CS450/550

Programming With OpenGL
Part I

Adapted From: Angel and Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012
Objectives

Get you started ASAP!!!
OpenGL API Architecture

OpenGL as a state machine
OpenGL State

OpenGL is a state machine: collection of variables that define how OpenGL currently operates.

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 memory layouts. OpenGL defines cross platform mappings for OGL types that are always the right size.
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

`glUniform3f(x, y, z)`

- **Function name**: `glUniform3f`
- **Dimensions**: `(x, y, z)` are floats
- **Belongs to GL library**

`glUniform3fv(p)`

- **p** is a pointer to an array
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)
```

```c
glClearColor(GL_COLOR_BUFFER_BIT)
```

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

Generate a square on a solid background
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 … Modern Retained Mode Graphics

Put all vertex and attribute data in array

Send array to GPU for storage

Request that the GPU render when needed
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)

```cpp
vec2 vertices[2] = (vec2(0.0, 0.0),
                   vec2( 0.0, 1.0),
                   vec2(1.0, 1.0));
vec3 colors[3] = {vec3(1.0, 0.0, 0.0),
                  vec3(1.0, 0.0, 0.0),
                  vec3(1.0, 0.0,0.0));
```
**Vertex Array Object (VAO)**

Manages all info to render an object (positions, colors, ..,) with a single call

```c
GLuint abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
```

At this point we have a **current vao** but no contents

We’ll associate the VAO with several Vertex Buffer Objects (VBO’s)
Vertex Buffer Object (VBO)

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

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

glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Make it currently active

int NumPoints = 6;

vec2 vertices[] = {
    vec2(-0.5, -0.5),
    vec2(-0.5, 0.5),
    vec2(0.5, 0.5),
    vec2(0.5, 0.5),
    vec2(0.5, -0.5),
    vec2(-0.5, -0.5),
};

GLuint vao[1];
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);  // bind to make it active.

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

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

GLuint loc = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

void display(void)
{
    // clear the window
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumPoints);
    glFlush();
    glutSwapBuffers();
}

#version 150

in vec2 vPosition;  // at table index 0

void main()
{
    gl_Position = vec4(vPosition, 0.0, 0.0);
}
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. Get pointer to the shader variable
2. Enable the vertex attributes that are in the shaders
3. Describe the form of the data in the vertex array

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

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

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

Replace my diagrams with new data flow diagrams!
int NumPoints = 6;

vec2 vertices[] = {
    vec2(-0.5, -0.5),
    vec2(-0.5, 0.5),
    vec2(0.5, 0.5),
    vec2(0.5, -0.5),
    vec2(-0.5, -0.5),
};

GLuint vao[1];
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);  //bind to make it active.

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

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

GLuint loc = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

void display(void)
{
    // clear the window
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumPoints);
    glFlush();
    glutSwapBuffers();
}

#version 150

in vec2 vPosition;  //at table index 0

void main()
{
    gl_Position = vec4(vPosition, 0.0, 0.0);
}
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, 6);
    glFlush();
}
```

Buffer Objects on GPU that contain vertex arrays
Aside on Primitives…

Discuss possible primitives here..
points, lines, triangles, etc.
why you have to tell drawArrays how to treat the verts!
OpenGL Primitives

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
int NumPoints = 6;
vec2 vertices[] = {
    vec2(-0.5, -0.5),
    vec2(-0.5, 0.5),
    vec2(0.5, 0.5),
    vec2(0.5, 0.5),
    vec2(0.5, -0.5),
    vec2(-0.5, -0.5),
};

GLuint vao[1];
genVertexArrays(1, vao);
BindVertexArray(vao[0]); //bind to make it active.

GLuint buffer;
GenBuffers(1, &buffer);
BindBuffer(GL_ARRAY_BUFFER, buffer);

BufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
GLuint loc = GetAttribLocation(program, "vPosition");
EnableVertexAttribArray(loc);
VertexAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

void display(void)
{
    // clear the window
    Clear(GL_COLOR_BUFFER_BIT);
    DrawArrays(GL_TRIANGLES, 0, NumPoints);
    Flush();
    glutSwapBuffers();
}

#version 150

in vec2 vPosition;  //at table index 0

void main()
{
    gl_Position = vec4(vPosition, 0.0, 0.0);
}
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);

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

glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices), sizeof(colors), colors);
```

Target

Offset

#bytes to use

Data
Example: Modern Simple Program

Single VBO, vertex position attribute
Multiple VBO, vertex position+ color attributes
Single VBO, BufferSubData, vertex position + color attributes
**Rules of Thumb**

- **VAO vs. VBO**
  - Use a separate VAO for each batch (thing, mesh, etc.) that you want to draw
  - Each of those “things” will have a set of VBOs for its attributes (pos, color, normals, etc.)
  - When you bind a VAO, it does all the proper binding of its VBOs and sets necessary state to draw the object

- **VBOs**
  - Best (for performance) to put several attributes into one VBO to reduce number of binds
  - If an geo object is static, definitely put into as few VBOs as possible
  - If some attribute is dynamic (say colors), then separate VBOs are preferred (most likely due to caching issues)
  - Interleaved layouts can be inefficient due to memory alignment
Rules of Thumb....

• WebGL does not have VAOs, so you have to manage the VBO’s yourself
• When using a single VBO to hold multiple attributes (positions, colors, normals, etc.)
  • Use the OFFSET in glVertexAttribPointer to properly link the attributes to variables in the shader
• If you want to draw subsets of a VBO in different ways, use the OFFSET and NUMPOINTS in glDrawArrays to select those subsets