GLSL Help Session

Throwing some shade
Overview

- **OpenGL Pipeline**
- **Shader Programs**
- **Vertex Shaders**
- **Fragment Shaders**
- **Framebuffer Objects**
- **Summary**
- **Bonus: Geometry Shaders**
OpenGL Pipeline

"Technology is incredible! You can now store and recall items and Pokemon as data via PC!" ~Red/Green
Basic OpenGL & GLSL Pipeline

C++ Side
- VBO
- VAO

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

Basic OpenGL & GLSL Pipeline

C++ Side
- VBO
- VAO

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

Basic OpenGL & GLSL Pipeline

C++ Side
- VBO
- VAO

GPU Side
- Vertex Shader
- Fragment Shader
- FBO
- Screen
Basic OpenGL & GLSL Pipeline

C++ Side

- VBO
- VAO

Supply geometry data

GPU Side

- Vertex Shader
- Fragment Shader
- FBO
- Screen

in vec -> out vec
Basic OpenGL & GLSL Pipeline

C++ Side
- VBO

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

Specify how data is to be read by Vertex Shader
Basic OpenGL & GLSL Pipeline

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

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

Receive each vertex one by one in vertex shader
Basic OpenGL & GLSL Pipeline

C++ Side

- VBO
- VAO

GPU Side

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

- FBO
- Screen

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

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

**GPU Side**
- Vertex Shader
  - Input: `in vec
  - Output: `out vec`
- Fragment Shader
  - Input: `in vec`
  - Output: `out vec`

Output end results to next stage

Must output: Final (projected) vertex position

Other things to output:
- Vertex color
- Vertex world-space position
- Vertex normal
- Texture coordinate
- Per-vertex calculated data
Basic OpenGL & GLSL Pipeline

C++ Side

VBO

GPU Side

Fragment Shader

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

GPU Side

Vertex Shader

- in vec
- out vec

Fragment Shader

- in vec
- out vec

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

Do work on each pixel

FBO

Screen
Things you might think to output:
- Final pixel color (usual default)
- Pixel position
- Partially computed light
- Screen depth
Basic OpenGL & GLSL Pipeline

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

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

FBO stores output in texture(s)

in vec → out vec

Screen
Basic OpenGL & GLSL Pipeline

C++ Side

- VBO
- VAO

GPU Side

- Vertex Shader
- Fragment Shader
- FBO
- Screen

Send output to screen!
Basic OpenGL & GLSL Pipeline

C++ Side

VBO

GPU Side

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 (not the screen!), then use image as the texture for a teapot which actually ends up on the screen.

We’ll cover this later.
Shader Programs

"If you need to make a difficult decision and you let someone else decide for you, you will regret it, no matter how it turns out." ~ someone in Ruby/Sapphire
Complete Shader Program

- There are different shader “sub-types” (kinds of shaders)
- Need at least a vertex and a fragment shader
  - There are other types:
    - Geometry
    - Tessellation
Minimum Shader Program

Vertex → Fragment
Bigger shader program

Vertex → Geometry → Fragment
Shader Programs (cont.)

- A shader program is one complete pipeline of sub-shader types
  - EG: Vertex A → Fragment B
    - Could have Vertex B, C, D, ..., used in other programs, and likewise for other shader subtypes
- 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). These will usually be different on every invocation of the sub-shader.

- **Uniform data**
  - Constant across all invocations of the sub-shader 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

- Uniform
- Uniform
- Uniform
- Uniform
- Vertex Data 143
- Vertex Data 144
- Vertex Data 145
- Vertex Data 146

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

Uniform
Vertex Data 143

Uniform
Vertex Data 144

Uniform
Vertex Data 145

Uniform
Vertex Data 146

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.

glUniform3f(…)

26/79
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!

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
  - Model, view, projection matrices: Every vertex gets multiplied by the same ones.
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 Shader Programs (SP)

- You can use more than one SP
  - Different types of objects can use specific SPs
    - Wooden Objects = Wooden SP
    - Metal Objects = Metal SP
  - Or you can use a general SP for everything
- Bind one SP per render call
  - Bind, draw, unbind, bind the next one, …
  - An SP is a render pipeline: using more than one on a single object will render it multiple times!
Vertex Shaders

“I see now that the circumstances of one’s birth are irrelevant. It is what you do with the gift of life that determines who you are.”
~ Mewtwo
What are they good for?

