

Computer Graphics

CSE 167 [Win 17], Lecture 6: OpenGL 1

Ravi Ramamoorthi

<http://viscomp.ucsd.edu/classes/cse167/wi17>

To Do

- Questions/concerns about assignment 1?
- Remember it is due Jan 30. Ask me or TAs re problems
- HW 2 (much) more difficult than HW 1
 - START EARLY
 - Will cover all needed material mostly Tue next week

Demo: Surreal (many years ago)



This Lecture

- Introduction to OpenGL and simple demo code
 - mytest1.cpp ; you compiled mytest3.cpp for HW 0
 - Include skeleton code on all platforms for programs
- I am going to show (maybe write) actual code
 - Online code helps you understand HW 2 better
 - ASK QUESTIONS if confused!!
- Simple demo of mytest1 (and maybe hw2)
- This lecture deals with very basic OpenGL setup. Next 2 lectures will likely be more interesting

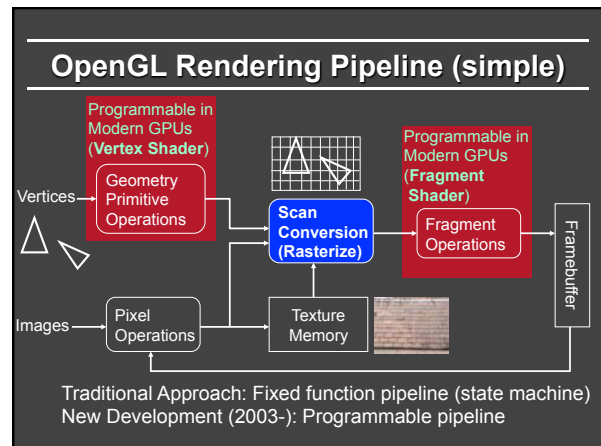
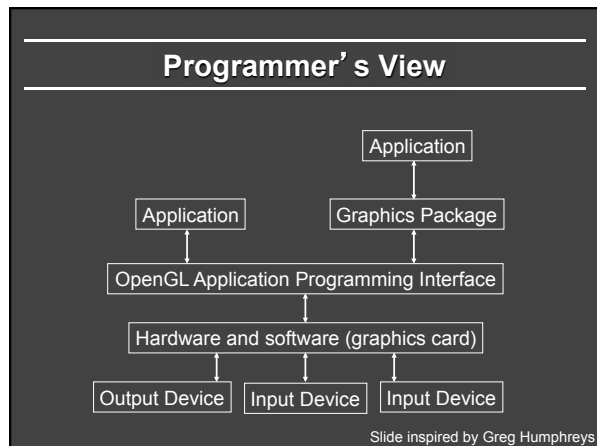
Outline

- *Basic idea about OpenGL*
- Basic setup and buffers
- Matrix modes
- Window system interaction and callbacks
- Drawing basic OpenGL primitives
- Initializing Shaders

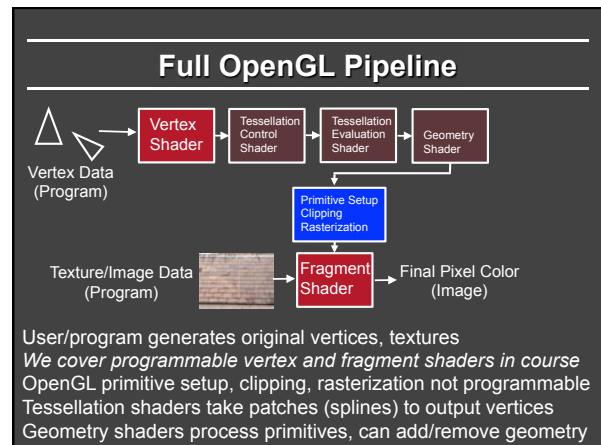
Best source for OpenGL is the redbook. Of course, this is more a reference manual than a textbook, and you are better off implementing rather reading end to end.

Introduction to OpenGL

- OpenGL is a graphics *API*
 - Portable software library (platform-independent)
 - Layer between programmer and graphics hardware
 - Uniform instruction set (hides different capabilities)
- OpenGL can fit in many places
 - Between application and graphics system
 - Between higher level API and graphics system
- Why do we need OpenGL or an API?
 - Encapsulates many basic functions of 2D/3D graphics
 - Think of it as high-level language (C++) for graphics
 - History: Introduced SGI in 92, maintained by Khronos
 - Precursor for DirectX, WebGL, Java3D etc.



