Get Your Engine Ready for Vulkan on Mobile

ARM

Hans-Kristian Arntzen

Engineer

GDC 2016

03 / 16 / 2016

Contents

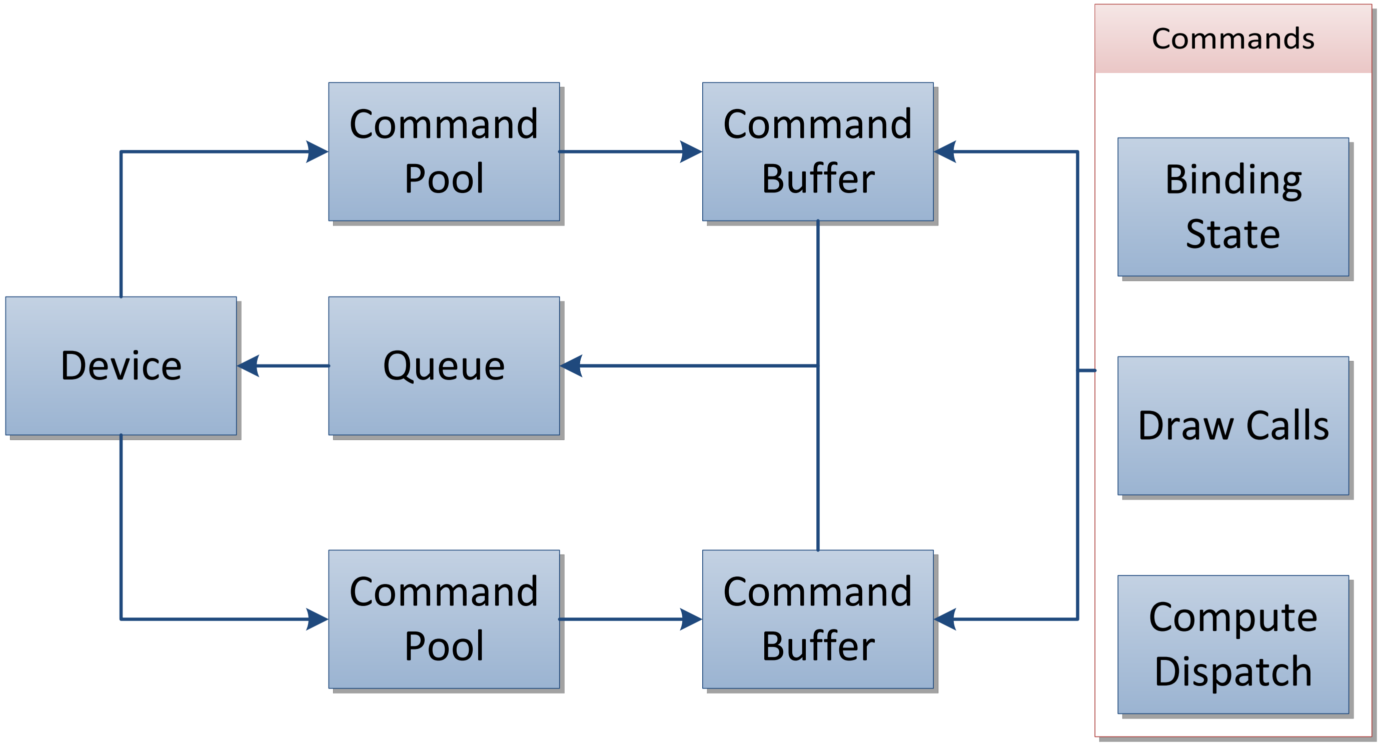
- Background
- Command buffers and queues
- Pipelines
- Synchronization
- Strategies for asynchronous GPU
- Moving to SPIR-V shaders

Background

- Vulkan is a brand new, industry standard graphics and compute API
- Aims to give developers more control over modern graphics chips
- Better control of when and where work happens
- Explicit control of memory resources
- Little to no magic happening in driver
- First class multithreading support
- Gives far more responsibility to API user to get things right
  - Production drivers disable validation meaning crashes or corruption with API misuse
  - Public, open-source validation and debug layers important

Command Buffers and Queues

- Binding state and dispatching work happens in command buffers
- All state is contained in command buffers
- Command buffers are submitted to the device

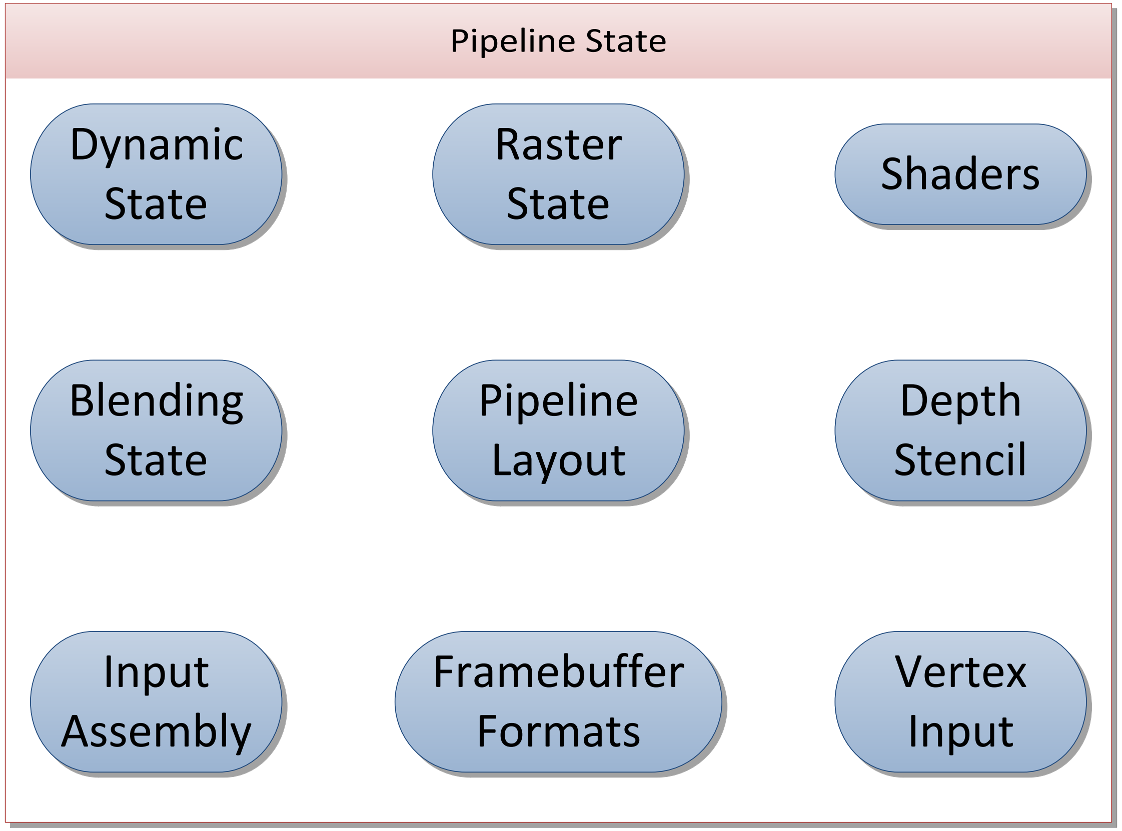


Pipelines

- Vulkan bundles state into big monolithic pipeline state objects
- Driver has full knowledge during shader compilation

```
vkCreateGraphicsPipelines(...)
;

vkBeginRenderPass(...);
vkCmdBindPipeline(pipeline);
vkCmdDraw(...);
vkEndRenderPass(...);
```



Synchronization

- Work submitted to the GPU is completed out of order
- The real challenge of learning Vulkan is understanding this part
- Hazards are resolved by API user
  - Reading from texture after rendering to it
  - Reading a texture before uploading it completes
  - Using results from compute shader before it completes
  - Reading back data on CPU before GPU completes
  - Deleting objects while in use by GPU
- Vulkan gives you the tools you need to deal with this
  - Pipeline barriers and events
  - Semaphores
  - Fences

Fences

- Fences let you keep track of GPU progress
- Similar to OpenGL fences
- When submitting work to the GPU, register a fence to be signalled

```
vkCreateFence(...);

vkBeginCommandBuffer(...);
vkCmdBeginRenderPass(...);
vkCmdDraw(...);
vkCmdEndRenderPass(...);
vkEndCommandBuffer(...);

vkQueueSubmit(... fence);
vkWaitForFences(... fence);
```

Semaphores

- Device-side fences
- Transfer ownership and control between queues
- Used in swapchain

```
vkQueueSubmit(queue, { .signalSemaphores = semaphore });

// Wait until GPU is done before displaying or compositing.
vkQueuePresentKHR(queue, { .waitSemaphores = semaphore });
```

Pipeline Barriers

- Within a GPU queue, commands complete out of order
  - Fragments still blend in correct order and state commands are fully in order
- Pipeline Barriers are used to enforce ordering of certain commands
- Pipeline Barriers generally have four parameters
  - Before barrier, which pipeline stages do we wait for?
  - After those stages complete, which pipeline stages do we unblock?
  - When barrier is triggered, which caches do we flush?
  - When barrier is triggered, which caches do we invalidate?

Synchronizing Render Targets

```
...
vkCmdEndRenderPass(cmd, renderToTexture);

// Resolve the hazard
VkMemoryBarrier barrier = {
    .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT, // Flush this
    .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,            // Invalidate this
};
vkCmdPipelineBarrier(cmd,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT,      // Wait for all stages
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,   // Before starting fragment
    ...,
    &memoryBarrier);                         // Then insert memory barrier

vkCmdBeginRenderPass(cmd, renderWithTexture);
...
```


Compute Shader Writes Uniform Buffer

```
vkCmdDispatch(cmd, Nx, Ny, Nz);

// Resolve the hazard
VkMemoryBarrier barrier = {
    .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT, // Flush this
    .dstAccessMask = VK_ACCESS_UNIFORM_READ_BIT, // Invalidate this
};
vkCmdPipelineBarrier(cmd,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // Wait for compute
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,  // Before starting vertex
    ...,
    &memoryBarrier);                      // Then insert memory barrier

vkCmdBeginRenderPass(cmd, renderWithUpdatedUBO);
...
```

Special Pipeline Stages

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
  - The very first stage where commands are parsed by the GPU
  - If used as srcStage, the pipeline barrier waits for nothing
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
  - Where commands retire
  - If used as dstStage, the pipeline barrier does not block any subsequent commands from executing
  - Useful for executing memory barriers without stalling subsequent commands
  - Also very useful when synchronizing with semaphores
- VK_PIPELINE_STAGE_HOST_BIT
  - For CPU readbacks
- VK_PIPELINE_STAGE_ALL_GRAPHICS/COMMANDS_BIT
  - Waits for everything

Strategies for Asynchronous GPU

- In Vulkan, the swapchain exposes a fixed number of images
  - No magic backbuffer
- Images belonging to the swapchain can be in one of three states
  - Application owned
  - GPU is rendering to it
  - Presentation engine is displaying it
- Overall goal for us is to avoid touching resources while in use by GPU
  - We want a high-level system for dealing with this in a clean way
  - We certainly do not want to track resources individually
- Need to deal with pipeline barriers
  - Semi-automatic solution seems to be a good fit

Dealing With Pipeline Barriers

- Two principle ways of dealing with hazards
  - Track invalidations for readers
  - Writers inject pipeline barriers ahead of time
- Tracking reads is painful and error prone
  - Objects are generally read many more times than they are written to
- Writers typically know future usage of objects
  - If rendering to a framebuffer, 99% of the time it will be read as a texture
  - If dispatching compute, you know where it's used later
  - If writers inject barriers right away, can forget about tracking
  - Your API abstraction can reflect this, with sensible defaults that cover the common case

```
BeginRenderPass(attachments, UsedInMemoryDomains =
MEMORY_DOMAIN_TEXTURE,
                UsedInPipelineStages = PIPELINE_STAGE_FRAGMENT);
```

Managing Pools and Memory

- Command buffers are transient in nature
  - We allocate, build and submit them in same frame
  - Reusing command buffers is not as useful as it sounds!
  - Having a central allocator for command buffers makes it very manageable
- Descriptor sets tend to be transient or completely static
  - If transient, we can allocate, write and forget the descriptor set
  - Otherwise, the descriptor set is completely static and will live for the entire program
- Freeing and reclaiming memory
  - Actually freeing memory and objects must be deferred
  - Write your own memory manager that deals with this

The Vulkan Mainloop Sketch: Start of Frame

```
VkSemaphore acquire, release; // Create these
uint32_t index;

// First, figure out which image we should render to.
vkAcquireNextImageKHR(swapchain, acquire, &index);

pContext->currentIndex = index;
pContext->setBackbuffer(pContext->pBackbuffers[index]);

// First, make sure that GPU resources are safe to reclaim.
pContext->pFenceList[index].waitAndResetAllFences();

// Command buffers, descriptor pools and memory in this frame can be recycled.
pContext->pPools[index].resetPools();
pContext->pMemoryManager->notifyGPUCompletedFrame();
pContext->replaceSemaphores(index, acquire, release);
```


The Vulkan Mainloop Sketch: End of Frame

Vulkan

```
// After building command buffers, submit them.
// We don't necessarily own the backbuffer quite yet, so we cannot
// write to it until the acquire semaphore signals.
vkQueueSubmit({
    .waitSemaphores = pContext->pAcquireSemaphores[currentIndex],

    // We only need to block writeout to the backbuffer.
    // We can still perform vertex shading safely!
    // This is extremely important for tiled GPUs!
    .waitStages = VK_PIPELINE_STAGES_COLOR_ATTACHMENT_OUTPUT_BIT,
    // When we complete our frame, signal the release semaphore.
    .signalSemaphores = pContext->pReleaseSemaphores[currentIndex],
    .signalFence = pContext->pFences[index].requestClearedFence(),
});
vkQueuePresentKHR({ .index = currentIndex,
    .waitSemaphores = pContext->pReleaseSemaphores[currentIndex] });
```


ARM

Moving to SPIR-V Shaders

- Vulkan supports shaders in SPIR-V format
  - Intermediate representation
  - Similar to LLVM IR
  - Feature set closely tied to GLSL
  - Not designed to be written by hand, but instead easy to consume for tools
  - Can just ship SPIR-V instead of GLSL in app
- Official GLSL to SPIR-V compiler available on Github
  - Suitable both as an offline tool as well as run-time library
  - https://github.com/KhronosGroup/glslang
  - Also the reference frontend for GLSL
- Opens up for new shading languages

Compiling GLSL Source to SPIR-V

```
$ cat myshader.vert

#version 310 es
layout(set = 0, binding = 0) uniform UBO {
    mat4 MVP;
};

layout(location = 0) in vec4 Position;

void main() {
    gl_Position = MVP * Position;
}

$ glslangValidator -V -o myshader.spv myshader.vert
```

Vulkan GLSL

- Vulkan introduces GL_KHR_vulkan_glsl
- Designed for offline tools, not actual OpenGL drivers
- Designed to target Vulkan and SPIR-V features
- Adds some features to GLSL
- Removes and/or changes some GLSL features
- Extends #version 140 and higher on desktop and #version 310 es for mobile content
- Can still write ES shaders with mediump support and run SPIR-V on desktop

Push Constants

- Push constants replace non-opaque uniforms
  - Think of them as small, fast-access uniform buffer memory
- Update in Vulkan with vkCmdPushConstants

```
// New
layout(push_constant, std430) uniform PushConstants {
    mat4 MVP;
    vec4 MaterialData;
} RegisterMapped;

// Old, no longer supported in Vulkan GLSL
uniform mat4 MVP;
uniform vec4 MaterialData;

// Opaque uniform, still supported
uniform sampler2D sTexture;1
```


Subpass Inputs

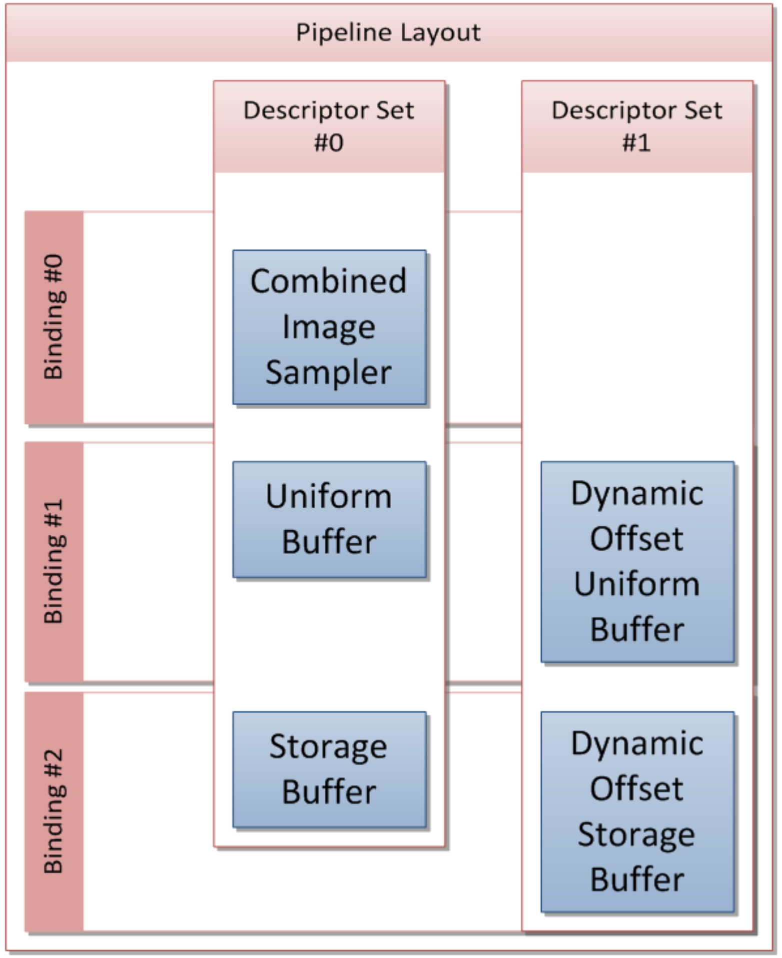
- Vulkan supports subpasses within render passes
- Standardized GL_EXT_shader_pixel_local_storage!

```
// GLSL
#extension GL_EXT_shader_pixel_local_storage : require
__pixel_local_inEXT GBuffer {
    layout(rgba8) vec4 albedo;
    layout(rgba8) vec4 normal;
    ...
} pls;

// Vulkan
layout(input_attachment_index = 0) uniform subpassInput albedo;
layout(input_attachment_index = 1) uniform subpassInput normal;
...
```

Shader Reflection in SPIR-V

- You will need to create a pipeline layout
- The layout describes which resource types are used in a pipeline
- Doing this by hand is not feasible
- Vulkan provides no built-in query interface
- Fortunately, there are free tools for this



Using Vulkan GLSL in OpenGL

- It is very likely that an engine targeting Vulkan will use Vulkan GLSL as a starting point
- A Vulkan enabled engine will likely also support OpenGL
- Vulkan GLSL is very close, but not quite compatible with GL
  - Descriptor sets not supported in GL
  - Vulkan has flat binding space compared to per-type binding spaces in GL
  - No push constants
  - Subtle differences like gl_InstanceIndex vs. gl_InstanceID
  - #ifdef VULKAN possible, but tedious and ugly

Introducing SPIR2CROSS Tool

- Developed while porting internal engine to Vulkan
  - Desire to target SPIR-V in all backends, including OpenGL ES
- Open sourced on github.com/ARM-software/spir2cross
  - Permissive open source license
- Supports full resource reflection of SPIR-V in runtime
  - Very handy for creating Vulkan pipeline layouts and set up descriptor pools automatically
- Can disassemble to readable and efficient GLSL
  - Designed to emit usable GLSL
  - Vulkan features can be remapped to GL compatible features
  - Emit both desktop and ES shaders, can also emit to ES 2.0
  - Full support for vertex, fragment, tessellation, geometry and compute shaders

SPIR2CROSS Example

```
// myshader.frag
#version 310 es
precision mediump float;
layout(binding = 0) uniform sampler2D sTexture;
layout(location = 0) in vec2 vTexCoord;
layout(location = 0) out vec4 FragColor;
void main() {
    FragColor = texture(sTexture, vTexCoord);
}

// Compile to SPIR-V
$ glslangValidator -H -V -o myshader.spv myshader.frag
```

Disassemble Back to GLSL

```
$ spir2cross myshader.spv --version 310 --es --dump-resources
ID 017 : vTexCoord (Location : 0) // Inputs
ID 009 : FragColor (Location : 0) // Outputs
ID 013 : sTexture (Set : 0) (Binding : 0) // Textures

#version 310 es
precision mediump float;
precision highp int;

layout(binding = 0) uniform mediump sampler2D sTexture;
layout(location = 0) out vec4 FragColor;
layout(location = 0) in vec2 vTexCoord;

void main()
{
    FragColor = texture(sTexture, vTexCoord);
}
```


Thank you!

ARM

To Find Out More….

ARM Booth #1624 on Expo Floor:

- Live demos of the techniques shown in this session
- In-depth Q&A with ARM engineers
- More tech talks at the ARM Lecture Theatre

http://malideveloper.arm.com/gdc2016:

- Revisit this talk in PDF and video format post GDC
- Download the tools and resources

More Talks From ARM at GDC 2016

Available post-show at the Mali Developer Center: malideveloper.arm.com/

Vulkan on Mobile with Unreal Engine 4 Case Study

Weds. 9:30am, West Hall 3022

Making Light Work of Dynamic Large Worlds

Weds. 2pm, West Hall 2000

Achieving High Quality Mobile VR Games

Thurs. 10am, West Hall 3022

Optimize Your Mobile Games With Practical Case Studies

Thurs. 11:30am, West Hall 2404

An End-to-End Approach to Physically Based Rendering

Fri. 10am, West Hall 2020