Using Linear Algebra to Change Digital Images with Filters

By Will Stout and Tyler Hoskins

Introduction:

Matrices are at the heart of linear algebra and their depiction as a matrix of numbers made them an obvious pick for how we should represent colors in computer graphics. The first computer to use color was the Apple II in 1977, which was able to display a matrix of pixels with different red, green, and blue color values. With the recent advent of colors within computers, practical digital image editing would soon become possible and after a decade of increasing processing power and faster computing, digital photo editing would become cost effective for consumers. Regardless of what editing software used, they all were coded with the same linear algebra in mind.

Abstract:

If you wanted to edit an image, there are a plethora of different ways to do it. Though not all ways to modify a digital image will use fundamental techniques of linear algebra, some

editing tools like scaling, translations, or rotations, all are able to be done because of how they're represented in code. We're going to explore how within a coding IDE(Integrated Development Environment) we can create some of the same editing tools that companies like Adobe developed when making Photoshop. These tools are going to enable us to edit the orientation, colors, and size of an image.

Process:

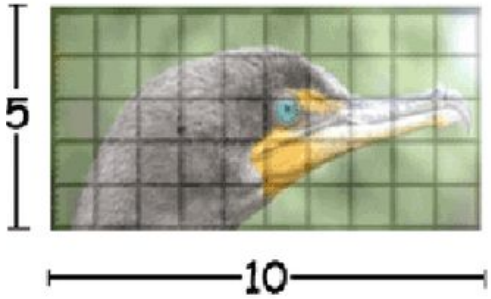
To create our own editing tools we're going to begin coding in C# with Visual Studio as our IDE. To put our software development into layman's terms, we're going to open up a new Project in Visual Studio, then create a GUI(Graphical User Interface) which allows the picture to be seen on screen. Then we're going to create an image representation by creating three, two dimensional matrices that when the computer renders the matrices together will create the current picture. The three matrices will hold all the red, green, and blue brightness values, similar to the illustration below. We now have a numerical representation of colors that when changed will change the image. The connection between photoshop and the matrices used is that Photoshop will keep track of the color matrices of the image you are editing and then when you

edit that image photoshop will concurrently edit the brightness values within the matrices.

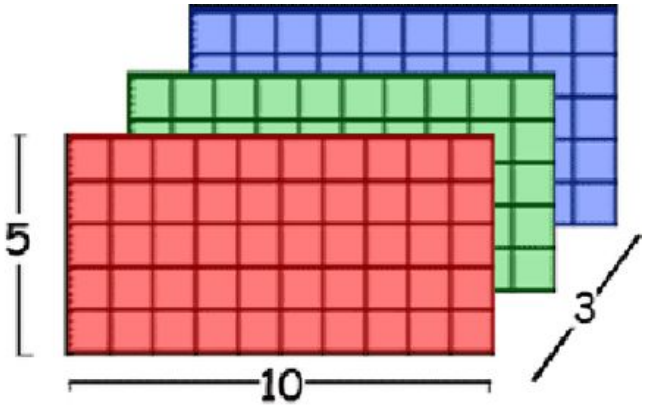
```
namespace image_editor_tutorial
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        void openImage()
        {
            DialogResult dr = openFileDialog1.ShowDialog();
            if (dr == DialogResult.OK)
            {
                file = Image.FromFile(openFileDialog1.FileName);
                pictureBox1.Image = file;
                opened = true;
            }
        }


        Image file;
        Boolean opened = false; // to check weather image is open in picture box or not
```

```
Image img = pictureBox1.Image;                              // storing image into img variable of image type fr
Bitmap bmpInverted = new Bitmap(img.Width, img.Height);   /* creating a bitmap of the height of imported pictu
                                                          and its attributes. A Bitmap is an object used to work


ImageAttributes ia = new ImageAttributes();                 //creating an object of imageattribute ia to chang
ColorMatrix cmPicture = new ColorMatrix(new float[][]       // now creating the color matrix object to change
{
    new float[]{0.393f, .349f+0.5f, .272f, 0, 0},
    new float[]{.769f+0.3f, .686f, .534f, 0, 0},
    new float[]{.189f, .168f, .131f+0.5f, 0, 0},
     new float[]{0, 0, 0, 1, 0},
     new float[]{0, 0, 0, 0, 1}
});
ia.SetColorMatrix(cmPicture);           //pass the color matrix to imageattribute object ia
Graphics g = Graphics.FromImage(bmpInverted);   /*create a new object of graphics named g, ; Create graphics o
                                              Graphics newGraphics = Graphics.FromImage(imageFile); is the forma

g.DrawImage(img, new Rectangle(0, 0, img.Width, img.Height), 0, 0, img.Width, img.Height, GraphicsUnit.Pixel,


/*   g.drawimage(image, new rectangle(location of rectangle axix-x, location axis-y, width of rectangle, heigh
location of image in rectangle x-axis, location of image in rectangle y-axis, width of image, height of image,
format of graphics unit,provide the image attributes   */


g.Dispose();                            //Releases all resources used by this Graphics.
pictureBox1.Image = bmpInverted;
```