- ### GPUs and Programmability
- Since 2003, can write vertex/pixel shaders
 - *Older fixed function pipeline deprecated, not taught*
 - Like writing C programs (see OpenGL book)
 - Performance >> CPU (even used for non-graphics)
 - Operate *in parallel* on all vertices or fragments
 - Are teaching CSE 167 with programmable shaders
 - And modern OpenGL (3.1+)



- ### Outline
- Basic idea about OpenGL
 - *Basic setup and buffers*
 - Matrix modes
 - Window system interaction and callbacks
 - Drawing basic OpenGL primitives
 - Initializing Shaders

- ### Buffers and Window Interactions
- Buffers: Color (front, back, left, right), depth (z), accumulation, stencil. When you draw, you write to some buffer (most simply, front and depth)
 - Buffers also used for vertices etc. Buffer data and buffer arrays (will see in creating objects)
 - No window system interactions (for portability)
 - But can use GLUT / FreeGLUT (or Motif, GLX, Tcl/Tk)
 - Callbacks to implement mouse, keyboard interaction

Basic Setup (can copy; slight OS diffs)

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    // Requests the type of buffers (Single, RGB).
    // Think about what buffers you would need...
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    // Need to add GLUT_3_2_CORE_PROFILE for Apple/Mac OS
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Simple Demo with Shaders");
    // glewInit(); // GLEW related stuff for non-Apple systems
    init (); // Always initialize first

    // Now, we define callbacks and functions for various tasks.
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(mousedrag);
    glutMainLoop(); // Start the main code
    deleteBuffers(); //Termination. Delete buffers generated in init()
    return 0; /* ANSI C requires main to return int. */
}
```

Outline

- Basic idea about OpenGL
- Basic setup and buffers
- *Matrix modes*
- Window system interaction and callbacks
- Drawing basic OpenGL primitives
- Initializing Shaders

Viewing in OpenGL

- *Inspired by old OpenGL. Now, only best practice, not requirement*
 - You could do your own thing, but this is still the best way to develop viewing
- Viewing consists of two parts
 - Object positioning: *model view* transformation matrix
 - View projection: *projection* transformation matrix
- Old OpenGL (no longer supported/taught in 167), two matrix stacks
 - GL_MODELVIEW_MATRIX, GL_PROJECTION_MATRIX
 - Could push and pop matrices onto stacks
- New OpenGL: Use C++ STL templates to make stacks as needed
 - e.g. stack <mat4> modelview; modelview.push(mat4(1.0));
 - GLM libraries replace many deprecated commands. Include mat4
- Convention: camera always at the origin, pointing in the -z direction
- Transformations move objects relative to the camera
- In old OpenGL, *Matrices are column-major and right-multiply top of stack*. (Last transform in code is first actually applied). In new GLM, similarly (read the assignment notes and documentation).

Basic initialization code for viewing

```
#include <GL/glut.h> //also <GL/glew.h>; <GLUT/glut.h> for Mac OS
#include <stdlib.h> //also stdio.h, assert.h, glm, others

int mouseoldx, mouseoldy; // For mouse motion
GLfloat eyeloc = 2.0; // Where to look from; initially 0 -2, 2
glm::mat4 projection, modelview; // The.mvp matrices themselves

void init (void)
{
    /* select clearing color */
    glClearColor (0.0, 0.0, 0.0, 0.0);

    /* initialize viewing values */
    projection = glm::mat4(1.0f); // The identity matrix

    // Think about this. Why is the up vector not normalized?
    modelview = glm::lookAt(glm::vec3(0,-eyeloc,eyeloc),
                           glm::vec3(0,0,0), glm::vec3(0,1,1));
    // (To be cont'd). Geometry and shader set up later ...
```

Outline

- Basic idea about OpenGL
- Basic setup and buffers
- Matrix modes
- *Window system interaction and callbacks*
- Drawing basic OpenGL primitives
- Initializing Shaders

Window System Interaction

- Not part of OpenGL
- Toolkits (GLUT) available (red book: freeglut)
- Callback functions for events
 - Keyboard, Mouse, etc.
 - Open, initialize, resize window
 - Similar to other systems (X, Java, etc.)
- Our main func included

```
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutMouseFunc(mouse);
glutMotionFunc(mousedrag);
```

Basic window interaction code

```
/* Defines what to do when various keys are pressed */
void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 27: // Escape to quit
            exit(0) ;
            break ;
        default:
            break ;
    }
}

