Interactive Computer Graphics Using OpenGL Programming with C++

By Chad Jordan – December 6th 2008

In this guide you will learn:

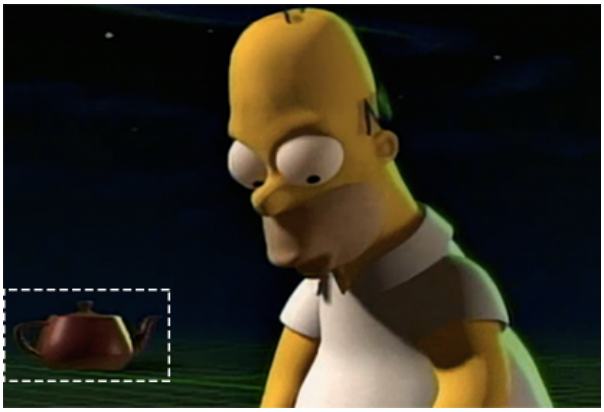
1. The origins of interactive computer graphics in C++ using the OpenGL API
2. Understanding the process of the computer graphics pipeline and 3D theory
3. Drawing and animating primitive shapes in a 3D matrix
4. Creating GUI menu systems for mouse interaction with 2D and 3D graphics
5. Tree traversal concepts and understanding quaternions using matrix algebra
6. Writing vertex & fragment shaders in GLSL to run on a programmable GPU

Introduction

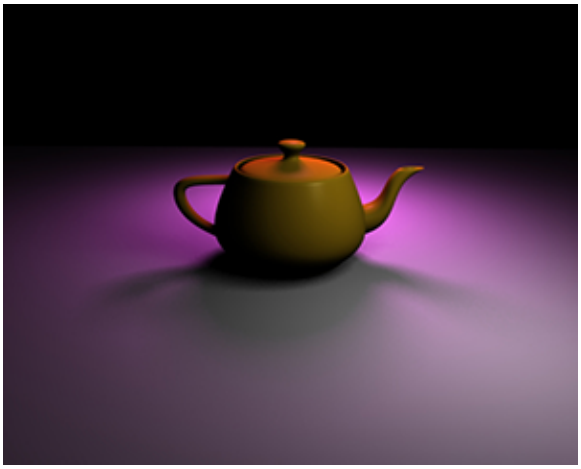
When I say, "the origins of interactive computer graphics" I'm not talking about the moment Evan Sutherland presented his Ph.D. thesis on computer graphics in 1963. I'm referring to creating computer graphics out of nothing, having only a code editor, and enthusiasm for exploring the world of computer programming. For my code examples, I will be using a code editor in Linux called Vim. Any readers of this guide should know that the subject matter herein is a more advanced area of programming than my previous guides. This means you'll need to be somewhat familiar (and comfortable) with topics that cover the principles of polygonal tessellation, vector & scalar mathematics, splitting matrices, and programming shaders.

The definition of computer graphics is a mathematical representation of creating and manipulating 2D and 3D images with the aid of computer programming. The term, computer graphics has most commonly been associated with scientific visualizations in computer science since it originated in that field of study.

For those that are not aware, the Utah Teapot on my cover page has existed a lot longer than the days of Cinema 4D, 3d Studio Max, and Maya. It also goes much further back than the Stanford Bunny. The Utah Teapot was created in the mid-1970's by computer graphics researcher Martin Newell, a member of the pioneering graphics program at the University of Utah. During my semester of taking Computer Graphics, one of the things I had to learn how to program was bringing the Utah Teapot into a 3D scene using Bezier patches. The infamous Utah Teapot has remained as one of the most iconic memorable images created in computer graphics to this day. The teapot has made its appearance in the 1995 Pixar film Toy Story, and The Simpsons Treehouse of Horror VI episode: Homer^3. Many people do not realize just how iconic this little teapot has been in the history of computer graphics.



As of more recently, I tried my own hand at modeling and rendering the Utah Teapot in Blender. The image on the left used Blender's internal render engine, and the image on the right used the Yafaray render engine from a kitchen scene I modeled.

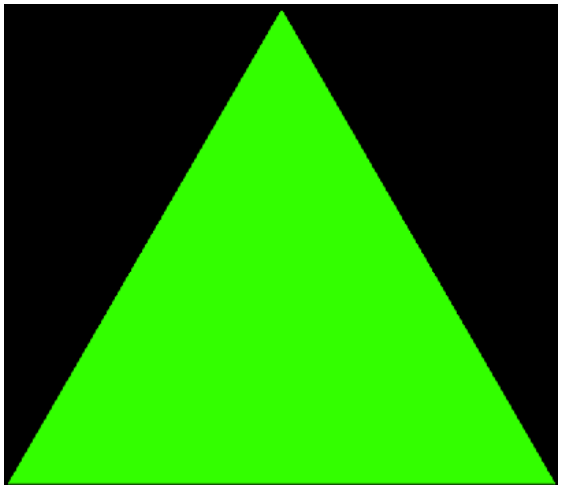


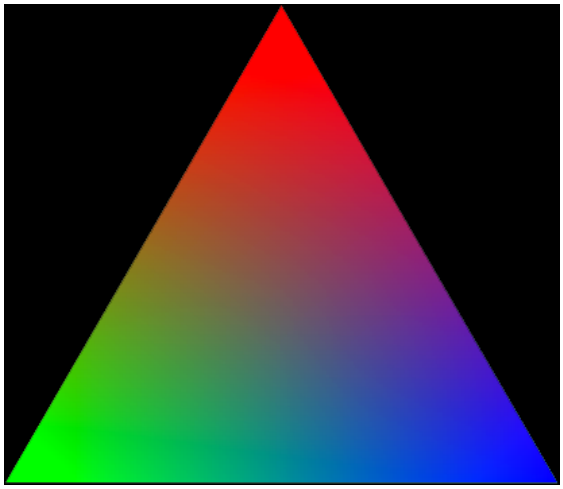
When people have asked me, "Do you have to be good at math to be a coder?" I always tell them it definitely helps, but it's far more important to think systematically with a logical mind. Being a good problem solver is important. Full disclosure, I'm not great at math. I've always considered myself very average with it, but I've been better at programming and I know how to tie in the math with the code using OpenGL. Why am I mentioning this? Because people that are interested in programming should know that math really starts making more of an appearance when creating computer graphics and games, not necessarily with basic programming on the command line. This is especially the case when it comes to OpenGL programming. OpenGL ***(Open Graphics Library)*** is a cross-language, cross-platform application programming interface ***(API)*** for rendering 2D and 3D vector graphics. While most of my programming experience with OpenGL has compiled through the CPU, the API is typically used to interact with the graphics processing unit *(GPU)*, to achieve hardware-accelerated rendering, which I will be demonstrating later on.

Computer graphics on a holistic scale, especially when created from scratch using only programming is another step up on the comprehensive platform. Overall, there is a lot more involved on the theoretical side, the mathematical side, and the coding side of OpenGL programming. In this guide, I'll be covering concepts revolving around 2D graphic primitives, scene graphs, 3D interactive programming, splitting matrices using matrix multiplication, and basic principles of geometry and trigonometry applied to OpenGL. Therefore, the terminologies and references that I will be using in this guide are more directed at programmers that have already had a couple of years of programming experience beyond the basics and studied the essentials of computer graphics. My goal in this guide is to break down some of the programs that I've written using this API with C++ to provide a good learning experience and a glimpse into what it's like to delve into the core of computer graphics.

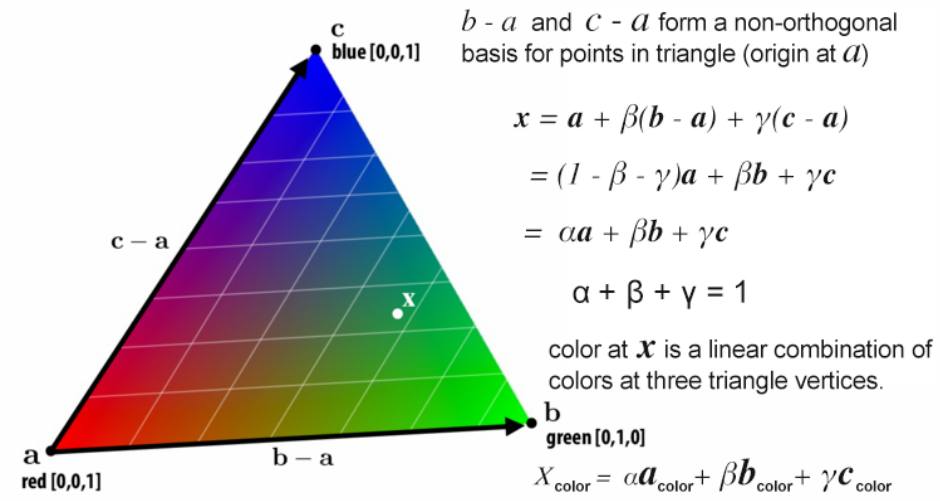
Your First VPL Program

"Hello Wor….Triangle?" Yes, this is the famous "Hello Triangle" program in OpenGL. Just as the "Hello World!" program exists with statically-typed compiling systems, the "Hello Triangle!" program exists in Visual Programming Languages (VPL). When it comes to OpenGL, everything occurs in 3D space, but the screen or *window* is a 2D array of pixels so a large part of the graphical results in OpenGL has to do with transforming all 3D coordinates to 2D pixels that fit on your screen. In OpenGL the process of converting 3D coordinates to 2D pixels is managed by the graphics pipeline. The graphics pipeline *(which I will be elaborating more on shortly)* can be divided into several steps where each step requires the output of the previous step as its input. We can pass sRGB values to the vertices which gives us the result of color interpolation.





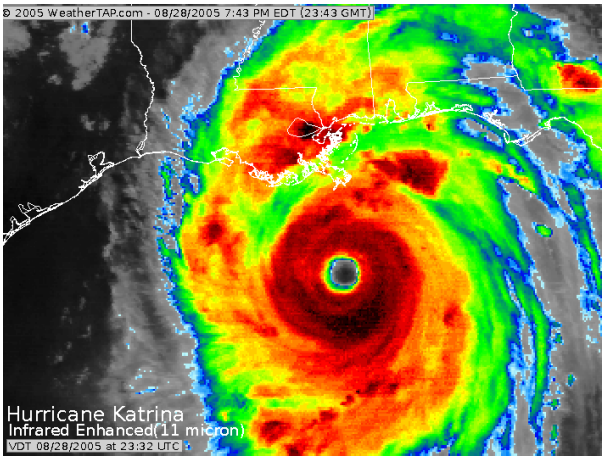
What do I mean when I say color interpolation? Color interpolation chooses between color operations occurring in the sRGB color space or in a *(light energy linear)* linearized RGB color space. Having chosen the appropriate color space, component-wise linear interpolation is used. This triangle is an example of Barycentric Interpolation:

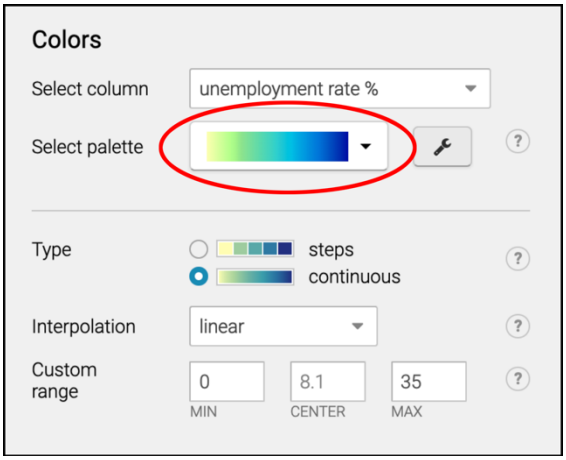


Breaking a color down into components significantly assists with interpolation, as we now have only three numbers that we need to move in-between values. This means three linear interpolations, one for each component, following a well-known formula. Given two colors **A**

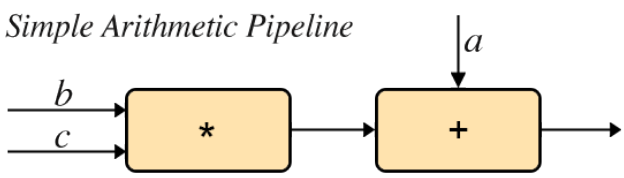
and **B**, each with components r, g, and b, we can derive the in-between color **C** at time γ *(γ is normalized and moves between 0 and 1)*. What this means is that the triangle's edges **b - a** and **c - a** form a non-orthonormal basis for points in the triangle. The offset of any point in the triangle from vertex *(point x)* from vertex a is given by a linear combination of these vectors.

So now that you know how it works, you may be asking yourself, what exactly can we use color interpolation for? Color interpolation is used all the time in software to represent data whether for scientific graphs with matlab, or graphic design purposes using color palettes in Photoshop. Larger companies may be wanting a color scheme to help decide on a theme for their website. Interpolating colors can help you create your own custom color palettes since interpolating data can allow you to see a vast array of colors between other colors that you may or may not consider using. Interpolating data for different colors can also determine levels of severity with doppler radar, or how you represent data for the unemployment rate in the US.

