Porting to Vulkan

Lessons Learned

Who am I?

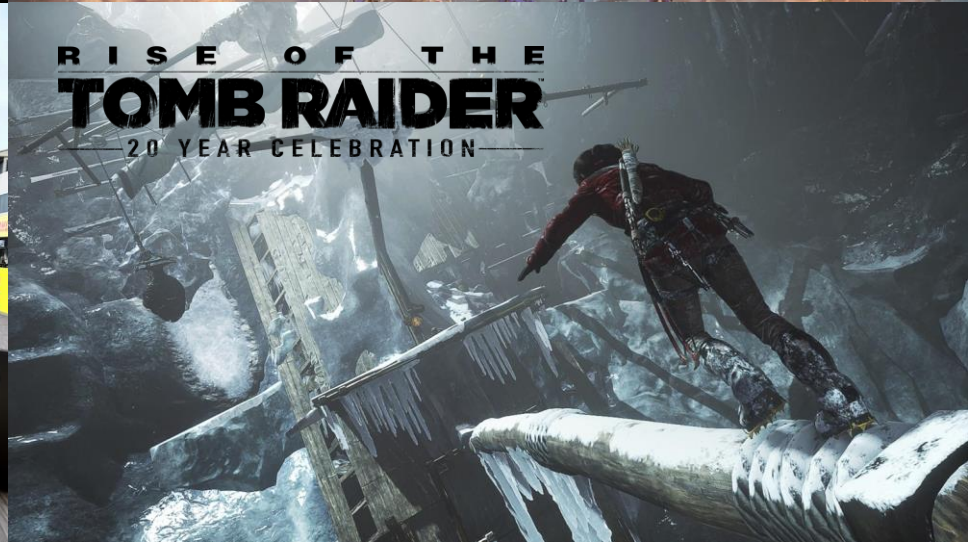
Feral Interactive - Mac/Linux/Mobile games publisher and porter

Alex Smith - Linux Developer, led development of Vulkan support


MAD MAX


WARHAMMER 40,000
DAWN OF WAR III


PIRELLI
GOODYEAR
elf
Canon
F1 2017
Formula 1


RISE OF THE
TOMB RAIDER
20 YEAR CELEBRATION

Vulkan Releases

- **Mad Max**
  - Originally released using OpenGL in October 2016
  - Beta Vulkan patch in March 2017
  - Vulkanised 2017 talk “Driving Change: Porting Mad Max to Vulkan”

- **Warhammer 40,000: Dawn of War III**
  - Released in June 2017
  - OpenGL by default, Vulkan as experimental option

- **F1 2017**
  - Released in November 2017
  - First Vulkan-exclusive title

- **Rise of the Tomb Raider**
  - Released in April 2018
  - Vulkan-exclusive

From Beta to Production

- First two beta releases weren’t production quality
- Gave us a lot of feedback
  - Had an email address for users to report problems to us
  - Driver configuration issues
  - Hardware-specific issues
  - Big help in avoiding issues for Vulkan-exclusive releases
- Many improvements made - will be detailing some of these:
  - Memory management
  - Descriptor sets
  - Threading

Memory Management

- Biggest area which needed improvement to become production quality
- Problem areas:
  - Overcommitting VRAM
  - Fragmentation

