Dieter Schmalstieg

Introduction to OpenGL

What is OpenGL?

- A low-level graphics API *specification*
  - Not a library
    - Interface is platform independent
    - But implementation is platform dependent
  - Defines
    - Abstract rendering device
    - Set of functions to operate the device
  - “Immediate mode” API
    - Drawing commands
    - No concept of permanent objects

API and Vendor Overview

Graphics API

- DirectX 12 (Microsoft)
- OpenGL
- OpenGL ES
- *Vulkan (Khronos group)*
- *Mantle (AMD)*
- *Metal (Apple)*

GPU Hardware Vendors

- Intel (GPU+CPU combined)
- NVIDIA (GeForce, Tegra)
- AMD (Radeon)
- Qualcomm (Adreno)
- Imagination (PowerVR)
- AMD (Mali, only design)

OpenGL Implementation

- Platform provides OpenGL *implementation*
  - Part of graphics driver, or
  - Runtime library built on top of driver

- Initialization through platform specific API
  - WGL (Windows)
  - GLX (Unix/Linux)
  - EGL (mobile devices)
  - …

Basic Concepts

- Context
- Resources
- Object Model
  - Objects
  - Object Names
  - Bind Points (Targets)

The Context

- Represents an instance of OpenGL
- One process can have multiple contexts
  - Contexts can share resources
- *Current context* for a given thread
  - One to one mapping
    - Only one current context per thread
    - Context only current in one thread at the same time
  - OpenGL operations work on current context

Pipeline Overview

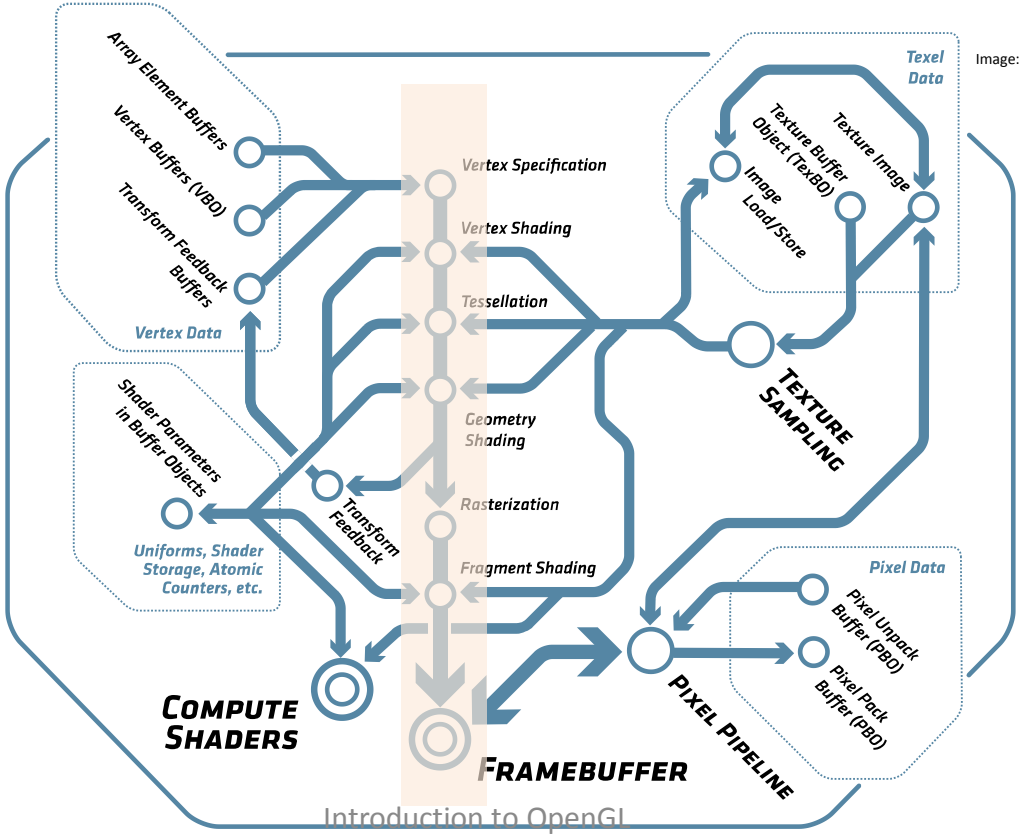


Image: The Khronos Group Inc.

