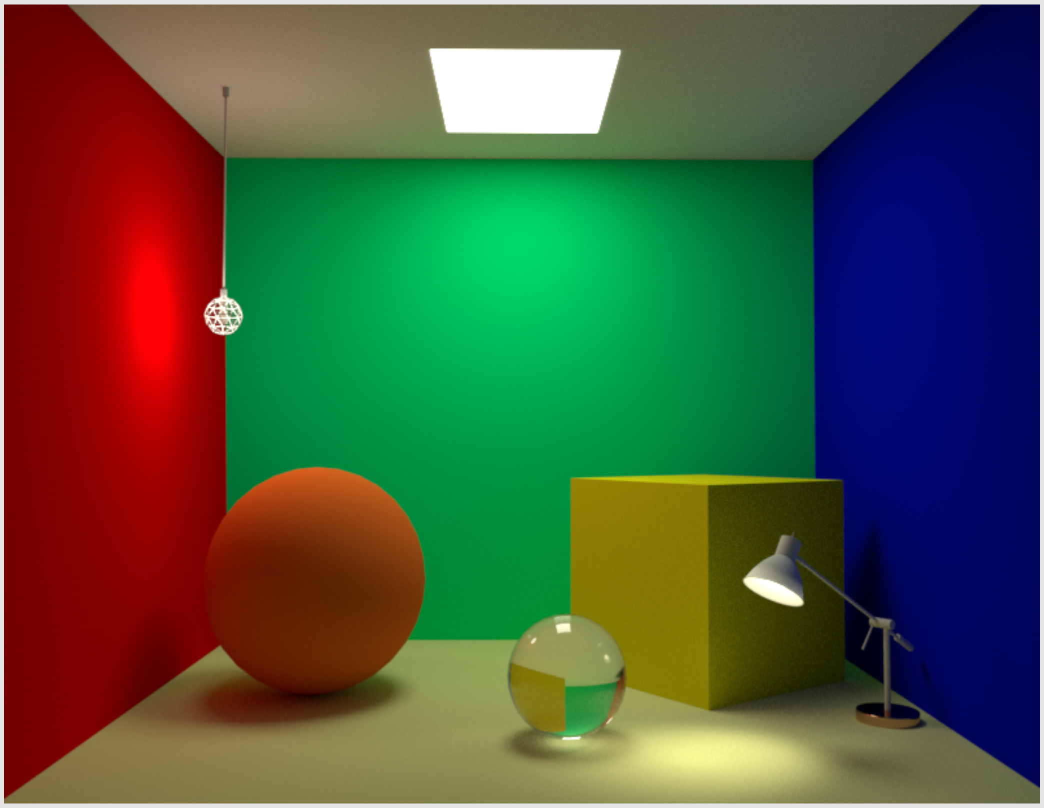


LIGHT CASTING IN OPENGL & GLSL WITH C++

By Chad Jordan

August 20th, 2011

Introduction

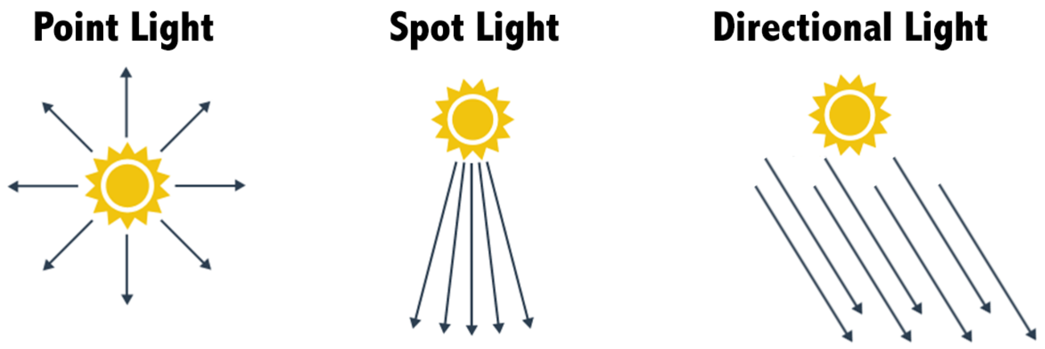
In this guide you will learn:

1) How to conceptualize and implement light casting for 2D and 3D objects in OpenGL
2) How to understand and apply GLSL attributes to objects in 2D & 3D space
3) Light attenuation, and symmetric shadowcasting, with OpenGL & GLSL

Computer Graphics has been a long running study in the field of computer science, and the origins of this research goes back to the 1960's and 70's. Over these decades of research, OpenGL has remained the cornerstone of nearly all on-going computer graphics implementation and installation for computer systems world-wide. Without light casters, any programs that you write would just output a black window with no visible content. This guide will be very similar to my previous guide on Interactive Computer Graphics, but more of an emphasis on light casters within the OpenGL API in C++ programming. The intention of this knowledge is to vastly improve your understanding of computer graphics shading languages, and how to apply higher quality to your OpenGL projects. The higher quality of illumination you achieve means the more realistic your scene will display. The research of Computer Graphics for greater visuals is dependent upon greater clarity. Forensic science and research are better understood when presented with a greater quality of simulated sequences. This is one vital area where computer graphics comes in. These simulations really need better detail and that detail will not display as clearly if we don't have better illumination in our scenes. You will need to be fairly comfortable mathematically while reading this guide as I will be discussing principles of linear algebra, geometry, vector mathematics, and physics, along with ray casting, material attributes, emissive objects, and volumetric light. Like most of my previous guides, this document will require at the very least, an intermediate level of programming knowledge; a comfortable level of reading over the ins and outs of computer graphics, and how light casting works in the virtual world. With all of this in mind, let's jump in.

Contemplating Casters

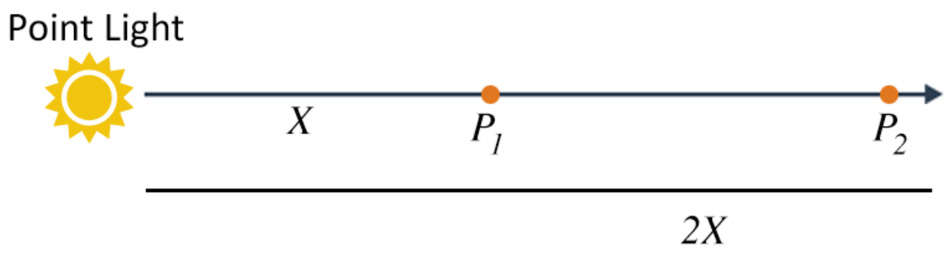
A light source that *casts* light upon objects is called a light caster. There are three primary types of light sources in OpenGL known as **point** lights, **spot** lights, and **directional** lights. Beyond that, we have techniques we can use to create different shading/lighting effects such as light



mapping, emissive objects *(emitting light from objects)*, specular reflections, diffuse shaders, and ambient occlusion/global illumination. Looking to the above example, point lights are light sources at a specific position that scatter light in all directions from said light position. These are typically good to spread out across your scene, almost as if you are wanting to place them into multiple lamps that are in multiple locations of a room. In most 3D applications we'd like to simulate a light source that only illuminates an area close to the light source and not the entire scene. This is why we use **point** lights. To understand this on a deeper level, point lights function as a **single position** *(light is evenly spread out in all directions)* artificial light and we can calculate their intensity as the light increases or decreases with a process known as **The Attenuation Factor**. In computer graphics, this method is how we are able to mimic the behavior of point lights. In the real world, light attenuation follows the **Inverse Square Law** in physics which says, the intensity of an effect such as illumination or gravitational force changes in inverse proportion to the square of the distance from the source. Therefore, in this particular instance, the decrease in light intensity equal to the inverse of the square of the distance between the light source and the object.

When looking at the Inverse Square Law mathematically, we would state the ratio between the light intensity at two different points equals to the square of the inverse ratio of their distance from the light source.

$$\frac{i_1}{i_2} = \left(\frac{d_2}{d_1}\right)^2$$

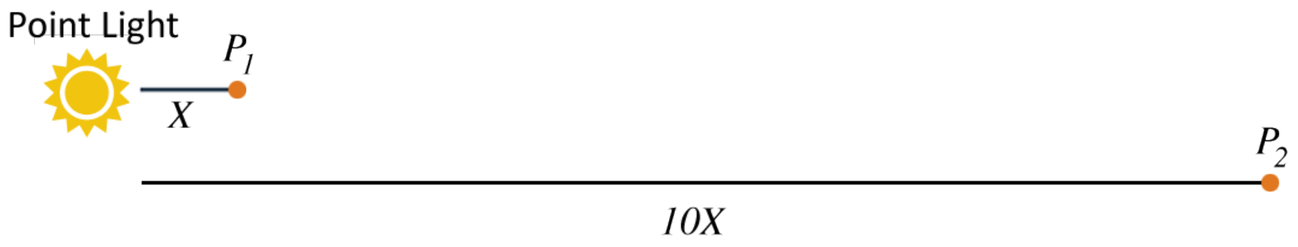


$i_1 = 4 * i_2$

For example, let’s say you have two points from the center of the point light, represented by P_1 and P_2 at a distance of X and $2X$ then the intensity of i at point 1 will equal four times the intensity at point two.

$i_2 = i_1 / 4$

In other words, the intensity at point 2 which is at twice the distance than point 1 will be a quarter of the intensity at point 1 which is closer to the center of the point light.



