

Shaders

CSCI 4229/5229

Computer Graphics

Fall 2026

What is a Shader?

- A shader is a computer program that runs on the GPU to calculate the properties of vertexes, pixels and other graphical processing
- Examples:
 - Vertex position or color computed by a program
 - Texture generated by a program
 - Per-pixel lighting
 - Image processing
 - Cartoon shading

How does a shader work?

- Shader Language used to specify operations
 - RenderMan, ISL, HLSL, Cg, GLSL
- Compile instructions into program
 - e.g. `glCompileShader()`
- Shader performs calculations as part of graphics pipeline
- Runs calculations on GPU instead of CPU

What is a Shader Language?

- Typically C/C++ like
 - for, while, if, ... for control flow
 - Adds special types like vec4 (4 component vector) and mat4 (4x4 matrix) and operators
 - Predefined variables used to get data (gl_Vertex) and return result (gl_Position)
- Simplifies and extends C/C++ for efficiency
 - Matrix & vector operations supported in hardware Graphics Processing Unit (GPU)
 - Built-in functions like normal, blend, etc.

GL Shader Language (GLSL)

- Often call “GLSLang”
- Added to OpenGL 2.0
 - First appeared as extension in OpenGL 1.4
 - Can be accessed in older versions using extensions
 - GL Extension Wrangler (GLEW) often used
- Geared to real time graphics
 - Inserted into OpenGL pipeline
 - Vertex Shader to manipulate vertexes
 - Fragment Shader to manipulate pixels

GLSL Resources

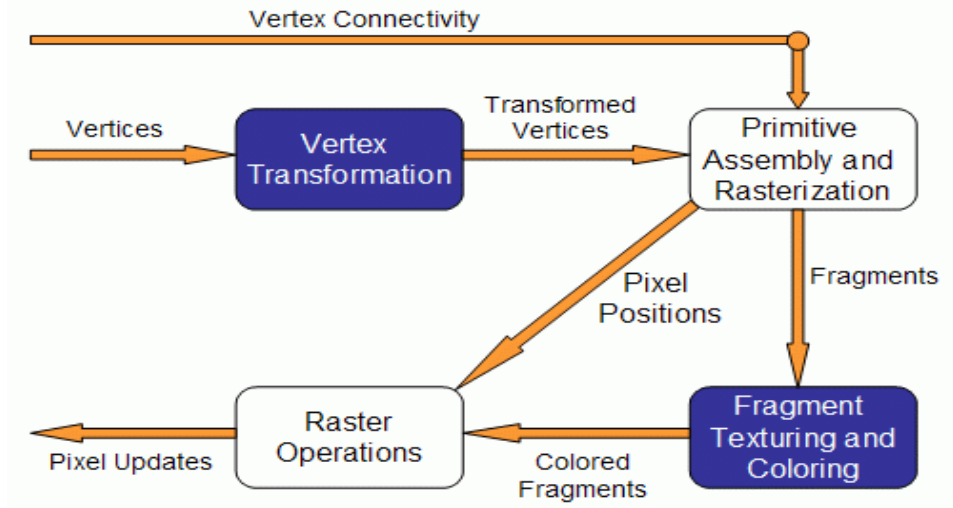
- OpenGL Programming Guide (9^{ed})
 - Merges the old Red and Orange books
 - Don't get older editions
- GLSL Quick Reference
 - “Cheat sheet”
- Many online references
 - <http://www.lighthouse3d.com/opengl/glsl/>
 - Watch out for old stuff (OpenGL < 2)
 - Don't be confused by newest stuff (OpenGL 4)

