Shaders: Review

- Small programs that run on the GPU
  - “Shader” is an archaic misnomer
- Written in OpenGL Shading Language (GLSL)
- Shaders all have a main function that is run when the shader program is run
- No GLSL headers
Shader Execution Model

- Streaming input is usually things like vertex normals and fragment world positions
- Uniform input examples: model matrix, camera matrix, textures, time
- Output is to another part of the OpenGL pipeline, such as from a vertex shader to a fragment shader or a fragment shader to a frame buffer
Shader Execution Model

- Remember that multiple instances of the same shader program are executed on different portions of the streaming input.

- Each instance:
  - Reads from the same uniform input as the other shader instances.
  - Reads from a unique streaming input.
  - Writes its own output.
  - Executes independently of the other instances (cannot communicate with other instances).

- This is known as Single Instruction Multiple Thread execution, or SIMT.
  - Each instance runs the exact same code on its unique input.
The OpenGL Pipeline: Review
#version 150

uniform mat4 u_Model;
uniform mat4 u_View;
uniform mat4 u_Projection;

in vec3 vs_Pos;
out vec3 fs_Pos;

void main(){
    vec4 pos = u_Projection * u_View * u_Model * vec4(vs_Pos, 1.0f);
    fs_Pos = vec3(pos);
    gl_Position = pos;
}
Vertex Shader: Example

```glsl
#version 150
uniform mat4 u_Model;
uniform mat4 u_View;
uniform mat4 u_Projection;
in vec3 vs_Pos;
out vec3 fs_Pos;

void main(){
    vec4 pos = u_Projection * u_View * u_Model * vec4(vs_Pos, 1.0f);
    fs_Pos = vec3(pos);
    gl_Position = pos;  // This is the built-in GLSL vertex shader position output. It MUST be written to by the vertex shader.
}
```

The model, view, and projection matrices are the same for all vertices of a mesh, so we use uniform variables to send them to the shader.

The position of each vertex is unique, so we use an “in” variable to represent it.

We also want to pass the interpolated position of our vertex to the fragment shader, so we declare an “out” variable to store it.
#version 150

in vec4 fs_Pos;

out vec4 out_Color;

void main(){
    vec4 color = abs(sin(fs_Pos));
    color.w = 1;
    out_Color = color;
}

This fragment shader colors the geometry based on its position in screen space, since fs_Pos is passed from the vertex shader.

Note that the sine function and absolute value function automatically work on vectors!
GLSL

- Like C++ but without:
  - Pointers
  - Dynamic memory allocation
  - Recursion
  - User-defined classes
- Like C++ with:
  - Built-in linear algebra types (e.g. vector, matrix)
  - Built-in constructors
  - An excellent built-in math library
  - Input/output qualifiers
GLSL Syntax: Vectors

- Can store the following types: float, int, uint, bool
- Varying numbers of elements: vec2, vec3, vec4
  - Also ivec_, uvec_, bvec_ for ints, uints, and bools.
- Can access components with three different accessor categories
  - .x, .y, .z, .w (Position/direction)
  - .r, .g, .b, .a (Color)
  - .s, .t, .p, .q (Texture coordinate)
  - GLM also supports these accessors, so you can use them in your C++ code, too!
GLSL Syntax: Vectors

- Vector constructors:
  - Straightforward: `vec3 v = vec3(1, 2, 3);`
  - Extend: `vec3 v = vec3(vec2(1, 2), 3);`
  - Shorten: `vec3 v = vec3(vec4(1,2,3,4));`

- Vector swizzling:
  - `vec4 v = vec4(1, 2, 3, 4);`
  - `vec3 rgb = v.rgb; // <1, 2, 3>`
  - `vec3 bgr = v.bgr; // <3, 2, 1>`
  - `vec3 rrr = v.rrr; // <1, 1, 1>`
  - `v.r = 5; // <5, 2, 3, 4>`

