Practical DirectX 12

- Programming Model and Hardware Capabilities

Gareth Thomas & Alex Dunn
AMD & NVIDIA

Agenda

- DX12 Best Practices
- DX12 Hardware Capabilities
- Questions

Expectations

Who is DX12 for?

- Aiming to achieve maximum GPU & CPU performance
- Capable of investing engineering time
- Not for everyone!

Engine Considerations

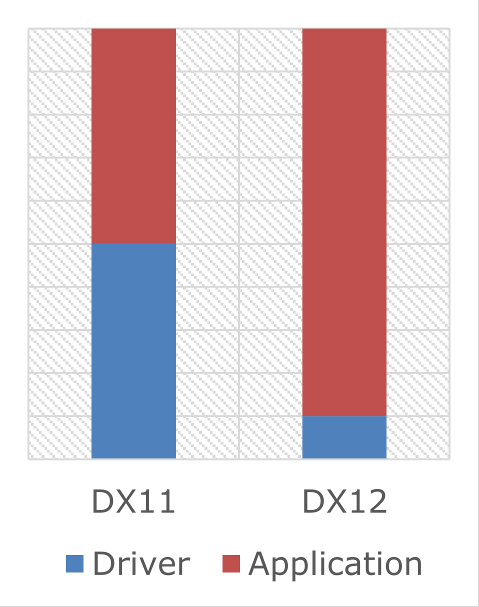
Need IHV specific paths

- Use DX11 if you can't do this

Application replaces portion of driver and runtime

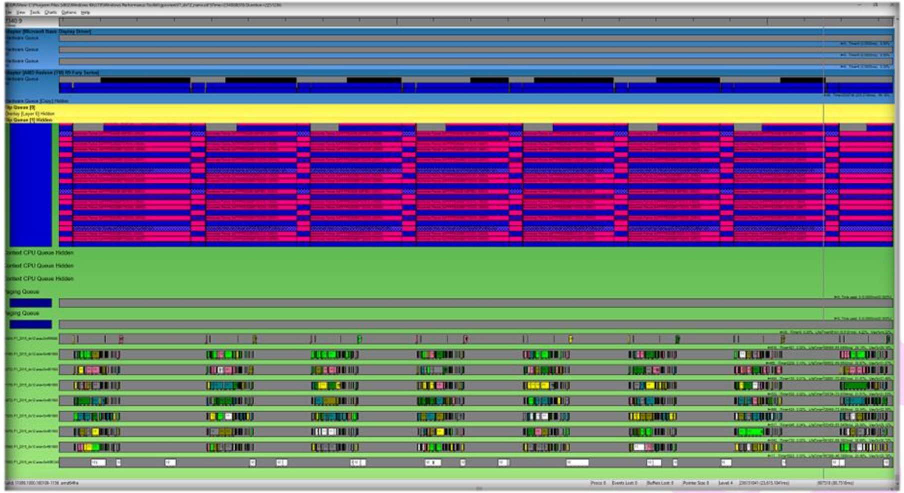
- Can't expect the same code to run well on all consoles, PC is no different
- Consider architecture specific paths

Look out for NVIDIA and AMD specifics



Work Submission

- Multi Threading
- Command Lists
- Bundles
- Command Queues



Multi-Threading

DX11 Driver:

- Render thread (producer)
- Driver thread (consumer)

DX12 Driver:

- Doesn't spin up worker threads.
- Build command buffers directly via the CommandList interface

Make sure your engine scales across all the cores

- Task graph architecture works best
- One render thread which submits the command lists
- Multiple worker threads that build the command lists in parallel

Command Lists

Command Lists can be built while others are being submitted

- Don’t idle during submission or Present
- Command list reuse is allowed, but the app is responsible for stopping concurrent-use

Don’t split your work into too many Command Lists

Aim for (per-frame):

