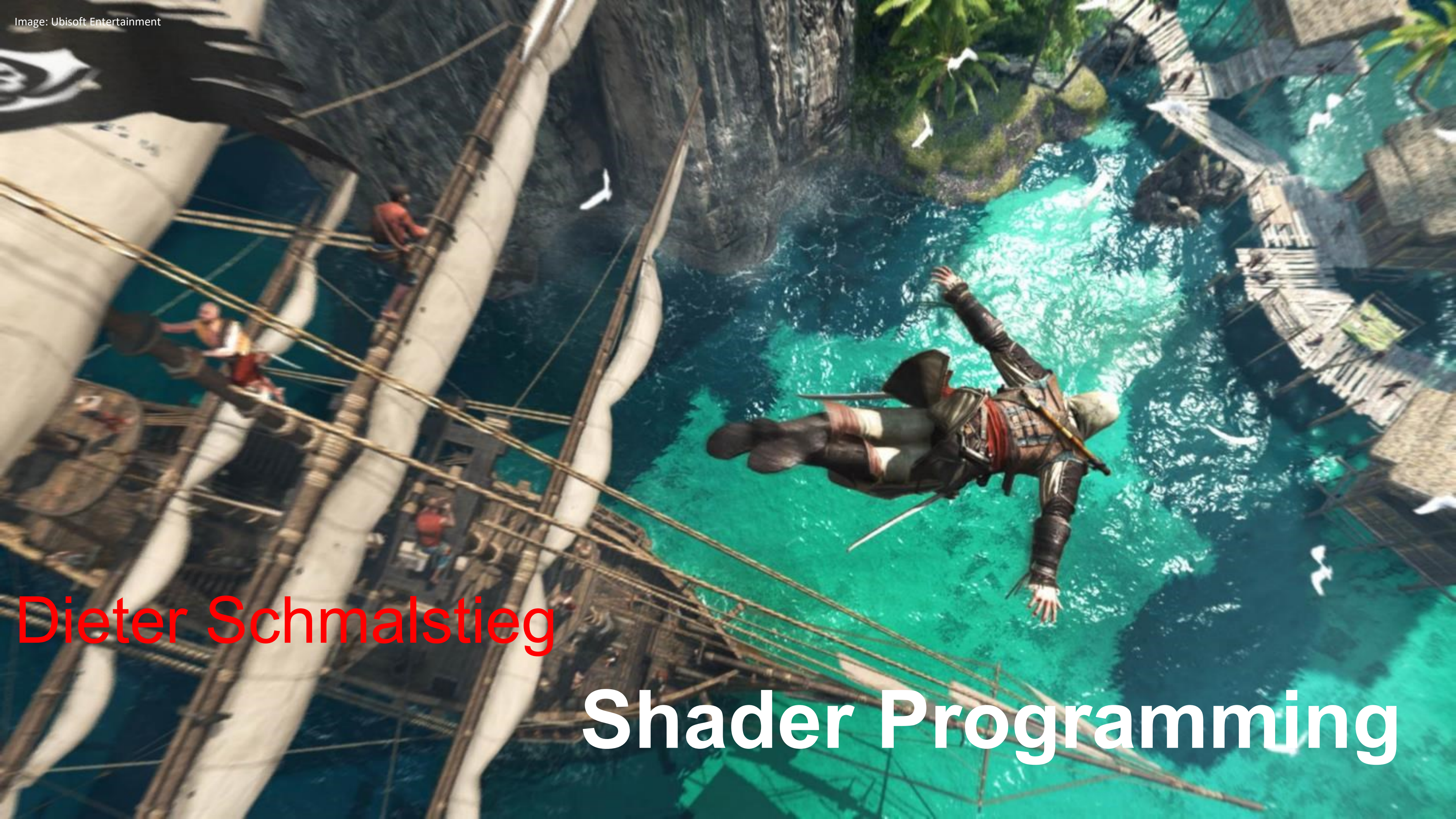


Image: Ubisoft Entertainment

Shader Programming

Dieter Schmalstieg

Today's Agenda

- The GPU execution model
- How to write shader programs

Graphics Application Programmer Interface

- Hardware-vendor independent interface
  - Interface is hardware independent
  - But implementation is hardware dependent
- Defines
  - Abstract rendering device
  - Set of functions to operate the device

API and Vendor Overview

Graphics API

- DirectX (Microsoft)
- OpenGL
- OpenGL ES
- Vulkan (Khronos group)
- Metal (Apple)