/* Reshapes the window appropriately */
void reshape(int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    // Note that the field of view takes in a radian angle
    projection = glm::perspective(30.0f / 180.0f * glm::pi<float>(),
        (GLfloat)w / (GLfloat)h, 1.0f, 10.0f);
    glUniformMatrix4fv(projectionPos,1, GL_FALSE, &projection[0][0]);
    // To send the projection matrix to the shader
}
```

Mouse motion (demo)

```
/* Defines a Mouse callback to zoom in and out */
/* This is done by modifying gluLookAt */
/* The actual motion is in mousedrag */
/* mouse simply sets state for mousedrag */
void mouse(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON) {
        if (state == GLUT_UP) {
            // Do Nothing ;
        }
        else if (state == GLUT_DOWN) {
            mouseoldx = x ; mouseoldy = y ; // so we can move wrt x , y
        }
    }
    else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    { // Reset gluLookAt
        eyeloc = 2.0 ;
        modelview = glm::lookAt(glm::vec3(0, -eyeloc, eyeloc),
            glm::vec3(0, 0, 0), glm::vec3(0, 1, 1));
        // Send the updated matrix to the shader
        glUniformMatrix4fv(modelviewPos,1, GL_FALSE, &modelview[0][0]);
        glutPostRedisplay() ; // Redraw scene
    }
}
```

Mouse drag (demo)

```
void mousedrag(int x, int y) {
    int yloc = y - mouseoldy ; // We will use the y coord to
    zoom in/out
    eyeloc += 0.005*yloc ; // Where do we look from
    if (eyeloc < 0) eyeloc = 0.0 ;
    mouseoldy = y ;

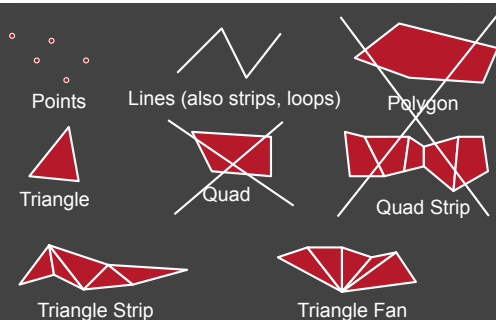
    /* Set the eye location */
    modelview = glm::lookAt(glm::vec3(0, -eyeloc, eyeloc),
        glm::vec3(0, 0, 0), glm::vec3(0, 1, 1));
    // Send the updated matrix over to the shader
    glUniformMatrix4fv(modelviewPos,1, GL_FALSE, &modelview[0][0]);

    glutPostRedisplay() ;
}
```

Outline

- Basic idea about OpenGL
- Basic setup and buffers
- Matrix modes
- Window system interaction and callbacks
- *Drawing basic OpenGL primitives*
- Initializing Shaders

New OpenGL Primitives (fewer)



Geometry

- Points (GL_POINTS)
Stored in Homogeneous coordinates
- Line segments (GL_LINES)
 - Also (GL_LINE_STRIP, GL_LINE_LOOP)
- Triangles (GL_TRIANGLES)
 - Also strips, fans (GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN)
- More complex primitives (GLUT): Sphere, teapot, cube, ...
 - Must now be converted into triangles (which is what skeleton does)

Old OpenGL: Drawing

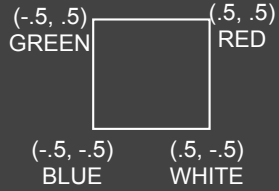
- Enclose vertices between glBegin() ... glEnd() pair
 - Can include normal C code and attributes like the colors
 - Inside are commands like glVertex3f, glColor3f
 - Attributes must be set *before* the vertex
- Assembly line (pass vertices, transform, shade)
 - These are vertex, fragment shaders on current GPUs
 - *Immediate Mode*: Sent to server and drawn
- Client-Server model (client generates vertices, server draws) even if on same machine
 - glFlush() forces client to send network packet
 - glFinish() waits for ack, sparingly use synchronization
- New OpenGL: **Vertex Array Objects** (next)

Old OpenGL: Drawing (*not used*)

```
void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);

    // draw polygon (square) of unit length centered at the origin
    // This code draws each vertex in a different color.
    // The hardware will blend between them.
    // This is a useful debugging trick. I make sure each vertex
    // appears exactly where I expect it to appear.