- 15-30 Command Lists
- 5-10 ‘ExecuteCommandLists’ calls

Command Lists #2

Each ‵ ExecuteCommandLists’ has a fixed CPU overhead

- Underneath this call triggers a flush
- So batch up command lists

Try to put at least 200**µs** of GPU work in each ‘ExecuteCommandLists’, preferably 500**µs**

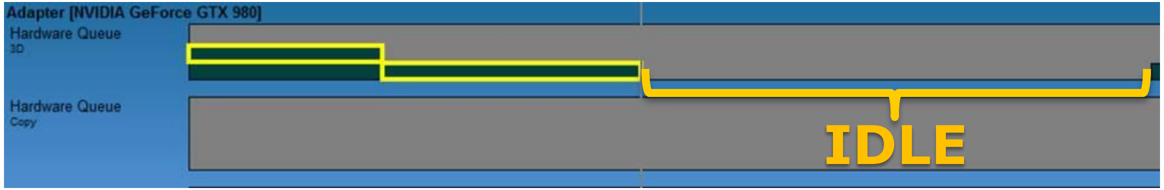
Submit enough work to hide OS scheduling latency

- Small calls to ‘ExecuteCommandLists’ complete faster than the OS scheduler can submit new ones

Command Lists #3

Example:

What happens if not enough work is submitted?



- Highlighted ECL takes ~20**μs** to execute
- OS takes ~60**μs** to schedule upcoming work
- == 40**μs** of idle time

Bundles

Nice way to submit work early in the frame

Nothing inherently faster about bundles on the GPU

- Use them wisely!

Inherits state from calling Command List – use to your advantage

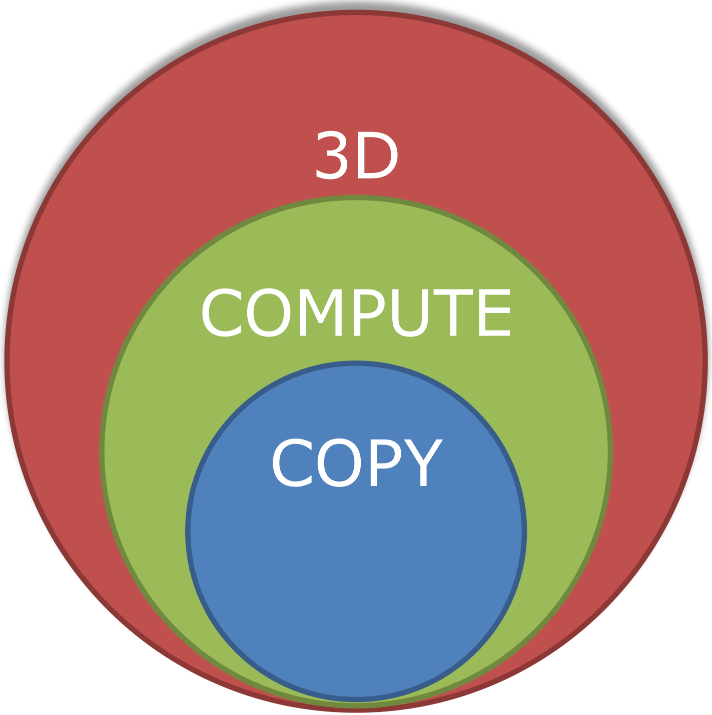
- But reconciling inherited state may have CPU or GPU cost

Can give you a nice CPU boost

- NVIDIA: repeat the same 5+ draw/dispatches? Use a bundle
- AMD: only use bundles if you are struggling CPU-side.

Multi-Engine

- 3D Queue
- Compute Queue
- Copy Queue



Compute Queue #1

Use with great care!

- Seeing up to a 10% win currently, if done correctly

Always check this is a performance win

- Maintain a non-async compute path
- Poorly scheduled compute tasks can be a net loss

Remember hyperthreading? Similar rules apply

- Two data heavy techniques can throttle resources, e.g. caches

If a technique suitable for pairing is due to poor utilization of the GPU, first ask "why does utilization suck?"