OpenGL Deprecation

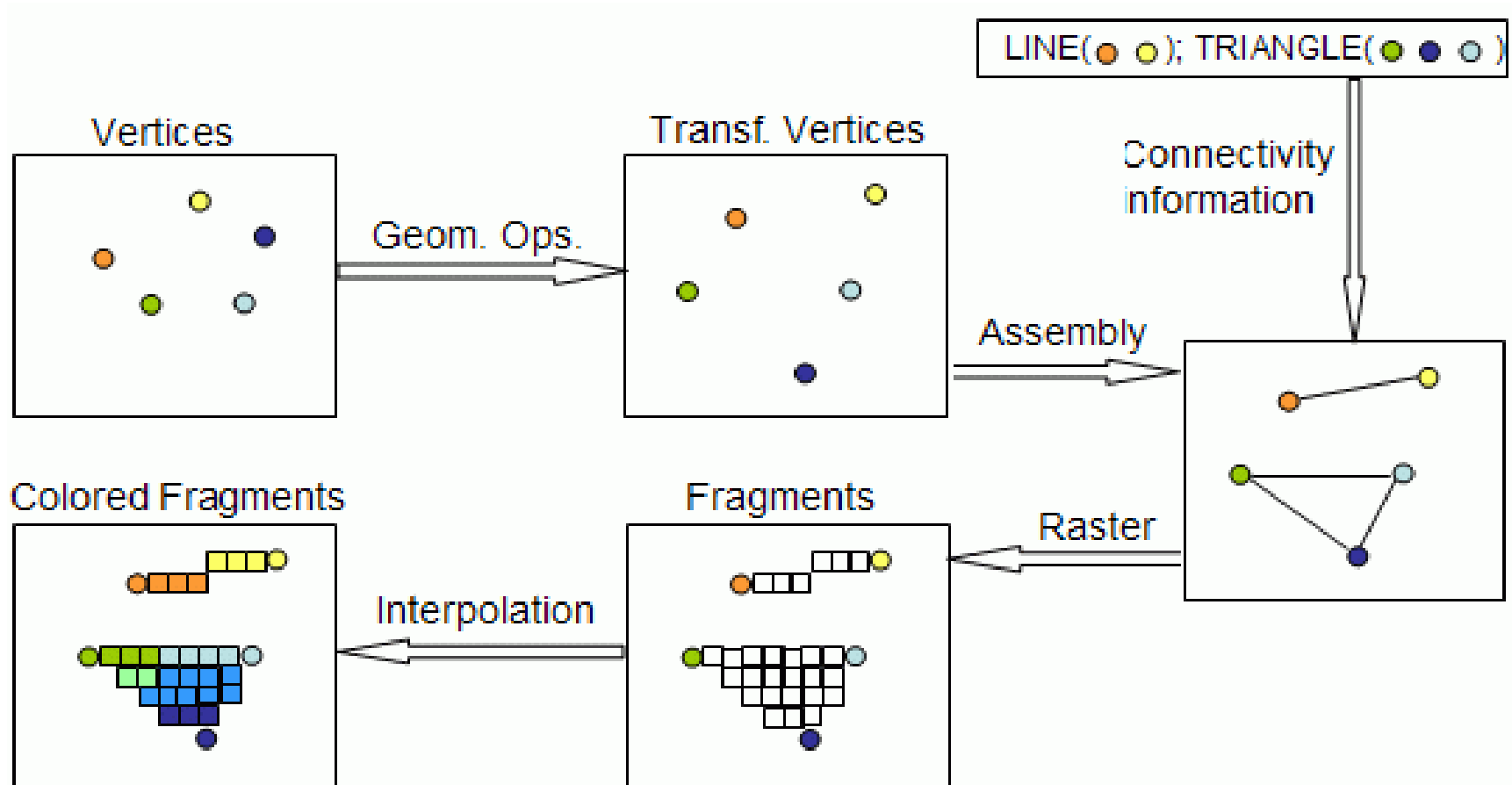
- I will mostly use OpenGL 2.x
 - Feature rich
 - Flat learning curve
 - Advanced examples will use GL4 and Vulkan
- OpenGL Core Profile concentrates on rendering
 - Improved execution time performance
- User must provide deprecated functionality
 - Steepens the learning curve
 - Deprecated features in Compatibility Profile
 - Increases reliance on third party libraries

Where does GLSL fit?

- Vertex shader
 - Transformations, color, texture coordinates, ...
- Fragment shader
 - Textures, Color Interpolation, Fog, ...
- OpenGL still does Z-buffering, etc.



