

Computer Graphics

CSE 167 [Win 17], Lecture 8: OpenGL 2

Ravi Ramamoorthi

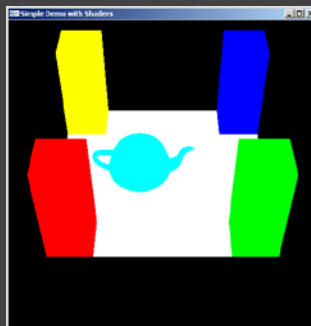
<http://viscomp.ucsd.edu/classes/cse167/wi17>

To Do

- Continue working on HW 2. Can be difficult
- Class lectures, programs primary source
- Can leverage many sources (GL(SL) book, excellent online documentation, see links class website)
- It is a good idea to copy (and modify) relevant segments

Methodology for Lecture

- Make mytest1 more ambitious
- Sequence of steps
- Demo



Review of Last Demo

- Changed floor to all white, added global for teapot and teapotloc, moved geometry to new header file
- Demo 0 [set DEMO to 4 all features]

```
#include <GL/glut.h> //also <GL/glew.h>; <GLUT/glut.h> for Mac OS
#include "shaders.h"
#include "geometry.h"

int mouseoldx, mouseoldy ; // For mouse motion
GLfloat eyeloc = 2.0 ; // Where to look from; initially 0 -2, 2
GLfloat teapotloc = -0.5 ; // ** NEW ** where the teapot is located
GLint animate = 0 ; // ** NEW ** whether to animate or not
GLuint vertexshader, fragmentshader, shaderprogram ; // shaders

const int DEMO = 0 ; // ** NEW ** To turn on and off features
```

Outline

- Review of demo from last lecture
- *Basic geometry setup for cubes (pillars), colors*
 - *Single geometric object, but multiple colors for pillars*
- Matrix Stacks and Transforms (draw 4 pillars)
- Depth testing (Z-buffering)
- Animation (moving teapot)
- Texture Mapping (wooden floor)
- Best source for OpenGL is the red book and GLSL book. Of course, this is more a reference manual than a textbook, and you are better off implementing rather reading end to end.

Geometry Basic Setup 1

```
const int numobjects = 2 ; // number of objects for buffer
const int numobj = 3 ;
const int ncolors = 4 ;
GLuint VAOs[numobjects+ncolors], teapotVAO ; // VAO (Vertex Array
Object) for each primitive object
GLuint buffers[numobj*numobjects+ncolors], teapotbuffers[3] ; //
** NEW ** List of buffers for geometric data
GLuint objects[numobjects] ; // ** NEW ** For each object
GLenum PrimType[numobjects] ;
GLsizei NumElems[numobjects] ;

// For the geometry of the teapot
std::vector<glm::vec3> teapotVertices;
std::vector<glm::vec3> teapotNormals;
std::vector<unsigned int> teapotIndices;

// To be used as a matrix stack for the modelview.
std::vector<glm::mat4> modelviewStack;
```

Geometry Basic Setup 2

```
// ** NEW ** Floor Geometry is specified with a vertex array
// ** NEW ** Same for other Geometry

enum {Vertices, Colors, Elements} ; // For arrays for object
enum {FLOOR, CUBE} ; // For objects, for the floor

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, 1.0, 1.0}, {1.0, 1.0, 1.0}, {1.0, 1.0, 1.0}, {1.0, 1.0, 1.0}
} ;
const GLubyte floorinds[1][6] = { {0, 1, 2, 0, 2, 3} } ;
const GLfloat floortex[4][2] = {
    {1.0, 1.0}, {0.0, 1.0}, {0.0, 0.0}, {1.0, 0.0}
} ;
```

Cube geometry (for pillars)

```
const GLfloat wd = 0.1 ;
const GLfloat ht = 0.5 ;
const GLfloat _cubecol[4][3] = {
    {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}, {1.0, 1.0,
    0.0} } ;
const GLfloat cubeverts[8][3] = {
    {-wd, -wd, 0.0}, {-wd, wd, 0.0}, {wd, wd, 0.0}, {wd, -wd, 0.0},
    {-wd, -wd, ht}, {wd, -wd, ht}, {wd, wd, ht}, {-wd, wd, ht}
} ;
GLfloat cubecol[12][3] ;
const GLubyte cubeinds[12][3] = {
    {0, 1, 2}, {0, 2, 3}, // BOTTOM
    {4, 5, 6}, {4, 6, 7}, // TOP
    {0, 4, 7}, {0, 7, 1}, // LEFT
    {0, 3, 5}, {0, 5, 4}, // FRONT
    {3, 2, 6}, {3, 6, 5}, // RIGHT
    {1, 7, 6}, {1, 6, 2} // BACK
} ;
```