- Optimize the compute job first *before* moving it to async compute

Compute Queue #2

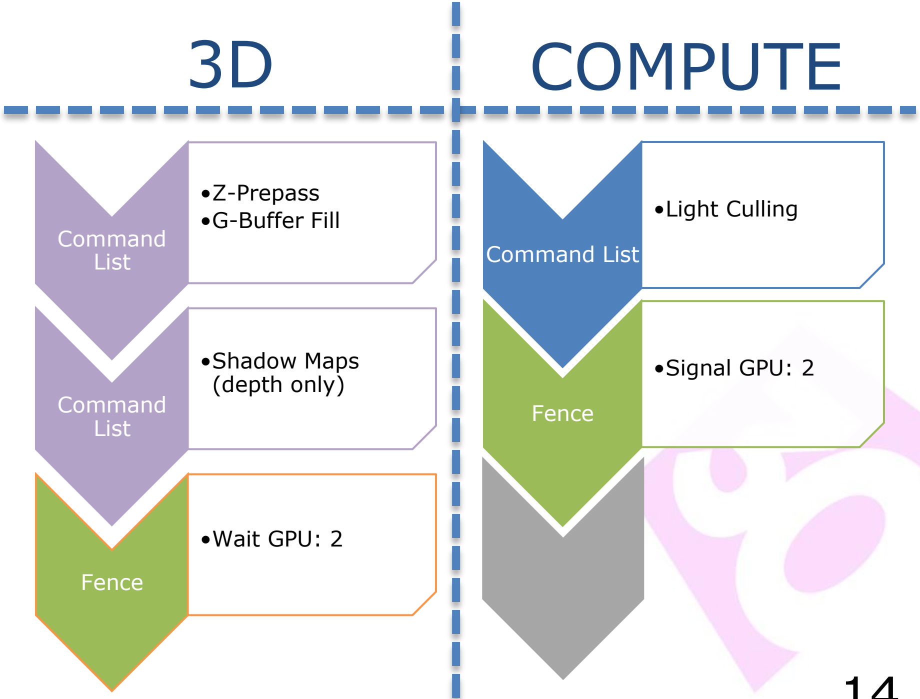
Good Pairing		**Poor Pairing**	
Graphics	Compute	Graphics	Compute
Shadow Render (I/O limited)	Light culling (ALU heavy)	G-Buffer (Bandwidth limited)	SSAO (Bandwidth limited)

(Technique pairing doesn’t have to be 1-to-1)

Compute Queue #3

Unrestricted scheduling creates opportunities for poor technique pairing

- Benefits are;
  - Simple to implement
- Downsides are;
  - Non-determinism frame-to-frame
  - Lack of pairing control

