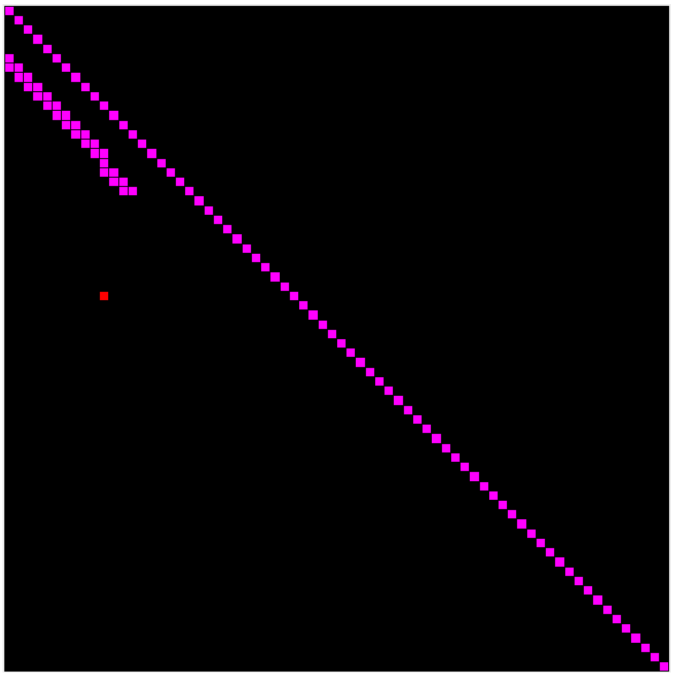
COMP3811: Computer Graphics
Coursework 1
Rasterization
Giorgos Kosta ID:201257368

Assignment 1:

```
void PixelWidget::DrawLine(pixel start_p, pixel end_p){

    //calculate manhattan distance between the two points
    float steps = fabs((end_p.x-start_p.x)) + fabs((end_p.y-start_p.y));

    for(int i = 0; i<steps; i++){ //number of steps = distance


        //step size i/steps
        float x= (end_p.x + ((i/steps) * (start_p.x-end_p.x)));
        float y= (end_p.y + ((i/steps) * (start_p.y-end_p.y)));

        SetPixel((int)x,(int)y,RGBVal(255,0,255));

    }


}
```

For choosing the step size, 3 different methods were tested.The first method for calculating the step size was in just a for loop with increments of 0.01 which was plenty fast and all pixels were set. However for small line segments, this was really wasteful, thus investigating the possibility of using distance as a guide for the step size. The second method used was the euclidean distance which is not enough for some cases. The third and final option was the most robust and relatively least wasteful, the manhattan distance. By using this step sizing, the parameterized line will have the minimum number of steps needed to light all the pixels that are required.



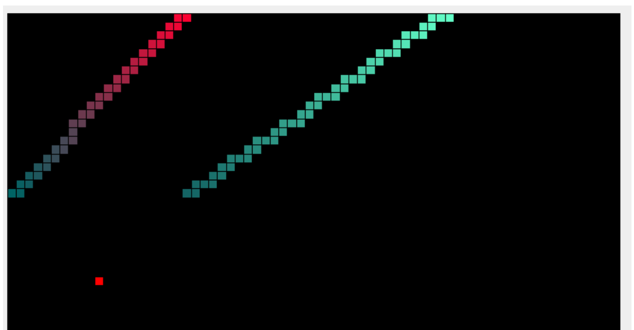
```
pixel a,b,c,d;
a.x=0.0;
a.y=0.0;
b.x=69.0;
b.y=69.0;

c.x=0.0;
c.y=5.0;
d.x=13.65;
d.y=20.0;


DrawLine(a,b);
DrawLine(d,c);
```

In the above screenshot, line DC from the code has 28 pixels set. However if the euclidean distance was used only 18 pixels would be set, and if a step size of 0.01 was used 27 pixels. Only if a step size of 0.001 would find 28 pixels which is wasteful. Therefore the above two lines drawn by using the manhattan distance as a guide for the step size set the correct number of pixels.