Resources

- Act as
  - Sources of input
  - Sinks for output
- Examples
  - Buffers
    - Linear chunks of memory
  - Images
    - 1D, 2D, or 3D arrays of *texels*
    - Can be used as input for *texture sampling*
  - State objects…

Object Model

- OpenGL is object-oriented
  - But in its own, strange way
- Object instances are identified by a *name*
  - Basically just an unsigned integer handle
- Commands work on *targets*
  - Each target has an object currently *bound* to the target
  - Commands will work on bound object
- OpenGL's style of object-oriented programming
  - Target ⇔ type
  - Commands ⇔ methods

Binding

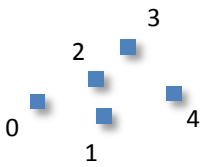
- Bind a name to a target = “activate” an object
  - Bound object becomes current for that target
    - “Latched state”
    - Might change soon (`EXT_direct_state_access`)
  - Object is created when a name is first bound
- Notable exceptions: Shader Objects, Program Objects
  - Some commands work directly on object names

Example: Buffer Object

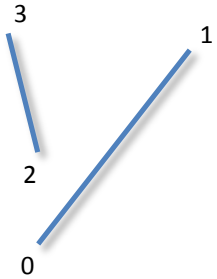
```
1  GLuint my_buffer;

2  // request an unused buffer object name
3  glGenBuffers(1, &my_buffer);

4  // bind name as GL_ARRAY_BUFFER
5  // bound for the first time ⇒ creates new array buffer object
6  glBindBuffer(GL_ARRAY_BUFFER, my_buffer);

7  // put some data into my_buffer
8  glBufferStorage(GL_ARRAY_BUFFER, …);

9  // probably do something else…

10 glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
11 // use my_buffer…

12 // delete buffer object, free resources, release buffer object name
13 glDeleteBuffers(1, &my_buffer);
```