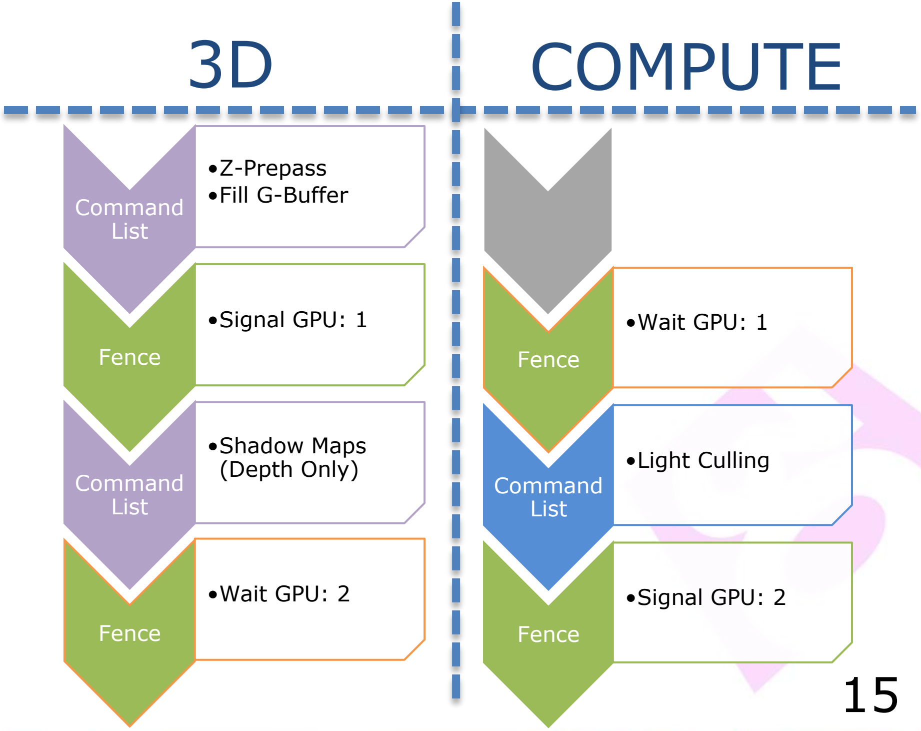


Compute Queue #4

Prefer explicit scheduling of async compute tasks through smart use of fences

- Benefits are;
  - Frame-to-frame determinism
  - App control over technique pairing!

- Downsides are;
  - It takes a little longer to implement



Copy Queue

Use the copy queue for background tasks

- Leaves the Graphics queue free to do graphics

Use copy queue for transferring resources over PCIE

- Essential for asynchronous transfers with multi-GPU

Avoid spinning on copy queue completion

- Plan your transfers in advance

NVIDIA: Take care when copying depth+stencil resources – copying only depth may hit slow path

Hardware State

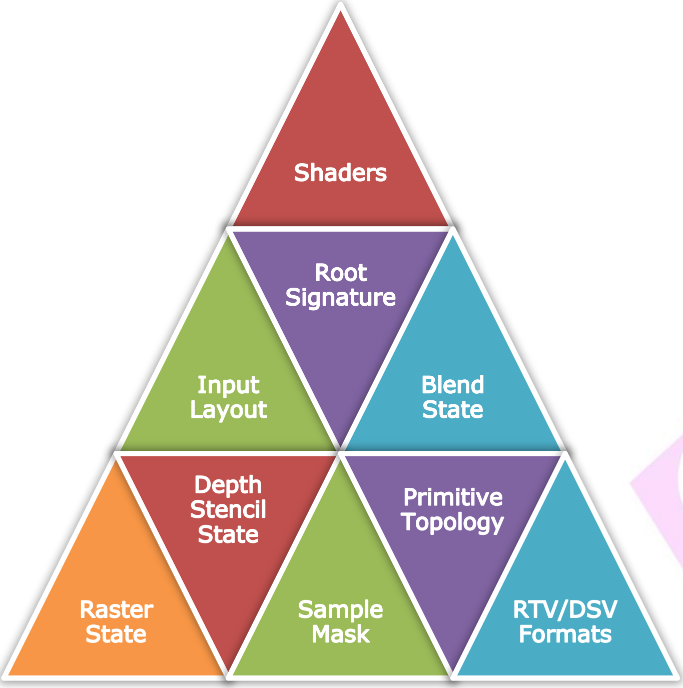
- Pipeline State Objects (PSOs)
- Root Signature Tables (RSTs)

Pipeline State Objects #1

Use sensible and consistent defaults for the unused fields

The driver is not allowed to thread PSO compilation

- Use your worker threads to generate the PSOs
- Compilation may take a few hundred milliseconds



Pipeline State Objects #2

Compile similar PSOs on the same thread

- e.g. same VS/PS with different blend states
- Will reuse shader compilation if state doesn’t affect shader
- Simultaneous worker threads compiling the same shaders will wait on the results of the first compile.

