D3D12 and Vulkan: Lessons learned

Dr. Matthäus G. Chajdas
Developer Technology Engineer, AMD
Overview

The age of D3D12 & Vulkan has begun!
Caveat emptor

- D3D11 drivers are really well optimized
  - Use your knowledge to outsmart & outperform the D3D11 driver
  - D3D12 was not invented to write a legacy API driver on top
- Other issues
Your engine

D3D12 booster

Vulkan booster
Stage 0

Game

“The renderer”

IRenderSystemD3D9-LookAlike

OpenGL
D3D11
D3D9, OpenGL ES, etc.
Stage 0.5

Game

“The renderer”

IRenderSystemD3D11-LookAlike

OpenGL

D3D11

D3D9, OpenGL ES, etc.
No booster
Stage 1

Game

“The renderer”

IRenderSystemD3D11-LookAlike

D3D11

D3D12 & Vulkan
Stage 2

Game

“The renderer”

IRenderSystemD3D12-LookAlike

D3D11

D3D12 & Vulkan

Not high level enough

Not low level enough
Not enough boost
Stage 3

Game

“The renderer”

Low-level rendering layer

OpenGL
D3D11

Low-level rendering layer

D3D12
Vulkan
Weeeeh!
State of the nation

- Engines are transitioning to support Vulkan and D3D12
  - D3D11 support still required
  - Most are midway between Stage 1 and 2
- Lots of thought needed to get the best out of all APIs
  - Multi-queue support requires additional work
  - Needs to scale down to D3D11
- Targeting D3D12/Vulkan and running on D3D11 is the recommended way
Design for the future

- I’ll point out common design issues
- Get your engine ready
- Turn your knowledge into better performance
Resource Barriers
Barrier control

- Barriers are a new concept in D3D12/Vulkan
- Sad truth: **Everyone** gets them wrong
- Two failure cases:
  - Too many or too broad: **Bad performance**
  - Missing barriers: **Corruptions**
- D3D11 driver does this under the hood – and quite well
What’s a barrier, anyway?

**Render target to texture**

- Probably a decompression is needed (& cache flush)
- What will happen changes between vendors and GPU generations – can be a no-op, can be a wait for idle, can be a full cache flush
What’s a barrier, anyway?

**UAV** to **resource**
- If done badly, it will cost – flush or wait for idle
- If done correctly, those transitions can be free
Missing barriers

- Format problems – GPU/driver specific corruption
- Synchronization problems – time-dependent corruption
Subresources
Subresources

- Need to be tracked individually
  - Downsampling
  - Shadow map atlas

- If you transition all subresources, use `D3D12RESOURCE_BARRIER_ALL_SUBRESOURCES` instead of going one-by-one
Placed resources & initial states

- Render targets created as placed resources etc. **must** be cleared before use
- Go into **clear state directly**, don’t start with some random state and transition
Unnecessary transitions

- Transitioning to wrong type
  - Not common but still occasionally happens
  - Make sure to check with validation layer

- Read-read transitions
  - Moving between two read states, i.e. from index buffer to shader resource
  - Moving to union of all future states requires only one barrier

![Diagram showing unnecessary transitions]

- [ ]
- [x]
Costly transitions

- COMMON is for copies/present, not a general “catch all” state
- Usually you want shader access
  - In D3D12: PS_RESOURCE | NON_PS_RESOURCE
  - In Vulkan: VK_ACCESS_SHADER_READ_BIT
Barrier control – Worst case #1

- Worst-case barrier system – too many barriers
  - Material system going wrong
  - For maximum damage, do it per stage

```
Shader stage #1
| Buffer 0 |
| Texture 0 |
| Texture 1 |

Shader stage #2
| Texture 0 |
| Texture 1 |
| Texture 2 |
```
Barrier control – Worst case #1