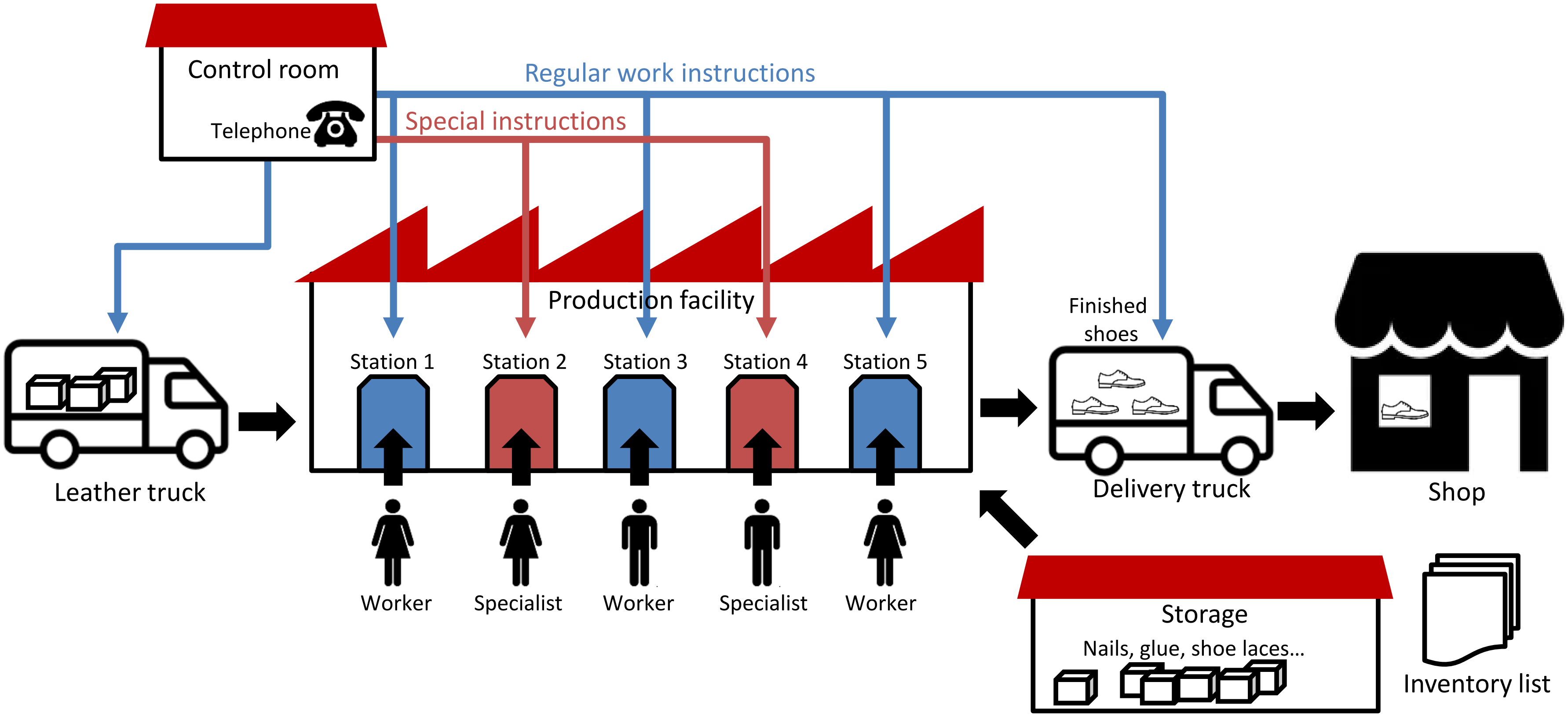
GPU Hardware Vendors

- Intel (Iris, X^e)
- NVIDIA (GeForce)
- AMD (Radeon)
- Qualcomm (Adreno)
- Imagination (PowerVR)
- Apple
- ARM (Mali, only design)

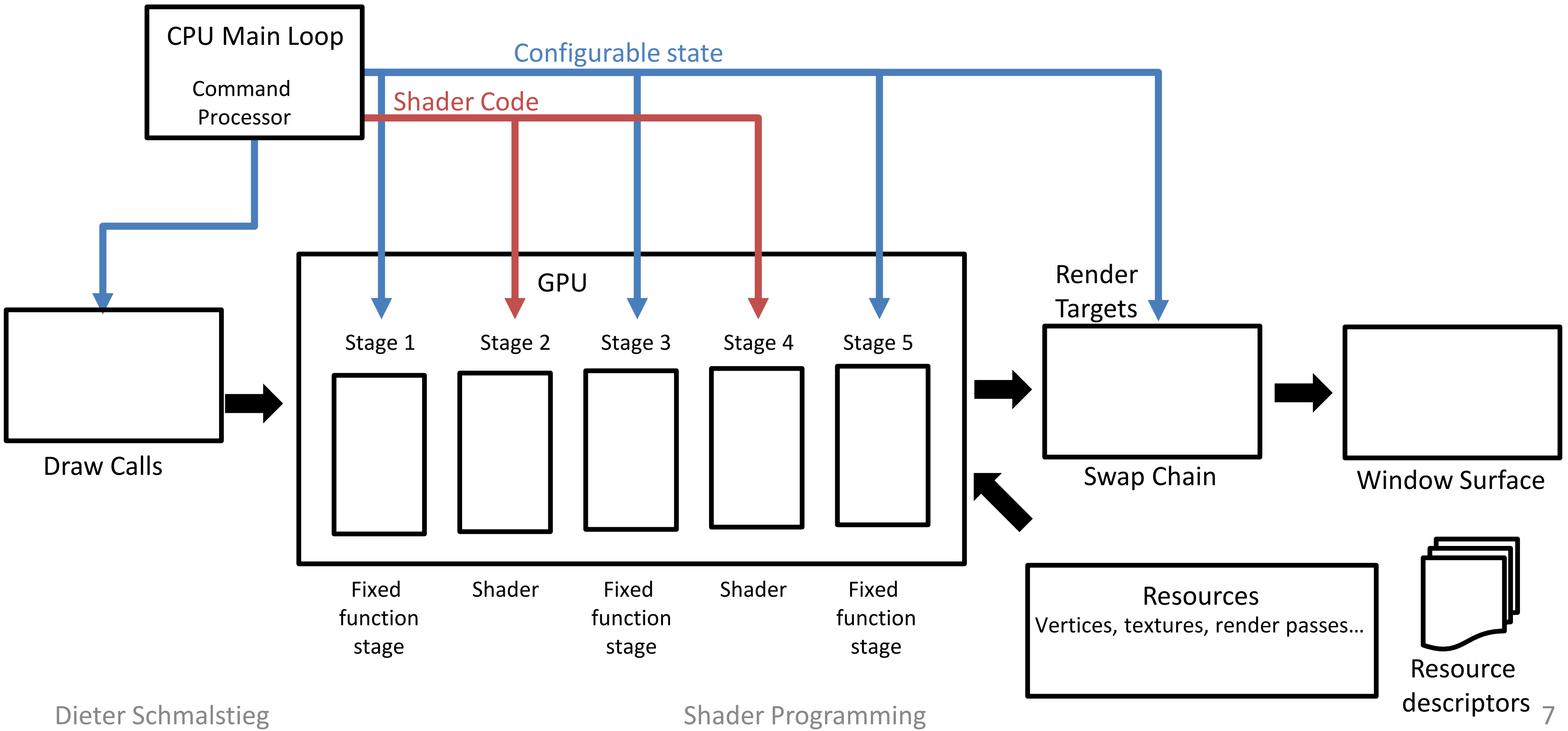
A Story About GPU Programming

- Imagine you want to produce shoes
- CPU, single-threaded
  - The master shoemaker is alone in the shop
  - One task is done after the other
- CPU, multi-threaded
  - The master shoemaker has a few apprentices
  - They talk directly to one another to split the work
- GPU: 10000 worker threads…?