With this code we begin instantiating the image object, then extrapolate the pixels into the color matrix object named ColorMatrix. The image we input is the left image (below), and it will be interpreted by the code as the right image.



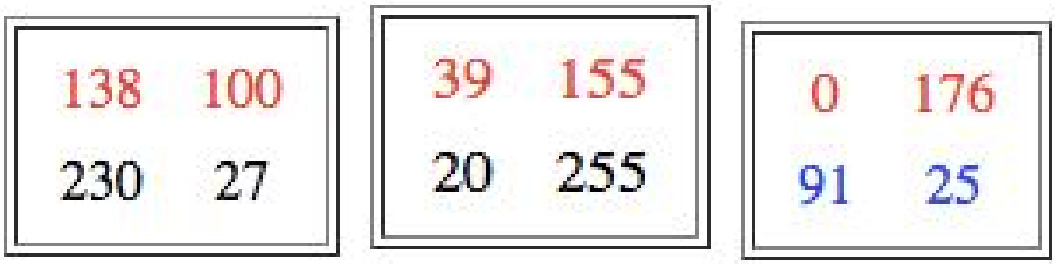
Original Color Image



Matlab RGB Matrix

Blurring an Image:

Blurring images finds itself at the intersection of different fields in mathematics. Linear Algebra, Computer Science, and even Statistics. Originally developed as a tool in statistics to find the normal distribution of a set of values, Computer Science created another practical use for it, blurring an image, which is one of the most simple image manipulation techniques you can code. Let's say that the matrices below are a representation of the brightness of the red, green, and blue brightness values of a digital image that has 4 pixels in it (2x2). The values are typically from 0 to 255, with 0 being no color is output (black) to maximum color output (full red, green, or blue)



Red Green Blue

To perform the blur we are going to iterate through all of the pixels and apply our Gaussian Normal Distribution. This requires us to look at each pixel individually and then it's direct neighbors brightness values, the larger the blur you want the more neighbors you'll include in your calculation. Now we're going find the average value of all the cells we looked at and then place that new value in a new RBG matrix. If we to put that value back into the matrix we just got it from, we would end up blurring colors that weren't originally there, and creating a false blur. Another type of blur is a scaling blur where matrix multiplication is used to multiply the brightness of pixels and then dividing by the number of terms multiplied to the second power. It creates a much more contrast based blur.

By combining the three matrices into one, you can see how common Linear Algebra concepts come to life. If you wanted to get the combined brightness values out of the matrices we would use simple Ax=B form, like such.

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

To show what this looks like in real life, let's use an extreme example of a 2x2 blue color brightness matrix. If we were to start with the matrix below and apply a blur that very softly mixed the brightness values together, we will see something like the following.