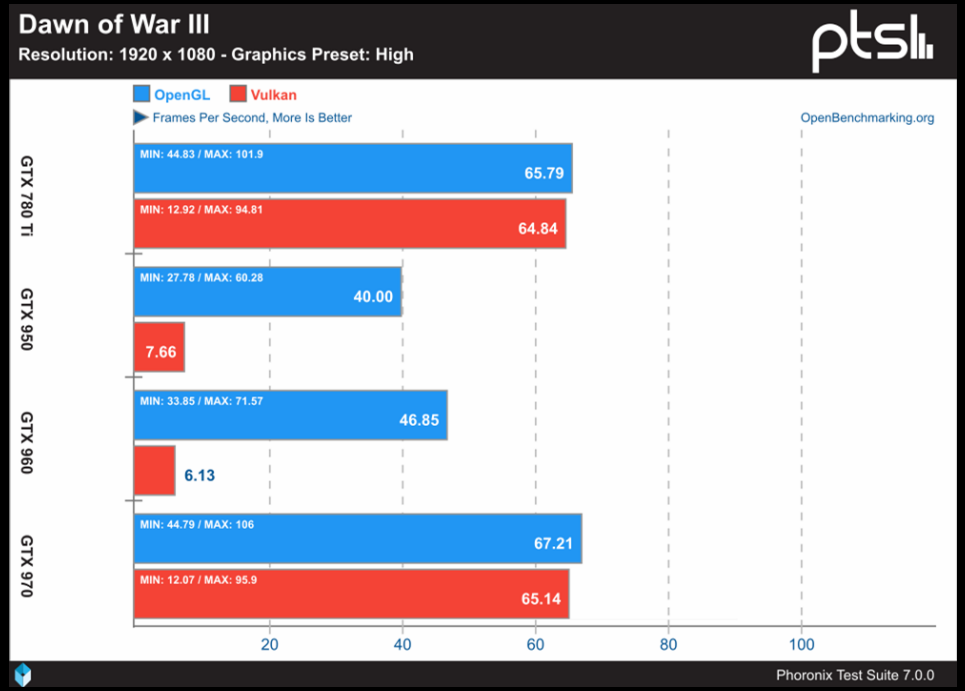
Overcommitting VRAM

- Can happen from users playing with higher graphics settings than they have enough VRAM for
  - Don’t want to just crash in this case - it can still be made to perform reasonably well
  - We try to allow this, within reason

- Driver is not going to handle it for you!
  - When you exhaust available space in a heap, vkAllocateMemory() will fail
    - On Linux AMD/NV/Intel at least, may differ on other platforms
  - Have to handle this, e.g. if allocation from a DEVICE_LOCAL heap fails, fall back to a host heap

- Doing it naively can cause performance problems

Overcommitting VRAM



Source: https://www.phoronix.com/scan.php?page=article&item=dow3-linux-perf&num=4

Overcommitting VRAM

- DoW3 loads all of its textures and other resources on a loading screen
- Render targets and GPU-writable buffers are allocated after, once it starts rendering
- On 2GB GPUs, higher texture quality settings use up most of VRAM
- Behaviour after a device local allocation failure was always to just fall back to a host heap
  - Textures have already filled up the available device space
  - Render target allocations fail, so get placed in host heap instead
  - Say goodbye to your performance!

Overcommitting VRAM

- Solution: require render targets and GPU-writable buffers to be placed in VRAM
- If we fail to allocate, try to make space:
  - Defragment (discussed later)
  - Move other resources to the host heap
- Doing this brought DoW3's Vulkan performance in line with GL when VRAM-constrained
- Useful to have a way to simulate having less VRAM for testing
  - Heap size limit: behaves as though sizes given by VkPhysicalDeviceMemoryProperties are smaller
  - Early failure limit: behaves as though vkAllocateMemory() fails when less is used than the reported heap size
    - In real usage this will fail early due to VRAM usage by the OS, other apps, etc.

Fragmentation

- We allocate large device memory pools and manage these internally
  - Generally the recommended memory management strategy on Vulkan
  - vk(Allocate|Free)Memory() are expensive!

- Over time, these can become fragmented
  - Due to resource streaming, etc.
  - Resources end up spread across multiple pools with gaps in between

- Memory usage becomes higher than it needs to be
  - More pools are allocated
  - Pools can’t be freed while they still have any resources in them


