Image: DICE

Dieter Schmalstieg

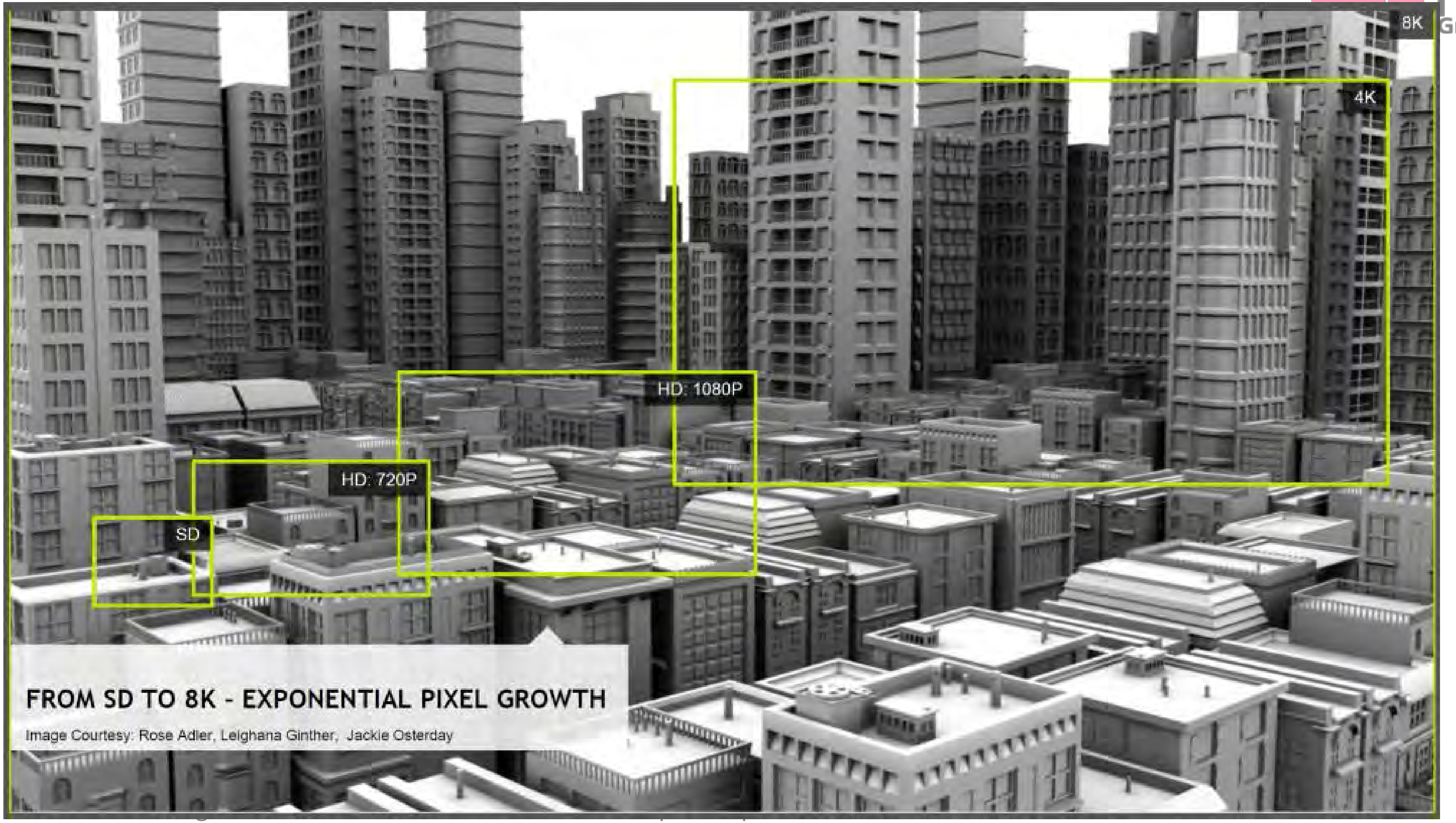
The Graphics Pipeline

What do we want?

- Computer-generated imagery (CGI) of complex 3D scenes in real-time
- Computationally extremely demanding
  - Full HD at 60 Hz:

$$1920 \times 1080 \times 60\ \mathrm{Hz} = 124\ \mathrm{Mpx/s}$$

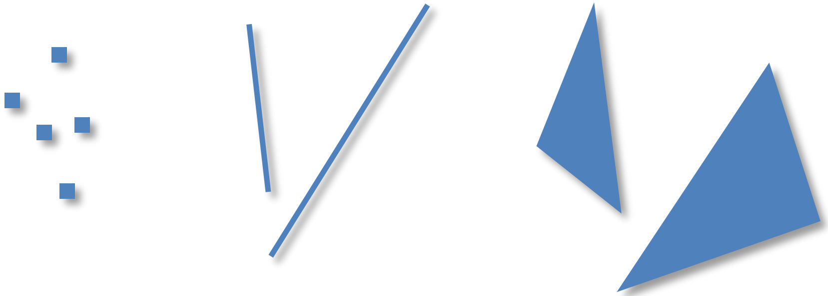
  - And that's just output data!
- Requires specialized hardware


8K
4K
HD: 1080P
HD: 720P
SD
FROM SD TO 8K - EXPONENTIAL PIXEL GROWTH
Image Courtesy: Rose Adler, Leighana Ginther, Jackie Osterday

Solution

Most of real-time graphics is based on

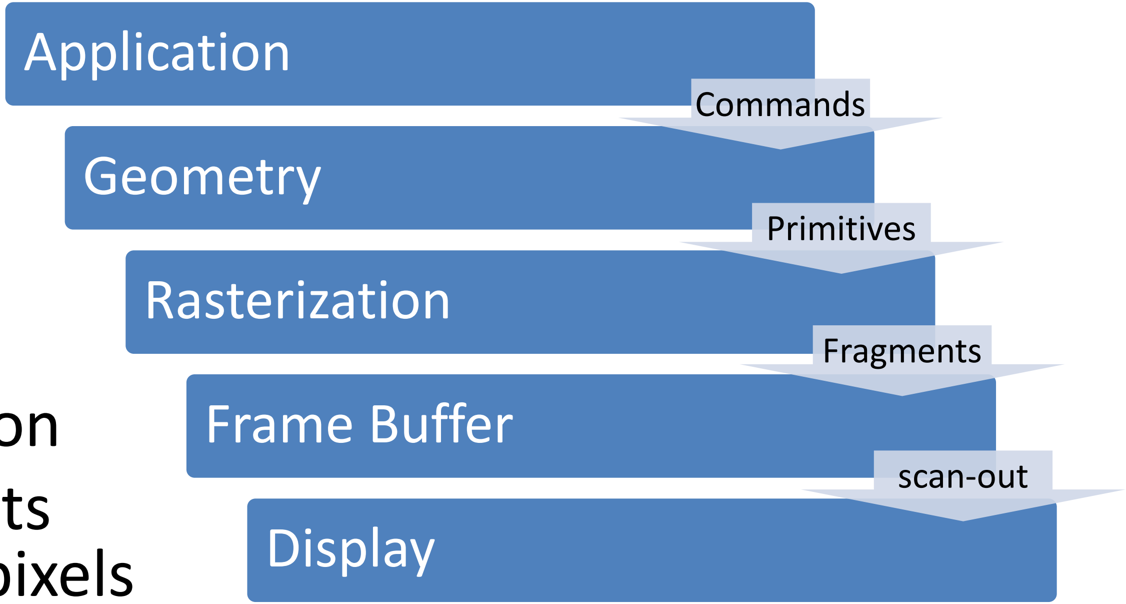
- Rasterization of graphic *primitives*
  - Points
  - Lines
  - Triangles
  - …



- Implemented in hardware
  - *Graphics processing unit* (GPU)

The Graphics Pipeline

- High-level view:
- "Fragment":
  - Sample produced during rasterization
  - Multiple fragments are *merged* into pixels



Application Stage

- Generate database
  - Usually only once
  - Load from disk or generate algorithmically
  - Build acceleration structures (hierarchy, …)
- Repeat main loop
  - Input event handlers
  - Simulation → modify data structures
  - Database traversal → issue **graphics commands**
- Until exit

Graphics Commands

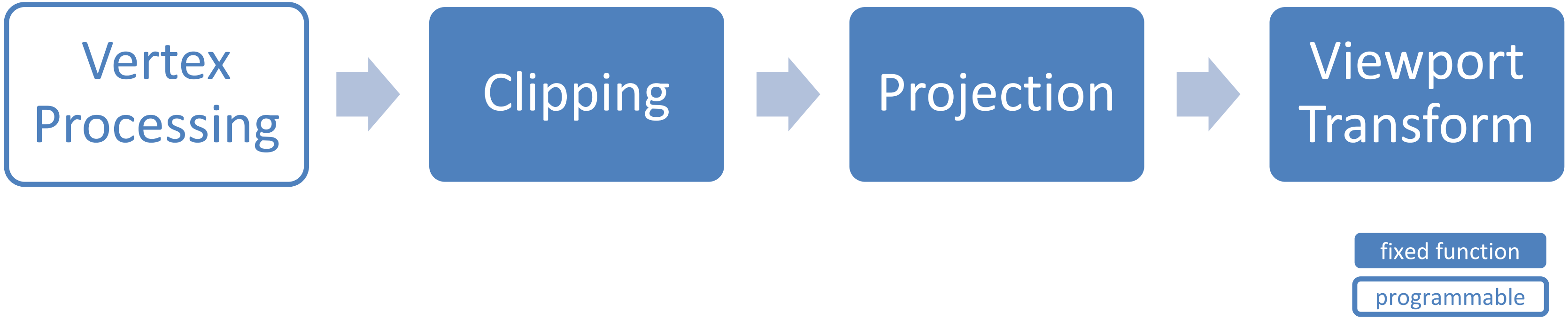
- Graphics command stream from CPU to GPU
  - Specify primitives
  - Manage resources
  - Modify GPU state