- Can also access by index like an array
GLSL Syntax: Matrices

- Only store floating point numbers, single or double precision
- Square: mat2, mat3, mat4
- Rectangular: matMxN, where M and N are 2, 3, or 4.
- Stored column-major
  - Accessing index N will give you the Nth column of the matrix
  - Think of it as an array of column vectors
GLSL Syntax: Matrices

- Matrix constructors:
  - `mat3 i = mat3(1.0); // 3x3 identity matrix`
  - `mat3 s = mat3(2.0) // 3x3 matrix with a diagonal of 2s`
  - `mat2 m = mat2(1, 2, // [1, 2]
                   3, 4); // [3, 4]`

- Accessing elements
  - `float f = m[col][row];`
  - `float x = m[0].x; // X component of the first column`
  - `vec2 yz = m[1].yz // YZ components of the second column`
GLSL Syntax: Operations

- `vec3 v = vec3(1,2,3);`
- `vec3 v0 = 2.0 * v; //Scalar mult.`
- `vec3 v1 = v0 + v; //V + V`
- `vec3 v2 = v0 * v; //Component-wise multiplication`

- `mat3 m = mat3(1.0);`
- `vec3 mv = m * v;`
- `vec3 vm = v * m;`
Matrix functions:
  ○ transpose(m)
  ○ determinant(m)
  ○ inverse(m)

Remember that some operations are costly (e.g. inverse), so you may want to do them in your CPU code and pass the result to your shader.
GLSL Syntax: Operations

- Vector relational functions
  - \( \text{vec3 } p = \text{vec3}(1, 2, 3) \)
  - \( \text{vec3 } q = \text{vec3}(3, 2, 1) \)
    - \( \text{bvec3 } b = \text{equal}(p, q); \quad //\langle F, T, F \rangle \)
    - \( b = \text{lessThan}(p, q); \quad //\langle T, F, F \rangle \)
    - \( b = \text{greaterThan}(p, q) \quad //\langle F, F, T \rangle \)
    - \( \text{bool } b2 = \text{any}(b) \quad //\text{True} \)
    - \( b2 = \text{all}(b) \quad //\text{False} \)
GLSL Syntax: Operations

- Trigonometry
  - sin, cos, tan, asin

- Exponential
  - pow(x, y) //x^y
  - exp(x) //e^x
  - exp2(x) //2^x
  - log(x) //ln(x)
  - log2(x) //log_2(x)
  - sqrt(x)
  - inversesqrt(x)
GLSL Syntax: Operations

- Other useful functions:
  - abs(x)
  - sign(x) // -1, 0, or 1
  - min(x, y)
  - max(x, y)
  - clamp(x, a, b); //If x < a, x = a. If x > b, x = b.
  - floor(x), ceil(x)
  - step(edge, x) //Returns 0 if x < edge, else returns 1
  - smoothstep(edge0, edge1, x) //Interpolates x along a Hermite spline
GLSL Syntax: Operations

- Geometric functions
  - length(x)
  - distance(a, b)
  - dot(v1, v2)
  - cross(v1, v2)
  - normalize(x)
  - reflect(r, n)  //Returns r reflected across n
  - refract(r, n, idx)  //Returns r refracted by idx around n
  - faceforward(x, y)  //Returns x such that it faces the same direction as y
GLSL Syntax: Other

- Arrays
- Structs
- Function calls
- const
- if/while/for
  - Due to the parallel nature of shader execution, branches need to be managed carefully!
  - Shaders run as slow as the slowest thread
Shading: Review (Lambert shading)

- Given a point of intersection, a surface normal, and a light source, we can use the Lambert shading calculation for smooth shading.
- **Brightness** = clamp(dot(normalize(L - P), N), 0, 1) + ambient
  - ambient is some constant value you add to prevent points that are fully in shadow from being 100% dark.
Shading: Phong specular highlights

- float specular = max(pow(dot(H, N), u_shininess), 0)
- H = (V + L) / 2
- u_shininess controls how diffuse the highlight is