Root Signature Tables #1

Keep the RST small

- Use multiple RSTs
- There isn’t one RST to rule them all…

Put frequently changed slots first

Aim to change one slot per draw call

Limit resource visibility to the minimum set of stages

- Don’t use D3D12_SHADER_VISIBILITY_ALL if not required.
- Use the DENY_*_SHADER_ROOT_ACCESS flags

Beware, no bounds checking is done on the RST!

Don’t leave resource bindings undefined after a change of Root Signature

Root Signature Tables #2

AMD: Only constants and CBVs changing per draw should be in the RST

AMD: If changing more than one CBVs per draw, then it is probably better putting the CBVs in a table

NVIDIA: Place all constants and CBVs in RST

- Constants and CBVs in the RST do speed up shaders
- Root constants don’t require creating a CBV == less CPU work

Memory Management

- Command Allocators
- Resources
- Residency

Command Allocators

Aim for number of recording threads * number of buffered frames + extra pool for bundles

- If you have hundreds of allocators, you are doing it wrong

Allocators only grow

- Can never reclaim memory from an allocator
- Prefer to keep them assigned to the command lists

Pool allocators by size where possible

Resources – Options?

Type	Physical Page	Virtual Address
Committed	✓	✓
Heap	✓	✗
Placed	✗	✓
Reserved	✗	✓

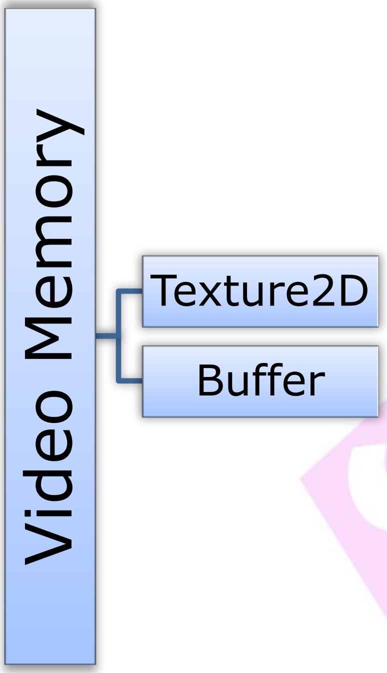
Committed Resources

Allocates the minimum size heap required to fit the resource

App has to call MakeResident/Evict on each resource

App is at the mercy of OS paging logic

- On ‘MakeResident’, the OS decides where to place resource
- You're stuck until it returns



Heaps & Placed Resources

Creating larger heaps

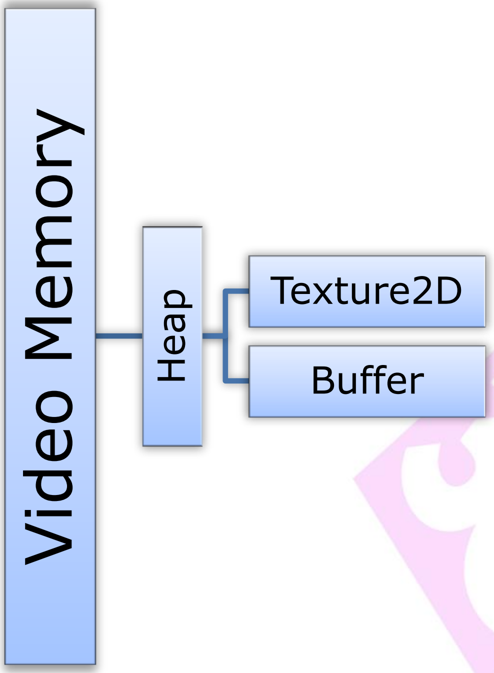
- In the order of 10-100 MB
- Sub-allocate using placed resources

Call MakeResident/Evict per heap

- Not per resource ☺

This requires the app to keep track of allocations

- Likewise, the app needs to keep track of free/used ranges of memory in each heap



Residency

MakeResident/Evict memory to/from GPU

- CPU + GPU cost is significant so batch MakeResident and UpdateTileMappings
- Amortize large work loads over multiple frames if necessary
- Be aware that Evict might not do anything immediately

MakeResident is **synchronous**

- MakeResident will not return until the resource is resident
- The OS can go off and spend a LOT of time figuring out where to place resources. You're stuck until it returns
- Be sure to call on a worker thread

Residency #2

How much vidmem do I have?