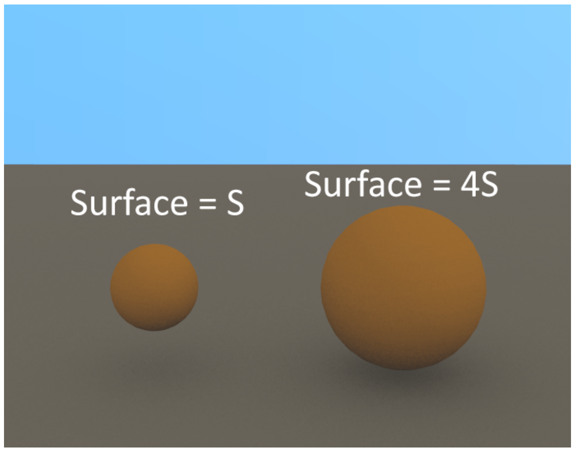
$i_1 = 100 * i_2$

$i_2 = i_1 / 100$

If the distances are X and 10X, then the intensity of the distant point will be 100th the intensity at the closer point and so forth.

We know this to be correct because the equation that states the surface area of a sphere is *4* times Pi times the square of the radius. If the radius increases by a factor of *2*, the surface area increases by a factor of *4*.

$$S = 4\pi r^2$$



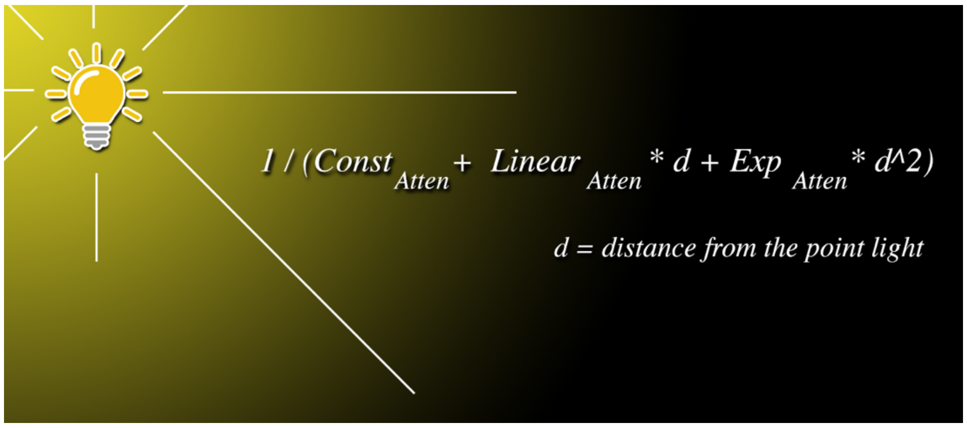
This means that the same number of photons are distributed across a surface which is *4* times larger when the distance from the light source increases by a factor of *2*. The Sphere Surface Area equation allows us to calculate our light attenuation factor, and in the case of a point light in 3D graphics with the inverse square law, it is represented by a geometric point which means as we get closer to the light source, the intensity approaches infinity.

$$\infty \longleftarrow \frac{i_1}{i_2} = \left(\frac{d_2}{d_1}\right)^2 \longrightarrow 0$$

We can expect the maximum of the attenuation factor to be *1*, otherwise we will increase the light intensity. Without getting too bogged down with the semantics of mathematical terms, we can use a more simplified lighting equation where the light intensity of the point light is multiplied by an attenuation factor that will be calculated using distance and a few other parameters that we can control.

$$0 <= \text{Attenuation Factor} <= 1$$

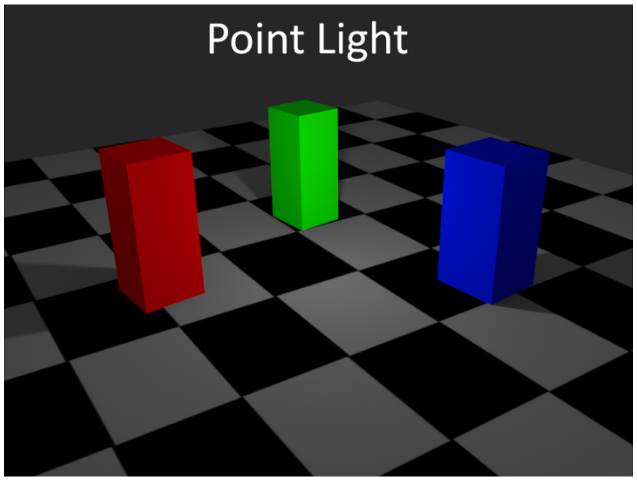
The Attenuation Factor is a fraction with the numerator being *1* and the denominator being the sum of several terms. First, we have a constant attenuation term. Notice if we set this term to be *1* we basically guarantee that the final attenuation factor will be no more than *1*, because in general, all three terms in this equation are expected to be positive. In computer graphics, most instances you will want the constant attenuation term to be *1*. The second attenuation term is known as the linear attenuation because it is multiplied by the distance. We can expect that as the distance increases, this part of the denominator



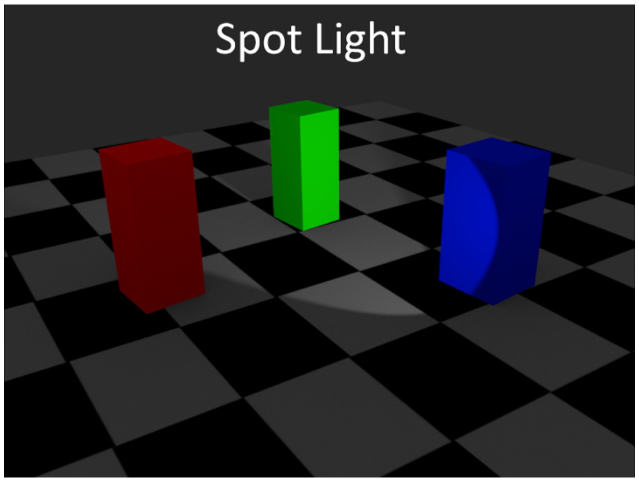
$$1 / (Const_{Atten} + Linear_{Atten} * d + Exp_{Atten} * d\text{^}2)$$

d = distance from the point light

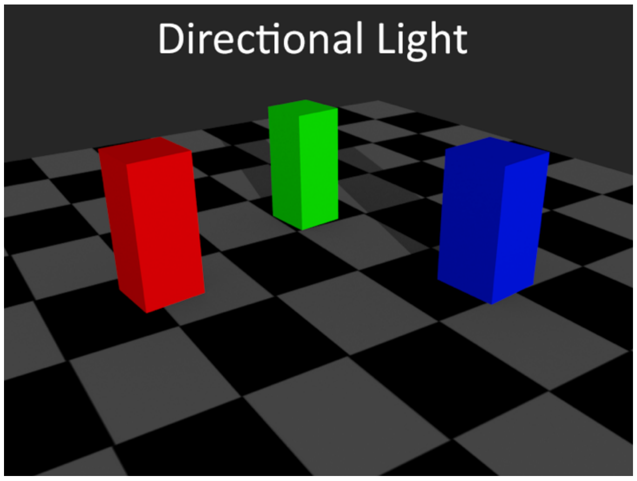
increases on a linear scale. Lastly, we have the exponential attenuation term which is multiplied by the distance which is then raised to the power of *2*. As the distance increases this part of the denominator increases exponentially. The **spot** light does share similar characteristics with the point light. They both have a position in the 3D world and since they are man-made their intensity attenuates as the distance from the object to the light source grows. The unique characteristic of the spot light is that it has a specific direction. We expect that the spot light to affect specific objects depending on the angle of its axis. Here are a couple of screenshots from a C++ 3D program that I wrote with OpenGL to sample light sources, and below each image some OpenGL code for each attribute.



```
struct LightAttenuation  {
        float Constant = 1.0f;
        float Linear = 0.0f;
        float Exp = 0.0f;
};

struct PointLight  {
        vec3 LocalPos;
        BaseLight Base;
        Attenuation Atten;
};

class PointLight public BaseLight  {
        public: Vector3f WorldPos = Vector3f (0.0f, 0.0f, 0.0f);
        LightAttenuation Attenuation;

        void CalcLocalPos(const WorldTrans& worldTransform);
        const Vector3f& GetLocalPos () const  {

        return LocalPosition;
        }
}
```



```
struct SpotLight  {
        vec3 Direction;
        PointLight Base;
        float Cutoff;
};

vec4 CalcSpotLight(SpotLight l, vec3 Normal)  {
        vec3 LightToPixel = normalize(LocalPos0 –
l.Base.LocalPos);
        float SpotFactor = dot(LightToPixel, l.Direction);

        if (SpotFactor > l.Cutoff)  {
            vec4 Color = CalcPointLight(l.Base, Normal);
            float SpotLightIntensity = (1.0 – (1.0 –
SpotFactor)/(1.0 – l.Cutoff));
        return Color * SpotLightIntensity;
        }
        else  {
            return vec4(0,0,0,0);
        }
}
```



```
struct DirectionalLight  {
        BaseLight Base;
        vec3 Direction;
};

uniform DirectionalLight gDirectionalLight;
uniform int gNumPointLights;
uniform PointLight gPointLights[MAX_POINT_LIGHTS];
uniform Material gMaterial;
uniform vec3 gCameraLocalPos;
```

Per these examples, I created a BaseLight structure with the color, ambient, and diffuse intensity. Diffuse intensity is also used for specular lighting. GLSL doesn't support inheritance as C++ so in the *DirectionalLight* structure we have the *BaseLight* as a member and an additional vector for the direction. In these code samples we can see the attenuation structure with the point light is composed of the *BaseLight*, the attenuation and the position in local space. We have a two-dimensional array for the *PointLight* structure and the array cannot be dynamic so you will need to set the maximum value according to the requirements of your engine. You will often use fewer lights than the maximum so we also have an integer uniform that you can update with the actual number of point light sources. *CalcLightInternal* takes the index of the light source and the normal as parameters. We start by calculating the light direction vector by subtracting the local position of the current pixel from the local position of the light source.

