AMD

RADEON
TECHNOLOGIES GROUP

GDC2016: RIGHT ON QUEUE

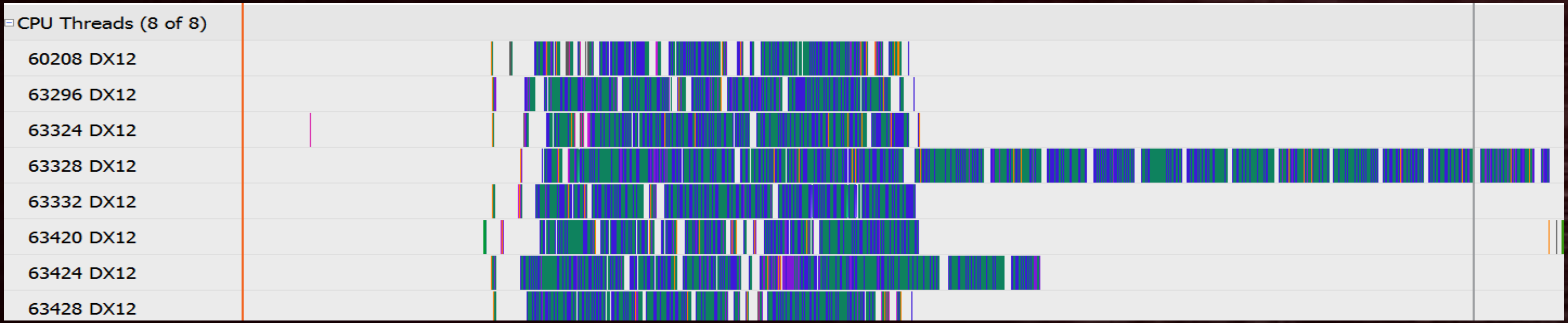
STEPHAN HODES
DAN BAKER
DAVE OLDCORN

AMD

GENERAL ADVICE

80% OF THE PERFORMANCE COMES FROM UNDERSTANDING THE HARDWARE
20% COMES FROM USING THE API EFFICIENTLY

DIRECT3D 12 CPU PERFORMANCE



- Direct3D 12 is designed for low CPU overhead
- Use multithreaded command list recording
- Avoid creation/destruction of heaps at runtime
- Avoid CPU/GPU synchronization points

DIRECT3D 12 **GPU** PERFORMANCE

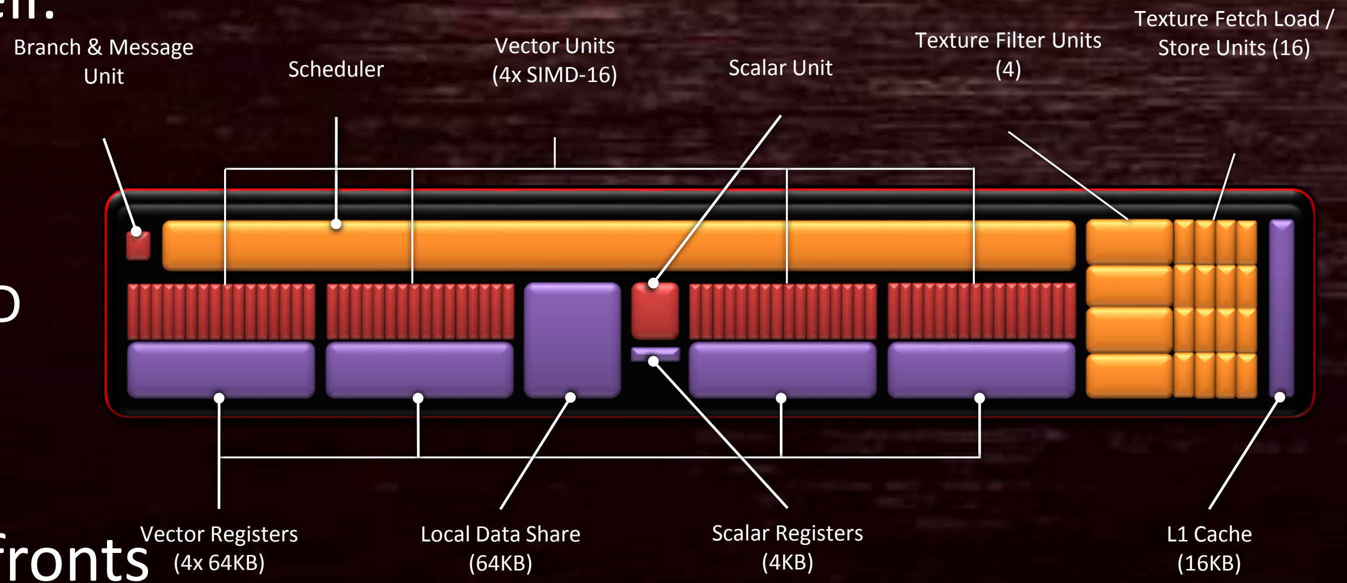
- Direct3D 11 drivers have been optimized over the past 8 years
- Initial DirectX 12 ports tend to be significantly slower than DirectX 11
  - Redesign Engine to take full advantage of DirectX 12
  - Async Queues help to beat DirectX 11 performance

Agenda:

- General Performance advice
- Descriptor sets
- Multiple asynchronous queues
- Understanding Barriers
- Memory management best practices

GCN IN A NUTSHELL

- Hardware hasn’t changed – Direct3D 11 advice still applies
  - –Current AMD hardware in a nutshell:
    - – Several Compute Units (CU)
      - – 64 on FuryX
    - – 4 SIMD per CU
    - – Max. 10 wave fronts in flight per SIMD
    - – 64 threads per wave front



  - –High VGPR count can limit # wave fronts
    - –Use CodeXL: http://gpuopen.com/gaming-product/amd-codexl-analyzercli/

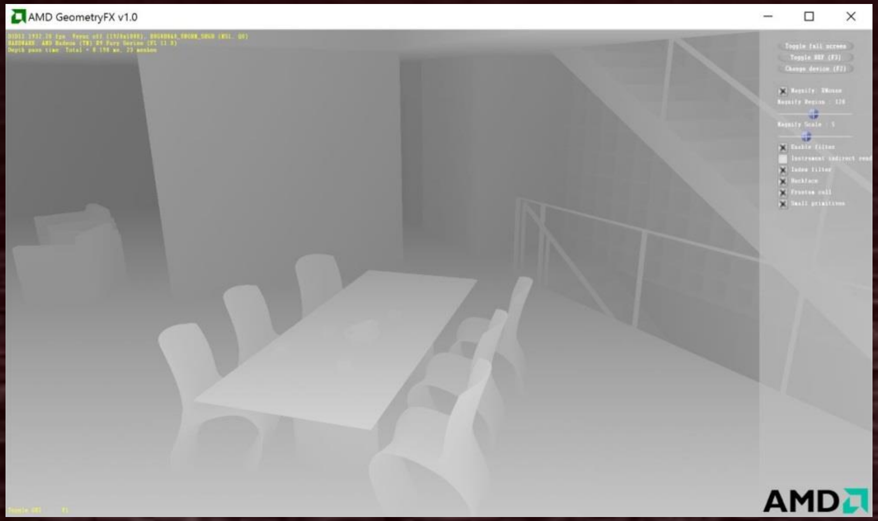
GCN VGPR Count	<=24	28	32	36	40	48	64	84	<= 128	> 128
Max Waves/SIMD	10 ☺	9	8	7	6	5	4	3	2 ☹	1 ☹

General

~~DIRECTX12~~ PERFORMANCE ADVICE

Most performance advice still applies

- Cull: Don’t send unnecessary work to the GPU
  - –Consider compute triangle filtering
  - –Go see: Graham Wihlidal on “Optimizing the Graphics Pipeline With Compute” on Friday
- Sort: Avoid unnecessary overhead
  - –Sort draws by pipeline (and within pipeline by PS used)
  - –Render front to back
- Batch, batch, batch (sorry guys)
  - –small draw calls don’t saturate the GPU



http://gpuopen.com/gaming-product/geometryfx/

GPUOpen

DIRECT3D 12 PERFORMANCE ADVICE - PROFILING

Add In-Engine performance counters

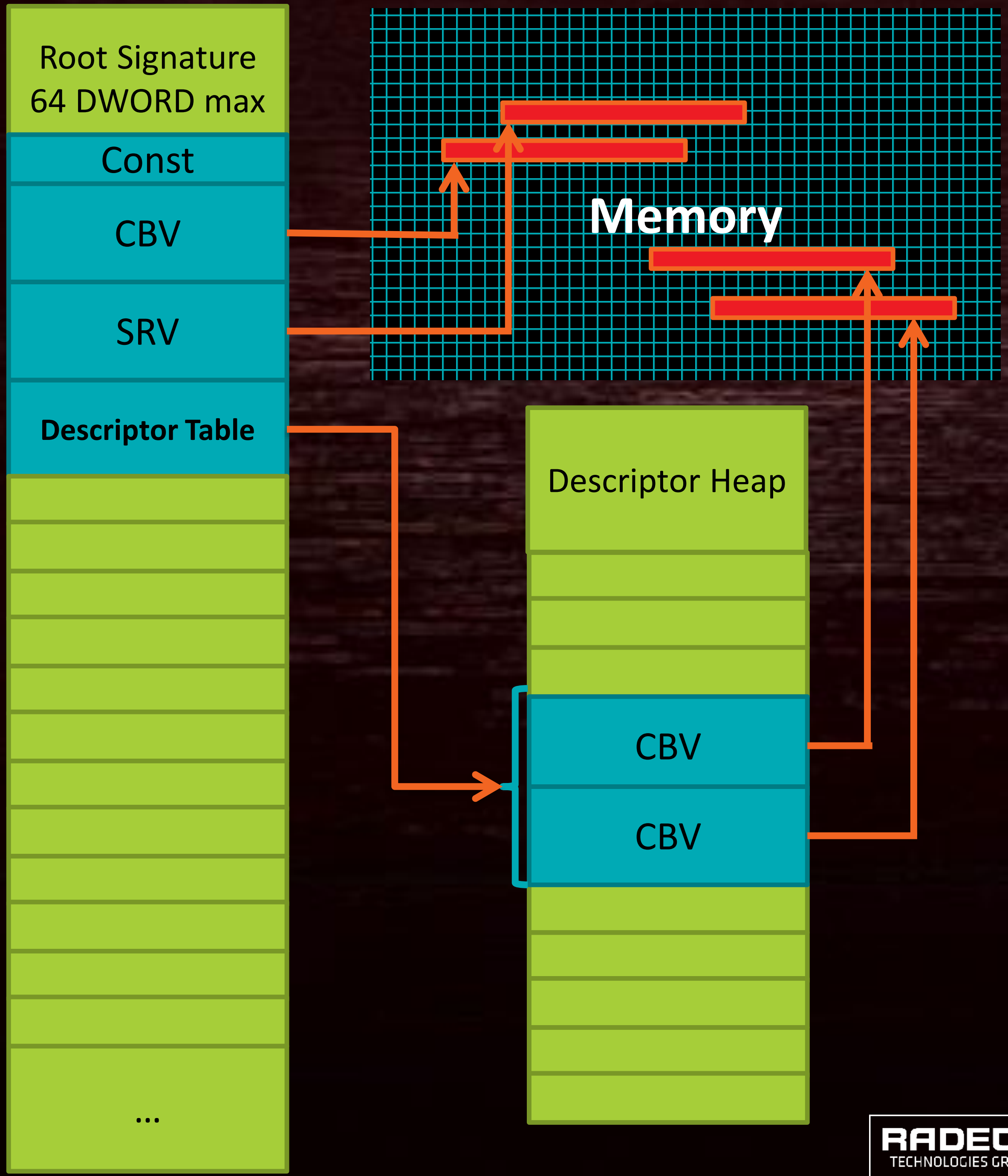
- D3D12_QUERY_TYPE_TIMESTAMP
  - Don‘t stall retrieving the results
- D3D12_QUERY_DATA_PIPELINE_STATISTICS
  - **VSInvocations / IAVertices : Vertex Cache Efficiency**
    http://gpuopen.com/gaming-product/tootle/
  - **CPrimitives / IAPrimitives: Cull rate**
    http://gpuopen.com/gaming-product/geometryfx
  - **PSInvocations / RT resolution: Overdraw**
  - **PSInvocations / CPrimitives: Geometry bound?**
    - Keep in mind that depth only rendering doesn’t use PS
    - Depth test reduces PSInvocations

```
typedef struct
D3D12_QUERY_DATA_PIPELINE_STATISTICS
{
     UINT64 IAVertices;
     UINT64 IAPrimitives;
     UINT64 VSInvocations;
     UINT64 GSInvocations;
     UINT64 GSPrimitives;
     UINT64 CInvocations;
     UINT64 CPrimitives;
     UINT64 PSInvocations;
     UINT64 HSInvocations;
     UINT64 DSInvocations;
     UINT64 CSInvocations;
 } D3D12_QUERY_DATA_PIPELINE_STATISTICS;
```

