Image Manipulation

20-MAR-17

Report to accompany my image manipulation programs

Niall Hunt

CS1021
Intro to Computing 2

Assignment #1

Contents

Solution

Part 1: Brightness and Contrast

Part 1

In part 1 of this assignment I created a program that will affect the brightness and contrast of a given picture that is stored in memory.

To do this nested for loops are used to iterate through each pixel. This is the standard way of accessing each element of a two dimensional array.

Each pixel is a word sized value. In each pixel there is a red, green and blue (RGB) component. The components are byte sized values. So, for each pixel we will have to change the R, G and B values. This requires a third for loop.

```
for( int j = 0; (j < picHeight); j++ )
{
        for( int i = 0; (i < picWidth); i++ )
        {
                for( int count = 0; (count < 3); count++ )
                {
                        //Do something to the R/G/B value
                }
        }
}
```

The formula used for brightness and contrast is:

$$\mathbf{R/G/B_{ij} = ((R/G/B_{ij} \times \alpha) / 16) + \beta}$$

With α being the contrast and β being the brightness.

Part 1: Brightness and Contrast

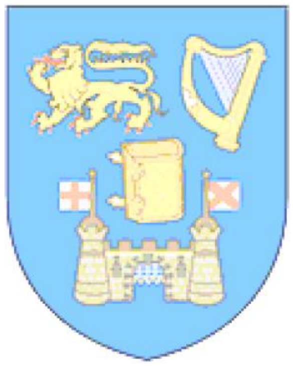
The pseudocode implementation of this is as follows:

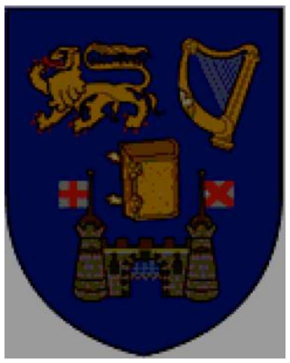
```
if( alpha != 16)
{
        R/G/BValue *= alpha;
        R/G/BValue /= 16;
}

R/G/BValue += beta;

if( R/G/BValue > 255)
        R/G/BValue = 255;
if( R/G/BValue < 0)
        R/G/BValue = 0;
```

My program then stores these new values and updates the displayed picture.

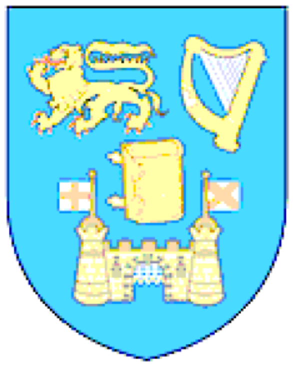
Part 1: Brightness and Contrast

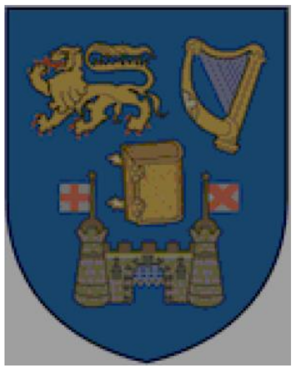
Original


Brightness increased


Brightness decreased

Original


Contrast Increased


Contrast Decreased

Part 2

Part 2: Motion Blur

In part 2 of this assignment I wrote a program that created a box blur effect on the picture. This can be used to approximate Gaussian blur if applied three times to a picture.

To do this the pixel is effected by its surrounding pixels. As each time this occurs we would be using pixels that have already been blurred, we need to create a copy of the picture. This is simple as we loop through every pixel and store the same values into memory. The copy is stored directly after the picture. This is calculated by the following equation:

((picWidth x picHeight) + 1) x 4)

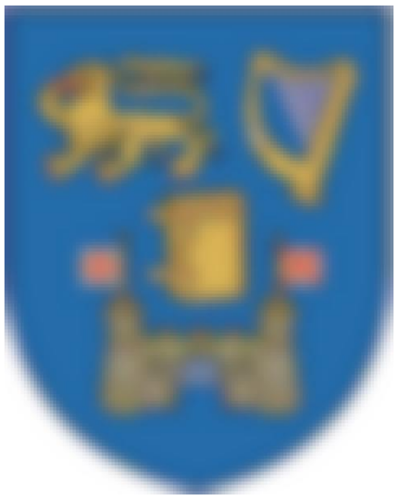
The blurring of each image is done by averaging each of the R, G and B values of the pixels in an n x n matrix surrounding it. The size of the matrix affects how blurry the picture becomes. The radius of this effect is controlled by an inputted parameter into the subroutine. The

```
int deviation = (int) (n / 2);

runningTotal = 0;
for( int countY = -deviation; (countY < deviation); countY++)
{
        y = j + countY;

        for( int countX = -deviation; (countX < deviation); countX++)
        {
                x = i + countX;
                if( x < 0 || x >= picWidth)
                        x = i;
                if(y < 0 || y >= picHeight)
                        y = j;

                R/G/BValue = memory.LoadByte(pic[x, y]);
                runningTotal += R/G/BValue;
        }
}

memory.StoreByte((runningTotal / (n * n)), copiedPic[i, j]);
```