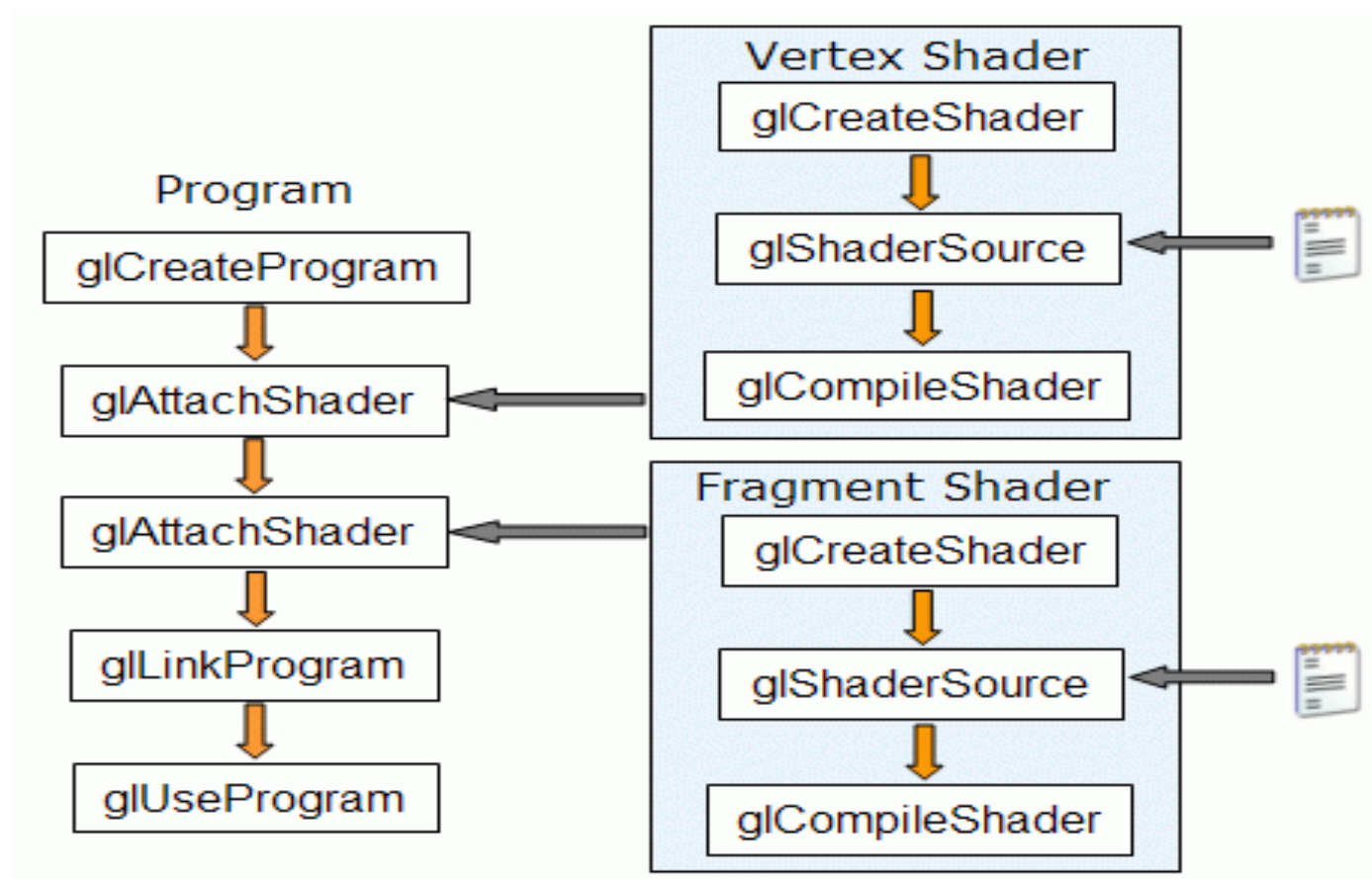
Fixed Pipeline Example



How is this different from what we have done before?

