Lecture 13:
OpenGL Shading Language (GLSL)
Motivation

- Last week, we discussed the many of the new “tricks” in Graphics require low-level access to the Graphics pipeline.

- For example, using textures, which is a pain in regular OpenGL.
Graphics History

- **Software Rendering**
  - Old-fashioned way (Use ray casting)
- **Fixed-Function**
  - Immediate Mode (What we have used)
- **Programmable Pipeline (~2001)**
  - GL Shading Language (What we will learn)
- **CUDA/OpenCL (GPGPU) (~2008)**
  - General Purpose GPU Programming
Ray Casting (Software Rendering)

- Wolfenstein
- Quake 1
Fixed-Function

- Quake 3
- Half-Life
Programmable Pipeline (Shaders)

- Doom 3
- Far Cry
GPGPU

- General Purpose Graphics Processing Unit
- Utilize graphics card for computation
  - Do work other than graphics
Old Fixed-Function
Programmable Pipeline
Why GPU?

- Having two (CPU and GPU) is better than one...

- GPU is dedicated to floating point operations (a traditionally expensive operation on a CPU)
  - It is especially optimized for linear algebra (e.g. matrix manipulation)

- GPU is multi-core (multi-processor)
  - Imagine that you have one processor per pixel, how would you program what the processor should do?
ASUS ROG GeForce GTX 1080 STRIX-GTX1080-A8G-GAMING 8GB 256-Bit GDDR5X PCI Express 3.0 HDCP Ready Video Card

Cooler: Triple Fans
- Single Fan
- Triple Fans

Option: STRIX-GTX1080-A8G-GAMING
- STRIX-GTX1080-A8G-GAMING
- TURBO-GTX1080-8G

Boost Clock: 1835 MHz
- 1733 MHz
- 1835 MHz

- 8GB 256-Bit GDDR5X
- Core Clock OC mode: 1695 MHz
- Gaming mode: 1670 MHz
- Boost Clock OC mode: 1835 MHz
- Gaming mode: 1809 MHz
- *Retail goods are with default Gaming Mode, OC Mode can be adjusted with one click on GPU Tweak II
- 1 x Native DVI-D 2 x Native HDMI 2.0 2 x Native DisplayPort 1.4
- 2560 CUDA Cores
- PCI Express 3.0

8 New from $799.99  1 Used from $769.99
Floating Point Operations (FLOPs)

Theoretical Peak Floating Point Operations per Clock Cycle, Double Precision

- **HD 3870**
- **HD 4870**
- **HD 5870**
- **HD 6970**
- **HD 6970**
- **HD 7970 GHz E6**
- **Tesla K20**
- **Tesla K20X**
- **FirePro W9100**
- **FirePro S9150**
- **Tesla K40**
- **Tesla K40**
- **Xeon Phi 7120 (KNC)**
- **Xeon Phi 7290 (KNL)**

**INTEL Xeon CPUs**
- X5482
- X5492
- W5590
- X5690
- X5680
- E5-2690
- E5-2697 v2

**NVIDIA Tesla GPUs**
- Tesla C1060
- Tesla C1060
- Tesla M2090
- Tesla C2050

**AMD Radeon GPUs**
- HD 3870
- HD 4870
- HD 5870
- HD 6970
- HD 6970
- HD 7970 GHz E6
- Tesla K20
- Tesla K20X
- FirePro W9100
- FirePro S9150
- Tesla K40
- Tesla K40
- Xeon Phi 7120 (KNC)
- Xeon Phi 7290 (KNL)

**INTEL Xeon Phis**
- E5-2699 v3
- E5-2699 v3
- E5-2699 v4

**End of Year**
- 2008
- 2010
- 2012
- 2014
- 2016
Accessing Programmable Pipeline

Introducing Shaders!
CPU + GPU

- CPU: Typical C++ code
- GPU: OpenGL Shading Language (GLSL)