A Big Shoe Factory



A Big Graphics API

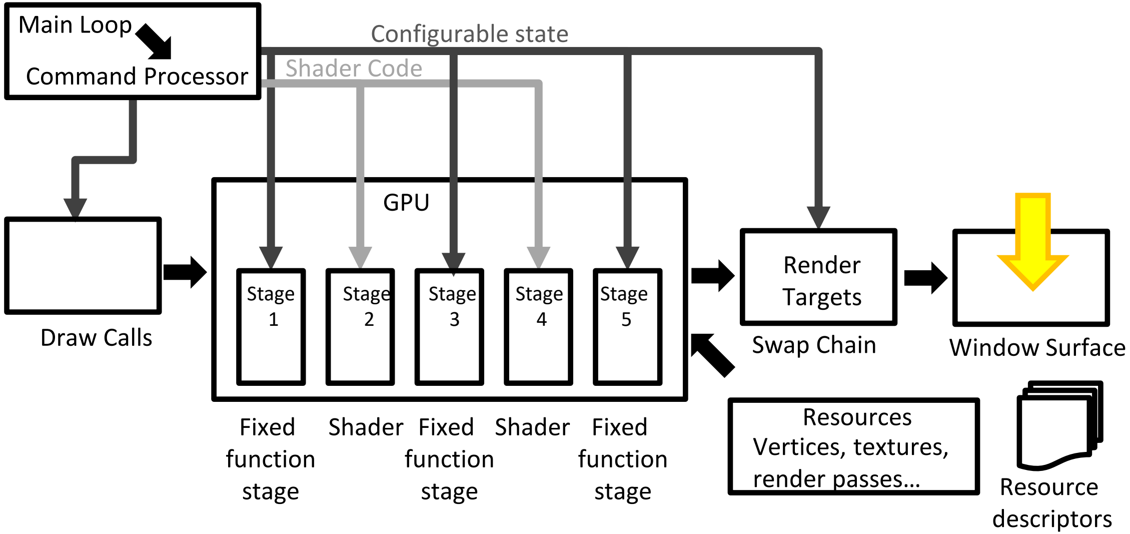


Initialization

- Initialize the graphics system
- Get the "command processor" object
  - Context (OpenGL) or command queue (Vulkan) – see later slides
- Select a render device
  - Which physical GPU
    - If there are multiple
  - Which features are needed
    - Number of viewports, single or double float…
  - Which operation modes are needed
    - Graphics, compute, memory transfer

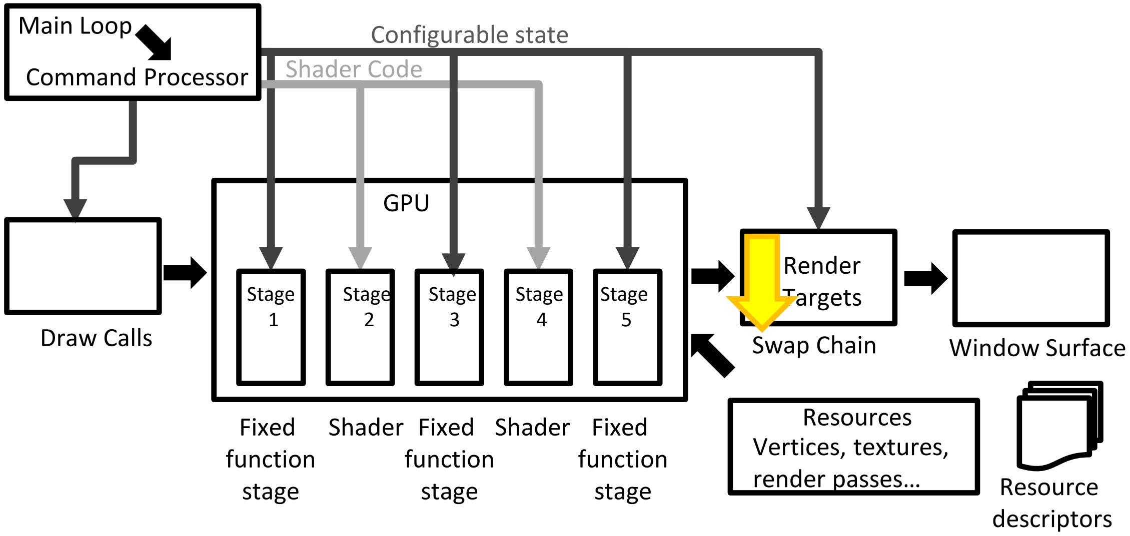
Surface

- Object connected to window or entire screen
- Requires a platform-specific window manager library (e.g., GLFW)



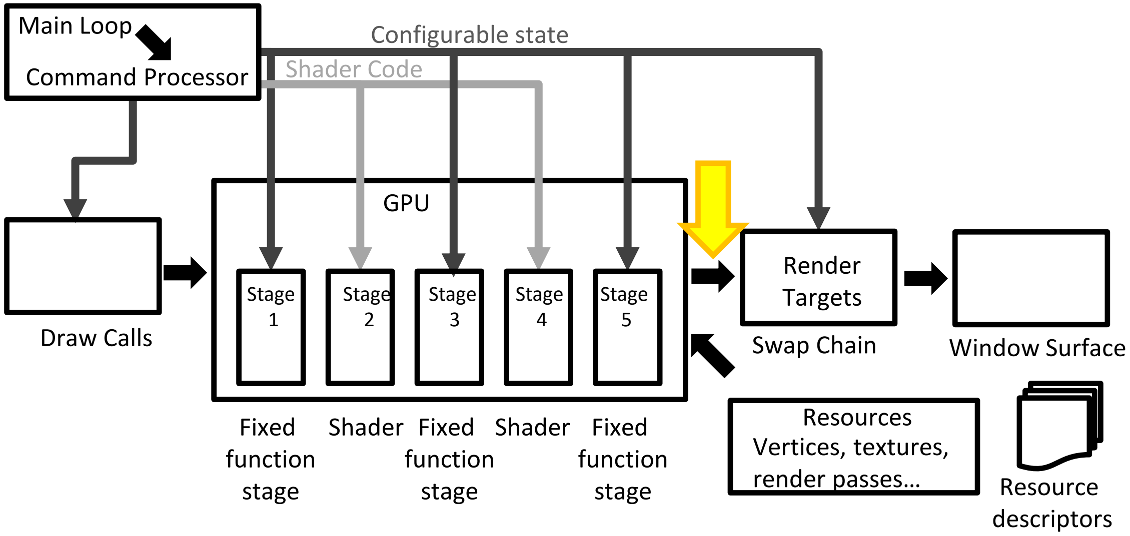
Swap Chain

- Contains render targets (images) waiting to be shown
- Single, double, triple buffering