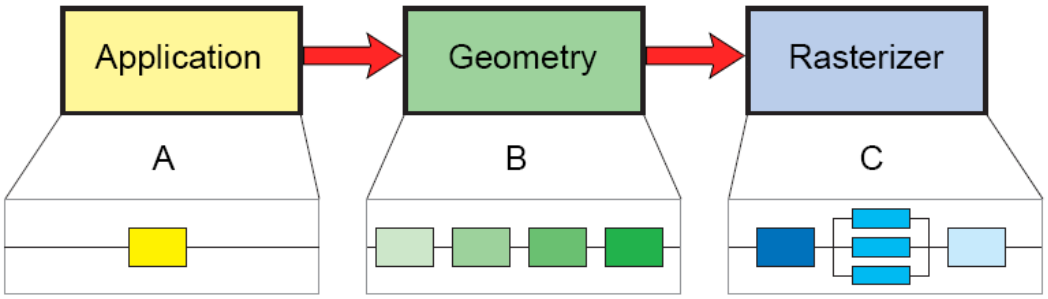




So how do we consider all of these processes from a conceptual standpoint? At this stage of the guide we take a look at the modeling-rendering paradigm of the computer graphics pipeline. In many situations, especially in CAD applications and the development of complex images, such as for movies we can separate the modeling of the scene from the production of the image. On one side of the application we have the API, and the other side is a combination of hardware and software that implements the functionality of the API. Pipelining is similar to the assembly line of a car plant. As the chassis passes down the line, a series of operations is performed on it, each using specialized tools and workers until the assembly process is complete. At any one-time multiple cars are under construction and there is a significant delay or latency between when a chassis starts down the assembly line and the finished vehicle is complete. Here is a basic representation of the arithmetic pipeline which has an adder and a multiplier that computes the same basic operations for a single processor $a + (b * c)$.



Suppose you need to carry out the same computation with many values of a, b, and c. The multiplier can pass on the results of its calculation to the adder and start its next multiplication while the adder carries out the second step of the calculation on the first set of data. Whereas it requires the same amount of time to calculate the results for any one set of data, when working on two sets of data at a time, our total time for calculation is significantly shortened. This basic pipeline represents the increased amounts of data blocks added to a normal pipeline.

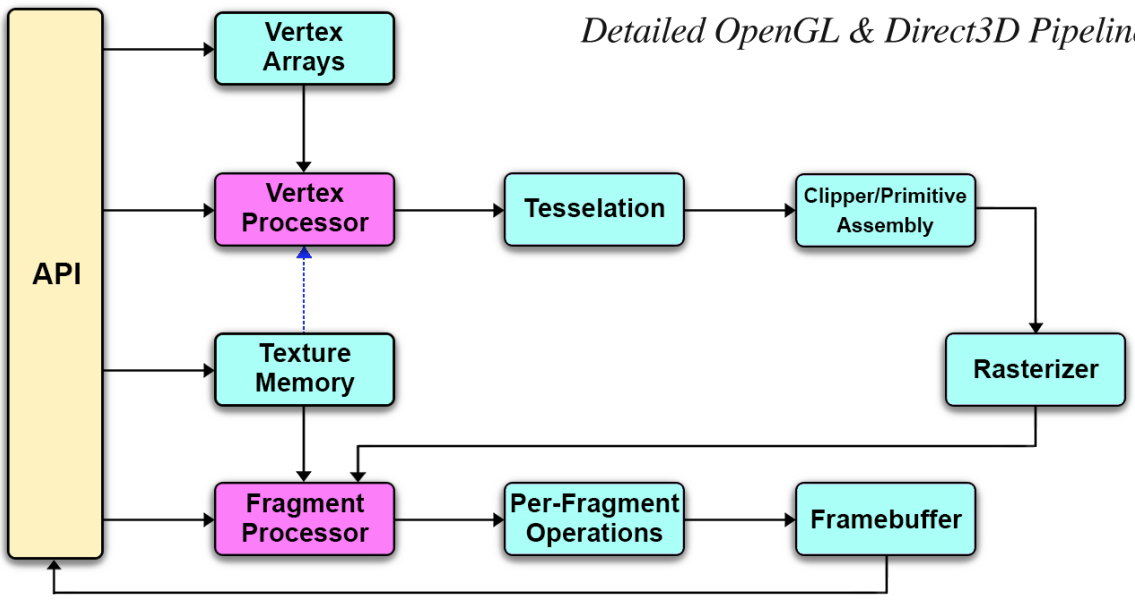


Computer Graphics Pipeline

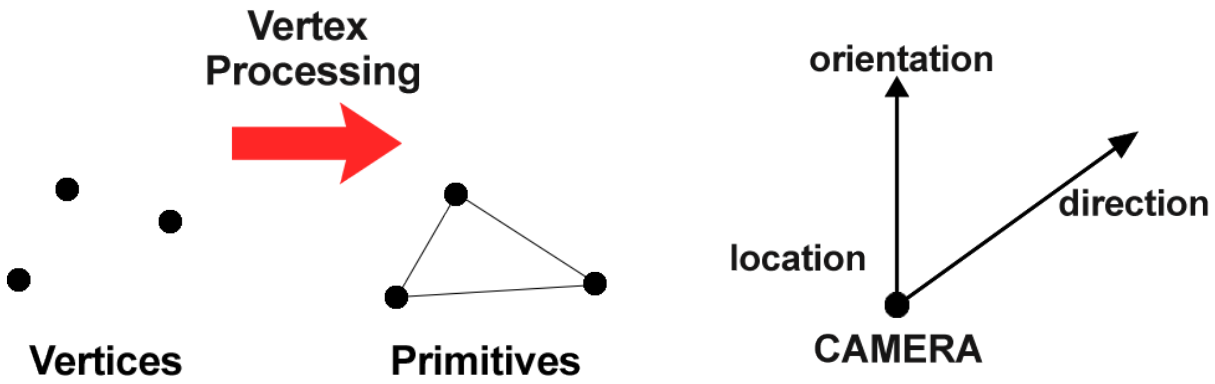
However, the more data blocks that we add to our pipeline, this means more processing time occurs which results in more latency in the system and its important to balance latency against increased throughput when evaluating the performance of the pipeline. There are four essential parts to the graphics pipeline:

1) **Vertex processing**
2) **Clipping and primitive assembly**
3) **Rasterization**
4) **Fragment processing**

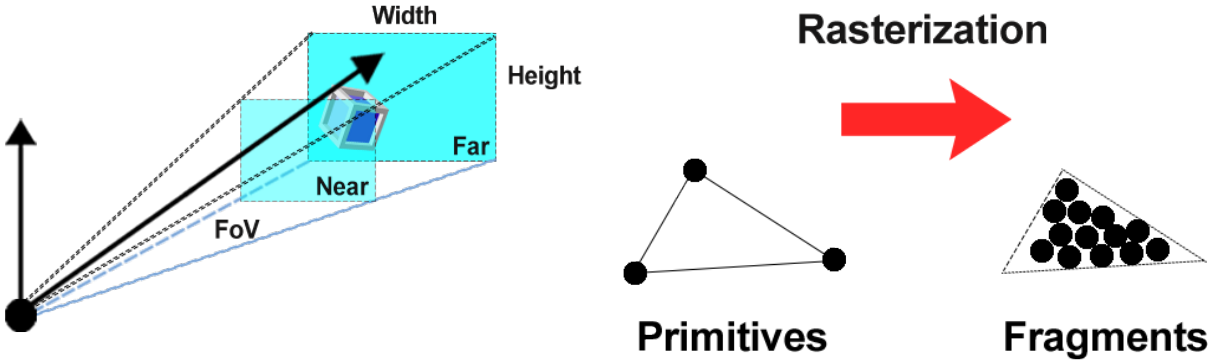
While the general process of the graphics pipeline is fairly static, not every pipeline represents the same data blocks, and of course, there are more details that can be included depending on what is included. This is a more detailed look at the OpenGL pipeline as well as Direct3D.



Vertex processing is typically one of the first blocks of the pipeline, and each vertex is processed independently. The two major functions of this block are to carry out coordinate transformations and to compute a color for each vertex. Within this data block there are four additional stages of processing to understand. The first one involves arranging the objects in 3D space, and this is referred to as model transformation. The next is the view transformation which sets the positioning and orientation of the camera in 3D space. The camera has three parameters: *location, direction, and orientation*. These three parameters must be defined and written when first creating your scene.

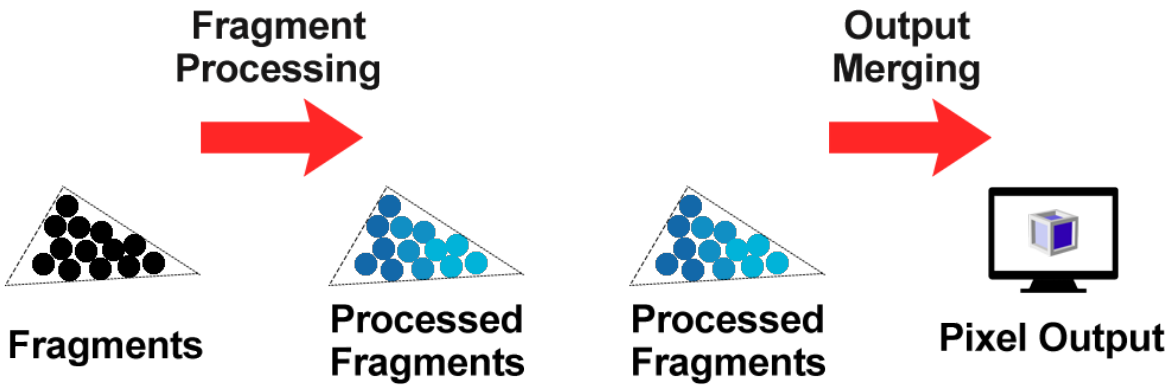


The third stage of vertex processing is projection transformation *(also referred to as perspective transformation)* and this is used to define our camera settings. Essentially it sets up what can be seen by the camera such as field of view, aspect ratio, and optional near and far planes. This is part of the **clipping & assembly** process in the imaging system. The fourth and final stage of vertex processing is viewport transformation, which outputs everything for the next step in the graphics pipeline, rasterization.

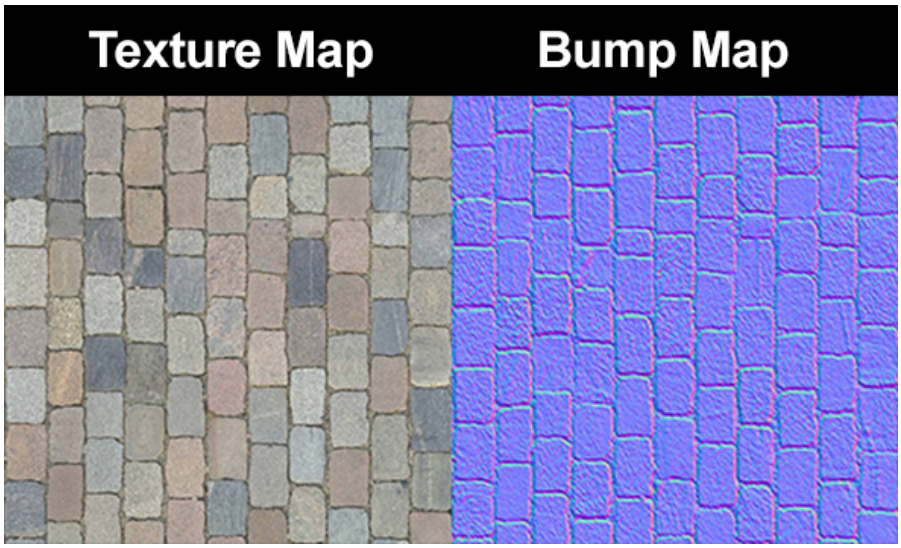


At this stage the primitives that emerge from the clipper are still represented in terms of their vertices and must be further processed to generate pixels in the frame buffer. An example of this, if three vertices specify a triangle filled with a solid color, the rasterizer must determine which pixels in the frame buffer are inside the polygon. The output of **rasterization** is a set of fragments for each primitive. The fragment can be thought of as a potential pixel that carries information with it, including its color and location. This is used to update the corresponding pixel in the frame buffer. Fragments can also carry along depth information that allows later stages to determine if a particular fragment lies behind other previously rasterized fragments

for a given pixel. **Fragment processing** takes in the fragments generated by the rasterizer and updates the pixels in the frame buffer. Put another way, it calculates the final colors textures based on the given parameters from the user.



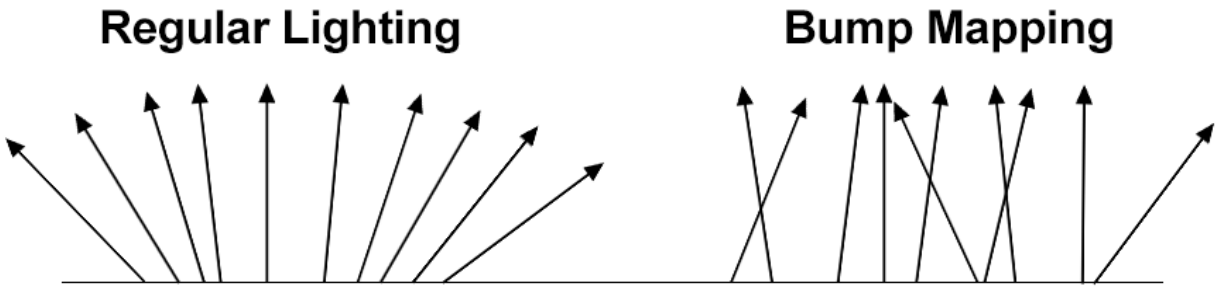
The color of a fragment may be altered by texture mapping or bump mapping as shown here:



The color of the pixel that corresponds to the fragment can also be read from the frame buffer and blended with the fragment's color to create translucent effects. Regardless if you're building interactive 3D scenes with programming, or building a 3D scene with render software like Maya, these scenes are filled with meshes each consisting of hundreds and possibly thousands of triangles. You can boost the realism by wrapping (UV/Normal Mapping) 2D textures on flat surfaces, hiding the fact that the polygons are just tiny flat triangles. While flat textures are helpful with realism, when observed much closer the mesh is easy to see the underlying flat surfaces. The vast majority of real-life surfaces aren't flat, but clearly show plenty of bumpy details.