- CPU passes data to GPU

Some processing happens on the CPU:
- Get mouse movement
- Load and parse data from disk

Some processing happens on the GPU:
- Transform point from Object to World space
- Solve lighting equation
- Render
CPU + GPU (Continued)

- Think of this as two separate processors. What we’ll learn today are:
  1. What the GPU should do (and what the CPU should do).
     - In particular different “shaders” to fit into the different parts of the graphics pipeline
  2. The “language” used by the GPU (i.e. OpenGL Shading Language, or OpenGL SL, or GLSL).
  3. How to communicate between CPU and GPU
     - Sharing “chunks of memory”
     - Passing variable names and values
Types of Shaders

- Most common:
  - Vertex
  - Fragment (Pixel)

- Newer:
  - Geometry
  - Tessellation
  - Compute
Two Things to Learn:

- OpenGL Shading Language (GLSL)
  - Syntax
  - Sequence and processes
    - Vertex Shader
    - Pixel Shader

- Communication between CPU and GPU
  - What type of information can be passed
  - How to pass information

- Note that you also need to know how to load a shader program, but that’s “magic incantation” that (mostly) stays the same every time.
Interfacing with the GPU

- Modern OpenGL
- Mobile/Web Programming
- Direct3D
- Compute Languages
  - Stream, OpenCL, CUDA
- Future
Client/Server Architecture

- Think of like a network
- Client = the CPU
- Server = the GPU
- The less they talk, the better
  - Less waiting for memory to transfer
  - Bandwidth is almost always the bottleneck
- Send data over once, and have it ready to go
Immediate Mode (Fixed-Function pipeline)

- Every ‘gl’ command is being sent to the GPU every iteration.
- Very inefficient
- Think of it like having an interpreter versus a compiler for C++

Michael Shah
Fixed Function (Left) vs. Programmable (Right)
Optimizations Available

- Vertex Array
- Display Lists
- Vertex Buffer Objects
- Pixel Buffers
- Frame Buffer Objects
Vertex Array

- No more glBegin/glEnd
- Store Vertices, indices, normals, colors, texture coordinates in an array
- Minimizes OpenGL calls on Client (CPU)
- Data still being passed from Client to Server every frame
  - A little better because it’s all at once
  - Stylistically, much cleaner code as well

Michael Shah
Display List

- Compiled geometry
- Send it to the Server (GPU) once, and it is stored in memory on Server
- Geometry cannot be updated, without deleting display list.

```c
someGeometryDisplayList = glGenLists(1);
glNewList(someGeometryDisplayList,GL_COMPILE);
    drawShape();
glEndList();
```

Michael Shah
Vertex Buffer Object (VBO)

- Improves upon Vertex Array
- We can store data on the Server (GPU)
  - Just like display lists!
- We get a pointer into the Server (GPU) so that we can update it!
  - As good as display lists!
  - Because we have a pointer to the data, multiple clients can access data.
    - Multiple instances of a program can access data
Modifiers to Variables Uniform/Varying/Attribute

- **Uniform** – Read only data passed from CPU into GPU (vertex/fragment shaders)
- **Attributes** – Input value which change every vertex.
  - Read only – Used only in vertex shader
- **Varying** – Variables passed from one shader to another

Depending on GLSL version:
- **GLSL <3.0: Varying** – Pass data from vertex to fragment shader.
- **GLSL >=3.0: in / out** – in is the variable being passed in, out is the variable being passed out
  - Read-only in fragment shader.
  - Read/Write in Vertex Shader
Order matters... Info is only passed in one direction (starting with vertex shader)

- Note that we can run this loop multiple times (e.g. for shadow, or achieve specific shading effects (such as toon-shading))
OpenGL Shading Language
GLSL

- C-Like Language
  - Made for GPU Architecture
    - Because GPU’s are different than CPU’s
  - Still Cross Platform
- Debugging – Still as difficult
- Domain Specific Language
  - *Compiled at runtime!!!!!
GLSL Primitives