DESCRIPTOR SETS

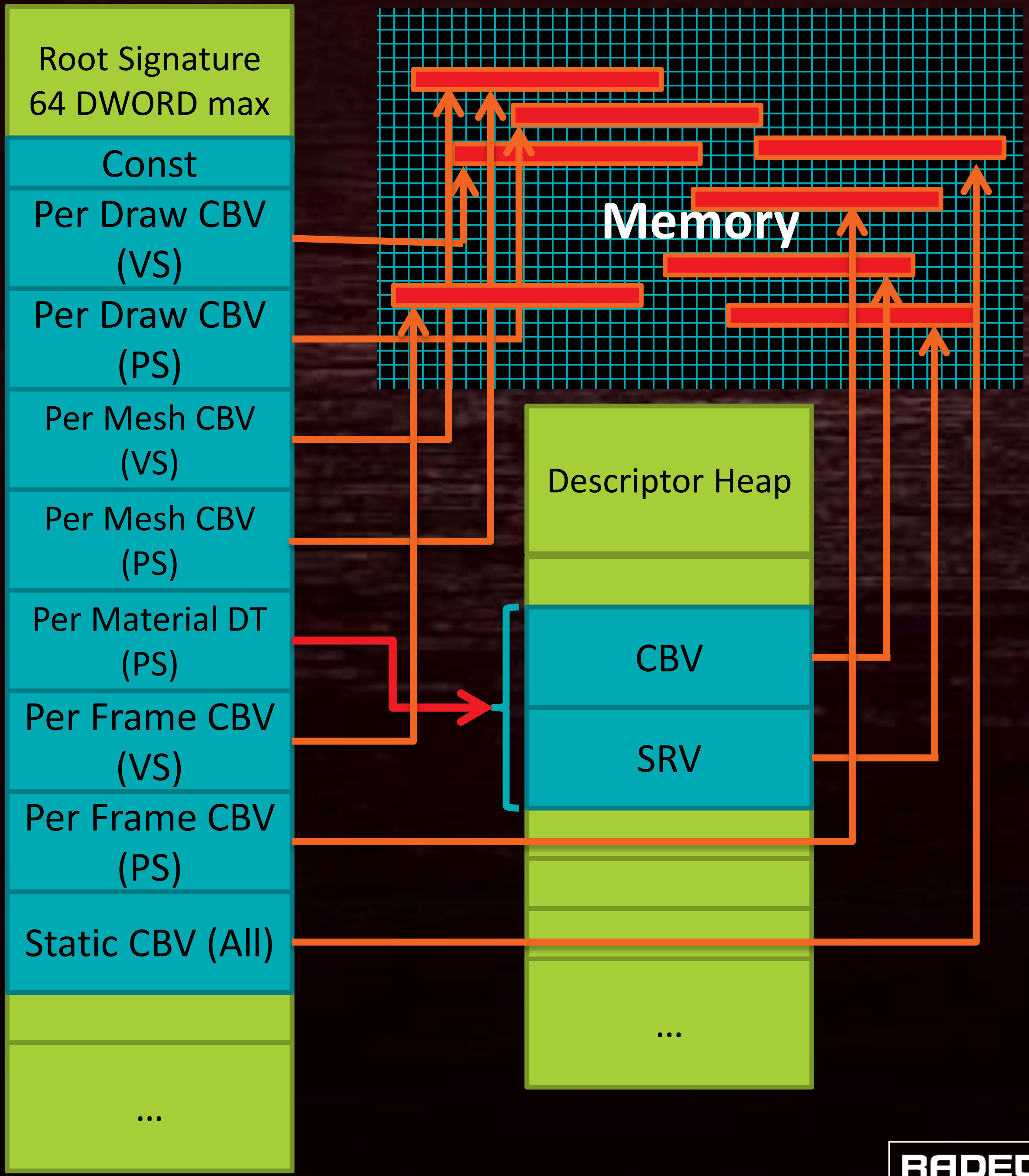
DESCRIPTOR SETS

- Root signature:
  - Maximum size: 64 DWORD
  - Can contain
    - Data (takes up a lot of space!)
    - Descriptors (2 DWORD)
    - Pointer to Descriptor Table
  - Keep a single Descriptor Heap
    - Use as Ringbuffer
  - Use static samplers
    - Maximum of 2032
    - Do not count to the 64 DWORD limit



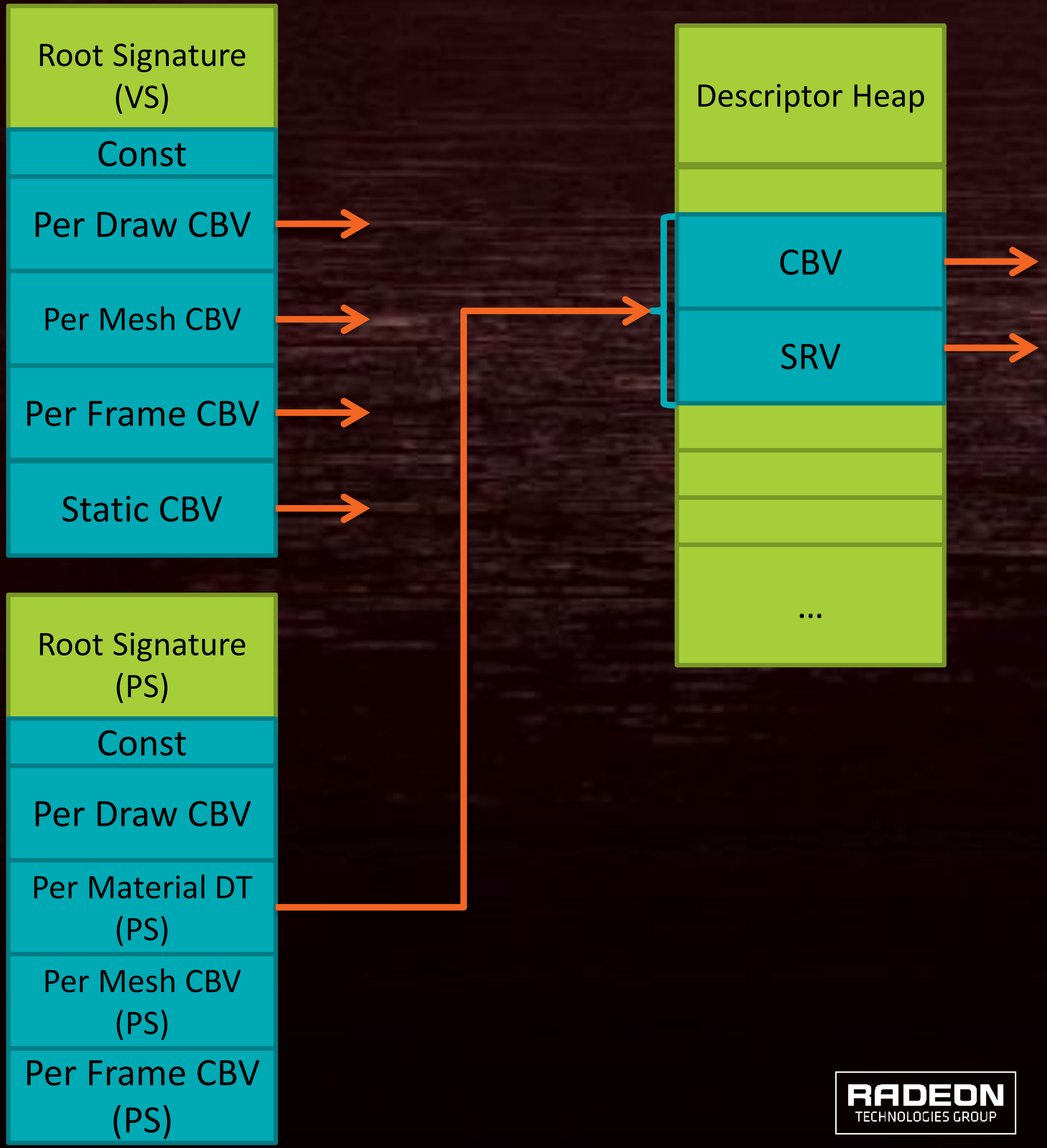
DESCRIPTOR SETS

- Only put small, heavily used constants which change per draw directly into the root signature
- Split Descriptor Tables by frequency of update
  - Put most volatile elements first
- Use D3D12_SHADER_VISIBILITY flag
  - Not a mask
  - Duplicate entries to set exact visibility



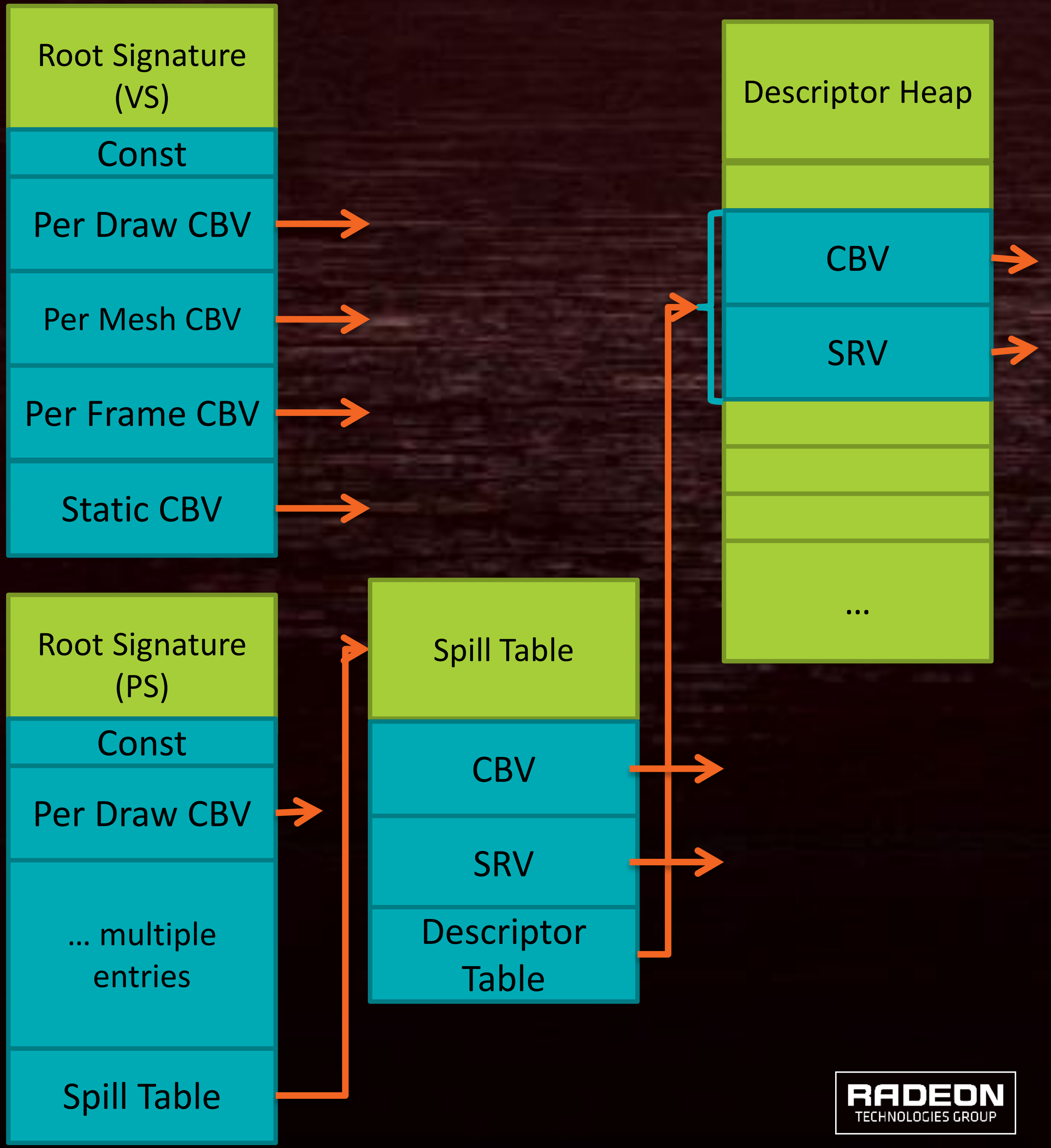
DESCRIPTOR SETS

- Root copied to SGPR on launch
  - –Layout defined at compile time
  - –Only what’s required for each shader stage



DESCRIPTOR SETS

- Root copied to SGPR on launch
  - –Layout defined at compile time
  - –Only what's required for each shader stage
  - –Too many SGPR ->
    Root Signature will spill into local memory
- Most frequently changed entries first
- Avoid spilling of Descriptor Tables!