Initialize Geometry Function

```
// This function takes in a vertex, color, index and type array
void initobject(GLuint object, GLfloat * vert, GLint sizevert, GLfloat * col, GLint
sizecol, GLubyte * inds, GLint sizeind, GLenum type) {
    int offset = object * numperobj ;
    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);}
```

Initialize Cubes with Colors 1

```
void initcubes(GLuint object, GLfloat * vert, GLint sizevert, GLubyte *
inds, GLint sizeind, GLenum type) {
    for (int i = 0; i < ncolors; i++) {
        for (int j = 0; j < 8; j++)
            for (int k = 0; k < 3; k++)
                cubecol[j][k] = _cubecol[i][k];
        glBindVertexArray(VAOs[object + i]);
        int offset = object * numperobj;
        int base = numobjects * numperobj;
        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[base + i]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(cubecol), cubecol,
        GL_STATIC_DRAW);
    }
```

Initialize Cubes with Colors 2

```
...
    // 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); }
}
//in init
initobject(FLOOR, (GLfloat *) floorverts, sizeof(floorverts), (GLfloat
*) floorcol, sizeof (floorcol), (GLubyte *) floorinds, sizeof
(floorinds), GL_TRIANGLES);
initcubes(CUBE, (GLfloat *) cubeverts, sizeof(cubeverts), (GLubyte
*) cubeinds, sizeof(cubeinds), GL_TRIANGLES);
loadteapot();
```

Drawing with/without Colors

```
// And a function to draw with them, similar to drawobject but with color
void drawcolor(GLuint object, GLuint color) {
    glBindVertexArray(VAOs[object + color]);
    glDrawElements(PrimType[object], NumElems[object], GL_UNSIGNED_BYTE, 0);
    glBindVertexArray(0);
}

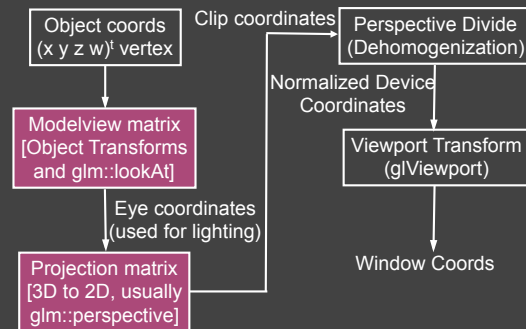
void drawobject(GLuint object) {
    glBindVertexArray(VAOs[object]);
    glDrawElements(PrimType[object], NumElems[object], GL_UNSIGNED_BYTE, 0);
    glBindVertexArray(0);
}

void loadteapot() // See source code for details if interested
```

Outline

- Review of demo from last lecture
 - Basic geometry setup for cubes (pillars), colors
 - Single geometric object, but multiple colors for pillars
 - *Matrix Stacks and Transforms (draw 4 pillars)*
 - Depth testing (Z-buffering)
 - Animation (moving teapot)
 - Texture Mapping (wooden floor)
- Best source for OpenGL is the red book and GLSL book. Of course, this is more a reference manual than a textbook, and you are better off implementing rather reading end to end.

Summary OpenGL Vertex Transforms



Transformations

Matrix Stacks

- Old OpenGL: glPushMatrix, glPopMatrix, glLoad, glMultMatrixf
- Useful for hierarchically defined figures, placing pillars
- Current recommendation is STL stacks managed yourself, which is done in mytest2. (You must manage the stack yourself for HW 2).

Transforms

- Write your own translate, scale, rotate for HW 1 and HW 2
- Careful of OpenGL convention: In old-style, **Right-multiply** current matrix (last is first applied). glm operators follow this sometimes.