Graphics Driver

- Graphics hardware is shared resource
- Large user mode graphics driver
  - Prepares command buffers
- Graphics kernel subsystem
  - Schedule access to hardware
- Small kernel mode graphics driver
  - Submit command buffers to hardware

This is were new APIs (Apple Metal, AMD Mantle, Khronos Vulkan) try to be more efficient

Geometry Stage



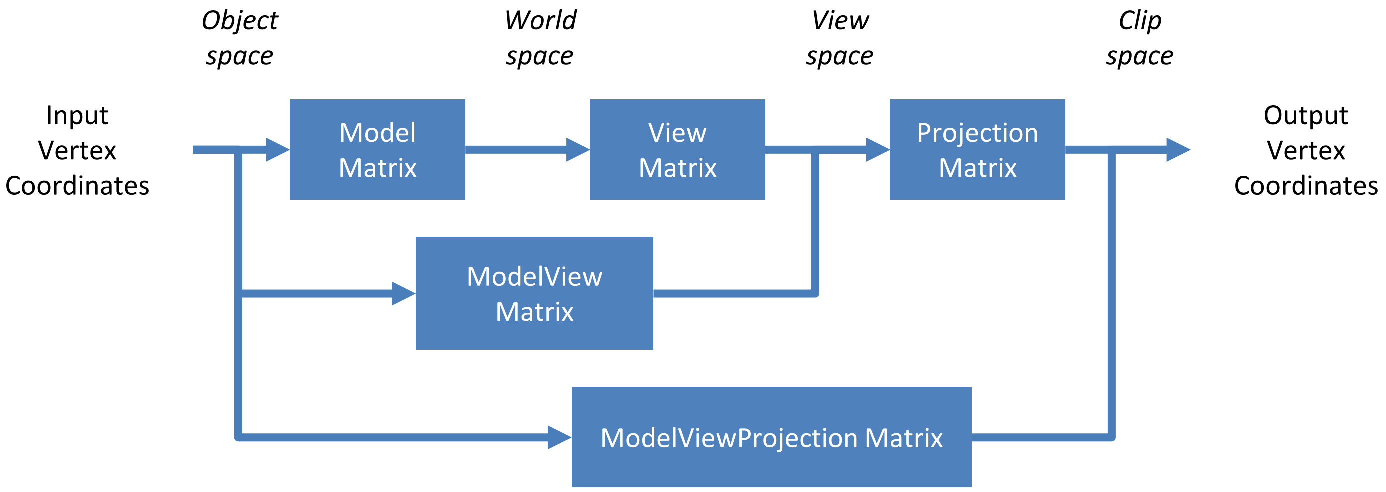
Vertex Processing

- Input vertex stream
  - Composed of arbitrary *vertex attributes* (position, color, …)
- Is transformed into stream of vertices mapped onto the screen
  - Composed of their *clip space* coordinates and additional user-defined attributes (color, texture coordinates, …)
  - Clip space: homogeneous coordinates
- By the *vertex shader*
  - GPU program that implements this mapping
  - Historically, "shaders" were small programs performing lighting calculations

Object space vertices Screen space vertices

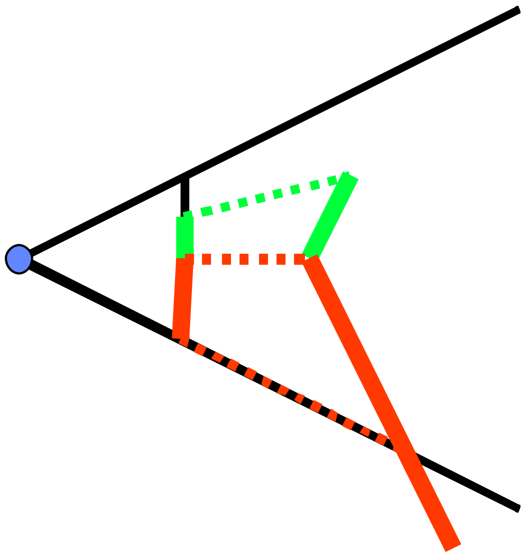
Vertex Coordinate Transformation

Common model in rasterization-based 3D graphics



Geometry Stage Tasks

- Clipping
  – Primitives not entirely in view are clipped to avoid projection errors
- Projection
  – Projects clip space coordinates to the image plane
  → Primitives in *normalized device coordinates*
- Viewport Transform
  – Maps resolution-independent normalized device coordinates to a rectangular window in the frame buffer, the *viewport*
  → Primitives in window (pixel) coordinates



Rasterization Stage

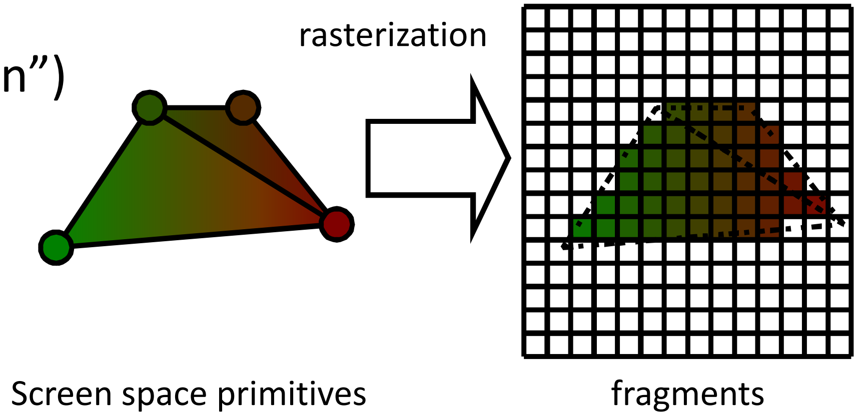
Primitive Assembly → Primitive Traversal → Fragment Shading → Fragment Merging

fixed function

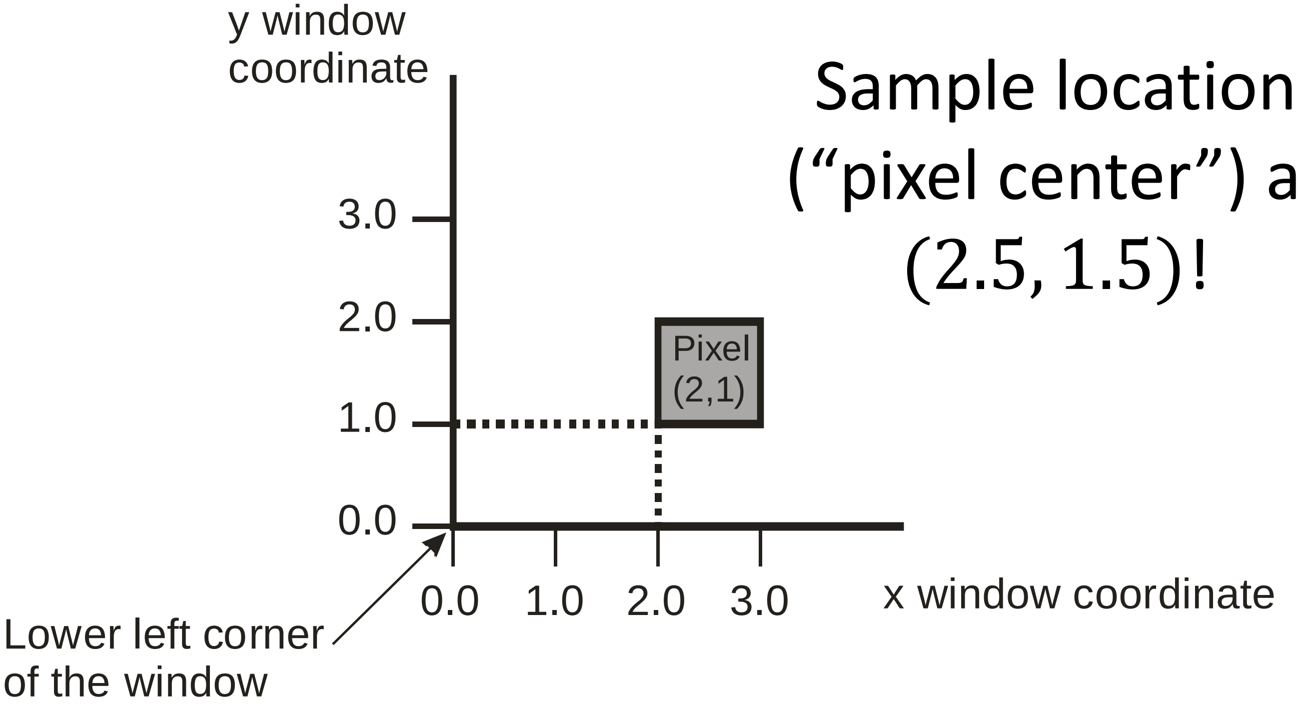
programmable

Rasterization Stage Tasks

- Primitive assembly
  - Backface culling
  - Setup primitive for traversal
- Primitive traversal ("scan conversion")
  - Sampling (triangle → fragments)
  - Interpolation of vertex attributes (depth, color, …)
- Fragment shading
  - Compute fragment colors
- Fragment merging
  - Compute pixel colors from fragments
  - Depth test, blending, …



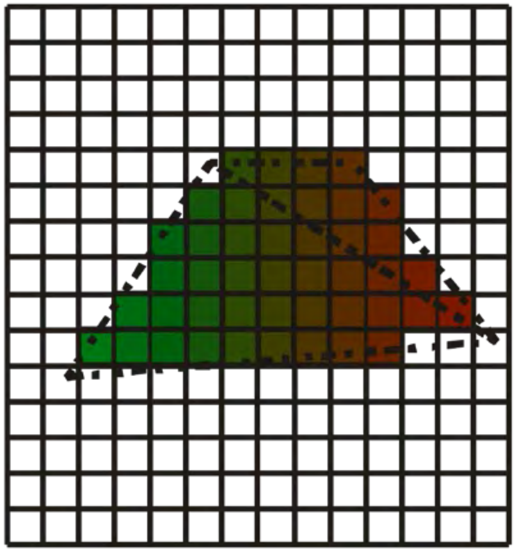
Rasterization – Coordinates



Sample location ("pixel center") at $(2.5, 1.5)$!