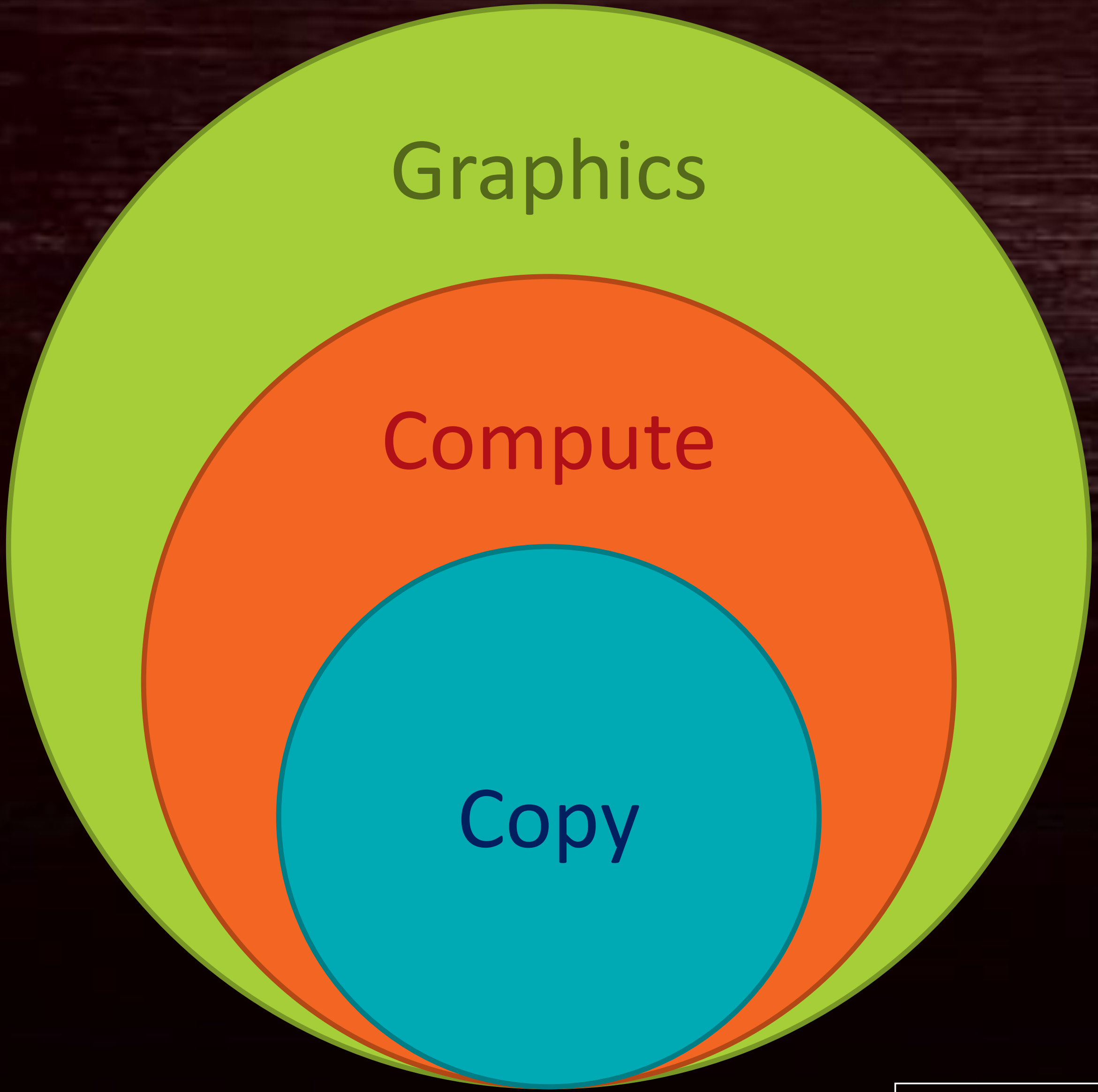


ASYNC QUEUES

D3D12 – ADDITIONAL PERFORMANCE UNLEASHED

QUEUE TYPES

- Copy queue:
  - Used to copy data
  - Optimized for PCIe transfers
  - Does not steal shader resources!
- Compute queue:
  - Use for copying local/local
  - Use for compute tasks that can run async with graphics
- Graphics queue
  - Can do everything
  - Draws are usually the biggest workload



QUEUE TYPES

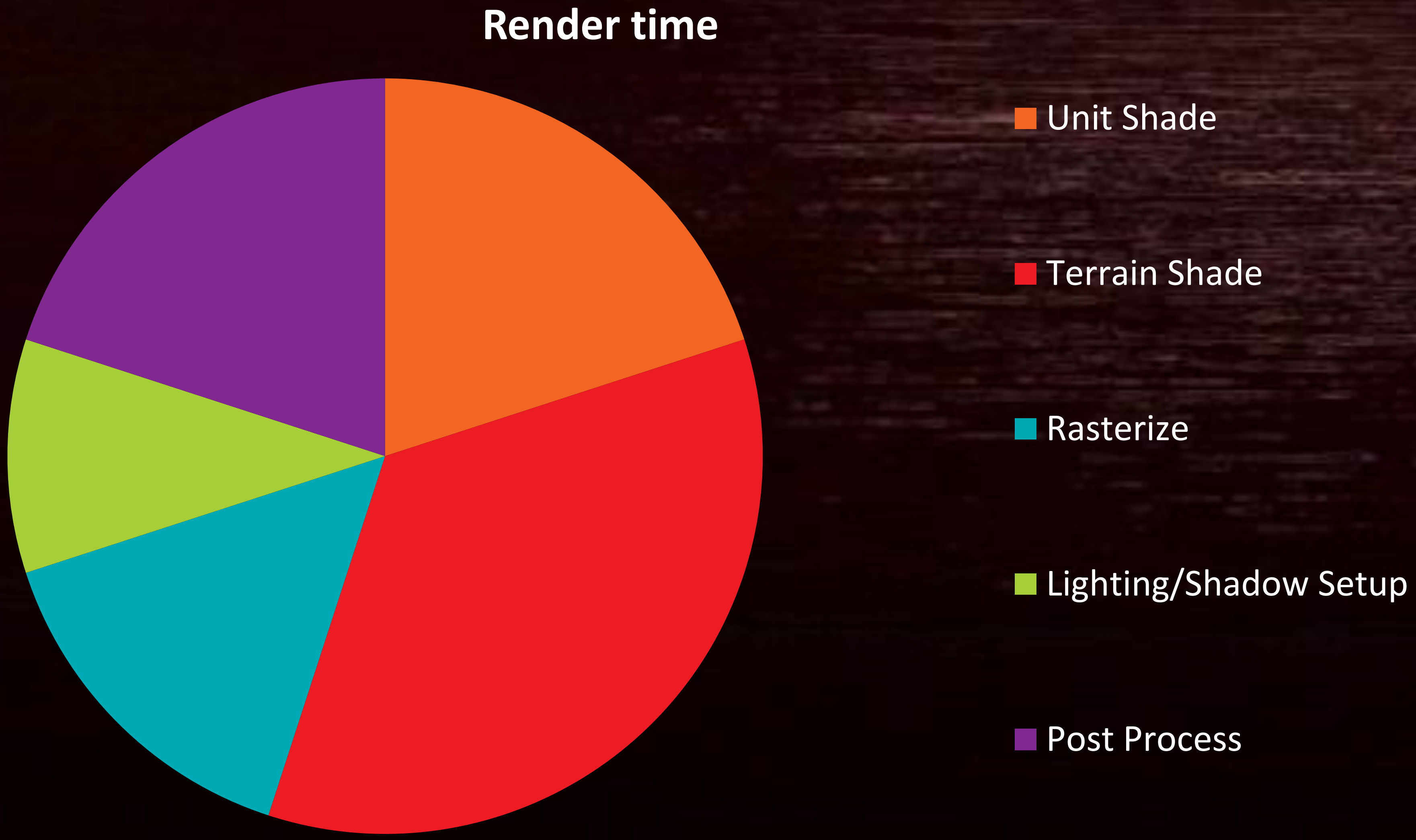
- Async queue usage can gain extra performance “for free”
  - Helps you beat DirectX 11 performance
- Resources are shared
  - Schedule workloads with different bottlenecks together
    - Shadows are usually limited by geometry throughput
    - Compute is usually bound by fetches, rarely ALU limited
      - Use LDS to optimize memory efficiency
  - Async compute will affect performance of the graphics queue
    - Keep this in mind when profiling – keep a synchronous path in your engine

ASYNC QUEUE USAGE

- Implementation advice
  - Build a job based renderer
    - This will help with barriers, too!
  - Manually specify which tasks should run in parallel
- Jobs should not be too small
  - Keep number of fences/frame in single digit range
  - Each signal stalls the frontend and flushes the pipeline

ASYNC COMPUTE IN ASHES

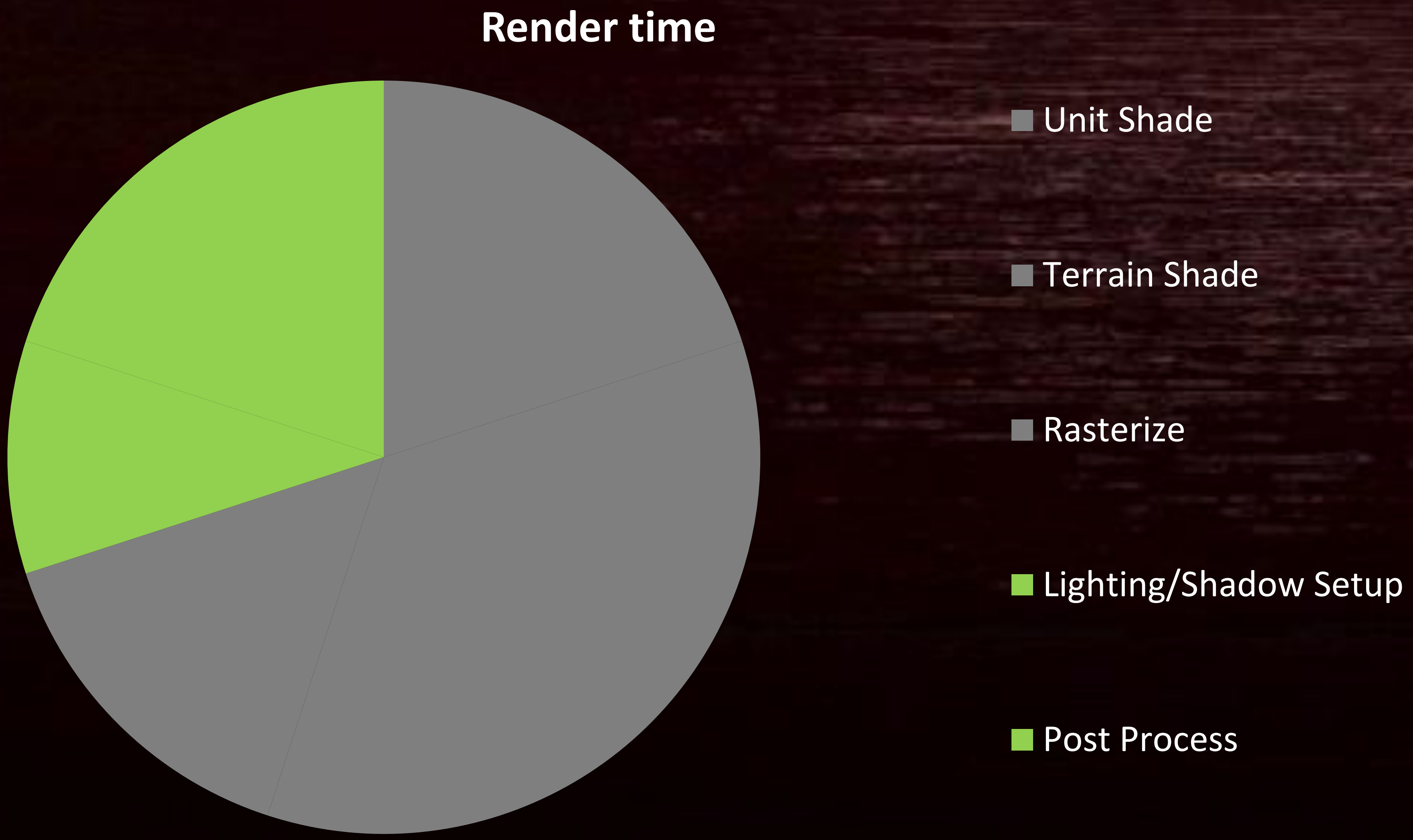
WHERE OUR RENDERING GOES



FRAME OBSERVATIONS

- Lighting and most Shadow work is compute shader
- Post Process is also a compute shader
- What percent of frame is possible to put in a compute queue