- “Late binding”, or fixing up resources per draw
- ```
```for (auto& stage : stages) {
  for (auto& resource : resources) {
    if (resource.state & STATE_READ == 0) {
      ResourceBarrier (1, &resource.Barrier (STATE_READ));
    }
  }
```
- Let’s take a look what happens here!
Barrier control – Worst case #1

- Ideal flow

- Per material/stage anti-pattern
  - One barrier per stage per resource
  - Barriers scattered all over the command list

- In the worst case, multiple wait-for-idle back-to-back
Barrier control – Worst case #2

- “Base state” or redundant transitioning
- Transition to target state followed by restore
Funny barriers

- ResourceBarrier (0, nullptr)
  - Nothing changed, thank you!
  - Indicates your state tracking is doing the wrong thing
- Previous state equal to next state
  - Happens more than you believe – just say no
- Always remember – driver assumes you’re doing the optimal thing, doesn’t go through any heuristic itself!
Get ready for the future

- You should **not** have to track all resource state
- 99% of your resources are immutable – read-only. Trust me 😊
- Find “transitions” points – when do passes end?
  - Batch barriers here
  - Only transition what you need

Design first!

<table>
<thead>
<tr>
<th>G-Buffer</th>
<th>Shadow maps</th>
<th>Shading</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Batch transitions   Batch transitions   Batch transitions
Barrier debugging tips

- Have a write/read bit
- Log all transitions
  - Grep & spreadsheets are your friends
  - Check for # transitions, transition type, etc.
- Number of transitions should be in the order of number of writable resources
  - Again, log and grep are your friends
  - If it’s over 9000, something is fishy!
Barrier debugging tips

- Have a barrier-everything mode
  - Same as the “worst-case” mode described previously
  - For debugging only

- Ensure your resources are in a known state at least once per frame
  - For example, at frame end/start
  - Transition everything into a known state – that resolves problems like TAA or shadow atlas breakage
Going forward

- Even better, eventually

- Give driver time to handle the transition
- “Split barrier” in D3D12
- vkCmdSetEvent + vkCmdWaitEvents
Summary: Barriers

- Make sure to transition all the resources that need it (but not more)
- Go into the most specific state you can
- Remember you can combine various states
Launch control
Launch control

- How to feed the GPU
  - Submitting command lists, first and foremost
  - Per-frame resource updates & tracking second
CPU threading
CPU threading

- Don’t limit parallelism by assigning cores manually
- Use a task/job system
  - Uses all cores automatically
  - Requires extra care for efficient work submission and resource synchronization
What happened?

- Thread pool gone wild 😊
  - CPU tasks submitted work at the end
  - Task boundary became CPU/GPU sync point
- Take control over the command lists after the tasks have finished
What happened?

- Each fence is basically a wait-for-*idle* on the GPU (more or less)
- Better:
  - Protect *per-frame* resources
  - Unlikely you can start working on a command list “mid-frame” anyway
  - Protect many resources with a single fence
- Make sure your job system can do this
- Batch up submissions as much as possible
- Submit early to **keep the GPU busy at all times**
Ideal submission

CL = Command list
Command allocators

- Command allocators are defined to be “grow only”
  - Record 100 draw calls on fresh allocator will allocate memory
  - Resetting and recording the same draw calls again will **not** allocate memory again
  - Try to reuse command allocators for similar workloads
- Recycling allocators will grow them to the worst-case size
- In total, number of allocators should be roughly
  \# threads \times \# frames buffered \times \# GPUs
  - We’ve seen 20,000 allocators being allocated – lots of memory waste
- Make sure to reuse allocators/command lists and don’t recreate per frame
Designing for Multithreading

Know your workload and schedule appropriately

G-Buffer

SSAO

Post

 Shadow map

Shade

Design first!
Also: Renderpasses

- Build a high-level graph of your frame
- Tell the renderer about it via Vulkan’s render-passes and subpasses
- Allows the driver to pick an optimal schedule
Also: Renderpasses

- Allows you to express “don’t care” nicely
- Much more about this can be found in the “Vulkan Fast Paths” talk
Debugging hints

- Have an option to submit all command lists in one submission
  - Helps with timing issues
  - If not possible, you have in-frame GPU/CPU synchronization 😞

- Have an option to wait for any command list
  - Helps with upload/resource synchronization
  - Some resource gets corrupted? Flush the GPU before updating it
Summary: Submission