- Functions that run on every vertex
  - 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 vertex’s color
Example: Animation

● Before transforming vertex, multiply position by a periodic function, say \((1 + 0.5\sin(t))\)
  ○ Position oscillates in magnitude
  ○ End result?
    ■ Object scales up and down in size over time
    ■ How does it change if you add this step in the middle of the transformation process rather than beginning?
Fragment Shaders

“A Caterpie may change into a Butterfree, but the heart that beats inside remains the same.” ~ Brock
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, we 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 (Rasterization)

Start with vertices
Triangle Interpolation (Rasterization)

Interpolate to fill in triangle

Color = I(position, vertex values)
Triangle Interpolation (Rasterization)

Fragment shader does work per pixel
Interpolation

- Not limited to color!
  - Works for normal vectors (remember to renormalize)
    - Tangents, 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.
  - Requires `glEnable(GL_DEPTH_TEST)`
Z-Buffer Example

Current pixel color (displayed)

Current pixel

Depth 0

Depth 1

The closest pixel color is what we'll display!
Z-Buffer Example

Compare new pixel’s depth to current pixel’s depth

Current pixel color
Current pixel depth
New pixel’s depth

Depth 0
Depth 1

Z-Buffer
Z-Buffer Example

Current pixel color

Current pixel depth

New pixel’s depth

Depth 0

Winner

Depth 1

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

Current pixel color

Current pixel depth

Depth 0

Depth 1

Z-Buffer
Z-Buffer Example

- Current pixel color
- Current pixel depth
- Depth 0
- New pixel’s depth
- 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

Winner

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

New pixel is closer, compute the new value, toss out old one.
Framebuffer Objects

Just a Souvenir Shop
Nothing Suspicious about It
No Need to Be Alarmed
~Shop in Mahogany Town
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

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

Geometry
Uniforms

Shader does work Computes output

FBO stores data

FBO data is a texture uniform in later pass

Screen
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. Draw quad to take up whole screen
   b. Vertex shader: Pass-through
   c. Fragment shader: Rather than compute lighting, evaluate blur kernel at each pixel
      i. Works just like Filter
   d. Output to default FBO
   e. See blurry picture on screen
Blur Pipeline

C++ Side
- VBO
  - VAO

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

C++ Side
- VBO
- VAO

Vertex Shader
Fragment Shader
FBO
Uniform
Screen

GPU Side
- VBO
- VAO

Vertex Shader
Fragment Shader
FBO

Render scene just like before
Blur Pipeline

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

Send to custom FBO, not default

**GPU Side**
- Vertex Shader
- Fragment Shader
- VAO
- Uniform
- FBO
- Screen
Blur Pipeline

C++ Side
- VBO

VAO

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

Render a full screen quad using Blur shader
Blur Pipeline

C++ Side

- VBO

GPU Side

- Vertex Shader
- Fragment Shader
- VAO

Set image in FBO as a uniform texture for next pass
Blur Pipeline

C++ Side
- VBO
- VBO

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

Pass quad through Vertex Shader. It doesn't do much.
Evaluate blur kernel at each pixel in texture (which is spread across quad).
Blur Pipeline

C++ Side

VBO

GPU Side

VAO

Vertex Shader

FBO

Fragment Shader

Uniform

VAO

Vertex Shader

FBO

Fragment Shader

Send output to default FBO, which shows up on screen!
Summary

"..." ~ Red in Gold/Silver
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 -> 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 …
  - Bricks Game
- FBOs - Storage for intermediate data
  - You’ll use this in Lab 8 (Particles)
Geometry Shaders

"Porygon! Porygon" ~ Porygon dispensing wisdom about life in general
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: Billboardding

- Billboardding 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

View plane
But there’s a problem

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

Output from naive billboardiing
Improved Billboard

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

Output from improved billboard
Billboard rotation

Original Method:
- Vector to billboard
- Look vector
- View plane
- Rotation

Improved Method:
- Rotated billboard
- View plane
Examples

Shadertoy:

● [Dolphin](#)
● [Fishing Village](#)
● [Noise Contour Post-Processing Filter](#)
The End

Thanks for coming!