Primitive Types

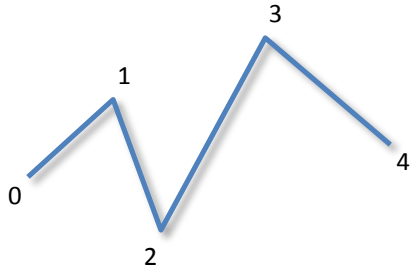
GL_POINTS

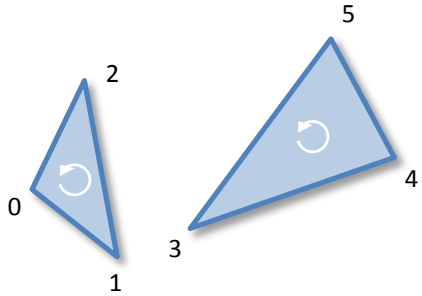


GL_LINES

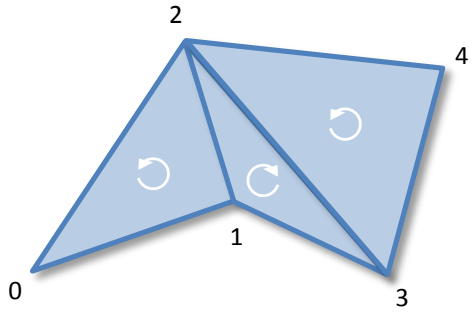


GL_LINE_STRIP





GL_TRIANGLES

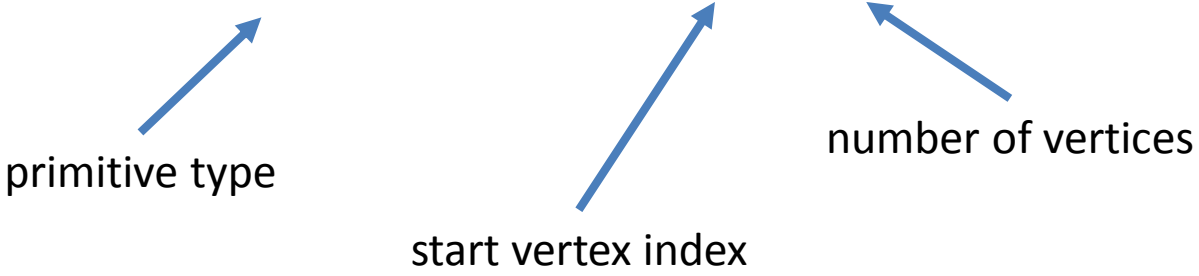


GL_TRIANGLE_STRIP

Draw Call

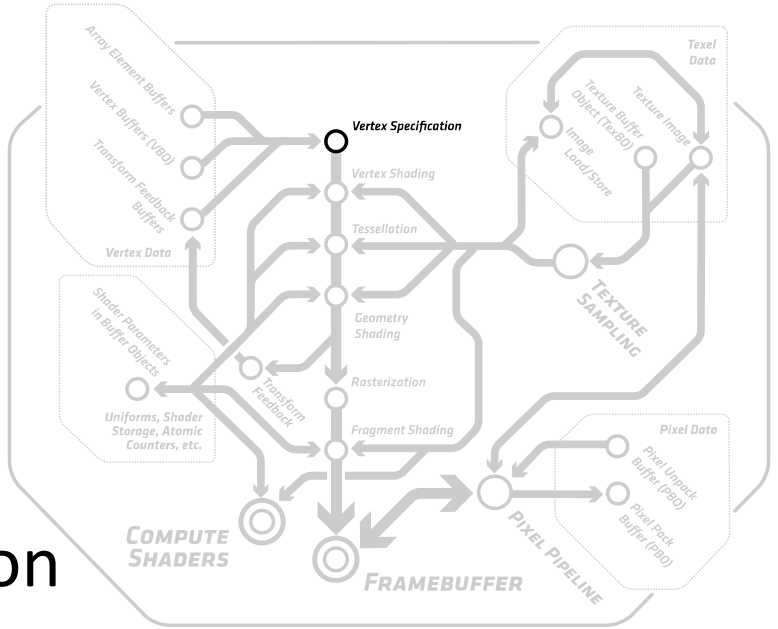
- After pipeline is configured:
  - issue *draw call* to actually draw something.

E. g.: `glDrawArrays(GL_TRIANGLES, 0, 33);`



Vertex Specification

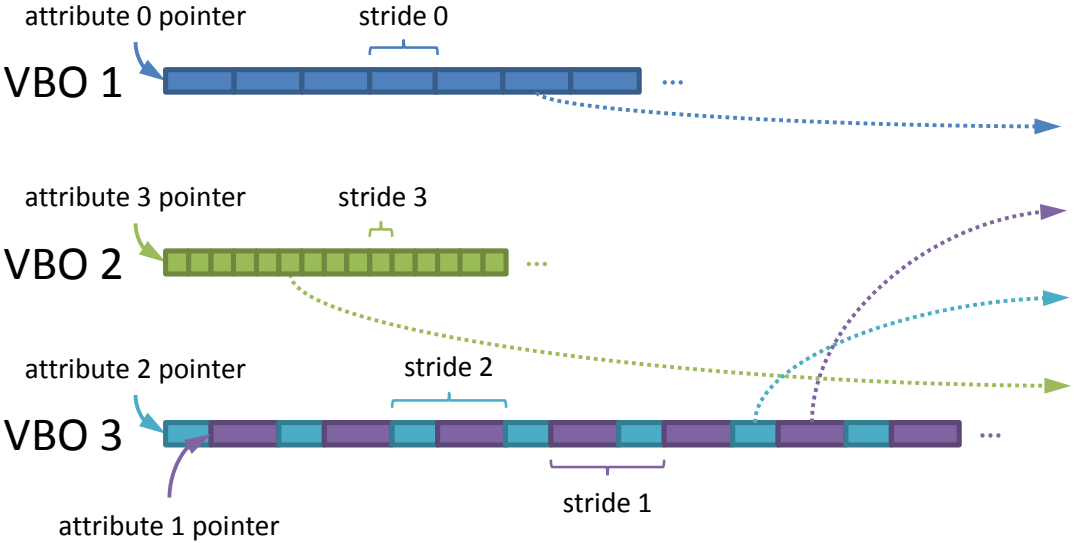
- Wires together pipeline input
- Configures vertex fetch
  - Reads vertex attribute data from buffers
  - Performs data format conversion
  - Feeds vertex attributes into vertex shader



Vertex Array Object (VAO)

- Holds the state that configures the vertex specification stage
  - Buffer object bindings
  - Vertex attribute mapping

Example Layout



Vertex Shader (`gl_VertexID == 5`):

```
layout(location = 0)
in vec3 position;

layout(location = 1)
in vec3 normal;

layout(location = 2)
in vec2 texcoords;

layout(location = 3)
in vec4 color;
```