- float, int, bool

Example:

```c
float f = 10.0;
```

or

```c
float f = float(10);
```

(Casting not allowed, choose your type wisely!)
GLSL Types

- **Vector: vec2, vec3, vec4**
  - Overloaded to take in each type
  - Example: `vec4 temp = vec4(1.0, 1.0, 1.0, 1.0)`

- **Matrix: mat2, mat3, mat4**
  - Example: `glModelViewMatrix` is a mat4 type
  - Access first column with: `mat[0]`
  - Access second column with: `mat[1]`
    - Example: `mat[1] = vec3(1.0, 1.0, 1.0);`
  - Access individual element: `mat[2][1]`;
Some more examples

- **Pass in vec3 into a vec4**
  
  \[
  \begin{align*}
  \text{vec3 color} &= \text{vec3}(1.0, 0.5, 0.25) \\
  \text{vec4 temp} &= \text{vec4}(\text{color}, 1.0)
  \end{align*}
  \]

- **Get components of vector**
  
  \[
  \begin{align*}
  \text{color.x} &= 1.0; \text{ or } \text{color}[0] = 1.0 \\
  \text{color.y} &= 1.0; \text{ or } \text{color}[1] = 1.0;
  \end{align*}
  \]
Swizzling [1]

- Swizzling is the ability to rearrange a vector's components
- `vec2 pos = temp.xy; // retrieve x and y component, or any of xyzw`
- `vec2 pos = temp.yz;`
- `vec4 = color.agbr; // opposite of rgba`
More GLSL Keywords [1][2]

**Control:** if, else, do, for, while, break, continue, return, discard

**Spec:** const, attribute, uniform, varying

**Type:** float, int, bool, true, false, void, lowp, mediump, highp, precision, invariant

\{vec2, vec3, vec4\}, \{ivec2, ivec3, ivec4\}, \{bvec2, bvec3, bvec4\}, \{mat2, mat3, mat4\}

**Structure:** struct, sampler1D, sampler2D, samplerCube

**Build-in:**

- **Attributes:** gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord[#]
- **Uniforms:** gl_ModelViewMatrix, gl_ModelViewProjectionMatrix, gl_NormalMatrix
- **Varyings:** gl_FragColor, gl_BackColor, gl_TexCoord[#]

- Note that some of these default keywords might not be there depending on GLSL version number (for example: gl_ModelViewMatrix isn’t there any more)
Write a Simple Shader

Basic Vertex and Fragment Shader
Writing a Shader

- Writing a shader, compiling a shader, and loading a shader
- Compile at runtime
  - So we do not need to recompile entire source, if we only modify shader.
Simple Vertex Shader

```glsl
#version 120

void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Simple Fragment Shader

```
void main()
{
  vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
  gl_FragColor = color;
}
```
The cpp file

```cpp
{
    string source;
    loadFile(vertexShaderName, source);
    vertexShaderID = loadShader(source, GL_VERTEX_SHADER);

    source="";
    loadFile(fragmentShaderName, source);
    fragmentShaderID = loadShader(source, GL_FRAGMENT_SHADER);

    program = glCreateProgram();
    glAttachShader(program, vertexShaderID);
    glAttachShader(program, fragmentShaderID);

    glLinkProgram(program);

    glUseProgram(program);
}
```
Drawing with immediate mode for now

- Ideally we’ll use a Vertex Buffer Object (VBO). But to keep it simple in this example:

```cpp
if (filled) {
    glEnable(GL_POLYGON_OFFSET_FILL);
    glColor3f(0.5, 0.5, 0.5);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    //mySolarSystem->render();
    drawTriangle();
}
```
Final Image
Communications

Passing Vertex Buffer Objects from CPU -> Shaders
Passing variables from CPU -> Shaders
Passing variables between Shaders
Modifiers to Variables Uniform/Varying/Attribute

- **Uniform** – Read only data passed from CPU into GPU (vertex/fragment shaders)
- **Attributes** – Input value which change every vertex.
  - Read only – Used only in vertex shader
- **Varying** – Variables passed from one shader to another

Depending on GLSL version:
- **GLSL <3.0**: **Varying** – Pass data from vertex to fragment shader.
- **GLSL >=3.0**: **in / out** – in is the variable being passed in, out is the variable being passed out
  - Read-only in fragment shader.
  - Read/Write in Vertex Shader
Communications (Attrib)

https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL)

```c
/* We're going to create a simple diamond made from lines */
const GLfloat diamond[4][2] = {
  { 0.0, 1.0 }, /* Top point */
  { 1.0, 0.0 }, /* Right point */
  { 0.0, -1.0 }, /* Bottom point */
  { -1.0, 0.0 } }; /* Left point */

const GLfloat colors[4][3] = {
  { 1.0, 0.0, 0.0 }, /* Red */
  { 0.0, 1.0, 0.0 }, /* Green */
  { 0.0, 0.0, 1.0 }, /* Blue */
  { 1.0, 1.0, 1.0 } }; /* White */

/* Allocate and assign two Vertex Buffer Objects to our handle */
glGenBuffers(2, vbo);

/* Bind our first VBO as being the active buffer and storing vertex attributes (coordinates) */
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);

/* Copy the vertex data from diamond to our buffer */
/* 8 * sizeof(GLfloat) is the size of the diamond array, since it contains 8 GLfloat values */
glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(GLfloat), diamond, GL_STATIC_DRAW);

/* Specify that our coordinate data is going into attribute index 0, and contains two floats per vertex */
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);

/* Enable attribute index 0 as being used */
glEnableVertexAttribArray(0);
```
Communications (Attrib)

/* Bind our second VBO as being the active buffer and storing vertex attributes (colors) */
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);

/* Copy the color data from colors to our buffer */
/* 12 * sizeof(GLfloat) is the size of the colors array, since it contains 12 GLfloat values */
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(GLfloat), colors, GL_STATIC_DRAW);

/* Specify that our color data is going into attribute index 1, and contains three floats per vertex */
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

/* Enable attribute index 1 as being used */
glEnableVertexAttribArray(1);

/* Attach our shaders to our program */
glAttachShader(shaderprogram, vertexshader);
glAttachShader(shaderprogram, fragmentshader);

/* Bind attribute index 0 (coordinates) to in_Position and attribute index 1 (color) to in_Color */
/* Attribute locations must be setup before calling glLinkProgram. */
glBindAttribLocation(shaderprogram, 0, "in_Position");
glBindAttribLocation(shaderprogram, 1, "in_Color");

/* Link our program */
/* At this stage, the vertex and fragment programs are inspected, optimized and a binary code is generated for the shader. */
/* The binary code is uploaded to the GPU, if there is no error. */
glLinkProgram(shaderprogram);
**Vertex Shader**

```glsl
#version 150

// in_Position was bound to attribute index 0 and in_Color was bound to attribute index 1
in vec2 in_Position;
in vec3 in_Color;

// We output the ex_Color variable to the next shader in the chain
out vec3 ex_Color;

void main(void) {
    // Since we are using flat lines, our input only had two points: x and y.
    // Set the Z coordinate to 0 and W coordinate to 1
    gl_Position = vec4(in_Position.x, in_Position.y, 0.0, 1.0);

    // GLSL allows shorthand use of vectors too, the following is also valid:
    // gl_Position = vec4(in_Position, 0.0, 1.0);
    // We're simply passing the color through unmodified
    ex_Color = in_Color;
}
```

**Fragment Shader**