The idea behind normal mapping is that instead of interpolating the vertex normals across the triangle face *(which creates the smoothness we are trying to get rid of)* they can simply be

sampled from a texture. This represents the real world better because most surfaces *(especially the ones we are interested in for gaming)* are not that smooth such that light will be reflected back in accordance with the way we interpolate the normals. Instead, the bumps on the surface will make it reflect back at different directions, according to the general direction of the surface at the specific location where the light hits. For each texture these normals can be calculated and stored in a special texture which is called a normal map. During lighting calculations in the fragment shader the specific normal for each pixel is sampled and used as usual. The following images show the difference between the normals in regular lighting and bump mapping:



Given everything that I've explained regarding normal maps, the following mathematical formula represents how UV mapping is applied and projected on a tangent plane in OpenGL.

Let P(u, v) be a parametric surface

- **Perturb the normal N to N'**
- **N' is the normal of offset surface P'**
- **The displacement must lie on the tangent plane of P**
- **Two arrays that contain partials of B(u, v) can be precomputed**

$$N(u,v) = P_u \times P_v \frac{P_u \times P_v}{|P_u \times P_v|}$$

$$P'(u,v) = P(u,v) + B(u,v)N(u,v)/|N|$$

$$N'(u,v) = P'_u(u,v) \times P'_v(u,v)$$

where

$$P'_u = P_u + B_u N/|N| + BN_u/|N|$$

$$P'_v = P_v + B_v N/|N| + BN_v/|N|$$

$$N' = N + [B_u(N \times P_v)/|N| + B_v(P_u \times N)]/|N|$$

$$= N + [B_u(N \times P_v)/|N| - B_v(N \times P_u)/|N|]$$

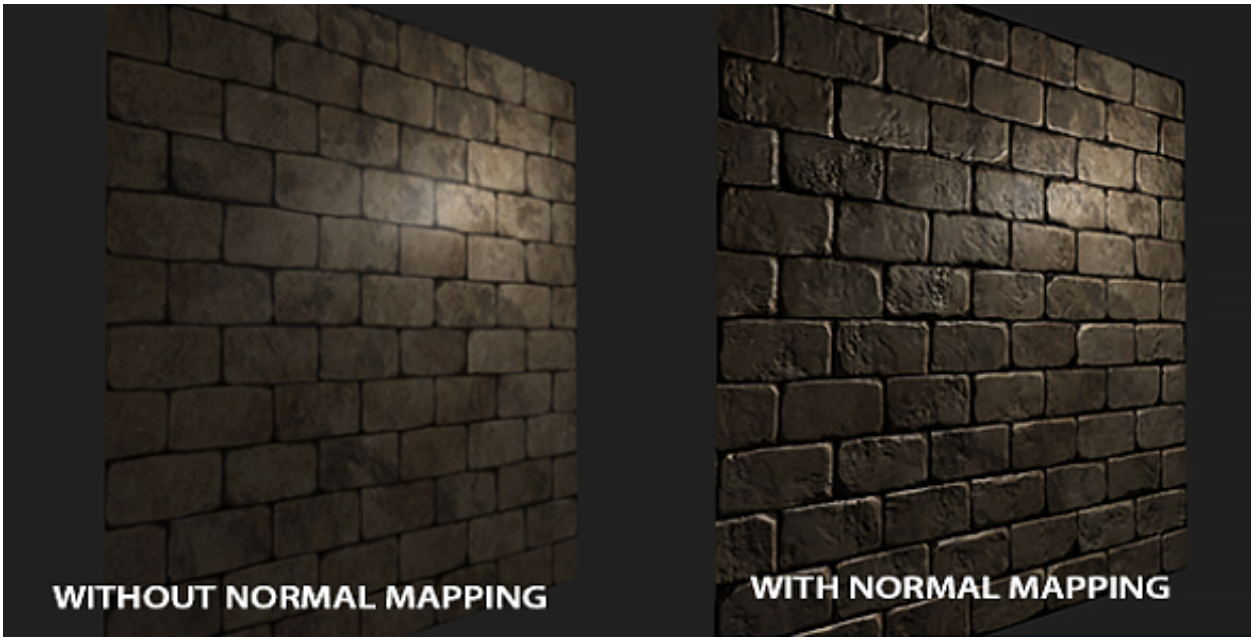
$$= N + [B_u A - B_v B] = N + D$$

A and B are on the tangent plane of $P(u,v)$

This formula is a great source when interpreted into OpenGL code for writing normal maps into your applications, not only for tangent surfaces but for object and material displacement as well. In many ways, normal maps are what bring your objects to life and allow for more vivid scenery. The only drawback to some of these more detailed formulas is trying to find the

delicate balance between detail and latency for processing power. Cutting back on subsurface division with lower polygonal objects and relying on better UV mapping techniques can significantly assist in swifter processing and response times. One way to do this is considering a simple plane used as a wall. A plane has a total of 4 vertices (one point at each corner). We can easily map textures to flattened objects and apply a normal map to it in order to give the illusion of detailed bricks. With this method we save thousands of system resources, polygons, and time invested into creating actual bricks along a flattened plane.

A visual example of this is a brick wall surface. What all do we know about a brick wall? We know that a brick surface is rough, and not always completely flat. Bricks are highly textured objects that contain small, sunken areas of cement with a lot of detailed holes and cracks. If you view a bricks surface in a well-lit scene it's pretty apparent what the texture looks like, but in a slightly more faded scene such as this one with just a single point light, we see just how much clearer and more detailed the texture appears when normal maps are applied.



This example is a great visual representation of how the above mathematical formula (properly implemented into the code) can create wonderful vivid results. Now that you understand some of the basic theories and methodologies behind graphics manipulation and the graphics pipeline, let's look at some of the syntax methods for defining and calling functions in OpenGL code.

No matter which programming language you're using with the OpenGL API, we're still defining and declaring our variables, constants, Booleans, and writing functions as we would with any other language. However, this time the way we're calling and implementing the functionality is what is changing. This time we're programming visual effects and images along with our functionality which means we have a lot more that we're having to implement than before. This also means new methods of learning and writing syntax.

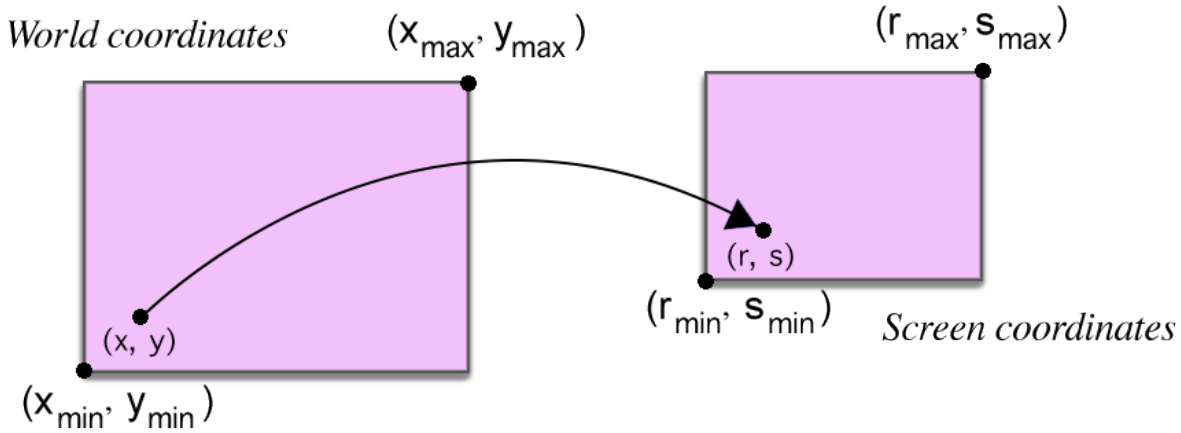
Data Types in OpenGL

An example of this is how we use data types. In traditional C types, we use `int` and `float` but in OpenGL we use `GLint` and `GLfloat`. These types are defined in the OpenGL header files and usually in an obvious way, for example: `#define GLfloat float`
Using OpenGL types allows additional flexibility for implementations where you may want to change floats to doubles without alerting existing application programs. Considering a vertex function, if the user wants to work in two dimensions with integers, then the syntax would be: `glVertex2i (GLint xi, GLint yi)` and three dimensions with floating-points would be: `glVertex3f (GLfloat x, GLfloat y, GLfloat z)` Finally, if we use an array to store the information for a three-dimensional vertex: `GLfloat vertex[3]` and then use: `glVertex3fv (vertex)`

Vertices can define a variety of geometric primitives different numbers of vertices are required depending on the primitive. We can group as many vertices as we want between the functions `glBegin` and `glEnd`. The argument of `glBegin` specifies the geometric type that we want our vertices to define. Hence, a line segment can be specified as follows:

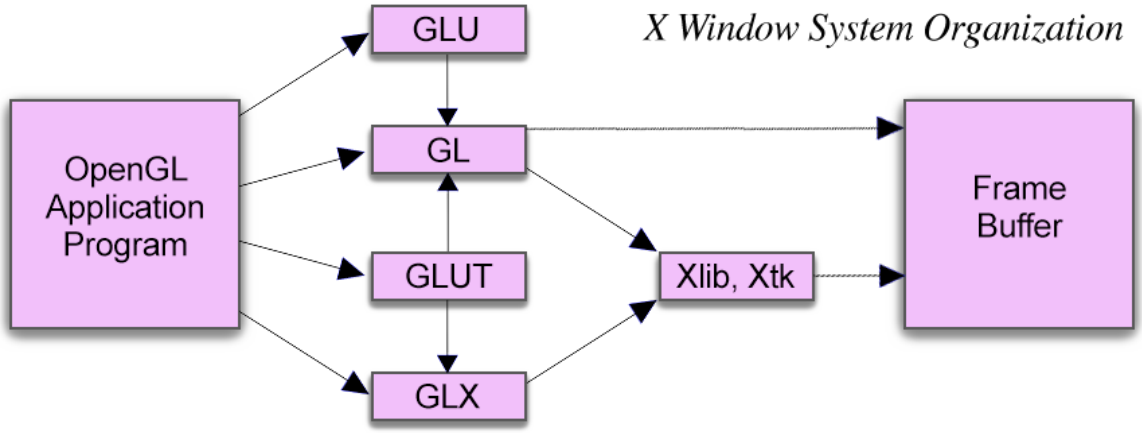
```
glBegin(GL_LINES);
   glVertex3f(x1,y1,z1);
   glVertex3f(x2,y2,z2);
glEnd();
```

We can use the exact same syntax to draw vertex points by copying and pasting the above code, and simply calling `GL_POINTS` in place of `GL_LINES`. It's entirely understandable if it seems puzzling how to interpret the values of x, y, and z with vertices. You may be asking yourself, *what units are they in? Are they measured in feet, meters, or microns? What about the point of origin?* The simplest response is that it's entirely up to you. One of the major advances in graphics software systems occurred when the graphics systems allowed the user to work in any coordinate system that they desired. The advent of **device-independent graphics** freed application programmers from worrying about the details of input and output devices.

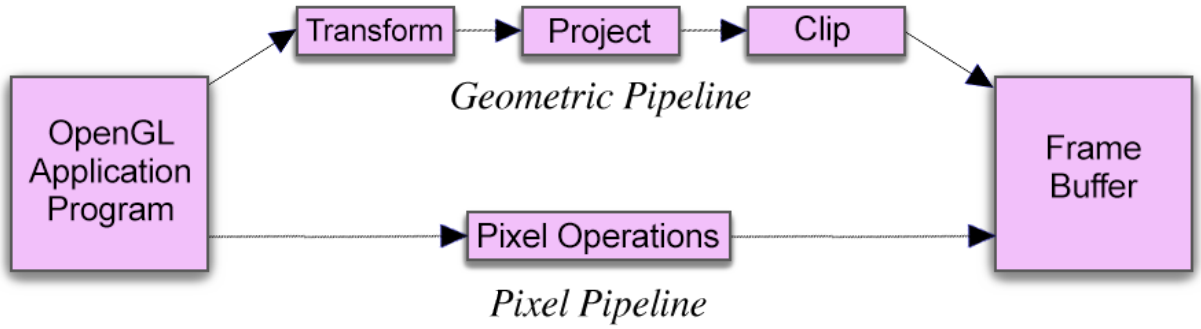


This user's coordinate system became known as the **world/object coordinate system**, or **application model**. Screen coordinates are expressed in integer types, because the center of

any pixel in the frame buffer must be located on a fixed grid. This is because pixels are inherently discrete and we specify their locations using integers. OpenGL's API structure is similar to most modern graphics API's including Java3D and DirectX. Therefore, any effort you put into learning OpenGL will carry over to other software platforms. That being said, OpenGL is important to learn because unlike DirectX, OpenGL is a cross-platform system so you can write applications across Windows, Mac, and Linux operating systems. Most OpenGL applications will be accessing functions through three libraries. Functions in the main GL library have names that begin with the letters 'gl' and are stored in a library. The next library is the Graphics Library Utility (GLU) which uses only GL functions but contains code for creating common objects and simplifying viewing. In order to interface with the window system and get input from external devices into our programs, we need at least one more library. For the 'X' Window System this is called GLX, for Windows, it's WGL, and for Macintosh it's AGL. Rather than using a different library for each system, we can use a readily available library called the OpenGL Utility Toolkit (GLUT) which provides the minimal functionality that should be expected in any modern windowing system.



