CS450/550

Programming With OpenGL
Part I

Adapted From: Angel and Shreiner: Interactive Computer Graphics6E © Addison-Wesley 2012
Objectives

Get you started ASAP!!!
OpenGL API Architecture
  OpenGL as a state machine
Functions
  Types
  Formats
Simple program
OpenGL State

OpenGL is a state machine
OpenGL functions are of two types

Primitive generating
- Can cause output if primitive is visible
- How vertices are processed and appearance of primitive are controlled by the state

State changing
- Transformation functions
- Attribute functions
- Under 3.1 most state variables are defined by the application and sent to the shaders [More on this later!]
## GL Data Types

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Typical Corresponding C Type</th>
<th>OpenGL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>signed char</td>
<td>GLbyte</td>
</tr>
<tr>
<td>s</td>
<td>short</td>
<td>GLshort</td>
</tr>
<tr>
<td>l</td>
<td>int or long</td>
<td>Glint, GLsizei</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>Glfloat</td>
</tr>
<tr>
<td>d</td>
<td>double</td>
<td>GLdouble</td>
</tr>
<tr>
<td>ub</td>
<td>unsigned char</td>
<td>GLubyte</td>
</tr>
<tr>
<td>us</td>
<td>unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>ui</td>
<td>unsigned int</td>
<td>GLuint</td>
</tr>
</tbody>
</table>

Implementations vary in mapping C types to OpenGL types. Use OGL types to avoid porting issues.
Lack of Object Orientation

OpenGL is not object oriented so that there are multiple functions for a given logical function

- `glUniform3f`
- `glUniform2i`
- `glUniform3dv`
OpenGL function format

\texttt{glUniform3f}(x, y, z)

function name

\texttt{glUniform3fv}(p)

\texttt{p} is a pointer to an array

x, y, z are floats

belongs to GL library

dimensions

OpenGL #defines

Most constants are defined in the include files `gl.h`, `glu.h` and `glut.h`

Note `#include <GL/glut.h>` should automatically include the others

Examples

```c
glEnable(GL_DEPTH_TEST)

glClearColor(GL_COLOR_BUFFER_BIT)
```

Include files also define OpenGL data types: `GLfloat`, `GLdouble`, ....
A Simple Program (?)

Generate a square on a solid background
It used to be easy

```c
#include <GL/glut.h>
void mydisplay(){
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_QUAD);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0,5, 0,5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
  glEnd()
}
int main(int argc, char** argv){
  glutCreateWindow("simple");
  glutDisplayFunc(mydisplay);
  glutMainLoop();
}
```

Where are:

- Lights
- Cameras
- Materials
- Etc.
main.c – removed defaults

```c
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    glewInit();
    init();
    glutMainLoop();
}
```

For extensions on PC
includes gl.h
includes gl.h
specify window properties
display callback
set OpenGL state and initialize shaders
enter event loop
Modern Version...
Program Structure

Most OpenGL programs have a similar structure that consists of the following functions

**main()**: specifies the callback functions
- opens one or more windows with the required properties
- enters event loop (last executable statement)

**init()**: creates geometry and sets the state variables
- Create and link buffers
- Viewing parameters
- Attributes
- Shader: read, compile, link

**display()**: redraws the scene
- clears buffer
- Sends draw request with instructions on how to interpret the buffer
Recall … Retained Mode Graphics

Put all vertex and attribute data in array

Send array to GPU to be rendered immediately – ie. the GPU does not save it

Almost OK, but problem is we would have to send array over each time we need another render of it

Better to send array over and store on GPU for multiple renderings
Display Callback

Once we get data to GPU, we can initiate the rendering with a simple callback

```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glFlush();
}
```

Buffer Objects on GPU that contain vertex arrays

**Vertex Arrays**

Vertices can have many attributes

- Position
- Color
- Texture Coordinates
- Application data

A vertex array holds these data (like a set of parallel arrays to hold attributes)

```c
point2 vertices[3] = {point2(0.0, 0.0),
        point2(0.0, 1.0),
        point2(1.0, 1.0)};

color3 colors[3] = {color3(1.0, 0.0, 0.0),
        color3(1.0, 0.0, 0.0),
        color3(1.0, 0.0, 0.0)};
```

Vertex Array Object (VAO)

Manages all info to render an object (positions, colors, ..,)

At this point we have a current vao but no contents
We’ll put Vertex Buffer Objects (VBO’s) into them!

Glunit abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
Vertex Buffer Object (VBO)

Allocate memory buffer ON the card
Copies data from app arrays to buffer on card

Gluint buffer;
 glGenBuffers(1, &buffer);
 glBindBuffer(GL_ARRAY_BUFFER, buffer);

BufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

Linking Arrays to Shaders

Shaders do the heavy lifting and need access to the arrays of attributes (vertices, colors, etc.)

We need a way to tie the shaders to the arrays that you’ve created in the buffer objects
Vertex Attributes in Shaders

Vertex attributes are named in the shaders [Much more on shaders later!]

```glsl
in vec4 vPosition;
in vec3 vColor;
void main(){
    gl_Position = vPosition;
    ...
}
```

Shader linker forms a table

<table>
<thead>
<tr>
<th>index/loc</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>vPosition</td>
</tr>
<tr>
<td>1</td>
<td>vColor</td>
</tr>
</tbody>
</table>

Application can get index from table and tie it to an application variable

Connecting Shader to Arrays

1. Enable the vertex attributes that are in the shaders
2. Describe the form of the data in the vertex array

```c
#define BUFFER_OFFSET( offset )
  ((GLvoid*) (offset))

GLuint loc = glGetUniformLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );

glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );  //2
```

- `loc`: attribute index in shader
- `2`: # of elements for that attribute
- `GL_FLOAT`: attribute type
- `0`: don’t normalize data [0-1]
- `0`: stride in the array
- `BUFFER_OFFSET(0)`: offset of first component in current VBO

VBOs are Flexible

(VVVV) (NNNN) (CCCC) (separate VBO for each attribute)
One VAO, create and bind separate VBO for v, n, c respectively

(VVVVNNNNCCCC) (packed into single VBO)
One VAO, create and bind a single VBO
Use a single array with all data packed contiguously
Use ‘offset’ when linking attributes to shader variables

(VNCVNCVNCVNC) (interleaved)
One VAO, create and bind a single VBO
Interleave the attributes
Use ‘stride’ and ‘byte’ offsets to tie to shader variables

glBufferSubData –alternative spec.

Fills, but does not allocate buffer (must be allocated with glBufferData)

```c
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices)+sizeof(colors), null, GL_STATIC_DRAW);
```

```c
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
```

```c
glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(colors), colors);
```
Example: Modern Simple Program

Single VBO, vertex position attribute
Multiple VBO, vertex position + color attributes
Single VBO, BufferSubData, vertex position + color attributes