WHERE OUR RENDERING GOES



SHADOW MAP

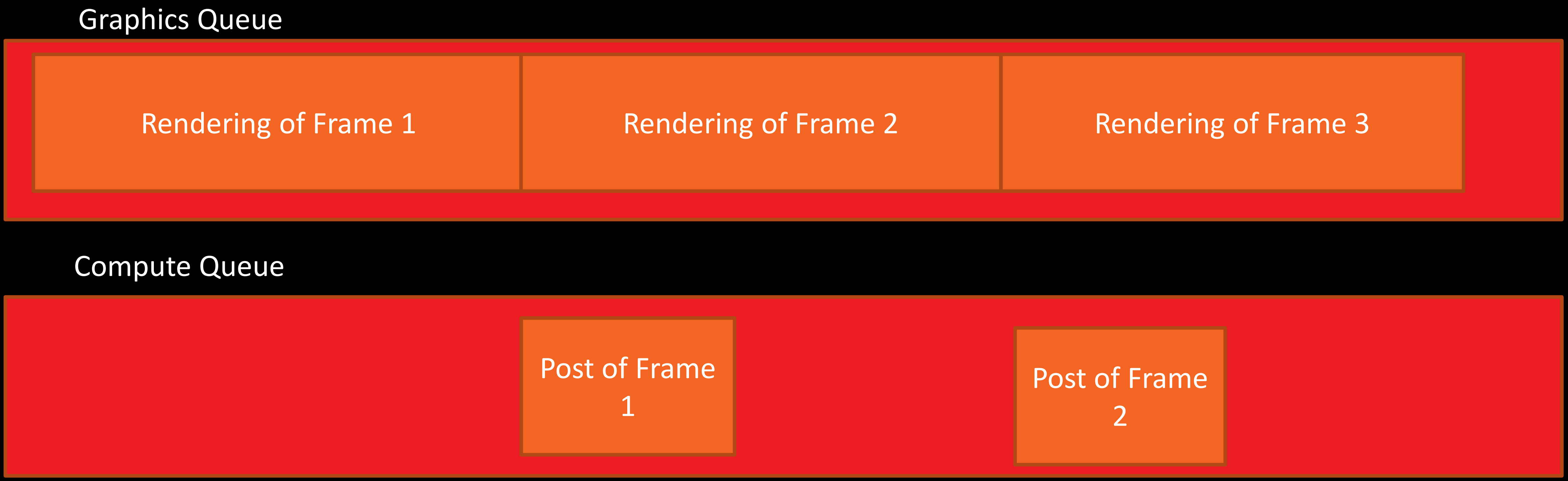
- Terrain projected shadows
- Simple tech
- But wide Gaussian blur to prevent aliasing
- Can take 2ms – but, can be a frame late
- Could blur while frame is rendering

POST PROCESS

- 3 part post
- Simple Gaussian blur (narrow, 5x5)
- Complex glare effect (large, screen sized non symmetric lens effects)
- Color curve – ACES
- Happens and end of frame, nothing to overlap with
  - Or is there?

FRAME OVERLAP

- Overlap the post of one frame with the beginning of the next frame



Graphics Queue

Rendering of Frame 1

Rendering of Frame 2

Rendering of Frame 3

Compute Queue

Post of Frame 1

Post of Frame 2

WITHOUT INTRODUCING TOO MUCH LATENCY

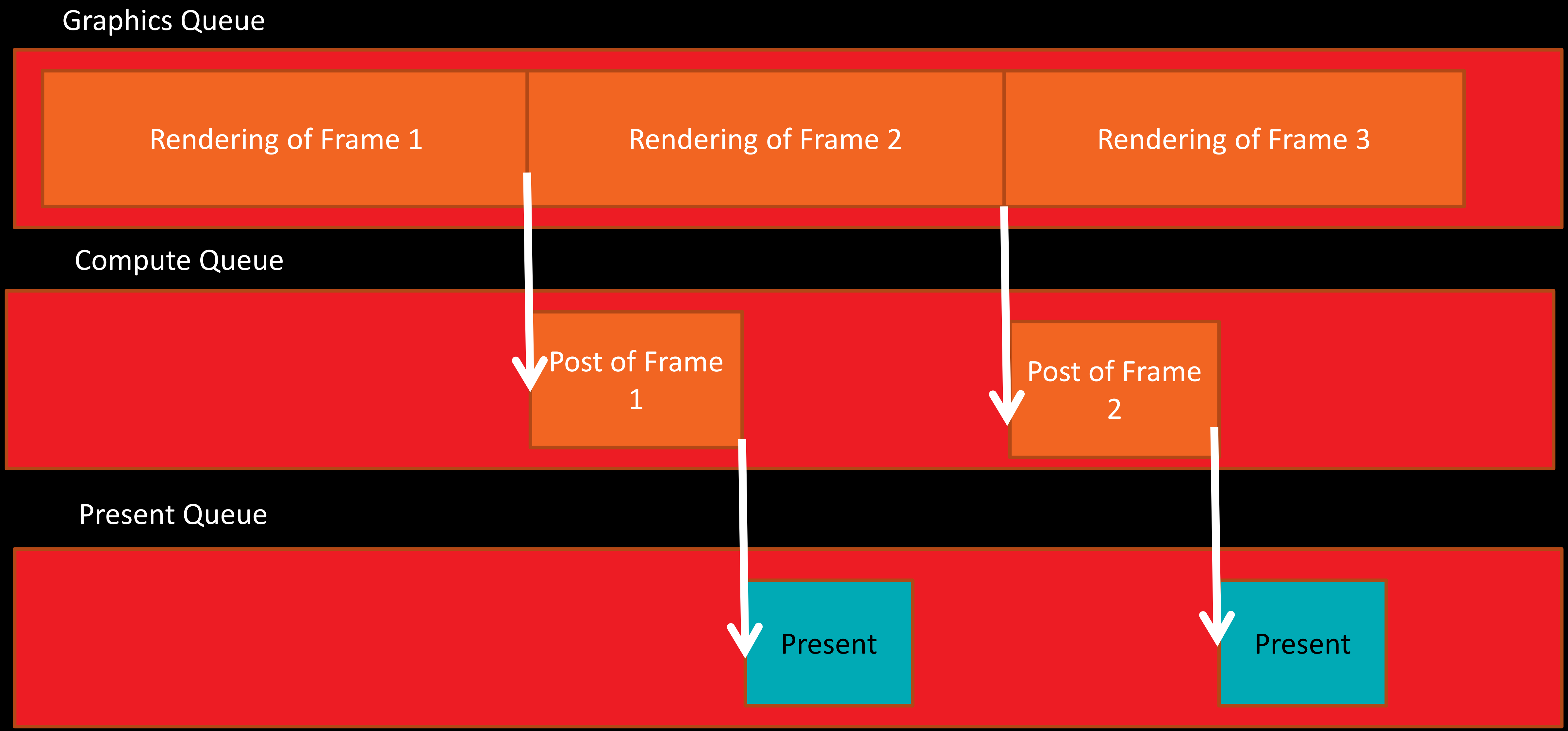
AMD

- Overlapping frames could be complex in engine
- Engine queues up entire frame at time, no concept of previous frame during rendering
- Turns out we can have Direct3D 12 overlap frames for us

BASIC IDEA

- Set number of queueable frames to 3 over 2
- Create a separate present queue from graphics queue
- At the end of the rendering, instead of issuing present – issue a compute task and signal the post to render
- When post is completed – signals an alternate graphics queue to do the actual present

