Technical Description
Assignment #2

March 28, 2017

Nicholas Miller

Table of Contents

Objectives

There are a few core objectives to this assignment, all of which built on top of the previous assignment.

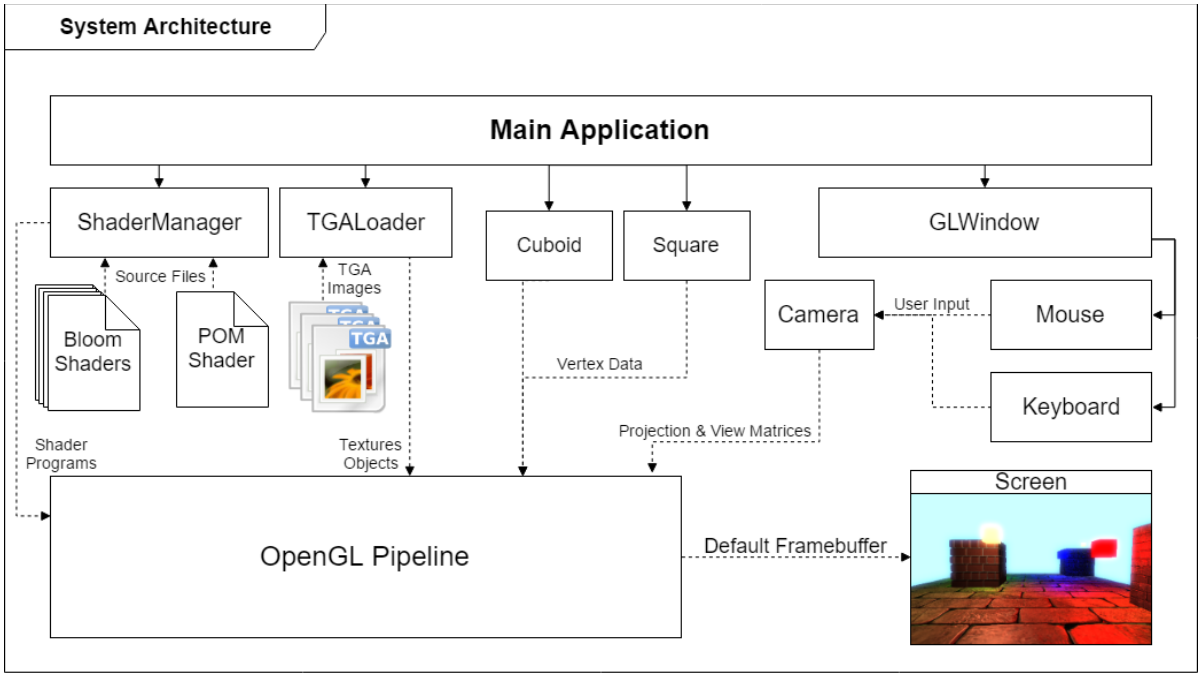
1. A texture loader will be implemented so that images can be used within various shading techniques.
2. Implementation of parallax occlusion mapping shader effect.
3. Implementation of bloom shader effect.

System Architecture

Before addressing the first objective, let's first look at the revised system architecture to show what changes have been made between the first and second assignments.

As before, we still have `ShaderManager`, which is used for compiling and linking shader programs. We also still have model data that specify vertex data (notice that `Sphere` has been removed and `Square` has been added). This change enables us to perform post-processing effects we'll discuss later. Also, we have the same camera system as before.

The only real change to the architecture is that there is now a `TGALoader`, which is capable of loading some types of of TGA image files. All other changes include small adjustments and changes to the data inputs (i.e., which information is being sent to the OpenGL pipeline).



Texture Loader

The first objective is to create a means of loading texture images (specifically, TGA images) which will allow us to create a more dynamic range of effects and provides a way of adding realism to our scene. To do this, we must first create a `TGALoader` class. This is a helper class which contains only one function, **`Load`**. This function expects a file-path to a TGA image and it return a texture object , or handle, that references the image after it has been stored in GPU memory.

TGA File Format

To create this function, we must first understand the TGA file format. The TGA file format is a very basic format, and is probably the simplest format that supports the alpha channel (which is often used for transparency). The file format begins with a header section. This specifies basic information about the contents of the file and includes things such as the image's type and dimensions, what channels and pixel bit-depths are used, and whether or not Run-Length Encoding (RLE) compression is used. The next sections, which are optionally provided, are the image ID field the color map data. The image ID is a user data section typically used to store the image creation date and time, and the color map data section contains what valid colors can be used by the image (granted it is a paletted image type). For images that specify a color map, image data will be specified using indices into this color map. For our purposes, we will not be concerned with any of these types of TGA files.

The next section is the actual image data. For TrueColor files, this is where the actual RGB/RGBA data goes. Optionally, this section can be compressed using RLE which is a relatively simple compression algorithm. The image files we will be loading, will be TrueColor, 24-bit RGB channels (or 32-bit RGBA), with RLE compression. According to the specification, this corresponds with image type 10.

After the image data section, there are various miscellaneous sections that provide developers greater control with how to use the TGA image. For our purposes, however, we will not care about these extra features or sections, and will stop reading the image file after the image data section.

Let us start by showing the code for how we read the file (beginning with the header section which is only 18 bytes long):

```
Texture TGALoader::Load(string path)
{
	//First create a Texture struct to store the result.
	Texture texture;
	texture.id = 0; //from OpenGL (later)
	texture.path = path;

	//Create a file input-stream to read the TGA file.
	ifstream instream(path.c_str(), ios::binary);

	//Create a buffer to store the contents read (we will do
	//this for each chunk read).
	char* buf = new char[18];
	instream.read(buf, 18);

	//Extract header information.
	//Most fields are uninterested except for the image type and spec.
	int imageIDLength = buf[0];
	int colorMapType = buf[1];
	int imageType = buf[2];
	int colorMapFirstIndex = (buf[4] << 8) | buf[3];
	int colorMapLength = (buf[6] << 8) | buf[5];
	int colorMapBitsPerPixel = buf[7];

	//This is the image information we care most about.
	ImageSpecification imageSpec;
	imageSpec.xOrigin = (buf[9] << 8) | buf[8];
	imageSpec.yOrigin = (buf[11] << 8) | buf[10];
	imageSpec.width = (buf[13] << 8) | buf[12];
	imageSpec.height = (buf[15] << 8) | buf[14];
	imageSpec.pixelDepth = buf[16];
	imageSpec.imageDescriptor = buf[17];

	//... (more shown later) ...
```