Example Code

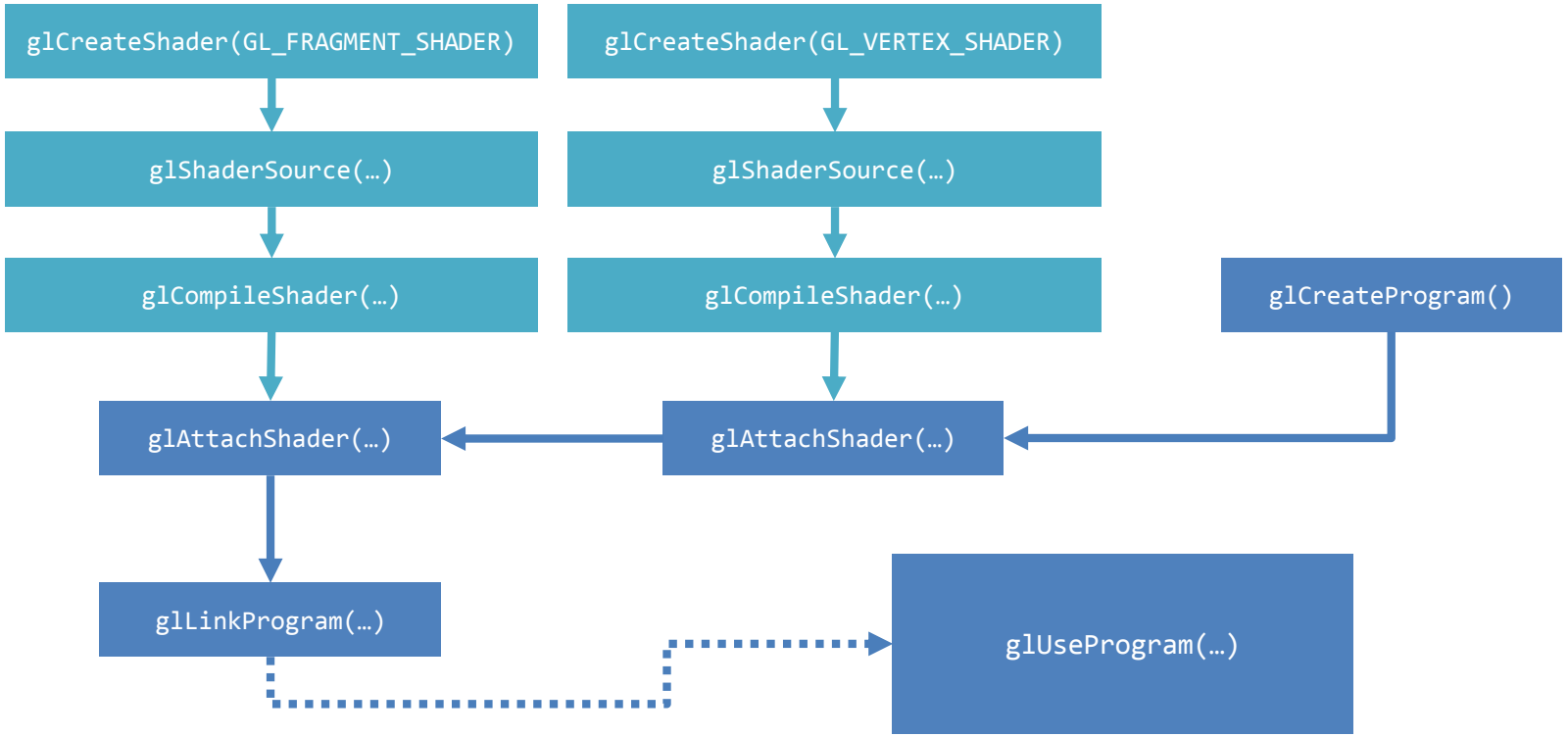
```
1   GLuint my_vao;
2   glGenVertexArrays(1, &my_vao);

3   glBindVertexArray(my_vao);
4   // configure layout of vertex attributes

5   glBindBuffer(GL_ARRAY_BUFFER, vbo1);
6   // attribute 0: position
7   glEnableVertexAttribArray(0);
8   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

9   glBindBuffer(GL_ARRAY_BUFFER, vbo2);
10  // attribute 3: color
11  glEnableVertexAttribArray(3);
12  glVertexAttribPointer(3, 4, GL_UNSIGNED_BYTE, GL_TRUE, 0, (void*)0);

13  glBindBuffer(GL_ARRAY_BUFFER, vbo3);
14  // attribute 1: surface normal
15  glEnableVertexAttribArray(1);
16  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 20, (void*)8);
17  // attribute 2: texture coordinates
18  glEnableVertexAttribArray(2);
19  glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 20, (void*)0);

20  // probably do something else…

21  glBindVertexArray(my_vao);
22  // render…

23  glDeleteVertexArrays(1, &my_vao);
```

