Vulkan

(including Vulkan Fast Paths)

Łukasz Migas

Software Development Engineer – WS Graphics

AMD

Let’s talk about OpenGL (a bit)

- History
  - 1.0 - 1992
  - 1.3 - 2001 – multitexturing
  - 1.5 - 2003 – vertex buffer object
  - 2.0 - 2004 – GLSL
  - 3.0 - 2008 – framebuffer object
  - 3.2 - 2009 – geometry shader
  - 3.3 - 2010 – sampler object
  - 4.1 - 2010 – get_program_binary
  - 4.3 - 2012 – compute shaders
- Legacy
  - glBegin(); / glEnd();

What is Vulkan?

- Software change
- Closer to the metal - more control done by App
- If OpenGL is JavaScript, Vulkan is C++

Why Vulkan?

- Reduced CPU overhead comparing to OpenGL
- Thin layer API - efficiency and performance
- Improved scalability across multiple threads
- Greater developer control
- Asynchronous Compute ☺
- Multi GPU*

CPU Overhead – Identifying problems

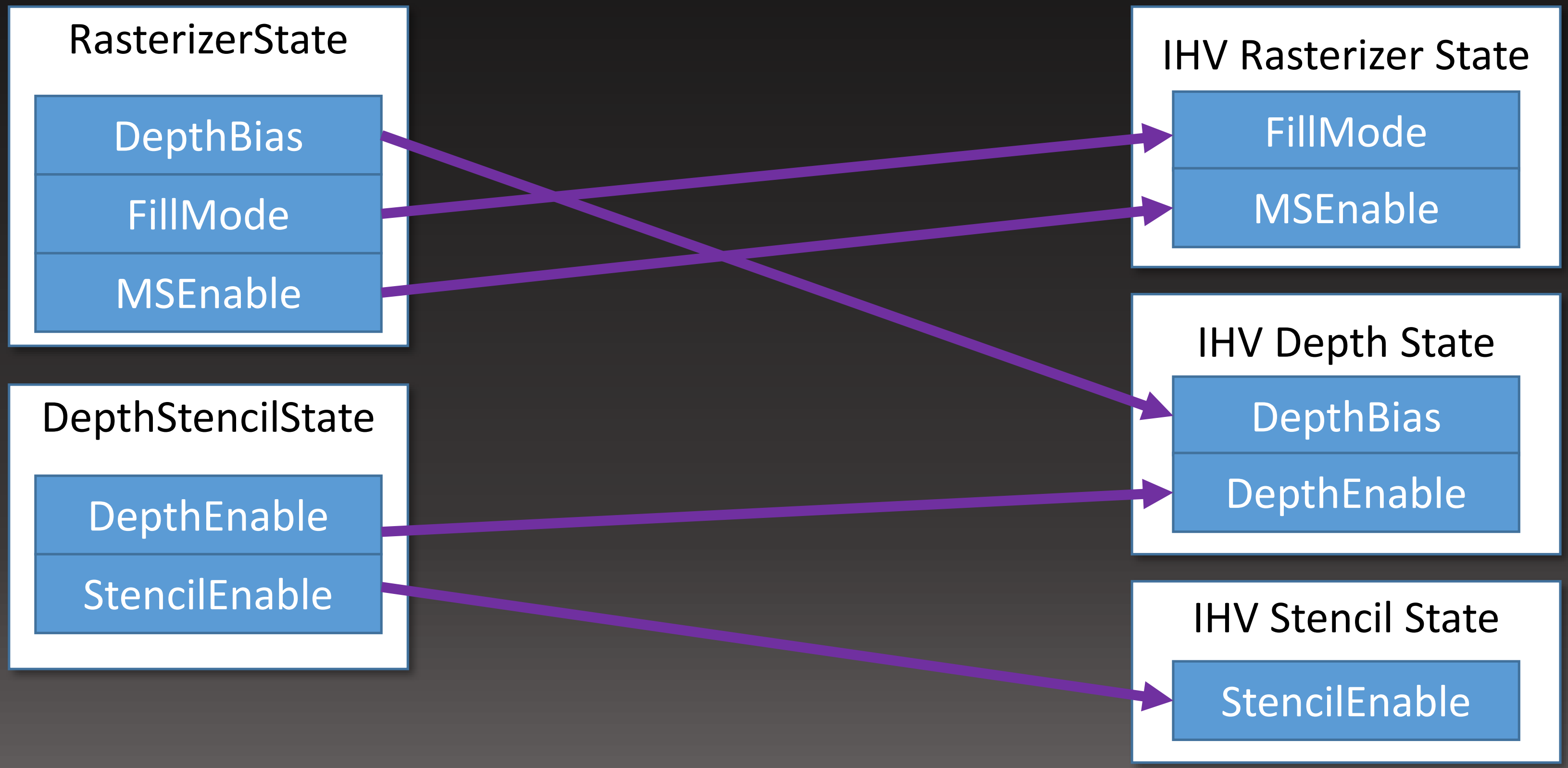
- What's wrong with this code?

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClearDepth(1.0f);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
glShadeModel(GL_SMOOTH);
```

- What's wrong with this code?
  - `glMapBuffer( myBufferID, GL_WRITE_ONLY_ARB ); / glUnmapBuffer();`
  - `glFinish();`

CPU Overhead – Identifying problems

- State objects do not match HW state



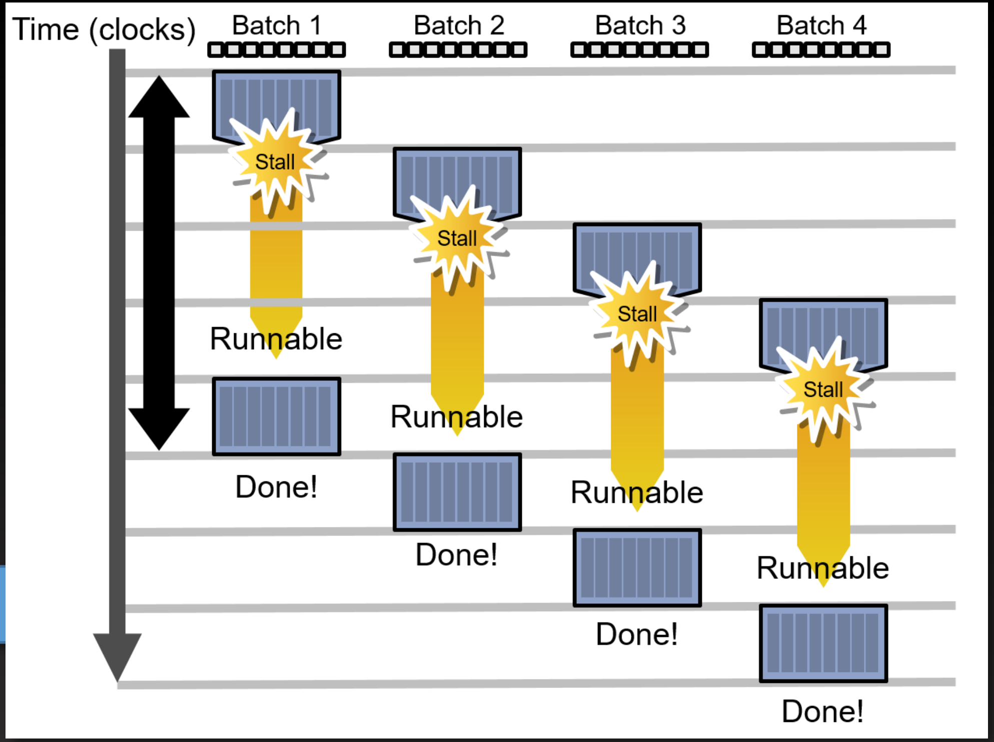
AMD

RADEON TECHNOLOGIES GROUP

CPU Overhead – Identifying problems

- Drivers resolve state at Draw:
  - Hazard
  - Resource lifetime
  - Residency management (state tracking)
  - Get*
  - Redundant binding

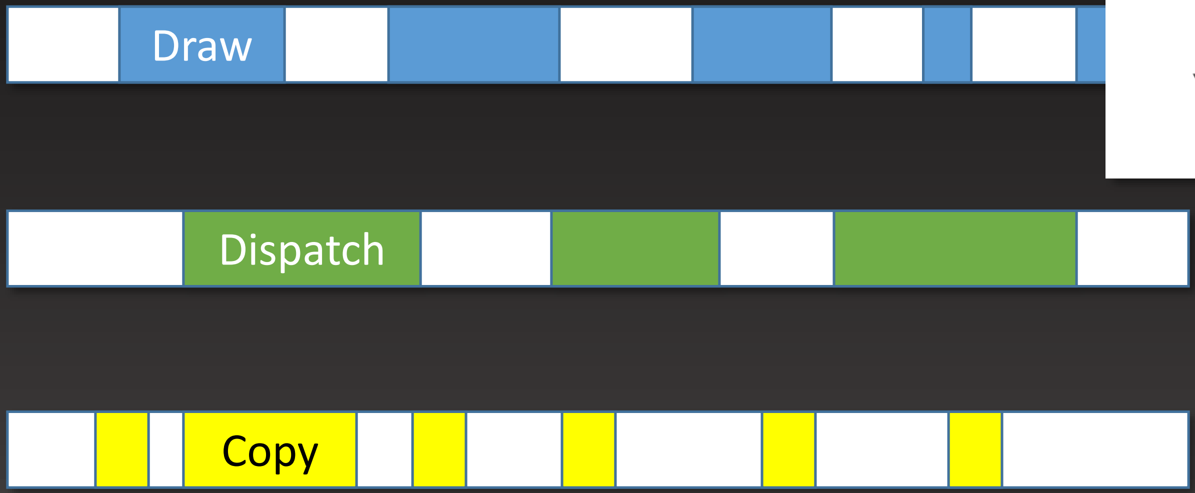
What is Async Compute?



Direct Queue:

Compute Queue:

Copy Queue:



What is Vulkan?

- Views
- Pipelines
- Descriptor Set Layouts / Descriptor Pools / Pipeline Layouts
- Render Passes / Subpasses
- Command Queues and Command Buffers
- Synchronization / Barriers

Pipelines

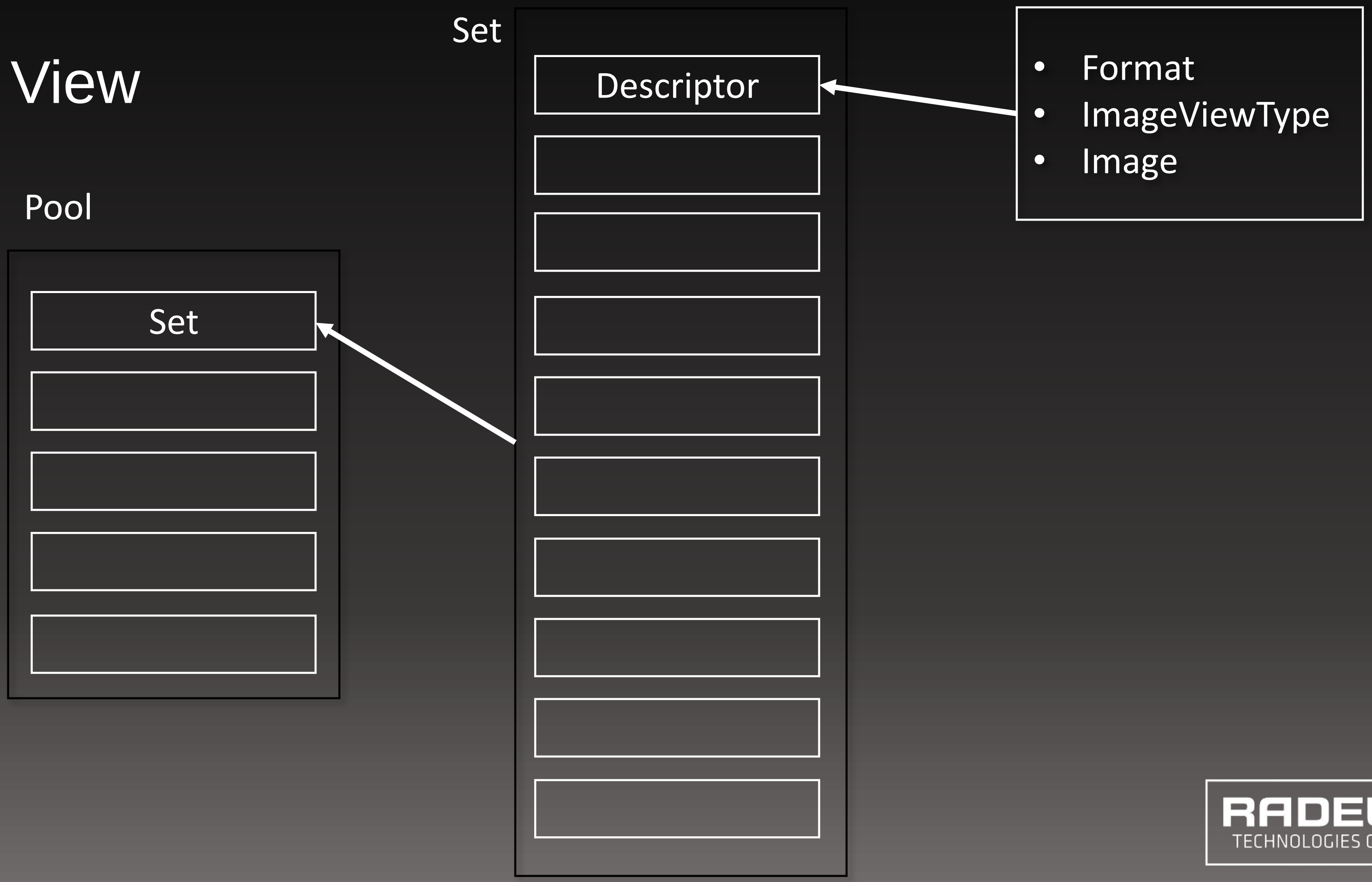
- Inside:
  - Input assembly
  - Shaders
  - States
  - Topology
  - Viewport*
  - Scissor*
- Outside:
  - Resource bindings
  - Viewport*
  - Scissor*
  - Blend constants*
  - Stencil ref*
  - Stencil masks*

Pipeline

- Vertex Shader
- Fragment Shader
- Geometry Shader
- Tess Evaluation
- Tess Control
- BlendState
- RasterizerState
- DepthStencilState
- InputLayout
- RenderPass
- PipelineLayout
- SampleDesc

Descriptor Sets Binding Layout

- Descriptor <-> View
- Sets
- Pools



Pipeline Layout

0: DescriptorSetLayout
1: DescriptorSetLayout
2: PushConstants

Sampled Image
Sampled Image
Uniform Buffer
Sampler

Sampler
Sampler
Sampler

1.0f, 0.0f, 0.0f, 0.0f
0.0f, 1.0f, 0.0f, 0.0f
0.0f, 0.0f, 1.0f, 0.0f
0.0f, 0.0f, 0.0f, 1.0f

Some GLSL, please?

- layout(set=0,binding=0) uniform sampler s0;
- layout(set=0,binding=1) uniform samplerBuffer sb0;
- layout(set=0,binding=2) uniform texture2D t0;
- layout(set=0,binding=3) uniform samplerBuffer sb1[4];
- layout(set=0,binding=4) uniform texture2D t1[2];
- layout(push_constant) uniform BlockPushConstants {
-     vec4 some_number;
- } PushConstants;

One set design

- Place all Descriptors in one giant Descriptor Set
  - layout (set=0, binding=N) uniform texture2D textures[hugeNumber]
- Leave the one giant Descriptor Set always bound
  - No more binding/updating Descriptor Sets for each draw/dispatch
  - Instead use Push Constant

- Constants data: draws which need to source {per-frame, per-pass, and per-draw} constants
  - Each pass (few passes per frame) gets a separate UNIFORM_BUFFER_DYNAMIC Descriptor
  - Buffer contents: [per-frame] [per-pass] [draw0] [draw1] [draw2] . . . [drawN]
  - Per-frame data is duplicated for each pass and can be accessed with immediate offsets
  - Per-pass data can be accessed with immediate offsets
  - Per-draw uses the dynamic base offset supplied in the Push Constant

Renderpasses

- Renderpasses are chunks of back to back GPU work
  - Represented by a Vulkan object
  - Contain one or more sub-passes
  - All rendering happens inside a renderpass
    - Even if it has only a single subpass
  - Dependencies between subpasses are part of the renderpass
    - Driver can schedule work based on future knowledge
    - Driver generates a DAG from dependency information
- Renderpasses are a time machine for drivers!

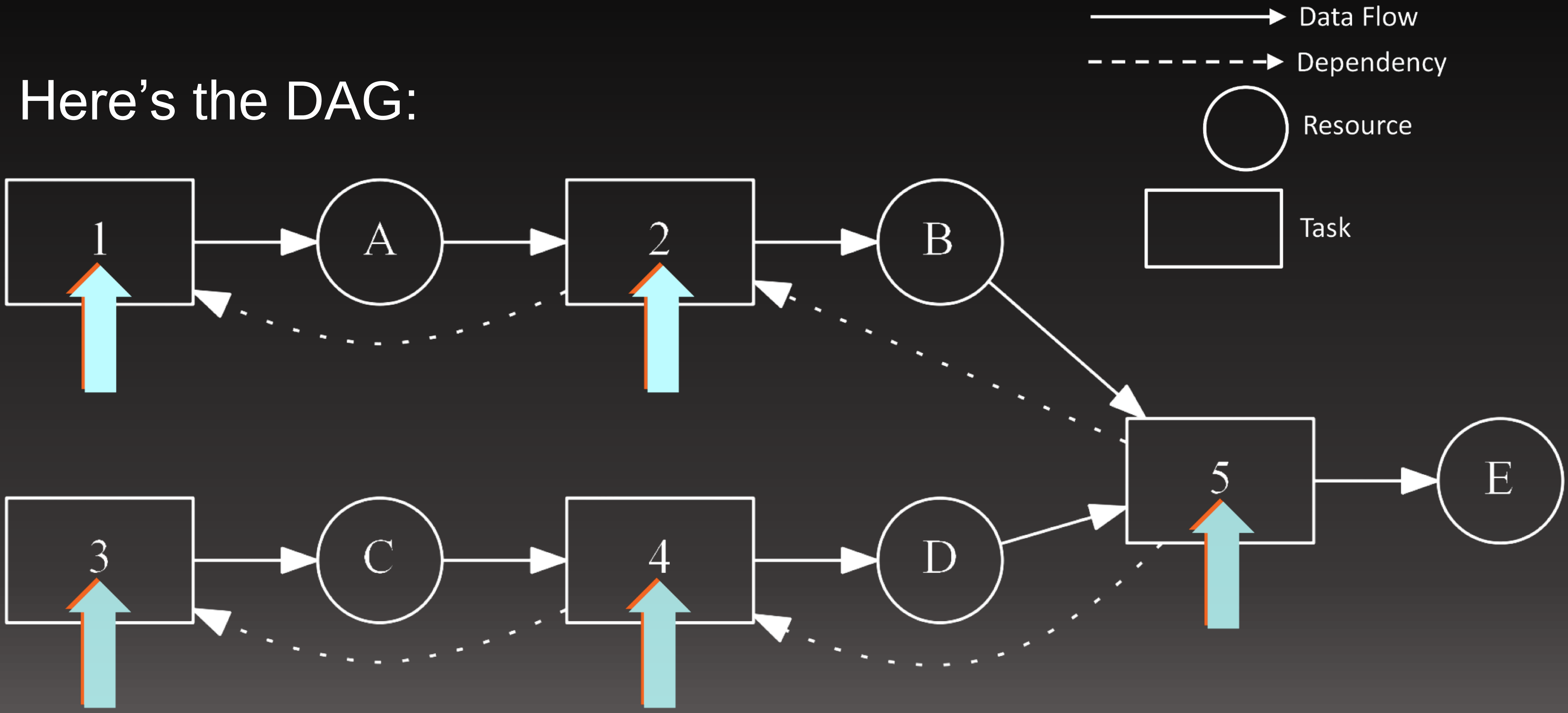
Renderpasses in words

- Consider the following:
  - Subpass 1 produces resource A…
  - Which is consumed by subpass 2, producing resource B
  - Subpass 3 produces resource C…
  - Which is consumed by subpass 4, producing resource D
  - Finally, subpass 5 consumes resources B and D, producing final output E

- Blah, blah, blah; loads of text
  - But this is what API order calls look like

Renderpass in pictures

- Here's the DAG:



Renderpass information

- Arrays of attachments, subpasses and dependency information
- Any number of attachments can be used by a renderpass
  - They are referenced by subpasses
- Each attachment contains the following:
  - Format and sample count
  - Load operation – where to get the data from (memory, clear, or don’t care)
  - Store operation – where to leave the data (memory, or don’t care)
    - There are separate load and store operations for stencil
  - Expected layout at the beginning and end of the renderpass
    - Driver will insert layout changes for you

Graph Building

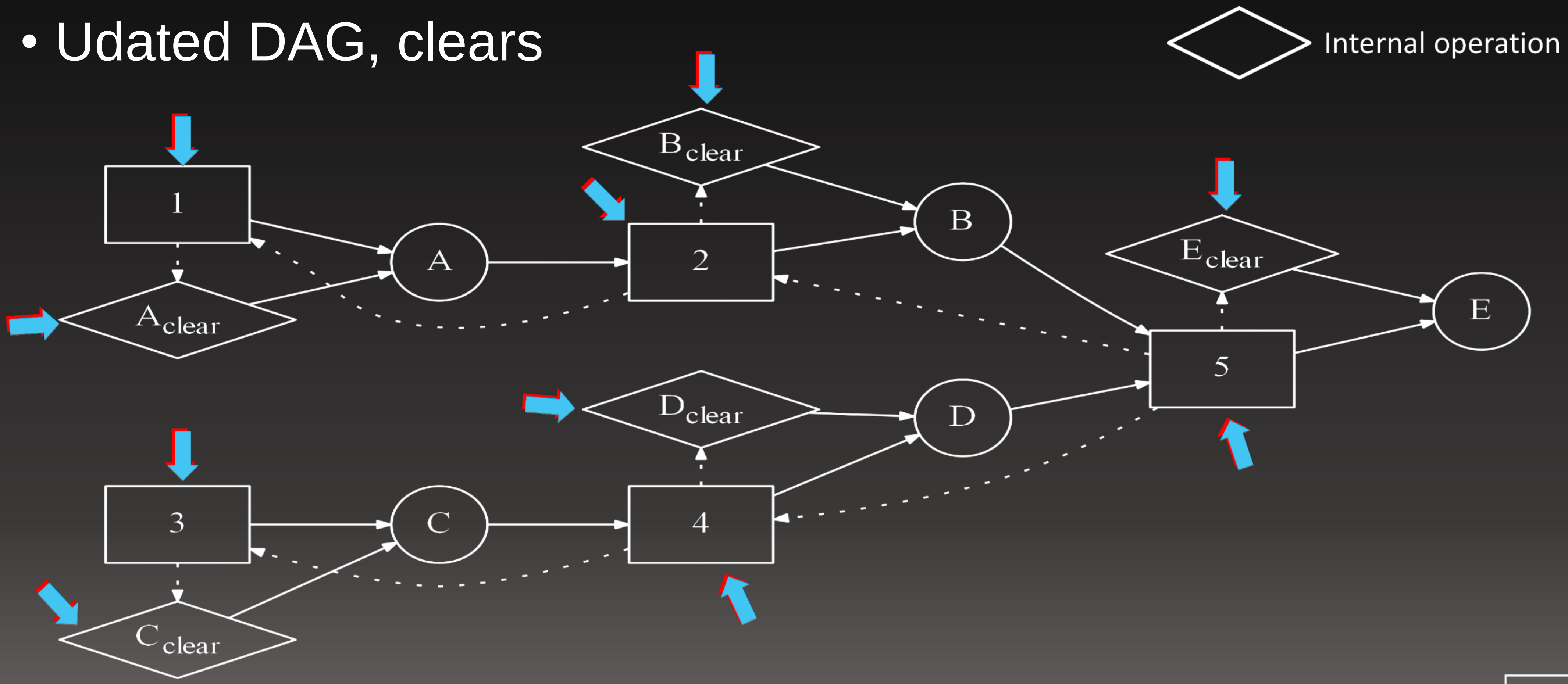
- Driver uses renderpass structures to form a DAG
  - Subpasses produce and consume data
  - Resource barriers inserted automatically by driver
  - Scheduling information generated at renderpass creation time
- A DAG of one node isn't helpful
  - Need renderpasses to include multiple subpasses to be useful

But wait, there's more

- Internal driver operations
  - Attachments have initial and final states
    - Clears are part of beginning a subpass, for example
  - Attachments go from being outputs to being inputs
    - Flush color caches, invalidate texture caches, change layouts, insert fences
  - Some surfaces require more attention
    - Compressed depth not directly readable by shaders, for example
    - Requires internal driver decompression

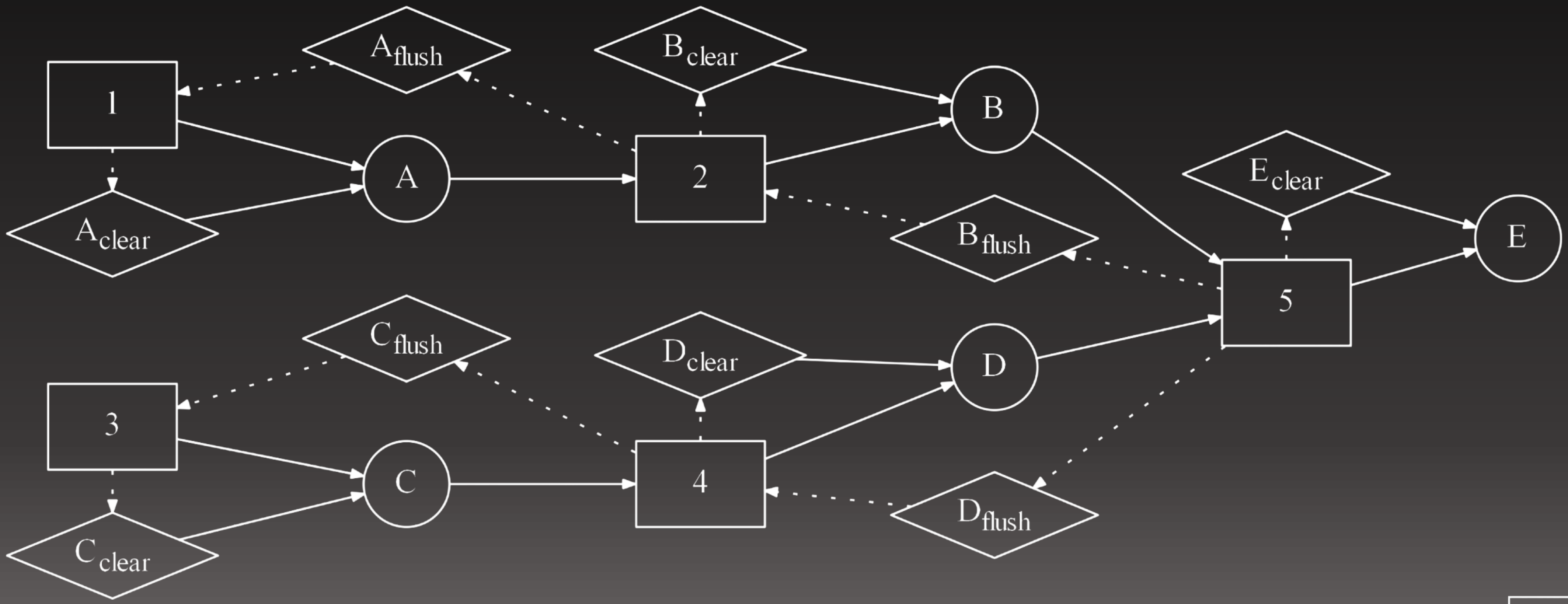
Load OPs

- Udated DAG, clears



Flush

- Udated DAG, flushes

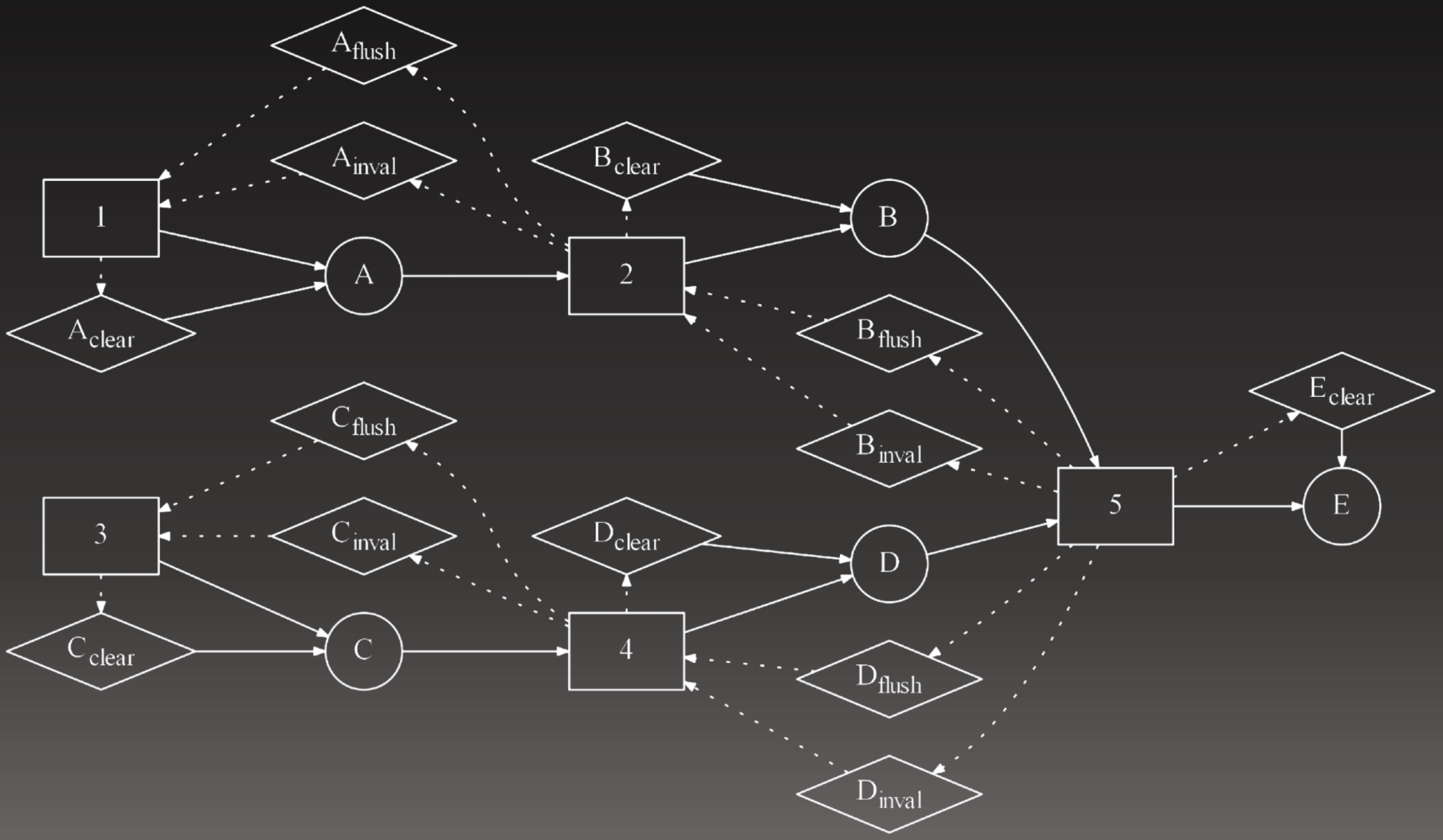


AMD

RADEON TECHNOLOGIES GROUP

Invalidate

- Udated DAG, flushes, invalidation



Predicting the future

- Renderpasses allow drivers to predict the future
  - Not really a prediction
  - you told it what you were going to do
  - Schedule clears, internal blits, cache operations, etc.
    - All done statically
    - When the renderpass is built
- “I can do that in the app, ‘cuase I’m a 1337 haxxorz”
  - Well, no, you can’t
  - Some of the internal driver operations aren’t exposed in the API
  - Some are only needed on some hardware

Let’s get crazy

- Pipelines are built with respect to renderpasses
  - Each Pipeline knows which renderpass it will be used with, and in which subpass
  - Renderpass knows where subpass outputs go
  - Renderpass knows the format of all attachments

Renderpasses - summary

- Renderpasses encapsulate data and execution flow
  - Driver can schedule internal work
  - Remove surprises at render time
  - Determine the fate of data early
- Many opportunities for GPU performance
  - Eliminate stalls and pipeline bubbles
  - Interleave internal operations with rendering
  - Optimize cache utilization
  - Choose formats and allocation strategies based on data flow

Synchronization

- We have three synchronization primitives
  - Fences
  - Semaphores
  - Wait events
- Fences allow to synchronize GPU and CPU work
  - Frame sync
  - Protect frame resources with a fence
- Semaphore is a heavy-weight, cross queue sync
- Wait events are light-weight, in queue sync

Barriers

- Synchronization
  - Make sure writes have finished before reads start
  - Timing issues if missed
- Visibility
  - Caches are visible to other units
  - Partial results, flickering, etc.
- Decompression
  - Make sure formats match
  - Corruption if missed

Barriers

- For any of the barriers
  - Make sure to transition into the union of the read states
  - Or them together – avoid VK_ACCESS_MEMORY_READ_BIT
- Batch as many barriers as you can into one call
- Need to specify source/destination queue
- Place transition close to semaphore

Barriers

- Avoid transitioning everything
  - Barriers have a cost!
  - Cost often scales with resolution
  - Cost changes between GPU generations
- Use render passes when possible
- Think about the required state

Thank you!

Disclaimer & Attribution

AMD

RADEON
TECHNOLOGIES GROUP