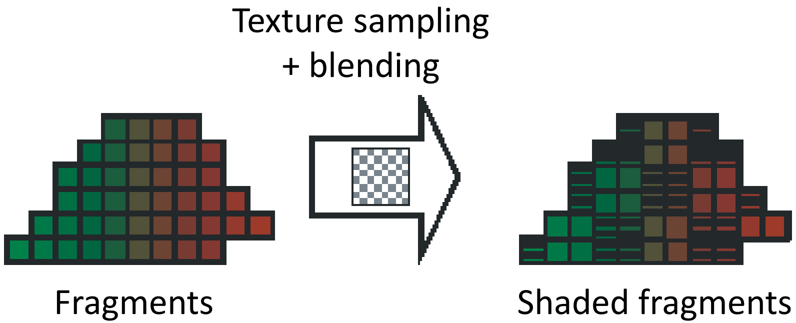
Rasterization – Rules

- Different rules apply for each primitive type
  - “Fill convention”
- Non-ambiguous!
  - Avoids artifacts
- Polygons
  - Pixel center contained in polygon
  - Pixels on edge: only one rasterized



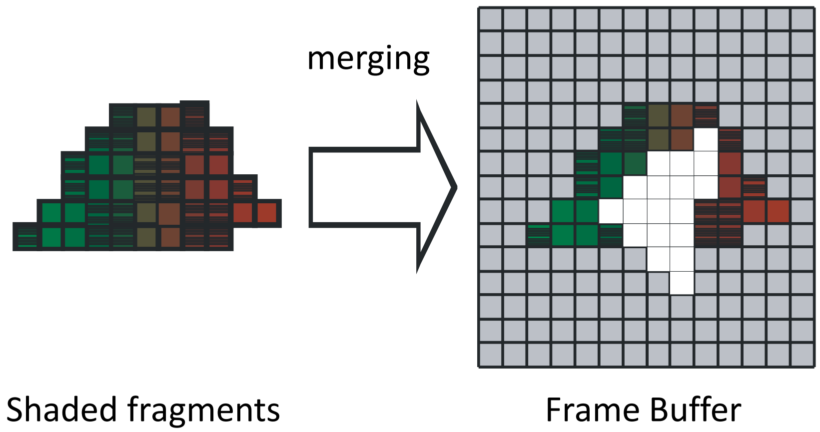
Fragment Shading

- Given the interpolated vertex attributes
  – Output by the vertex shader
- The *fragment shader* computes color values for each fragment
  – Apply textures
  – Lighting calculations
  – …

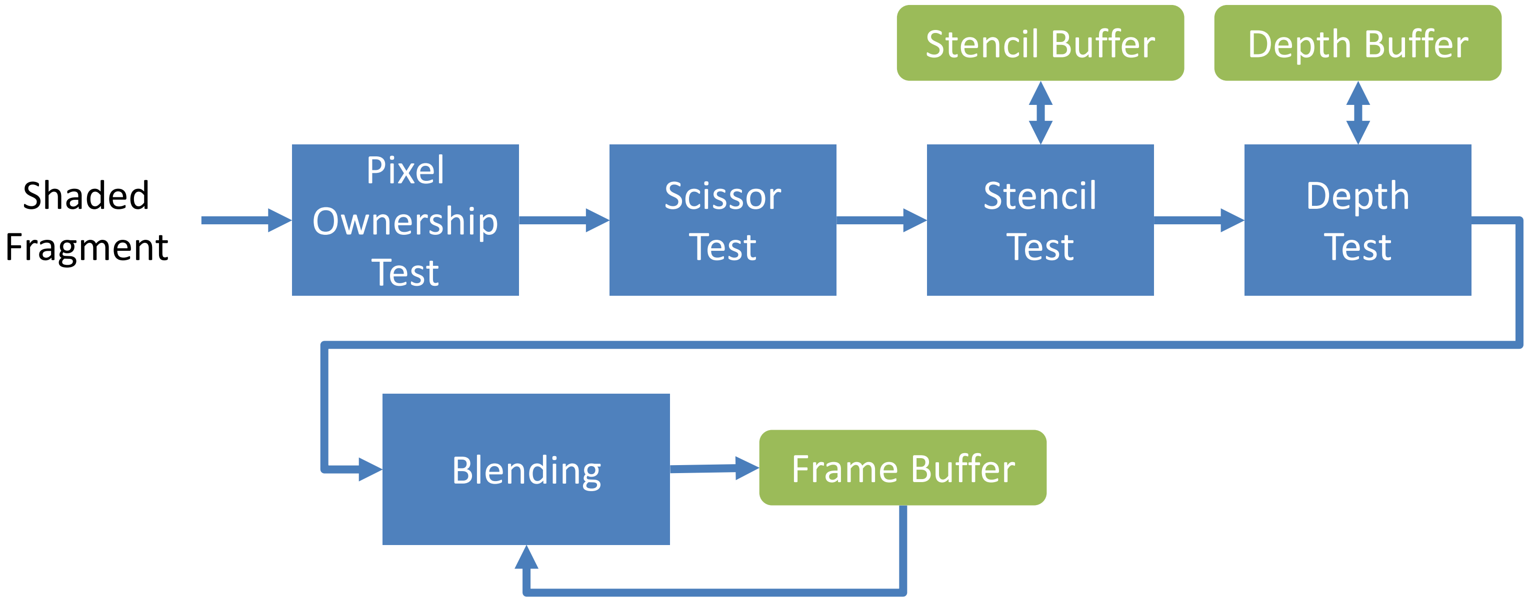


Fragment Merging

- Multiple primitives can cover the same pixel
- Their fragments need to be composed to form the final pixel values
  - Blending
  - Resolve visibility via depth buffering

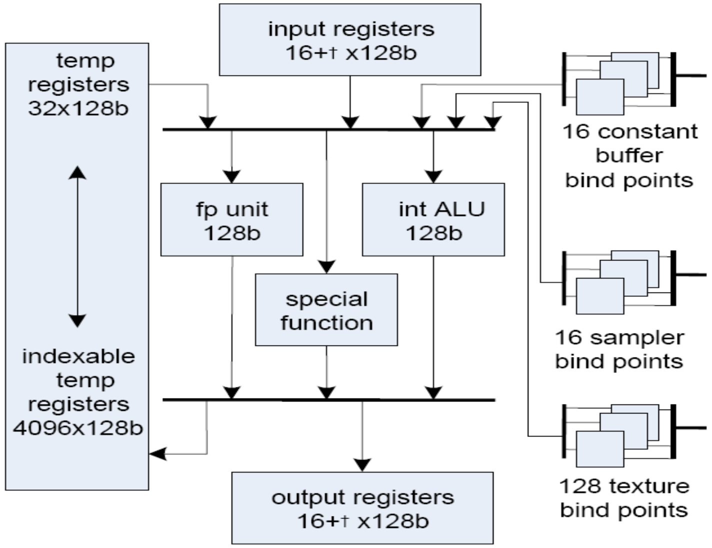


Fragment Merging Workflow



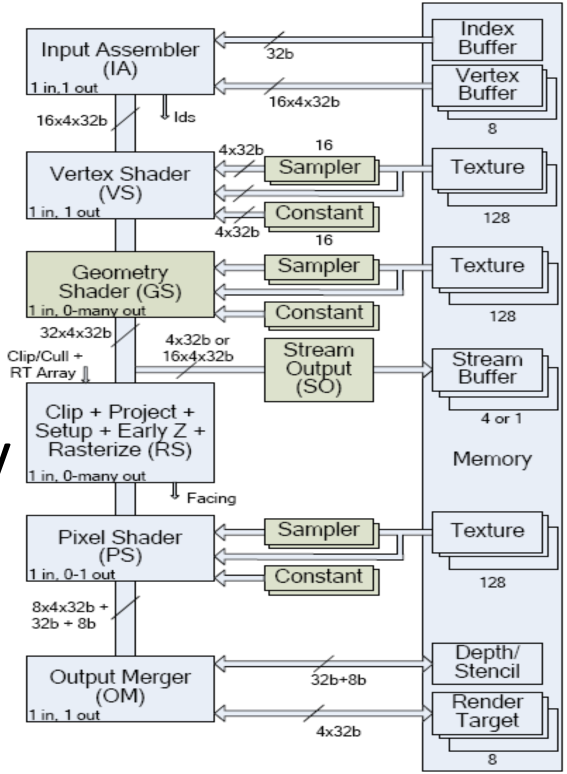
Unified Shader Model

- Same instruction set and capabilities for all shaders
- Dynamic load balancing between vertex and fragement shaders
- IEEE-754 floating point
- Enables new GPGPU languages like CUDA



Geometry Shader

- Between vertex and pixel shader
- Can generate primitives dynamically
- Procedural geometry
  - E.g., growing plants
- Geometry shader can write to memory
  - Called „stream output“
  - Enables multi-pass for geometry



Geometry Shader Examples

Figure 5: From left —— render to cube map, particle system, instancing, shadow volume, displacement mapping.

- Cube Map: GS instances every triangle 6x
- Particles produced by GS as stream-out
- Plants created as parameterized instances
- Shadow volume created by GS extrusion
- Displacements created by GS

Tesselation Shader

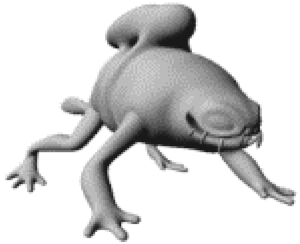
- Since DirectX11
- New shader stage: Tessellation
- Input: low-detail mesh
- Output: high-detail mesh

CryEngine3

Authoring without Tessellation

Sub-D modeling

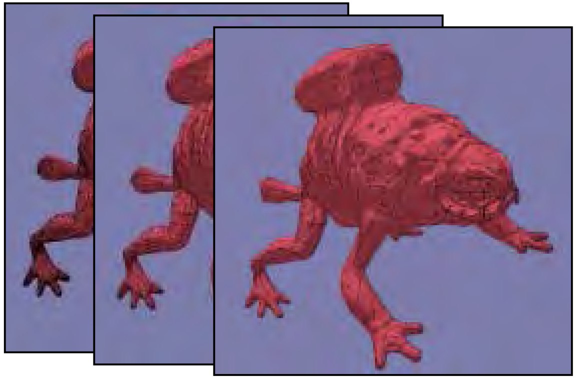
Animation



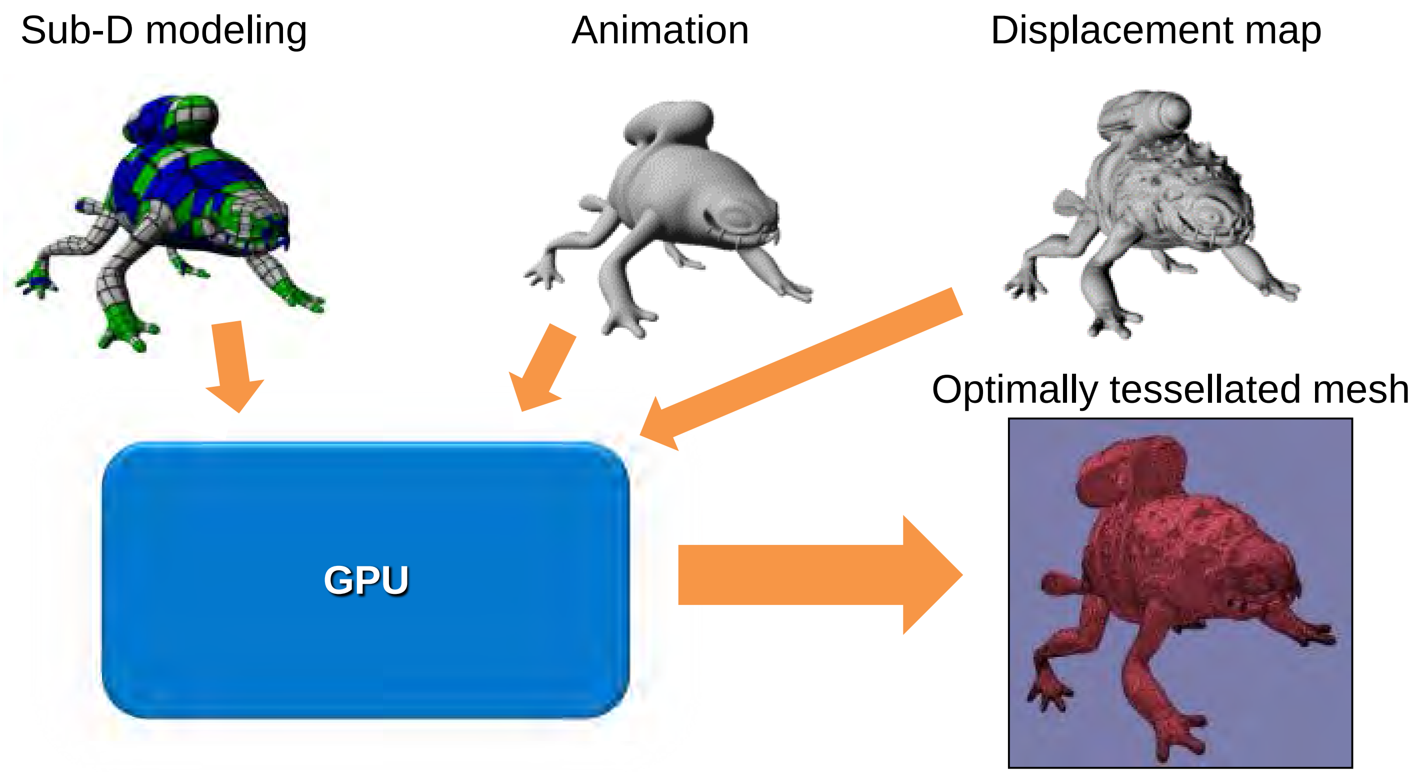
Displacement map

Polygon mesh

Generate LODs

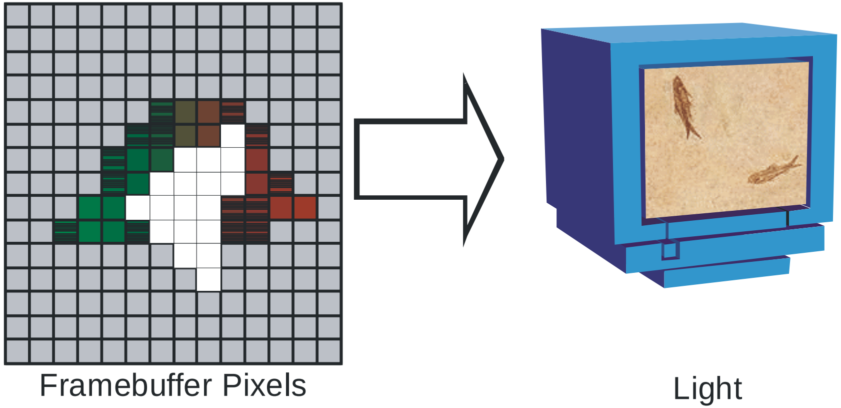


Authoring with Tessellation



Display Stage

- Gamma correction
- Historically: digital to analog conversion
- Today: digital scan-out, HDMI encryption?

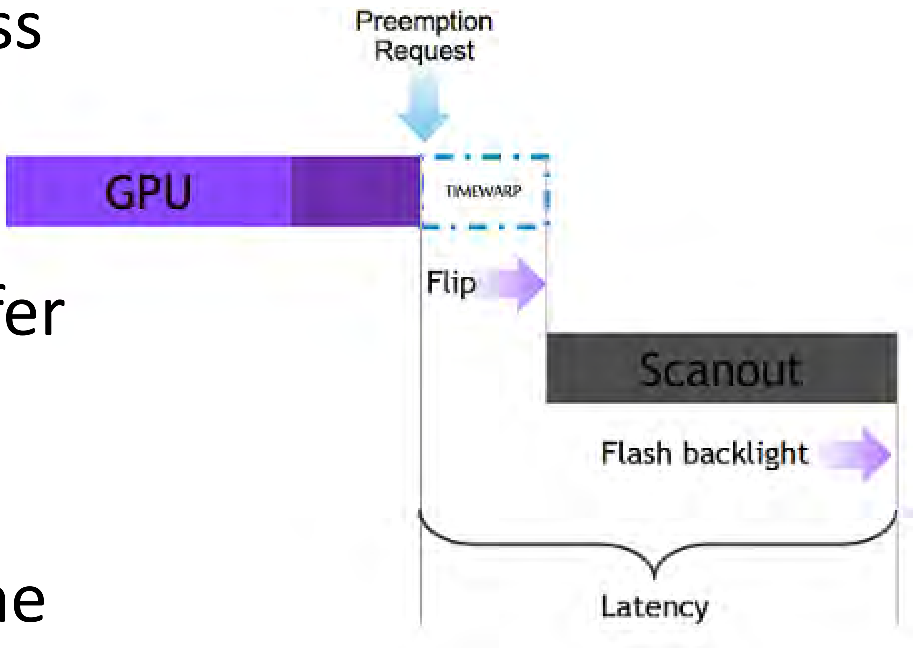


Display Format

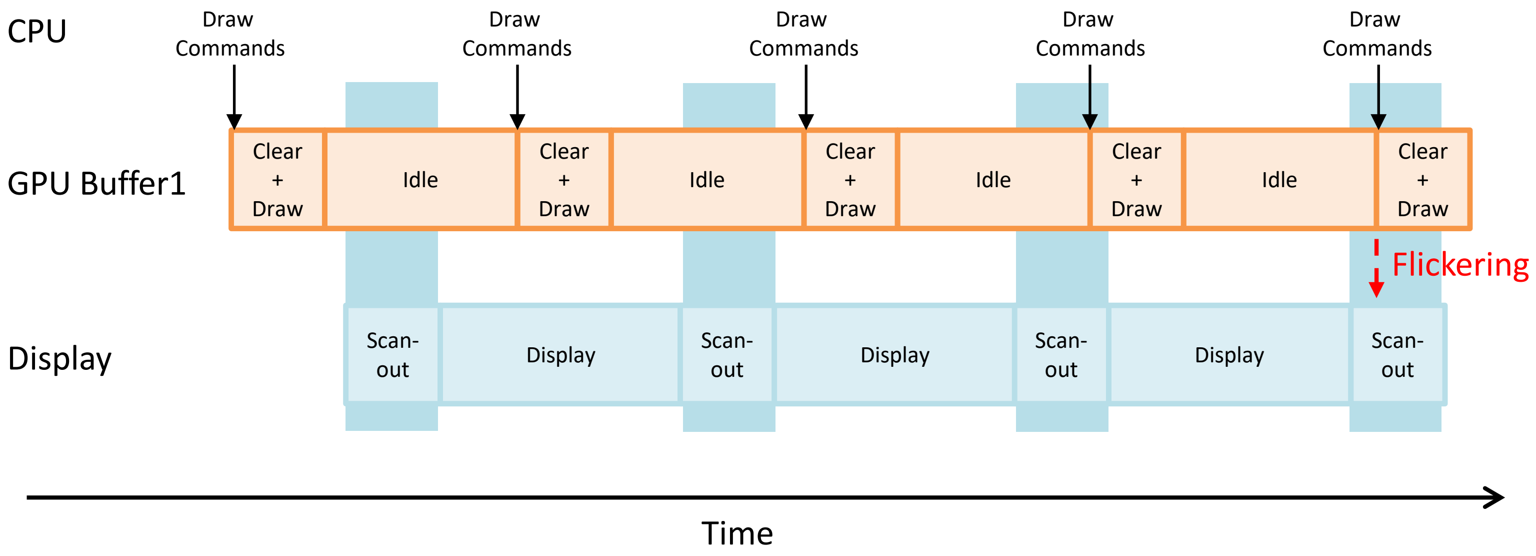
- Frame buffer pixel format:
  RGBA vs. index (obsolete)
- Bits: 16, 32, 64, 128 bit floating point, …
- Double buffering for smooth animation
- Quad-buffering for stereo graphics
- Overlays (extra bitplanes)
- Auxilliary buffers: alpha, stencil, depth

What is Display Synchronization?

- Need to synchronize access to frame buffer between GPU and display
- Cannot change frame buffer during display scan-out
- Need proper buffering in the whole graphics pipeline

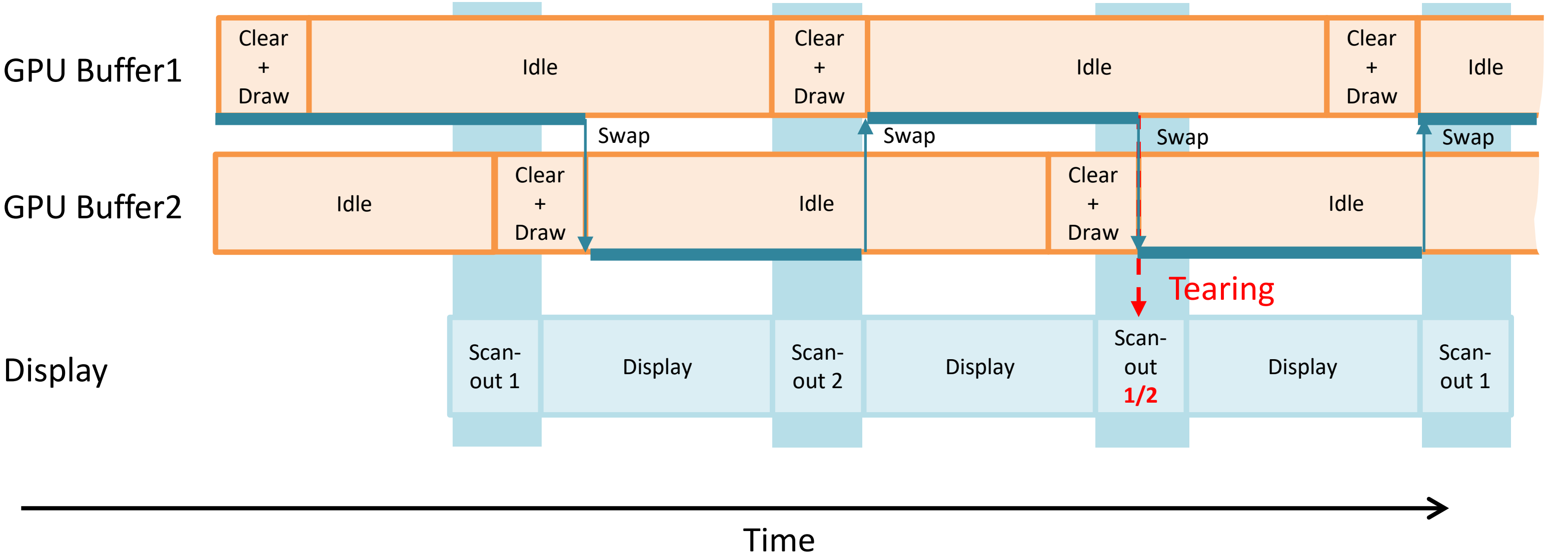


Single Buffering



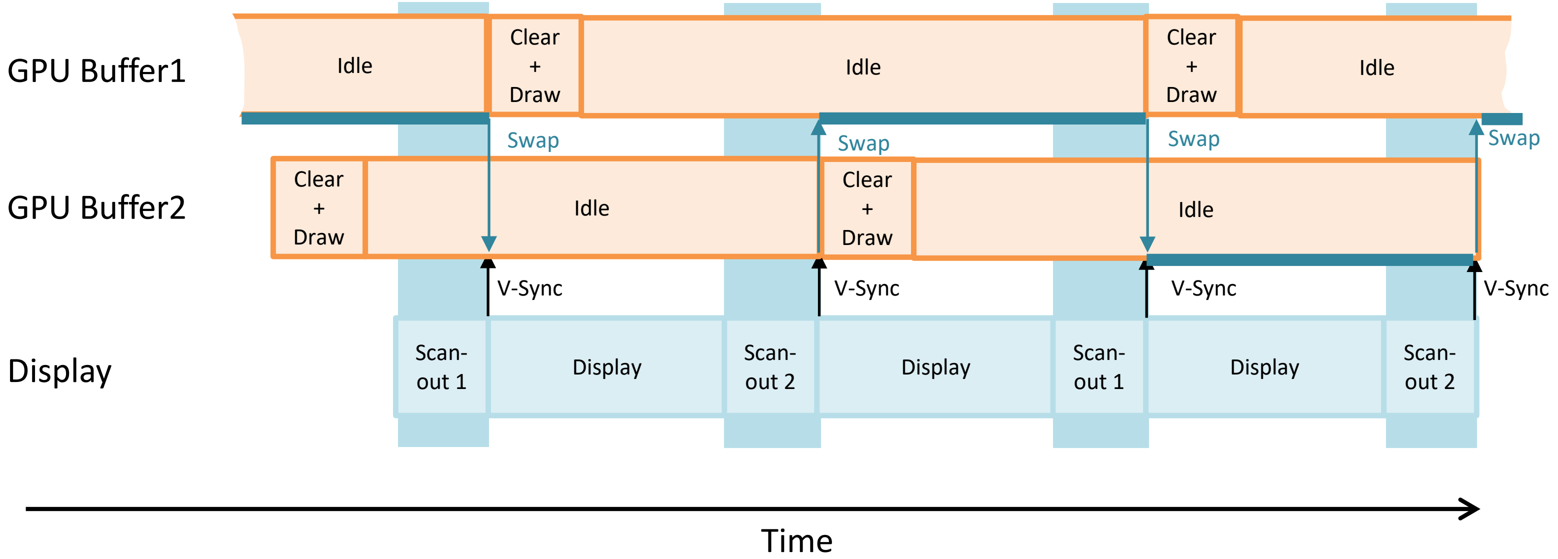
Double Buffering without V-Sync

- Front buffer used for scan-out GPU→display
- SwapBuffers() changes front and back buffer
- No flickering, but tearing artefacts



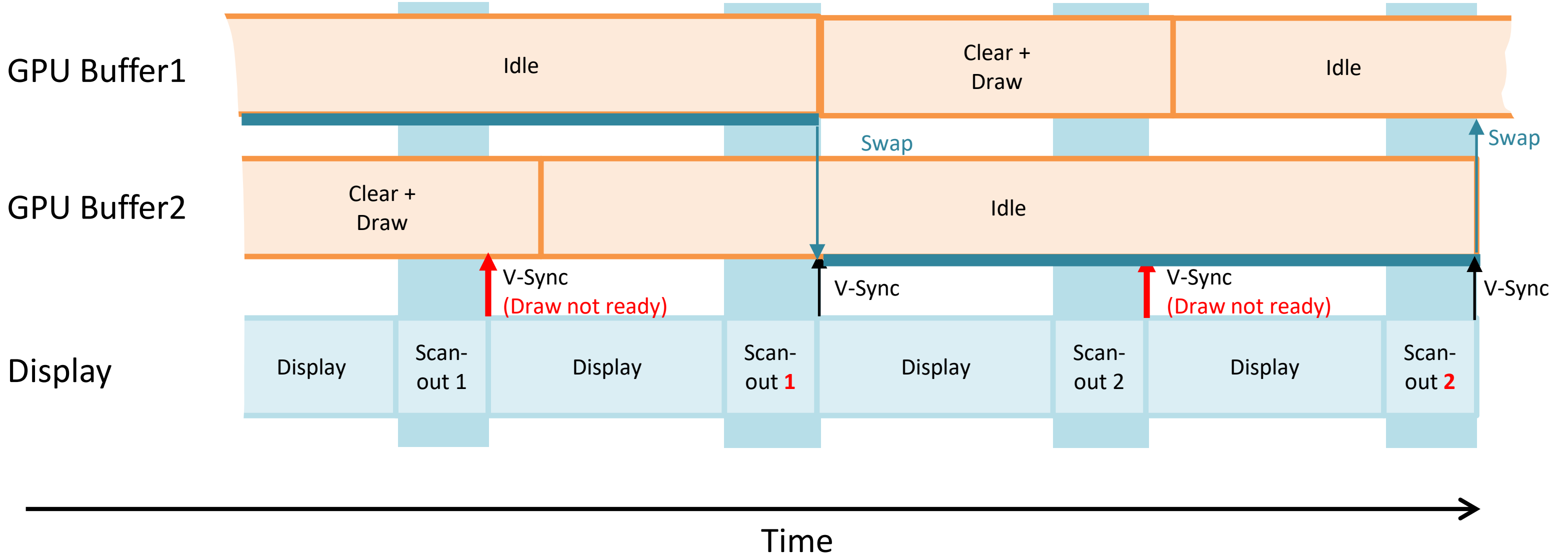
Double Buffering with V-Sync, Fast

- V-Sync means the display is done with frame scan-out from GPU
- When rendering is **fast**, the frame rate is limited by display rate
- Additional latency of max. one frame time



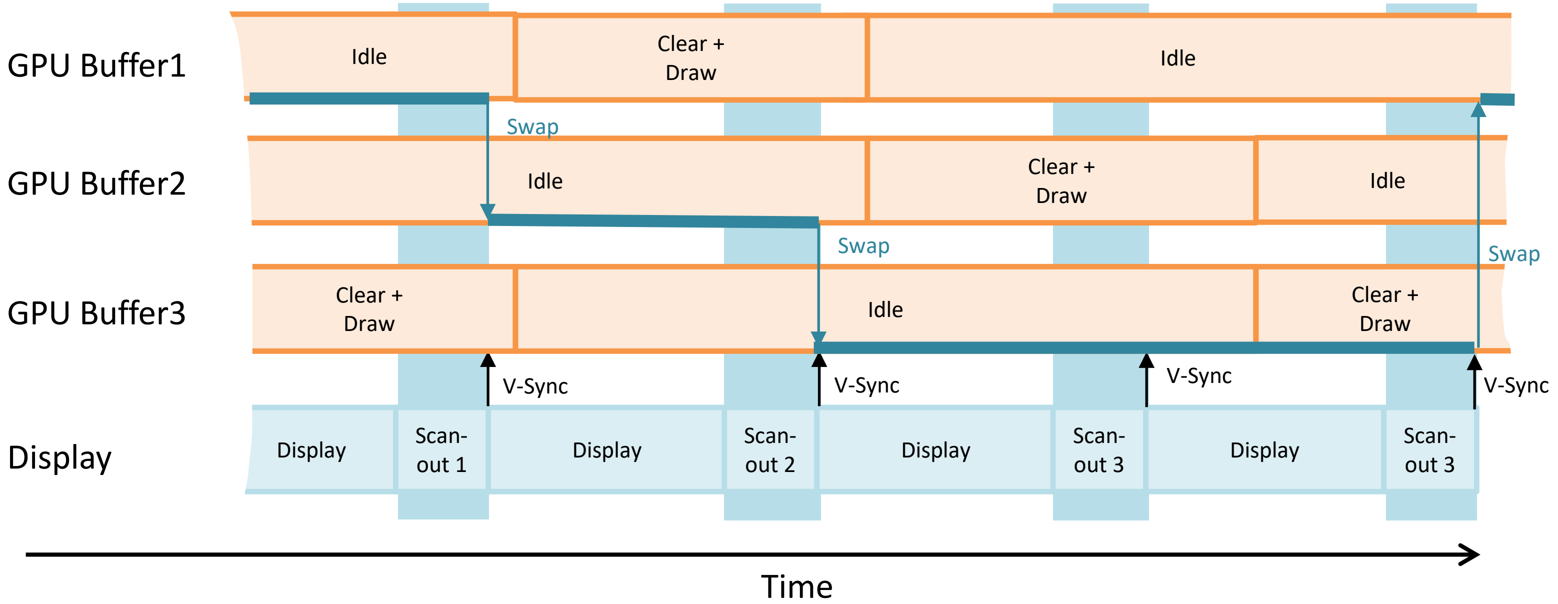
Double Buffering with V-Sync, Slow

- When rendering is **slow**, frame rates can only be integer fractions of display rate
- Display rate 60Hz → frame rates 60, 30, 20, 15, … (but not 59) possible
- Adaptive V-Sync (NVIDIA) turns off V-Sync automatically if rendering is slow



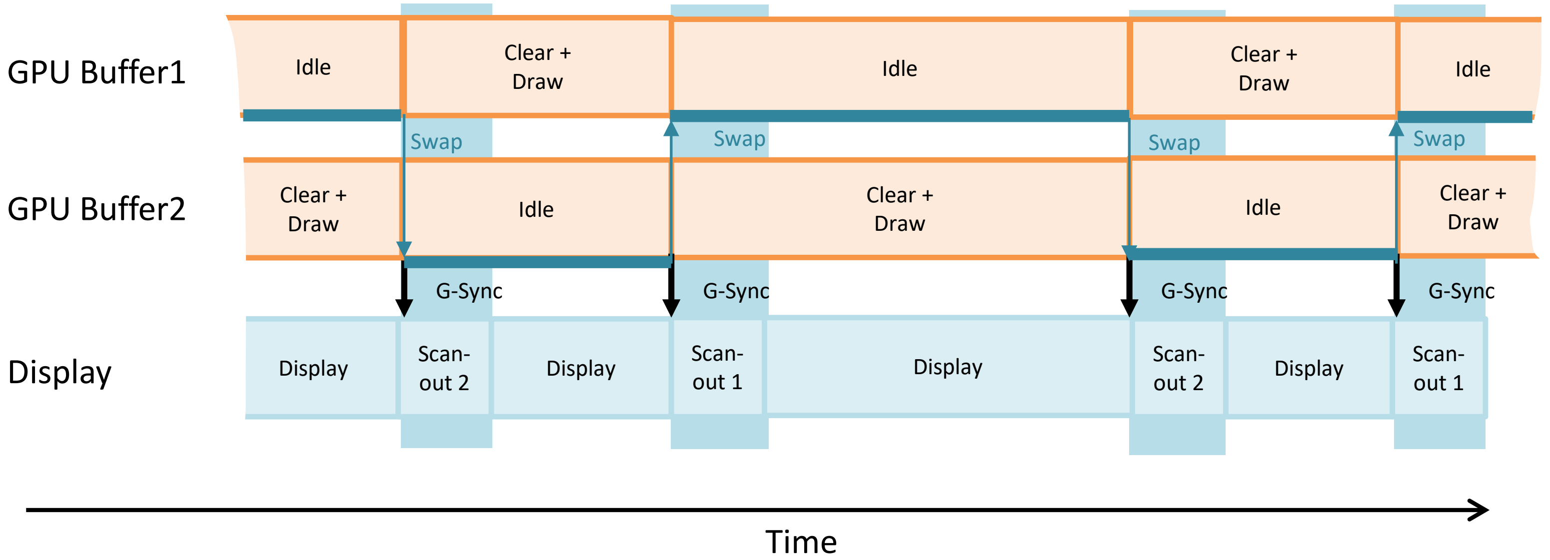
Triple Buffering with V-Sync

- Triple buffering only makes sense together with V-Sync
- When rendering is **slow**, third buffer allows continuous drawing
- When rendering is **fast**, two back-buffers can be alternated (wasted frames!)



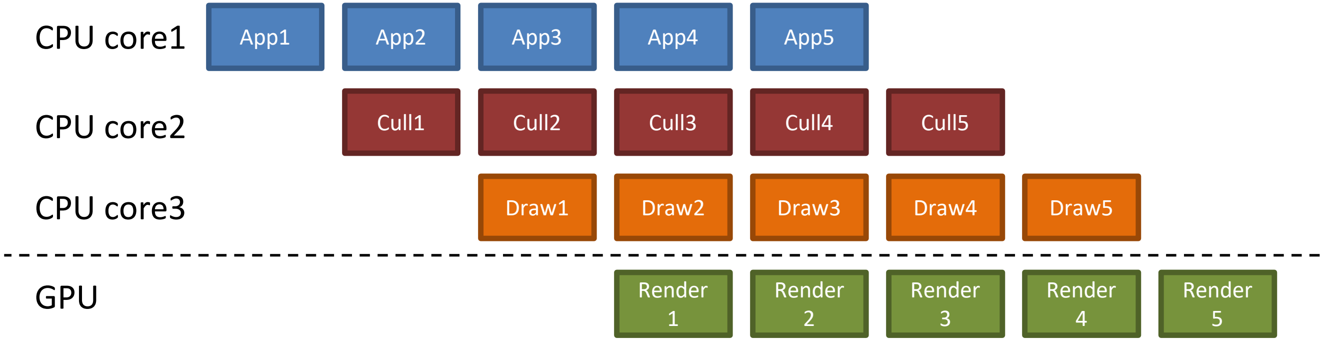
G-Sync / FreeSync

- Display scan-out can be triggered by GPU
- Requires additional display hardware feature
- G-Sync (NVIDIA), FreeSync (AMD)



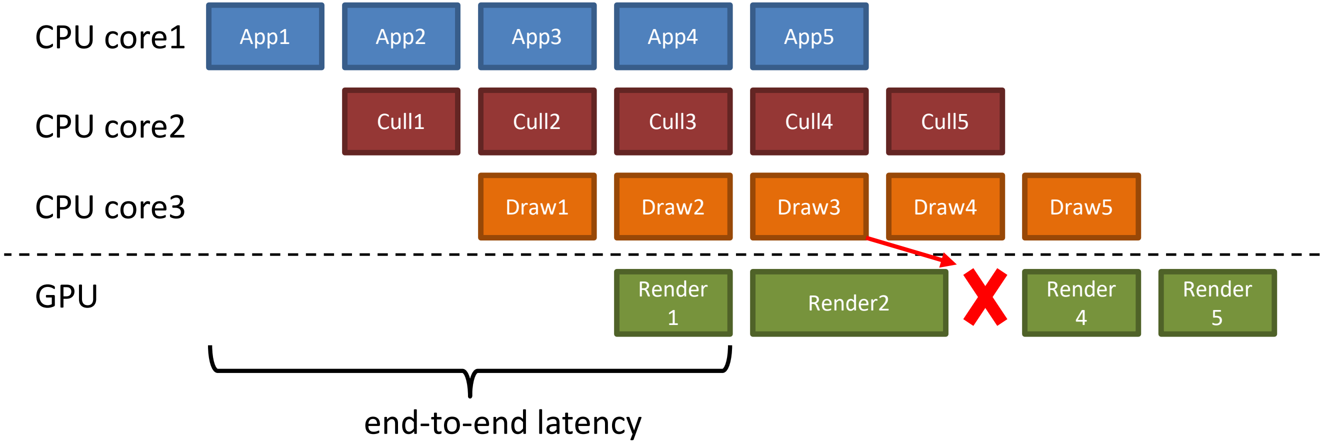
Multi-Threaded Rendering Pipeline

- App stage: simulate 3D world
- Cull stage: determine object in view frustum
- Draw stage: issue OpenGL commands to driver (includes optimizations such as mode sorting)
- Everything must work at target frame rate!



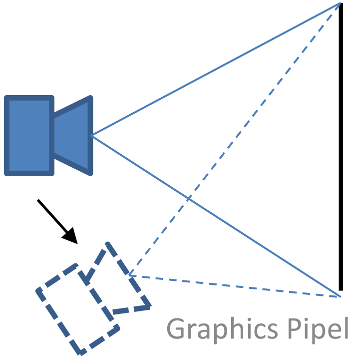
Minimizing Pipeline Latency

- Always aim for minimal latency
- Fixed size buffer from stage to stage
- Never wait for (downstream) consumer!



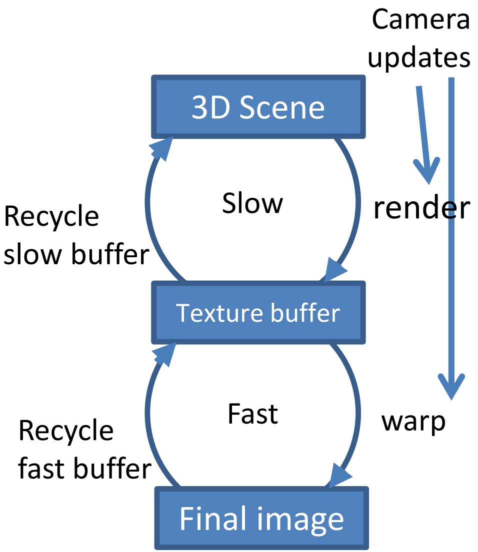
Warping

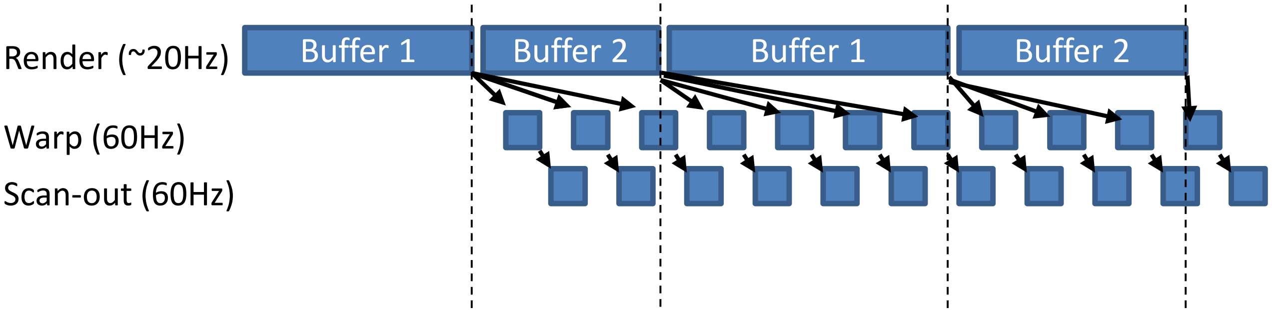
- Additional warping stage
- Pass 1: Render to Texture
- Pass 2: Perspective texture mapping with new camera parameters



Multi-Framerate Rendering

- Decouple Rendering and Display
- Slow pass: render to texture (image larger than viewport)
- Fast pass: Warp latest rendered image using latest camera
- Cheap warping using perspective texture mapping
- Oculus Rift: “Asynchronous Warping” from head tracker

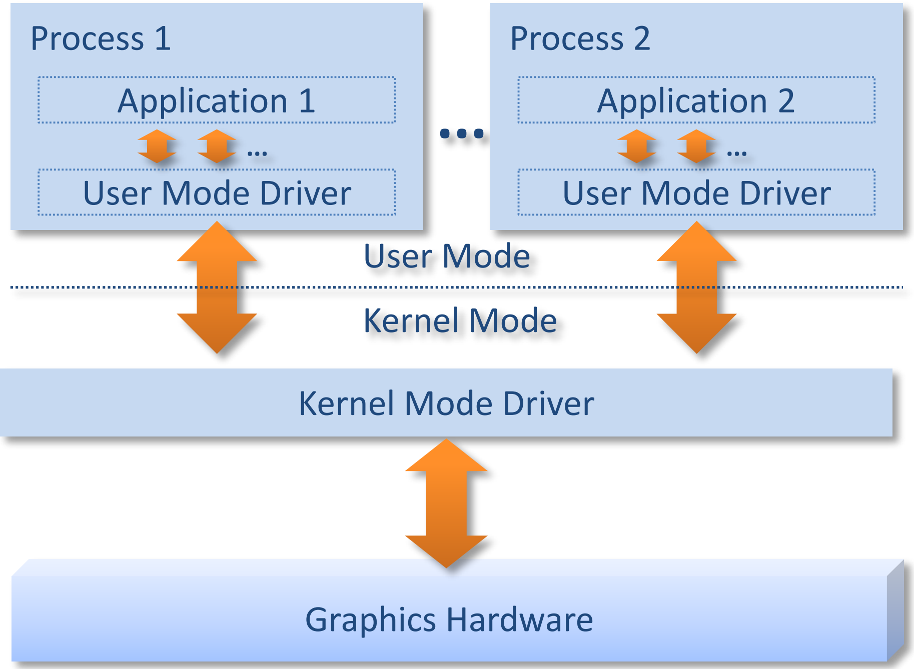




NEXT-GEN GRAPHICS API

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



What is a GPU Pipeline?

- Every application sees a virtualized GPU
  - Via **rendering context** object provided by driver to app
- Inside a context, one must use commands to configure GPU **pipeline state** before actual rendering
- Pipeline state consists of
  - State variables: blend, depth, culling, etc.
  - Shaders
  - Layout: how to map settings/bindings at each stage's shader

Traditional Pipeline Design

- Traditional = OpenGL (any) or DirectX (< version 12)
- Problem 1
  - Incremental configuration of state is costly
  - Submitting configuration changes to driver requires immediate validation, conversion, buffering
- Problem 2
  - Pipeline state *not* made explicit in rendering context
  - Drivers must have per-game optimizations built in

Next-Gen Pipeline Design

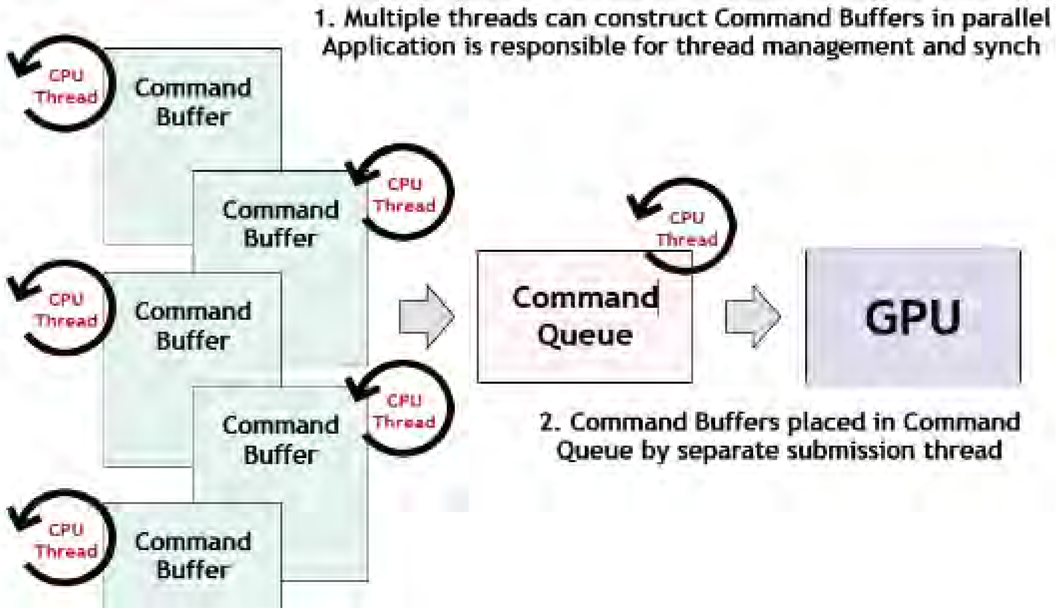
- New API: DirectX 12, Vulkan (OpenGL successor)
- Better fit for modern hardware (including mobile)
- Make (CPU driver side of) pipeline programmable
  - Multi-threading at your own risk
- Split rendering context into command buffers and dispatch queues
- Make pipeline state and render passes explicit
- Low overhead

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*
- Multi-pass rendering just *switches* pipeline states
- Build *many* command buffers from *many* threads
- Can use *synchronization* primitives across buffers
  - Event, barrier, semaphore, fence

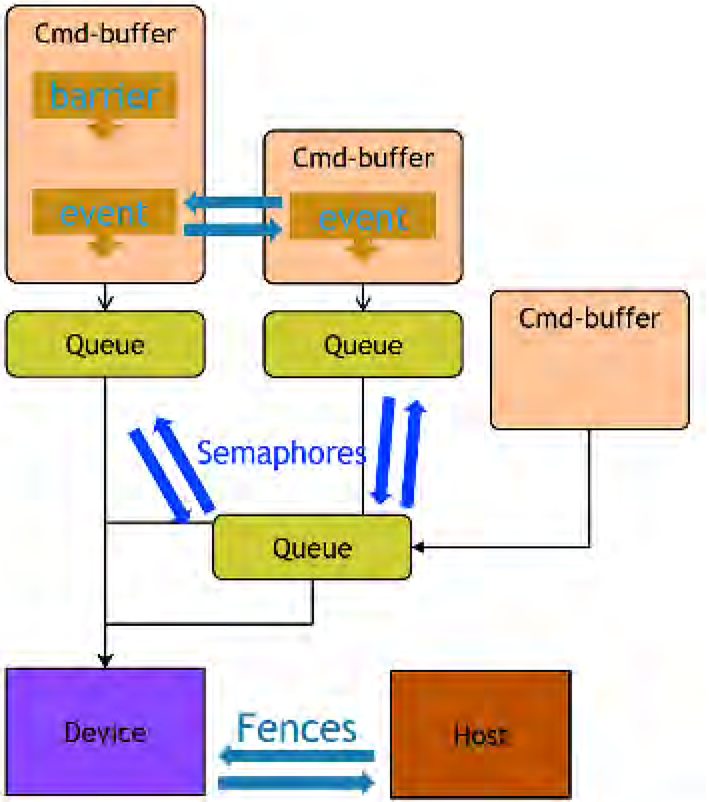
Queues

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



Synchronization

- **Events/barriers** synchronize within a queue's buffer(s)
- **Semaphores** synchronize across queues
- **Fences** synchronize between GPU and CPU

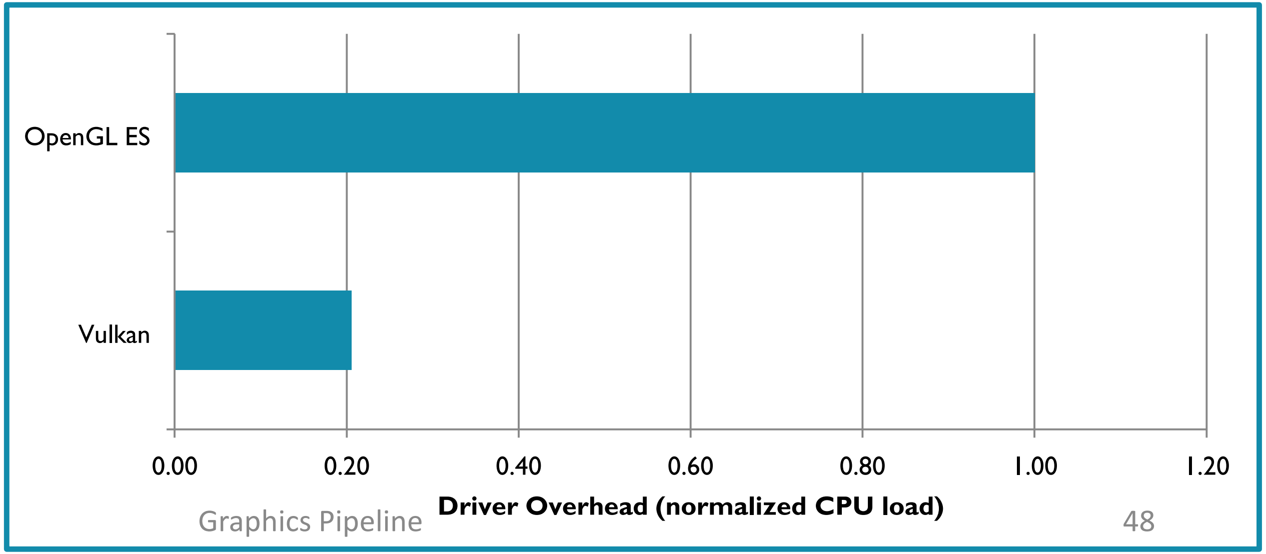


Vulkan Shaders

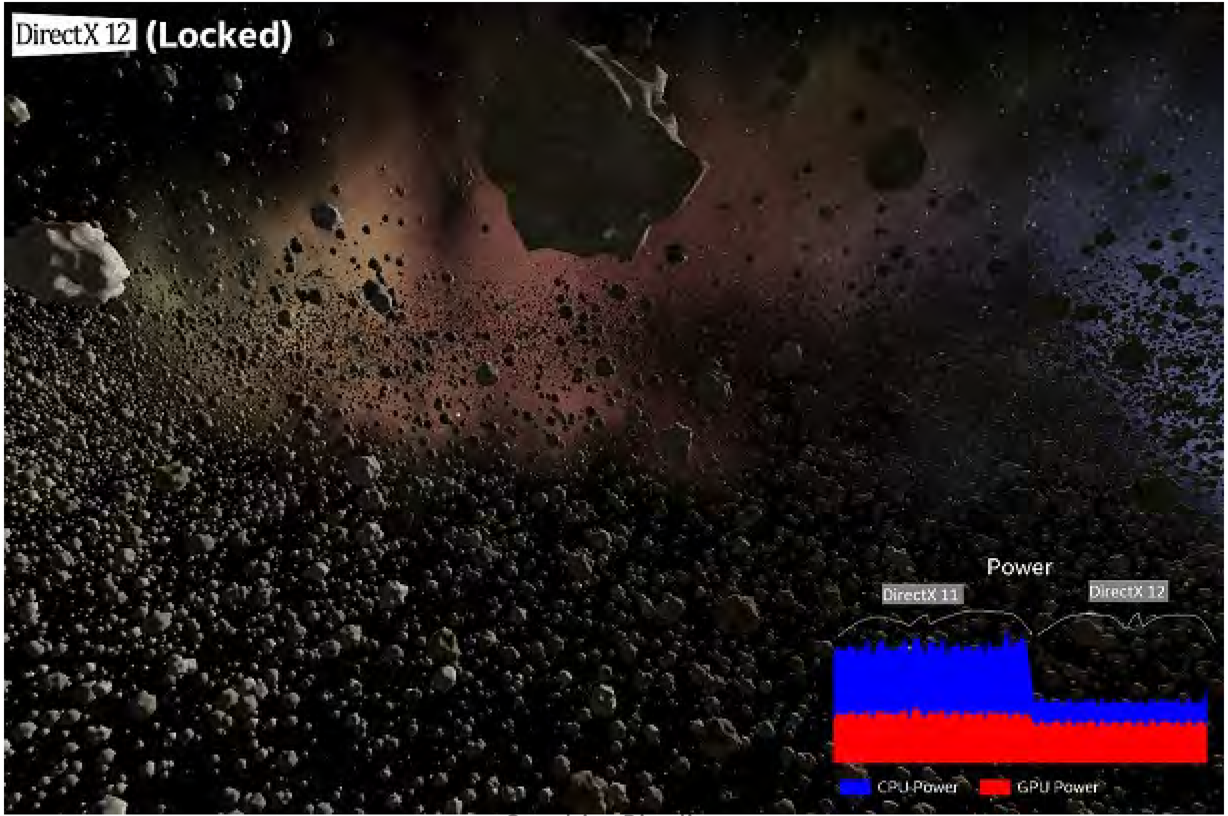
- Compilation separated into front-end and back-end
- Front-end: GLSL, OpenCL, or new shader languages
- Back-end = SPIR-V
  - Standard Portable Intermediate Representation
  - Byte-code derived from LLVM
- Games ship with SPIR-V, not shader source
- Vendor neutral: JIT compile to NVIDIA, AMD...
  - HLSL uses intermediate code too, but not vendor-neutral

ARM Vulkan Benchmark

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



Microsoft Asteroids Demo

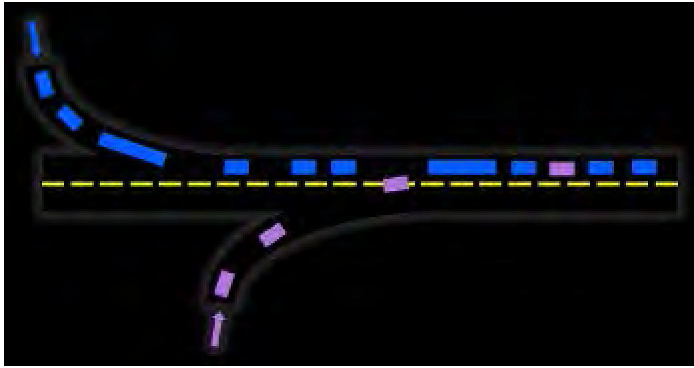


Context Switch Problem

- Multiple queues submit commands
- Every queue has different context (pipeline state)
- Only one queue may submit
- Without preemption
  - New jobs must wait
- With preemption
  - New high-priority job can interrupt running job
  - But context switch introduces overhead

Async Compute

- GPU with async compute
  - Multiple queues can submit commands
  - No overhead
- Particularly important when switching between graphics and compute shaders



AsyncCompute.mp4