GLSL Help Session

How to shader
Outline

● OpenGL Pipeline
● Shader Programs
● Vertex Shaders
● Fragment Shaders
● Framebuffer Objects
● Summary
● Bonus: Geometry Shaders
OpenGL Pipeline
Basic OpenGL & GLSL Pipeline

C++ Side

- VBO
- VAO

GPU Side

- Vertex Shader: in vec -> out vec
- Fragment Shader: in vec -> out vec
- FBO
- Screen
Basic OpenGL & GLSL Pipeline

**C++ Side**

- VBO

**GPU Side**

- Vertex Shader
- Fragment Shader
- FBO
- Screen

Supply geometry data
Basic OpenGL & GLSL Pipeline

C++ Side

VBO

GPU Side

VAO

Specify how data is to be read by Vertex Shader

in vec

Vertex Shader

out vec

Fragment Shader

in vec

out vec

FBO

Screen
Basic OpenGL & GLSL Pipeline

**C++ Side**
- VBO
- VAO

**GPU Side**
- Receive each vertex one by one in vertex shader
- Vertex Shader
- Fragment Shader
- FBO
- Screen

In vec

Out vec
Basic OpenGL & GLSL Pipeline

C++ Side

- VBO
- VAO

GPU Side

- Vertex Shader
  - Do work on each vertex
  - in vec
  - out vec

- Fragment Shader
  - in vec
  - out vec

Examples:
- Transform to world space
- Do camera/projection transform
- Calculate per-pixel lighting
- Other cool stuff
Basic OpenGL & GLSL Pipeline

C++ Side:
- VBO
- VAO

GPU Side:
- Vertex Shader
- Fragment Shader
- FBO
- Screen

Must output: Final (projected) vertex position

Other things to output:
- Vertex color
- Vertex world-space position
- Vertex normal
- Texture coordinate
- Per-vertex calculated data

Output end results to next stage
Basic OpenGL & GLSL Pipeline

C++ Side
- VBO
- VAO

GPU Side
- Vertex Shader
- Fragment Shader
- FBO (Framebuffer Object)
- Screen

Fixed-function interpolation - vertex in, pixel out
Interpolate vertex data across triangle to get value at pixel
Basic OpenGL & GLSL Pipeline

C++ Side

VBO

VAO

GPU Side

Vertex Shader

Fragment Shader

FBO

Screen

Receive interpolated pixel ("fragment") one by one in fragment shader
Basic OpenGL & GLSL Pipeline

C++ Side

VBO

VAO

Use interpolated vertex values to:

- Calculate lighting
- Discard pixels
- Do post-processing effects
  - Bloom, HDR
  - SSAO
  - Zillions more
- Do crazy stuff you wouldn’t expect

Vertex Shader

GPU Side

Fragment Shader

FBO

Do work on each pixel

Screen
Basic OpenGL & GLSL Pipeline

C++ Side

- VBO
- VAO

GPU Side

- Vertex Shader
- Fragment Shader
- FBO

Things you might think to output:

- Final pixel color (usual default)
- Pixel position
- Partially computed light
- Screen depth

Write pixel value to FBO

Screen
Basic OpenGL & GLSL Pipeline

C++ Side
- VBO
- VAO

GPU Side
- Vertex Shader
- Fragment Shader
- FBO

FBO stores output in texture(s)
Basic OpenGL & GLSL Pipeline

C++ Side

VBO

VAO

GPU Side

Vertex Shader

Fragment Shader

FBO

Send output to screen!
Basic OpenGL & GLSL Pipeline

**C++ Side**
- VBO
- VAO

**GPU Side**
- Vertex Shader
- Fragment Shader
- FBO

You can send your pixels to an offline FBO and reuse the generated texture in another shader.

Example: Render one scene into FBO, then use image as texture to texture a teapot which does end up on the screen.

We’ll cover this later.
Shader Programs
Complete Shader Program

● Made up of different shader “sub-types”
● Need at least a vertex and a fragment shader
  ○ Yes, always. No exceptions.
● Max one of each type
● There are others:
  ○ Geometry
  ○ Tessellation
Minimum Shader

Vertex → Fragment
Bigger shader

Vertex → Geometry → Fragment
Shader Programs (cont.)