Assignment 2

```
int r_diff=0,g_diff=0, b_diff=0;

//calculate diffs
r_diff = -(start_p.rgbVal._red - end_p.rgbVal._red);
g_diff = -(start_p.rgbVal._green - end_p.rgbVal._green);
b_diff = -(start_p.rgbVal._blue - end_p.rgbVal._blue);



//calculate manhattan distance between the two points
float steps = fabs((end_p.x-start_p.x)) + fabs((end_p.y-start_p.y));
steps*=10;
float stepsize=0.001;

for(int i = 0; i<=steps; i++){ //number of steps = distance


    //step size i/steps

    stepsize= i/steps;
    float x= (end_p.x + (stepsize * (start_p.x-end_p.x)));
    float y= (end_p.y + (stepsize * (start_p.y-end_p.y)));

    RGBVal newRGBVal;
    //add accumulative diff to each rgb val
    newRGBVal._red= (end_p.rgbVal._red + (r_diff*-stepsize));
    newRGBVal._green= (end_p.rgbVal._green + (g_diff*-stepsize));
    newRGBVal._blue= (end_p.rgbVal._blue + (b_diff*-stepsize));

    //if diff exceeds max rgb val, remain ta max
    newRGBVal._red >= MAX_RGB ? newRGBVal._red = MAX_RGB : newRGBVal._red = newRGBVal._red;
    newRGBVal._red <= MIN_RGB ? newRGBVal._red = MIN_RGB : newRGBVal._red = newRGBVal._red;

    newRGBVal._green >= MAX_RGB ? newRGBVal._green = MAX_RGB : newRGBVal._green = newRGBVal._green;
    newRGBVal._green <= MIN_RGB ? newRGBVal._green = MIN_RGB : newRGBVal._green = newRGBVal._green;

    newRGBVal._blue >= MAX_RGB ? newRGBVal._blue = MAX_RGB : newRGBVal._blue = newRGBVal._blue;
    newRGBVal._blue <= MIN_RGB ? newRGBVal._blue = MIN_RGB : newRGBVal._blue = newRGBVal._blue;


    SetPixel((int)x,(int)y,newRGBVal);

}
```

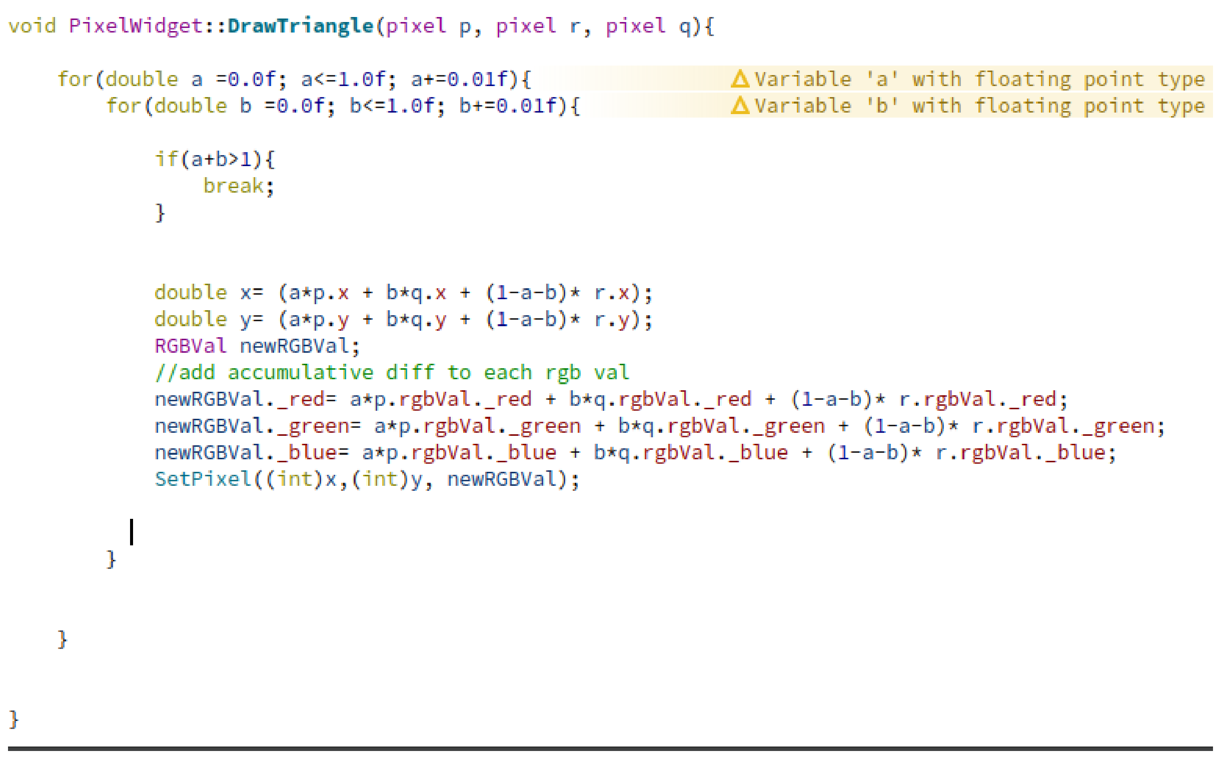
The difference between the RGB values of the two points is calculated. Then the colours are interpolated by using the step size with the difference that was pre-calculated.



```
pixel a,b,c,d;
a.x=20.3;
a.y=0.0;
a.rgbVal=RGBVal(255,0,50);

b.x=0.6;
b.y=20.8;
b.rgbVal=RGBVal(0,100,100);
DrawLine(b,a);
```