When it comes to primitives and attributes, we've learned a little regarding how GL_LINES and GL_POINTS can be used to draw 2D and 3D objects, but to build more geometric objects we can use the geometric primitives and image/raster primitives. Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves, and surfaces. These primitives pass through a geometric pipeline.



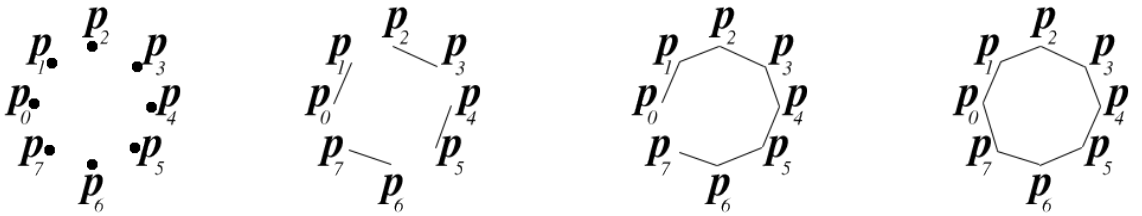
Raster primitives, such as arrays of pixels, lack geometric properties and cannot be manipulated in space in the same way as geometric primitives. They pass through a separate parallel

Drawing With Polygons

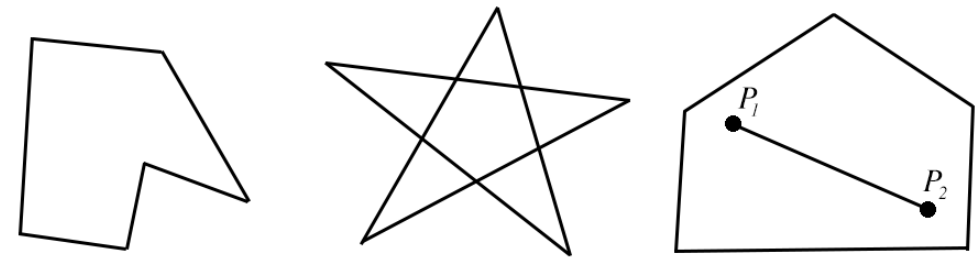
pipeline on their way to the frame buffer. The basic OpenGL geometric primitives are specified by sets of vertices and the programmer defines the objects with sequences of the following form:

```
glBegin(type);
    glVertex*(...);
    .
    .
    .
    glVertex*(...);
glEnd();
```

The value of type specifies how OpenGL assembles the vertices to define geometric objects. Other code and OpenGL function calls can occur between `glBegin` and `glEnd`. For example, we can change attributes or perform calculations for the next vertex between `glBegin` and `glEnd`. Finite sections of lines between two vertices called **line segments** in contrast to lines that are infinite in extent are very important in geometry and computer graphics.

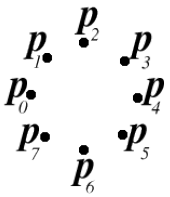


Polygons play a special role in computer graphics because we can display them rapidly and use them to approximate arbitrary surfaces. The performance of graphics systems is characterized by the number of polygons per second that can be rendered. A polygon can be rendered in a couple of ways. We can either render its edges or the interior with a solid color or pattern. However, the outer edges of a polygon are easily defined by an ordered list of vertices, if the interior is not well defined, then the lists of vertices may not be rendered in a desirable manner, or even rendered at all. In two dimensions, as long as no two edges of a polygon cross each other, we have a simple polygon. Although the locations of the vertices determine whether or not a polygon is simple, we can ask what a graphics system will do if it is given a nonsimple polygon to display.

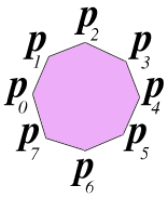


An object is convex if all points on the line segment between any two points inside the object or on its boundary are inside the object. Convex objects include triangles, tetrahedra, rectangles,

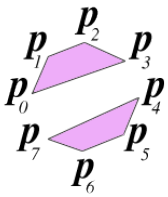
circles, spheres, and parallelepipeds. In three dimensions, polygons present a few more difficulties because, unlike all two-dimensional objects, all the vertices that define the polygon do not need to be located in the same plane. These basic principles regarding polygons help us to understand drawing and connecting vertices to line segments for 2D and 3D spaces. By learning these basics, we can move on to polygon types in OpenGL.



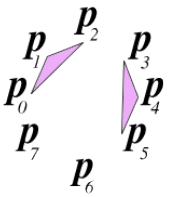
GL_POINTS



GL_POLYGON

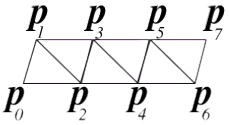


GL_QUADS

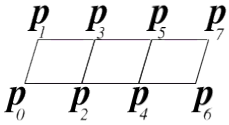


GL_TRIANGLE

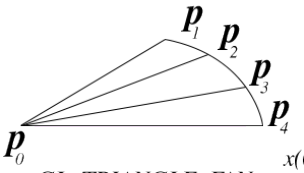
Fans and strips allow us to approximate many curved surfaces with ease. This means we can construct an approximation to a sphere to use a set of polygons defined by lines of latitude and longitude as shown below. We can do this very efficiently using either quad strips or triangle strips.



GL_TRIANGLE _STRIP

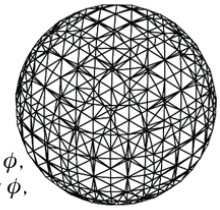


GL_QUAD_STRIP



GL_TRIANGLE_FAN

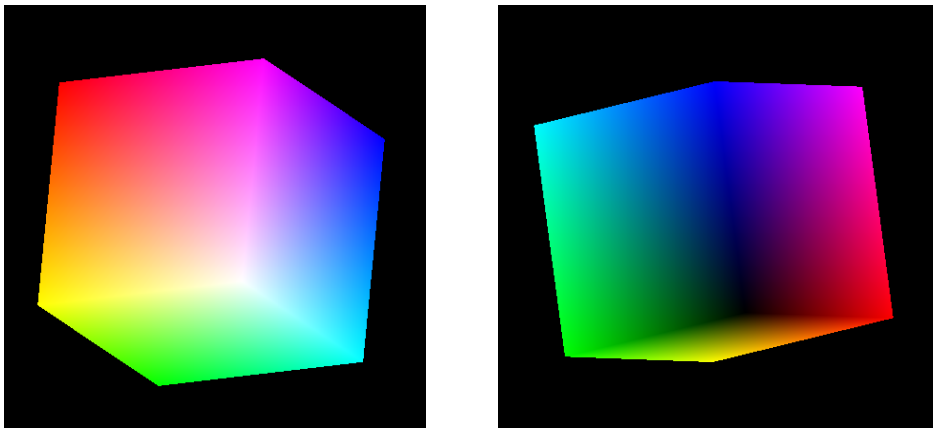
$x(\theta, \phi) = \sin\theta \cos\phi,$
$y(\theta, \phi) = \cos\theta \cos\phi,$
$z(\theta, \phi) = \sin\phi.$



Considering a unit sphere, we can describe it by the following three equations above. If we fix θ and draw curves we can change Φ, we get circles of constant latitude. By generating points at fixed increments of θ and Φ, we can define quadrilaterals as shown in the above sphere. Remembering that we must convert degrees to radians for the standard trigonometric functions, the code for the quadrilaterals corresponding to increments of 20 degrees in θ and to 20 degrees in Φ is:

```
for(phi=-80.0; phi<=80.0; phi+=20.0){
     phir=c*phi;
     phir20=c*(phi+20);
     glBegin(GL_QUAD_STRIP);
     for(theta=-180.0; theta<=180.0; theta+=20.0){
          thetar=c*theta;
          x=sin(thetar)*cos(phir);
          y=cos(thetar)*cos(phir);
          z=sin(phir);
          glVertex3d(x,y,z);
          x=sin(thetar)*cos(phir20);
          y=cos(thetar)*cos(phir20);
          z=sin(phir20);
          glVertex3d(x,y,z);
     }
     glEnd();
}
```

Hopefully with the information I've covered so far, it'll be much easier to grasp what is going on in the code. With that being said, let's jump deeper into the code. One of the first OpenGL programs I experimented with was a glui cube program. GLUI ***(GLUT-based C++ User Interface Library)*** properties provide controls such as buttons, checkboxes, radio buttons and spinners to OpenGL applications. The program demonstrates the use of 3D homogeneous coordinate transformations and simple data structures to rotate a cube with color interpolation on the axis of the users choosing. Both normals and colors are assigned to the vertices and the cube is centered at the origin so *"unnormalized"* normals are the same as the vertex values. In other words, an object becomes brighter if we orient it towards a light source. The orientation of an object surface plays an important role in the amount of light it reflects *(and thus how bright it looks like)*.

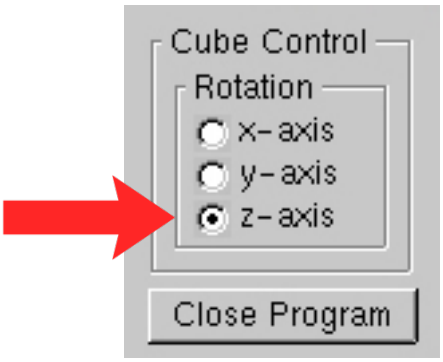


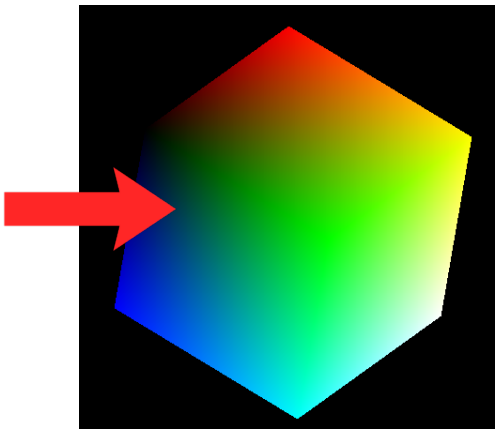
To begin we initialize the axis flag so the radio button begins on the z-axis, and the main render window. Next, we pass a series of floats to *theta* which applies all RGB values to the cube. **Example:** *GLfloat vertices[][3] = {{***(R)***-1.0,***(G)***-1.0,***(B)***-1.0}* the intersecting points of the vertices have floats which represent RGB values from left to right. Beginning on **line 23**, we use pointers to the windows and some of the controls we'll create.

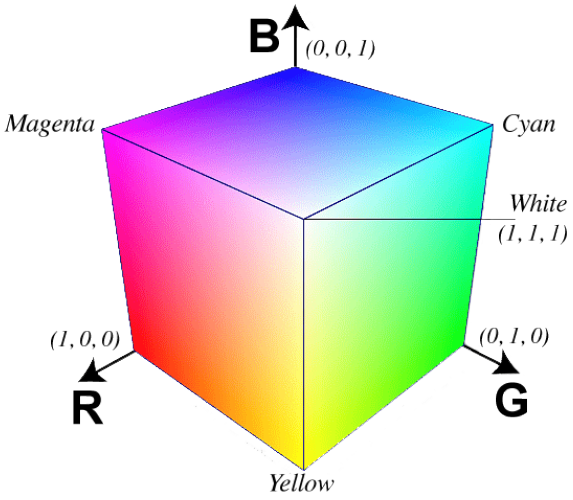
```
 1 #include <stdlib.h>
 2 #include <GLUT/glut.h>
 3 #include "GL/glui.h"
 4 #include <GLFW/glfw3.h>
 5
 6 int   axis=2;
 7 int   axis_flag=2;
 8 int   main_window;
 9 static GLfloat theta[] = {0.0,0.0,0.0};
10
11         GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
12         {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
13         {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
14
15         GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
16         {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
17         {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
18
19         GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
20         {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
21         {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
22
23 GLUI           *glui;
24 GLUI_RadioGroup *radio;
25 GLUI_Panel      *obj_panel;
```

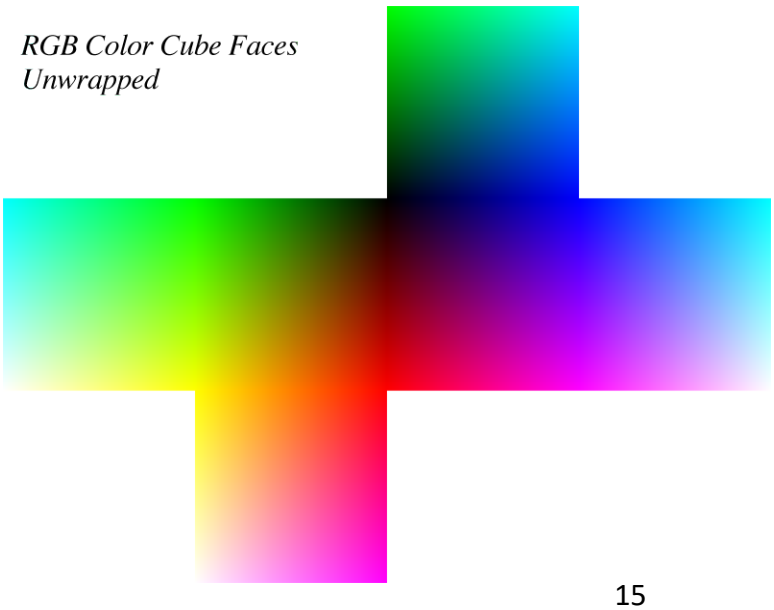
At this stage, we create callback functions for our GLUI control menu to display at runtime. We set the axis of rotation and then beginning on **line 34** we write a function to draw a polygon via the list of vertices. Next, on **line 51** we have to write a function to map the vertices to the faces of the cube so the cube will display with all six sides.

```
26
27 void control_cb( int control ) {
28
29   if ( control == 1 ) {
30     axis = axis_flag;
31   }
32 }
33
34 void polygon(int a, int b, int c , int d) {
35         glBegin(GL_POLYGON);
36                 glColor3fv(colors[a]);
37                 glNormal3fv(normals[a]);
38                 glVertex3fv(vertices[a]);
39                 glColor3fv(colors[b]);
40                 glNormal3fv(normals[b]);
41                 glVertex3fv(vertices[b]);
42                 glColor3fv(colors[c]);
43                 glNormal3fv(normals[c]);
44                 glVertex3fv(vertices[c]);
45                 glColor3fv(colors[d]);
46                 glNormal3fv(normals[d]);
47                 glVertex3fv(vertices[d]);
48         glEnd();
49 }
50
51 void colorcube(void) {
52         polygon(0,3,2,1);
53         polygon(2,3,7,6);
54         polygon(0,4,7,3);
55         polygon(1,2,6,5);
56         polygon(4,5,6,7);
57         polygon(0,1,5,4);
58 }
```