- A shader program is one pipeline of sub-shader types
  - EG: Vertex A → Fragment B
- If you want to reuse a sub-shader, you must make another shader program
  - Program 1: Vertex A -> Fragment B
  - Program 2: Vertex A -> Fragment C
Shader Inputs

Two main types:

● Per-element data
  ○ The streaming inputs to the shaders (e.g. vertices and interpolated pixels)

● Uniform data
  ○ Constant until changed explicitly on the CPU side
VS Executes Once Per Vertex

Uniform

Vertex Data 143

Uniform

Vertex Data 144

Uniform

Vertex Data 145

Uniform

Vertex Data 146
VS Executes Once Per Vertex

Position, Normal, Texture Coordinate…

Different on each invocation of shader function (but you already knew that).
VS Executes Once Per Vertex

Uniforms stay the same across all per-element invocations of the shader...
VS Executes Once Per Vertex

Unless you specifically change them in between draw calls

Note: Uniforms stay the same for everything drawn in a single draw call. If you want to change something in between, you need to break up your draw call.
FS Executes Once Per Fragment

Unless you specifically change them in between draw calls

Uniform

Frag Data 916

Uniform

Frag Data 917

Uniform

Frag Data 918

Uniform

Frag Data 919

Same deal for Frag Shader!

glUniform3f(…)

Note: Uniforms stay the same for everything drawn in a single draw call. If you want to change something in between, you need to break up your draw call.
Shader Inputs Examples (1/2)

- **Per-element examples**
  - Vertex Shader: The coordinates of a mesh in model space
  - Fragment shader: The vertex data of a triangle interpolated across its pixels

- **Uniform examples**
  - ModelViewProjection matrix: Every vertex gets multiplied by the same one.
Shader Inputs Examples (2/2)

● More Uniform examples
  ○ Material properties
    ■ Diffuse color, specular color
    ■ Textures used for texture-mapping
    ■ Other fancy maps (normal maps, AO maps, etc.)
  ○ Light position, color
  ○ Previously computed textures for post-processing
  ○ Any constant you want to change CPU-side
Multiple Shaders

- You can use more than one
  - One object gets this shader
  - An effect gets that shader
  - Everyone can have a shader
  - Or you can start with a general one for everything

- Only one is bound at a time
  - Wouldn’t make sense to use multiple shaders simultaneously
  - Use each one sequentially.
Vertex Shaders
What are they good for?

- Functions run on every vertex
  - At least transform coordinates from model space into world->camera->screen space
- Compute per-vertex lighting
- Animate vertices
- Something else creative?
Example: Lighting

1. Using light position, vertex position, vertex normal, and material, compute diffuse color
2. Repeat for specular
3. Add ambient hack
4. Output pixel’s color
Example: Animation

- Before transforming vertex, multiply position by \((1 + 0.5\times\sin(time))\)
- Position oscillates in magnitude
- End result?
  - Object scales up and down over time
  - How does it change if you add this step in the middle of the transformation process rather than beginning?
Fragment Shaders
What are they good for?

- Function runs on every pixel
  - This is for every projected triangle’s pixel!
  - Not just once per output pixel!
  - EG: If one triangle’s pixel ends up closer than another’s, recompute that pixel

- Can do much of what vertex shader can do, but more accurately
  - Per pixel is finer than per vertex
  - Per pixel is also more expensive
Wait a minute...

- I sent out per-vertex data from my Vertex shader
- But I get per-fragment data coming into my Fragment shader
- How does that work?
Triangle Interpolation

Start with vertices
Triangle Interpolation

Interpolate to fill in triangle

Color = I(position, vertex values)
Triangle Interpolation

Fragment shader does work per pixel
Interpolation

- Not limited to color!
  - Works for normal vectors (remember to renormalize)
    - Bitangents, binormals, etc.
  - Textures are interpolation champs
    - Texture coordinates, texture map values

- Provides smoother effect
  - Calculating lighting per pixel is more accurate

- FIXED FUNCTION
  - You do not have direct control, can only pick settings.
Z-Buffer

- The Z-Buffer keeps track of the depth of the pixels within the scene.
- Ranges from 0-1.
- When Fragment shader gets a pixel, it checks its depth with the current Z-Buffer value.
  - If the new pixel is behind the old pixel, don’t bother computing it.
Z-Buffer Example

Current pixel color

Current pixel depth

Depth 0

Depth 1

Z-Buffer
Z-Buffer Example

Current pixel color

Current pixel depth

New pixel’s depth

Depth 0

Depth 1

Z-Buffer

Compare new pixel’s depth to current pixel’s depth
Z-Buffer Example

Current pixel color

Current pixel depth

New pixel’s depth

Depth 0

Depth 1

Winner

Current pixel is closer, don’t both computing values for new one
Z-Buffer Example

Current pixel color

Current pixel depth

Depth 0

Z-Buffer

Depth 1
Z-Buffer Example

Current pixel color

Current pixel depth

Depth 0

New pixel’s depth

Z-Buffer

Depth 1

Compare new pixel’s depth to current pixel’s depth
Z-Buffer Example

Current pixel color

Current pixel depth

Depth 0

New pixel’s depth

Z-Buffer

Depth 1

New pixel is closer, compute the new value, toss out old one.
Z-Buffer Example

Current pixel color  Current pixel depth

Depth 0  Z-Buffer  Depth 1

New pixel is closer, compute the new value, toss out old one.
Framebuffer Objects
(FBOs)
Bundle of Textures