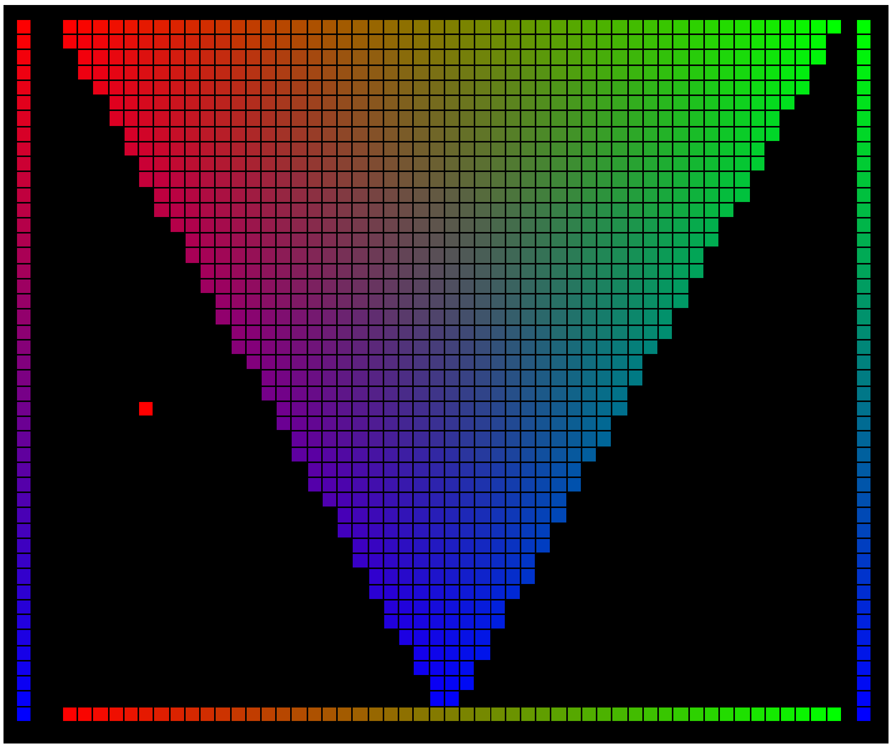
As it can be seen above, the colours have a smooth transition from the expected starting value to the final value of the end point.

Assignment 3

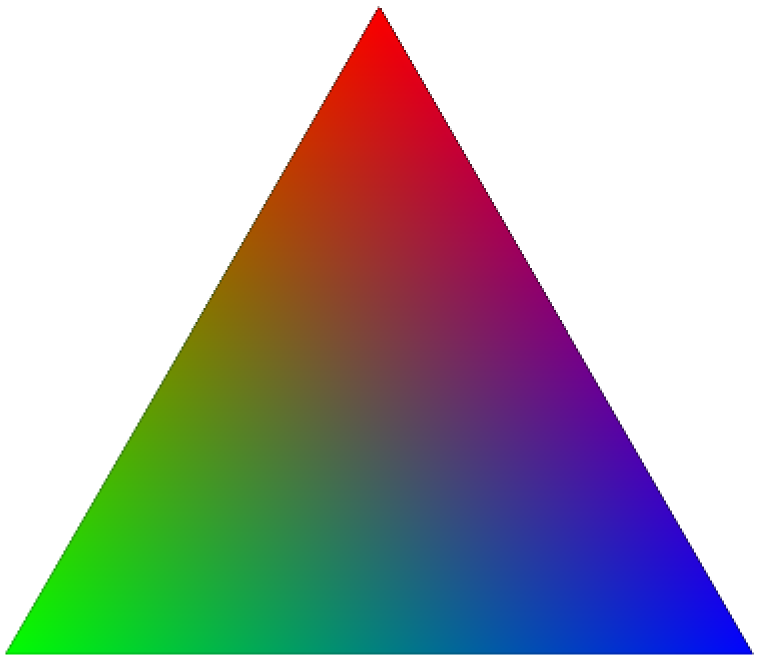


```
void PixelWidget::DrawTriangle(pixel p, pixel r, pixel q){

    for(double a =0.0f; a<=1.0f; a+=0.01f){                ⚠Variable 'a' with floating point type
        for(double b =0.0f; b<=1.0f; b+=0.01f){            ⚠Variable 'b' with floating point type

            if(a+b>1){
                break;
            }


            double x= (a*p.x + b*q.x + (1-a-b)* r.x);
            double y= (a*p.y + b*q.y + (1-a-b)* r.y);
            RGBVal newRGBVal;
            //add accumulative diff to each rgb val
            newRGBVal._red= a*p.rgbVal._red + b*q.rgbVal._red + (1-a-b)* r.rgbVal._red;
            newRGBVal._green= a*p.rgbVal._green + b*q.rgbVal._green + (1-a-b)* r.rgbVal._green;
            newRGBVal._blue= a*p.rgbVal._blue + b*q.rgbVal._blue + (1-a-b)* r.rgbVal._blue;
            SetPixel((int)x,(int)y, newRGBVal);

        }


    }


}
```

A nested loop with step sizes of 0.01 for alpha and beta is used to calculate the coordinates of each point in the triangle. Then the new interpolated RGB value is calculated with the same step sizes. #TODO step size from area



As seen here, each vertex of the triangle has its own max RGB value. In order to check the success of the interpolation, an already interpolated triangle from online sources(triangle bellow) proved that the drawn triangle above was successful.


Source: https://www.boristhebrave.com/2018/05/12/barycentric-perlin-noise/