Also gluLookAt (glm::lookAt), gluPerspective (glm::perspective)

- Remember just matrix like any other transform, affecting modelview
- See mytest for how to best implement these ideas

Drawing Pillars 1 (in display)

```
// 1st pillar: Right-multiply modelview as in old OpenGL
pushMatrix(modelview) ; // push/pop functions for stack
modelview = modelview * glm::translate(identity, glm::vec3(-0.4,
-0.4, 0.0)) ; // build translation matrix
glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(modelview)[0][0]) ;
drawcolor(CUBE, 0) ;
popMatrix(modelview) ;

// 2nd pillar
pushMatrix(modelview) ;
modelview = modelview * glm::translate(identity, glm::vec3(0.4,
-0.4, 0.0)) ; // build translation matrix
glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(modelview)[0][0]) ;
drawcolor(CUBE, 1) ;
popMatrix(modelview) ;

// Function pushes specified matrix onto the modelview stack
void pushMatrix(glm::mat4 mat) {
    modelviewStack.push_back(glm::mat4(mat)) ; }
```

Drawing Pillars 2

```
// 3rd pillar
pushMatrix(modelview) ;
modelview = modelview * glm::translate(identity,
glm::vec3(0.4, 0.4, 0.0)) ;
glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(modelview)[0][0]) ;
drawcolor(CUBE, 2) ;
popMatrix(modelview) ;

// 4th pillar
pushMatrix(modelview) ;
modelview = modelview * glm::translate(identity,
glm::vec3(-0.4, 0.4, 0.0)) ;
glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(modelview)[0][0]) ;
drawcolor(CUBE, 3) ;
popMatrix(modelview) ;

// This function pops a matrix from the modelview stack void
popMatrix(glm::mat4& mat) {
    if (modelviewStack.size()) {
        mat = glm::mat4(modelviewStack.back()) ;
        modelviewStack.pop_back() ;
    }
    else { // Just to prevent errors when popping from an empty stack.
        mat = glm::mat4(1.0f) ; }
}
```

Demo

- Demo 1
- Does order of drawing matter?
- What if I move floor after pillars in code?
- Is this desirable? If not, what can I do about it?

Outline

- Review of demo from last lecture
 - Basic geometry setup for cubes (pillars), colors
 - Single geometric object, but multiple colors for pillars
 - Matrix Stacks and Transforms (draw 4 pillars)
 - *Depth testing (Z-buffering)*
 - Animation (moving teapot)
 - Texture Mapping (wooden floor)
- Best source for OpenGL is the red book and GLSL book. Of course, this is more a reference manual than a textbook, and you are better off implementing rather reading end to end.

Double Buffering

- New primitives draw over (replace) old objects
- Can lead to jerky sensation
- Solution: double buffer. Render into back (offscreen) buffer. When finished, swap buffers to display entire image at once.
- Changes in main and display

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

glutSwapBuffers() ;

glFlush ();
```

Turning on Depth test (Z-buffer)

OpenGL uses a Z-buffer for depth tests

- For each pixel, store nearest Z value (to camera) so far
- If new fragment is closer, it replaces old z, color ["less than" can be over-ridden in fragment program]
- Simple technique to get accurate visibility
- (Be sure you know what fragments and pixels are)

Changes in main fn, display to Z-buffer

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

In init function

```
glEnable(GL_DEPTH_TEST) ;
glDepthFunc(GL_LESS) ; // The default option
```

Demo

- Demo 2
- Does order of drawing matter any more?
- What if I change near plane to 0?
- Is this desirable? If not, what can I do about it?

Outline

- Review of demo from last lecture
 - Basic geometry setup for cubes (pillars), colors
 - Single geometric object, but multiple colors for pillars
 - Matrix Stacks and Transforms (draw 4 pillars)
 - Depth testing (Z-buffering)
 - *Animation (moving teapot)*
 - Texture Mapping (wooden floor)
- Best source for OpenGL is the red book and GLSL book. Of course, this is more a reference manual than a textbook, and you are better off implementing rather reading end to end.

Demo

- Demo 3
- Notice how teapot cycles around
- And that I can pause and restart animation
- And do everything else (zoom etc.) while teapot moves in background

Drawing Teapot (in display)