As can be seen, we grab the information from the header and this allows us to determine what the rest of the file looks like. We can now move onto reading the image ID and color map sections, but since we don't care about image ID information or about loading paletted TGA files, we instead just read past these sections in order to access the image data section:

```
buf = new char[imageIDLength];
instream.read(buf, imageIDLength); //Skip image ID section.
delete[] buf;

buf = new char[colorMapLength];
instream.read(buf, colorMapLength); //Skip color map data section.
delete[] buf;
```

Reading image data is the final part that we read, and as we read the image data, we will build an array to store the raw pixel values. This array will later be sent to the GPU, which means we must be careful to format this array in a manner that OpenGL expects. Since we are only concerned with image type 10 (as briefly mentioned before), we must be able to decompress RLE images. For RLE images, the pixels are organized in the following manner:

- The image is separated into an array of *packets* that each contain a 1-byte *packet header* and *n*-byte *packet data*.
- Each packet can either specify a *run* of RLE compressed pixels or raw pixels. Each run specifies a block of pixels. The first block of pixels occurs at the image's origin and continues row by row until the end of the image is reached.
- For RLE packets, the packet's header must be a value greater than or equal to 0x80. Which is to say, the most significant bit (MSB) is 1 (indicating that the packet is an RLE packet). The lower 7 bits correspond to the number of occurrences (plus 1) that the pixel's color occurs. The RLE packet data contains the color for each pixel in the run. For RGB or RGBA pixels, there are 24 or 32 bits which means that the packet data is either 3 or 4 bytes long.
- For raw packets, the MSB of the packet header is 0. Then the lower 7 bits again are used to indicate how many pixels are in the run (again, we add 1 to this value). A raw pixel is uncompressed, so this means that the packet data consists of *all* the pixels in the run (one after another). The advantage of grouping raw pixels like this is that we do not need to specify that each pixel has an occurrence of one in the case of an image with many dissimilar pixels all in a row. Rather, we indicate how many pixels cannot be grouped together and have only one header to specify them all.
- Lastly, the image data (as well as every field in the file) is in *little-endian*. For the packet data in the image, TrueColor pixels appear as BGR or BGRA format (since the pixel color is treated as one field, it causes the blue component to appear first in the file).

For complete details on how to decode the image data, please see the ***TGALoader.cpp*** file.

Specifying an OpenGL Texture

Now that we have produced an array that contains the entire pixel data of the image, we can now look at how this data is sent to the GPU so that we can create a texture object. Like many OpenGL objects, we must first make a call to a **`glGen*`** function. For textures, we call **`glGenTextures`**, along with providing the number of textures we want to create (for the texture loader, we only need to create one at a time), then we pass a pointer to where this object will be stored, once created.

After creating the texture object, we can optionally specify some texture parameters. To do this, however, we must first make the texture object active by binding it to the OpenGL state's active texture object. Using the `Texture` struct from above, we simply make the following calls to create/store the texture object and to bind it:

```
//Create OpenGL texture object.
glGenTextures(1, &texture.id);
glBindTexture(GL_TEXTURE_2D, texture.id);
```

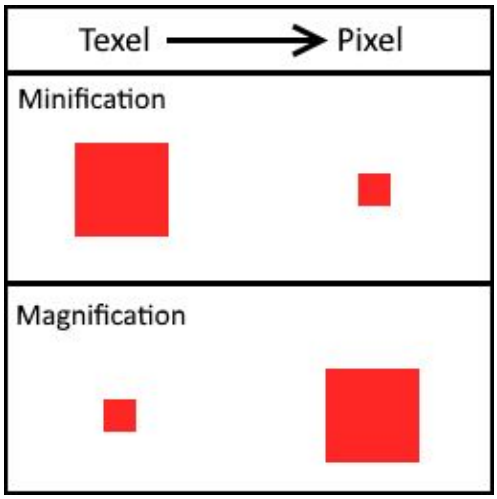
There are two parameters that are of interest, which answer the following questions:

1. How does a texture image get mapped to pixels when a single texel (or the base unit of a texture) spans multiple pixels?
2. How does a texture image get mapped to pixels when a single pixels spans multiple texels?

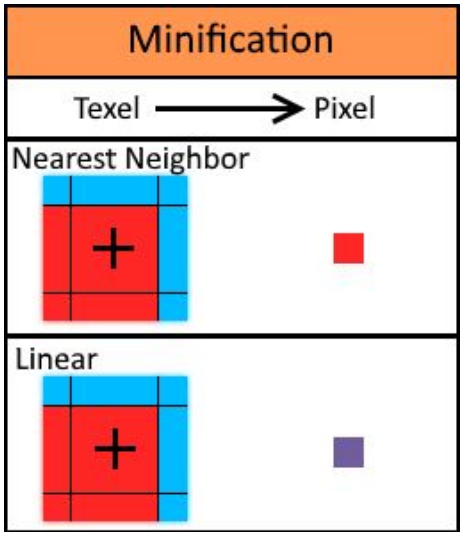
In the first question, the texel is larger than the pixel. This happens when a texture stretched, whether it is stretched by a model's vertex positions, or by the projection to the screen. To determine a corresponding pixel's value, a portion of the texture must be minified. Depending on how this minification process occurs is the objective of the texture's minification filter.

In the second question, the opposite is true. The texel is smaller than the pixel and must be magnified (which is done by the textures's magnification filter).

The figure on the right illustrates the goals of minification and magnification. There are two modes for this: *nearest*, and *linear*. A linear min/mag filter implies that that the pixel's color is determined by using a weighted average of nearby texels, whereas a 'nearest' filter implies that only the closest texel (in distance) will be used for the pixel's color



To show how a minification filter works, consider the second figure (below). On the left, there is a cross mark that indicate where the pixel to be sampled is located. For the nearest neighbor filtering, the texel at the pixel is used. For linear filtering, however, we consider all nearby texels and take the weighted average. This causes the result to appear purple in this example, which in effect causes textures to generally appear more blurred as opposed to pixelated. A similar, but reverse process occurs for the magnification filter.