Next, we calculate the distance from the light source to the pixel by calling **length()** on the direction vector. This is an internal GLSL function that is a cleaner approach to calculating normalized vectors.

```
vec4 CalcLightInternal(BaseLight Light, vec3 LightDirection, vec3 Normal) {
        vec4 AmbientColor = vec4(Light.Color, 1.0f) * LightAmbientIntensity *
        vec4(gMaterial.AmbientColor, 1.0f);

        float DiffuseFactor = dot(Normal, -LightDirection);

        vec4 DiffuseColor = vec4(0, 0, 0, 0);
        vec4 SpecularColor = vec4(0, 0, 0, 0);

        if (DiffuseFactor > 0)  {
           DiffuseColor = vec4(Light.Color, 1.0f) * Light.DiffuseIntensity *
           vec4(gMaterial.DiffuseColor, 1.0f) * DiffuseFactor;
                vec3 PixelToCamera = normalize(gCameraLocalPos – LocalPos0);
                vec3 LightReflect = normalize(reflect(LightDirection, Normal));
                float SpecularFactor = dot(PixelToCamera, LightReflect);
           if (SpecularFactor > 0)  {
                float SpecularExponent = texture2D(gSamplerSpecularExponent, TexCoord0).r * 255.0;
                SpecularFactor = pow(SpecularFactor, SpecularExponent);
                SpecularColor = vec4(Light.Color, 1.0f) * Light.DiffuseIntensity * vec4(gMaterial.SpecularColor, 1.0f) *
                SpecularFactor;
           }
        }

        return Color / Attenuation;
}
```

These code samples allow us to see a small glimpse into how we can define and check their parameters against the caster's individual effects. Now that we've seen how light attenuation works, what practical purposes does this serve? From a conceptual level light attenuation is very significant in several fields, including optics, telecommunications, forensic science, and environmental science. Understanding how light behaves as it passes through different materials is crucial for designing optical systems, analyzing water quality, and assessing the transparency of substances. As such, in computer graphics, light casters play a crucial role in simulating realistic lighting and shading effects within a virtual environment. The importance of light casters can be summarized in a few key points:

1. **Realism and Aesthetics:** Light casters contribute to creating visually appealing and realistic scenes by simulating how light interacts with surfaces. Proper lighting enhances the perception of depth, texture, and shape in 3D graphics.

2. **Highlighting Features:** Light casters are essential for emphasizing certain features of objects. By strategically placing and configuring light sources, artists and developers can highlight specific areas, creating focal points and directing the viewer's attention.

3. **Materials and Reflections:** Light casters play a role in simulating how light interacts with different materials. This includes effects like specular highlights, reflections, and refractions, allowing for the realistic representation of surfaces with varying levels of glossiness, transparency, and reflectivity.

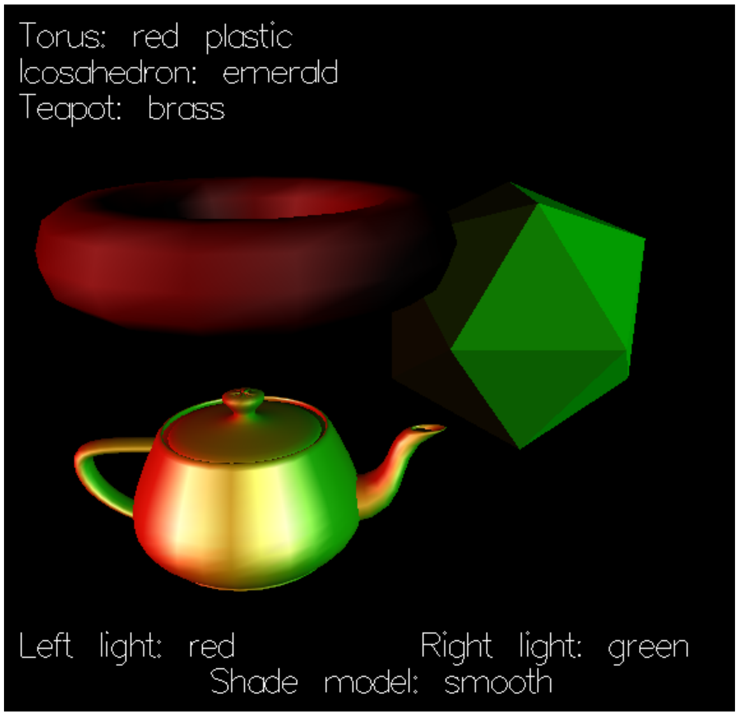
4. **Dynamic Lighting:** Light casters support dynamic lighting scenarios where the position and properties of light sources can change dynamically during runtime. This is particularly important for interactive applications like video games, where dynamic lighting enhances realism and player immersion.

5. **Global Illumination:** Advanced rendering techniques, such as global illumination, rely on realistic simulation of how light bounces and interacts with surfaces. Light casters contribute to the calculation of indirect lighting, enhancing the overall realism of the scene.

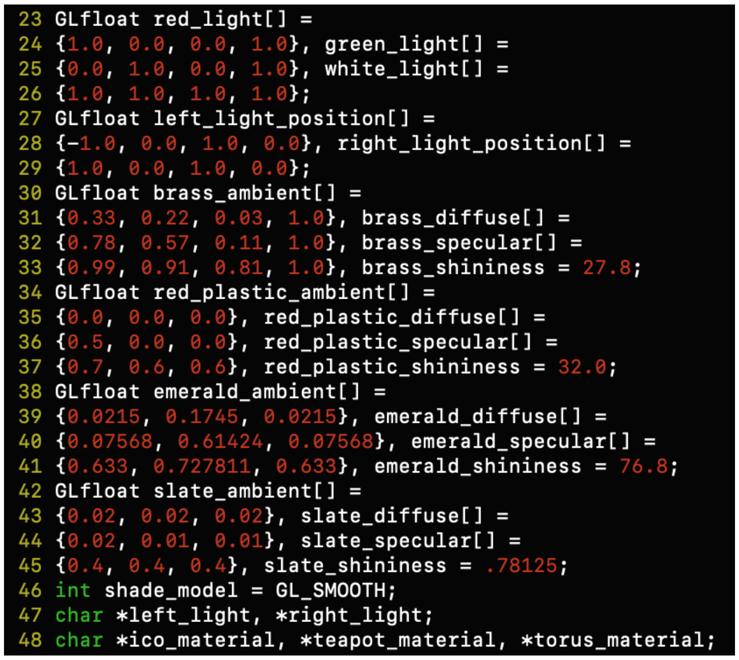
In summary, light casters in computer graphics are essential for creating visually appealing and realistic virtual environments. They contribute to the overall aesthetics, simulate lighting effects, and play a key role in shaping the visual experience of 3D graphics for scientific research. The type of shading for forensic investigation varies depending on the specific goals of the analysis, the type of data being visualized, and the preferences of forensic experts. The emphasis is often on clarity, precision and the ability to highlight important details for investigative purposes. For this, **Flat**, **Gouraud**, and **Phong** shaders are commonly used in forensic research with computer graphics. So far, we've taken a small glimpse into both the mathematical and programming perspectives from the basics of how light casting works, but we can also take a look at how adding materials, polygonal subdivision, and a few other dynamic light sources can really alter how we view lighting effects in the virtual world.

Material Attributes in OpenGL

Think about the surfaces of objects with different material attributes. Some may look duller, or not have the same depth due to not having a specular reflection. The way in which light bounces off of objects can make all of the difference in compiling/rendering your scene. Here is an example of different lighting techniques with OpenGL:



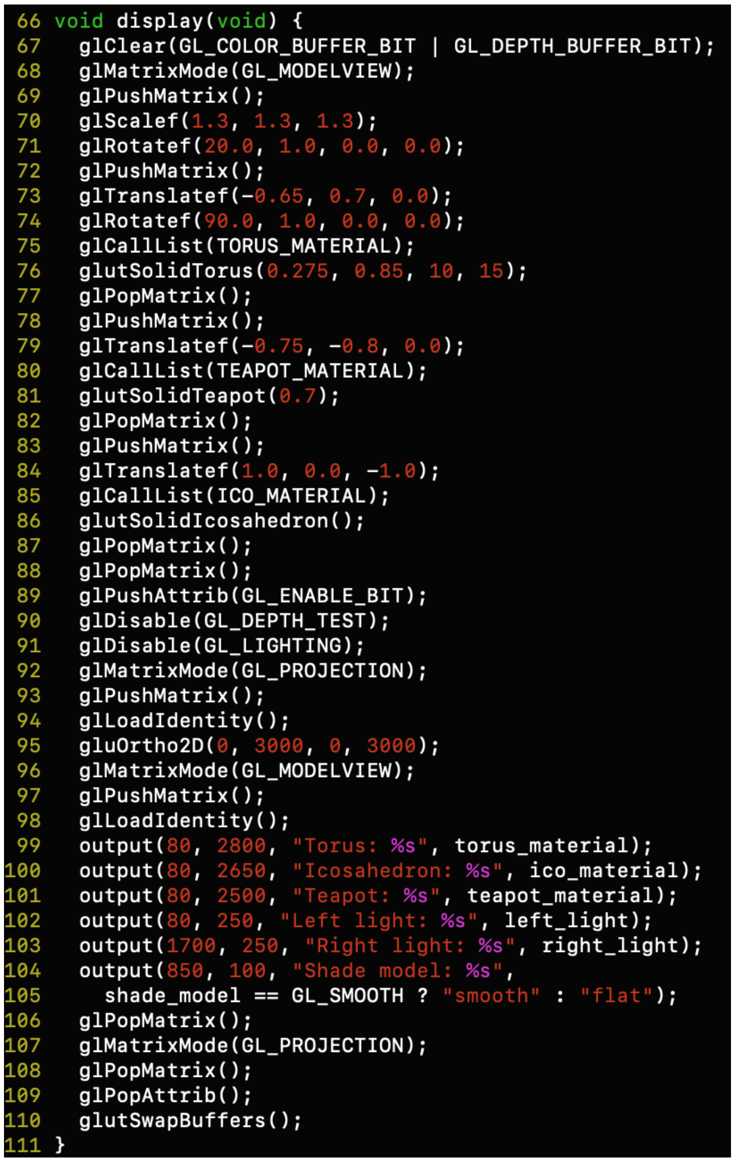
We see in this example that the torus is more diffused, whereas the icosahedron has more of an ambient look to it. The teapot is the most obvious among the three and stands out the most with specular reflection. Specular elements also define edges more vividly on our objects. But how do we achieve this? The material attributes and adjustments of the lighting makes all the difference. Let's take a deeper look at this example by jumping into the code.



```
23 GLfloat red_light[] =
24 {1.0, 0.0, 0.0, 1.0}, green_light[] =
25 {0.0, 1.0, 0.0, 1.0}, white_light[] =
26 {1.0, 1.0, 1.0, 1.0};
27 GLfloat left_light_position[] =
28 {-1.0, 0.0, 1.0, 0.0}, right_light_position[] =
29 {1.0, 0.0, 1.0, 0.0};
30 GLfloat brass_ambient[] =
31 {0.33, 0.22, 0.03, 1.0}, brass_diffuse[] =
32 {0.78, 0.57, 0.11, 1.0}, brass_specular[] =
33 {0.99, 0.91, 0.81, 1.0}, brass_shininess = 27.8;
34 GLfloat red_plastic_ambient[] =
35 {0.0, 0.0, 0.0}, red_plastic_diffuse[] =
36 {0.5, 0.0, 0.0}, red_plastic_specular[] =
37 {0.7, 0.6, 0.6}, red_plastic_shininess = 32.0;
38 GLfloat emerald_ambient[] =
39 {0.0215, 0.1745, 0.0215}, emerald_diffuse[] =
40 {0.07568, 0.61424, 0.07568}, emerald_specular[] =
41 {0.633, 0.727811, 0.633}, emerald_shininess = 76.8;
42 GLfloat slate_ambient[] =
43 {0.02, 0.02, 0.02}, slate_diffuse[] =
44 {0.02, 0.01, 0.01}, slate_specular[] =
45 {0.4, 0.4, 0.4}, slate_shininess = .78125;
46 int shade_model = GL_SMOOTH;
47 char *left_light, *right_light;
48 char *ico_material, *teapot_material, *torus_material;
```

Beginning on **line 23**, we define the arrays and variables related to the lighting and material properties through RGBA (Red, Green, Blue, Alpha) coordinates of the light. We have three arrays (red_light, green_light, and white_light) are defined to represent the different types of lights in the program. These values are in the range of [0.0, 1.0] where 1.0 is the maximum intensity. On **line 27** we create two arrays for the light positions, and then beginning on **line 30** we begin defining all of the material properties for diffuse, specular, and shininess materials for all three objects. All of these properties are crucial in OpenGL in order to achieve realistic lighting effects in 3D scenes. The values chosen for each property influence how light interacts with surfaces, affecting the visual appearance of the rendered objects. But there is still more that we can learn from this program.

This display function allows us to create our objects, and then display them based on the parameters that we pass to the function. For instance, beginning on **line 67** we are creating a

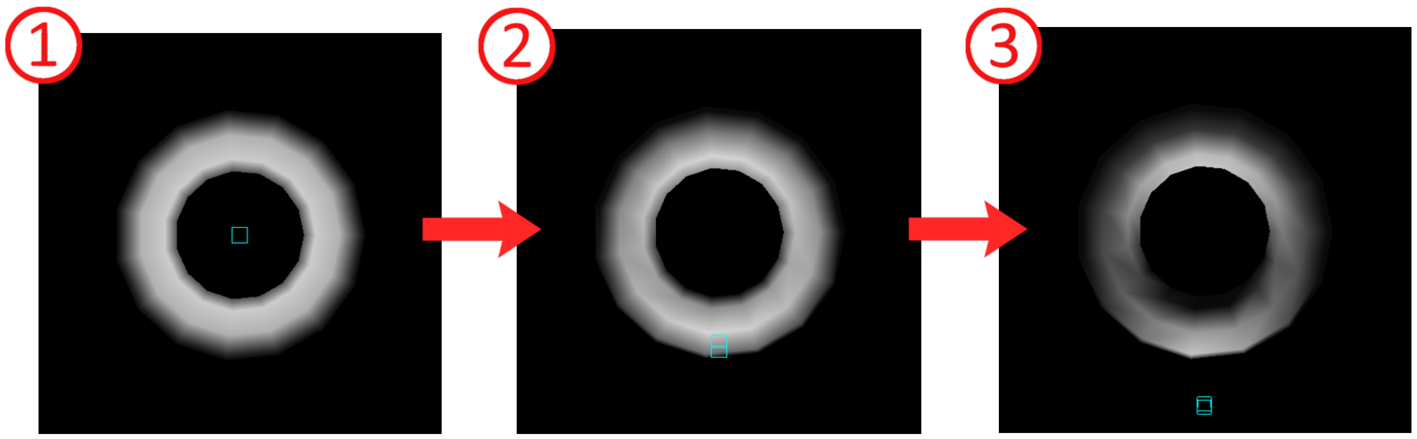


```
 66 void display(void) {
 67   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 68   glMatrixMode(GL_MODELVIEW);
 69   glPushMatrix();
 70   glScalef(1.3, 1.3, 1.3);
 71   glRotatef(20.0, 1.0, 0.0, 0.0);
 72   glPushMatrix();
 73   glTranslatef(-0.65, 0.7, 0.0);
 74   glRotatef(90.0, 1.0, 0.0, 0.0);
 75   glCallList(TORUS_MATERIAL);
 76   glutSolidTorus(0.275, 0.85, 10, 15);
 77   glPopMatrix();
 78   glPushMatrix();
 79   glTranslatef(-0.75, -0.8, 0.0);
 80   glCallList(TEAPOT_MATERIAL);
 81   glutSolidTeapot(0.7);
 82   glPopMatrix();
 83   glPushMatrix();
 84   glTranslatef(1.0, 0.0, -1.0);
 85   glCallList(ICO_MATERIAL);
 86   glutSolidIcosahedron();
 87   glPopMatrix();
 88   glPopMatrix();
 89   glPushAttrib(GL_ENABLE_BIT);
 90   glDisable(GL_DEPTH_TEST);
 91   glDisable(GL_LIGHTING);
 92   glMatrixMode(GL_PROJECTION);
 93   glPushMatrix();
 94   glLoadIdentity();
 95   gluOrtho2D(0, 3000, 0, 3000);
 96   glMatrixMode(GL_MODELVIEW);
 97   glPushMatrix();
 98   glLoadIdentity();
 99   output(80, 2800, "Torus: %s", torus_material);
100   output(80, 2650, "Icosahedron: %s", ico_material);
101   output(80, 2500, "Teapot: %s", teapot_material);
102   output(80, 250, "Left light: %s", left_light);
103   output(1700, 250, "Right light: %s", right_light);
104   output(850, 100, "Shade model: %s",
105     shade_model == GL_SMOOTH ? "smooth" : "flat");
106   glPopMatrix();
107   glMatrixMode(GL_PROJECTION);
108   glPopMatrix();
109   glPopAttrib();
110   glutSwapBuffers();
111 }
```