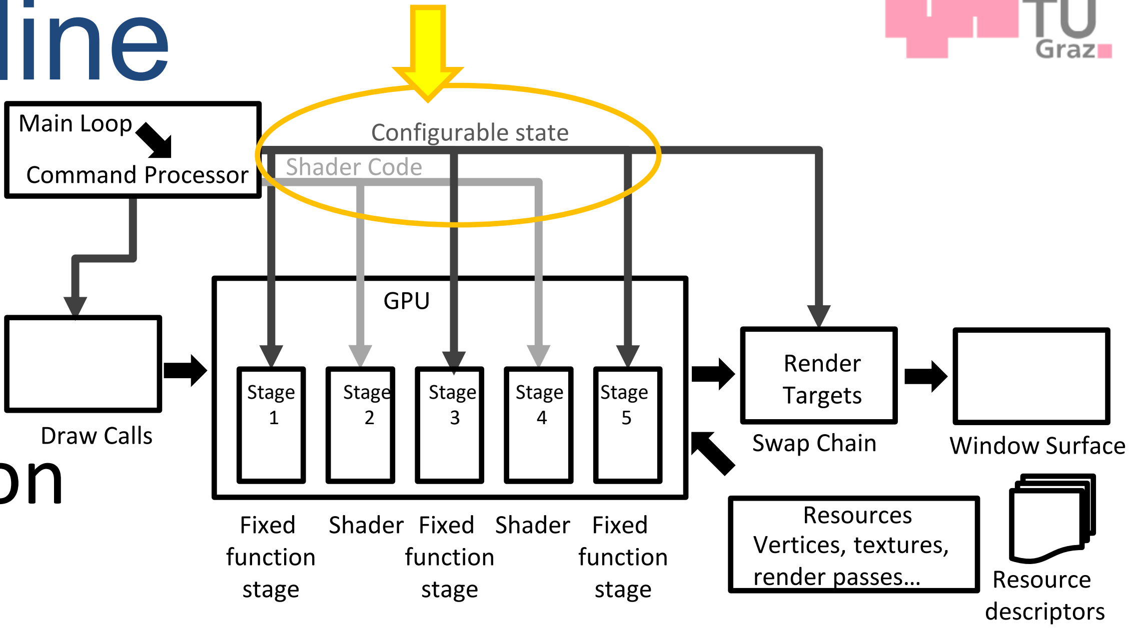


Framebuffer

- Points to render targets for color, depth, stencil

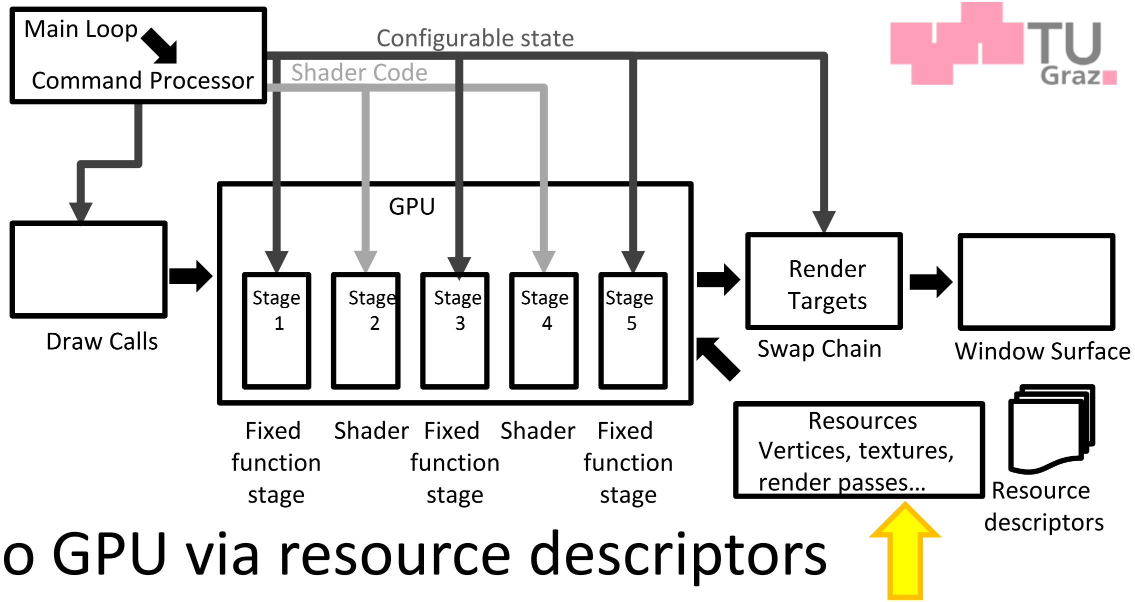


Graphics Pipeline



- Pipeline stores specific configuration of render state
- Configurable state: viewport size, depth buffer etc.
- Programmable state: shader programs

Resources

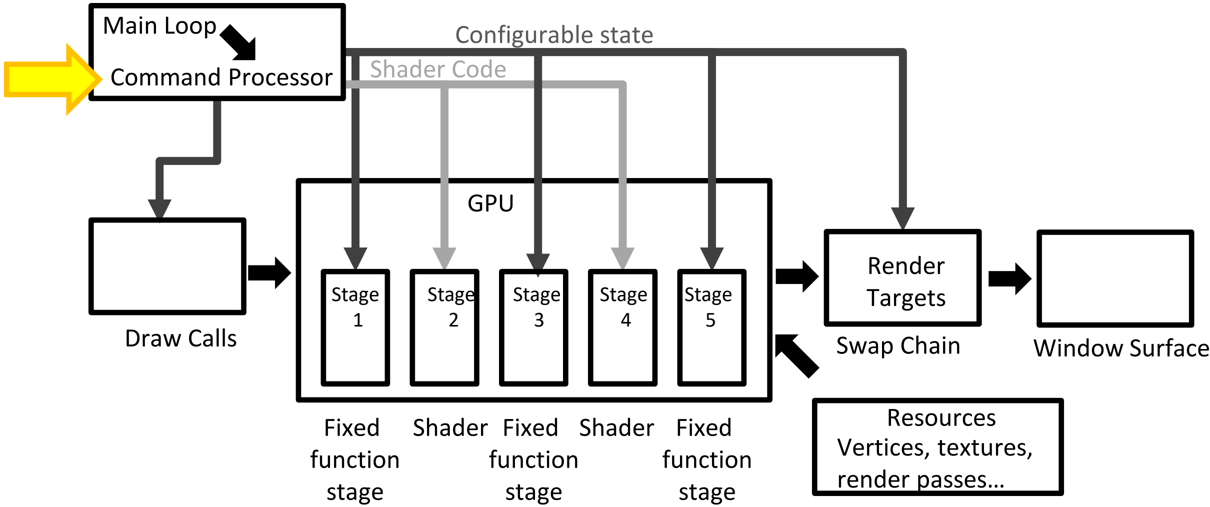


- Data stored in GPU local memory
- E.g, vertices, textures, other buffers
- Resource are described to GPU via resource descriptors
- Before use
  - Generate or download resource
  - Specify resource descriptors
  - Bind (=activate) the chosen resource
- Configurable state has “slots” for binding various resource types

Command Processor

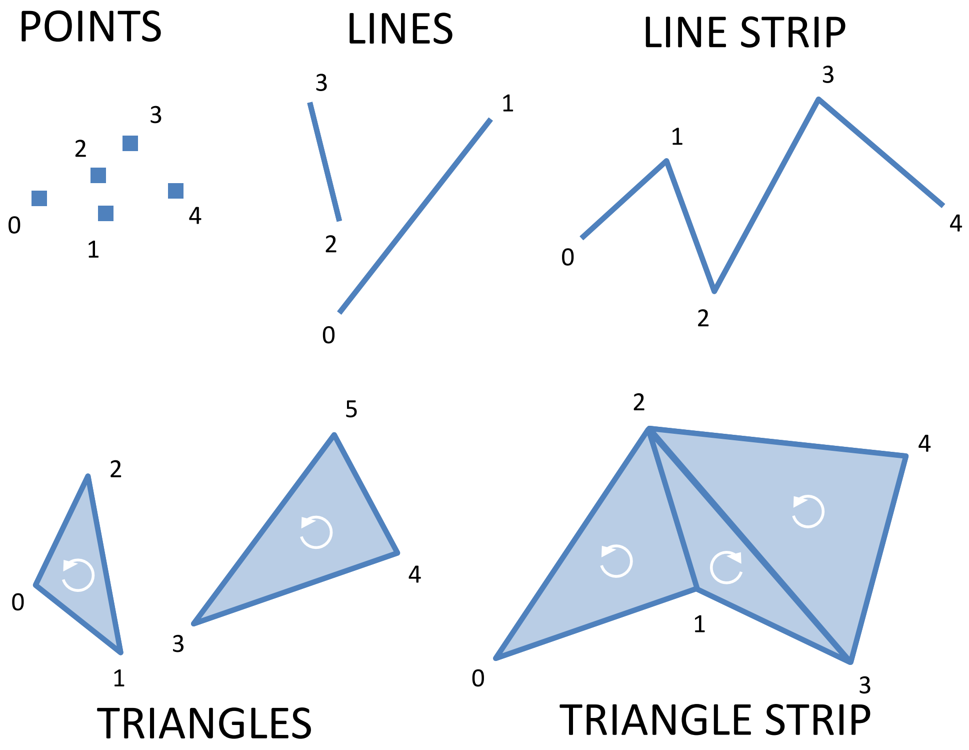
Example command sequence

- Begin render pass
- Bind pipeline (Vulkan) or pipeline components (OpenGL)
- Bind framebuffer
- Draw vertices (=draw call)
- End render pass



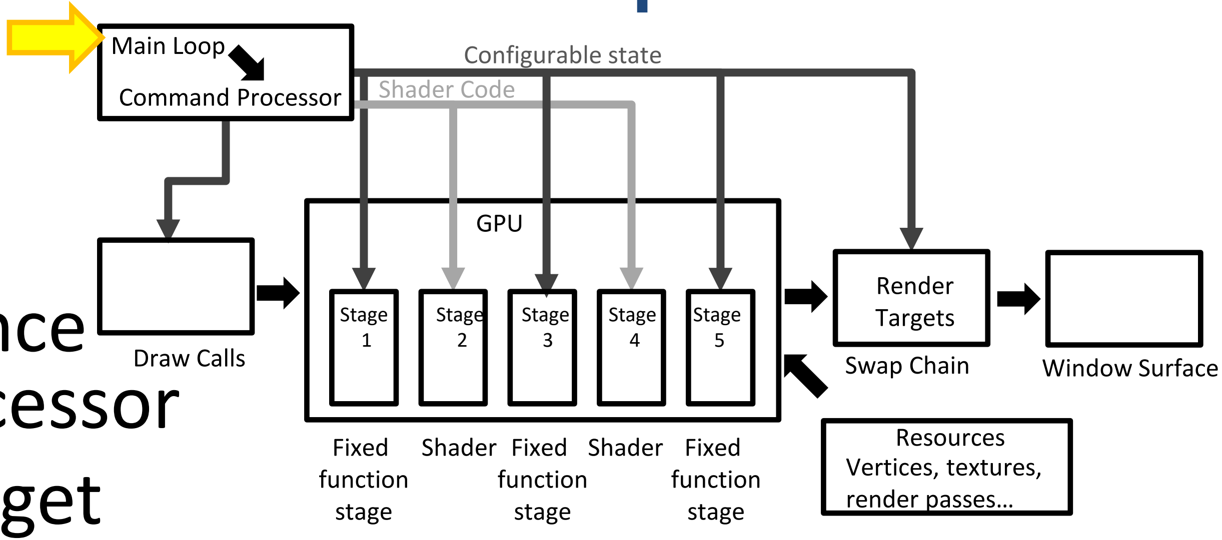
Draw Calls

- Specify
  - Which primitive type
  - Which vertex buffer
  - Start index in buffer
  - Stop index in buffer



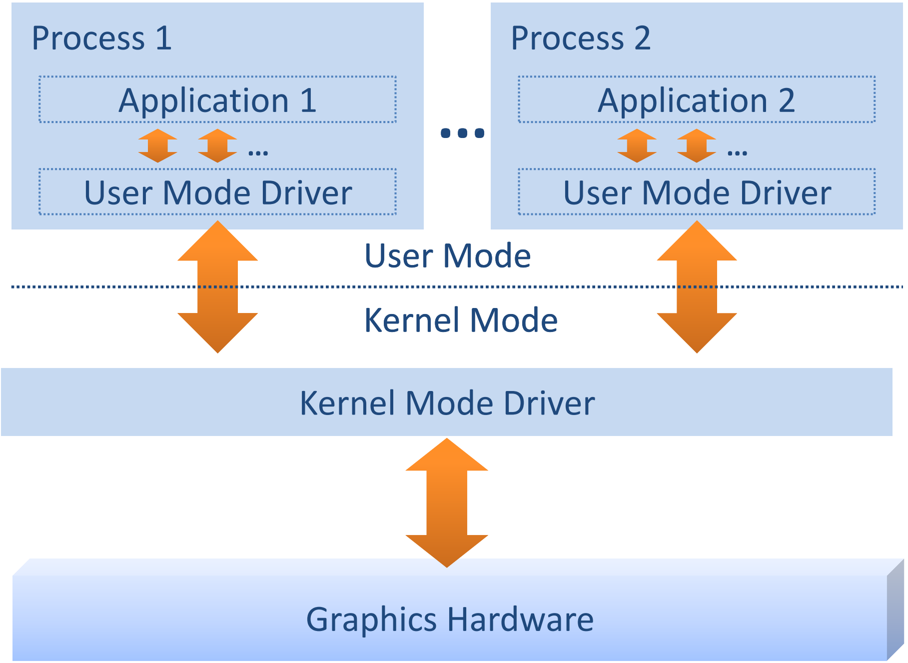
CPU Main Loop

- Get render target from swap chain
- Submit chosen command sequence to command processor
- Return render target to swap chain
- Command submission is very different between traditional and modern API



Graphics Driver Architecture

- User mode graphics driver
  - Minimize number of mode switches
  - Translation of graphics commands to instructions for the hardware
  - Batching, optimization, validation
  - Fine grained memory management
- Kernel mode graphics driver
  - Schedule access to hardware
  - Microkernel pattern, stability
  - Coarse grained memory management
  - Submits command buffers to GPU



Traditional API

- **Graphics context**: the system object in a traditional API
  - OpenGL, DirectX11 and below
  - Represents a virtual GPU
  - One process can have multiple contexts
  - Multiple contexts can share resources
- **Current context** for a given process
  - One to one mapping
    - Maximum of one current context per thread
    - Current context only assigned to one thread at the same time
  - All OpenGL operations work on current context

Pipeline State

- Inside a context, one must use commands to configure GPU **pipeline state** before rendering
- Pipeline state consists of
  – Programmable state (shaders)
  – Configurable state: blend, depth, culling, etc.
  – Layout: how to map settings at each stage’s shader

Problems with Traditional API

- Pipeline execution is largely asynchronous
  - CPU sends commands into a “black hole” and
  - CPU does not know exactly when commands are executed
- Configuration of state can only be done incrementally
  - Submitting configuration changes to driver requires immediate validation, conversion, buffering → high cost at runtime
  - Drivers must have per-game optimizations built in
- Pipeline state *not* made explicit in rendering context
  - Switching pipeline configurations is cumbersome
  - Must be done by sequences of state change commands
- Pipeline abstraction is single-threaded on CPU
  - Cannot multi-thread feeding the pipeline
  - Having too many draw calls becomes a bottleneck

Modern API

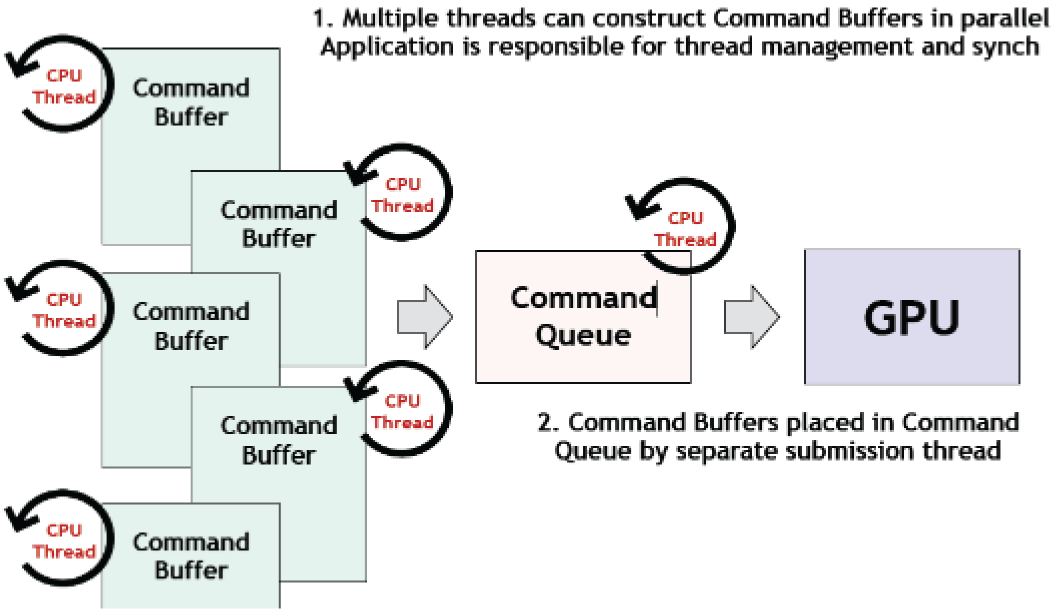
- DirectX 12, Vulkan (=OpenGL successor)
- Designed for modern GPU types (including mobile)
- Make (CPU driver side of) pipeline more programmable
- CPU multi-threading possible (at your own risk)
- Split rendering context into command buffers and queues
- Make pipeline state (and render passes) explicit
- Low overhead
- Fine-grained control (minimal program 1K lines of code)

Command Buffers

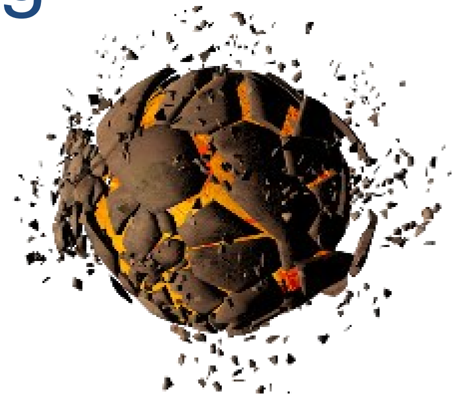
- Commands collected in command buffers
  - Optimize and validate command buffers during building, not during submission
  - Yields *immutable*, re-useable pipeline state configurations
  - Selected pipeline state variables can be declared *mutable*
- Command buffer submitted to *queue* for execution
- Build *many* command buffers from *many* threads
  - When the buffers are ready, one can submit them all at once
- Each command buffer just *switches* to its favorite pipeline
- Can use *synchronization* primitives across command buffers
  - Event, barrier, semaphore, fence

