Today

- Explanation of Lab1
- Introduction to GLSL
- Basic GLSL syntax
- Phong Shading Model
Lab 1

- Create a per-pixel Phong renderer
- Use GLSL to create vertex and pixel shaders
GLSL

- GLSL is a high-level language for writing GPU programs.
- Two types of shaders in GLSL: vertex shaders and fragment shaders.
- These shaders override specific parts of the graphics pipeline.
Per-pixel coloration

Default OpenGL pipeline

- Vertex Data
- World space -> NDC conversion, Per-vertex Shading
- Rasterization/Interpolation
- Per-pixel coloration
Shader Pipeline

- Vertex Data
- Vertex Shader
- Rasterization/Interpolation
- Fragment Shader
Notes on GLSL Pipeline

- The vertex and pixel shader pipeline was designed for graphics, so there are some issues using it for non-graphics programming.
- Since shaders replace existing parts of the pipeline, one must often re-implement and extend default algorithms.
- Many currently installed GPUs support GLSL shaders.
Vertex Shaders

- Vertex Shaders are responsible for all per-vertex calculations.
- In the default pipeline:
  - Convert vertices from object space to NDC.
  - Perform all lighting/texture coordinate calculations
Fragment Shaders

- Fragment shaders are responsible for all per-pixel calculations.
- Before the fragment shader, all per-vertex data is interpolated by the rasterizer.
- In the default pipeline:
  - Simply sets the output color of the pixel to the interpolated vertex color.
GLSL Syntax and Programming

- GLSL has very similar syntax to C.
- `main()` function is entry point for both fragment and vertex shaders.
- Some extra language support for vector and matrix operations.
- Lots of built-in functions and variables
GLSL Types

- float, vec2, vec3, vec4
- int, ivec2, ivec3, ivec4
- bool, bvec2, bvec3, bvec4
- mat2, mat3, mat4
GLSL Built-in Variables

- Handy reference guide describes all built-in functions and variables:
  - Bookmark this!
- It has all of the reference for OpenGL in general, so it will be very useful for the next few weeks
Vertex Shader Built-in Variables

- `gl_Vertex`: Object space position
- `gl_Normal`: Object space normal
Vertex Shader Outputs

- `gl_FrontColor`: Write color here
- `gl_Position`: Write output NDC position
Pixel Shader

- `gl_Color`: Interpolated per-vertex color
- `gl_FragColor`: Write color here
Both Shader Variables

- `gl_LightModelParameters.ambient`
- `gl_LightSource[0].diffuse`, etc.
- `gl_FrontMaterial.diffuse`, etc.
- `gl_ModelViewMatrix`
- `gl_ProjectionMatrix`
- `gl_NormalMatrix`: Transform for normal to world space.
Built-in Functions

- `dot()`
- `sin`, `pow`, etc. Similar to math.h, in radians.
- `ftransform()`: Perform fixed transform
- `reflect()`
- Many More!
Communication

- What if shaders need more information than is given by the built-ins?
- Special keywords
  - Vertex->Fragment: varying
  - CPU -> Both (Per Pass): uniform
  - CPU -> Vertex (Per Vertex): attribute
  - CPU -> Both (Large Data): Use Textures! (next week)
Sample Vertex Shader

```
 varying vec3 normal, lightDir;

 void main()
 {
     normal = normalize(gl_NormalMatrix * gl_Normal);
     vec3 worldPos = vec3(gl_ModelViewMatrix*gl_Vertex);
     lightDir = normalize(vec3(gl_LightSource[0].position) - worldPos);
     gl_Position = ftransform();
 }
```
Sample Fragment Shader

- varying vec3 normal, lightDir;
- void main()
- {
-   vec4 color = gl_LightModel.ambient * gl_FrontMaterial.ambient;
-   float NdotL = max(dot(normalize(normal), normalize(lightDir)), 0.0);
-   if (NdotL > 0.0)
-     color += (gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse * NdotL);
-   gl_FragColor = color;
- }
Enabling Shaders

- Good example included in Lab 1
- Can use that code in future projects

```c
p = glCreateProgram();
v = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(v, 1, &src, NULL);
glCompileShader(v);
glAttachProgram(p, v);
glAttachProgram(p, f);
glLinkProgram(p);
glUseProgram(p);
```
Setting uniforms

- $L =\operatorname{glGetUniformLocation}(p, \text{"var"});$
- $\operatorname{glUniform1f}(L, \ .03);$
- Shaders must declare var outside of main function with uniform keyword.
Simple Lighting Models

- Sum of several components
Simple Lighting Models

- Diffuse component equation
- $I_d = C_d \times \cos(a)$
  - $C_d$ the diffuse color, $L$ light vector, $N$ normal vector
  - I.E. $(N \cdot L)$
Simple Lighting Models

- Specular component equation

\[ I_s = C_s \times (R \cdot \text{Eye})^S \]

- \( C_s \) the specular color, \( S \) the ‘shininess’
- \( R = 2N \times (L \cdot N) - L \) (or reflect())
- \( \text{Eye} = -\text{Pos} \) (if in eye-space)

- Note: All vectors must be normalized!
Simple Lighting Models

- Lighting equation
- Color = Ia + Id + Is
  - Ia ambient intensity
  - Id diffuse intensity
  - Is specular intensity
Blinn Shading

- OpenGL uses a simplified version of Phong shading called “Blinn” shading, and only calculates lighting on the vertices.
- \((H \cdot N)^S\) is intensity of specular
Blinn Shading
Phong Shading (Verts)
Phong Shading (pixels)