Part 2: Motion Blur

This creates a box blur effect that is shown below. When this is repeated three times it approximates Gaussian blur.

Original


Gausssian Blur

Part 3: Bonus Effect

Part 3

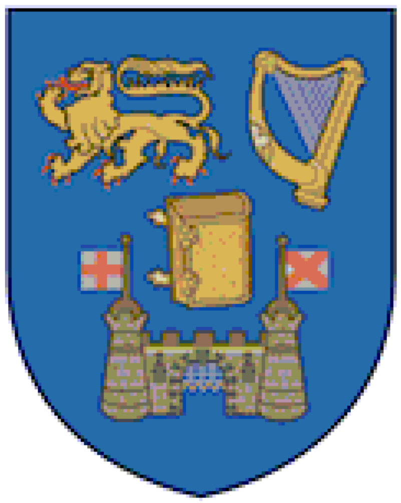
For my bonus effect I used the convolution matrix to create an edge detection effect. To do this the new RGB values are calculated by adding the surrounding pixels with the following matrix applied to it: $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

This part is similar to part 2 as you still have to access each RGB value for each pixel surrounding it. This time however before adding it to the running total you multiply it by the corresponding value in the matrix.

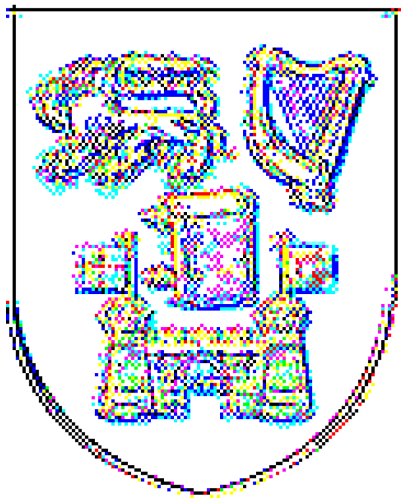
```
int deviation = (int) (n / 2);

runningTotal = 0;
for( int countY = -deviation; (countY < deviation); countY++)
{
        y = j + countY;

        for( int countX = -deviation; (countX < deviation); countX++)
        {
                x = i + countX;
                if( x < 0 || x >= picWidth)
                        x = i;
                if(y < 0 || y >= picHeight)
                        y = j;

                R/G/BValue = memory.LoadByte(pic[x, y]);
                if(x ==0 && y ==0)
                        R/G/BValue *= 8;
                else
                        RG/BValue *= -1;
                runningTotal += R/G/BValue;
        }
}

memory.StoreByte(runningTotal, copiedPic[i, j]);
```

Part 3: Bonus Effect

This gives the following results:



Original



Edge Detection

Testing

Part 1

I tested various values of alpha and beta to see their effects on the picture.

Alpha	Beta	Result	Desired?
0	0	Image is black	As expected for no contrast
32	0	Image is contrasted.	As expected
16	100	Image is much lighter	As expected
16	-100	Image is much darker	As expected

Part 2

When testing some unexpected results were found. For example one variation of my program only blurred a portion of the image as I had accidentally used the picWidth instead of the picHeight when looping through j values.

Another variation blanked out the top quarter of the image and then printed 4 smaller greyscale versions of the crest in a tiled manner in the top quarter. Although I couldn’t figure out exactly what caused this particular effect, I was able to fix it by changing the way I accessed memory. I wasn’t taking the values from the correct memory address, although I still don’t see how this resulted in the tiled effect.

I tested it with different sized Matrices. It works for n x n matrices when n is an odd number. This is expected as it is centered on the pixel you are changing

Part 3

Part three required only a little testing as it is similar in execution to part 2. Originally the colors were incorrect as I was accessing the wrong memory address. This was easily fixed as I found a duplicated line.