Shaders

- Programs for programable parts of pipeline
- Shader objects
  - Parts of a pipeline (vertex shader, fragment shader, etc.)
  - Compiled from GLSL code
    - OpenGL Shading Language
    - C-like syntax
- Program object
  - A whole pipeline
  - Shader objects linked together

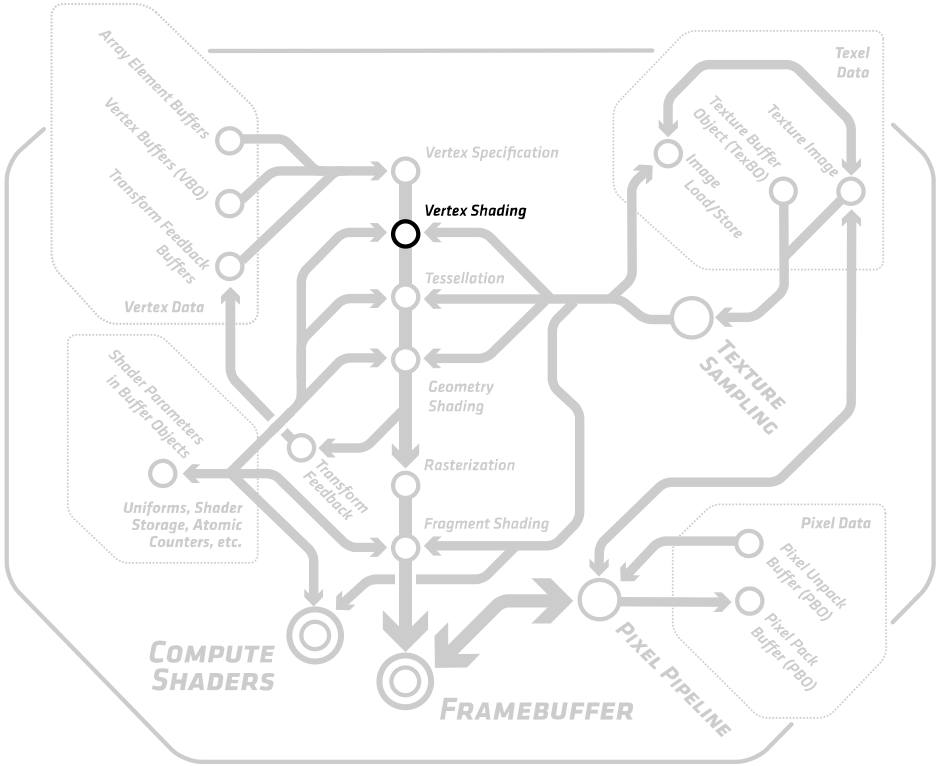
Anatomy of a GLSL Shader

```
1  #version 330

2  // uniforms
3  uniform vec4 some_uniform;

4  // inputs
5  layout(location = 0) in vec3 some_input;
6  layout(location = 1) in vec4 another_input;

7  // outputs
8  out vec4 some_output;

9  void main()
10 {
11     //…
12 }
```


glCreateShader(GL_FRAGMENT_SHADER)
glCreateShader(GL_VERTEX_SHADER)
glShaderSource(…)
glShaderSource(…)
glCompileShader(…)
glCompileShader(…)
glCreateProgram()
glAttachShader(…)
glAttachShader(…)
glLinkProgram(…)
glUseProgram(…)