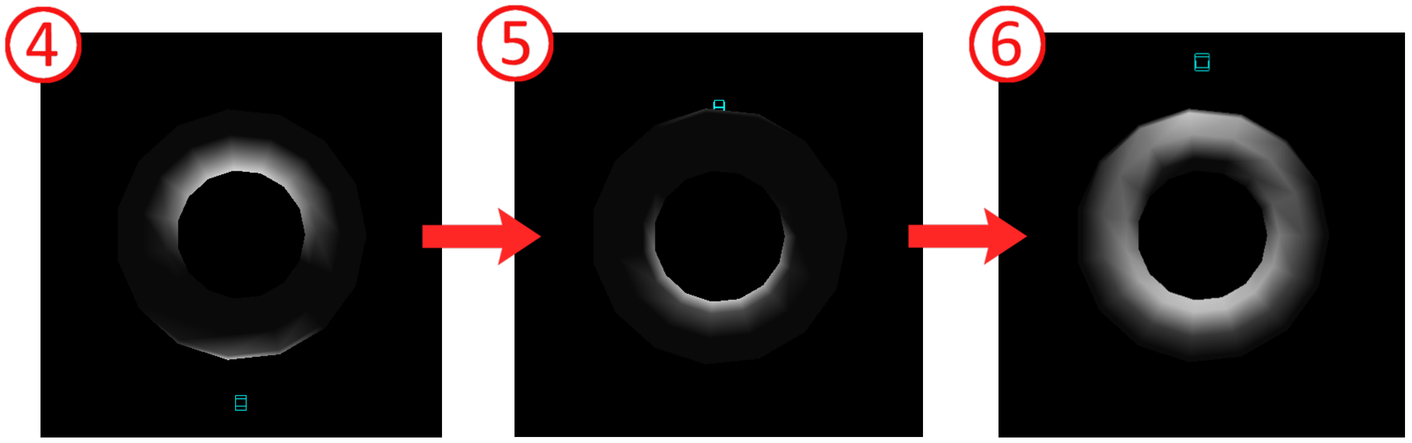
modelview matrix for the objects in the scene. On **line 70** the parameters that we pass for **glScalef** scale the objects up via the x, y, and z-axis. The **glRotatef** function applies a rotation transformation to the current matrix. The format of parameters passed to glRotatef are as follows: **(GLfloat angle, GLfloat x, GLfloat y, GLfloatz)** so in other words on **line 71**, **20.0** is the angle in degrees, **1.0** is the rotation on the x-axis, and **0.0**, **0.0** is no rotation on the y, and z-axis. This is why the objects in the rendered window above are not tilted or rotated in any other manner. The **glTranslatef** function allows us to move the location of the individual objects around in 3D space to a different location. Beginning on **line 99** we are calling a function **output** with specific parameters **80** (X-axis), and **2800** (y-axis) on screen, and then the format string (Torus: %s) and then the value of the variable **'torus_material'** indicating that the torus material is a string variable. So, what does that mean?

In OpenGL, a torus itself is a geometric primitive, a 3D shape defined by its mathematical properties, such as radius, thickness, and the number of segments. Per the above code, the concept of material in OpenGL refers to the visual properties assigned to an object, such as its color, shininess, reflection, and other visual characteristics. With this in mind, let's consider some of the geometric properties of subdivision in geometry. The number of segments in the geometry affects the allotted surfaces and points at which more light sources can bounce from. Also, since we're already on the topic of geometric primitives and specifically, the torus, I will demonstrate this with another OpenGL program that I've written in C.

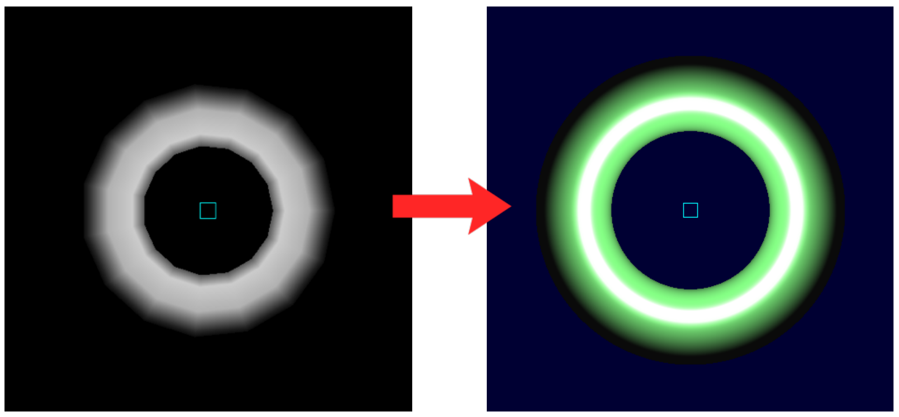
This is a simple light casting program written in C to demonstrate the lighting and transformation commands to render a torus model with a light. We move this light by a modeling transformation. The light position is reset after the modeling transformation is called. The eye position does not change. When the left-mouse button is pressed, it alters the modeling transformation *(X rotation)* by **30 degrees** with every click.



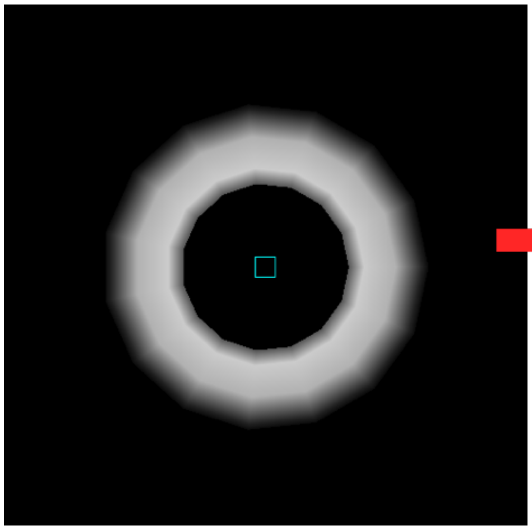
As we can see from these first three images, as the light source makes its way further around the torus, the light is going to change based on the angle of the spot light.

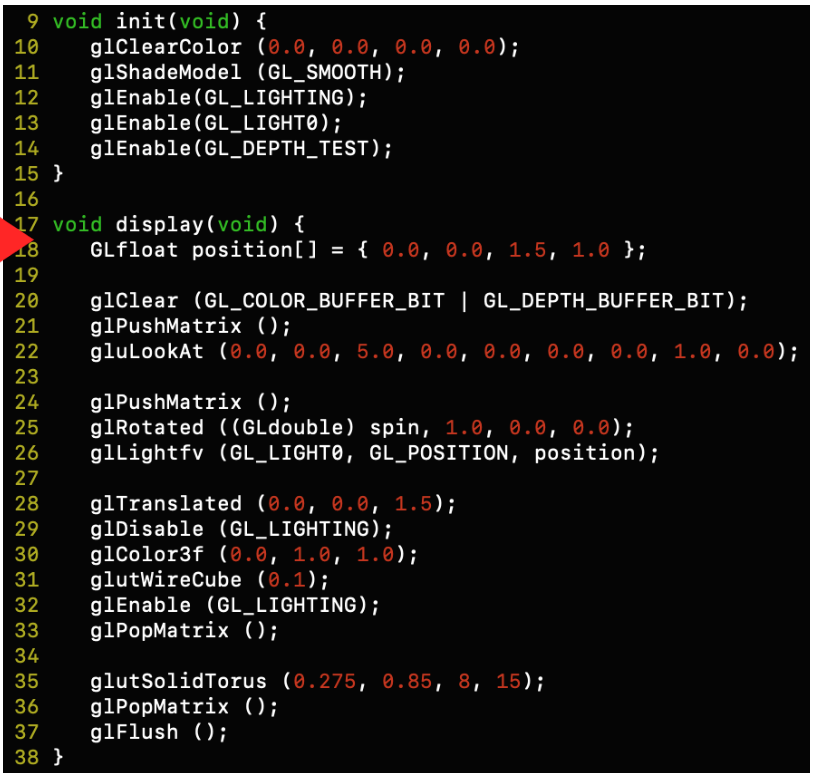


The first three screenshots demonstrate the divided segments in our geometry, but what if we wanted to subdivide the mesh of the torus in order to make a far smoother surface? We could do this by sampling the density of the mesh creating far more faces within the geometry.



At the point of origin, the difference is quite apparent. The torus on the left only has 15 samples, whereas the torus on the right has 150. The density and radius of the mesh has been significantly increased, so the amount of faces creates a much smoother surface. This change is also due to an alteration in the material that has been applied to the torus. Let's look at an example of how this is done in the code.





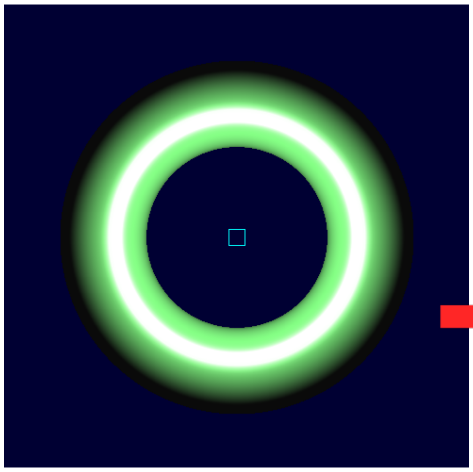
```
 9 void init(void) {
10    glClearColor (0.0, 0.0, 0.0, 0.0);
11    glShadeModel (GL_SMOOTH);
12    glEnable(GL_LIGHTING);
13    glEnable(GL_LIGHT0);
14    glEnable(GL_DEPTH_TEST);
15 }
16
17 void display(void) {
18    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
19
20    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
21    glPushMatrix ();
22    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
23
24    glPushMatrix ();
25    glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
26    glLightfv (GL_LIGHT0, GL_POSITION, position);
27
28    glTranslated (0.0, 0.0, 1.5);
29    glDisable (GL_LIGHTING);
30    glColor3f (0.0, 1.0, 1.0);
31    glutWireCube (0.1);
32    glEnable (GL_LIGHTING);
33    glPopMatrix ();
34
35    glutSolidTorus (0.275, 0.85, 8, 15);
36    glPopMatrix ();
37    glFlush ();
38 }
```

The **init** function at the top is very straight forward. We clear the color buffer by setting all RGBA values to 0. This is what we want when changing values throughout key input for the program while in runtime. **Line 11** sets the shading model to smooth shading. In smooth shading, colors are interpolated across the surface of polygons, resulting in a smooth transition between different vertex colors.