    glBegin(GL_POLYGON);
    glColor3f (1.0, 0.0, 0.0);
    glVertex3f (0.5, 0.5, 0.0);
    glColor3f (0.0, 1.0, 0.0);
    glVertex3f (-0.5, 0.5, 0.0);
    glColor3f (0.0, 0.0, 1.0);
    glVertex3f (-0.5, -0.5, 0.0);
    glColor3f (1.0, 1.0, 1.0);
    glVertex3f (0.5, -0.5, 0.0);
    glEnd();
    glFlush ();
}
```



Modern OpenGL: Floor Specification

```
const GLfloat floorverts[4][3] = {
    {0.5, 0.5, 0.0}, {-0.5, 0.5, 0.0}, {-0.5, -0.5, 0.0}, {0.5,
    -0.5, 0.0}
};
const GLfloat floorcol[4][3] = {
    {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}, {1.0, 1.0,
    1.0}
};
const GLubyte floorinds[1][6] = { {0, 1, 2, 0, 2, 3} }; //tris
const GLfloat floorverts2[4][3] = {
    {0.5, 0.5, 1.0}, {-0.5, 0.5, 1.0}, {-0.5, -0.5, 1.0}, {0.5,
    -0.5, 1.0}
};
const GLfloat floorcol2[4][3] = {
    {1.0, 0.0, 0.0}, {1.0, 0.0, 0.0}, {1.0, 0.0, 0.0}, {1.0, 0.0,
    0.0}
};
// all red and on top
const GLubyte floorinds2[1][6] = { {0, 1, 2, 0, 2, 3} }; //tris
```

Modern OpenGL: Vertex Array Objects

```
const int numobjects = 2 ; // number of objects for buffer
const int numberobj = 3 ; // Vertices, colors, indices
GLuint VAOs[numobjects]; // A Vertex Array Object per object
GLuint buffers[numberobj] ; // List of buffers for geometric data
GLuint objects[numobjects]; // For each object
GLenum PrimType[numobjects]; // Primitive Type (triangles, strips)
GLsizei NumElems[numobjects] ; // Number of geometric elements

// Floor Geometry is specified with a vertex array
enum {Vertices, Colors, Elements} ; // For arrays for object
enum {FLOOR, FLOOR2} ; // For objects, for the floor

//-----In init below (creates buffer objects for later use)-----
glGenVertexArrays(numobjects, VAOs); //create unique identifiers
glGenBuffers(numberobj*numobjects, buffers); //and for buffers
void deleteBuffers() { // Like a destructor
    glDeleteVertexArrays(numobjects, VAOs);
    glDeleteBuffers(numberobj*numobjects, buffers);}


```

Modern OpenGL: Initialize Buffers

```
void initobject (GLuint object, GLfloat * vert, GLint sizevert, GLfloat *
col, GLint sizecol, GLubyte * inds, GLint sizeind, GLenum type) {
    int offset = object * numberobj;
    glBindVertexArray(VAOs[object]);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[Vertices+offset]) ;
    glBufferData(GL_ARRAY_BUFFER, sizevert, vert, GL_STATIC_DRAW);
    // Use layout location 0 for the vertices
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[Colors+offset]) ;
    glBufferData(GL_ARRAY_BUFFER, sizecol, col, GL_STATIC_DRAW);
    // Use layout location 1 for the colors
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[Elements+offset]) ;
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeind, inds, GL_STATIC_DRAW);
    PrimType[object] = type;
    NumElems[object] = sizeind;
    // Prevent further modification of this VAO by unbinding it
    glBindVertexArray(0);
}
```

Modern OpenGL: Draw Vertex Object

```
void drawobject(GLuint object) {
    glBindVertexArray(VAOs[object]);
    glDrawElements(PrimType[object], NumElems[object],
        GL_UNSIGNED_BYTE, 0);
    glBindVertexArray(0); //unbind
}

void display(void) {
    glClear (GL_COLOR_BUFFER_BIT); // clear all pixels
    drawobject(FLOOR) ;
    drawobject(FLOOR2) ;
    glFlush (); // start processing buffered OpenGL commands
}
```

Initialization for Drawing, Shading

```
#include "shaders.h"
GLuint vertexshader, fragmentshader, shaderprogram ; // shaders

// Initialization in init() for Drawing
glGenVertexArrays(numobjects, VAOs) ;
glGenBuffers(numperobj*numobjects, buffers) ;

initobject(FLOOR, (GLfloat *) floorverts, sizeof(floorverts), (GLfloat *) floorcol, sizeof (floorcol), (GLubyte *) floorinds, sizeof (floorinds), GL_TRIANGLES) ;
initobject(FLOOR2, (GLfloat *) floorverts2, sizeof(floorverts2), (GLfloat *) floorcol2, sizeof (floorcol2), (GLubyte *) floorinds2, sizeof (floorinds2), GL_TRIANGLES) ;

