High Performance Vulkan

Lessons Learned from Source 2
John McDonald
Sections

- Introduction and Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Disclaimer

- Largely based on Dan Ginsburg’s Siggraph 2015 Vulkan talk, but updated for 1.0.
- 3385 changes to Vulkan since that talk.
  - So hopefully not a rehash even if you were there 😊
- Slides may be buggy!
- Guidance based on Desktop GPUs (AMD, Intel, NVIDIA)
  - Everything should work on mobile GPUs, but may not be optimal performance
Goals

- Thorough understanding of Vulkan concepts
- Concrete examples to follow for common resource updates
Goals

- Thorough understanding of Vulkan concepts
- Concrete examples to follow for common resource updates
- Avoid repeating our mistakes
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Source 2 Overview

- OpenGL, Direct3D 9, Direct3D 11, Vulkan
- Windows, Linux, Mac
- Dota 2 Reborn
Source 2 Rendering

- Direct3D 11-like Render system abstraction
- Multithreaded
  - D3D9/GL: software command buffers
  - D3D11: deferred contexts
  - Single submission thread
Source 2 Rendering (GL)
Source 2 Rendering (GL)
Source 2 Vulkan Port

- Started with GL and D3D11 renderer
  - D3D11 deferred contexts mapped well to Vulkan command buffers
  - Leveraged GLSL shader conversion already in GL layer
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
General Guidance

- Setting up the Validation layer should be part of day 1 tasks.
  - Validation should be enabled all through development
  - Without Validation, Vulkan **will not** report errors back to you, most likely will simply crash if given invalid parameters / commands.
- Spend a little time investing in app thread debugging tools
  - Single Threaded Job mode
  - Heap Validation tools (malloc/debug)
  - CPU-GPU sync point detection
- Invest in tools that can visualize threads well
General Guidance - Threading

- Threads should be as independent as possible
- Any cross talk between threads significantly reduces benefit from multithreading
General Guidance - Threading

- Threads should be as independent as possible
- Any cross talk between threads significantly reduces benefit from multithreading

Bad threading is worse than no threading
General Guidance Caveats

- However, per-thread resources lead to memory bloat
  - Over time, all threads consume the same large memory footprint
- Conceptually a form of internal fragmentation
- Can be difficult to stay on top of
Internal fragmentation visualized

Thread 0
Thread 1
Thread 2
Thread 3

Useful
Waste
Internal fragmentation visualized

- Thread 0
- Thread 1
- Thread 2
- Thread 3

Useful
Waste
Internal fragmentation visualized

- Thread 0
- Thread 1
- Thread 2
- Thread 3

Useful
Waste
Internal fragmentation visualized

- Thread 0
- Thread 1
- Thread 2
- Thread 3

Useful
Waste
Internal fragmentation visualized

- Thread 0
- Thread 1
- Thread 2
- Thread 3

Useful
Waste
Internal Fragmentation Consumption

- **Ideal**
  - Total = Sum( Per Thread Memory )

- **Worst Case**
  - Total = Max( Per Thread Memory ) * Num Threads

- Possible Solution Later
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Command Buffer Core Concepts

- Conceptually similar to D3D11 Deferred Context
- Inherit no* state from other command buffers
- `VkCommandBuffer`s are central to the split between specification of work and scheduling of work.
  - Between `vkBeginCommandBuffer` and `vkEndCommandBuffer`, a command buffer is said to be recording. This is the specification of work.
  - Once a Command Buffer has been Ended, it can be scheduled for execution with `vkQueueSubmit`.
Command Buffer Core Concepts

- Conceptually similar to D3D11 Deferred Context
- Inherit no* state from other command buffers
- `VkCommandBuffer`s are central to the split between specification of work and scheduling of work.
  - Between `vkBeginCommandBuffer` and `vkEndCommandBuffer`, a command buffer is said to be recording. This is the specification of work.
  - Once a Command Buffer has been Ended, it can be scheduled for execution with `vkQueueSubmit`.

* Except Renderpass information between primary->secondary command buffers, which are beyond the scope of this presentation.
Command Buffer Core Concepts

- A `VkCommandBuffer` is allocated from a `VkCommandPool` via a call to `vkAllocateCommandBuffers`.
- `VkCommandPool`s, like other pools, allow for lock-free allocation.
- Each `VkCommandBuffer`—and `VkCommandPool`—is "externally synchronized." This means the application promises not to act on the same Command Buffer or Command Pool from two threads simultaneously.
Command Buffers in Source 2

- Used where D3D11 deferred contexts were used
- One `VkCommandBuffer` per thread per render target
  - Except full screen passes—one command buffer only
- Single thread performs submission to queue
Command Buffer Allocation and Reuse

- ResourcePool is per thread. The ResourcePool spills to \texttt{VkCommandPool} (per thread)
- ResourcePool is per thread. The ResourcePool spills to `VkCommandPool` (per thread)

[Diagram showing resource pool allocation and reuse for threads 0, 1, and 2, with cmdbufs for each thread.]
- ResourcePool is per thread. The ResourcePool spills to VkCommandPool (per thread)
Command Buffer Allocation and Reuse

- ResourcePool is per thread. The ResourcePool spills to *VkCommandPool* (per thread)
ResourcePool is per thread. The ResourcePool spills to *VkCommandPool* (per thread)
Command Buffer Allocation and Reuse

- ResourcePool is per thread. The ResourcePool spills to `VkCommandPool` (per thread)
Command Buffer Allocation and Reuse

- ResourcePool is per thread. The ResourcePool spills to `VkCommandPool` (per thread).

### Diagram

- **Thread 0**: ResourcePool
- **Thread 1**: ResourcePool
- **Thread 2**: ResourcePool

### Timeline

- **Submit**
- **Main**

**Time**

---

---
All work passed to Submit thread via ThreadSafe Queue. Ordering is effectively arbitrary.
Command Buffer Allocation and Reuse

- All work passed to Submit thread via ThreadSafe Queue. Ordering is effectively arbitrary.

```
Thread 0  ResourcePool
Thread 1  ResourcePool
Thread 2  ResourcePool
```

```
<table>
<thead>
<tr>
<th>Thread0</th>
<th>CmdBuf</th>
<th>CmdBuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread1</td>
<td>CmdBuf</td>
<td>CmdBuf</td>
</tr>
<tr>
<td>Thread2</td>
<td>CmdBuf</td>
<td>CmdBuf</td>
</tr>
<tr>
<td>Submit</td>
<td></td>
<td>Submit</td>
</tr>
<tr>
<td>Main</td>
<td></td>
<td>Time</td>
</tr>
</tbody>
</table>
```
Command Buffer Allocation and Reuse

- All work passed to Submit thread via ThreadSafe Queue. Ordering is effectively arbitrary.

Thread 0
ResourcePool

Thread 1
ResourcePool

Thread 2
ResourcePool

Thread0
CmdBuf
CmdBuf

Thread1
CmdBuf
CmdBuf

Thread2
CmdBuf
CmdBuf
CmdBuf

Submit
Submit

At least one frame

Main

Time
Command Buffer Allocation and Reuse

- Recycle `VkCommandBuffer` back to the thread they came from for next time.
Submit in batches
- `vkQueueSubmit` flushes—so generally 1-2 per frame max.
- Faster to group submissions together

Minimize number of command buffers
- We are still investigating how to decrease our count

Use `VK_CMD_BUFFER_OPTIMIZE_ONE_TIME_SUBMIT_BIT`
- Optimize for one-time submission
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Pipeline Core Concepts

- All Pipeline state data is encapsulated in a reusable `VkPipeline` object.
  - Which Shaders are set and active
  - Stencil Test
  - Scissor Mode
  - Blend State
  - etc
- A `VkPipeline` is created by calling `vkCreateGraphicsPipelines` or `vkCreateComputePipelines`.
  - Creation of these may be very expensive, the cost may be alleviated by using `VkPipelineCache` objects.
- Once created a `VkPipeline` is immutable, but can be used from many threads at once.
**VkPipelineCache**

- **VkPipelineCache** is ARB_program_binary for Vulkan.
- Load from disk
- Give to Vulkan
- Makes **VkPipeline** creation magically faster (~20x faster)
- Before shutdown
  - Ask size
  - Get bytes from Vulkan
  - Write to disk

- Dota 2 Reborn’s **VkPipelineCache** is ~4 MB on disk.
Efficient Pipeline Construction

- Problem: Need efficient access to `VkPipeline` from many threads at once
- Bonus Problem: `VkPipeline` may be large, don’t want to multiply cost by N threads.
- Solution: Two Tiered Cache
  - Two Pipeline Maps
    - 1 Read Only Pipeline Map (Current)
    - 1 Read/Write Pipeline Map (Pending)
Pipeline State Creation Visualized

- Application is issuing D3D11-like rendering commands…

Pipeline State Hash
At draw time, state is hashed. Current cache is checked for existing `VkPipeline`.
Pipeline State Creation Visualized

- If doesn’t exist in Current, lock is grabbed and Pending cache is checked.
Pipeline State Creation Visualized

- Still can’t find it—create now.

Pipeline State Hash → Create Pipe

Current Pipe Map:
- Pipe0
- Pipe1
- Pipe2

Pending Pipe Map:
- Pipe3 (mutexed)
Pipeline State Creation Visualized

- Push into Pending Pipe Map, and return to caller

Pipeline State Hash

Create Pipe

Current Pipe Map

Pending Pipe Map

Pipe0
Pipe1
Pipe2
Pipe3

(mutexed)
During serial portion of frame, Current is updated with the contents of Pending.
Two Tier Pipeline Caching System

- This somewhat elaborate strategy reduces mutexing tremendously (to nearly 0 once we’re a few frames in)
- Room for improvement
Two Tier Pipeline Caching System

- This somewhat elaborate strategy reduces mutexing tremendously (to nearly 0 once we’re a few frames in)
- Room for improvement
  - Currently, only one \texttt{VkPipeline} will be created at a time (because of the mutex)
Two Tier Pipeline Caching System

- This somewhat elaborate strategy reduces mutexing tremendously (to nearly 0 once we’re a few frames in)
- Room for improvement
  - Currently, only one `VkPipeline` will be created at a time (because of the mutex)
  - Could create Reservation for pending `VkPipeline` but release mutex
  - Other threads would only block if they were waiting for that specific Reservation to be satisfied.
Two Tier Pipeline Caching System

- This somewhat elaborate strategy reduces mutexing tremendously (to nearly 0 once we’re a few frames in)
- Room for improvement
  - Currently, only one `VkPipeline` will be created at a time (because of the mutex)
  - Could create Reservation for pending `VkPipeline` but release mutex
  - Other threads would only block if they were waiting for that specific Reservation to be satisfied.
- In practice, we don’t see this causing stalls, so haven’t implemented—largely because `VkPipelineCache` makes construction cheap after the first run
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Descriptor Core Concepts

- No exact analog in D3D9/11.
- Shader accessible resources are represented by `VkDescriptorS`
  - `VkDescriptor` are arranged in Sets
  - Sets are allocated from Pools
  - Sets have Layouts, known at Pipeline creation time
    ```
    vkCreateDescriptorPool(...);
    vkCreateDescriptorSetLayout(...);
    vkAllocDescriptorSets(...);
    ```

**Descriptor Core Concepts**

- Shader accessible resources are represented by `VkDescriptorS`
  - `VkDescriptor` are arranged in Sets
  - Sets are allocated from Pools
  - Sets have Layouts, known at Pipeline creation time
    ```
    vkCreateDescriptorPool(...);
    vkCreateDescriptorSetLayout(...);
    vkAllocDescriptorSets(...);
    ```
  
- Layouts can be thought of as shader ABI.
- DescriptorSet is a concrete set of parameters being passed into the shader.

---

Descriptor Set Updates - Ideal

- Allocate and bake descriptor sets up front
- Group sets by frequency
- Only update changed sets (which would be “dynamic”)
Descriptor Set Updates - Ideal

- Allocate and bake descriptor sets up front
- Group sets by frequency
- Only update changed sets (which would be “dynamic”)

- This has been roughly the advice since Direct3D 10
- But APIs don’t require it (nor does Vulkan)
DescriptorSet - Reality

- Difficult to bake descriptors with Direct3D 11-like abstraction
- Our approach
  - Pre-allocate descriptor sets with fixed slots
  - Only bind to used slots
  - Update descriptors each draw
DescriptorSet - Reality

- Difficult to bake descriptors with Direct3D 11-like abstraction
- Our approach
  - Pre-allocate descriptor sets with fixed slots
  - Only bind to used slots
  - Update descriptors each draw

- This is a performance problem for us, but not yet clear how much.
How Much Performance Lost?

- Not actually sure.
- To answer the question requires changing the abstraction.
  - Chicken $\Rightarrow$ Egg $\Rightarrow$ Chicken $\Rightarrow$ …
- And changing the abstraction requires significant effort for unknown gains.
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
General Strategies

- Pool resources together
  - Sub-allocate from large pools
- Use per-thread pools to reduce contention
- Recycle dynamic pools on frame boundaries
## Resources – Current strategy

<table>
<thead>
<tr>
<th>Static Resources</th>
<th>Dynamic Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Global Pools</td>
<td>• Per-Thread Pools</td>
</tr>
<tr>
<td>• Device Only</td>
<td>• Host Visible (Persistently Mapped)</td>
</tr>
<tr>
<td>• Textures/Render Targets</td>
<td>• VB/IB/CBs</td>
</tr>
<tr>
<td>• 128MB Pools</td>
<td>• 8MB Pools</td>
</tr>
<tr>
<td>• VB/IB/CBs</td>
<td></td>
</tr>
<tr>
<td>• 8MB Pools</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Vertex/Index Buffers

- To update:
  - Grab new offset from per-thread pool
  - memcpy into pool
  - Bind VBs with: `vkCmdBindVertexBuffers(..,buffer,offset)`
  - Bind IBs with: `vkCmdBindIndexBuffer(..,buffer,offset,..)`

- Recycle pools when last GPU fence of frame retires

http://www.slideshare.net/basisspace/efficient-buffer-management
Dynamic Uniform Buffers

- **Differences from VB/IBs:**
  - UBOs are bound via descriptors
  - Use dynamic UBOs to avoid `vkUpdateDescriptorSets`
  - Pass UBO offset to `vkCmdBindDescriptorSets`
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Image Core Concepts

- ID3D11Texture == **VkImage**
- Images are created in fiveish steps:
  1. The image header is created using `vkCreateImage`
  2. The data store is suballocated from an appropriate pool
  3. The data store is bound to the header via `vkBindImageMemory`
  4. The image is transitioned to `VK_IMAGE_LAYOUT_GENERAL`
  5. The image is filled from a staging buffer using `vkCmdCopyBufferToImage`

- Initial (pool) allocation can be expensive (100s μs).
  - Suballocation is cheap (100s ns)
  - Binding also cheap (100s ns)
- Can reuse same data store for different purposes in different parts of the frame.
  - Huge memory savings!
Glossed over “some” details in step 5.
Fiveish?

- Glossed over “some” details in step 5.
Glossed over “some” details in step 5.
Glossed over “some” details in step 5.
Glossed over “some” details in step 5.
Glossed over "some" details in step 5.
Glossed over “some” details in step 5.
Fiveish?

- Glossed over "some" details in step 5.
Glossed over “some” details in step 5.
Fiveish?

- Glossed over "some" details in step 5.

```c
// Allocate GPU memory for staging resource
VkDeviceMemory pGPUStagingMemory = VK_NULL_HANDLE;
VkBuffer pStagingBuffer = VK_NULL_HANDLE;
g_pRenderDeviceVulkan->MemoryManager()->Allocate(
    nDataSize,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT,
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    &pGPUStagingMemory,
    &pStagingBuffer);
Assert( pGPUStagingMemory );

// Copy to the GPU staging memory
void *pStagingMemory = NULL;
DOVK( vkMapMemory( m_pDeviceVulkan, pGPUStagingMemory, 0, nDataSize, 0, ( void** ) &pStagingMemory ) );
Assert( pStagingMemory != NULL );
V_memcpy( pStagingMemory, pData, nDataSize );

vkUnmapMemory( m_pDeviceVulkan, pGPUStagingMemory );
```
Fiveish?