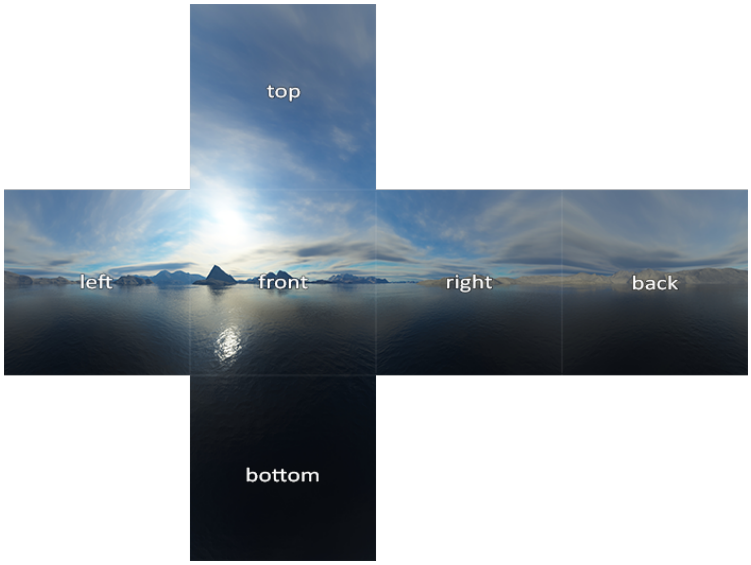








You may have seen the above-unwrapped example before in game development or VR scenarios which are commonly referred to as ***skyboxes***. This is an example of an unwrapped skybox that can be used in 3D environments to make the world around you seem bigger than it is.



These images are mapped to the inside of a cuboid for the player controller objects to move around in 3D space. This can be performed in OpenGL programming, game engines, or 3D render software.

These examples should provide a clearer insight into how the faces are mapped to the color cube that we are using in the GLUI cube program. Beginning on **line 60**, we display the callback, clear the frame buffer, the Z-buffer, draw and rotate the cube, and then swap the buffers. On **line 73** we write our spinCube function that controls the functionality of the cube. This is an idle callback that spins the cube .5 degrees around the selected *'Z'* axis, and then the GLUI code checks for the display window.

```
60 void display(void) {
61         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
62         glLoadIdentity();
63         glRotatef(theta[0], 1.0, 0.0, 0.0);
64         glRotatef(theta[1], 0.0, 1.0, 0.0);
65         glRotatef(theta[2], 0.0, 0.0, 1.0);
66
67         colorcube();
68
69         glFlush();
70         glutSwapBuffers();
71 }
72
73 void spinCube() {
74         theta[axis] += 0.5;
75         if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
76
77         if ( glutGetWindow() != main_window )
78               glutSetWindow(main_window);
79         glutPostRedisplay();
80         radio->set_int_val(axis);
81 }
```

Mouse interaction in graphics programs is useful for experimenting with quaternions and essential if you have keyboard input reserved for other functionality. This void function allows us to create mouse button inputs that can be used within the GLUT window for rotating the cube on the X, Y, and Z axes. I will discuss more examples with mouse interaction shortly.

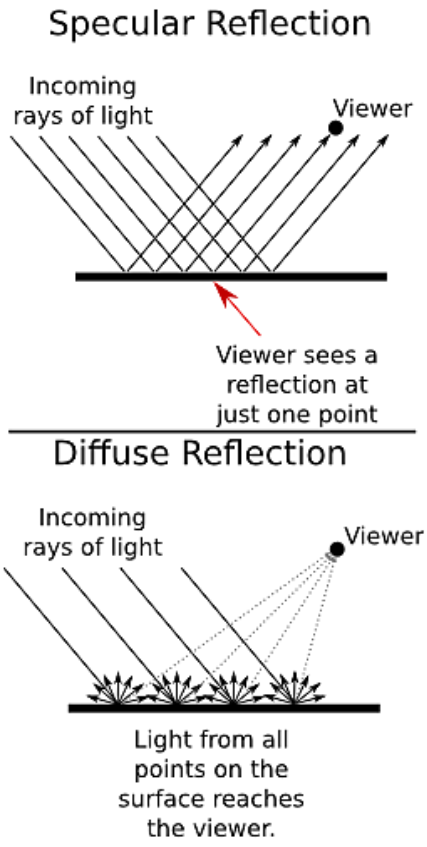
```
83 void mouse(int btn, int state, int x, int y) {
84
85         if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
86         if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
87         if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
88 }
```

0 = X
1 = Y
2 = Z

One of the great advantages of OpenGL programming is understanding how objects are stored and manipulated in 3D space. This type of knowledge also transitions to using render software like 3Ds Max, Maya, and Blender. Before there were full GUI *(Graphical User Interface)* software applications, programmers had to create all cameras, light sources, and primitive objects using VPLs like OpenGL and DirectX. The reality of writing sizable programs with VPLs is by the time you're done writing; you've basically written a miniature game engine.

Image generation with OpenGL primitives is a great step in expanding your knowledge base with camera coordinates and light sources. This init function initializes a material property, light source, lighting model, and depth buffer.

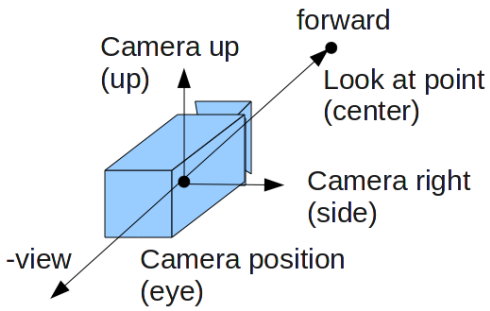
```
18 void init(void) {
19    GLfloat mat_specular[] = { .5, 1.0, .5, 1.0 };
20    GLfloat mat_shininess[] = { 30.0 };
21    GLfloat mat_diffuse[] = { .5, 1.0, .5, 0.0 };
22
23    GLfloat light_position1[] = { 1.0, 1.0, 1.0, 0.0 };
24    GLfloat light_position2[] = { -2.0, 1.5, 2.0, 0.0 };
25
26
27    GLfloat blue_light[] = { 0.0, 0.0, 2.0, 1.0 };
28    GLfloat red_light[] = { 2.0, 0.0, 0.0, 1.0 };
29
30    glClearColor (0.0, 0.0, 0.0, 0.0);
31    glShadeModel (GL_SMOOTH);
32
33    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
34    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
35    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
36
37
38    glLightfv(GL_LIGHT0, GL_POSITION, light_position1);
39    glLightfv(GL_LIGHT0, GL_SPECULAR, blue_light);
40    glLightfv(GL_LIGHT1, GL_POSITION, light_position2);
41    glLightfv(GL_LIGHT1, GL_SPECULAR, red_light);
42    glEnable(GL_LIGHTING);
43    glEnable(GL_LIGHT0);
44    glEnable(GL_LIGHT1);
45    glEnable(GL_DEPTH_TEST);
46    glMatrixMode(GL_PROJECTION);
47 }
```



Diffuse and specular lighting adds good quality of illumination to your scene by adding better perspective to viewpoints and depth of objects. In regard to creating primitive objects, it's not nearly as fast as using 3D render software, but we can still create them in OpenGL without too much trouble.

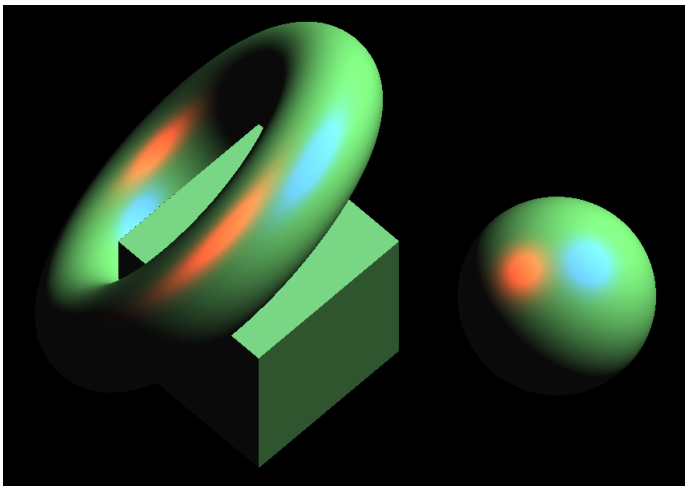
We have our lighting model, and now we need to add objects to the scene. In this display function, we use gluLookAt to place the camera at (eye) position, target direction, and up vector. In a simpler way, this means the camera is placed in front of the central origin of our scene. On **line 59** we use glPushMatrix as a function call for pushing the current matrix stack down by one, duplicating the current matrix. This is also referred to as a transformation matrix. The glTranslated and glTranslatef functions will control the angle and position of our objects. The glRotated function on **lines 61 and 62** controls the angle of the axis for rotation.

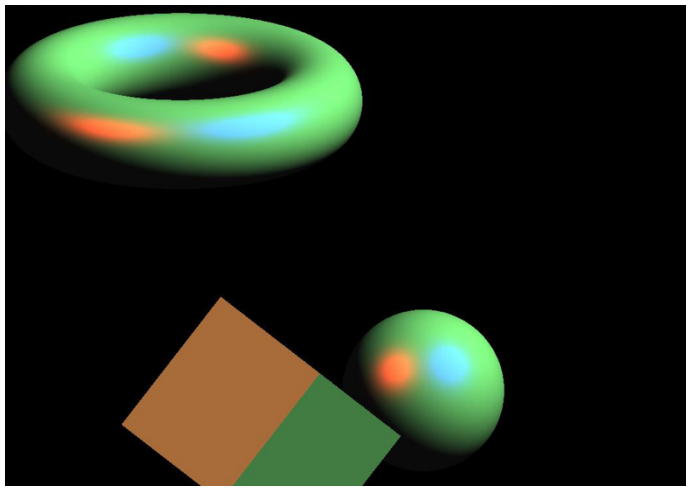
```
50 void display(void) {
51    gluLookAt(cameraX, cameraY, cameraZ, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
52
53    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
54    glPushMatrix();
55    glTranslated(1.0, 0, 0);
56    glutSolidSphere (.5, 200, 160);
57    glPopMatrix();
58
59    glPushMatrix();
60    glTranslated(-.5, 0, 0);
61    glRotated(30.0, 1, 0, 0 );
62    glRotated(45.0, 0, 1, 0 );
63    glutSolidCube (1.0);
64    glPopMatrix ();
65
66    glPushMatrix();
67    glTranslatef(-.75, 0.5, 0.0);
68    glRotatef(90.0, 1.0, 1.0, 0.0);
69    glutSolidTorus(0.275, 0.85, 100, 150);
70    glPopMatrix();
71
72    glFlush ();
73 }
```



This is the result of changing the gltranslate coordinates for the torus and cube from –
Torus: -.75, 0.5, 0.0 to -.75, 0.5, 1.5
and the cube: -.5, 0, 0 to 1.5, .45, 1

Camera coordinates set to X: 0, Y: 3, Z: 6

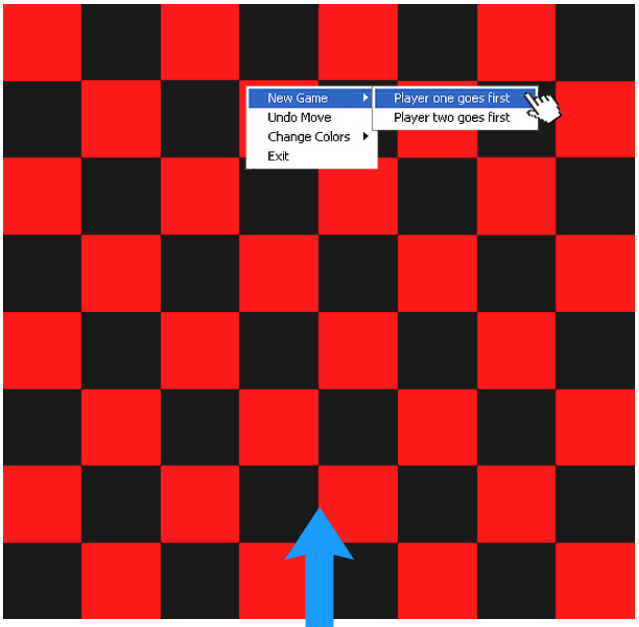


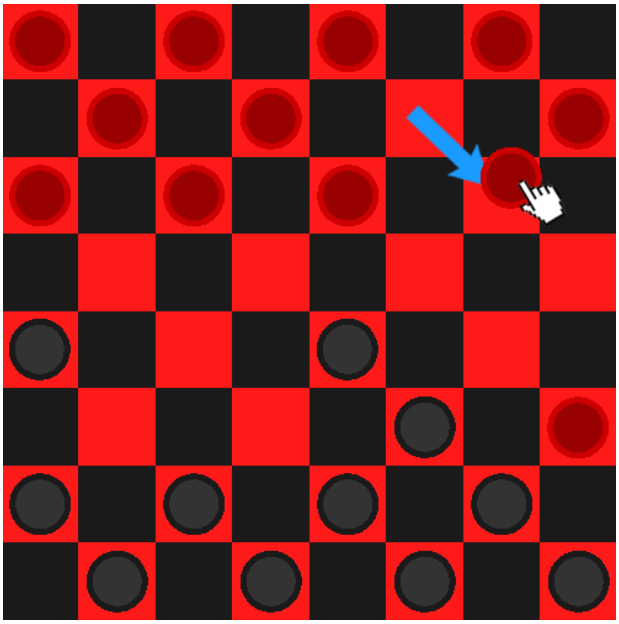


In OpenGL, we need to have the functionality in place to resize the windows at runtime. By doing so, the program has to understand the ability to resize *(re-translate)* the objects inside the GLUT window.

```
75 void reshape (int w, int h) {
76    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
77    glMatrixMode (GL_PROJECTION);
78    glLoadIdentity();
79    if (w <= h)
80       glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
81          1.5*(GLfloat)h/(GLfloat)w, -10.0, 100.0);
82    else
83       glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
84          1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 100.0);
85    glMatrixMode(GL_MODELVIEW);
86    glLoadIdentity();
87 }
```

Back on the topic of mouse interaction, these are bits and pieces of a Checkers and Chess program that I wrote in OpenGL for an interaction assignment in my Computer Graphics class.