Queues

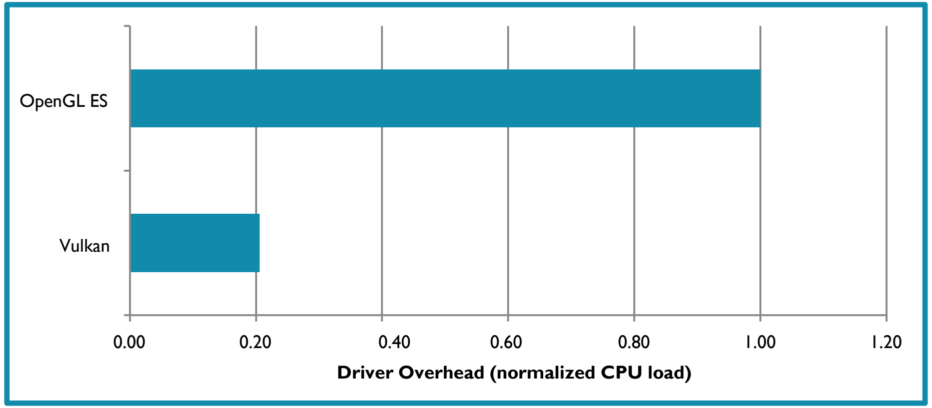
- Queues replace traditional contexts
- Insert command buffer into queue to schedule it



Example: Multi-Threaded Drawing

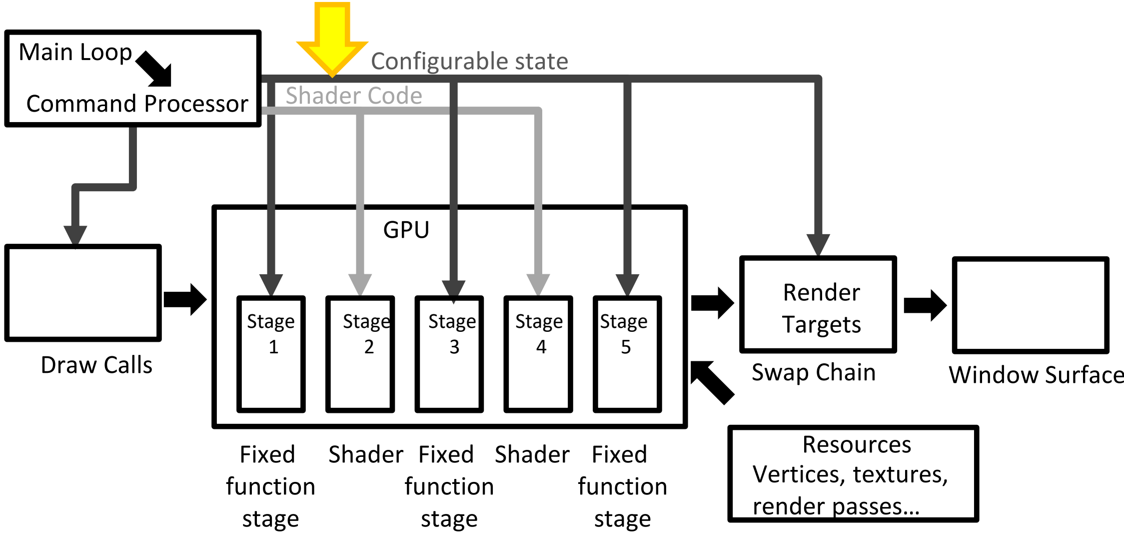


- ARM Vulkan Benchmark
  - ARM Cortex A-15/7, Mali T-628 MP6
  - 1000 meshes, 3 materials
  - 79% less CPU



Shader Programs

- For the programable parts of the pipeline
- Compiled from shader language
  – HLSL, GLSL
- C-like syntax
- Stream execution model:
  – Shaders are like callbacks, no "main loop" code



Language Elements

- Skalar data types: float, int, bool…
- Vector/matrix data types: vec2, vec3, vec4, ivec3, mat4…
- Struct, arrays (static size)
- Texture sampler: sample2D, sampler3D, …
- Control flow: if, else, while, for
- Function calls (no recursion)
- Swizzle operators: color1.rgb = color2.bga
- Mask operator: pos1.z = pos2.x + pos2.y

Build-In Functions

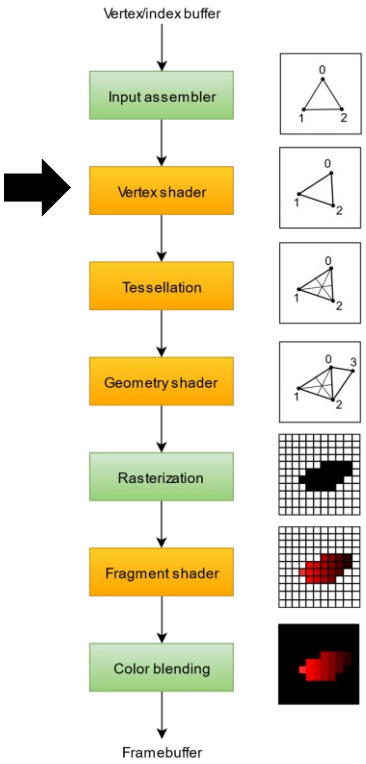
- Trigonometric
  - sin(), cos(), radians(), …
- Logarithm, exponentiation
  - log(), sqrt(), …
- Other
  - min(), max(), mod(), floor(), abs(), …
- Geometric
  - distance(), normalize(), dot(), length(), …
- Special functions
  - Linear interpolation
  - Reflection vector
  - Refraction vector

Shader Compilation

- Compilation separated into front-end/back-end
- Front-end: HLSL, GLSL, OpenCL, etc.
- Back-end = Bytecode
  - Microsoft HLSL uses proprietary bytecoe
  - Vulkan uses SPIR-V (standard portable intermediate representation)
- Applications ship with bytecode, not shader source
- Just-in-time compilation of bytecode
  - To hardware target platform (NVIDIA, AMD, ...)