FRAME OVERLAP

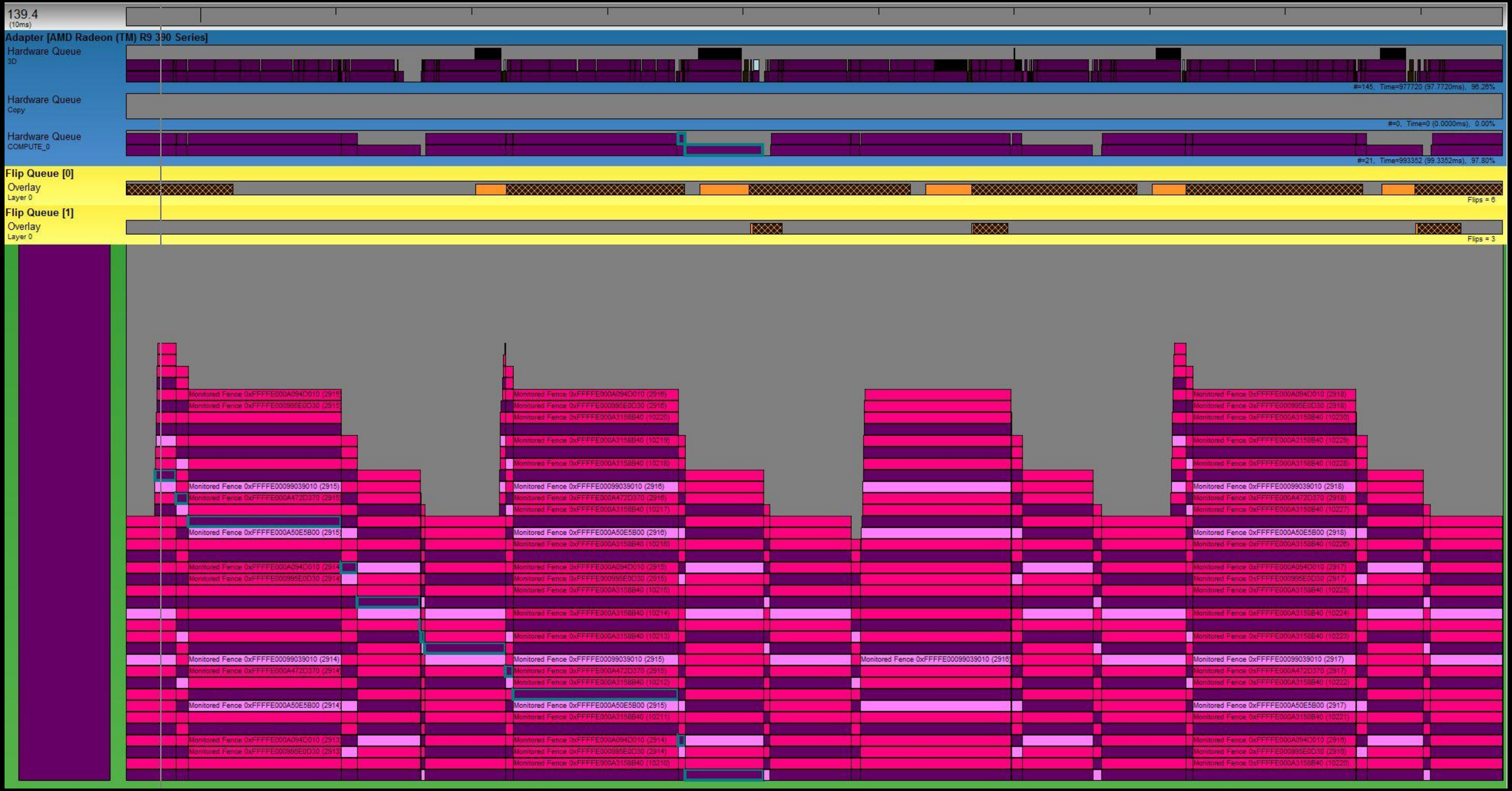


D3D12 SCHEDULER

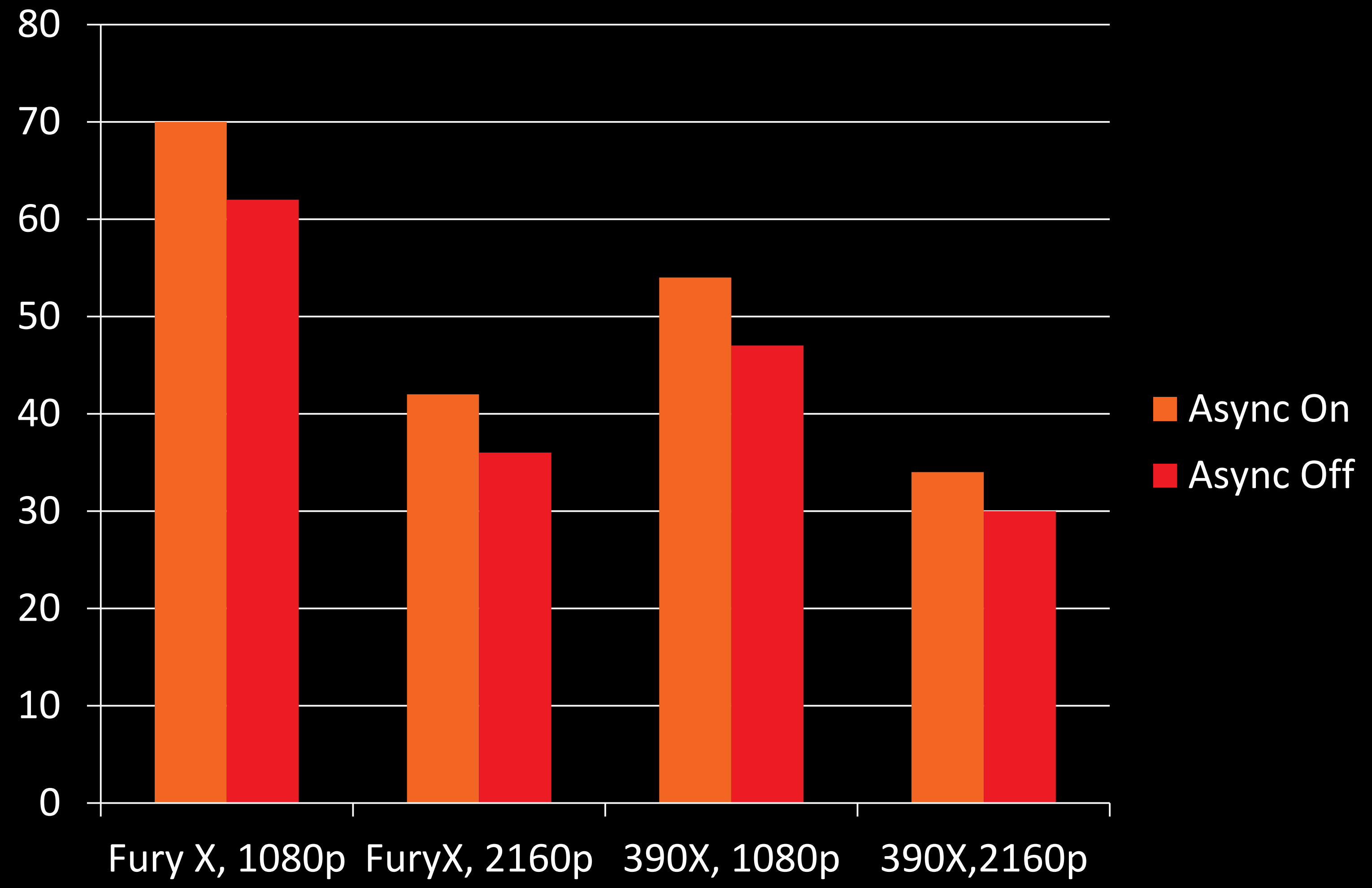
AMD

- Will take care of inserting command stream
- But…
  - No preemption on most cards
- Thus, break apart frame to have multiple submits, trying to keep command buffers in the 1-2ms range
- Windows can then insert present at the boundary
- End up with only about ½ to 1/3 extra latency

WHAT OUR FRAME LOOKS LIKE IN GPUVIEW


139.4
(10ms)
Adapter [AMD Radeon (TM) R9 390 Series]
Hardware Queue
3D
#=145, Time=977720 (97.7720ms), 96.26%
Hardware Queue
Copy
#=0, Time=0 (0.0000ms), 0.00%
Hardware Queue
COMPUTE_0
#=21, Time=993352 (99.3352ms), 97.80%
Flip Queue [0]
Overlay
Layer 0
Flips = 6
Flip Queue [1]
Overlay
Layer 0
Flips = 3
Monitored Fence 0xFFFFE000A094D010 (2915)
Monitored Fence 0xFFFFE000995E0D30 (2915)
Monitored Fence 0xFFFFE00099039010 (2915)
Monitored Fence 0xFFFFE000A472D370 (2915)
Monitored Fence 0xFFFFE000A50E5B00 (2915)
Monitored Fence 0xFFFFE000A094D010 (2914
Monitored Fence 0xFFFFE000995E0D30 (2914)
Monitored Fence 0xFFFFE00099039010 (2914)
Monitored Fence 0xFFFFE000A472D370 (2914)
Monitored Fence 0xFFFFE000A50E5B00 (2914)
Monitored Fence 0xFFFFE000A094D010 (2913)
Monitored Fence 0xFFFFE000995E0D30 (2913)
Monitored Fence 0xFFFFE000A094D010 (2916)
Monitored Fence 0xFFFFE000995E0D30 (2916)
Monitored Fence 0xFFFFE000A3158B40 (10220)
Monitored Fence 0xFFFFE000A3158B40 (10219)
Monitored Fence 0xFFFFE000A3158B40 (10218)
Monitored Fence 0xFFFFE00099039010 (2916)
Monitored Fence 0xFFFFE000A472D370 (2916)
Monitored Fence 0xFFFFE000A3158B40 (10217)
Monitored Fence 0xFFFFE000A50E5B00 (2916)
Monitored Fence 0xFFFFE000A3158B40 (10216)
Monitored Fence 0xFFFFE000A094D010 (2915)
Monitored Fence 0xFFFFE000995E0D30 (2915)
Monitored Fence 0xFFFFE000A3158B40 (10215)
Monitored Fence 0xFFFFE000A3158B40 (10214)
Monitored Fence 0xFFFFE000A3158B40 (10213)
Monitored Fence 0xFFFFE00099039010 (2915)
Monitored Fence 0xFFFFE000A472D370 (2915)
Monitored Fence 0xFFFFE000A3158B40 (10212)
Monitored Fence 0xFFFFE000A50E5B00 (2915)
Monitored Fence 0xFFFFE000A3158B40 (10211)
Monitored Fence 0xFFFFE000A094D010 (2914)
Monitored Fence 0xFFFFE000995E0D30 (2914)
Monitored Fence 0xFFFFE000A3158B40 (10210)
Monitored Fence 0xFFFFE00099039010 (2916)
Monitored Fence 0xFFFFE000A094D010 (2918)
Monitored Fence 0xFFFFE000995E0D30 (2918)
Monitored Fence 0xFFFFE000A3158B40 (10230)
Monitored Fence 0xFFFFE000A3158B40 (10229)
Monitored Fence 0xFFFFE000A3158B40 (10228)
Monitored Fence 0xFFFFE00099039010 (2918)
Monitored Fence 0xFFFFE000A472D370 (2918)
Monitored Fence 0xFFFFE000A3158B40 (10227)
Monitored Fence 0xFFFFE000A50E5B00 (2918)
Monitored Fence 0xFFFFE000A3158B40 (10226)
Monitored Fence 0xFFFFE000A094D010 (2917)
Monitored Fence 0xFFFFE000995E0D30 (2917)
Monitored Fence 0xFFFFE000A3158B40 (10225)
Monitored Fence 0xFFFFE000A3158B40 (10224)
Monitored Fence 0xFFFFE000A3158B40 (10223)
Monitored Fence 0xFFFFE00099039010 (2917)
Monitored Fence 0xFFFFE000A472D370 (2917)
Monitored Fence 0xFFFFE000A3158B40 (10222)
Monitored Fence 0xFFFFE000A50E5B00 (2917)
Monitored Fence 0xFFFFE000A3158B40 (10221)
Monitored Fence 0xFFFFE000A094D010 (2916)
Monitored Fence 0xFFFFE000995E0D30 (2916)
Monitored Fence 0xFFFFE000A3158B40 (10220)

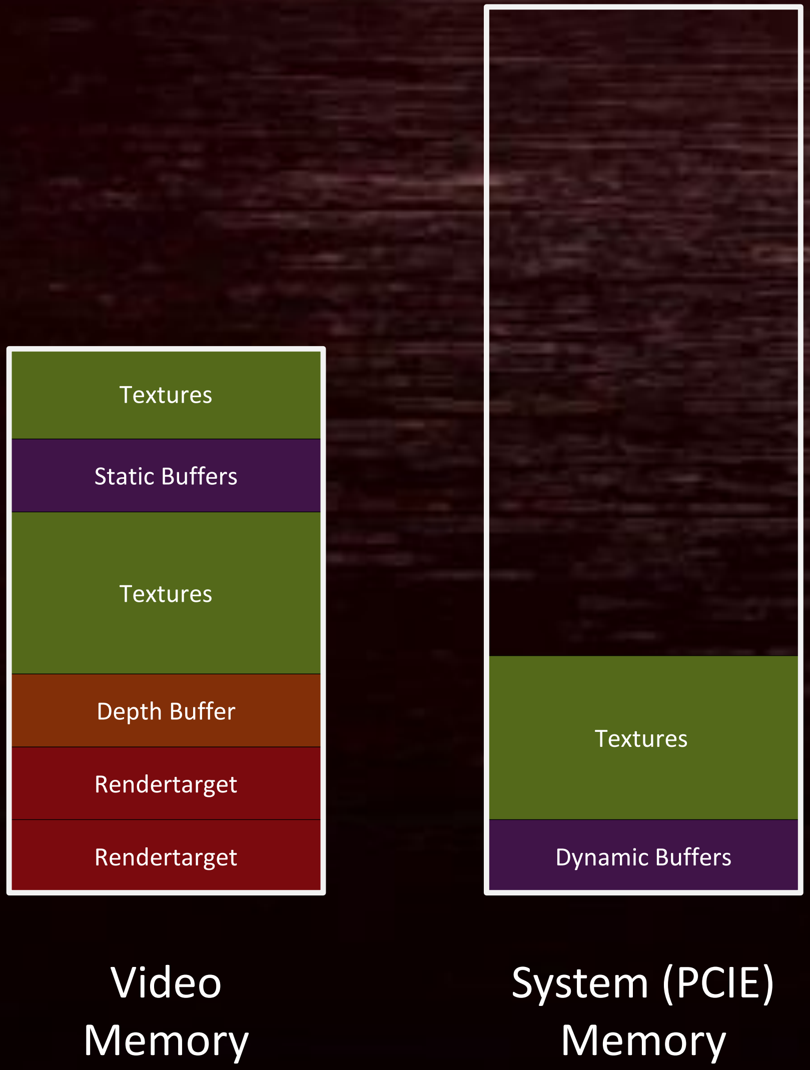
PERFORMANCE INCREASE ~15%



RESOURCE MANAGEMENT

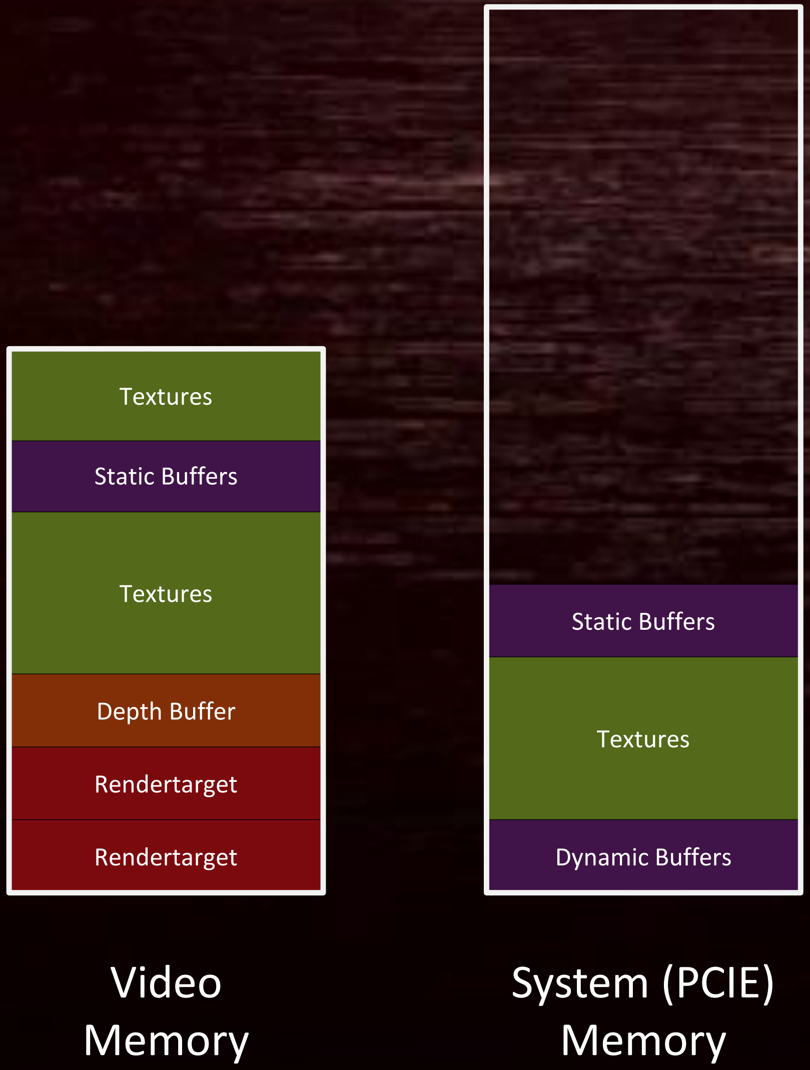
DIRECT3D 11 MEMORY MANAGEMENT

- OS component handles residency
  - (on each command buffer)
- Memory filled over time, mostly straight into video
- Eventually overflows
- Bumped to system memory



DIRECT3D 11 MEMORY MANAGEMENT

- Priority system under the hood
  - RT or DS or UAV unlikely to move
  - i.e. high bandwidth read write surface
- Still a chance of something important moving
- … nobody who noticed it seemed to like it very much!