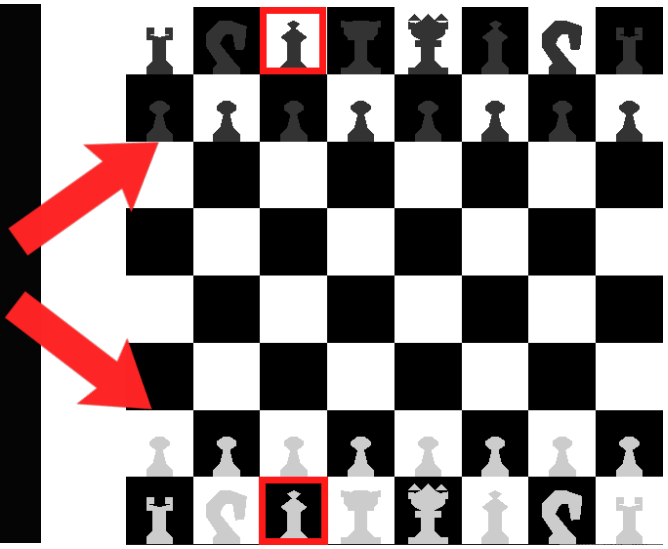
```
 93 void drawboard() {
 94       int x, y;
 95
 96       glPushMatrix();
 97       for(x = 0; x <= 7; x++) {
 98            for(y = 0; y <= 7; y++) {
 99                 glBegin(GL_QUADS);
100                 if((x + y) %2==0) {
101                      glColor3f(1.0, 0.0, 0.0);
102                 }
103                 else {
104                      glColor3f(0.2, 0.2, 0.2);
105                 }
106
107                 glVertex2f(x, y);
108                 glVertex2f(x+1.0, y);
109                 glVertex2f(x+1.0, y+1.0);
110                 glVertex2f(x, y+1.0);
111                 glEnd();
112            }
113       }
114       glPopMatrix();
115 }
116
117  void drawCircle(float x, float y, float radius, float* color) {
118       glColor3fv(color);
119       glBegin(GL_POLYGON);
120            for(GLfloat i = 0.0; i < 2.0 * M_PI; i+= 0.1)
121                 glVertex2f(radius * cos(i) + x, radius * sin(i) + y);
122         glEnd();
123 }
```

```
144 void checkers::mousedown(int ix, int iy) {
145      int myname = pick(ix, iy);
146      if(myname == -1) return;
147      int x = myname / 10;
148      int y = myname % 10;
149      if(x < 0 || x >= 8 || y < 0 || y >= 8) return;
150      if((isP1(x, y) && p1Turn) || (isP2(x, y) && !p1Turn)
151      || (jumped && x == selectedx && y == selectedy)) {
152           selectedx = x;
153           selectedy = y;
154           selectedtype = board[x][y];
155           piecex = (float)x;
156           piecey = (float)y;
157           offsetx = toBoardX(ix) - piecex;
158           offsety = toBoardY(iy) - piecey;
159           board[x][y] = 0;
160           moving = true;
161      }
162 }
163
164 void checkers::mouseup(float fx, float fy) {
165      if(!moving) return;
166      int x = ftoi(piecex + 0.5);
167      int y = ftoi(piecey + 0.5);
168      if(isLegalStep(x, y)) step(x, y);
169      else if(isLegalJump(x, y)) jump(x, y);
170      else cancelMove();
171      moving = false;
172 }
173
174 void checkers::mousemove(float x, float y) {
175      if(!moving) return;
176      piecex = x - offsetx;
177      piecey = y - offsety;
178 }
```

To begin drawing the Checkerboard, we consider the model on an 8x8 square grid. On **line 94**, we start by declaring two variables for the board where ‘x’ is the row and ‘y’ is the column. The first for loop is iterating through the horizontal axis, and the next one is looping through the vertical axis. On **line 100**, the if statement is checking for red or black squares and the next line we are giving a function call to select the red squares, otherwise we select the black squares. We declare four vertices clockwise from **lines 107 to 111** to face the player.

The next three functions from **lines 144 to 178** are to assist with moving the player pieces, checking for legal moves, and jumping other player pieces. Just as I used GL_POLYGON to draw the checkers pieces, in a more complex manner to build chess pieces, we learned earlier in this guide that we can draw and manipulate GL_QUADS, and GL_TRIANGLES at the vertex, which is exactly what I've done here to draw a 2D Bishop inside of a switch statement:

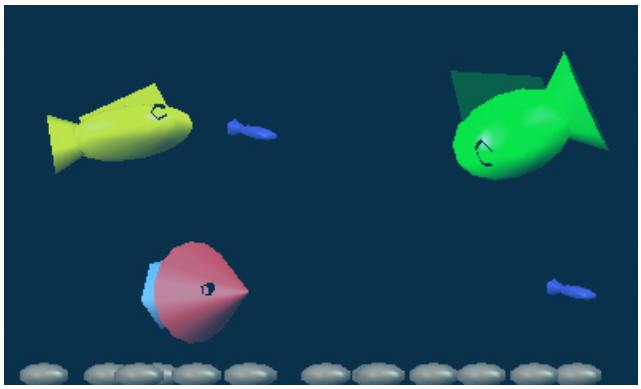
```
280          case BISHOP:
281               glBegin(GL_QUADS);
282                    glVertex2f(0.3, 0.0); //base
283                    glVertex2f(0.3, 0.1);
284                    glVertex2f(0.7, 0.1);
285                    glVertex2f(0.7, 0.0);
286                    glVertex2f(0.3, 0.1); //base connector
287                    glVertex2f(0.4, 0.2);
288                    glVertex2f(0.6, 0.2);
289                    glVertex2f(0.7, 0.1);
290                    glVertex2f(0.4, 0.2); //stem
291                    glVertex2f(0.4, 0.5);
292                    glVertex2f(0.6, 0.5);
293                    glVertex2f(0.6, 0.2);
294                    glVertex2f(0.5, 0.4); //large diamond
295                    glVertex2f(0.3, 0.55);
296                    glVertex2f(0.5, 0.7);
297                    glVertex2f(0.7, 0.55);
298                    glVertex2f(0.5, 0.65); //small diamond
299                    glVertex2f(0.4, 0.75);
300                    glVertex2f(0.5, 0.8);
301                    glVertex2f(0.6, 0.75);
302               glEnd();
303               break;
```

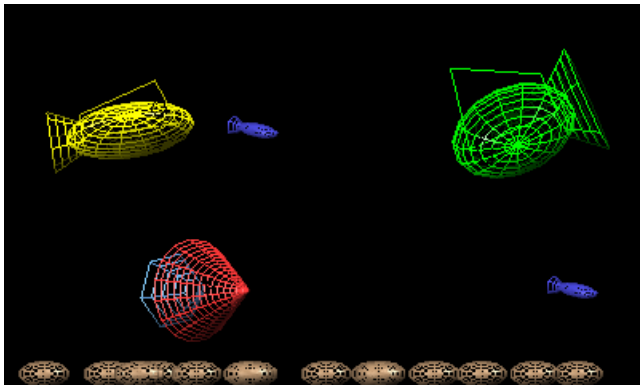


Obviously, this is only an example of one piece on the Chessboard and the Pawn and Bishop were the two easiest pieces to make. While this was a very tedious process, it is an excellent way of learning more about drawing and manipulating quads, triangles, and polygons in OpenGL. Just as we translated mouse functions to allow the user to move player pieces around the Checkers board, we do the same thing in Chess, using similar interaction methods:

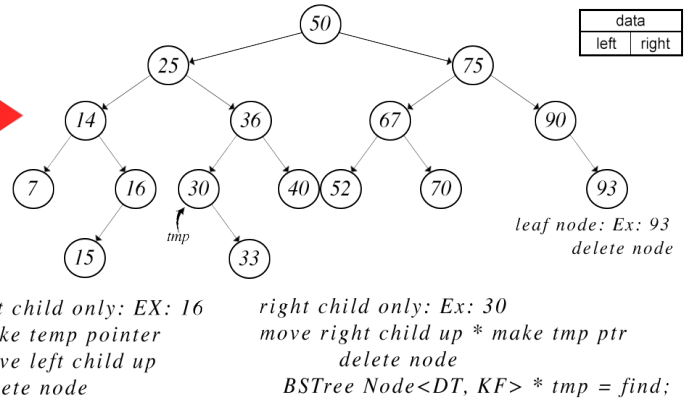
```
370 void chess::draw(bool isPicking) {
371      for(int i = 0; i < 8; i++)
372           for(int j = 0; j < 8; j++) {
373                glColor3fv((i+j)%2==0?boardcolor1:boardcolor2);
374                glBegin(GL_POLYGON);
375                     glVertex2i(i, j);
376                     glVertex2i(i+1, j);
377                     glVertex2i(i+1, j+1);
378                     glVertex2i(i, j+1);
379                glEnd();
380                if(board[i][j] != 0) drawpiece(i + 0.5, j + 0.5, board[i][j], isPicking?(ftoi(i)*10 + ftoi(j)):-1);
381           }
382      if(moving) drawpiece(piecex + 0.5, piecey + 0.5, selectedtype, -1);
383      if(upgrading_pawn) {
384           glColor3f(1.0, 1.0, 1.0);
385                glBegin(GL_POLYGON);
386                     glVertex2i(2, 4);
387                     glVertex2i(2, 5);
388                     glVertex2i(6, 5);
389                     glVertex2i(6, 4);
390           glEnd();
391           drawpiece(2.5, 4.5, (p1Turn?QUEEN+10:QUEEN), (p1Turn?QUEEN+10:QUEEN));
392           drawpiece(3.5, 4.5, (p1Turn?KNIGHT+10:KNIGHT), (p1Turn?KNIGHT+10:KNIGHT));
393           drawpiece(4.5, 4.5, (p1Turn?BISHOP+10:BISHOP), (p1Turn?BISHOP+10:BISHOP));
394           drawpiece(5.5, 4.5, (p1Turn?ROOK+10:ROOK), (p1Turn?ROOK+10:ROOK));
395      }
396 }
```

As far as drawing a tile or board grid of regular polygons in two dimensions, my examples with the Checkers and Chess boards are good examples of tessellation. We can also use similar function calls for three-dimensional space rather than two-dimensional. This is a fish tank program I made for a scene modeling assignment that uses GUI manipulators with a tree-traversal approach to draw and manipulate a basic 3D scene using primitive objects such as spheres, cylinders, cones, and boxes.





The objects are placed in a treenode, and then the child and sibling nodes are traversed in the data structure. I also used this approach, created a visual diagram, and explained it in my previous AI Battleships guide. I used similar principles in my fish tank program.




```
58 typedef struct treenode {
59   GLfloat m[16];
60   void (*f)();
61   struct treenode *sibling;
62   struct treenode *child;
63 }treenode;
64
65 treenode tank, pebble_1, pebble_2, pebble_3, pebble_4,
66 pebble_5, pebble_6, pebble_7, pebble_8, pebble_9,
67 pebble_10, pebble_11, pebble_12, pebble_13, pebble_14,
68 fish_1, fish1_tail, fish1_eyes,
69 fish_2, fish_3, fish_4, fish_5, fish_6;
70
71 void traverse(treenode* root) {
72    if(root==NULL) return;
73       glPushMatrix();
74       glMultMatrixf(root->m);
75       root->f();
76       if(root->child!=NULL) traverse(root->child);
77    glPopMatrix();
78    if(root->sibling!=NULL) traverse(root->sibling);
79 }
```

Creating light sources to illuminate the scene.