```
// ** NEW ** Put a teapot in the middle that animates
pushMatrix(modelview);
modelview = modelview * glm::translate(identity,
glm::vec3(teapotloc, 0.0, 0.0));
// The following two transforms set up and center the teapot
// Transforms right-multiply the modelview matrix (top of the stack)
modelview = modelview * glm::translate(identity, glm::vec3(0.0,
0.0, 0.1));
modelview = modelview * glm::rotate(identity, glm::pi<float>() /
2.0f, glm::vec3(1.0, 0.0, 0.0));
float size = 0.235f; // Teapot size
modelview = modelview * glm::scale(identity, glm::vec3(size, size,
size));
glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(modelview)[0][0]);
drawteapot();
popMatrix(modelview);

void drawteapot() { // drawteapot() function in geometry.h
glBindVertexArray(teapotVAO);
glDrawElements(GL_TRIANGLES, teapotIndices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
}
```

Simple Animation routine

```
// ** NEW ** in this assignment, is an animation of a teapot
// Hitting p will pause this animation; see keyboard callback

void animation(void) {
teapotloc = teapotloc + 0.005;
if (teapotloc > 0.5) teapotloc = -0.5;
glutPostRedisplay();
}
```

Keyboard callback (p to pause)

```
GLint animate = 0; // ** NEW ** whether to animate or not

void keyboard (unsigned char key, int x, int y)
{
switch (key) {
case 27: // Escape to quit
exit(0);
break;
case 'p': // ** NEW ** to pause/restart animation
animate = !animate;
if (animate) glutIdleFunc(animation);
else glutIdleFunc(NULL);
break;
default:
break;
}
}
```

Outline

- Review of demo from last lecture
- Display lists (extend init for pillars)
- Matrix stacks and transforms (draw 4 pillars)
- Depth testing or z-buffering
- Animation (moving teapot)
- *Texture mapping (wooden floor) [mytest3]*

New globals and basic setup

```
// In mytest3.cpp
GLubyte woodtexture[256][256][3]; // texture (from grsites.com)
GLuint texNames[1]; // texture buffer
GLuint istex; // blend parameter for texturing
GLuint islight; // for lighting
GLint texturing = 1; // to turn on/off texturing
GLint lighting = 1; // to turn on/off lighting

// In Display
glUniform1i(islight,0); // Turn off lighting (except on teapot, later)
glUniform1i(istex,texturing);
drawtexture(FLOOR,texNames[0]); // Texturing floor
// drawobject(FLOOR);

glUniform1i(istex,0); // Other items aren't textured
```

Simple Toggles for Keyboard

```
case 't': // ** NEW ** to turn on/off texturing;
texturing = !texturing;
glutPostRedisplay();
break;

case 's': // ** NEW ** to turn on/off shading (always smooth);
lighting = !lighting;
glutPostRedisplay();
break;
```

Adding Visual Detail

- Basic idea: use images instead of more polygons to represent fine scale color variation



Texture Mapping

- Important topic: nearly all objects textured
 - Wood grain, faces, bricks and so on
 - Adds visual detail to scenes
- Can be added in a fragment shader



Polygonal model



With surface texture

Setting up texture

```
initttexture("wood.ppm", shaderprogram) ; // in init()

// Very basic code to read a ppm file
// And then set up buffers for texture coordinates
void initttexture (const char * filename, GLuint program) {
    int i,j,k ;
    FILE * fp ;
    assert(fp = fopen(filename,"rb")) ;
    fscanf(fp,"%s %d %d %d %c") ;
    for (i = 0 ; i < 256 ; i++)
        for (j = 0 ; j < 256 ; j++)
            for (k = 0 ; k < 3 ; k++)
                fscanf(fp,"%c",&(woodtexture[i][j][k])) ;
    fclose(fp) ;
}
```

Texture Coordinates

- Each vertex must have a texture coordinate: pointer to texture. Interpolate for pixels (each fragment has st)

```
// Set up Texture Coordinates
glGenTextures(1, texNames) ;
glBindVertexArray(VAOs[FLOOR]) ;
glBindBuffer(GL_ARRAY_BUFFER, buffers[numobjects*numberobj+ncolors]) ;
glBufferData(GL_ARRAY_BUFFER, sizeof (floortex),
             floortex, GL_STATIC_DRAW) ;