- GLSL instructions can run on GPU
 - Matrix-vector multiplications done *fast*
- Without GLSL we influence the pipeline using parameters and fixed operations
 - Lighting calculated at vertexes
 - Textures calculated at fragments
 - Vertex-fragment interpolation
 - GL_SMOOTH bilinear interpolation
 - GL_FLAT constant using last vertex
- With GLSL we can calculate values directly

How does this work with OpenGL?



Other Shader Languages

- RenderMan
 - Lucasfilm - Pixar - Disney
- OpenGL Shader (ISL)
 - SGI Interactive Shader Language
- High-Level Shader Language (HLSL)
 - Microsoft DirectX 9
- NVIDIA's Cg
 - proprietary shading language

RenderMan

- First practical shading language (1988)
- De-facto entertainment industry standard
- Remains in widespread use today
- Generally used for off-line rendering
 - Uncompromising image quality
 - Little hardware acceleration
- Credits:
 - Jurassic Park, Star Wars Prequels, Lord of the Rings
 - Toy Story, Finding Nemo, Monsters Inc, ...
- No relation to OpenGL in syntax or structure

The Rest (ISL, HLSL, Cg, ...)

- Syntax different but similar approach
- Generally similar in structure
 - Vertex Shader
 - Fragment Shader
- Geared towards real time graphics
 - Hardware support
 - Performance stressed

GLSL Versions

- GLSL 1.0 = OpenGL 1.4 (2002)
 - The first portable shader
- GLSL 1.2 = OpenGL 2.0 (2004)
 - The shader we will use
- GLSL 1.3 = OpenGL 3.0 (2008)
 - Some changes in syntax
 - Deprecates some features
- GLSL 3.3 = OpenGL 3.3
 - From here on GLSL version match OpenGL
- Set minimum version using `#version`

GLSL 1.2 Variable Qualifiers

- uniform (e.g. `gl_ModelViewMatrix`)
 - input to vertex and fragment shader from OpenGL or application [read-only]
- attribute (e.g. `gl_Vertex`)
 - input per-vertex to vertex shader from OpenGL or application [read-only]
- varying (e.g. `gl_FrontColor`)
 - output from vertex shader [read-write], interpolated, then input to fragment shader [read-only]
- const (e.g. `gl_MaxLights`)
 - compile-time constant [read-only]

What is new in OpenGL 3&4

- Additional shaders
 - Geometry (OpenGL 3.2)
 - Tessellation (OpenGL 4.0)
 - Compute (OpenGL 4.3)
- New syntax for passing variables
 - “in” from previous stage
 - “out” to next stage
 - Deprecating most predefined variables
- Building objects from vertex arrays
- Deprecating OpenGL transformations

Vulkan

- Vulkan is what would have been GL5
- Breaks backwards compatibility, but strongly resembles OpenGL
- Requires you to be very explicit
- Close to the metal, little abstractions
- Super verbose, very steep learning curve
- Requires tons of scaffolding

GLSL 4 Variable Qualifiers

- `const`
 - compile-time constant
- `uniform`
 - data from CPU to shader
- `in`
 - per-vertex input to vertex shader
 - input from previous shader for others
- `out`
 - resulting vertex and fragment properties
 - output to next shader

The problem with shaders

- EXTREMELY hard to debug
 - No “print” statements
- You have to have to do lighting yourself
- Support is spotty
 - GLSL requires OpenGL 2.0 or extensions
 - Still somewhat a work in progress
 - Generally needs decent hardware
- So why use it?
 - Ultimate flexibility
 - Unsupported features (e.g. bump maps)

OpenGL Extension Wrangler (GLEW)

- Maps OpenGL extensions at run time
 - Provides headers for latest OpenGL
 - Finds vendor support at run time
- Check support for specific functions or OpenGL version at run time
 - Crashes if unsupported features are used
- Use only if you have to (Windows mostly)
 - Set -dUSEGLEW to selectively invoke it
 - Do NOT require GLEW (I don't need it)