- IDXGIAdapter3::QueryVideoMemoryInfo(…)
- Foreground app is guaranteed a subset of total vidmem
  - The rest is variable, app should respond to budget changes from OS

App must handle MakeResident fail.

- Usually means there's not enough memory available
- But can happen even if there is enough memory (fragmentation)

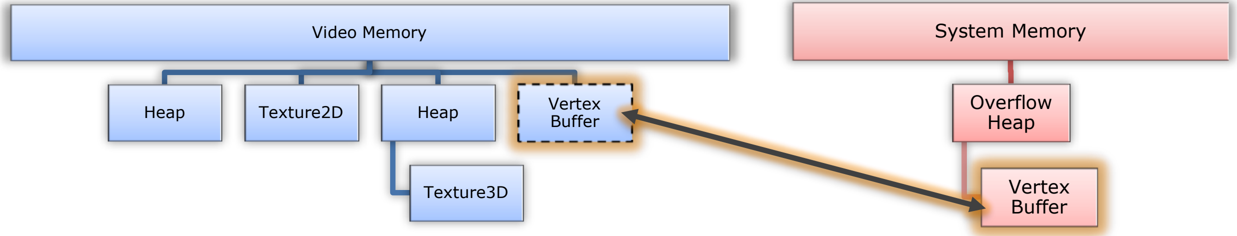
Non-resident read is a page fault! Likely resulting in a fatal crash

What to do when there isn't enough memory?

Vidmem Over-commitment

Create overflow heaps in sysmem, and move some resources over from vidmem heaps.

- The app has an advantage over any driver/OS here, arguably it knows what's most important to keep in vidmem



Idea: Test your application with 2 instances running

Resources: Practical Tips

Aliasing targets can be a significant memory saving

- Remember to use aliasing barriers!

Committed RTV/DSV resources are preferred by the driver

NVIDIA: Use a constant buffer instead of a structured buffer when reads are coherent. e.g. tiled lighting

Synchronization

- Barriers
- Fences

Barriers #1

<u>Don’t let resource barriers become a performance barrier!</u>

Batch barriers together

Use the minimum set of usage flags

- Avoiding redundant flushing

Avoid read-to-read barriers

- Get the resource in the right state for all subsequent reads

Use “split-barriers” when possible

Barriers #2

COPY_SOURCE will probably be significantly more expensive than SHADER_RESOURCE

- Enables access on the copy queue

Barrier count should be roughly double the number of surfaces written to

Fences

GPU Semaphore

- e.g. Make sure GPU is done with resource before evict

Each fence is about the same CPU and GPU cost as ExecuteCommandLists

Don’t expect fences to trigger signals/advance at a finer granularity than once per ExecuteCommandLists call

Miscellaneous

- Multi-GPU
- Swap Chains
- Set Stable Power State
- Pixel vs Compute

Multi GPU

Functionality now embedded in DirectX 12 API

Trade-offs for cross-adapter vs. linked-node

- See **Juha Sjöholm's** talk later today for more on this

Explicitly sync resources across devices

- Use proper CreationNodeMask

Be mindful of PCIe bandwidth

- PCI 3.0 (8x) – 8GB/s (expect ~6GB/s)
- PCI 2.0 (8x) – 4GB/s (expect ~3GB/s) ← Still common…
  - e.g. transferring a 4k HDR buffer will limit you to ~50/100 FPS right away

Swap Chains

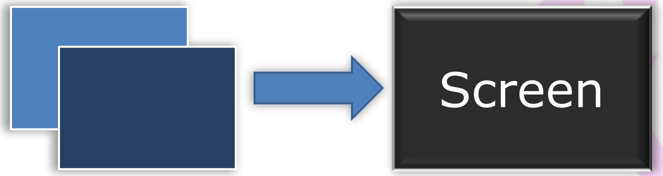
App must do explicit buffer rotation!

