Introduction to OpenGL

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Administrivia

- Assignment 1 due now
- Assignment 2 handed today
  - Upgrade your video card drivers [NVIDIA | ATI]
- Question: How many students are also registered for 563?

Agenda

- Review Monday’s GLSL material
- OpenGL
  - Shaders and uniforms
  - Vertex arrays and buffers
  - Multithreading
- Review Assignment 2

GLSL Review

- Rewrite with one if and one compare
  ```
  if (dist < wPrime)
  {
    if (dist < closestDistance)
    {
      closestDistance = dist;
    }
  }
  ```
GLSL Review

- Implement this concisely
  ```glsl
  bool PointInsideAxisAlignedBoundingBox(vec3 p, vec3 b0, vec3 b1)
  {
      // ...
  }
  ```

- Does your code also work for vec2?

GLSL Review

- What is the difference between a fixed function and programmable stage?
- Vertex shader
  - What is its input? Output?
- Fragment shader
  - What is its input? Output?
  - [true | false] Fragment shaders allow you to change the xy position
  - [true | false] A best practice is to roll your own functions instead of calling library functions
  - In general, build vs buy

OpenGL

- Is a C-based API
- Is cross platform
- Is run by the ARB: Architecture Review Board
- Hides the device driver details
- OpenGL vs Direct3D
  - Not going there – at least not on record

We are using GL 3.3 core profile

- No fixed function vertex and fragment shading
- No legacy API calls:
  - `glBegin()`
  - `glRotatef()`
  - `glTexEnvf()`
  - `AlphaFunc()`
  - ...

Recall the fixed function light map

Why was the alpha test remove?
OpenGL

Software stack:
- Application
- OpenGL API
- Device Driver
- GPU

Major objects:
- Framebuffers
- Vertex Arrays
- Textures
- Vertex Buffers
- Samplers
- Index Buffers
- Shader Programs
- Pixel Buffers
- Fixed Function State

We are not covering everything. Just surveying the most relevant parts for writing GLSL shaders.

Shaders

- **Shader object**: an individual vertex, fragment, etc. shader
  - Are provided shader source code as a string
  - Are compiled
- **Shader program**: Multiple shader objects linked together

Shader Objects

Compile a shader object:
```c
const char *source = "...";
GLuint sourceLength = ...
GLuint v = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(v, 1, &source, &sourceLength);
glCompileShader(v);
GLint compiled;
glGetShaderiv(v, GL_COMPILE_STATUS, &compiled);
if (compiled == GL_TRUE) {
    // success
}
glDeleteShader(v);
```
Shader Objects

Compile a shader object:

```c
const char *source = // ...
GLint sourceLength = // ...
GLuint v = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(v, 1, &source, &sourceLength);
glCompileShader(v);
GLint compiled;
glGetShaderiv(v, GL_COMPILE_STATUS, &compiled);
// success: compiled == GL_TRUE
// ...
glDeleteShader(v);
```

OpenGL functions start with `gl`. Why? How would you design this in C++?

Compile, but what does the driver really do?

Good developers check for error. Again, how would you design this in C++?

Calling `glGet*` has performance implications. Why?

Provide the shader's source code

Where should the source come from?

Why can we pass more than one string?
Shader Objects

Compile a shader object:

```c
const char *source = // ...;
GLint sourceLength = // ...;
GLuint v = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(v, 1, &source, &sourceLength);
glCompileShader(v);
GLint compiled;
glGetShaderiv(v, GL_COMPILE_STATUS, &compiled);
// success: compiled == GL_TRUE
// ...
glDeleteShader(v);
```

Link a shader program:

```c
GLuint p = glCreateProgram();
glAttachShader(p, v);
glinkProgram(p);
GLint linked;
gGetShaderiv(p, GL_LINK_STATUS, &linked);
// success: linked == GL_TRUE
// ...
gDeleteProgram(p);
```

Shader Programs

Compile a shader object:

```c
const char *source = // ...;
GLint sourceLength = // ...;
GLuint v = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(v, 1, &source, &sourceLength);
glCompileShader(v);
GLint compiled;
glGetShaderiv(v, GL_COMPILE_STATUS, &compiled);
// success: compiled == GL_TRUE
// ...
gDeleteShader(v);
```

Link a shader program:

```c
GLuint p = glCreateProgram();
glAttachShader(p, v);
glinkProgram(p);
GLint linked;
gGetShaderiv(p, GL_LINK_STATUS, &linked);
// success: linked == GL_TRUE
// ...
gDeleteProgram(p);
```

This process is just like compiling an OpenCL kernel. We will see later this semester.