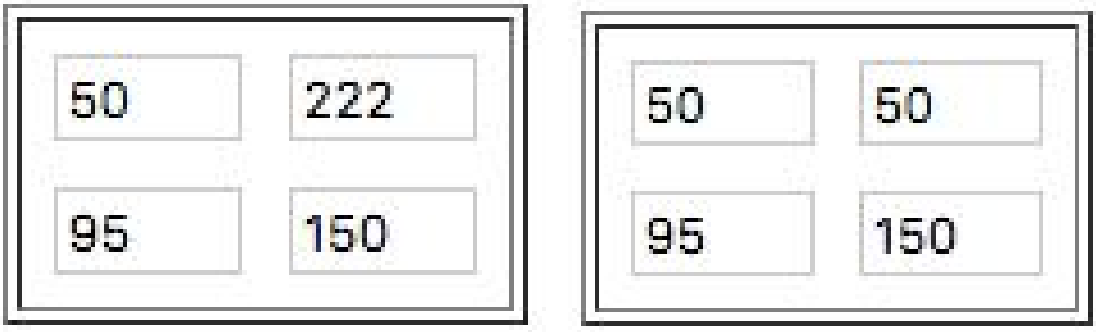
0	255
255	255

And we would produce a matrix that would look something like below.

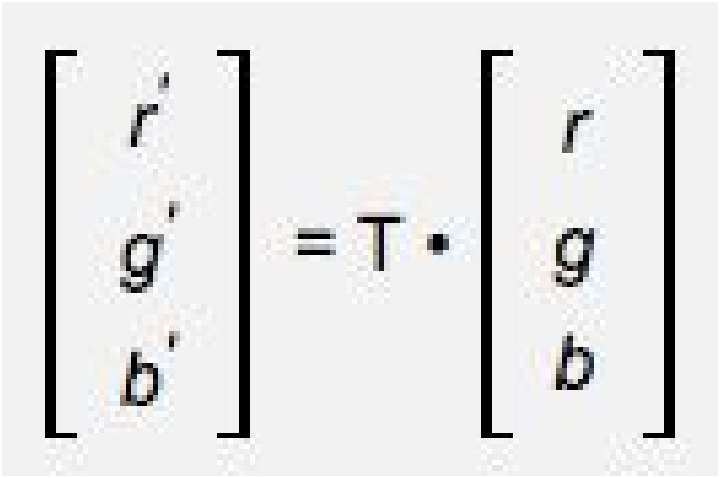
51	204
204	255

Transforming an Image:

Another way to alter an image in PhotoShop is to transform it. Let's say you have some image but you want to see what it would look like if you scaled the size of something in the image to something new. Imagine gigantic cows or skyscrapers the size of toothpicks. Coding this without Linear Algebra would require you to calculate where every pixel would end up and change their color matrices individually, but with Linear Algebra we can take a linear combination of the pixels that the cow exists in and just apply that mathematical combination to the color matrices. So if we want to stretch something one pixel farther we're going to select the pixels the cow is in, and apply the one pixel linear combination of that transform. We finish by updating all the color matrices to represent the transformed image. We must be careful however because scaling transforms might need to update many pixels. If we were to scale one pixel in small green color matrice it would look something like the following.



The left side is the original green brightness values and the right is the new scaled up. This is a painfully simple example, but it shows that whatever value was in the upper left corner has now been scaled to the right, and we would end up with the upper right pixel being significantly less green. The specifics behind doing transforms is simply expressed with the following matrix operation.



Conclusion:

In 1988 when Adobe first released Photoshop, it was hard to imagine that they could have done so without heavy use of Linear Algebra. Applications are often so straight forward it is hard to imagine there being another way of computing pixel brightness. With blur, it doesn't even seem like Linear Algebra is really being used, but behind the scenes we can see the all the tiny matrix multiplications being done. Then with scaling transformation we get to see how other parts of the field find their way into computing graphics. The use of Linear Algebra is completely inseparable from Computer Science and its digital photo editing uses.

References:

http://www.nibcode.com/en/blog/13/linear-algebra-and-digital-image-processing-part-II-filters

http://www.math.leidenuniv.nl/~kallecccj/lab/ip.pdf

http://www.math.harvard.edu/archive/21b_fall_04/exhibits/imageprocessing/index.html

https://math.stackexchange.com/questions/1858631/i-need-help-normalizing-a-gaussian-kernel-matrix-to-integer-values

https://en.wikipedia.org/wiki/Scaling_(geometry)