**Construction of an Aquarium using OpenGL**

**University of Massachusetts, Lowell**

EECE.5200-201

Spring 2016

Computer Aided Engineering Analysis project

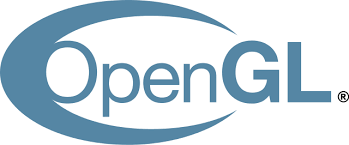
Submitted by

Hema Nalini Routhu

Sindhu Bhupathiraju

**Introduction:**

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications. As a software interface for graphics hardware, OpenGL's main purpose is to render 2D and 3D objects into a frame buffer. These objects are described as sequences of vertices (which define geometric objects) or pixels (which define images). OpenGL performs several processing steps on this data to convert it to pixels to form the final desired image in the frame buffer.



OpenGL constructs shapes from geometric primitives, thereby creating mathematical descriptions of objects. (OpenGL considers points, lines, polygons, images, and bitmaps to be primitives). Arrange the objects in 3-D space and select the desired vantage point for viewing the composed scene.  Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.

Convert the mathematical description of objects and their associated color information to pixels on the screen, this process is called *Rasterization.* A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation.

**OpenGL programming**:

The first thing to do when starting a new OpenGL project is to dynamically link with OpenGL.

* **Windows**: Add opengl32.lib to your linker input.
* **Linux**: Include -lGL in your compiler options.

The process by which a computer creates images from models is called *Rendering*. These models or objects are constructed from geometric primitives (points, lines and polygons) that are specified by their vertices.

The final rendered image consists of pixels drawn on the screen; a *pixel* is the smallest visible element the display hardware can put on the screen. Information about the pixels (for instance, what color they're supposed to be) is organized in memory into bit-planes. A bit-plane is an area of memory that holds one bit of information for every pixel on the screen; the bit might indicate how red a particular pixel is supposed to be, for example. The bit-planes are themselves organized into a *framebuffer*, which holds all the information that the graphics display needs to control the color and intensity of all the pixels on the screen.

**Libraries:**

#include <GL/gl.h>

#include <GL/glu.h>

#include <GL/glut.h>

#include <stdio.h>

#include <stdbool.h>

#include <math.h>

#include <stdlib.h>

#include <jpeglib.h>

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify the programming tasks

* The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix *glu.*
* The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, to hide the complexities of differing window system APIs. GLUT routines use the prefix *glut*.

The Construction of an Aquarium using OpenGL consists of the following divisions:

1. Drawing the objects on a clear window.
2. Texture mapping.
3. Surface of the Aquarium (NURBS curves and surfaces).
4. Display

* Fish
* Stone
* Moss

1. Lighting.
2. Movement of objects inside the Aquarium.

**Drawing the objects on a clear window:**

Drawing on a computer screen is different from drawing on paper in that the paper starts out white, and all you have to do is draw the picture. On a computer, the memory holding the picture is usually filled with the last picture you drew, so you typically need to clear it to some background color before you start to draw the new scene. The color you use for the background depends on the application.

Clearing the window:

Clearing an existing window screen is important as the previous image that has been in the memory will not be erased otherwise. As an example, these lines of code clear an RGBA mode window to black:

glClearColor(0.0, 0.0, 0.0, 0.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

The first line sets the clearing color to black, and the next command clears the entire window to the current clearing color. The single parameter to **glClear()** indicates which buffers are to be cleared. In this case, the program clears only the color buffer, where the image displayed on the screen is kept. Typically, you set the clearing color once, early in your application, and then you clear the buffers as often as necessary. OpenGL keeps track of the current clearing color as a state variable rather than requiring you to specify it each time a buffer is cleared.

Creating a Window:

Five routines perform tasks necessary to initialize a window.