- Glossed over "some" details in step 5.

```c
// Allocate GPU memory for staging resource
VkDeviceMemory pGPUStagingMemory = VK_NULL_HANDLE;
VkBuffer pStagingBuffer = VK_NULL_HANDLE;
g_pRenderDeviceVulkan->MemoryManager()->Allocate( nDataSize,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT,
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    &pGPUStagingMemory,
    &pStagingBuffer );

Assert( pGPUStagingMemory );

// Copy to the GPU staging memory
void *pStagingMemory = NULL;
DOVK( vkMapMemory( m_pDeviceVulkan, pGPUStagingMemory, 0, nDataSize, 0, ( void** ) &pStagingMemory ) );
Assert( pStagingMemory != NULL );

V_memcopy( pStagingMemory, pData, nDataSize );
vkUnmapMemory( m_pDeviceVulkan, pGPUStagingMemory );
```
Glossed over "some" details in step 5.
Mo’ Copies Mo’ Problems

- Asking to do a large allocation just-in-time (staging buffer)
- Extra copies of texture data make me sad
  1. Load in raw bits from disk into host memory.
  2. Convert to usable GPU format.
  3. Copy into Staging Buffer.
- This could be reduced by 1 (and usually 2 copies)!
  - ie, read directly from disk into a GPU accessible buffer!
Better Image Data Downloads

- Apply GL Concepts to Vulkan
  - Pixel Buffer Objects (PBO)
  - Persistently Mapped Buffers (PMB)
- PBO + PMB = “malloc’d” staging memory
- Suballocation wins again!
Better Image Data Downloads

- In the Renderer
  - At the beginning of time, allocate a single staging buffer with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_BUFFER_USAGE_TRANSFER_SRC_BIT`.
  - Map it

- In loading code
  - Read in image header from disk
  - Ask Renderer whether format can be directly consumed
  - If so, ask Renderer for allocation to read image bytes into
  - Once data has been read, notify Renderer

- No extra maps!
- No large allocations during image streaming!
Better Image Data Downloads

- **In the Renderer**
  - At the beginning of time, allocate a single staging buffer with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_BUFFER_USAGE_TRANSFER_SRC_BIT`.
  - Map it

- **In loading code**
  - Read in image header from disk
  - Ask Renderer whether format can be directly consumed
  - If not, read image bytes into temporary buffer
    - Then ask Renderer for allocation to read converted bytes
    - Convert from temporary buffer to Renderer allocation
  - Once data has been converted, notify Renderer

- No extra maps!
- No large allocations during image streaming!
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Remember this?

Internal Fragmentation

- Thread 0
- Thread 1
- Thread 2
- Thread 3

Useful
Waste

0 50 100 150

Dimensions: 720.0x540.0
- Remember this?
- Can’t solve for everything (notably opaque pools)
- But can make better for per-thread pools (dynamic VB/IB/CB)
Improving Internal Fragmentation

- **In the Renderer (dawn of time)**
  - At the beginning of time, allocate a single staging buffer with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_BUFFER_USAGE_TRANSFER_SRC_BIT`.
  - Map it, store pointer as `pBase`
  - Create a head offset, started at 0.

- **In the Renderer (on the fly)**
  - When asked for an allocation, atomic increment head by the size of the allocation.
    - If returned value + allocSize < totalSize, we can simply return `pBase + returned Value` for writer
    - If not, decide whether to create a new pool (requires lock), stall waiting for more data (CPU-GPU sync point) or return NULL.
  - Recycle

```c
LONG InterlockedAdd( LONG volatile *Addend, LONG Value );
```
Sections

- Goals
- Source 2 Overview
- General Guidance
- Command Buffers
- Pipelines
- Descriptor Set Updates
- Memory Management
- Image Management
- Internal Fragmentation
- Final Thoughts
Final thoughts

- Vulkan provides a modern, multithreaded API to GPU programming
- In exchange for peak performance and proportional taxation, Vulkan requires more from applications
Final thoughts

- Vulkan provides a modern, multithreaded API to GPU programming
- In exchange for peak performance and proportional taxation, Vulkan requires more from applications
- Something power, something responsibility.
Final thoughts

- Vulkan provides a modern, multithreaded API to GPU programming
- In exchange for peak performance and proportional taxation, Vulkan requires more from applications

- Something power, something responsibility.

- Questions?
  - mcjohn at valvesoftware dot com
  - @basisspace on twitter