A program needs at least a vertex and fragment shader.
Shader Programs

- Link a shader program:
  ```
  GLuint v = glCreateShader(GL_VERTEX_SHADER);
  GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
  // ...
  GLuint p = glCreateProgram();
  glAttachShader(p, v);
  glAttachShader(p, f);
  glLinkProgram(p);
  GLint linked;
  glGetShaderiv(p, GL_LINK_STATUS, &linked);
  // success: linked == GL_TRUE
  // ...
  glDeleteProgram(p);
  ```

Using Shader Programs

```
GLuint p = glCreateProgram();
// ...
// Part of the current state
// - How do you draw different objects with different shaders?
// - What is the cost of using multiple shaders?
// - How do you reduce the cost?
//   - Hint: write more CPU code — really.
```

Be a good developer again
Uniforms

```c
GLuint p = glCreateProgram();
// ...
glinkProgram(p);

GLuint m = glGetUniformLocation(p, "u_modelViewMatrix");
GLuint l = glGetUniformLocation(p, "u_lightMap");
glinkProgram();
mat4 matrix = // ...
gUniformMatrix4fv(m, 1, GL_FALSE, &matrix[0][0]);
gUniform1i(l, 0);
```

Each active uniform has an integer index location.

Not transposing the matrix

Uniforms can be changed as often as needed, but are constant during a draw call

mat4 is part of the C++ GLM library

GLM:  http://www.g-truc.net/project-0016.html#menu
Uniforms

```c
GLuint p = glCreateProgram();
// ...
glLinkProgram(p);
GLuint m = glGetUniformLocation(p, "u_modelViewMatrix");
GLuint l = glGetUniformLocation(p, "u_lightMap");

// Why not glUniform*(p, ...)?

glUniformMatrix4fv(m, 1, GL_FALSE, &matrix[0][0]);
glUniform1i(l, 0);
```

Drawing

- How do we transfer vertices from system memory to video memory?
- How do we issue `draw` calls?

It doesn’t matter if we’re using:

- OpenGL
- NVIDIA CUDA
- OpenCL

Efficiently transferring data between the CPU and GPU is critical for performance.

- Typical pre-Nahalem Intel System
- Separate system and video memory
- Need to transfer vertices from one to the other quickly


- 4 GB/s reads and writes
- Theoretical 128M 32 byte vertices/second
How good is 128M vertices/second?

Boeing 777 model: ~350 million polygons

Image from http://graphics.cs.uni-sb.de/MassiveRT/boeing777.html

Procedurally generated model of Pompeii: ~1.4 billion polygons

Image from http://www.vision.ee.ethz.ch/~pmueller/wiki/CityEngine/Documents

OpenGL has evolved since 1992 (GL 1.0)
- Immediate mode
- Display lists
- Client-side vertex arrays
- Vertex buffer objects (VBOs)

OpenGL

```
GLfloat v0[3] = { 0.0f, 0.0f, 0.0f };
// ...
```

Immediate Mode

```
GLenum g0 = (GL_TRIANGLES);
gVertex3f(v0);
gVertex3f(v1);
gVertex3f(v2);
gVertex3f(v3);
gVertex3f(v4);
gVertex3f(v5);
gEnd();
```

Pro: really simple

What’s the con?
OpenGL commands between `glNewList` and `glEndList` are not executed immediately. Instead, they are compiled into the display list.

Create one display list, just like `glCreateShader` creates a shader.

A single function call executes the display list. You can execute the same display list many times.

**Pros**
- Little function call overhead
- Optimized compiling: stored in video memory, perhaps vertex cache optimized, etc.