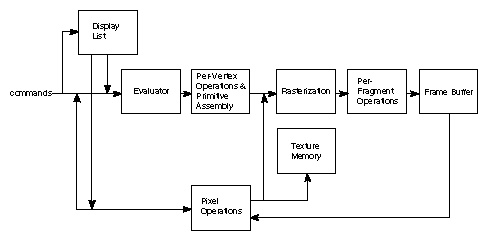
* **glutInit**(int \**argc*, char \*\**argv*) initializes GLUT and processes any command line arguments. **glutInit()** should be called before any other GLUT routine.
* We can also specify whether we want a single- or double-buffered window. Use **glutSetColor()** to do this. We can use this routine to indicate that we want the window to have an associated depth, stencil, and/or accumulation buffer.
* **glutInitWindowPosition**(int*x*, int*y*) specifies the screen location for the upper-left corner of your window.
* **glutInitWindowSize**(int *width*, int *size*) specifies the size, in pixels, of your window.
* int **glutCreateWindow**(char \**string*) creates a window with an OpenGL context. It returns a unique identifier for the new window. Until **glutMainLoop()** is called the window is not displayed.

**Texture Mapping:**

In-order to explain how texture mapping works let us consider an example, if you want to draw a large brick wall without texture mapping, for example, each brick must be drawn as a separate polygon. Without texturing, a large flat wall - which is really a single rectangle - might require thousands of individual bricks and even then the bricks may appear too smooth and regular to be realistic.

Texture mapping allows you to glue an image of a brick wall (obtained, perhaps, by scanning in a photograph of a real wall) to a polygon and to draw the entire wall as a single polygon. Texture mapping ensures that all the right things happen as the polygon is transformed and rendered.

OpenGL rendering structure:



**Surface of the Aquarium (NURBS curves and surfaces):**

The surface of the Aquarium has bit-planes which are controlled by an array of control points. The control points determine the x, y and z dimensions of the 2-D or 3-D surface. By changing the values of different control pixel points, the shape of surface can be changed.

 Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points Although evaluators are the only OpenGL primitive available to draw curves and surfaces directly, and even though they can be implemented very efficiently in hardware, they're often accessed by applications through higher-level libraries. The GLU provides a NURBS (Non-Uniform Rational B-Spline) interface built on top of the OpenGL evaluator commands. If we can understand NURBS, writing OpenGL code to manipulate NURBS curves and surfaces is relatively easy, even with lighting and texture mapping. The following is the line used to create a GLUnurbsObj object and it is initialized before using.

GLUnurbsObj \*theNurb;

* Start the curve or surface by calling **gluBeginCurve()** or **gluBeginSurface()**.
* Call **gluNurbsCurve()** or **gluNurbsSurface()** at least once with the control points (rational or non-rational), knot sequence and order of the polynomial basis function for your NURBS object. We can call these functions additional times to specify surface normal and/or texture coordinates.
* Call **gluEndCurve()** or **gluEndSurface()**to complete the curve or surface.

A set of properties associated with a NURBS object affects the way the object is rendered. These properties include how the surface is rasterized (for example, filled or wireframe) and the precision of tessellation.

void **gluNurbsSurface** (GLUnurbsObj \*nobj, GLint uknot\_count,   
GLfloat \*uknot, GLint vknot\_count, GLfloat \*vknot,   
GLint u\_stride, GLint v\_stride, GLfloat \*ctlarray,   
GLint uorder, GLint vorder, GLenum type);

The above command describes the vertices (or surface normals or texture coordinates) of a NURBS surface, *nobj*. Several of the values must be specified for both u and v parametric directions, such as the knot sequences (uknot and vknot), knot counts (uknot\_count and vknot\_count), and order of the polynomial (uorder and vorder) for the NURBS surface.

gluNurbsSurface(theNurb,

12, knots, 12, knots,

6 \* 3, 3, &ctrlpoints[0][0][0],

6, 6, GL\_MAP2\_VERTEX\_3)

The last parameter, type, is one of the two-dimensional evaluator types. We used GL\_MAP2\_VERTEX\_3 for control points. We also used other types, such as GL\_MAP2\_TEXTURE\_COORD\_2 to calculate and assign texture coordinates as shown below.

gluNurbsSurface(theNurb,

4, tknots, 4, tknots,

4, 2, &texpts[0][0][0],

2, 2, GL\_MAP2\_TEXTURE\_COORD\_2);

For example, to create a lighted (with surface normals) and textured NURBS surface, you may need to call this sequence:

gluBeginSurface(nobj);

gluNurbsSurface(nobj, ..., GL\_MAP2\_TEXTURE\_COORD\_2);

gluNurbsSurface(nobj, ..., GL\_MAP2\_NORMAL);

gluNurbsSurface(nobj, ..., GL\_MAP2\_VERTEX\_3);

gluEndSurface(nobj);

**Display:**

The Display window consists of cross-section of the Aquarium. The inside of the Aquarium consists of bumpy surface with Moss and some stones on it. The Fish present in the Aquarium can be controlled by the Mouse and Keyboard commands. The structures of the Surface, Moss, stone, Fish has different textures and control points which control the shapes and position of each one of them.

Knots:

The knot vector has a length of knot-vector-length = number-of-points + degree-of-curve + 1 and the thing that matters the most is the ratio and not the absolute values. So {0,1,2,3} is the same as {0,2,4,6}.

GLfloat knots[12] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};

GLfloat tknots[4] = {0.0, 0.0, 1.0, 1.0};

The Display function:

void display(void) {

GLfloat knots[12] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};

GLfloat tknots[4] = {0.0, 0.0, 1.0, 1.0};

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

init();

glPushMatrix();

glEnable(GL\_MAP2\_VERTEX\_3);

glEnable(GL\_TEXTURE\_2D);

glBindTexture (GL\_TEXTURE\_2D, texture);

glTexParameteri (GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri (GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

glTexParameteri (GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri (GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

glEnable(GL\_MAP2\_TEXTURE\_COORD\_2);

glMap2f(GL\_MAP2\_TEXTURE\_COORD\_2, 0, 1, 2, 2, 0, 1, 4, 2, &texpts[0][0][0]);

glTexImage2D(GL\_TEXTURE\_2D, 0, 3, 256, 256, 0, GL\_RGBA,

GL\_UNSIGNED\_BYTE, total\_texture);

glRotatef(15.0, 1.0, 1.0, 1.0);

gluBeginSurface(theNurb);

gluNurbsSurface(theNurb,

4, tknots, 4, tknots,

4, 2, &texpts[0][0][0],

2, 2, GL\_MAP2\_TEXTURE\_COORD\_2);

gluNurbsSurface(theNurb,

12, knots, 12, knots,

6 \* 3, 3, &ctrlpoints[0][0][0],

6, 6, GL\_MAP2\_VERTEX\_3);

gluEndSurface(theNurb);

glPopMatrix();

glDisable(GL\_TEXTURE\_2D);

glPushMatrix();

drawStone();

glPopMatrix();

glPushMatrix();

glLoadIdentity(); // Load the Identity Matrix to reset our drawing locations

glTranslatef(0.0f, 0.7f, -1.0f);

glTranslatef(xLocation, yLocation, zLocation); // Translate our object along the y axis

if (movingUp) // If we are moving up

xLocation -= 0.005f; // Move up along our yLocation

else // Otherwise

xLocation += 0.005f; // Move down along our yLocation

if (xLocation < -2.0f) // If we have gone up too far

movingUp = false; // Reverse our direction so we are moving down

else if (xLocation > 2.0f) // Else if we have gone down too far

movingUp = true; // Reverse our direction so we are moving up

if (yLocation < -2.0f) // If we have gone up too far

movingUp = false; // Reverse our direction so we are moving down

else if (yLocation > 2.0f) // Else if we have gone down too far

movingUp = true; // Reverse our direction so we are moving up

//glRotatef(yRotationAngle, 0.0f, 1.0f, 0.0f); // Rotate our object around the y axis

//glRotatef(spin, 0.0, 0.0, 1.0);

drawFish();

glPopMatrix();

/\*glPushMatrix();

glLoadIdentity(); // Load the Identity Matrix to reset our drawing locations

//glTranslatef(0.0f, 0.7f, -1.0f);

//drawStone();

glPopMatrix();\*/

glFlush ();

glutSwapBuffers();

}

**Fish**: *fish.c*

The control points along with the NURBS determine the shape and structure of the fish. The fish is created using Texels, texture filtering procedure.

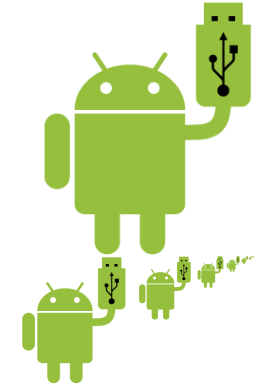
Texels:

Textures in OpenGL are made up of arrays of elements known as *texels*, which contain colour and alpha values. This corresponds with the display, which is made up of a bunch of pixels and displays a different colour at each point. In OpenGL, textures are applied to triangles and drawn on the screen, so these textures can be drawn in various sizes and orientation. The texture filtering options in OpenGL tell it how to filter the texels onto the pixels of the device, depending on the case.

There are three cases:

* Each texel maps onto more than one pixel. This is known as *magnification*.
* Each texel maps exactly onto one pixel. Filtering doesn’t apply in this case.
* Each texel maps onto less than one pixel. This is known as *minification*.

The following figure shows the magnification and minification:

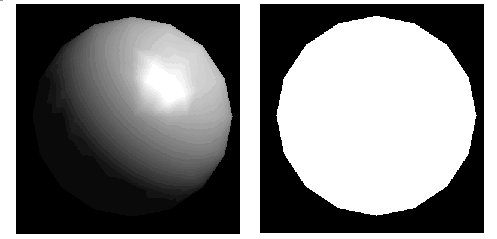


**Stone and Seaweed:**

We also have stone and seaweed in this model. Both the stone and the seaweed have been implemented using NURBS surface, control points. Respective textures from image files are applied for a near realistic appearance. There is no movement for either the stone or the seaweed.

**Lighting:**

OpenGL computes the color of each pixel in a final, displayed scene that's held in the framebuffer. Part of this computation depends on what lighting is used in the scene and on how objects in the scene reflect or absorb that light. As an example of this, recall that the ocean has a different color on a bright, sunny day than it does on a gray, cloudy day. The presence of sunlight or clouds determines whether you see the ocean as bright turquoise or murky gray-green. In fact, most objects don't even look three-dimensional until they're lit. The figure shows two versions of the exact same scene (a single sphere), one with lighting and one without.



With OpenGL, we can manipulate the lighting and objects in a scene to create many different kinds of effects. When we look at a physical surface, our eye's perception of the color depends on the distribution of photon energies that arrive and trigger our cone cells. Those photons come from a light source or combination of sources, some of which are absorbed and some of which are reflected by the surface. In addition, different surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions.

In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called ambient) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off.

In the OpenGL model, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (like headlights on an automobile), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction like a mirror or other shiny surface.

The OpenGL lighting model considers the lighting to be divided into four independent components: emissive, ambient, diffuse, and specular. All four components are computed independently and then added together.

* A*mbient* illumination is light that's been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions
* The *Diffuse* component is the light that comes from one direction
* The *Specular* light comes from a particular direction, and it tends to bounce off the surface in a preferred direction.

GLfloat mat\_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };

GLfloat mat\_specular[] = { 1.0, 1.0, 1.0, 1.0 }; GLfloat mat\_shininess[] = { 100.0 };

**Movement of objects inside Aquarium**:

The Fish can be controlled by using two commands in the program:

1. Mouse command.

By clicking the left button on mouse the fish will be assigned a particular direction.