For our purposes, we only choose the minification and magnification filters for the loaded texture. We can do this through the following code:

```
//Apply filtering parameters.
//When a pixel spans multiple texels, use a linear computation for the
pixel.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
//When a pixel is completely inside a texel, use the color at the spot
(nearest).
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

There are other texture parameters, but these should be sufficient for getting us started. The loader, generally shouldn't be too aware of what settings the caller may actually wish to use.

The final thing to specify a texture in OpenGL is to send the image data we extracted from before to the GPU. Doing this also requires that the corresponding texture object is bound, and is done with the following snippet of code:

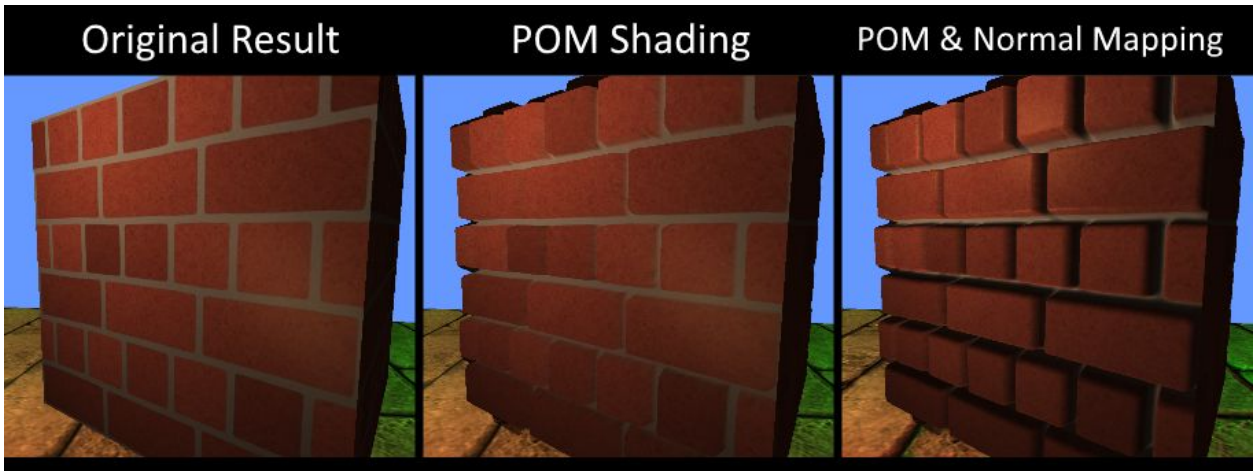
```
//Send image data to texture object.
//Note: OpenGL expects texture to be built starting from bottom-left.
if (attributeBitsPerPixel > 0)
{
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imageSpec.width,
imageSpec.height, 0, GL_RGBA, GL_UNSIGNED_BYTE, imageData);
}
else
{
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageSpec.width,
imageSpec.height, 0, GL_RGB, GL_UNSIGNED_BYTE, imageData);
}
```

The above code allows the texture loader to handle either RGB or RGBA textures and uses the appropriate parameters depending on the contents of the TGA file Another bit of information relevant to loading our TGA images is that OpenGL expects a pixel origin at the bottom-left of the image. The TGA file format supports multiple origins, but our TGA loader has been designed only to work with bottom-left origins. To support other origins, we would need to store the pixels in our image data array in a different manner.

Once we have specified the basic texture parameters and have sent the image data to the GPU, we can use the texture in our rendering. The texture will need to be accessed from within shader through the use of uniform *sampler2D* variables, and we'll need to pass texture coordinates as vertex attributes as well. This requires that we specify the texture coordinates in our model data.

Parallax Occlusion Mapping

Now that we have a `TGALoader`, we can begin to look at a parallax occlusion mapping (POM) shader. POM shading is an illusion that attempts to make surfaces appear like they have depth, when in fact, there is no real depth at all. For this to work, textures are needed. There needs to be two such textures, one containing image data (the diffuse map), and the other containing depth data (the depth map). Depending on the viewing angle and the information from the depth map, we distort the texture coordinates of the diffuse map, causing it to stretch and shrink in a way that fakes depth. For this to work, however, we need a good depth & diffuse map pair. Below illustrates the effects of what POM shading can achieve:



As the above image illustrates, POM shading can have a dramatic effect on the render result. However, alone, it can be difficult to see. Typically a different shading technique, known as normal mapping is used in order to pronounce the effects of the POM effect. For our purposes, we will first look at how normal mapping is achieved, then extend that to include the POM effect.

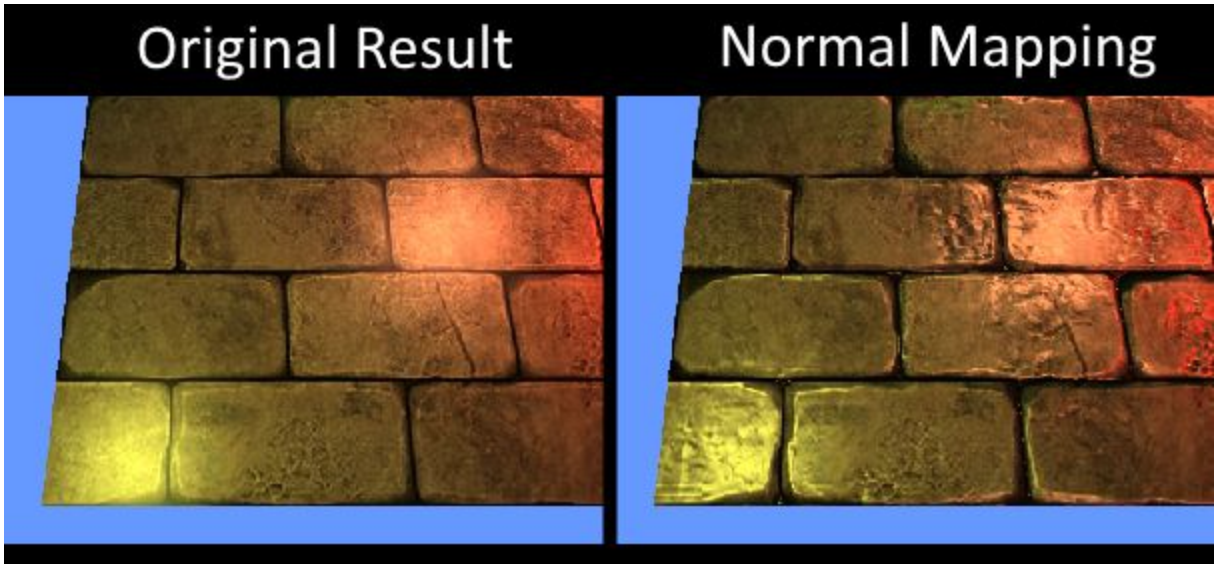
Normal Mapping

For normal mapping to be implemented, we need a diffuse map and another special texture, known as a normal map. A normal map is an image that contains normal vectors at each pixel location. A transformation occurs so that RGB colors can be represented as an XYZ vector. This is done as shown below:

$$(X,\ Y,\ Z) = 2 \bullet (R,\ G,\ B) - (1,\ 1,\ 1)$$

Effectively, it converts the range of [0, 1] into [-1, +1]. Also, it is important to note that the normal map stores normals in *tangent space*, relative to the surface it is applied to.

With these new normal vectors, we can use per-fragment lighting that corresponds directly to the texture image (as opposed to the interpolated normals from the vertex shader). When combined with the Phong Lighting Model, this allows the reflective properties of a surface to appear quite realistic as now surfaces can reflect as if they are rough or have depth. The image below illustrates the how lighting is changed:



On the left image, the lights cause reflection as if the stone is a flat surface (notice how the bright spots form a radial type of reflection that is perfectly blurred). On the right image, with normal mapping enabled, we see that these radial reflections are now distorted. This is due to the use of a different normal vectors at each pixel instead of the normal orthogonal the surface.

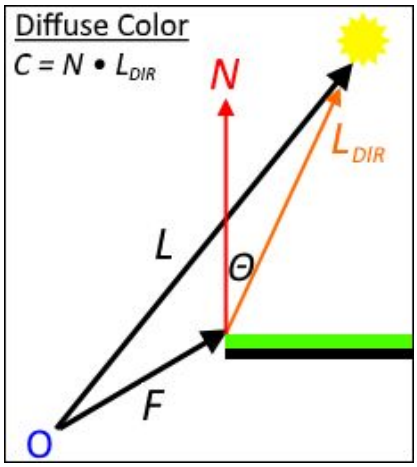
Phong Lighting Model

The Phong Lighting Model, which we briefly mentioned before, is a lighting model that allows for a realistic result. It works by computing the different lighting components (ambient, diffuse, and specular) on a per-fragment basis instead of a per-vertex basis.

For the ambient component of the light, we just use a small percentage of the sampled texture value. Inside the fragment shader, this looks as such:

```
//From diffuse map.
vec3 diffuseColor = texture(diffuseMap, texCoord).rgb;
vec3 ambient = 0.1 * diffuseColor;
```

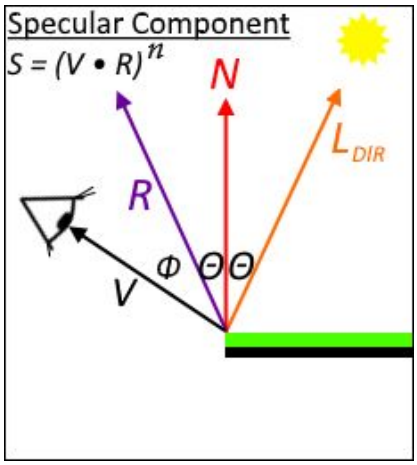
Next, for the diffuse component, we must take the dot product between the vector pointing from the fragment position to the light source (call this L_{DIR}) with the normal vector (N) at that fragment position. The figure on the right illustrates the math. This dot product represents how strong the diffuse reflection is. When the normal vector and light direction are perfectly in alignment (when parallel), the dot product will be 1. When completely orthogonal, the dot product will be 0. Effectively, it ranges from [0, 1]. We can then multiply this diffuse reflection scalar with the RGB color from the texture image to obtain the diffuse color.



The shader includes the following code to compute the diffuse color:

```
vec3 lightDir = normalize(lightPos - fragPos);
float dist = abs(length(lightPos - fragPos));
float diff = max(dot(lightDir, normal), 0.0f);
vec3 diffuse = diff * diffuseColor;
```

For the specular component, we use similar math, except now we are concerned with the viewing direction (V) and the reflected light direction (R). When we take the dot product of these two vectors, we end up with a scalar similar to before, but for the specular reflection. Again, its range is within [0, 1].



When we take this scalar value to the power of a value n, which is the surface material's *shininess*, we end up with a close approximation to how different material's reflect specular light in the real world. An increased shininess causes the specular light to concentrate around the point of reflection, while a lower shininess causes the specular light to dilute across a broader region.

Once we have the specular component which also falls in the range [0, 1], we then multiply it by the *light's* color. This causes the result to approximate the effect of the light appearing reflected through the surface. The code for computing the specular color is shown on the next page.

```
//Specular light.
float specularStrength = 0.2f;
float shininess = 32;
vec3 viewDir = normalize(viewPos - fragPos);
vec3 reflectDir = reflect(-lightDir, normal);
float spec = pow(max(dot(viewDir, reflectDir), 0.0f), shininess);

vec3 lighColor = vec3(1.0f, 0.76f, 0.39f)
vec3 specular = specularStrength * spec * lightColor;
```

The final step of the Phong Lighting Model is to combine the ambient, diffuse, and specular colors. This is shown here:

```
vec4 color = vec4(ambient + diffuse + specular, 1.0f);
```

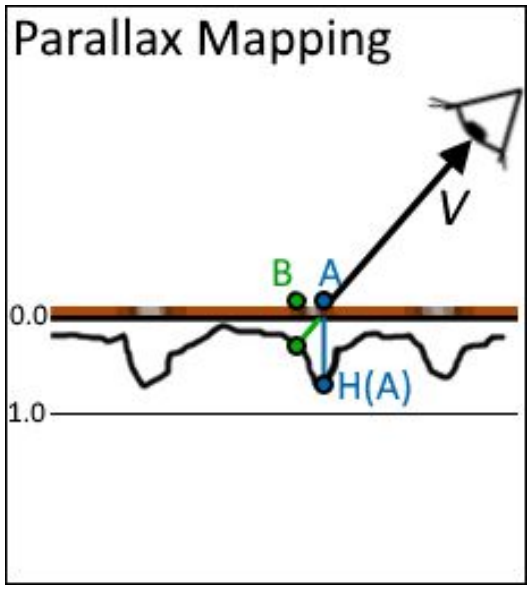
TBN Transformation

Now that the Phong Light Model has been shown, we can easily finalize the normal mapping by using the per-fragment normals (extracted from the normal map) in our lighting computations. This is good, if the surface to be rendered to is along the XY plane, but doesn't work well for other orientations. Because the normal map contains normals in tangent space, lighting will fail when the surface's normals don't point generally in the +Z direction. Since the light position, view position, and fragment position are all in world-space, either these variables, or the normal vectors must be transformed so that all computations are done in the same space.

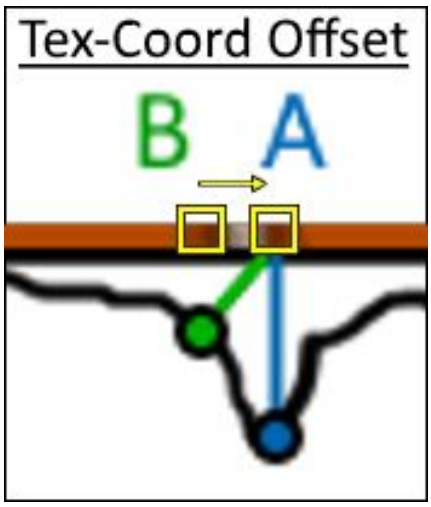
We choose to transform the light, fragment, and view positions as this is computationally less expensive. To do this, we need to create a TBN matrix, which is created from a vertex's *normal*, *tangent*, and *bitangent* vectors (hence TBN). The tangent and bitangent vectors are created as additional vertex attributes and passed right along to the vertex shader. The computation of these vectors is shown in full detail by looking at either the revised ***Cuboid.cpp*** or the ***Square.cpp*** code files. Once passed to the vertex shader, we transform the lighting parameters to put them into tangent space to be consistent with the normal vectors.

Finishing POM

Now that we have covered how normal mapping works, we can complete the discussion of parallax occlusion mapping by looking at how texture coordinates are distorted based on the viewing angle and depth map. First let's take a look at the following diagram to see how POM shading works.



In the diagram, we have a cross section of the 2D image for a brick texture. Let's assume that we are trying to determine what the distorted texture coordinate should be at point A. For this, we must first look at the fragment-to-view direction vector which is denoted as *V*. By extending this vector into the depth map, we determine that point B is what should be seen at point A. In other words, we can distort this texture such that the texels at B are used for the texture coordinate at A. This is elaborated in the next diagram.



The challenge of POM shading has thus been simplified to determining the texture-coordinate offset, that when applied to texture-coordinate A, returns the texel at B. This depends on the viewing angle *V* and the depth map information. There are several parallax mapping techniques; however, we will cover parallax *occlusion* mapping, which is a relatively inexpensive method to generate a high-quality result.

For parallax occlusion mapping, we divide the depth map's range into several layers, then we step backwards along the fragment-to-view direction vector (into the surface) which projects the current texture-coordinate offset. For each layer, we look at the depth value at the current texture-coordinate offset and compare it to the previously obtained depth value. When the depth value at the current layer is less than the depth at the previous layer, this means that the fragment-to-view direction vector has intersected the surface's depth (from the depth map) at some point between these two layers. As a final approximation to determine the texture-coordinate offset to use, we attempt to find the intersection point between the fragment-to-view vector and the surface's depth by taking a linear interpolation between the two layers. The depth at each layer is used as the weighting to determine if the intersection is closer to one layer versus the other. As we can imagine, if we increase the number of layers that the depth range is divided into, the linear interpolation will cause the texture-coordinate offset to be quite accurate. Another important observation for POM to work is that computations must be again performed in tangent space since we have a depth-map that relies on tangent space computations.

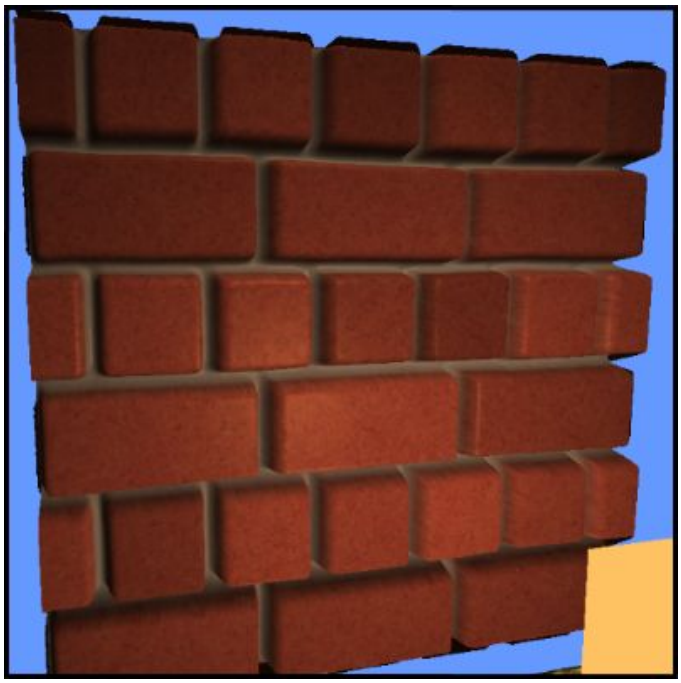
The final tweak to the POM shading algorithm is to discard any fragments if the resulting texture-coordinate wraps around the boundary. This allows edges to appear nice as it trims away the unwanted geometry that appears on the other side of the surface.

Below is the shader code for the POM algorithm:

```
vec2 ParallaxOcclusionMapping(vec2 texCoord, vec3 viewDir)
{
    const float minLayers = 10;
    const float maxLayers = 100;
    //If viewing from orthogonal, use maximum layers,
    //If viewing from parallel, use minimum layers.
    float numLayers = mix(maxLayers, minLayers, abs(dot(vec3(0.0, 0.0,
1.0), viewDir)));