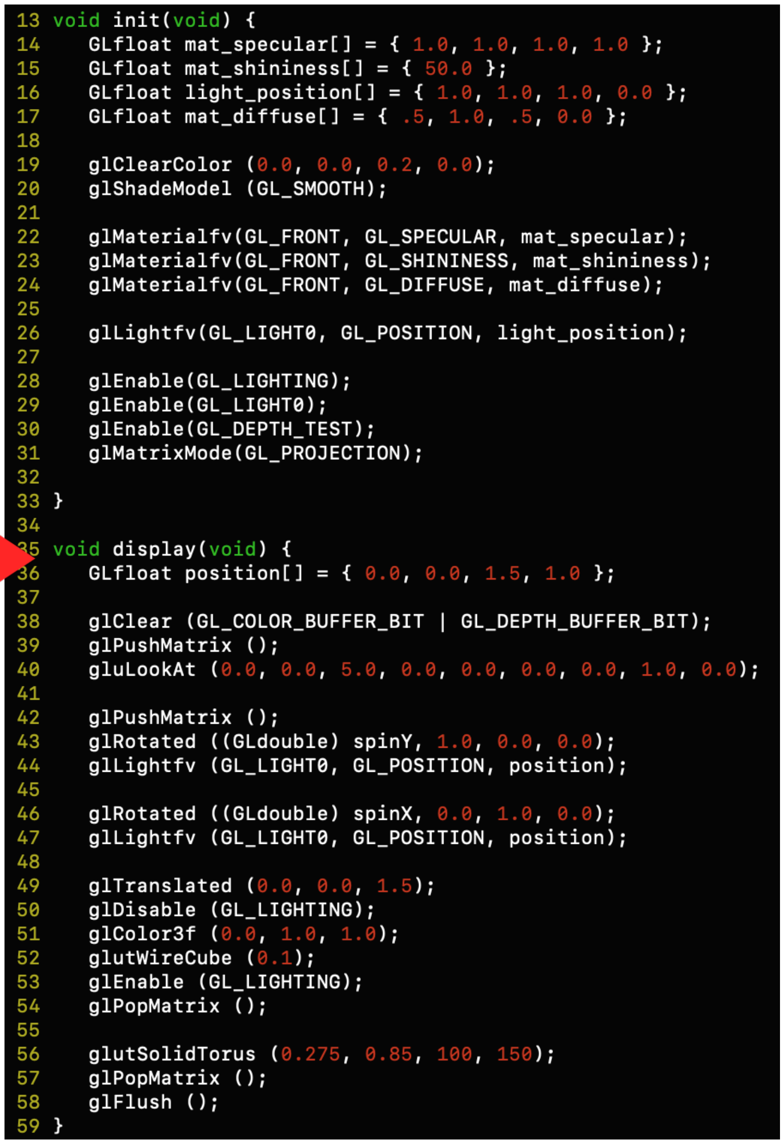
The remaining lighting is very standard enabling lighting functions, and **GL_DEPTH_TEST** on **line 14** ensures that objects that are closer to the camera appear in front of objects that are farther away. The depth test compares the depth values of pixels in the framebuffer to determine visibility. *In more clear English terms,* the **init** function initializes several OpenGL settings for rendering a 3D scene. It sets the clear color to black, configures smooth shading, enables lighting calculations, enables the first light source (GL_LIGHT0), and enables depth testing for correct depth ordering of objects. This function is typically called once during the initialization phase of an OpenGL program.

The **display** function on **line 17**, has more to do with setting up exactly what is happening in your scene, and it obviously has a lot more occurring inside of the function. Essentially, the display function sets up a simple 3D scene with a rotating wireframe cube *(without lighting)* and a solid torus *(with lighting)*. The scene includes a light source positioned at (0.0, 0.0, 1.5), and the viewpoint is set to (0.0, 0.0, 5.0). The variable **spin** controls the rotation of the coordinate system about the x-axis.

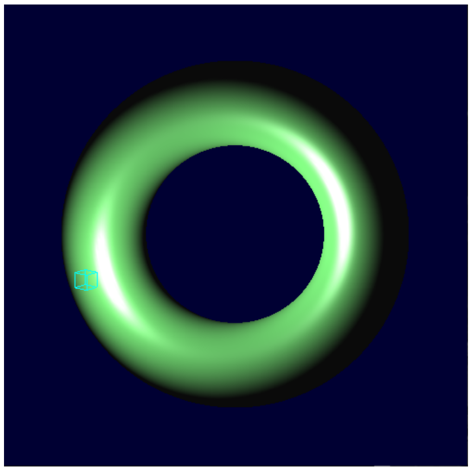
In my updated version, there is obviously some changes I made to this. In this **init** function I define more material properties such as specular reflection *(mat_specular)*, shininess exponent *(mat_shininess)*, light position *(light_position)*, and diffuse reflection *(mat_diffuse)*. Starting on **line 22** I set the material

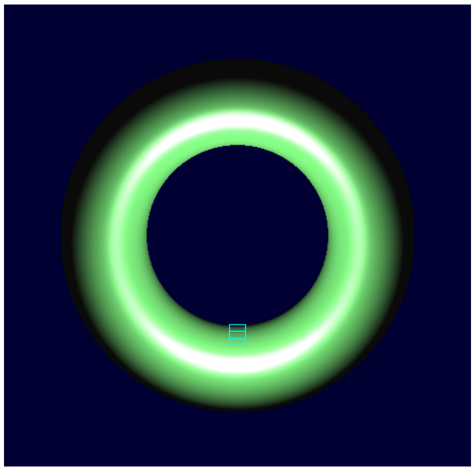


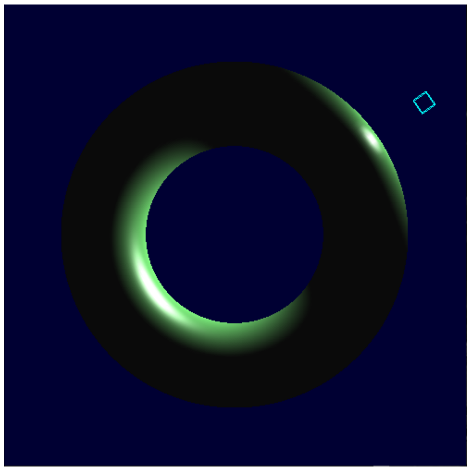
properties for the front faces of polygons, specular reflection, shininess, and diffuse reflection are specified using the previously defined arrays. My display function here also has more than the display function on the previous page. In this display function, the big difference is the radius and samples of the torus, the color, material attributes, lighting, and the ability to rotate on the y-axis in addition to the x-axis. These alterations give us the following effects:



```
13 void init(void) {
14    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
15    GLfloat mat_shininess[] = { 50.0 };
16    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
17    GLfloat mat_diffuse[] = { .5, 1.0, .5, 0.0 };
18
19    glClearColor (0.0, 0.0, 0.2, 0.0);
20    glShadeModel (GL_SMOOTH);
21
22    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
23    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
24    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
25
26    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
27
28    glEnable(GL_LIGHTING);
29    glEnable(GL_LIGHT0);
30    glEnable(GL_DEPTH_TEST);
31    glMatrixMode(GL_PROJECTION);
32
33 }
34
35 void display(void) {
36    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
37
38    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
39    glPushMatrix ();
40    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
41
42    glPushMatrix ();
43    glRotated ((GLdouble) spinY, 1.0, 0.0, 0.0);
44    glLightfv (GL_LIGHT0, GL_POSITION, position);
45
46    glRotated ((GLdouble) spinX, 0.0, 1.0, 0.0);
47    glLightfv (GL_LIGHT0, GL_POSITION, position);
48
49    glTranslated (0.0, 0.0, 1.5);
50    glDisable (GL_LIGHTING);
51    glColor3f (0.0, 1.0, 1.0);
52    glutWireCube (0.1);
53    glEnable (GL_LIGHTING);
54    glPopMatrix ();
55
56    glutSolidTorus (0.275, 0.85, 100, 150);
57    glPopMatrix ();
58    glFlush ();
59 }
```







What a clear difference exploring other lighting and materials can give you, but this is only scraping the surface of what can be accomplished with OpenGL lighting effects.