**Cons**
- Compiling is slow. How do you support dynamic data?
- Usability: what is compiled into a display list and what isn’t?
Drawing: Display Lists

```c
GLuint dl = glGenLists(1);  
glNewList(dl, GL_COMPILE); 
// ... 
glBegin(GL_TRIANGLES); 
// ... 
// You guys are good developers
```

Drawing: Client-side Vertex Arrays

Point GL to an array in system memory

```c
GLfloat vertices[] = {...}; // 2 triangles = 6 vertices = 18 floats 
glEnableClientState(GL_VERTEX_ARRAY); 
glVertexPointer(3, GL_FLOAT, 0, vertices); 
glDrawArrays(GL_TRIANGLES, 0, 18); 
glDisableClientState(GL_VERTEX_ARRAY); 
```

Store vertices in an array

```c
GLfloat vertices[] = {...}; // 2 triangles = 6 vertices = 18 floats 
glEnableClientState(GL_VERTEX_ARRAY); 
glVertexPointer(3, GL_FLOAT, 0, vertices); 
glDrawArrays(GL_TRIANGLES, 0, 18); 
glDisableClientState(GL_VERTEX_ARRAY); 
```

Ugh, tell GL we have vertices (positions, actually)

*Managing global state is painful*
Drawing: Client-side Vertex Arrays

```c
GLfloat vertices[] = {...}; // 2 triangles = 6 vertices = 18 floats

// Pointer to our vertices

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

Draw in a single GL call

| Pro: little function call overhead |
| Con: bus traffic |
```

Stride, in bytes, between vertices. 0 means tightly packed.

Pro: little function call overhead
Con: bus traffic
Drawing: Vertex Buffer Objects

- **VBO**: Vertex Buffer Object

  - Like client-side vertex arrays, but:
    - Stored in driver-controlled memory, not an array in your application
    - Provide hints to the driver about how you will use the buffer

  - VBOs are the only way to store vertices in GL 3.3 core profile. The others are deprecated.

  We can use textures, but let’s not jump ahead.

```cpp
gluInt vbo;
gluFloat* vertices = new gluFloat[3 * numberOfVertices];
gluNbuffer(GL_ARRAY_BUFFER_ARB, vbo);
gluBuffer(vbo, numberObytes, vertices, GL_STATIC_DRAW_ARB);
// Also check out gluBufferSubData
delete [] vertices;
gluDeleteBuffers(1, vbo);
```
Drawing: Vertex Buffer Objects

GLuint vbo;
GLfloat* vertices = new GLfloat[3 * numberOfVertices];
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER_ARB, vbo);
glBufferData(GL_ARRAY_BUFFER_ARB, numberOfBytes, vertices, GL_STATIC_DRAW_ARB);
// Also check out glBufferSubData
delete [] vertices;
glDeleteBuffers(1, &vbo);

Copy from application to driver-controlled memory. GL_STATIC_DRAW should imply video memory.

Usage Hint
- Static: 1-to-n update-to-draw ratio
- Dynamic: n-to-m update to draw (n < m)
- Stream: 1-to-1 update to draw

It’s a hint. Do drivers take it into consideration?
Drawing: Vertex Buffer Objects

- In general:

  Say no to drugs too, please.

  Immediate Mode
  VBOs

Vertex Array Objects

- VBOs are just buffers
  - Raw bytes
  - **VAOs:** Vertex Array Objects
    - Interpret VBOs as actual vertices
    - Used when issuing `glDraw*`
    - You are not responsible for the implementation details

VBO Layouts

<table>
<thead>
<tr>
<th>Separate Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>positionBuffer:</td>
</tr>
<tr>
<td>normalBuffer:</td>
</tr>
<tr>
<td>textureCoordinateBuffer:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-interleaved Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>noninterleavedBuffer:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interleaved Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleavedBuffer:</td>
</tr>
</tbody>
</table>

VBO Layouts: Tradeoffs

- Separate Buffers
  - Flexibility, e.g.:
    - Combination of static and dynamic buffers
    - Multiple objects share the same buffer

- Non-interleaved Buffer
  - How is the memory coherence?