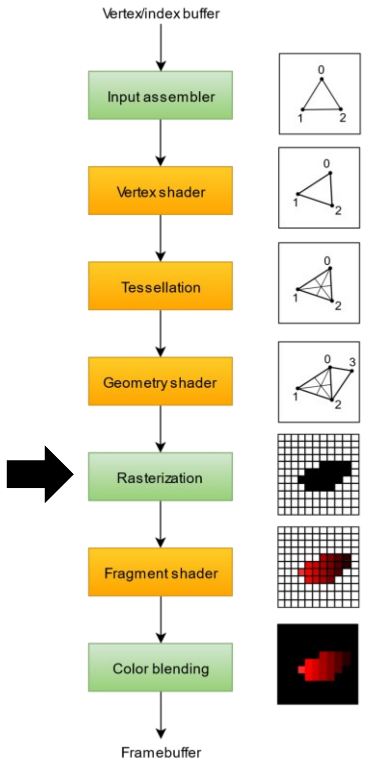
Vertex Shader

- Processes each vertex
- Input: vertex attributes
- Output: vertex attributes
- Mandatory output: gl_Position



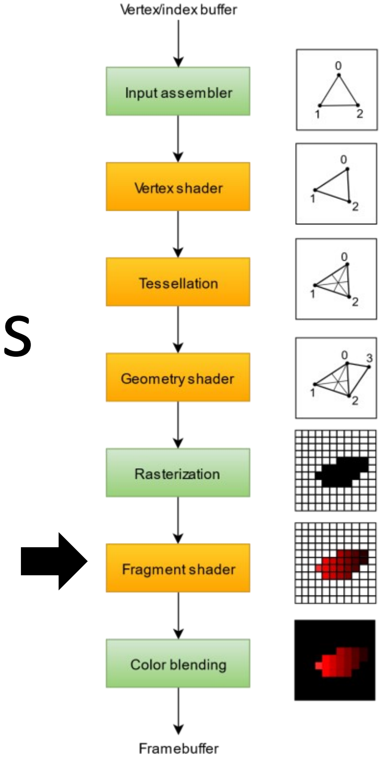
Rasterizer

- Generates fragments covering a primitive
- Fixed-function unit
- Input: primitives, vertex attributes
- Output: fragments with interpolated vertex attributes

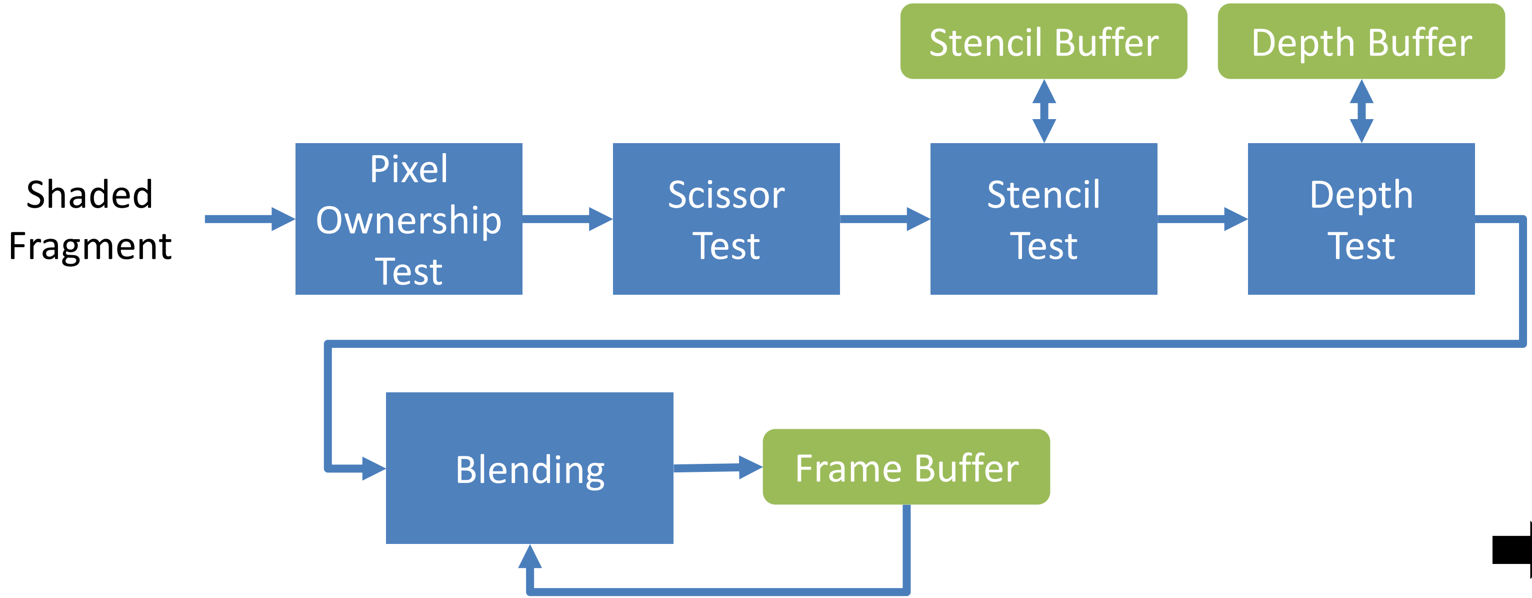


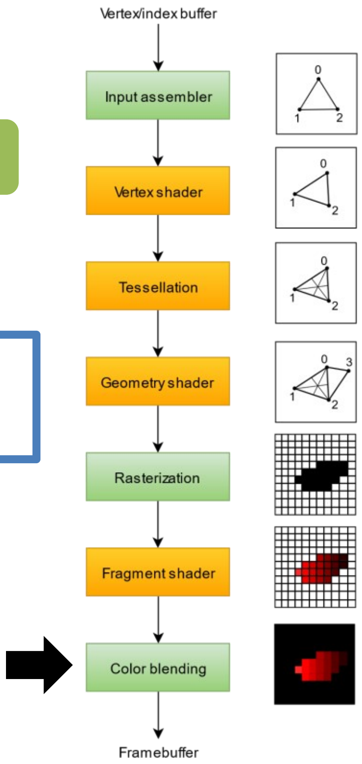
Fragment Shader

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



Fragment Merging





Anatomy of a GLSL Shader

```glsl
#version 330
// uniform inputs are constant for all shader invocations
uniform vec4 some_uniform;
// inputs are varying with each shader invocation
layout(location = 0) in vec3 some_input;
layout(location = 1) in vec4 another_input;
// outputs
out vec4 some_output;
void main()
{
    //…
}
```

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

```
#version 330
uniform mat4 mvp_matrix; // model-view-projection matrix
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec4 vertex_color;
out vec4 color;
void main()
{
    gl_Position = mvp_matrix * vec4(vertex_position, 1.0f);
    color = vertex_color;
}
```

Example: Fragment Shader

```
#version 330
layout(location = 0) in vec4 color; // interpolated color
out vec4 fragment_color;
void main()
{
    fragment_color = color;
}
```

Thank You!

Questions ?