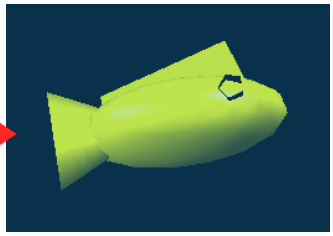
```
168 void lightinit() {
169    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
170    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
171    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
172    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
173
174    glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
175    glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
176    glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
177    glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
178
179    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
180    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
181    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
182    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
183
184    glShadeModel(GL_SMOOTH);
185    glEnable(GL_LIGHTING);
186    glEnable(GL_COLOR_MATERIAL);
187    glEnable(GL_LIGHT0);
188    glEnable(GL_LIGHT1);
189    glDepthFunc(GL_LEQUAL);
190    glEnable(GL_DEPTH_TEST);
191    glEnable(GL_NORMALIZE);
192    glEnable(GL_BLEND);
193    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
194    glClearColor(1.0, 1.0, 1.0, 1.0);
195 }
```

We learned earlier that objects in three-dimensional space are really just manipulated primitive objects like spheres, cubes, rectangular boxes, cones and more. In many ways, this is the exact same situation in the real world. My fish tank is just a rectangular box made of five GL_QUADS with one *"plate of glass"* removed from the top of the rectangle. As you can see from **line 199** the RGBa color values from left to right give me the glass look with opacity applied. The remaining values are just creating scalability for width, height, and length of the tank parts. This is by no means the end-all code for drawing the tank, but it does allow us to have a function for drawing the tank parameters.

```
198 void draw_tank() {
199    glColor4f(0.1, 0.5, 1.0, 0.3);
200    glBegin(GL_QUADS);
201       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/2.0); //bottom
202       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/-2.0);
203       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/-2.0);
204       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/2.0);
205
206       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/2.0); //left
207       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/2.0, TANK_LENGTH/2.0);
208       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/2.0, TANK_LENGTH/-2.0);
209       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/-2.0);
210
211       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/2.0); //right
212       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/2.0, TANK_LENGTH/2.0);
213       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/2.0, TANK_LENGTH/-2.0);
214       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/-2.0);
215
216       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/2.0); //front
217       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/2.0);
218       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/2.0, TANK_LENGTH/2.0);
219       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/2.0, TANK_LENGTH/2.0);
220
221       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/-2.0); //back
222       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/-2.0, TANK_LENGTH/-2.0);
223       glVertex3f(TANK_WIDTH/2.0, TANK_HEIGHT/2.0, TANK_LENGTH/-2.0);
224       glVertex3f(TANK_WIDTH/-2.0, TANK_HEIGHT/2.0, TANK_LENGTH/-2.0);
225    glEnd();
226 }
```

```
266 void draw_submarinefish_1() {
267    glColor3f(1.0, 1.0, 0.0);
268    glPushMatrix();
269       glScalef(0.8, 0.3, 0.6);
270       if(wireframe) {
271          glutWireSphere(0.5, 15, 15);
272       } else {
273          glutSolidSphere(0.5, 15, 15); //body
274       }
275    glPopMatrix();
276    glPushMatrix();
277       glTranslatef(-0.5, 0.0, 0.0);
278       glRotatef(90.0, 0.0, 1.0, 0.0);
279       gluCylinder(quadric_object, 0.2, 0.1, 0.2, 6, 4); //tail
280    glPopMatrix();
281    glPushMatrix();
282       if(wireframe) {
283          glBegin(GL_LINE_LOOP);
284       } else {
285          glBegin(GL_POLYGON); //dorsal fin
286       }
287          glVertex3f(-0.3, 0.0, 0.0);
288          glVertex3f(0.3, 0.0, 0.0);
289          glVertex3f(0.2, 0.3, 0.0);
290          glVertex3f(-0.3, 0.15, 0.0);
291       glEnd();
292    glPopMatrix();
293    glColor3f(0.0, 0.0, 0.0);
294    glPushMatrix();
295       glTranslatef(0.2, 0.02, -0.3);
296       glRotatef(90.0, 0.0, 0.0, 1.0);
297       gluCylinder(quadric_object, 0.05, 0.05, 0.6, 5, 1); //eyes
298    glPopMatrix();
299 }
```

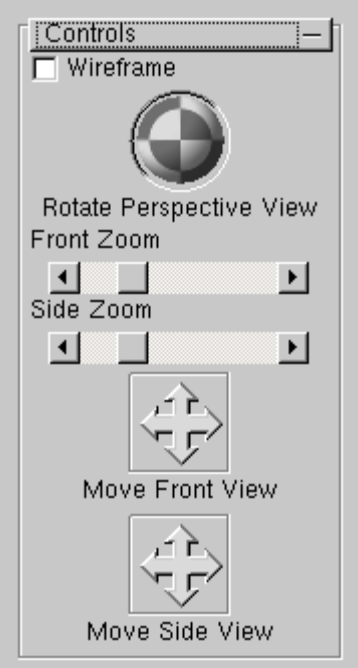
This function draws the parameters of the yellow submarine fish in my tank.



I can also use Bezier patches and vertices to draw the Utah Teapot, a teacup and a teaspoon directly into my fish tank scene.

Creating GLUI Menus/Control Panels

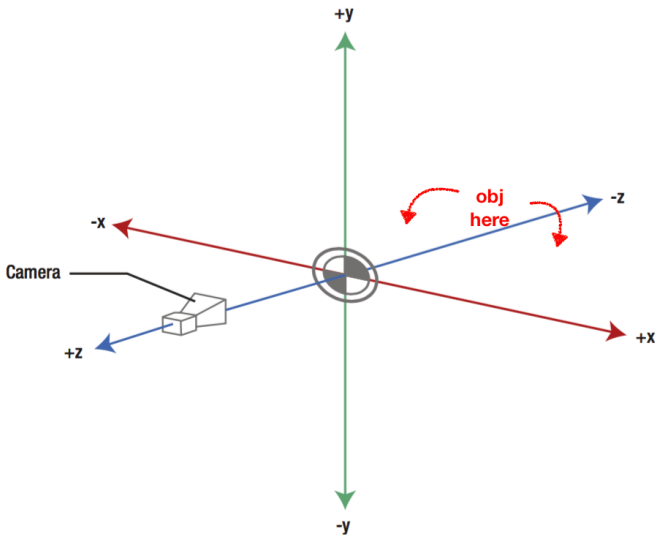
OpenGL is very good at creating GLUI menus and controls for our applications. This is the code I wrote for the GLUI controls in my fish tank program. Beginning on **line 576**, I create the side sub-window *(master control panel)*. The next following lines of code are just adding the buttons for the required functionality that the assignment calls for within the control panel. As you can see, this is done with little effort.




```
576   glui = GLUI_Master.create_glui( "GLUI", 0, 400, 50 );
577
578   /***** Control for object params *****/
579
580   GLUI_Rollout* obj_panel = new GLUI_Rollout(glui, "Controls", false );
581
582   new GLUI_Checkbox( obj_panel, "Wireframe", &wireframe);
583
584   GLUI_Rotation *view_rot = new GLUI_Rotation(obj_panel, "Rotate Perspective View", view_rotate);
585   view_rot->set_spin( 1.0 );
586
587   new GLUI_StaticText( obj_panel, "Front Zoom");
588   GLUI_Scrollbar* fz = new GLUI_Scrollbar(obj_panel, "Front Zoom", GLUI_SCROLL_HORIZONTAL,
589                                           &front_zoom);
590   fz->set_float_limits(0.1,4.0);
591
592   new GLUI_StaticText( obj_panel, "Side Zoom" );
593   GLUI_Scrollbar* sz = new GLUI_Scrollbar(obj_panel, "Side Zoom", GLUI_SCROLL_HORIZONTAL,
594                                           &side_zoom);
595   sz->set_float_limits(0.1,4.0);
596
597   GLUI_Translation *trans_front =
598     new GLUI_Translation(obj_panel, "Move Front View", GLUI_TRANSLATION_XY, pan_front);
599   trans_front->set_speed( .05 );
600
601   GLUI_Translation *trans_side =
602     new GLUI_Translation(obj_panel, "Move Side View", GLUI_TRANSLATION_XY, pan_side);
603   trans_side->set_speed( .05 );
604
605    GLUI_Master.set_glutIdleFunc(refresh);
606
607    glutMainLoop();
608 }
```

Quaternions With Matrix Algebra

No matter what kind of 3D environment you're working in whether it's for computer graphics or game development, you're most likely going to deal with quaternions at some point. A **quaternion** is a complex number of the form $w + xi + yj + zk$, where w, x, y, z are real numbers and i, j, k are imaginary units that satisfy certain conditions. Quaternions provide a convenient mathematical notation for representing spatial orientations and rotations of elements in three-dimensional space. An ordered pair of real numbers $z = (a, b)$ is a complex number. The first component is called the real part and the second component is called the imaginary part. Moreover, equality, addition, subtraction, multiplication and division are defined as follows:



$(a, b) = (c, d)$ if and only if $a = c$ and $b = d$

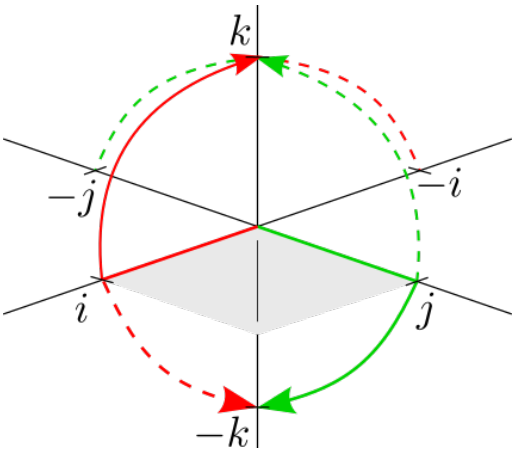
$(a, b) \pm (c, d) = (a \pm c, b \pm d)$

$(a, b)(c, d) = (ac - bd, ad + bc)$

$$\frac{(a, b)}{(c, d)} = \left(\frac{ac+bd}{c^2+d^2}, \frac{bc-ad}{c^2+d^2}\right)$$

Quaternions are typically represented as:

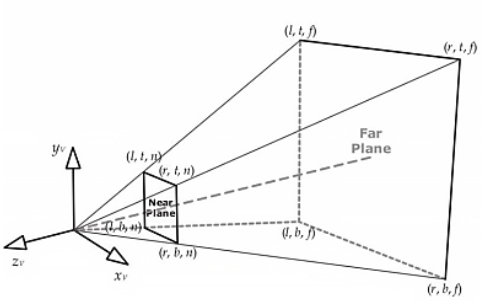
$a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k}$



While they are used in pure mathematics, they're also used in practical and gaming applications in OpenGL. We can write our own game engines that simulate physics to roll or launch tumbling rigid bodies in a 3D environment. This can be done using the same real-world formulas and just programming them in using OpenGL. This also brings us to another topic when using quaternions, and that is matrix multiplication. Matrix multiplication is the most important matrix operation. In practical applications, it's used in areas such as network theory, linear-system equations, transformation of co-ordinate systems, and population dynamics. However, the usefulness of matrix multiplication in computer graphics is its ability to convert geometric data into different coordinate systems with scaling, translations, and rotations. Put bluntly, matrix multiplication is really just a compact way of representing a series of vectors you want to apply in the direction of a motion vector in rows and columns.

The OpenGL fixed pipeline provides 4 different types of matrices which I refer to as the MAT4: (GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE and GL_COLOR). The transformation routines for these matrices are as follows; **glLoadIdentity()**, **glTranslatef()**, **glRotatef()**, **glScalef()**, **glMultMatrixf()**, **glFrustum()** and **glOrtho()** which I have used in multiple code examples so far. In linear algebra *(also referred to as matrix algebra),* matrix splitting is an expression that represents a given matrix as a sum or difference of matrices. The questions arise, why do we do this? What is the point of splitting matrices, why is it so important, and how does it benefit us in OpenGL? The short answer is, ever since 1992 with all first revisions of OpenGL 1x and 2x there was a projection matrix, but no model or view matrix. Ergo, we created our own using matrix multiplication. It wasn't until this year that OpenGL 3.0 added some new blocks to the pipeline using vertex and fragment shaders with GLSL. Hence, my earlier example of the OpenGL 3.0 pipeline that I made on page 5. The older methods were *"replaced"* by GLSL programming. However, by splitting our projection matrix, and combining our model and view matrices, we retrieve coordinates for the perspective matrix.

