# Let's Build a 3D Graphics Engine: Colors



by Kyle Sloka-Frey 19 Jun 2013

Difficulty: Intermediate Length: Medium Languages: English

Programming Platform Agnostic HTML5

This post is part of a series called Let's Build a 3D Graphics Software Engine.

Let's Build a 3D Graphics Engine: Rasterizing Triangles and Quads

Let's Build a 3D Graphics Engine: Dynamic Lighting

Welcome! This is the sixth part in our Let's Build a 3D Graphics Engine series covering the basics of 3D graphics systems. This time we are going to be talking about color and how to add it into our existing classes. We are also going to create a few useful functions to make handling lighting easier, which is what our next and final part will be about.

## Recap

Adding color to our objects is not going to be too huge of an undertaking, so the only two class that we are going to focusing on heavily are our point class and our camera class.

As a refresher, here is what they look like:

```
Point Class
{
    Variables:
        num tuple[3]; //(x,y,z)
    Operators:
        Point AddVectorToPoint(Vector);
        Point SubtractVectorFromPoint(Vector);
        Vector SubtractPointFromPoint(Point);
        Null SetPointToPoint(Point);
    Functions:
        drawPoint; //draw a point at its position tuple
}

Camera Class
{
    Vars:
        int minX, maxX;
        int minY, maxY;
        int minZ, maxZ;
        array objectsInWorld; //an array of all existent objects
    Functions:
        null drawScene(); //draws all needed objects to the screen
}
```

So far, our theoretical engine has almost all of the basics in place, including:

- `Point` and `Vector` classes (the building blocks of our engine).
- Transformation functions for our points.
- A `Camera` class (sets our viewport, and culls points outside of the screen).
- Three classes for rasterizing (line segments, circles, and polygons).

Now let's add some color!

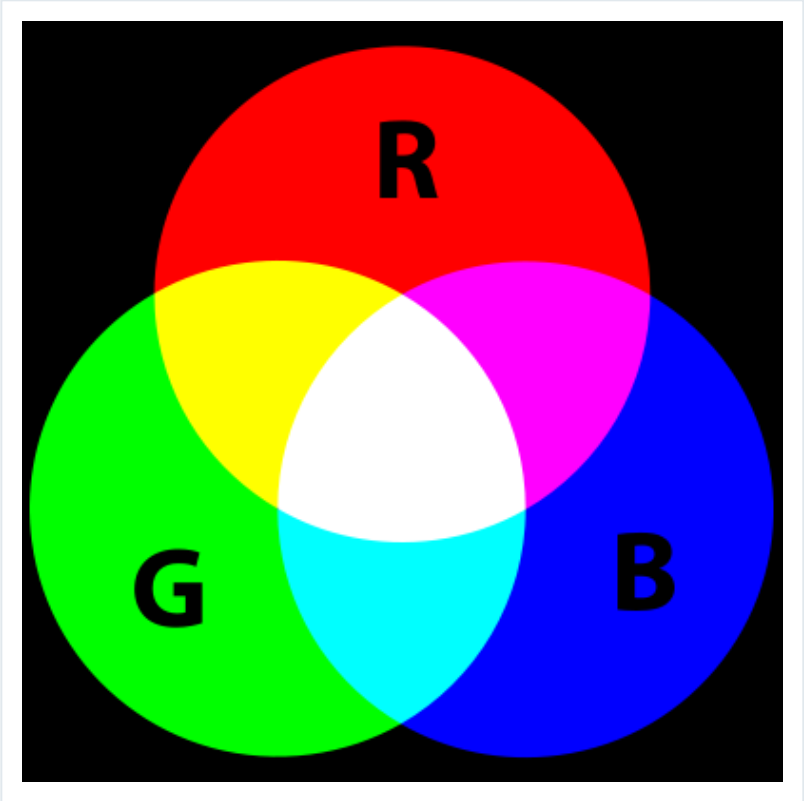
# Color for Everyone!

Our engine is going to handle colors by storing their values within its `Point` class. This allows each point to have its own individual color, making lighting and shading calculations much simpler (for people, at least - sometimes it is less efficient to code an engine this way). When figuring out a scene's lighting or shading, we can easily provide

the function with a list of points, and then churn through each of them, using their distance from the light to alter their color accordingly.

One of the most common ways to store color in programming is to use red, green, and blue values to create the actual desired color (this is typically called additive color mixing). By storing a value of 0-255 in each of these color segments, you can easily create a wide variety of colors. (This is how most APIs determine color, so for compatibility reason it makes sense to use this method).

Then, depending on the graphics API that you are using, you can pass these values in either decimal form (`255,0,0`), or in hexadecimal form (`0xFF0000` or `#FF0000`). We're going to use decimal format in our engine since its a bit easier to work with. Also, if your graphics API does use hexadecimal values, then it likely has a function for converting from decimal to hexadecimal, so this shouldn't be a problem.




To get our color implementation started we are going to add in three new variables to our Point class: `red`, `blue`, and `green`. There is nothing too outlandish going on quite yet, but here is what our `Point` class's new outline could look like:

```
Point Class
{
    Variables:
        num tuple[3]; //(x,y,z)
        num red, green, blue; //could be abbreviated to r,g,b if desired
    Operators:
        Point AddVectorToPoint(Vector);
        Point SubtractVectorFromPoint(Vector);
        Vector SubtractPointFromPoint(Point);
        Null SetPointToPoint(Point);
    Functions:
        drawPoint; //draw a point at its position tuple
}
```

That is all we need to store our point's color. Now we just need to adjust our camera's draw function so that it uses the specified color.

This is going to change drastically depending on which graphics API you are using, but they should all have a similar function to this:

```
object.setColor(red, green, blue)
```

If your graphics API happens to use hexadecimal values for color instead of decimal, then your function would look similar to this:

```
object.setColor(toHex(red,green,blue))
```

That last bit uses a `toHex()` function (again, the function names will differ from API to API) to convert an RGB value into a hexadecimal value so that you don't have to.

After having made these changes, you should now be able to have colored points within your scene. To take it a step further, we are going to adjust each of our rasterization classes so that our entire shape can be colored.

To add this to our classes, we simply have to add in color handling to their constructor functions. This might look like:

```
lineSegment::constructor(startX, startY, endX, endY, red, green, blue)
{
    this.startX = startX;
    this.startY = startY;
    this.endX = endX;
```

```
    this.endY = endY;
    this.red = red;
    this.green = green;
    this.blue = blue;
}
```

Then, we just need to modify its return points function so that it sets each point in its array to have the specified color. The new function would look like this:

```
function returnPointsInSegment()
{
    //create a list to store all of the line segment's points
    var pointArray = new Array();
    //set this function's variables based on the class's starting and ending points
    var x0 = this.startX;
    var y0 = this.startY;
    var x1 = this.endX;
    var y1 = this.endY;
    //define vector differences and other variables required for Bresenham's Algorithm
    var dx = Math.abs(x1-x0);
    var dy = Math.abs(y1-y0);
    var sx = (x0 & x1) ? 1 : -1; //step x
    var sy = (y0 & y1) ? 1 : -1; //step y
    var err = dx-dy; //get the initial error value
    //set the first point in the array
    pointArray.push(new Point(x0,y0,this.red,this.green,this.blue));

    //Main processing loop
    while(!((x0 == x1) && (y0 == y1)))
    {
        var e2 = err * 2; //hold the error value
        //use the error value to determine if the point should be rounded up or down
        if(e2 => -dy)
        {
            err -= dy;
            x0 += sx;
        }
        if(e2 < dx)
        {
            err += dx;
            y0 += sy;
        }
        //add the new point to the array
        pointArray.push(new Point(x0, y0,this.red,this.green,this.blue));
    }
    return pointArray;
}
```

Now, every point within the line segment should be the same color that was passed into the line segment.  You can use this method to set colors up in your other rasterizing

classes as well, and soon your scene will come alive with color!

Let's put our new features into action by making a program to show them off.



# Playing With 16.7 Million Colors

Using additive color mixing, we can easily create over 16.7 million different colors using just the simple (`r,g,b`) notation. We are going to create a program that takes advantage of this vast number of colors.

Using key presses, we are going to allow the user to control an object's red, green, and blue values individually, allowing them to make it into any color that they would like.

The specifications for our program are as follows:

- Draw an object to the screen.
- If the user presses **A** then lower the object's red value; if they press **Q** then raise it.
- If the user presses **S** then lower the object's green value; if they press **W** then raise it.
- If the user presses **D** then lower the object's blue value; if they press **E** then raise it.
- Redraw the object after its color has been updated.
- Make sure to cap the color values, preventing them from dropping below 0 or rising above 255.

With all of that in mind, let's take a look at what a basic outline of our program might look like:

```
01 main{
02
03     //setup for your favorite Graphics API here
04     //setup for keyboard input (may not be required) here
05
06     var camera = new Camera(); //create an instance of the camera class
07
08     //set the camera's view space
09     camera.minX = 0;
```

```
    camera.maxX = screenWidth;
    camera.minY = 0;
    camera.maxY = screenHeight;
    camera.minZ = 0;
    camera.maxZ = 100;

    //store our colors so that they can be manipulated
    var red, green, blue;

    //draw initial object and set it to a variable

    while(key != esc) {
        if(key press = 'a')
        {
            if(red > 0)
            {
                red --;
                object.red = red;
                //redraw object
            }
        }
        if(key press = 'q')
        {
            if(red < 255)
            {
                red ++;
                object.red = red;
                //redraw object
            }
        }
        if(key press = 's')
        {
            if(green > 0)
            {
                green --;
                object.green = green;
                //redraw object
            }
        }
        if(key press = 'w')
        {
            if(green < 255)
            {
                green ++;
                object.green = green;
                //redraw object
            }
        }
        if(key press = 'd')
        {
            if(blue > 0)
            {
                blue --;
                object.blue = blue;
                //redraw object
            }
        }
```

```
        if(key press = 'e')
        {
            if(blue < 255)
            {
                blue ++;
                object.blue = blue;
                //redraw object
            }
        }
    }
}
```

Now we can play around with our object and make it into any color that you desire!

Check out the demo here - repeatedly press the **Q**, **W**, **E**, **A**, **S**, and **D** keys to change the color of the square.

# Conclusion

With color added into our engine, we've got everything that we need in place to finally handle some lighting. In the next article, we will be looking at creating lighting sources, and creating some functions to allow those sources to affect our points' colors. The depth that lighting adds to an engine is extremely satisfying, so make sure that you check it out!


## Kyle Sloka-Frey

Data Scientist, Father, Game Dev, Developer, Futurist

Kyle is a data scientist, father, game dev, and all around analytics and organization lover. He enjoys space and all things Futurism.

kyleslokafrey