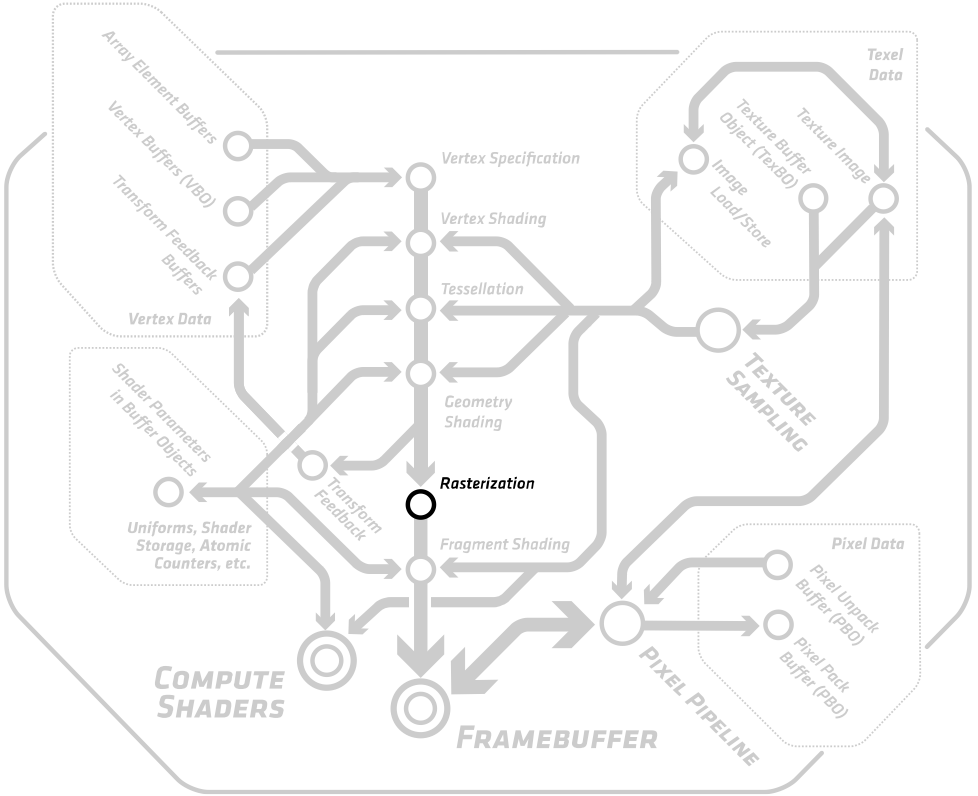
Vertex Shader

- Processes each vertex
- Input: vertex attributes
- Output: vertex attributes
  - mandatory: gl_Position



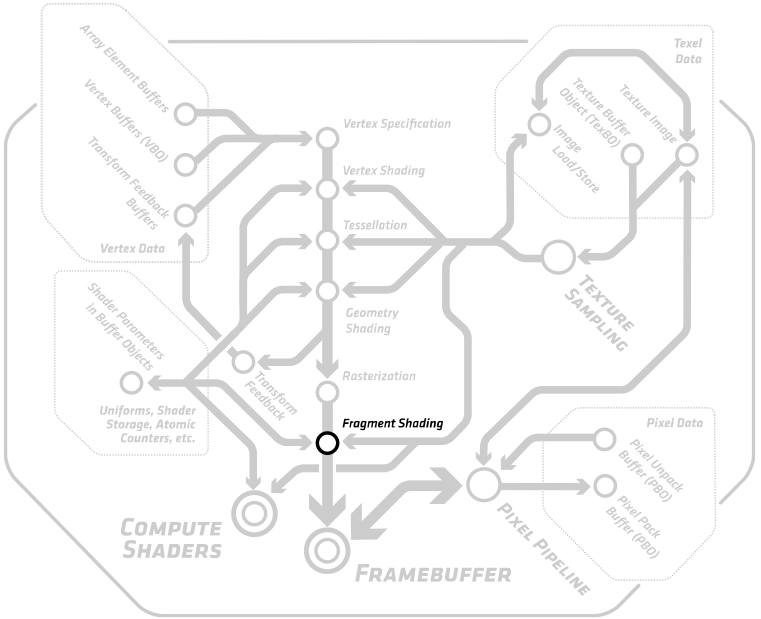
Rasterizer

- Fixed-function
- Rasterizes primitives
- Input: primitives
  - Vertex attributes
- Output: fragments
  - Interpolated vertex attributes



Fragment Shader

- Processes each fragment
- Input: interpolated vertex attributes
- Output: fragment color



Built-in Variables

- Interface to fixed-function parts of pipeline
  - E. g. vertex shader:
    - `in int gl_VertexID;`
    - `out vec4 gl_Position;`
  - E. g. fragment shader:
    - `in vec4 gl_FragCoord;`
    - `out float gl_FragDepth;`

Example: Vertex Shader

```
1  #version 330

2  layout(location = 0) in vec3 vertex_position;
3  layout(location = 1) in vec4 vertex_color;

4  out vec4 color;

5  void main()
6  {
7      gl_Position = vec4(vertex_position, 1.0f);
8      color = vertex_color;
9  }
```

Example: Fragment Shader

```
1  #version 330
2  in vec4 color;
3  layout(location = 0) out vec4 fragment_color;
4  void main()
5  {
6      fragment_color = color;
7  }
```

Fragment Merging