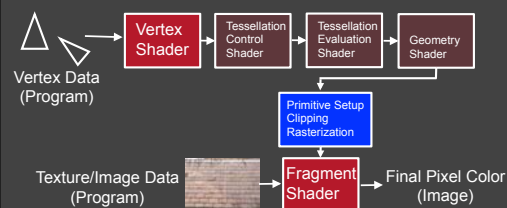
// In init() for Shaders, discussed next
vertexshader = initshaders(GL_VERTEX_SHADER, "shaders/nop.vert") ;
fragmentshader = initshaders(GL_FRAGMENT_SHADER, "shaders/nop.frag") ;
shaderprogram = initprogram(vertexshader, fragmentshader) ;
```

Demo (change colors)

Outline

- Basic idea about OpenGL
- Basic setup and buffers
- Matrix modes
- Window system interaction and callbacks
- Drawing basic OpenGL primitives
- *Initializing Shaders*

Full OpenGL Pipeline



User/program generates original vertices, textures
We cover *programmable vertex and fragment shaders* in course
OpenGL primitive setup, clipping, rasterization not programmable
Tessellation shaders take patches (splines) to output vertices
Geometry shaders process primitives, can add/remove geometry

Simplified OpenGL Pipeline

- User specifies vertices (via vertex arrays)
- For each vertex in parallel
 - OpenGL calls user-specified vertex shader:
Transform vertex (ModelView, Projection), other ops
- For each primitive, OpenGL rasterizes
 - Generates a *fragment* for each pixel the fragment covers
- For each fragment in parallel
 - OpenGL calls user-specified fragment shader:
Shading and lighting calculations
 - OpenGL handles z-buffer depth test unless overwritten
- Modern OpenGL is “lite” basically just a rasterizer
 - “Real” action in user-defined vertex, fragment shaders

Shader Setup

- Initializing (shader itself discussed later)
1. Create shader (Vertex and Fragment)
 2. Compile shader
 3. Attach shader to program
 4. Link program
 5. Use program
- Shader source is just sequence of strings
 - Similar steps to compile a normal program

Shader Initialization Code

```
GLuint initshaders (GLenum type, const char *filename) {
    // Using GLSL shaders, OpenGL book, page 679 of 7th edition
    GLuint shader = glCreateShader(type) ;
    GLint compiled ;
    string str = textFileRead (filename) ;
    const GLchar * cstr = str.c_str() ;
    glShaderSource (shader, 1, &cstr, NULL) ;
    glCompileShader (shader) ;
    glGetShaderiv (shader, GL_COMPILE_STATUS, &compiled) ;
    if (!compiled) {
        shadererrors (shader) ;
        throw 3 ;
    }
    cout<<"Shader file " <<filename<<" successfully compiled."<<endl;
    return shader ;
}
```

Linking Shader Program

```
GLuint initprogram (GLuint vertexshader, GLuint fragmentshader)
{
    GLuint program = glCreateProgram() ;
    GLint linked ;
    glAttachShader (program, vertexshader) ;
    glAttachShader (program, fragmentshader) ;
    glLinkProgram (program) ;
    glGetProgramiv (program, GL_LINK_STATUS, &linked) ;
    if (linked) glUseProgram (program) ;
    else {
        programerrors (program) ;
        throw 4 ;
    }
    cout<<"Shader program successfully attached and linked." << endl;
    return program ;
}
```

Basic (nop) vertex shader

- In shaders/ nop.vert.glsl nop.frag.glsl
 - Written in GLSL (GL Shading Language)
 - Vertex Shader (out values interpolated to fragment)

```
# version 330 core
// Do not modify the above version directive to anything older.

// Shader inputs
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
// Shader outputs, if any
out vec3 Color;
// Uniform variables
uniform mat4 modelview;
uniform mat4 projection;

void main() {
    gl_Position = projection * modelview * vec4(position, 1.0f);
    Color = color; // Just forward this color to the fragment shader
}
```

Basic (nop) fragment shader

```
# version 330 core
// Do not modify the version directive to anything older than 330.

// Fragment shader inputs are outputs of same name from vertex shader
in vec3 Color;

// Uniform variables (none)

// Output
out vec4 fragColor;

void main (void)
{
    fragColor = vec4(Color, 1.0f);
}
```