void mouse(int button, int state, int x, int y) {

switch (button) {

case GLUT\_LEFT\_BUTTON:

if (state == GLUT\_DOWN)

glutIdleFunc(spinDisplay);

break;

case GLUT\_MIDDLE\_BUTTON:

if (state == GLUT\_DOWN)

glutIdleFunc(NULL);

break;

case GLUT\_KEY\_LEFT : deltaAngle = -0.01f; break;

case GLUT\_KEY\_RIGHT : deltaAngle = 0.01f; break;

case GLUT\_KEY\_UP : deltaMove = 0.5f; break;

case GLUT\_KEY\_DOWN : deltaMove = -0.5f; break;

default:

break;

}

}

1. Keyboard Commands:

The fish moves up, down, right, left using the inputs “U”, “D”, “R”, “L” buttons of the keyboard. The x-coordinates determine the left and right movement of Fish. The y-coordinates determine the up and down movement.

void keyboard(unsigned char key, int x, int y) {

switch (key) {

case 27: // ESCAPE key

exit (0);

break;

case 'r':

glPushMatrix();

xLocation += 0.005f;

glPopMatrix();

break;

case 'l':

glPushMatrix();

xLocation -= 0.005f;

glPopMatrix();

break;

case 'u':

glPushMatrix();

yLocation += 0.005f;

glPopMatrix();

break;

case 'd':

glPushMatrix();

yLocation -= 0.005f;

glPopMatrix();

break;

case 'f':

glPushMatrix();

zLocation += 0.005f;

glPopMatrix();

break;

case 'b':

glPushMatrix();

zLocation -= 0.005f;

glPopMatrix();

break;

}

glutPostRedisplay();

}

**3-D View:**

Functionality to view the model by changing the angle of viewing in two directions has been added. By using the up, down, left, right keys, the viewing angle can be changed in two directions.

**Future Scope:**

This model of the aquarium can be further improved by adding separate motion to the tail of the fish. Also fins, and movement to the fins can be added to simulate a more realistic look of fish swimming. More realistic lighting effects can be used to simulate the behavior of light underwater. Shaders can be used

**Program:**

testAquarium.c

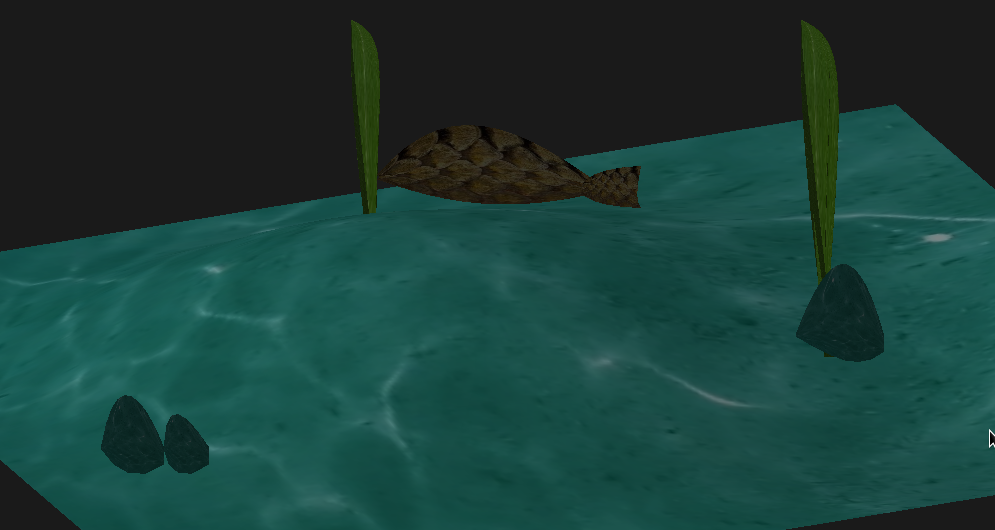
fish.c

stone.c

headers.h

buildAquarium.sh.

**Output**:



**References:**

* <http://www.ics.uci.edu/~gopi/CS211B/opengl_programming_guide_8th_edition.pdf>
* <http://www.glprogramming.com/red/>
* <http://www.learnopengles.com/android-lesson-one-getting-started/>
* <https://www.opengl.org/>
* <https://www.khronos.org/opengles/>