- You can think of an FBO as a group of textures.
- The outputs of your fragment shader fill in these textures.
- You have control on the C++ side of a number of settings for how they work:
  - Memory usage, precision, data type, multisampling...
Default FBO

- By default, your fragment shader outputs data to OpenGL’s default FBO
- Writing to this means writing to the screen
- You’ll want to write to the default FBO eventually
Custom FBOs

- Sometimes you’ll want to compute some intermediate data before you write to the screen
- Save this data in an FBO
- Use results in later invocation of shader
FBO Pipeline

Geometry
Uniforms

Shader does work
Computes output

FBO stores data

Shader does work
Computes output

Screen

Inputs on later pass almost always different from inputs on first pass

FBO data is a texture uniform in later pass
Custom FBOs

● Open up many possibilities
● Somewhat complicated (especially in practice)
● Multiple Render Targets
  ○ FBO has multiple textures, fragment shader has an output for each texture in the FBO.
● Best explained through an example
Example: Blur

1. Render scene normally
   a. Output to custom FBO

2. Render textured quad using scene image
   a. Vertex shader: Draw quad to take up whole screen
   b. Fragment shader: Rather than compute lighting, evaluate blur kernel at each pixel
      i. Works just like Filter
   c. Output to default FBO
   d. See blurry picture on screen
Blur Pipeline

C++ Side

VBO

VAO

GPU Side

Vertex Shader

Fragment Shader

FBO

Uniform

Screen

Render scene just like before
Blur Pipeline

C++ Side

VBO

GPU Side

VAO

Vertex Shader

FBO

Fragment Shader

Uniform

Send to custom FBO, not default
Blur Pipeline

C++ Side
- VBO

GPU Side
- Vertex Shader
- Fragment Shader
- FBO
- Uniform
- Screen

Render a full screen quad using Blur shader

VBO

VAO
C++ Side

VBO

VAO

VBO

VAO

GPU Side

Vertex Shader

Fragment Shader

VAO

FBO

Uniform

Vertex Shader

Fragment Shader

FBO

Screen

Set image in FBO as a uniform texture for next pass
Pass quad through Vertex Shader. It doesn't do much.
Evaluate blur kernel at each pixel in texture (which is spread across quad).
Send output to default FBO, which shows up on screen!
Summary
Multiple Shader Passes

- You can use many shaders in order to produce a final picture
- You can accomplish a lot by rendering into FBOs with shaders and then using those textures as inputs to other shaders down the line
  - Blur was an example of this
Complex scene example

1. Render scene from light’s point of view
   a. Store shadow map
2. Render initial scene
   a. Metal shaders
   b. Glass shaders
   c. Normal phong shaders
   d. Apply lighting based on shadow map
   e. Store as HDR data in FBO
3. Render particle effects into their own FBOs
4. Do post processing
   a. Sample depth output to compute SSAO
   b. Combine particle effects into scene by comparing depth buffers
   c. Hi-pass filter pass - store in another FBO
      i. Blur pass on that pass - store in yet another FBO
   d. Combine everything in final pass to get bloom effect -&gt; output to default (screen) FBO
Shader → FBO Pattern

- Basic building block for many effects
  - Render geometry with shader, save output to FBO
  - Use texture as input to another shader pass
- Usually render scene geometry first
- Then do post-processing using that data
Intuition Summary

- Shaders can do lots of things despite having very structured input
  - Lighting, particle effects …
  - It might seem like shaders are only for computing lighting. They can do a lot more! Play with examples - get practice.

- FBOs - Storage for intermediate data
Geometry Shaders

Bonus round!
What does the Geometry Shader do?

- The geometry shader can output additional vertices
- Saves space
- Allows for procedural effects
- Note: not the same as a tessellation shader
Example: Billboardning

- Billboardning creates 2D quads out of points.
- The sprites always face the camera.
- Useful for achieving all sorts of effects:
  - Volumetric effects
  - Sprites
  - View-dependent detail hacks
    - Leaves on tree
    - Grass
Billboarding (cont.)

- Input: 3-D point, camera matrix
- For each point
  - Generate 4 points
  - Each point is pushed out along the camera’s uv axes so that each forms the corner of a quad
  - Output 2 triangles from these points
Billboard construction:

- Start with point
- Generate points using camera \((u,v)\) vectors
- Stitch together a quad
But there’s a problem

- Placing quads in the camera plane does not account for rotation
- Quads ends up projected orthogonally
- Not too bad from far away at small angles, but up close it’s very noticeable

Naive billboard
Improved Billboarding

- Rotate billboard to correct for difference between look vector and vector to the billboard
  - More fun with trigonometry!
Billboard rotation

Original Method:

Vector to billboard

Look vector

θ

View plane

Improved Method:

Rotated billboard

θ
Examples
The End

Thanks for coming!