// Use layout location 2 for texcoords
glEnableVertexAttribArray(2) ;
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), 0) ;

glActiveTexture(GL_TEXTURE0) ;
glEnable(GL_TEXTURE_2D) ;

glBindTexture (GL_TEXTURE_2D, texNames[0]) ;
```

Specifying the Texture Image

- `glTexImage2D( target, level, components, width height, border, format, type, data )`
- target is `GL_TEXTURE_2D`
- level is (almost always) 0
- components = 3 or 4 (RGB/RGBA)
- width/height MUST be a power of 2
- border = 0 (usually)
- format = `GL_RGB` or `GL_RGBA` (usually)
- type = `GL_UNSIGNED_BYTE`, `GL_FLOAT`, etc...

Texture Image and Bind to Shader

```
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB, 256, 256, 0, GL_RGB,
             GL_UNSIGNED_BYTE, woodtexture) ;
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
               GL_LINEAR) ;
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
               GL_LINEAR) ;
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT) ;
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT) ;

// Define a sampler. See page 709 in red book, 7th ed.
GLint texsampler ;
texsampler = glGetUniformLocation(program, "tex") ;
glUniform1i(texsampler,0) ; // Could also be GL_TEXTURE0
istex = glGetUniformLocation(program,"istex") ;
```

Drawing with Texture

```
// And a function to draw with textures, similar to drawobject
void drawTexture(GLuint object, GLuint texture) {
    glBindTexture(GL_TEXTURE_2D, texture);
    glBindVertexArray(VAOs[object]);
    glDrawElements(PrimType[object], NumElems[object],
        GL_UNSIGNED_BYTE, 0);
    glBindVertexArray(0);
}
```

Final Steps for Drawing (+Demo)

- Vertex shader (just pass on texture coords)

```
layout (location = 2) in vec2 texCoords;
out vec2 texcoord; // similar definitions for positions and normals
uniform int istex ;
void main() {
    gl_Position = projection * modelview * vec4(position, 1.0f);
    mynormal = mat3(transpose(inverse(modelview))) * normal ;
    myvertex = modelview * vec4(position, 1.0f) ;
    texcoord = vec2 (0.0, 0.0); // Default value just to prevent errors
    if (istex != 0){ texcoord = texCoords; }
```

- Fragment shader (can be more complex blend)

```
uniform sampler2D tex ;
uniform int istex ;
void main (void) {
    if (istex > 0) fragColor = texture(tex, texcoord) ;
}
```

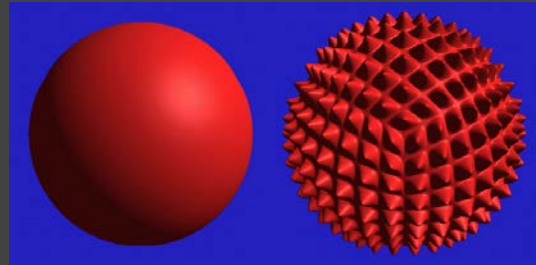
More on Texture (very briefly)

Full lecture later in course

- Optimizations for efficiency
- Mipmapping
- Filtering
- Texture Coordinate generation
- Texture Matrix
- Environment Mapping

If very ambitious, read more in OpenGL

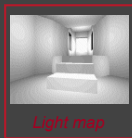
Displacement Mapping



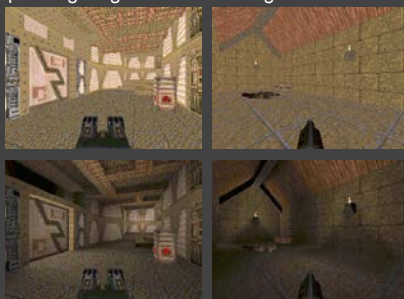
Illumination Maps

- Quake introduced *illumination maps* or *light maps* to capture lighting effects in video games

Texture map:



Texture map
+ light map:



Environment Maps



Images from *Illumination and Reflection Maps*:
Simulated Objects in Simulated and Real Environments
 Gene Miller and C. Robert Hoffman
 SIGGRAPH 1984 "Advanced Computer Graphics Animation" Course Notes

Solid textures

Texture values indexed
by 3D location (x,y,z)

- Expensive storage, or
- Compute on the fly,
e.g. Perlin noise →