- Interleaved Buffer
  - Faster for static buffers
  - Proportional to the number of attributes

- Hybrid?
Vertex Throughput: VBO Layouts

- Figure 3.2: Batches with different number of vertex and texture coordinates. 64k triangles per batch and n 4-float texture coordinates.

Making lots of `glDraw*` calls is slow. Why?
Vertex Throughput Tips

- Optimize for the *Vertex Caches*
- Use smaller vertices
  - Use less precision, e.g., `half` instead of `float`
  - Compress, then decompress in vertex shader
  - Pack, then unpack in vertex shader
  - Derive attributes or components from other attributes
  - How many components do you need to store a normal?

Know your architecture!

---

**Table 3.3:** Impact of precision of vertex positions on performance, comparing `32-bit` and `16-bit` precision, with and without normal baking. Dataset: Rappy Buddha (134 triangles) and Fiji (116 triangles), with 144 triangles per batch.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Precision</th>
<th>Normal Baked</th>
<th>Triangle Count</th>
<th>GPU Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fiji</td>
<td>32-bit</td>
<td>No</td>
<td>144</td>
<td>Fast</td>
</tr>
<tr>
<td>Rappy Buddha</td>
<td>32-bit</td>
<td>No</td>
<td>144</td>
<td>Fast</td>
</tr>
<tr>
<td>Fiji</td>
<td>16-bit</td>
<td>No</td>
<td>144</td>
<td>Slow</td>
</tr>
<tr>
<td>Rappy Buddha</td>
<td>16-bit</td>
<td>No</td>
<td>144</td>
<td>Slow</td>
</tr>
<tr>
<td>Rappy Buddha</td>
<td>32-bit</td>
<td>Yes</td>
<td>144</td>
<td>Fast</td>
</tr>
<tr>
<td>Fiji</td>
<td>32-bit</td>
<td>Yes</td>
<td>144</td>
<td>Fast</td>
</tr>
</tbody>
</table>

**GL_SHORT**
- Faster on NVIDIA...
- Slower on ATI
Vertex Throughput Tips

- Know your architecture!

> GL_SHORT normals faster than GL_FLOAT on NVIDIA
  - But not ATI
  - Still true today?

GL_BYTE normals use less memory than GL_SHORT or GL_FLOAT but are slower
  - Why?
  - Still true today?

Table 3.4: Impact of precision of vertex attributes on performance with our glass benchmark, rendering 1.595 triangles in a single VBO.

<table>
<thead>
<tr>
<th>Precision</th>
<th>CPU</th>
<th>GPU</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_FLOAT</td>
<td>239</td>
<td>243</td>
<td>240</td>
</tr>
<tr>
<td>GL_SHORT</td>
<td>247</td>
<td>258</td>
<td>251</td>
</tr>
<tr>
<td>GL_BYTE</td>
<td>252</td>
<td>265</td>
<td>258</td>
</tr>
<tr>
<td>Other</td>
<td>255</td>
<td>268</td>
<td>262</td>
</tr>
</tbody>
</table>

Do you believe me yet?
Multithreaded Rendering

- Quake 4 CPU usage
  - 41% - driver
  - 49% - engine
- Split render work into two threads:

Multithreaded OpenGL Drivers

- Driver CPU overhead is moved to a separate core
- Application remains unchanged
- What happens when you call glGet*?

Multithreaded Rendering

- Tradeoffs
  - Throughput vs latency
  - Memory usage – double buffering
    - Cache pollution
  - Synchronization
  - Single core machines
    - DOOM III era

Multithreaded OpenGL Drivers

- Driver CPU overhead is moved to a separate core
- Application remains unchanged
- What happens when you call glGet*?

Not Covered Today

- Textures
- Framebuffers
- State management
- …

Useful for GPGPU – and graphics, obviously
### Class Poll
- Multithreaded graphics engine design class?
- More graphics-related classes?
- Itching to get to GPGPU and GPU computing?

### OpenGL Resources
- **OpenGL/GLSL Quick Reference Card**
- **OpenGL Spec**
- **OpenGL Forums**
  - [http://www.opengl.org/discussion_boards/](http://www.opengl.org/discussion_boards/)