WHAT WDDM2 DOES:

- Tells **the app** where the limit is
  - The app knows what resources are important better than the OS/driver
- You can see that it’s about to go wrong
  - Intervene!
    - Use lower resolution textures, drop higher mips, change formats to BC1
    - Move less demanding resources to system memory
  - Or don’t.
    - It will still migrate as a backup plan
      - Probably works out OK for small oversubscriptions, 5-10% or so
      - Will probably be a pretty awful user experience if it’s 20% plus
      - Much more likely to see stuttering and inconsistent framerates

RESERVATION

- You can say “I really need this much”

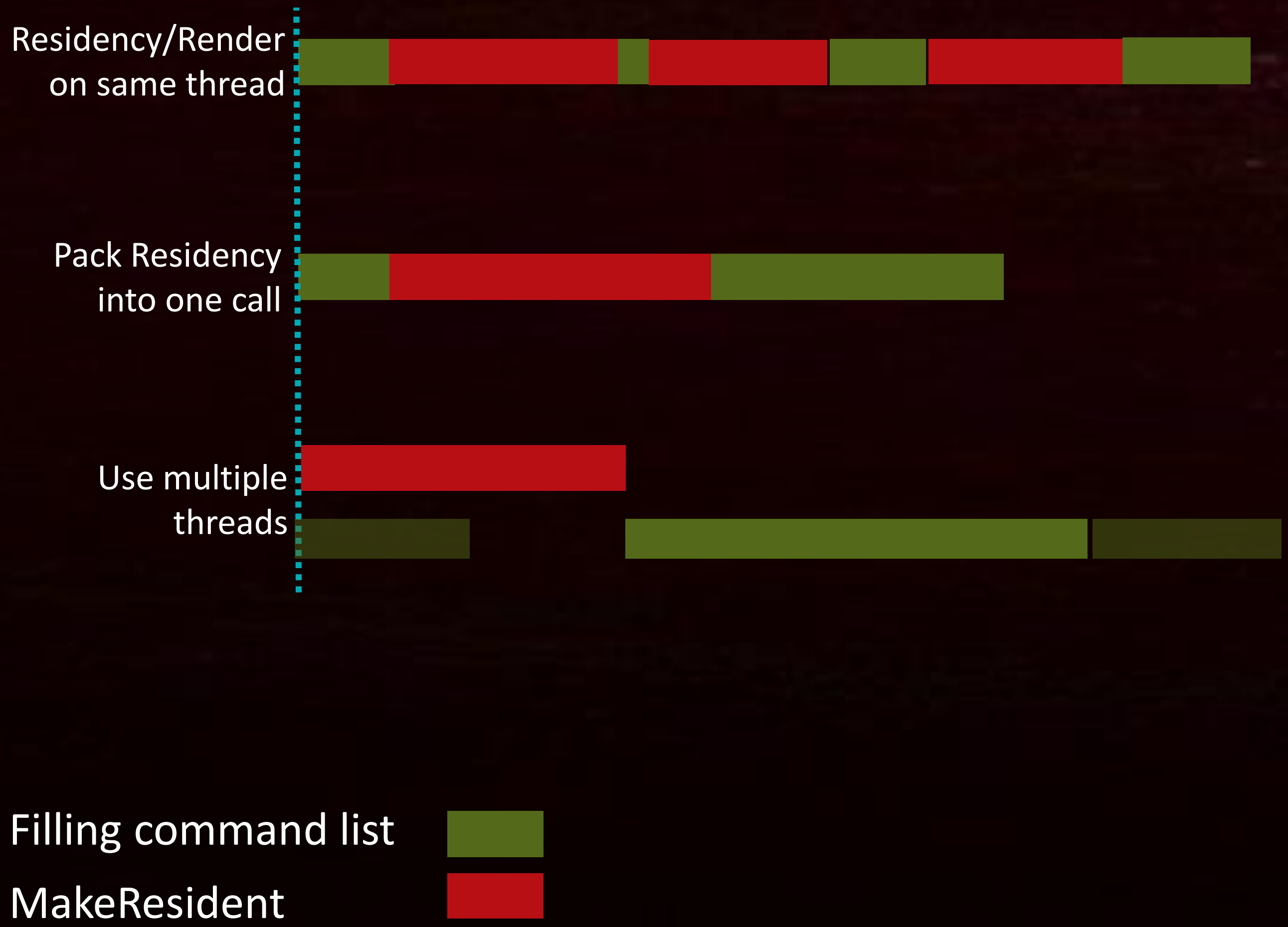
  `IDXGIAdapter3::SetVideoMemoryReservation`

- OS will tell you how much you can reserve in `QueryVideoMemory`:
  - If you’re the foreground app, it starts at about half of VRAM on an idle system
  - If it’s less, it probably means another heavyweight app is already running.
    - Might it be wise to pop up dialog claiming other apps need to be closed?

MINIMUM SPECS AND USER OPTIONS

- Memory exhaustion is a min spec issue
- **You need to know roughly what memory you need**
  - Track this during development
  - Don’t allow design / art to surprise you!
- Set a hard cap on options for 1GB, 2GB, 3GB etc, boards
  - Don’t allow apps to pick crazy settings on low-memory boards
  - You deserve everything you get if you allow 4K on a 1GB board
- Not a total solution because of other apps in the system
  - But combined with the reservation, you should have enough control
    - More than you did in 11, frankly

MAKERESIDENT



Residency/Render on same thread

Pack Residency into one call

Use multiple threads

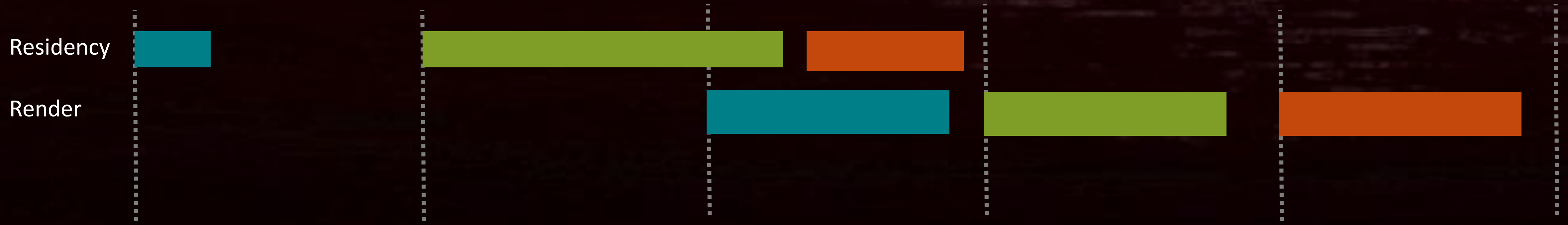
Filling command list 🟩

MakeResident 🟥

- MakeResident is **synchronous**
  - Blocks until allocation is available
- Batch them up
- **Must** move it off render thread
  - Paging operations will interleave with your rendering reasonably gracefully
- Need to do it ahead of use
  - Otherwise you're going to stutter

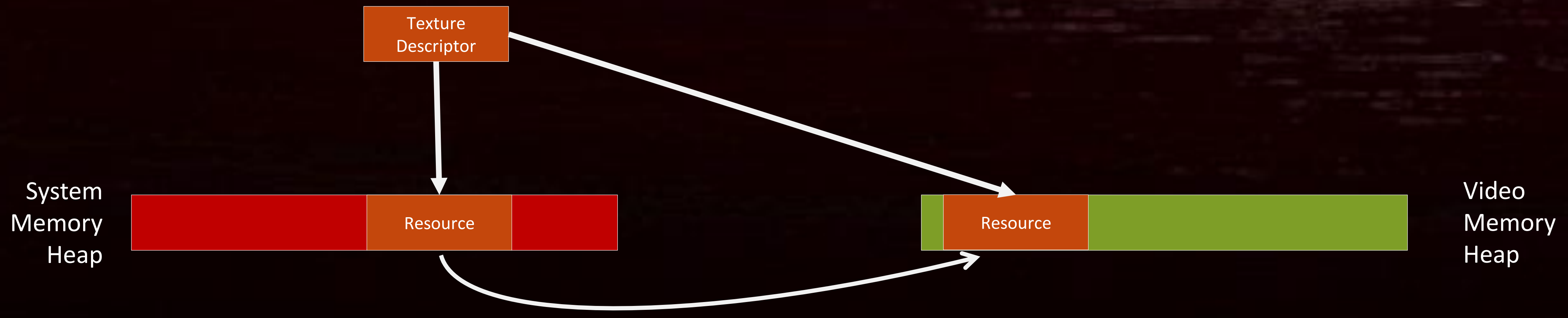
RUN-AHEAD STRATEGIES

- Predict further ahead of time what might be used now and later
- Run a couple of frames ahead of render thread
  - –More buffering == less stuttering
  - –BUT pumps latency into the system



RUN-AHEAD STRATEGIES

- Don’t actually use residency at all!
- Preload resources you might use to system memory
  - –Don’t even have to move them immediately
  - –On use, copy into local then rewrite descriptors or remap pages
  - –Reverse operation and evict local copy when you need to cut memory usage



Copy (using copy queue, PCIE transfer)

RUN-AHEAD STRATEGIES

- Big challenge for VR apps
  - Long latency solutions obviously unworkable
  - Will have to use system memory judiciously and have good look-ahead in the streaming

BARRIERS

HOW TO AVOID SHOOTING YOURSELF IN THE FOOT

(AND OCCASIONALLY IN THE FACE)

BARRIERS

- What is a barrier?

Synchronisation

ensure strict and correct ordering of work

Visibility

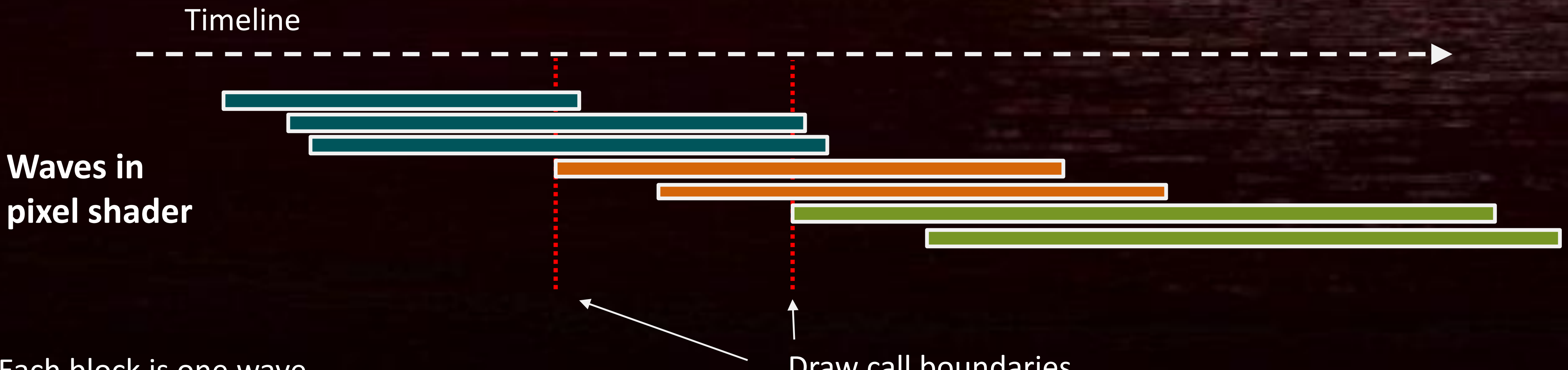
ensure previously written data is visible to target units