Vulkan Memory
Defrag
Heaps
Heap 0
Size: 4096 MiB (4294967296 bytes) Limit Hidden Limit Reset
Allocated: 1656 MiB (1737191552 bytes, 40.447143 % of heap)
Used: 1584 MiB (1661171075 bytes, 95.623944 % of allocated)
Flags:
DEVICE_LOCAL_BIT
Heap 1
Types
Type 0
Type 1
Type 2
Type 3
Type 4
Type 5
Type 6
Type 7
Heap: 0
Domain: Device
Flags:
DEVICE_LOCAL_BIT
Resource Types:
Buffer
Allocated: 410 MiB (429916160 bytes)
Used: 402 MiB (422295339 bytes, 98.227370 % of allocated)
Allocations
UniformBuffer
Allocated: 8 MiB (8388608 bytes)
Used: 1 MiB (1212656 bytes, 14.455986 % of allocated)
Allocations
WritableBuffer
Allocated: 128 MiB (135184384 bytes)
Used: 120 MiB (126372072 bytes, 93.481265 % of allocated)
Allocations
Image
Allocated: 768 MiB (805306368 bytes)
Used: 718 MiB (752894976 bytes, 93.491745 % of allocated)
Allocations
Allocation 0 (128 MiB)
Allocation 1 (128 MiB)
Allocation 2 (128 MiB)
Allocation 3 (128 MiB)
Allocation 4 (128 MiB)
Allocation 5 (128 MiB)
RenderTarget
Allocated: 341 MiB (358396032 bytes)
Used: 341 MiB (358396032 bytes, 100.000000 % of allocated)
Allocations
Type 8
Type 9
Type 10
Resource Sets

▼ Vulkan Memory

Defrag

▼ Heaps

▼ Heap 0

Size: 4096 MiB (4294967296 bytes) Limit Hidden Limit Reset

Allocated: 1818 MiB (1906788992 bytes, 44.395891 % of heap)

Used: 1465 MiB (1536695699 bytes, 80.590758 % of allocated)

Flags:

• DEVICE_LOCAL_BIT

▶ Heap 1

▼ Types

▶ Type 0

▶ Type 1

▶ Type 2

▶ Type 3

▶ Type 4

▶ Type 5

▶ Type 6

▼ Type 7

Heap: 0

Domain: Device

Flags:

• DEVICE_LOCAL_BIT

Resource Types:

• Buffer

• Allocated: 410 MiB (429916160 bytes)

• Used: 402 MiB (422295435 bytes, 98.227393 % of allocated)

▶ Allocations

• UniformBuffer

• Allocated: 8 MiB (8388608 bytes)

• Used: 0 MiB (781984 bytes, 9.321976 % of allocated)

▶ Allocations

• WritableBuffer

• Allocated: 128 MiB (135184384 bytes)

• Used: 120 MiB (126372072 bytes, 93.481265 % of allocated)

▶ Allocations

• Image

• Allocated: 896 MiB (939524096 bytes)

• Used: 565 MiB (593470464 bytes, 63.167136 % of allocated)

▼ Allocations

▼ Allocation 0 (128 MiB)

▼ Allocation 1 (128 MiB)

▼ Allocation 2 (128 MiB)

▼ Allocation 3 (128 MiB)

▼ Allocation 4 (128 MiB)

▼ Allocation 5 (128 MiB)

▼ Allocation 6 (128 MiB)

• RenderTarget

• Allocated: 375 MiB (393775744 bytes)

• Used: 375 MiB (393775744 bytes, 100.000000 % of allocated)

▶ Allocations

▶ Type 8

▶ Type 9

▶ Type 10

Fragmentation

- Solution: implemented a memory defragmenter
  - Moves resources around to compact them into as few pools as possible
  - Free pools which become empty as a result

- F1 2017: done at fixed points, fully defragments all allocated memory
  - During loading screens
  - When we're struggling to allocate memory for a new resource

- Rise of the Tomb Raider: also done periodically in the background
  - Semi-open world, infrequent loading screens
  - Tries to keep the amount of memory actually used versus the total size of the pools above a threshold
  - Rate-limited to avoid having too much impact on performance

▼ Vulkan Memory

Defrag

▼ Heaps

▼ Heap 0

Size: 4096 MiB (4294967296 bytes) Limit Hidden Limit Reset

Allocated: 1818 MiB (1906788992 bytes, 44.395891 % of heap)

Used: 1465 MiB (1536695699 bytes, 80.590758 % of allocated)

Flags:

• DEVICE_LOCAL_BIT

▶ Heap 1

▼ Types

▶ Type 0

▶ Type 1

▶ Type 2

▶ Type 3

▶ Type 4

▶ Type 5

▶ Type 6

▼ Type 7

Heap: 0

Domain: Device

Flags:

• DEVICE_LOCAL_BIT

Resource Types:

• Buffer

• Allocated: 410 MiB (429916160 bytes)

• Used: 402 MiB (422295435 bytes, 98.227393 % of allocated)

▶ Allocations

• UniformBuffer

• Allocated: 8 MiB (8388608 bytes)

• Used: 0 MiB (781984 bytes, 9.321976 % of allocated)

▶ Allocations

• WritableBuffer

• Allocated: 128 MiB (135184384 bytes)

• Used: 120 MiB (126372072 bytes, 93.481265 % of allocated)

▶ Allocations

• Image

• Allocated: 896 MiB (939524096 bytes)

• Used: 565 MiB (593470464 bytes, 63.167136 % of allocated)

▼ Allocations

▼ Allocation 0 (128 MiB)

▼ Allocation 1 (128 MiB)

▼ Allocation 2 (128 MiB)

▼ Allocation 3 (128 MiB)

▼ Allocation 4 (128 MiB)

▼ Allocation 5 (128 MiB)

▼ Allocation 6 (128 MiB)

• RenderTarget

• Allocated: 375 MiB (393775744 bytes)

• Used: 375 MiB (393775744 bytes, 100.000000 % of allocated)

▶ Allocations

▶ Type 8

▶ Type 9

▶ Type 10

Vulkan Memory

Defrag

Heaps

Heap 0

Size: 4096 MiB (4294967296 bytes) Limit Hidden Limit Reset

Allocated: 1554 MiB (1629964928 bytes, 37.950578 % of heap)

Used: 1470 MiB (1542037907 bytes, 94.605588 % of allocated)

Flags:

- DEVICE_LOCAL_BIT

Heap 1

Types

Type 0

Type 1

Type 2

Type 3

Type 4

Type 5

Type 6

Type 7

Heap: 0

Domain: Device

Flags:

- DEVICE_LOCAL_BIT

Resource Types:

- Buffer
  - Allocated: 410 MiB (429916160 bytes)
  - Used: 402 MiB (422295435 bytes, 98.227393 % of allocated)
  - Allocations
- UniformBuffer
  - Allocated: 8 MiB (8388608 bytes)
  - Used: 0 MiB (781984 bytes, 9.321976 % of allocated)
  - Allocations
- WritableBuffer
  - Allocated: 120 MiB (126795776 bytes)
  - Used: 120 MiB (126372072 bytes, 99.665837 % of allocated)
  - Allocations
- Image
  - Allocated: 640 MiB (671088640 bytes)
  - Used: 571 MiB (598812672 bytes, 89.230042 % of allocated)
  - Allocations
    - Allocation 0 (128 MiB)
    - Allocation 1 (128 MiB)
    - Allocation 2 (128 MiB)
    - Allocation 3 (128 MiB)
    - Allocation 4 (128 MiB)
- RenderTarget
  - Allocated: 375 MiB (393775744 bytes)
  - Used: 375 MiB (393775744 bytes, 100.000000 % of allocated)
  - Allocations

Type 8

Type 9

Type 10

Resource Sets

Descriptor Sets

- Initial implementation rewrote descriptors per-draw every frame
  - Per-frame descriptor pools
  - Reuse with vkResetDescriptorPool() once frame fence completed
- Worked reasonably well on desktop
- Very costly on some mobile implementations

Descriptor Sets

- New strategy: persistent descriptor sets, generated and cached as needed
- Look up using a key based on the bound resources
- Use (UNIFORM|STORAGE)_BUFFER_DYNAMIC descriptors
  - Works well with ring buffers for frequently updated constants
  - Just bind existing set with the offset of the latest data, no need to update or create from scratch
- Performance results over original implementation:
  - Up to 5% improvement on desktop in Rise of the Tomb Raider benchmark
  - ~30% improvement on Arm Mali in GRID Autosport benchmark

Descriptor Sets

- Descriptor pools are created as needed when existing pools are empty

- Need to keep an eye on how many sets/pools you have at a time
  - They can have a VRAM cost
  - No API to check, but can manually calculate when driver source available (e.g. AMD)
  - Could reach ~50MB used by pools in RotTR on AMD
  - Periodically free sets which haven't been used in a while – reduced to ~20MB