```glsl
#version 150

// It was expressed that some drivers required this next line to function properly
precision highp float;

in vec3 ex_Color;
out vec4 gl_FragColor;

void main(void) {
    // Pass through our original color with full opacity.
    gl_FragColor = vec4(ex_Color, 1.0);
}
```
Magic!! Vertex to Fragment Shader

- If you are wondering how a vertex shader passes data to a fragment shader...
  - Because vertex shaders operate at a “per vertex” level and fragments operate at a “per pixel” level

- What gets sent from one to another??

- The magic process is “blending”
  - Defaults to linear blend. Can be changed to different blend funcs
Communications (Uniform)

- Sometimes you want to pass just a single number (not per vertex (i.e. attribute) information via vertex buffer objects).
- These are called “uniform” variables. These can include:
  - Rendering information (e.g. projection matrix)
  - Rendering resource (e.g. a texture ID)
  - Control variable (e.g. renderMode)
  - .... In the vertex shader below, the value of “Scale”

```glsl
uniform float Scale;
void main(void)
{
  vec4 a = gl_Vertex;
  a.x = a.x * Scale;
  a.y = a.y * Scale;
  gl_Position = gl_ModelViewProjectionMatrix * a;
}
```
Communications (Uniform)

GLint loc = glGetUniformLocation(ProgramObject, "Scale");
if (loc != -1)
{
    glUniform1f(loc, 0.432);
}

- ProgramObject is the bound shader program
- glUniform has many forms:
  - glUniform1f, glUniform2f, ... 4f (float)
  - glUniform1i... 4i (int)
  - glUniform1ui ... 4ui (unsigned int)
  - glUniform1fv... 4fv (floating array)
  - glUniformMatrix2fv...4fv
Versions of OpenGL

Confusions galore...
Many versions of OpenGL

- OpenGL 2.0 (2004) introduces the idea of shaders
  - OpenGL SL 1.0 (or 1.1) is a part of OpenGL 2.0

- Currently, OpenGL is at 4.6 (2017). GLSL is also at 4.6
  - When OpenGL 3.1 was introduced, GLSL changed versioning to match (to 3.1), which was previous 1.3.1

- 3.0 -> 3.1 (2009) is said to be the “big move” where fixed pipeline is considered deprecated.
OpenGL ES

- OpenGL ES (ES = Embedded Systems) is meant for smart phones, devices, etc.
  - OpenGL ES 1.0 (2003) implemented OpenGL 1.3
  - OpenGL ES 2.0 (2007) is a reduced set of OpenGL 2.0
  - But ES 2.0 “jumped ahead” and eliminated almost all fixed-pipeline operations (e.g. glBegin, glEnd)

- OpenGL ES 2.0 is not backward compatible with previous versions

- Current version is ES 3.2 (2015)
WebGL

- WebGL is OpenGL for browsers, in particular HTML5 canvas element.

- WebGL 1.0 is based on OpenGL ES 2.0 (2011)

- Current version is WebGL 2.0 (2017)
  - based on OpenGL ES 3.0
  - Largely backward compatible
CUDA / OpenCL

- Made for general purpose graphics programming (GPGPU)
- Popular known use is bitcoin mining

- CUDA is a language specific to nVidia (2007)
  - Obviously different from GLSL in that there are no graphics related keywords or concepts. But the general idea (of parallel computing) is similar

- OpenCL is an open standard version (2009) of CUDA led by the Khronos Group
  - The Khronos Group is a non-profit consortium founded by major graphics companies
  - Currently Khronos is in charge of the OpenGL, OpenCL, OpenGL ES, WebGL, WebCL, and a bunch of related APIs.
Moving Forward in this Class...

- The C++ platform of GLUT only supports up to OpenGL 1.2 for Macs, but supports OpenGL 2.0 (or higher) in Windows.
  - As it stands, I am not aware of an “easy” C++ cross-platform environment that supports advanced OpenGL
  - “Complicated” platforms like Qt, wx, GTK exist, but the learning curve is high. One possible path forward is FLTK

- Side-stepping all the platform, hardware, GL version, issues, we switch to WebGL for the rest of the class
  - PROS: modern, latest OpenGL concepts. Easy to “show off”
  - CONS: uh, toss out all of your work in C++ so far and switch to Javascript