Advanced Lighting Techniques

Phong-Blinn shading is a shading model used to simulate the interaction of light with surfaces with more efficient calculations for specular highlights. The Phong-Blinn shading model provides more visually appealing methods to simulate the reflection of light on surfaces, producing realistic highlights and shading effects. To help us better understand the Phong-Blinn model, it can be expressed with the following equation:

$$I = k_a * I_a + k_d * I_d * (N * L) + k_s * I_s * (H * V)^n$$

- I is the final intensity of light reaching the viewer
- k_a, k_d, and k_s are the ambient, diffuse, and specular reflection coefficients
- I_a, I_d, and I_s are the intensities of ambient, diffuse, and specular light
- N is the normalized surface normal
- L is the normalized direction from the surface to the light source
- H is the normalized half-angle vector between L and V (the normalized direction from the surface to the viewer)
- n is the specular exponent

*<u>This calculation translates to **GLSL** as:</u>*

```
vec3 lightDir = normalize(lightPos - FragPos);
vec3 viewDir = normalize(viewPos - FragPos);
vec3 halfwayDir = normalize(lightDir + viewDir);
```

<u>This converts the Blinn-Phong specular reflection:</u>

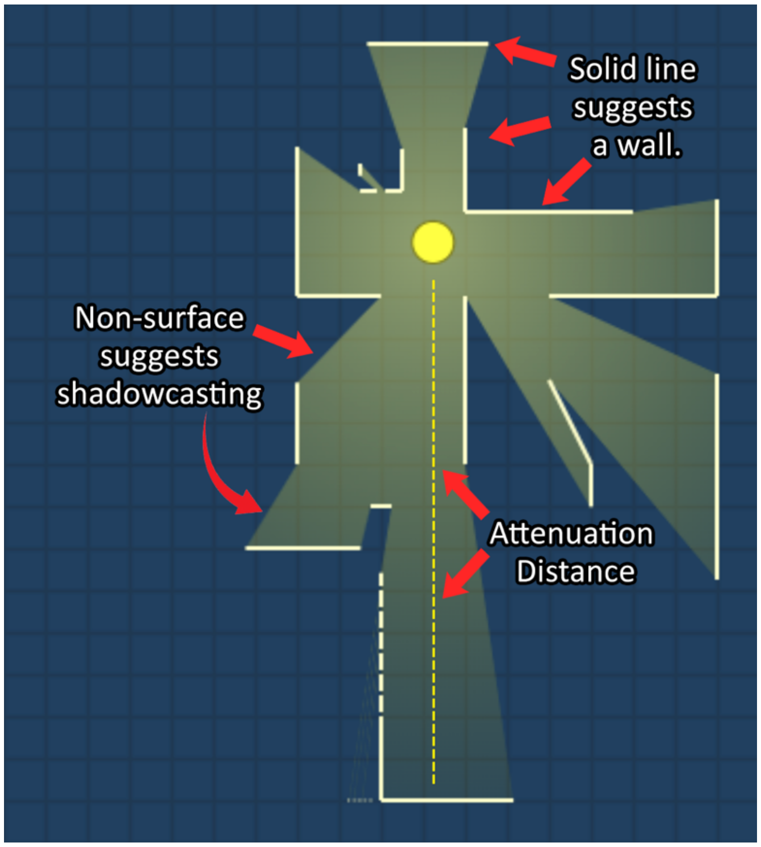
```
float specularIntensity = pow(max(dot(N, H), 0.0), shininess);
vec3 specular = specularColor * specularIntensity;
```

<u>and finally, Ambient reflection:</u>

```
vec3 ambient = ambientColor;
```

So far, we've learned how well point lights serve as a great example for light attenuation, and per my example on page 3, we understand this to be especially true from a two-dimensional perspective. This now brings us to the topic of **symmetric shadowcasting**. Shadowcasting is a common technique for calculating field of view. With the right implementation, it is fast, consistent, and symmetric. Before we get to the implementation, let's first contemplate how we can go about creating some of these effects.

The following example has a light source placed in a 2D environment surrounded by different walls that create a blockade which results in shadowcasting. When the light passes by a blocked surface, we are essentially in a blind spot. The light attenuation distance displays how far the light can hit and reflect off of objects. In a 2D top-down map it is sometimes useful to calculate which areas are visible from a given point. For example, you might want to hide what's not visible from the player's location, or you might want to know what areas would be lit by a torch. We can also add a lot more lights to illuminate more of the map to better illustrate environment lighting.


Solid line suggests a wall.
Non-surface suggests shadowcasting
Attenuation Distance

As we can see, light casting on a two-dimensional plane involves simulating the effects of light sources and their interaction with objects on a flat surface. This is commonly used in 2D graphics or games where a 3D representation is not required. In my previous guide entitled, ***Interactive Computer Graphics*** I provided a brief explanation about how to write a vertex shader and a fragment shader in GLSL. In 2D light casting environments such as this, we can use them again. Let's consider a simple scenario with a point light source and objects on a 2D plane. Here's a simplified explanation of how light casting can be implemented in a 2D plane using a **vertex shader** in GLSL:

```
layout(location = 0) in vec2 inVertexPosition;
out vec2 fragPosition;

void main() {
   fragPosition = inVertexPosition;
```

Passing the vertex position to the fragment shader

```
    gl_Position = vec4(inVertexPosition, 0.0, 1.0);
}
```

Output transformed vertex position

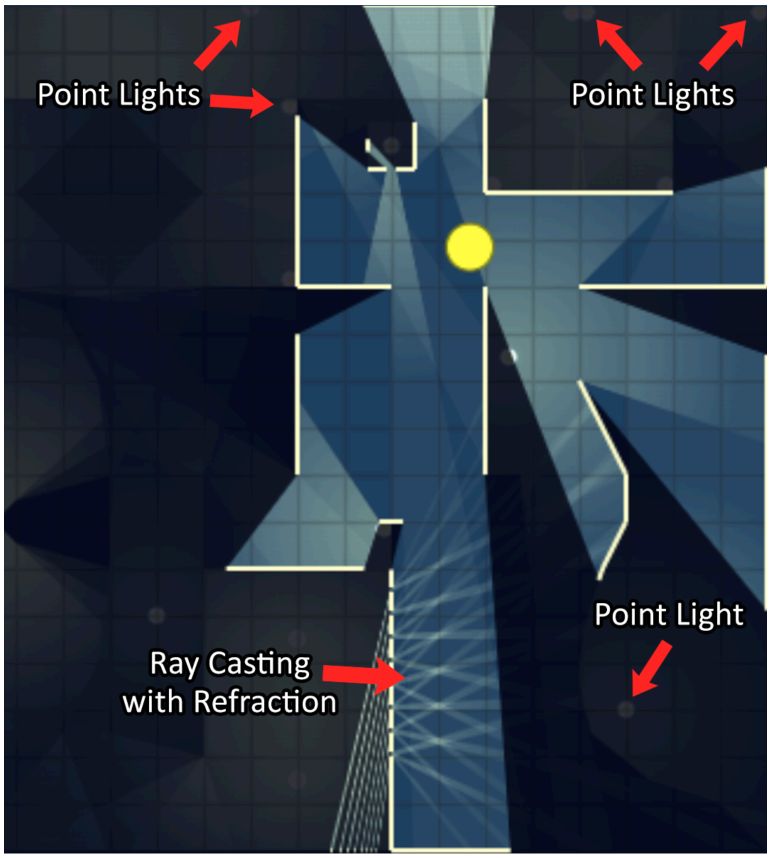
and the **fragment shader:**

```
in vec2 fragPosition;
out vec4 fragColor;

uniform vec2 lightPosition;
uniform vec3 lightColor;
uniform float lightRadius;

void main() {
    float distanceToLight = distance(fragPosition, lightPosition);

    float attenuation = 1.0 / (1.0 + 0.1 * distanceToLight + 0.01 * distanceToLight * distanceToLight);

    vec3 intensity = lightColor * attenuation;
    fragColor = vec4(intensity, 1.0);
}
```

Calculate the distance from the fragment to the light

Calculate the attenuation factor based on distance

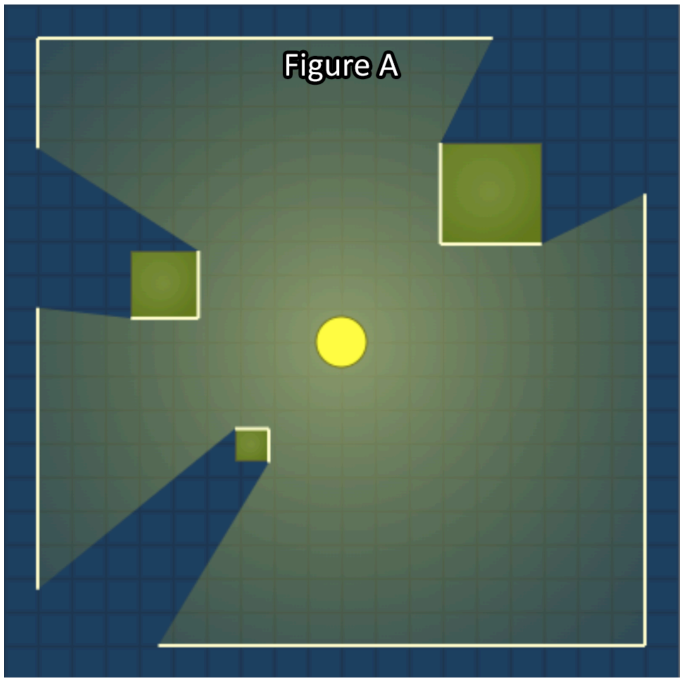
Calculate the intensity of the light and outputting the final color.



In your own application, you would set the appropriate uniforms *(lightPosition, lightColor, and lightRadius)* based on the position, color, and radius of your light source. You would then render your objects using these basic shaders.

The above vertex and fragment shaders are examples of how this light casting application is implemented, which also reproduces shadowcasting effects. I just had to alternate the uniform parameters, and account for several other functions. Another area distinctive to the following effects in these examples are known as subtractive algorithms. Subtractive algorithms start with everything visible then subtract the hidden areas; additive algorithms start with nothing visible then add the visible areas. I'll describe an additive algorithm that works with line segments, not only solid blocks or grids.

This example shows how we can manipulate shadowcasting effects using different size blocks and the silhouettes created by said objects.