- Freeing individual sets can lead to pool fragmentation
  - Allocations from pools occasionally fail when this happens
  - In practice hasn't been found to be much of a problem

Threading

- Vulkan gives much greater opportunity for multithreading
- Use for resource creation and during rendering

Threading - Pipeline Creation

- On Vulkan, unless you have few pipelines, it’s best to create them ahead of time rather than as needed at draw time, to avoid stuttering
- Pipelines can be created on multiple threads simultaneously
- Our previous OpenGL releases have often had loading screens to pre-warm shaders
  - Can be several minutes (when driver cache is clear) for games with lots of shaders
- Rise of the Tomb Raider has a lot of pipeline states (10s of thousands)
  - Semi-open world, few loading screens to be able to create them on
  - Too many to pre-create at startup in a reasonable time
  - Have VkPipelineCache/driver-managed caches, but still care about the first-run experience

Threading - Pipeline Creation

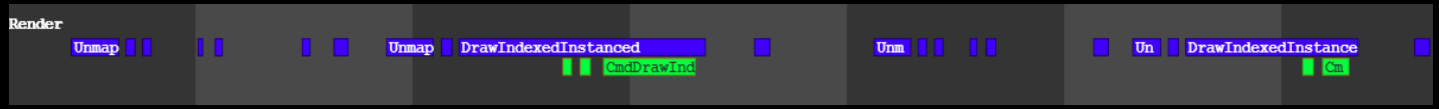
- Create pipelines for current area using multiple threads during initial load
  - Use (core count - 1) threads
  - Pipeline creation generally scales very well the more threads you use

- Continue to create pipelines for surrounding areas on a background thread during gameplay
  - Set priority lower to reduce impact on the rest of the game

- In many cases pipeline creation completes within the time taken to load everything else for an area
  - Rarely end up on a loading screen waiting exclusively for pipeline creation

Threading - Rendering

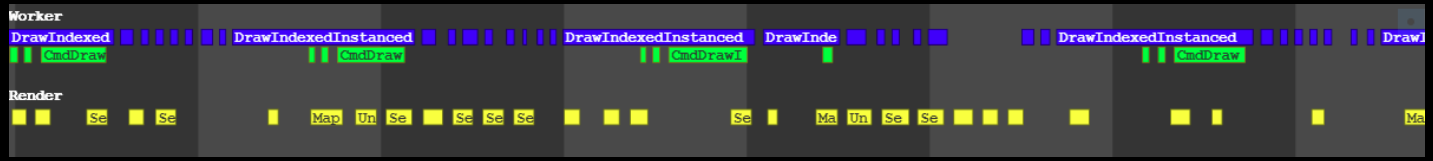
- Current ports have been D3D11-style engines - mostly single-threaded API usage
- Our Vulkan layer has to do a bunch of work every draw/dispatch
  - Look up/create descriptor sets
  - Look up pipeline
  - Resource usage tracking (for barriers)



- Would often end up bottlenecked on the rendering thread in intensive scenes

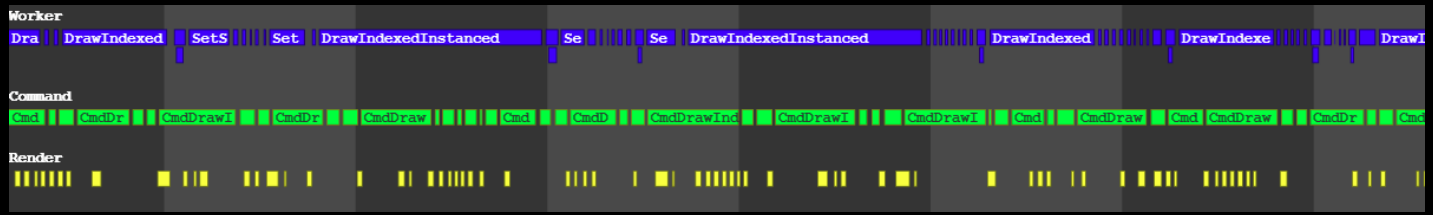
Threading - Rendering

- Solution: offload work done in the Vulkan layer to other thread(s)
- Calls into the Vulkan layer in the game rendering thread only write into a command queue consumed by a worker thread, which does all the heavy lifting for each draw
  - Game rendering logic and Vulkan layer work now execute in parallel



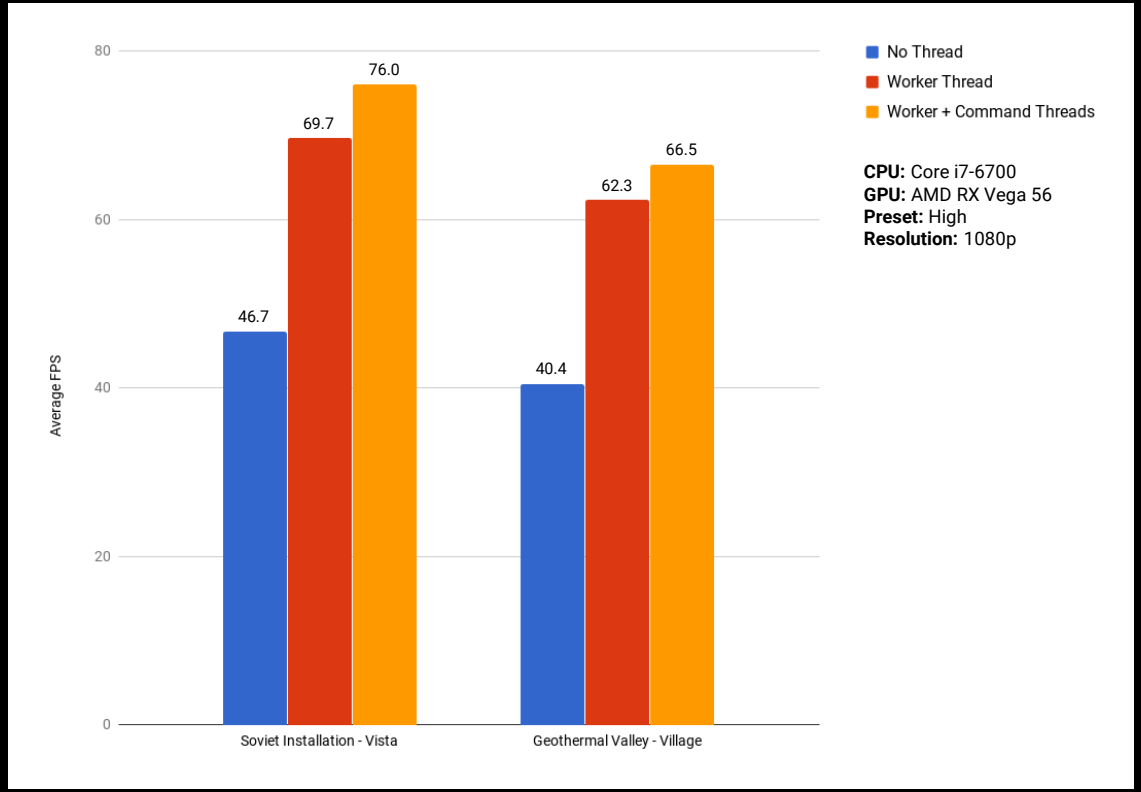
Threading - Rendering

- Can also optionally offload all vkCmd* (plus a few other) calls from that thread to another
  - Quite a bit of CPU time on the worker thread was being spent in the driver
  - Driver work now gets executed in parallel with our work



- Enabled in RotTR for machines with 6 or more hardware threads
  - Up to 10% performance improvement in some CPU limited tests
  - With fewer HW threads, hurts performance slightly due to competing for CPU time with other game threads

Threading - Rendering



Summary

- Vulkan has been a fairly good experience for us so far
  - Desktop drivers are pretty solid
  - On Linux, have several open-source drivers - a huge help both in debugging and understanding how the driver behaves
  - Tools are continually improving
- Our Vulkan support is getting better with every release
- Expect to be targeting Vulkan for Linux releases going forward
- Planning to release our first Android title (GRID Autosport) later this year


FERAL