Format conversion

ensure data is in a format compatible with the target units

SYNCHRONISATION

- Caused because of depth of GPU pipeline
  - Example: UAV RAW/WAW barrier
  - Avoiding shader waves overlapping in execution



Each block is one wave

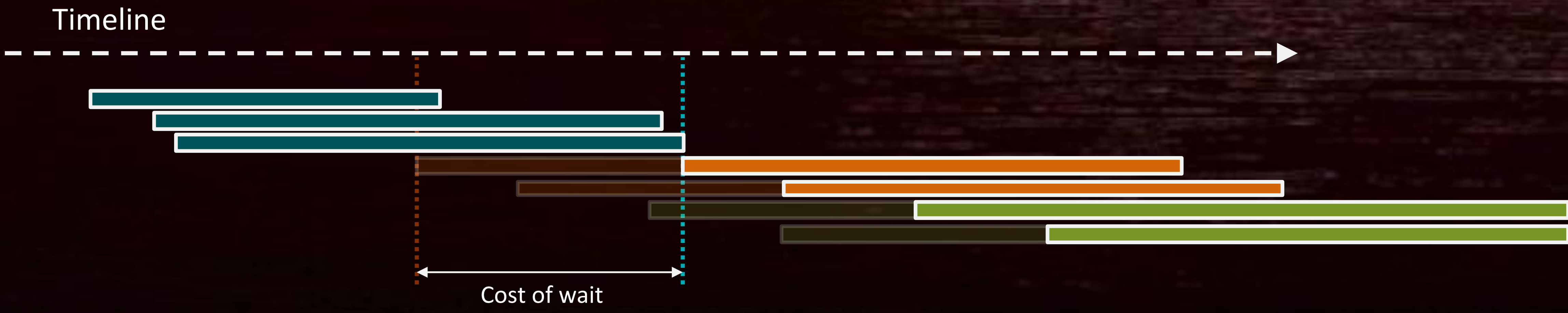
Colours indicate waves belonging to different draw calls

Draw call boundaries

Note overlap of draw calls simultaneously executing

SYNCHRONISATION BARRIER

- Assume draw 3 depends on draw 1
  - –What does a one-piece barrier do?



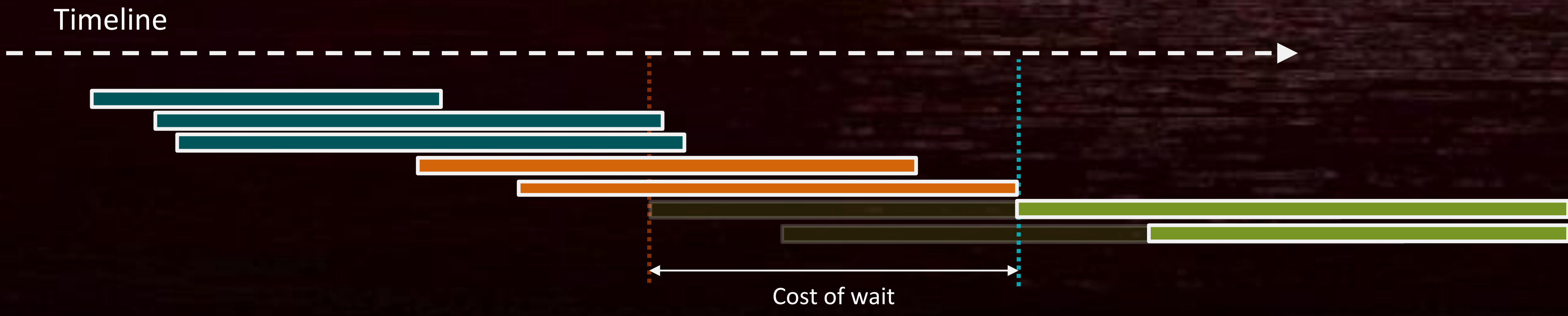
Barrier after draw 1

SYNCHRONISATION BARRIER

- A one-piece UAV barrier is saying:

  "I have just finished with a UAV, make it ready to use again right now."

  – The driver inserts a signal, and then waits on it; guaranteed no overlap of work

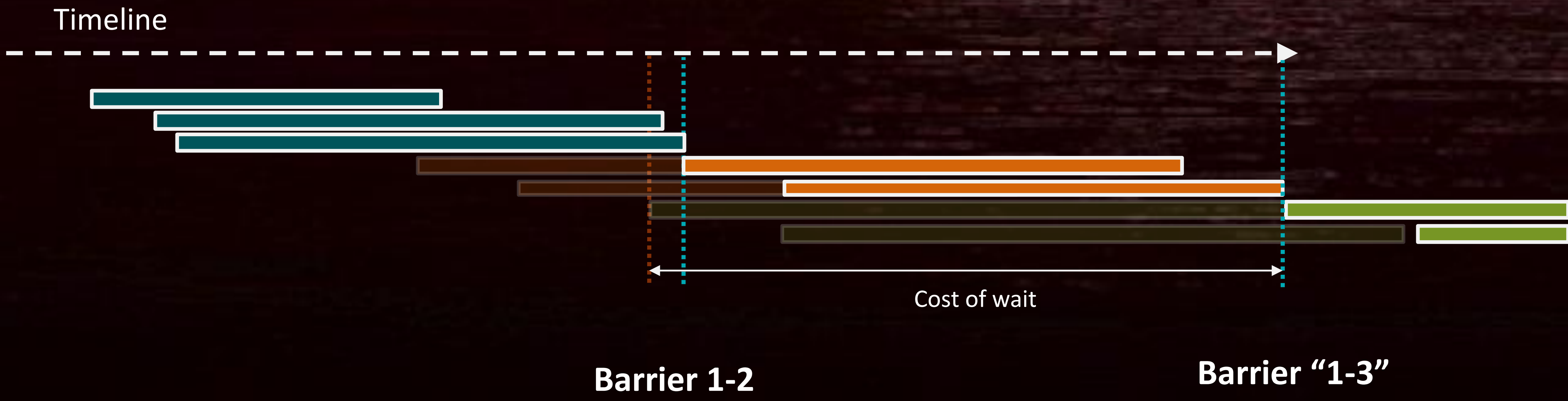


Barrier after draw 2

SYNCHRONISATION SINGLE BARRIER

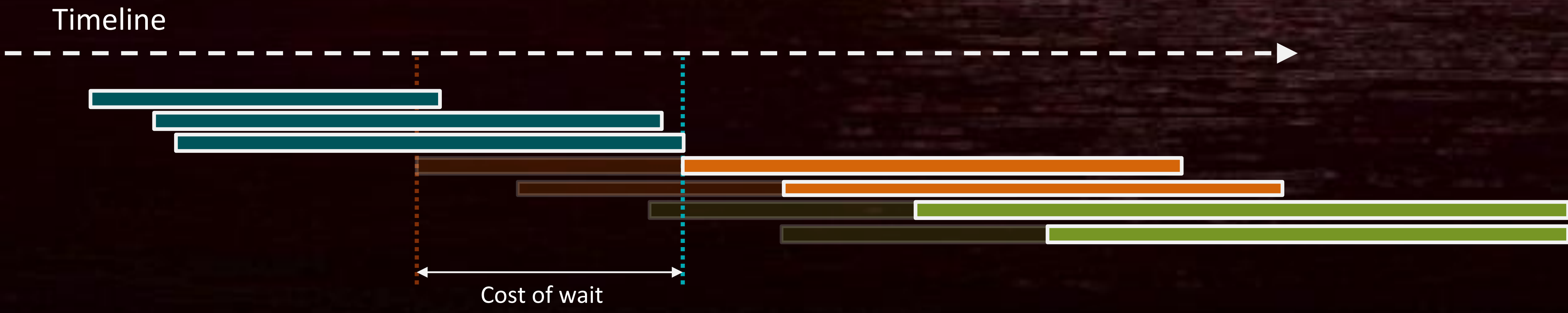
- Assume draw 3 AND draw 2 depend on different resources written in draw 1
  - What do two individual barriers do?

"I have just finished with a UAV, make it ready to use again right now."



SYNCHRONISATION MULTIPLE BARRIER

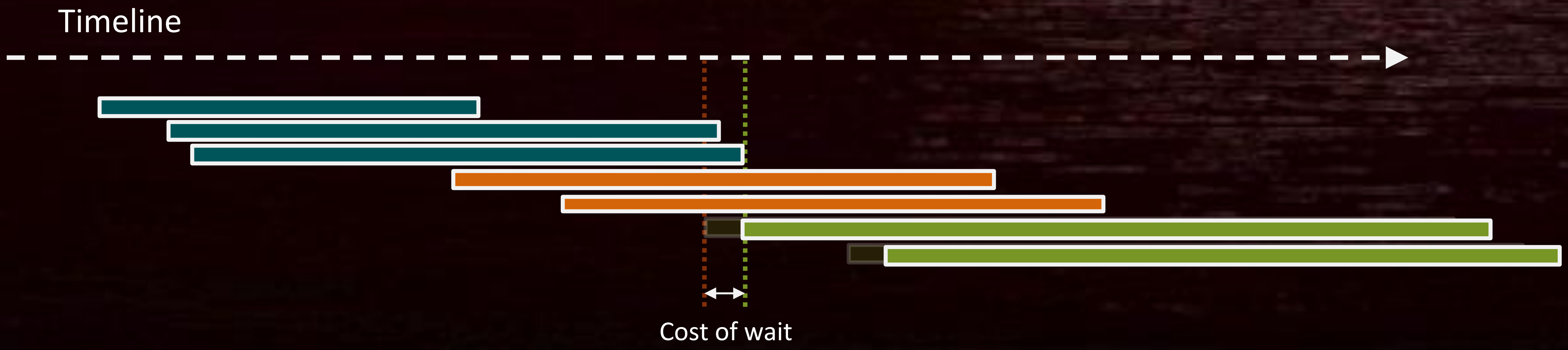
- Putting both barriers in the same barrier call makes them both happen at once



Barrier 1-2 AND 1-3

SYNCHRONISATION SPLIT BARRIER

- Split barrier between draw 1 and draw 3
  - “Done” after draw 1, “Make ready” before draw 3
  - Now draw 2 is unaffected, and 3 only has to wait for 1 to finish

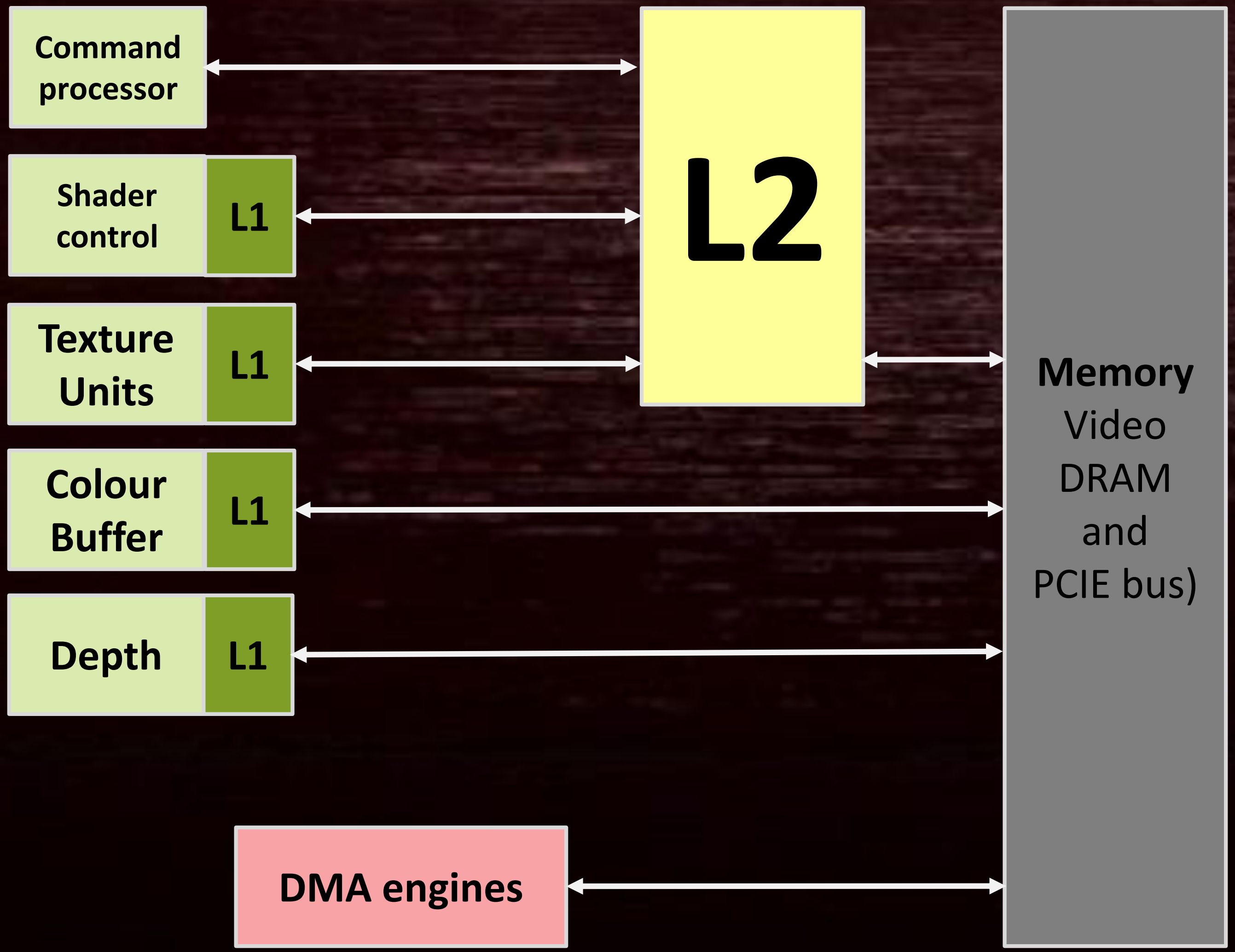


SYNCHRONISATION, SUMMARY

- Split barriers reduce synchronisation
  - –If there is other work between end of last use and start of new use
- Multiple simultaneous barriers can also reduce synchronisation
  - –Gets all the barriers out of the way in one go