- Track resources at a per-frame granularity
- Know your frame structure
- Threading is essential to get good CPU usage
Async compute

Graphics queue

Async compute

Multi-queue
Multi-Queue

- D3D12 and Vulkan expose multiple queue types: Copy, graphics, compute
  - On Vulkan, check the queue capabilities and how many are present
  - On D3D12, one of every kind is guaranteed to be available – but no scheduling guarantees are given
- Compute queue is getting a lot of good use
- Copy queue is not used much – could use more love
Graphics and Compute

- We see great results from async compute so far
- Run compute load while graphics queue is idle
- We typically see one compute command list running in parallel with one fence for sync
  - That’s fine
  - The more compute the better 😊
Async compute

- Pit of success

<table>
<thead>
<tr>
<th>G-Buffer + Z-Buffer</th>
<th>Shadow maps</th>
<th>Shading</th>
<th>Post-Processing</th>
</tr>
</thead>
</table>

SSAO, light tile classification

Different bottlenecks – maximized GPU usage with async
Async compute

- Pit of no success

<table>
<thead>
<tr>
<th>G-Buffer + Z-Buffer</th>
<th>Shadow maps</th>
<th>Shading</th>
<th>Post-Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSAO, light tile classification</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resource competition – can be worse than running sequentially
Async compute

- Pit of even more success

**Design first!**

- G-Buffer + Z-Buffer
- Shadow maps
- Shading

- Post-Processing
- SSAO, light tile classification

Actual frame end – frames overlap
Copy to the rescue?

- Copy queue is low-latency, low-speed, but it’s separate hardware
  - Copy queue is optimized for transfer over PCIe, not for GPU local copies
  - For PCIe, it is the **fastest way** to transfer data
  - Avoid waiting on copy queue from graphics/compute
  - Ideal use of copy queue is streaming data over a few frames

- Haven't seen much use so far
  - Talk to us why?
  - For copying between adapters, copy queue is also best – consider shared swapchain though
Summary: Multi-queue

- Use the compute queue to fill up the GPU
- Use copy queue to saturate PCIe
- Know your frame structure to find the best location to schedule async work
Other issues
Resources

- On average, things work just fine
  - Uploads rarely a problem, but remember to look at the copy queue
  - On-GPU management mostly ok
  - Packing sometimes not as tight as it could be, check alignment!
- For “high-frequency” resources like frame buffers, prefer CreateCommittedResource in D3D12
- Lots of issues with residency and budget
  - Time travel back to yesterday and watch Dave Oldcorn’s & Stephan Hodes’ talk "Right on Queue - Advanced DirectX12 programming" [If time travel is not invented until the talk replace with presentation URL]
  - It’s an ugly topic – too much to cover here. Talk to me afterwards!
Debug runtime & Validation layers

- D3D12 and Vulkan have validation layers
- The driver **does not validate** for performance reasons
- We assume your application is **perfect**
- During development, make sure to pass validation warning/error free
  - If your app doesn’t support validation, add support for that now!
  - **Any undefined behavior will bite you**, especially with Vulkan – much wider hardware variety
- Please don’t play spec lawyer yourself – if something is unclear or in doubt, contact IHV partner to clarify
  - Spec and validation layers are constantly evolving
  - Various corner cases haven’t been fully understood yet
Mysteries that need more R&D

- **ExecuteIndirect**
  - Haven’t seen serious problems with this yet
  - Mostly used for draw auto and dispatch indirect – we expect more crazy use down the line
  - See “Optimizing the Graphics Pipeline With Compute” on Friday

- **Bundles**
  - Not enough game experience yet
  - Unclear how to get performance out of it – we’re still gathering data

- **mGPU**
  - Not enough game experience yet but in general seems to be “easy” enough
  - Copies through system memory should go on copy queue
  - Shared swapchain is good – but needs Windows 10 1511
Closing remarks

- Vulkan and D3D12 deliver on their promises
  - Require additional thought
  - Just trying to reimplement D3D11 does not provide a benefit!
  - Engines require re-thinking to take advantage of the explicit APIs going forward

- Many driver issues are now app issues
  - Synchronization (barriers!)
  - Memory management (uploads, residency)
  - This means you have the power to fix most issues!
Who’s awesome? You’re Awesome!

@jasperbekkers
@baldurk
@gwahlidal
@maverikou

\[ L_0(\lambda, \chi) = \int_0^\lambda \int f \, f' \, \cos \theta \, d\theta \]

@martinjifuller
@repi

@dankbaker

Raymund Fülöp

Dean Sekulić
Markus Rogowsky

Thanks to KSP to let me use screenshots! Go Jebediah!