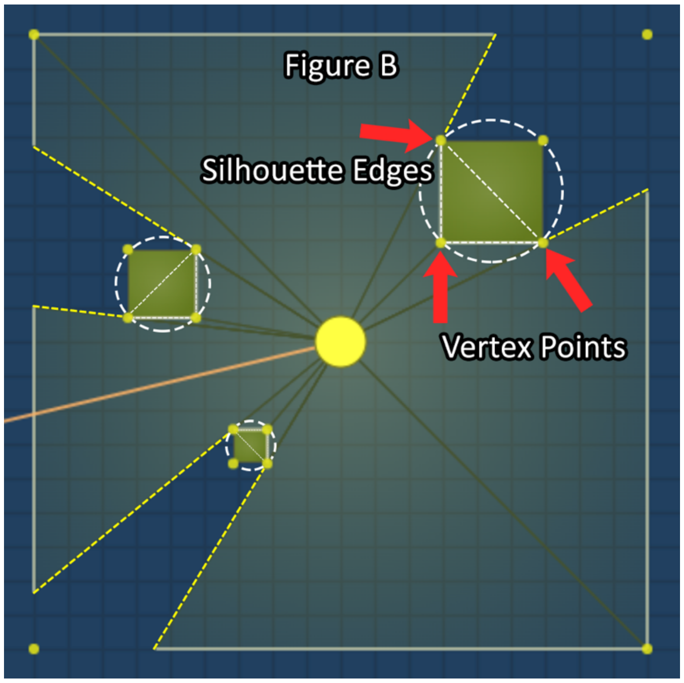
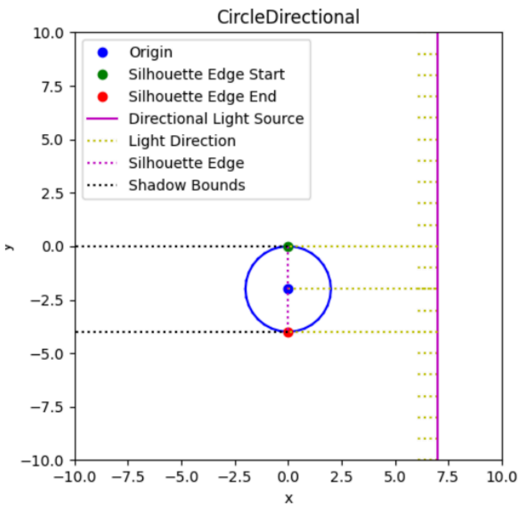
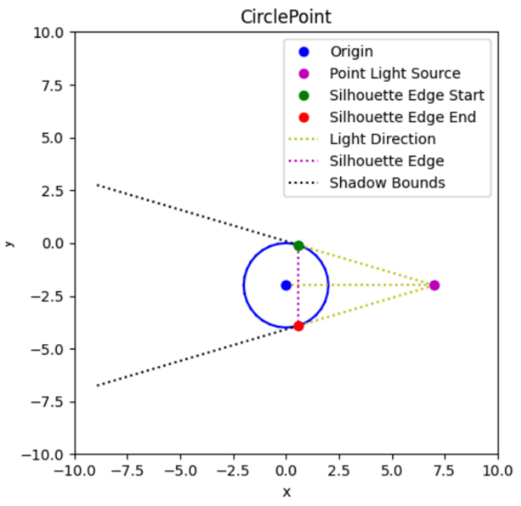


Figure A simply displays a 2D sandbox demo with three different sized blocks that I added into a point light scene. **Figure B** demonstrates visual depictions of angularity with shadowcasting and ray casting passes. The next question is, how do we keep track of which walls the sweep ray passes through? Only the nearest wall is visible. How do you figure out which wall is nearest? The simplest thing is to calculate the distance from the center to the wall. Whenever the nearest wall ends, or if a new wall is nearer than the others, we create a triangle showing a visible region. The union of these triangles is the area that is visible from the central point. <u>**Note:**</u> Creating a triangle involves intersecting the previously active wall with the sweep ray. As a result, the new edge of the triangle may be longer or shorter than the sweep ray, and the far edge of the triangle may be shorter than the previously active wall.

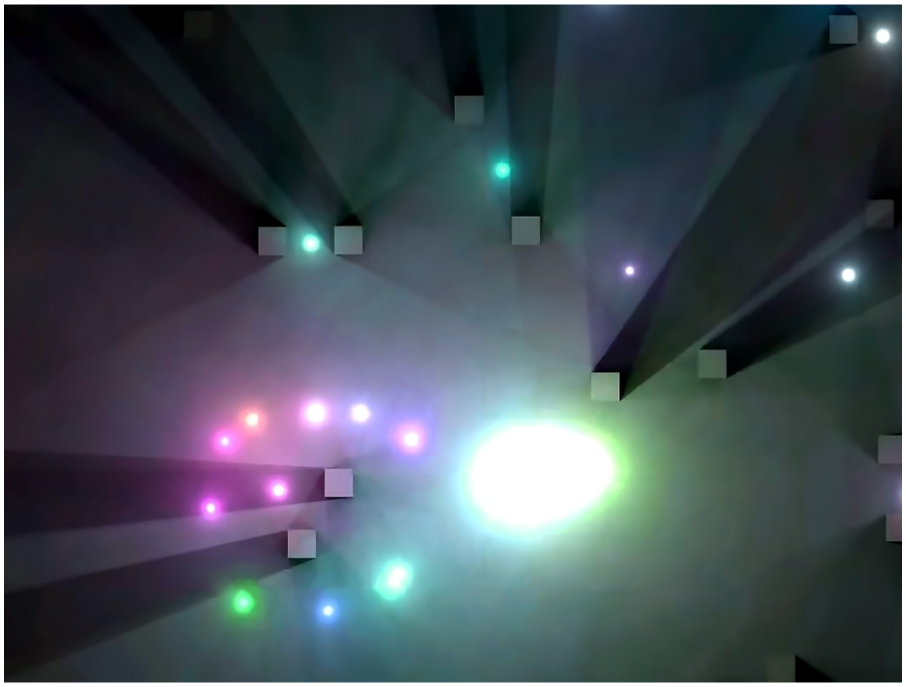
The next question is, how can we better understand the angularity and silhouette edges of passing light by objects on a 2D plane? Let's consider the circumference of a circle. If we look at the circle mathematically it's very easy to represent it in 2D space, all we need is a center point and a radius to its border. First, we find the direction perpendicular to the rays of light. If we then travel in that direction (and the inverted) from the center point by the radius of the circle we'll find the points that make up the



silhouette edge. The angle of the rays will determine the shape of the shadow cast behind the circle. What's even better is that we can use trigonometry to find all points along its circumference as long as we know these two properties. It's this principle that is the starting point of the trigonometric algorithm. As long as we ensure that the shapes all have a certain level of symmetry to its bounding circle, then we can use trigonometry as a starting point and then adjust the results based on how close our shape is to a circle. With this information we can define the span of our algorithm. At one end the silhouette edge is close to running through the center point, but never quite reaching it. On the other end the light is close enough for a single primitive to represent the silhouette edge.



If we amplify this process, we can create even more vivid effects using light casting in OpenGL & GLSL with just one click of the mouse. This is another example of light casting on a two-dimensional plane in GLSL except it uses a much more advanced approach with pixel shaders.



The last question is, why create these programs? What can we learn from these processes?

Player Vision – The simplest operation is to limit the player's vision by intersecting the output with the limit of visibility. For example, intersect the output of the algorithm with a circle to limit the radius you can see. Intersect with a gradient-filled circle to make the light fall off with distance. Intersect with a cone to build a "flashlight" effect that lets you see farther in front of you but not much behind you.

Mapping Objects – Visibility can also be used for calculating what areas a torch lights up. My above example takes the union of the areas lit up by each torch, then intersects that with the area the player can see.

AI Behaviors – Visibility can also be used to build AI behaviors. For example, let's suppose the enemy AI wants to throw a grenade to hit one of the players, but also wants to stand in a place where the players can't shoot back. Grenades need to be close enough to hit the player, and also not behind an obstruction.

CAD (Computer Aided Design) – In CAD applications, accurate lighting simulations help designers visualize how objects will look in different lighting conditions. This is crucial for architects, engineers, and product designers to assess the aesthetics and functionality of their designs.

Architectural Visualization – Architects use light casting to create realistic renderings of buildings and interiors. It allows them to showcase the impact of natural and artificial lighting on the spaces they design.

Virtual Reality (VR) and Augmented Reality (AR) – Realistic lighting is essential in VR and AR environments to create a sense of presence. Simulating accurate shadows and reflections enhances the perception of depth and realism for users.

Conclusion

This concludes my guide on light casting in OpenGL and GLSL. While this guide touched up on some fairly in-depth topics with light, my goal was to make this significantly shorter than my previous OpenGL guide, and focus more on a single, interconnected topic rather than branching out into numerous areas of graphics programming. There is obviously a vast amount of further research and implementation to create light casting effects, but this guide was not intended to be that long. OpenGL can easily be expanded both mathematically, and programming-wise that this guide could have grown to 80+ pages in no time. When I look back to my journey with OpenGL and GLSL programming, I can safely say it did not happen over a fortnight. Computer graphics programming is next-level and not easy to produce, but is a fascinating technology if you're willing to work hard to learn it. I hope this guide was helpful in teaching you more about light casting with computer graphics. Unless otherwise specified, all diagrams, code examples, and application-rendered images were created by me. If you have any questions about this guide or any other general inquiries, you can email me at cjordan@wondercreationstudios.com

Resources Used:

- Engel, Wolfgang. *Programming Vertex and Pixel Shaders* – 2004
- Lengyel, Eric. *Mathematics for 3D Game Programming and Computer Graphics (Second Edition)* – 2003