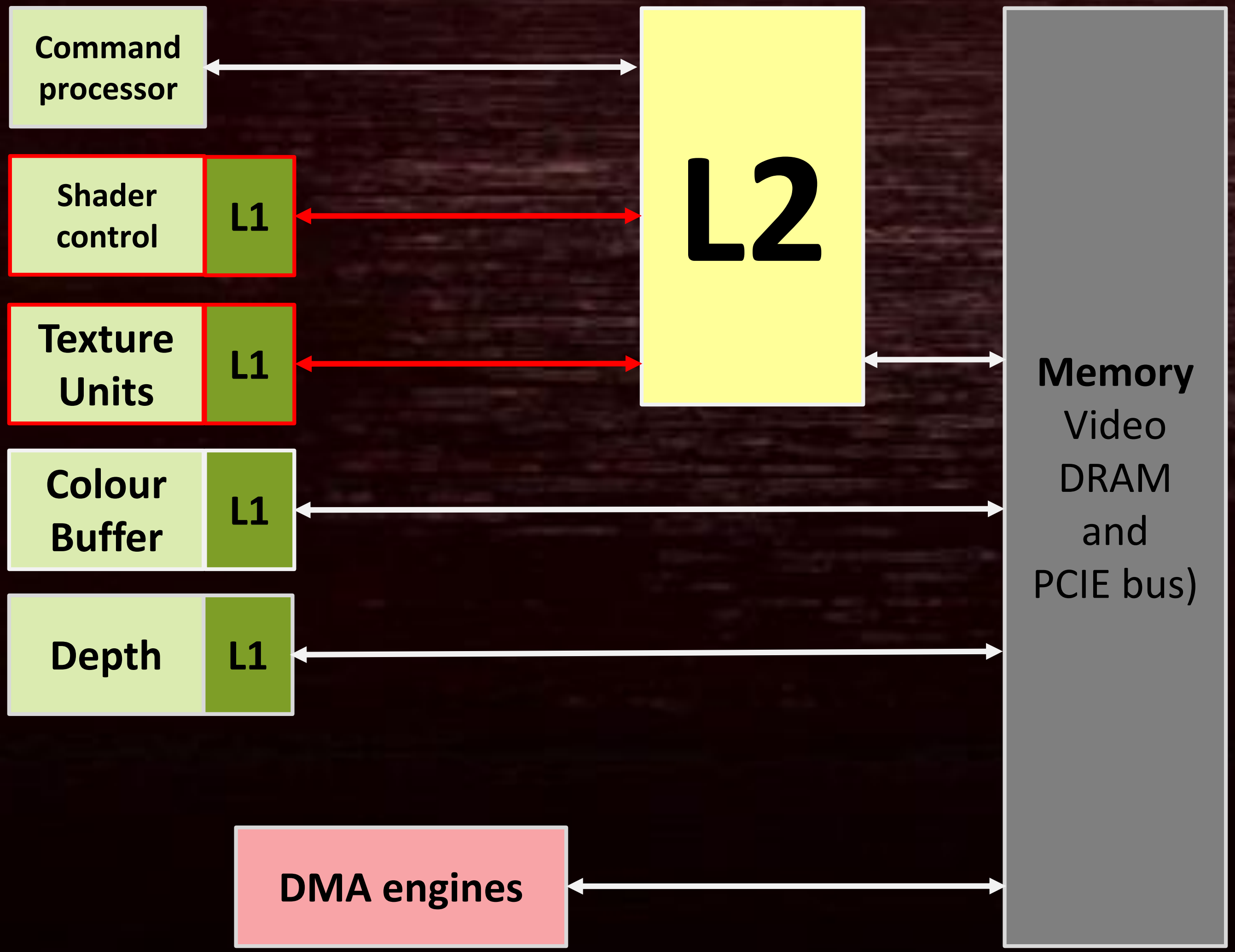
VISIBILITY

- Many small L1 "caches"
- Big L2 cache
  - –connected mostly to shader core



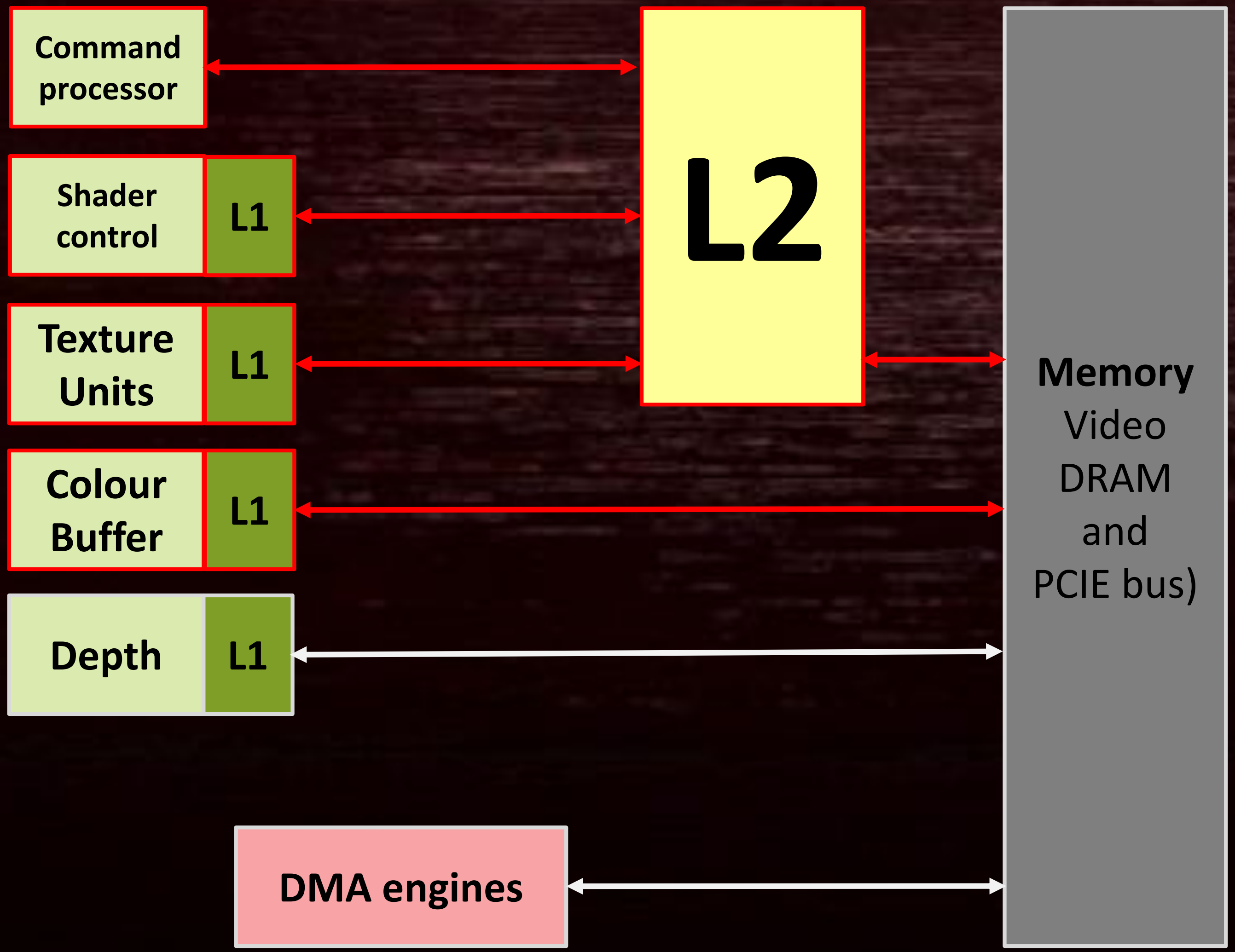
VISIBILITY – SIMPLE BARRIER

- UAV of buffer -> SHADER_RESOURCE | CONSTANT_BUFFER
  - Flushes Texture L1 to L2
  - Flushes shader L1
  - … that's it



VISIBILITY – TRANSITION TO THE COMMON STATE

- RENDER_TARGET -> COMMON
  - –Flushes Colour L1
  - –Flushes maybe all the L1s
  - –Flushes the L2
- More expensive
  - –Takes longer
  - –More memory traffic



VISIBILITY

- Multiple barriers in a single call reduce the cost of visibility
  - Flushes union of all flushes
  - Consider previous cases shown
    - Add extra RT->SRV cases cost nothing over RT->COMMON – free!
- Split barrier can also reduce cost of visibility
  - Note that this implies effort spent to watch and cancel out barriers

DECOMPRESSION

- RT and DS surfaces perform far better when compressed
  - Can be factor of 2 or more
- Two different kinds of compression on latest hardware
  - Full must be decompressed to be read other than RT or DS
  - Part will also work as SRV
- If you need decompressions, you have to take the hit somewhere
  - But it’s not hard to decompress when you don’t really need to
  - Essential to avoid these

BARRIERS, OPTIMISATION

- Barriers that don’t contain decompressions take ‘some µs’
- Barrier GPU cost is (mostly) measurable with timestamps
  - Rare that it should be more than a few %
    - Exceptions include decompresses of huge AA surfaces
- Shouldn’t need much more than **two barriers per written surface**

BAD PATTERNS

- RT->SRV->Copy_source->SRV->RT
  - Don’t forget you can combine states by OR-ing state flags together
  - Never do read to read barriers
  - Put it into the right state **first time**

- “Sometimes I copy from this, so I’ll always do RT->SRV|Copy”
  - RT->SR may be very cheap, RT->SRV|Copy may be very expensive
  - Put it into the **right** state first time

WORSE PATTERNS – SYMPTOMS OF WORKING TOO LOCALLY

- **If you do these your engine is not a Direct3D 12 engine**
  - Must think ahead, must think at a higher level
- “I don’t know what state this object is in next, so I’ll transition everything to COMMON at the end of every list”
  - The cost of this is enormous
    - Forces all surfaces to decompress
    - Most command lists effectively wait for idle before starting
- Only considering barriers just at use, and / or in an inner loop
  - Prevents combining barriers

THE WORST POSSIBLE EXAMPLE

```
void UploadTextures()
{
  for(auto resource : resources)
  {
    pD3D12CmdList->Barrier(resource, Copy);
    pD3D12CmdList->CopyTexture(src, dest);
    pD3D12CmdList->Barrier(resource, SR);
  }
}
```

TWO barriers per resource upload

Each is probably serialising at the GPU

THIS IS BETTER

```
void UploadTextures()
{
  BarrierList list;
  for(auto resource : resources)
    AddBarrier(list, resource, Copy)
  pD3D12CmdList->Barrier(list);
  list->clear();
  for(auto resource : resources)
     pD3D12CmdList-> CopyTexture(src, dest);
  for(auto resource : resources)
    AddBarrier(list, resource, SR)
  pD3D12CmdList->Barrier(list);
}
```

ONE barrier call with ALL resources

Now do the uploads

One more barrier to finish off

BARRIER SUMMARY

- Yeah, this is a bit hard
- Surfaces written every frame are the main problem
  - Written surface corruption? Barrier missing.
  - Two barriers per surface per frame is the target
    - (and fewer barrier calls).
- Use the tools

THANK YOU

Our thank you is a free video card for someone!

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION