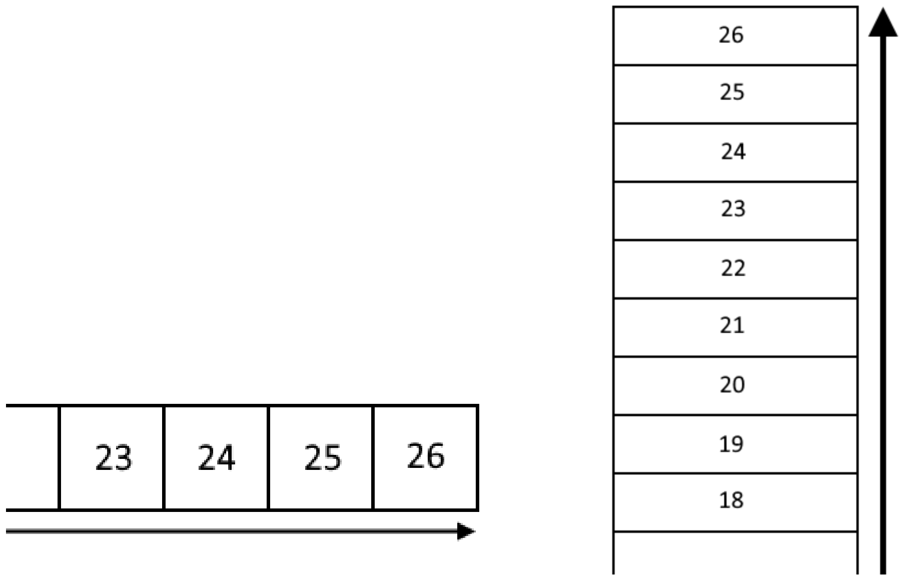


The pixel cannot be changed once this average value is calculated though, as this would affect later calculations made on pixels in the same diagonal.

One way to get around this is by storing the new pixel in memory, and only changing it at the end when all calculation has already taken place.

By doing it this way the pixels will not be affected by the changes. However, one problem that arises is that the pixels are stored backwards in a sense.

Since the program reads them from left to right the result on the stack would be that the rightmost pixel would be stored on it last.



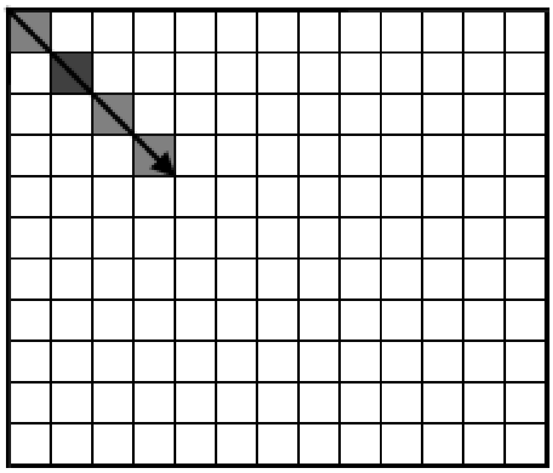
There are two methods of solving this issue that were put under my consideration. The first one was to subtract the size of the image from the stack point so that it would point to the start of the image and then go pixel by pixel.

The second method was to simply move in the opposite direction, as in left to right and instead of adding to the image address, take away from it until the start again.

Due to not wanting to interfere with the stack pointer, it was decided that the second method would work better.

In order to make sure that the subroutine was never taking values outside the range of the image, a simple check was performed. By going through each pixel on the diagonal to it, and seeing if it was within the limit of the image and the radius of the pixel. If it started to go outside the image on one side, this value was recorded and then the same process was done on the other side.

After these extreme points of the image were found the program then started taking the average value from the pixels, starting at the most top left one.



While checking if the radius of the motion blur went outside the image, the number of pixels that was within the border was recorded. This value would later be used to calculate the average value of each pixel.

After the loop has gone through each x and y value for each pixel, a new loop going backwards through the image and setting each pixel started. After this the subroutine would terminate and the image would have motion blur applied to it.

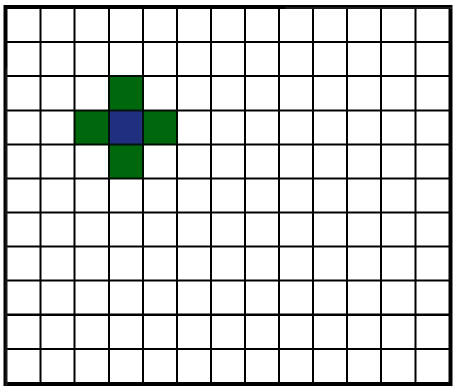
```
while (Radius > 0 and x < 0 and y < 0)
{
    y-=1
    x-=1
    Radius-=1
    Totaldistance +=1
}
startPosX = x
startPosY = y

while (Radius > 0 and x > width and y > width)
{
    y+=1
    x+=1
    Radius-=1
    Totaldistance +=1
}


x = startPosX
y = startPosY
PixelCount = Totaldistance
while (Totaldistance > 0)
{
    AveragePixel += Image[x,y]

    Totaldistance = 0
    x+=1
    y+=1
}

Average = Average/PixelCount
```

Part three: Bonus Effect

The bonus effect was something to be chosen, a list of suggestions was also provided. After a while of researching the various effects that could be applied to the image I finally decided to choose a convolution matrix or kernel as it is also called.

A kernel is an n by n matrix which is applied to an image to have various effects on that image; this includes blurring, sharpening, edge detection, embossing and more. The matrix is applied around and onto every single pixel in the image. Here is an example of an edge detection matrix kernel.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$



In the given example, the pixel highlighted in blue would be multiplied by negative four, and the surrounding green pixels would be multiplied by one. All this would then be added and the result would be the new pixel value.

While very basic, the kernel is a powerful way of affecting images in a variable manner, since all that has to be changed is the values in the matrix.

The subroutine begins by start at the pixel [1,1]. Since any pixels above or to the left of that would be too close to the edge and therefore the matrix would apply outside of the image, these areas are ignored.

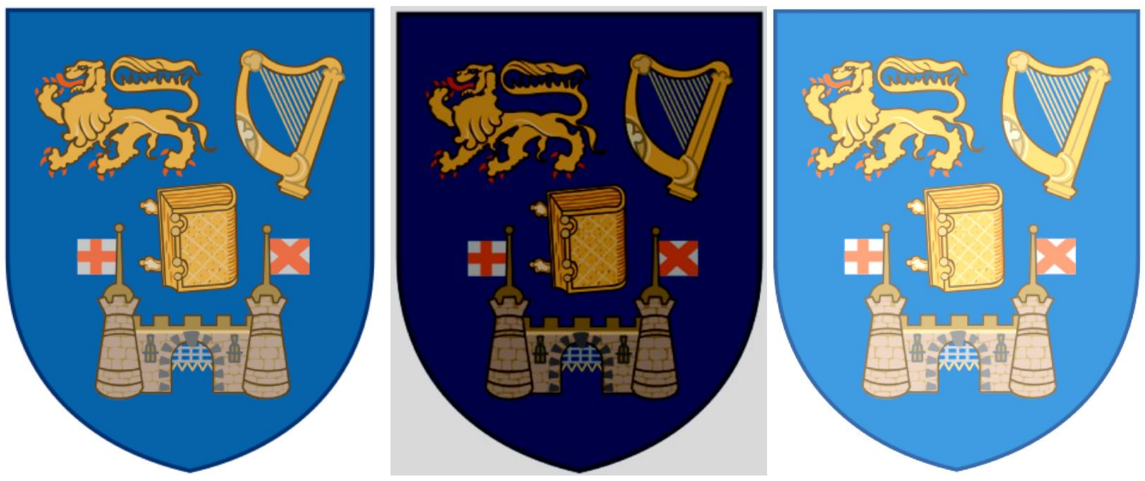
The maximum width and height of the image is also reduced by one to account for the issue on the other side.

After this the subroutine goes through each pixel within the boundaries and applies the matrix to them, these values are stored in memory similarly to the motion blur effect. After the convolution is done the pixels are replaced in the image again and the subroutine ends.

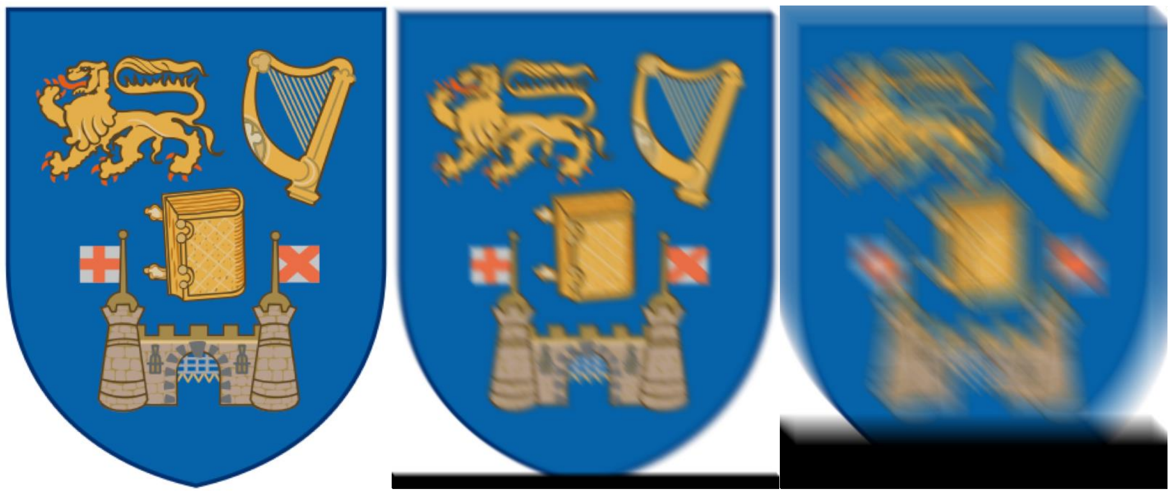
Since the kernel involves multiplying, the values for individual R, G and B values can get quite high, this causes a complication as the byte sized values can grow to much larger than bite sized if not careful. This means, before storing the pixel value in memory, it first has to be checked if it’s within the allowed range for a byte.

Conclusion:

The results of the subroutines was mostly as expected. Presented here are the results, including the original image:

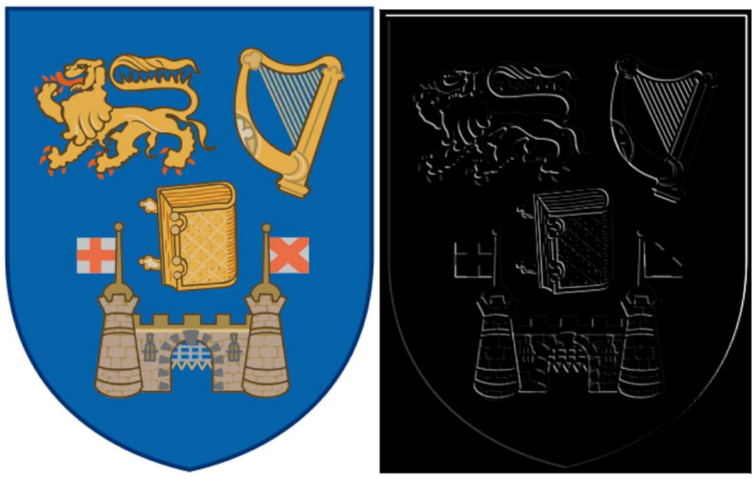


The brightness and contrast adjustment worked exactly as expected, however the motion blur had a strange effect at the bottom of the image.



As the radius of the blur was increased, a black bar at the bottom emerged more and more. This might be due to the subroutine reading memory that it should not be. This, or an error with dividing smaller values which are limited towards the end.

Lastly the kernel worked perfectly, applying an edge detect effect to the image.



In conclusion, all the subroutines worked as intended apart from the motion blur, which had a strange effect at the bottom of the image.