Perspective Projection (OpenGL)



$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Programming Shaders With GLSL

This brings me to the final section of this guide with programming vertex and fragment shaders using GLSL programming. Shaders use **GLSL** (Graphics Library Shading Language) a unique language similar to C syntax which executes directly on the graphics pipeline. The language uses a lot more math involving vectors and matrices, and is easily the most complex programming language I've written in. One of my assignments for my Computer Graphics class involved displaying Moiré effects and Fresnel reflection with sinusoidal functions programmed through vertex and fragment shaders to run on a programmable Nvidia GPU. **Part 1** called for computing the color in a fragment shader using the following sinusoidal function:

$$\text{Output}_{\text{color}} = 0.5 * (1 + \sin(\text{val} * x^2 + \text{val} * y^2))$$

I used the fragment's texture coordinate (gl_TexCoord[0].xy) for the x and y values with val passed as the uniform variable scaleValue (e.g. scaleValue.x). At runtime, the program begins at absolute zero, and 'scaleValue' can be changed infinitely in the positive or negative direction via the '+' and '–' keys as the program runs.
The syntax would be represented as:

val = scaleValue
x == gl_TexCoord[0].x
y == gl_TexCoord[0].y

vec4 a *(vec4 is a data type of 4 floats)*
{ x, y, z, w }
{ r, g, b, a }

a.x a.g
a.z a.a
a[0] – a[3]

Moiré effects on a quadrilateral plane

The quadrilateral of the vertex grid is composed of 400 smaller quadrilaterals. The vertex shader will be written to manipulate the vertices before the quadrilaterals are rasterized. For the animation of the vertex grid, we use the variable, 'theTime' in order to achieve the animated movement of the plane. This is a mathematical representation of how the uniform variable 'theTime' will be used in this program:

$$newObjSpacePos_x = oldObjSpace_x$$
$$newObjSpacePos_y = oldObjSpace_y + 5 * theTime^2 * (oldObjSpace_x^2 + oldObjSpace_y^2)$$
$$newObjSpacePos_z = oldObjSpace_z$$

For the vertex shader, I'm passing the color from glColor*f ('gl_Color') directly down to the fragment shader without modifying it. Then, I multiply the color by the input from the vertexShader, and outputs the result as the final fragment color. Next, we pass the texture coordinate for unit #0 (gl_multiTexCoord0) down unmodified.

```
uniform float theTime;

void main( void ) {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_Position.y = gl_Position.y + 5 * theTime * theTime * (gl_Position.x *
  gl_Position.x + gl_Position.y * gl_Position.y);

  gl_FrontColor = gl_Color;
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

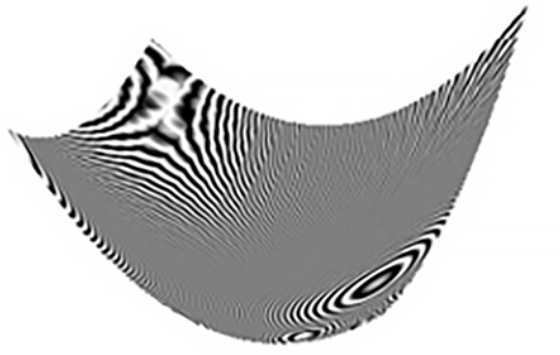
Here we initialize static data types, and GLSL implementation.

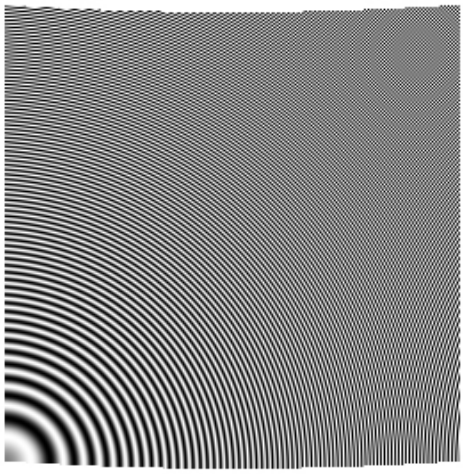
```
static void init() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
}

static void initGLSL(const GLchar* vShaderFile, const GLchar* fShaderFile) {
    program = glCreateProgramObjectARB();
    initShaderv(program, vShaderFile);
    initShaderf(program, fShaderFile);
    glUseProgramObjectARB(program);
}
```

We need implementation in place to draw, manipulate, and animate the plane in the program, so the following draw function helps to accomplish that functionality.

```
static void draw() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode( GL_MODELVIEW );

    glUseProgram(program);
    setParameterf(program, "theTime", theTime);
    setParameter3f(program, "scaleValue", valueR, valueG, valueB );
    glLoadIdentity();
    int i,j;

         glColor3f(0.0, 0.0, 0.0);
         glRotatef(xRot, 1.0, 0.0, 0.0);
         glRotatef(yRot, 0.0, 1.0, 0.0);
         glRotatef(zRot, 0.0, 0.0, 1.0);

    glBegin( GL_QUADS );
    for (i = 0; i<QUAD_TESSELATION;i++)
        for (j=0; j<QUAD_TESSELATION;j++) {
            glTexCoord3f( i*DELTA_QUAD, j*DELTA_QUAD, 0 );
            glVertex3f( -.5+i*DELTA_QUAD, -.5+j*DELTA_QUAD , 0 );

            glTexCoord3f( (i+1)*DELTA_QUAD, j*DELTA_QUAD, 0 );
            glVertex3f( -.5+(i+1)*DELTA_QUAD, -.5+j*DELTA_QUAD , 0 );

            glTexCoord3f( (i+1)*DELTA_QUAD, (j+1)*DELTA_QUAD, 0 );
            glVertex3f( -.5+(i+1)*DELTA_QUAD, -.5+(j+1)*DELTA_QUAD , 0 );

            glTexCoord3f( i*DELTA_QUAD, (j+1)*DELTA_QUAD, 0 );
            glVertex3f( -.5+i*DELTA_QUAD, -.5+(j+1)*DELTA_QUAD , 0 );
        }
    glEnd();

    glFlush();
    glutSwapBuffers();
}
```

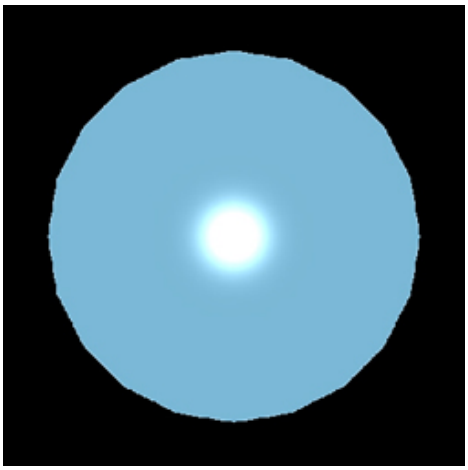
After we successfully have the code in place for the keyboard inputs, we can scale up the vertices and rotate the plane. The following output images demonstrate this functionality and the Moiré effects for the vertex and fragment shaders.





Part 2 of the assignment called for a simplified version of the Fresnel reflection model as an approximation based on the work of Fresnel Schlick. Fresnel reflection occurs in transparent and opaque materials. The effect is that as incident light becomes oriented closer to the gazing angle of the surface, more light is reflected yielding a slight glowing appearance.
The approximated Fresnel formula is:

Phong illumination model

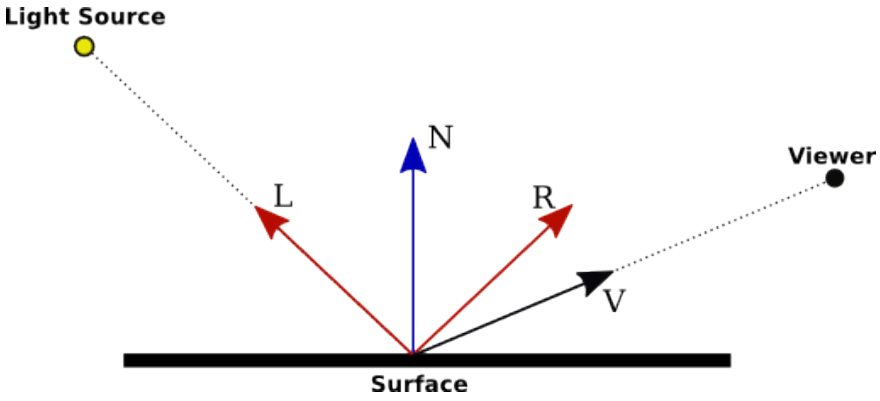


$$F = R_s + (1 - R_s)(1 - \cos\alpha)^5$$

where

$$R_s = (n_1 - n_2)/(n_1 + n_2)$$

α is the angle between the lighting direction and the normal n_1, n_2 depends on the index of reflection but generally are in the range $0.0 - 2.0$
For this assignment we use $n_1 = 1.0$, $n_2 = 2.0$



The Fresnel effect produces specular highlights only around the outer edges, specifically in the direction of the light. In this code example, I pass the model view matrix down the fragment shader and multiply the position by the vertexShader and then output the specular reflection around the silhouette of the illumination model.

```
varying vec3 N;
varying vec3 v;

void main(void) {

    v = vec3(gl_ModelViewMatrix * gl_Vertex);
    N = normalize(gl_NormalMatrix * gl_Normal);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

For the vertex portion of the Fresnel effect, we're at the point of origin with the Eye Coordinates so 'EyePos' is (0,0,0). In this function we also calculate the lighting direction of ambient, diffuse, specular, total terms, Fresnel shading, and then write the total colors to the Phong model.

```
varying vec3 N;
varying vec3 v;
uniform float phong;
uniform float fresnel;

void main (void) {

   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 E = normalize(-v);
   vec3 R = normalize(-reflect(L,N));

   vec4 Iamb = gl_FrontLightProduct[0].ambient;

   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);

   vec4 Ispec = gl_FrontLightProduct[0].specular
              * pow(max(dot(R,E),0.0), gl_FrontMaterial.shininess);

  vec4 Itot = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;

  float n1 = 1.0;
  float n2 = 2.0;
  float r = (n1 - n2)/(n1 + n2);
  float A = dot(R,E);
  float F = min(max(r + (1-r) * (1-sin(A))*(1-sin(A))*(1-sin(A))*(1-sin(A))*(1-sin(A)), 0.0), 1.0);

  vec4 white = {1.0, 1.0, 1.0, 1.0};
  vec4 black = {0.0, 0.0, 0.0, 0.0};

  if(phong == 1.0 && fresnel == 1.0) {
    gl_FragColor = white * F + Itot * (1.0 - F);
  }
  if(phong == 1.0 && fresnel != 1.0) {
    gl_FragColor = Itot;
  }
  if(fresnel == 1.0 && phong != 1.0) {
    gl_FragColor = black + white * F;
  }
  if(fresnel != 1.0 && phong != 1.0) {
    gl_FragColor = black;
  }
}
```

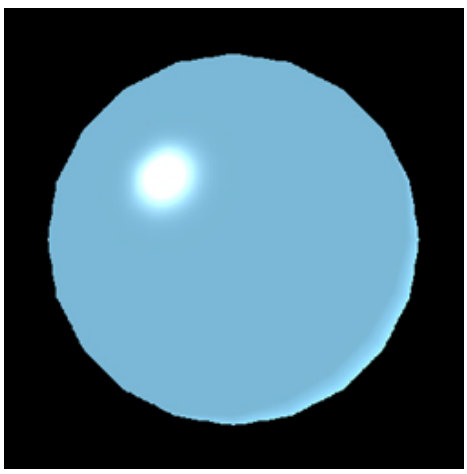
Just as I initialized static data types, and GLSL implementation for the quadrilateral plane, I do the same again passing material and lighting effects for the Phong illumination model.

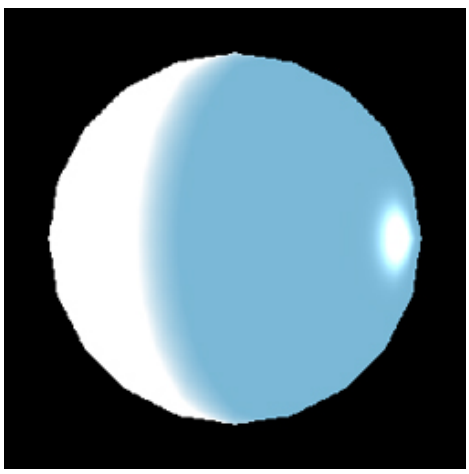
```
21 static void init() {
22     glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
23     glEnable(GL_DEPTH_TEST);
24     glShadeModel(GL_SMOOTH);
25 
26         glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
27         glLightfv(GL_LIGHT0, GL_AMBIENT, lightColor);
28         glLightfv(GL_LIGHT0, GL_DIFFUSE, lightColor);
29         glLightfv(GL_LIGHT0, GL_SPECULAR, lightColor);
30 
31         glMaterialfv(GL_FRONT, GL_AMBIENT, matColor);
32         glMaterialfv(GL_FRONT, GL_DIFFUSE, matColor);
33         glMaterialfv(GL_FRONT, GL_SPECULAR, matColor);
34         glMaterialfv(GL_FRONT, GL_SHININESS, matShininess);
35 
36         glEnable(GL_LIGHTING);
37         glEnable(GL_COLOR_MATERIAL);
38         glEnable(GL_LIGHT0);
39 }
40 
41 static void initGLSL(const GLchar* vShaderFile, const GLchar* fShaderFile) {
42     program = glCreateProgramObjectARB();
43     initShaderv(program, vShaderFile);
44     initShaderf(program, fShaderFile);
45     glUseProgramObjectARB(program);
46 }
```

The Fresnel scene is very straight forward as we are just drawing a sphere and relying on the GLSL implementation to apply the Fresnel effects.

```
48 static void draw() {
49 
50     glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
51     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
52     glMatrixMode( GL_MODELVIEW );
53 
54     glUseProgram(program);
55     setParameterf(program, "phong", phong);
56     setParameterf(program, "fresnel", fresnel);
57     glLoadIdentity();
58 
59         glColor3f(0.4, 0.6, 0.7);
60         glutSolidSphere(0.8, 20, 20);
61 
62     glFlush();
63     glutSwapBuffers();
64 }
```

Once we have all of the code in place for the keyboard input and compile the main files we can run the program to see and control the lighting effects:





Conclusion

While there was a lot of information discussed in this guide, a proper and accurate introduction to this kind of subject cannot be summed up in only a few pages. On a scalable level considering the overall subject, the amount of content I covered was still barely scraping the surface of computer graphics. Even with my small code examples that I explained throughout, had I demonstrated the complete amount of code to write these programs, this guide would have been vastly larger in size. However, I believe the topics I covered were essential to learning some of the more important areas of computer graphics. Thus, helping to provide a better understanding the origins of how this type of technology was created and applied to applications. This was the biggest motivation behind writing this guide in the manner that I have. Overall, I hope this guide was helpful for you, and that my visual content assisted well in the learning process. With the exception of the Toy Story, Simpsons, and weather doppler images, all diagrams, code, and application-rendered images within this guide were created by me and converted digitally from my original notes. If you have any questions about this guide or any other general inquiries, you can email me at technologicguy@gmail.com

---

Resources Used:

- Lengyel, Eric. *Mathematics for 3D Game Programming & Computer Graphics – Second Edition* – 2003
- Engel, Wolfgang. *Programming Vertex and Pixel Shaders* – 2004
- Rost J. Randi, Lichtenbelt, Barthold, Kessenich M. John, Olano Marc. *OpenGL Shading Language* – 2004