    float layerDepth = 1.0f / numLayers; //Size of each layer.
    float currentLayerDepth = 0.0f;
    vec2 P = viewDir.xy / viewDir.z * 0.1f;
    vec2 deltaTexCoord = P / numLayers;
```

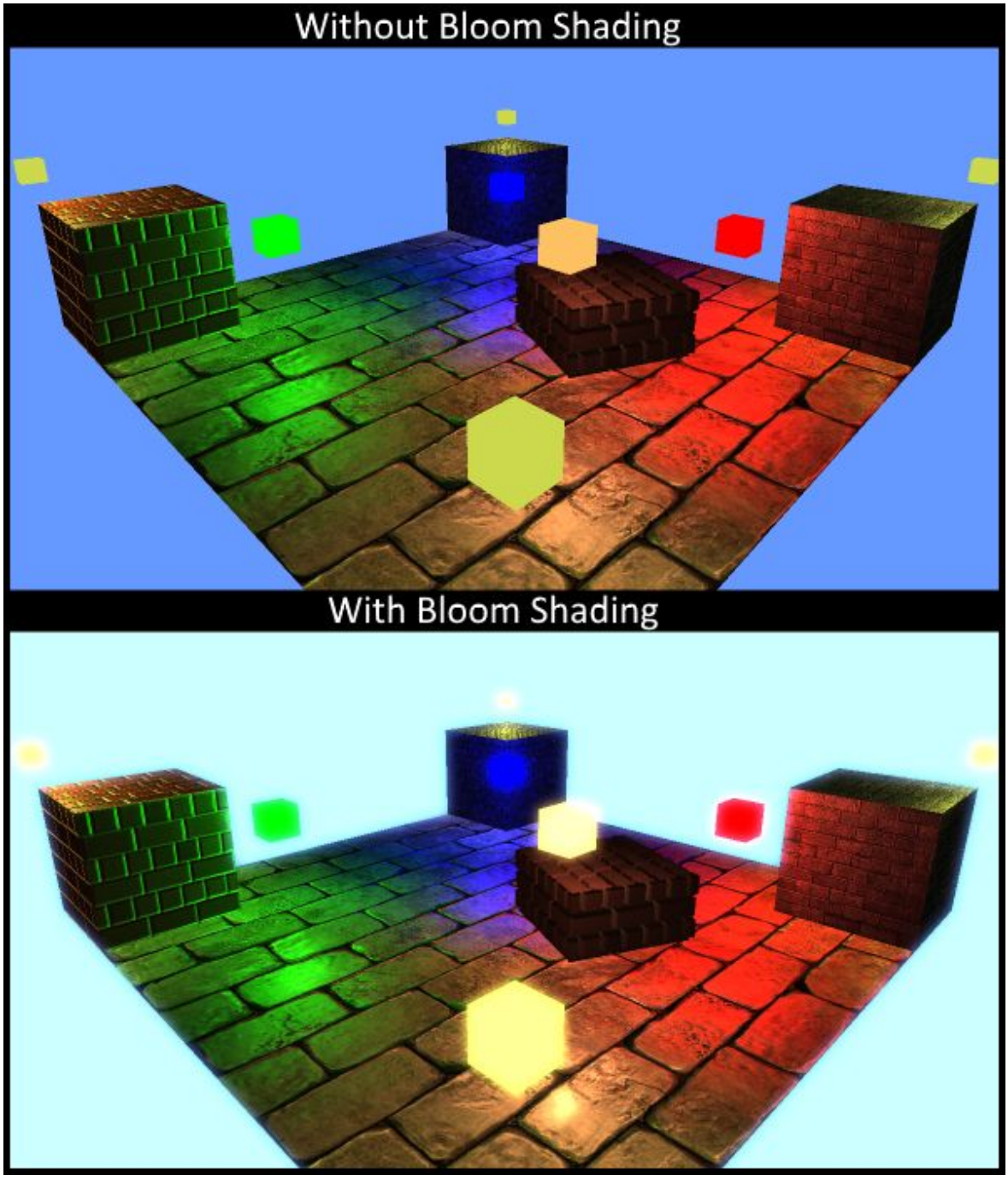
```
    vec2 currentTexCoord = texCoord;
    float currentDepthMapValue  = texture(depthMap, texCoord).r;

    while (currentLayerDepth < currentDepthMapValue)
    {
        currentTexCoord -= deltaTexCoord;
        currentDepthMapValue = texture(depthMap, currentTexCoord).r;
        currentLayerDepth += layerDepth;
    }

    //Lerp between the last two points.
    vec2 previousTexCoord = currentTexCoord + deltaTexCoord;
    float afterDepthDifference = currentDepthMapValue - currentLayerDepth;
    float beforeDepthDifference = texture(depthMap, previousTexCoord).r -
currentLayerDepth + layerDepth;

    //When weight = 1, then fully beforeDepth.
    //When weight = 0, then fully afterDepth.
    float weight = afterDepthDifference / (afterDepthDifference -
beforeDepthDifference);
    return mix(currentTexCoord, previousTexCoord, weight);
}
```

With this we get the final result of parallax occlusion mapping with normal mapping:



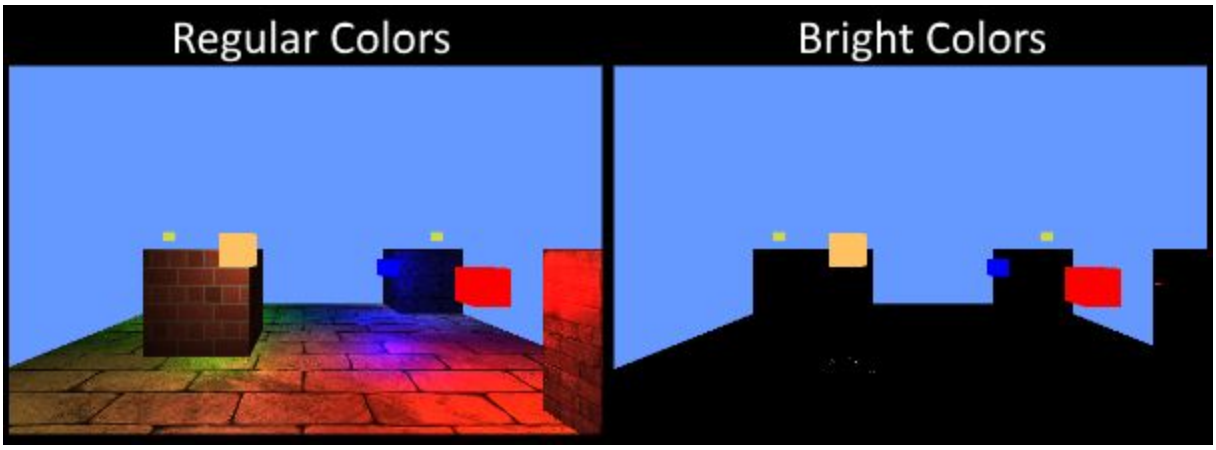
Bloom Shading

The final objective of this assignment is to showcase the bloom shader effect. This effect is a post-processing effect, which means that it takes the result of a framebuffer, and manipulates it further before showing it to the screen. The below image illustrates what bloom shading can achieve:

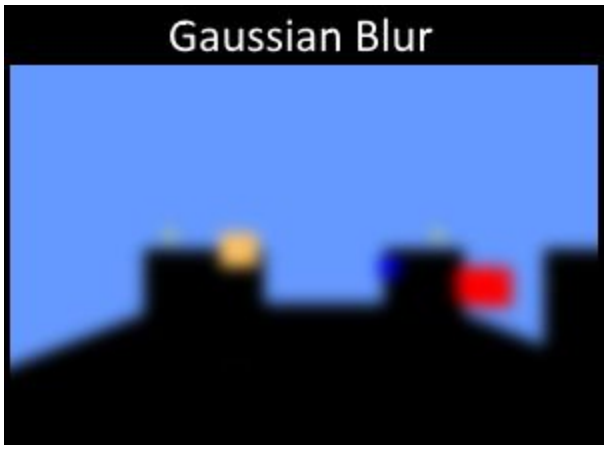


As can been seen in the image above, bright spots are more pronounced, and they tend to bleed into other regions of the screen. The bloom effect is created through the following sequence of steps:

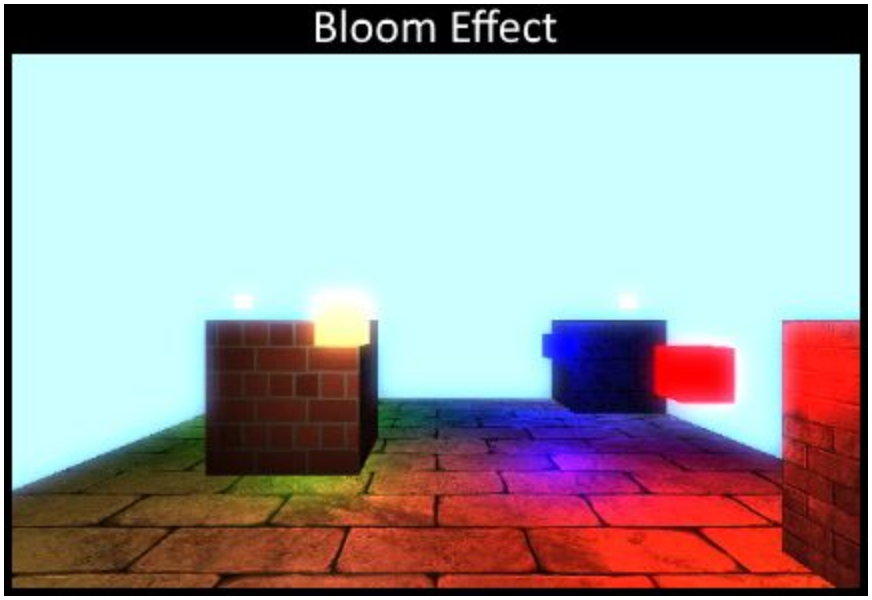
1. During the first step, we render the scene, generating two color results from the fragment shader(s). The first result is the regular color as if no effect took place. The second result is all the colors considered 'bright' beyond a certain threshold. If the color is bright enough, it appears as-is into the color buffer, otherwise black ends up as the output from the fragment shader. For our demo scene, this produces the following two color buffers:



2. The second step involves taking the bright colors outputted from the first step and blurring them so that the bright regions bleed into other surrounding areas. For this step, we will use a gaussian blur filter. The results look like this:



3. The final step is to take the blurred bright colors and additively combine it to the original result. This causes the original result to appear brighter where the colors in the blurred image are not black and also adds the bleeding effect to the original image:



Framebuffers

To understand how to implement the bloom effect, we must first understand how to use framebuffers to process the result. Recalling from the first assignment, the OpenGL pipeline always produces a default framebuffer. This framebuffer contains a color buffer, and optionally a depth buffer or stencil buffer. Previously, we have rendered all content into the default framebuffer's color buffer and depth buffer. Then OpenGL handles the rest by sending this to the window's client area.

However, for the bloom effect, we must take the result of our rendering and use it for further processing. Rather than using the default framebuffer, we create a new framebuffer object to render into and we capture the result by binding its color buffer to a texture object. Before we cover the code, we must first remember that for the bloom effect, we need two color results (one for the regular colors and one for the bright colors). This means that our rendered result must have two color buffers attached to the same framebuffer. We can do this relatively simply using the following calls:

```
glGenFramebuffers(1, &bloomFBO);
glBindFramebuffer(GL_FRAMEBUFFER, bloomFBO);
glGenTextures(2, colorBuffers);
for (int i = 0; i < 2; i++)
{
```

```
      glBindTexture(GL_TEXTURE_2D, colorBuffers[i]);
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 600, 400, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
      glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + i,
GL_TEXTURE_2D, colorBuffers[i], 0);
}
GLuint attachments[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, attachments);
```

As seen if the above code, we start by generating a *framebuffer object* (FBO) and then we make it currently active to the OpenGL context by binding it. We then generate 2 textures which will be used that the color buffers for the two results (regular colors and bright colors). When these are created, we just specify how large these buffers are by making a call to **`glTexImage2D`** without specifying any image data. Afterwards, we specify the texture parameters and link the color buffer attachments 0 and 1 to the respective textures.

To tell OpenGL that we will be rendering into the two color buffers, we must make a call to **`glDrawBuffers`**, which enables fragment shaders to output to multiple color buffer attachments.

The final step in creating our framebuffer is to provide it with a depth buffer. We do this using a *render buffer object* (RBO), which allows us to easily specify the storage of the depth buffer and link it to the framebuffer in use:

```
//Attach depth buffer.
glGenRenderbuffers(1, &rboDepth);
glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 600, 400);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, rboDepth);
```

Once we have done this, we need to be mindful that is it possible that the window to be resized. For this resize, we also resize the linked storages for the color and depth buffers within the render loop. The details for resizing the storages are shown in ***Main.cpp***.

With this framebuffer in use, we can now render the scene to it. For there are two shaders. The first shader renders most of the scene and supports normal mapping. This was done to allow normal mapping to be seen in combination with the bloom effect. The second shader is a simple

shader that only extras the bright colors from the rendered result. This is used to right the colored cubes that represent the different light sources. The shader for the lights looks like this:

```
#version 330 core

layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;

in vec3 fragColor;

void main()
{
	FragColor = vec4(fragColor, 1.0f);

	//No need to check for brightness, lights should always be sent.
	BrightColor = FragColor;
}
```

Of particular interest, we see that this shader has two output colors, located at color buffer attachment 0 for the regular colors and color buffer attachment 1 for the bright colors.

After rendering into this FBO, we can take the bright color's color buffer (recalling this was linked to a texture object) and render it to a rectangle the size of the screen. When we render it, we will need to use a different FBO, as well as the gaussian blur shader. We have used a flexible and efficient gaussian blur algorithm which can only blur along one axis at a time. For this reason, we actually will need two FBO's to perform the blur, and we will pass the rendered result back and forth, blurring both horizontally and vertically each time we pass the rendered result to the next FBO. This time, instead of having two color buffers for one FBO, we have two FBO's, each with one color buffer. The code to set up these FBO's is shown below:

```
//Create ping-pong FBO's to perform gaussian-blurring.
glGenFramebuffers(2, pingPongFBOs);
glGenTextures(2, pingPongColorBuffers);
//Attach color buffers (1 buffer per ping-pong FBO)
for (int i = 0; i < 2; i++)
{
	glBindFramebuffer(GL_FRAMEBUFFER, pingPongFBOs[i]);
	glBindTexture(GL_TEXTURE_2D, pingPongColorBuffers[i]);
		glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 600, 400, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

```
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
	glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, pingPongColorBuffers[i], 0);
}
```

For these FBO's there is no need to set up depth buffers as there will not be any depth testing. Once again, we also will need to resize these color buffer storages based on the actual window size, and we do this in the render loop as well.

Gaussian Blur

Let's now take a look at how gaussian blur is computed more specifically. To compute the blurred fragment result at any given location, we must take the current fragment's color contribution with a large weighting, then combine it with nearby fragments with lower weights as they become more radially distant from the original fragment. Since for our color buffer, fragment's correspond 1:1 with texels, this makes the mathematics for referencing the nearby fragments relatively simple. The code for the gaussian blur shader is shown below:

```
#version 330 core

out vec4 color;

in vec3 fragColor;
in vec2 fragTexCoord;

uniform sampler2D image;
uniform bool horizontal;
uniform float weight[5] = float[]  (0.227027f, 0.1945946f, 0.1216216f,
0.054054f, 0.016216f);

void main()
{
	float dist = 2.0f;

	//Get texel size (width & height).
	vec2 texelSize = dist / textureSize(image, 0);

	//Contribution of the blur from the current fragment.
	vec3 result = texture(image, fragTexCoord).rgb * weight[0];

	//Allow nearby fragments to affect the current fragment's output as
well.
```

```
	if (horizontal)
	{
		//Horizontal Pass
		for (int i = 1; i < 5; i++)
		{
			result += texture(image, fragTexCoord + vec2(texelSize.x
* i, 0.0f)).rgb * weight[i];
			result += texture(image, fragTexCoord - vec2(texelSize.x
* i, 0.0f)).rgb * weight[i];
		}
	}
	else
	{
		//Vertical Pass
		for (int i = 1; i < 5; i++)
		{
			result += texture(image, fragTexCoord + vec2(0.0f,
texelSize.y * i)).rgb * weight[i];
			result += texture(image, fragTexCoord - vec2(0.0f,
texelSize.y * i)).rgb * weight[i];
		}
	}
	//Send gaussian-blurred result to colorbuffer.
	color = vec4(result, 1.0f);
}
```

As mentioned earlier, we use the above shader for both vertical and horizontal blur passes, this means we must control how this shader works using the uniform variable, *horizontal*. Another note, is that we have taken the texel size and multiplied it by some distance value. This allows the blur to appear larger without having to run the shader as many times.

Now that the gaussian blur shader and FBO's have been set up, we can resume looking at how the bloom effect uses the blurring as part of its effect. We take the original output from the bright color's color buffer, then blur horizontally. Afterwards, we blur that result vertically. We repeat pairs of horizontal and vertical blurs as many times as we want, then send the final blurred result to the next stage of the post-processing effect. To blur back and forth, we use the ping-pong FBO's created above and render into the opposite one each time. The code to pass the result back-and-forth is shown next:

```
bool horizontal = true;
int gaussianBlurPasses = 5;
for (int i = 0; i < gaussianBlurPasses * 2; i++)
{
	glBindFramebuffer(GL_FRAMEBUFFER, pingPongFBOs[horizontal]);
	glClear(GL_COLOR_BUFFER_BIT);
	glUniform1i(glGetUniformLocation(gaussian_blur, "horizontal"),
horizontal);

	glActiveTexture(GL_TEXTURE0);
	//On first iteration, perform gaussian blur on the bright colors
	//then render into the horizontal framebuffer.
	//Every other iteration should take from the other
	//framebuffer's colorbuffer.
	glBindTexture(GL_TEXTURE_2D, (i == 0) ? colorBuffers[1] :
pingPongColorBuffers[!horizontal]);
	glUniform1i(glGetUniformLocation(gaussian_blur, "image"), 0);

	//Draw the rectangle to match the screen.
	model = glm::mat4();
	model = glm::translate(model, glm::vec3(window->width / 2.0f,
window->height / 2.0f, 0.0f));
	glUniformMatrix4fv(glGetUniformLocation(gaussian_blur, "model"), 1,
GL_FALSE, glm::value_ptr(model));
	renderTarget->Draw();
	horizontal = !horizontal;
}
```

When the blurring is complete, the final result will be stored in the first FBO, which was to store the vertical blurs. We take this FBO's color buffer, along with the color buffer from the first step and then additively combine them to complete the bloom effect. This is done using the final post-processing fragment shader:

```
#version 330 core

in vec3 fragColor;
in vec2 fragTexCoord;

out vec4 color;

uniform sampler2D scene;
uniform sampler2D bloomBlur;
```

```
uniform int bloomEnabled;

void main()
{
	vec3 sceneColor = texture(scene, fragTexCoord).rgb;
	vec3 bloomColor = texture(bloomBlur, fragTexCoord).rgb;
	color = vec4(sceneColor + bloomColor * bloomEnabled, 1.0f);
}
```

For this shader, we should render into the default framebuffer so that the bloom effect appears on the screen. And that's it!

One thing that could be improved with the bloom effect is to use *high dynamic range*, or HDR. This would allow the lighting computations to be more accurate and adjustable based on an exposure setting. As we can seem from the images shown earlier, there are some splotches of color that have lost detail because of clamping to the maximum value for one of the R, G, or B components. With HDR, we can temporarily exceed this boundary using floating-point color buffers, then tone the HDR colors back to LDR (low dynamic range, or [0, 1] range) in a consistent manner. Though, implementing HDR exceeds the scope of this assignment and remains as an extracurricular task.

References

https://learnopengl.com/
https://en.wikipedia.org/wiki/Truevision_TGA
http://www.dca.fee.unicamp.br/~martino/disciplinas/ea978/tgaffs.pdf
https://en.wikipedia.org/wiki/Phong_reflection_model