- IDXGISwapChain3::GetCurrentBackBufferIndex()

To replicate VSYNC-OFF

- SetFullScreenState(TRUE)
- Use a borderless fullscreen window
- Flip model swap-chain mode

Very rich, new API!



Set Stable Power State

```
HRESULT ID3D12Device::SetStablePowerState( BOOL Enable );
```

- This reduces performance
- Alters performance ratio of GPU components within chip

Don’t do it! (Please)

Pixel vs Compute - Performance

NVIDIA		AMD	
□ No shared memory? □ Threads complete at same time? □ High frequency cbuffer accesses? □ 2D buffer stores?	□ Using group shared memory? □ Expect out-of-order thread completion? □ Using high # regs? □ 1D/3D buffer stores	□ Benefit from depth/stencil rejection? □ Requires graphics pipeline? □ Want to leverage color compression?	□ Everything else ☺
Pixel Shader	Compute Shader	Pixel Shader	Compute Shader

<u>Best performance gained from following these guidelines</u>

(Consider the perf benefit of using async compute)

Hardware Features

- Conservative Rasterization
- Volume Tiled Resources
- Raster Ordered Views
- Typed UAV Loads
- Stencil Output

Hardware Features Stats

	AMD Radeon		**NVIDIA GeForce**		**Intel HD Graphics**
	GCN 1.1	GCN 1.2	Kepler	Maxwell 2	Skylake
Feature Level	12_0		11_0	12_1	12_1
Resource Binding	Tier 3		Tier 2		Tier 3
Tiled Resources	Tier 2		Tier 1	Tier 3	Tier 3
Typed UAV Loads	Yes		No	Yes	Yes
Conservative Rasterization	No		No	Tier 1	Tier 3
Rasterizer-Ordered Views	No		No	Yes	Yes
Stencil Reference Output	Yes		No		Yes
UAV Slots	full heap		64		full heap
Resource Heap	Tier 2		Tier 1		Tier 2

Conservative Rasterization

Draws all pixels a primitive touches

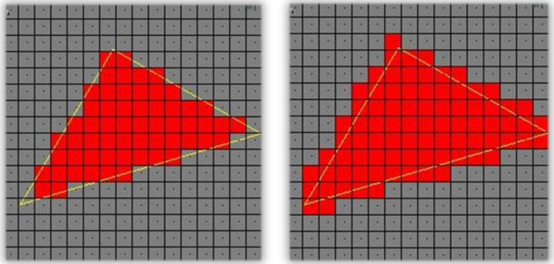
- Different Tiers – See spec

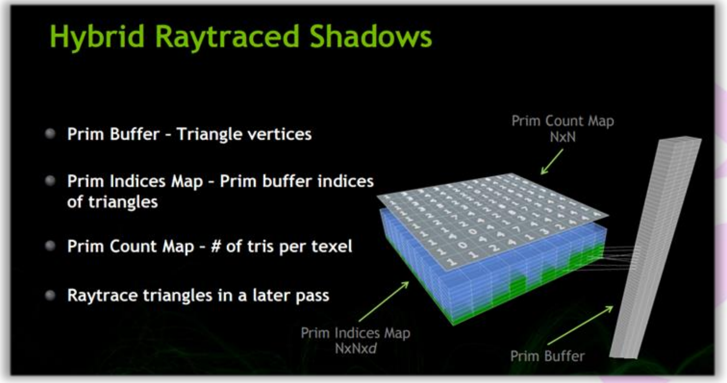
Possible before through GS trick but relatively slow

- See; J. Hasselgren et. Al, "Conservative Rasterization", GPU Gems 2

Now we can use rasterization do implement some nice techniques!

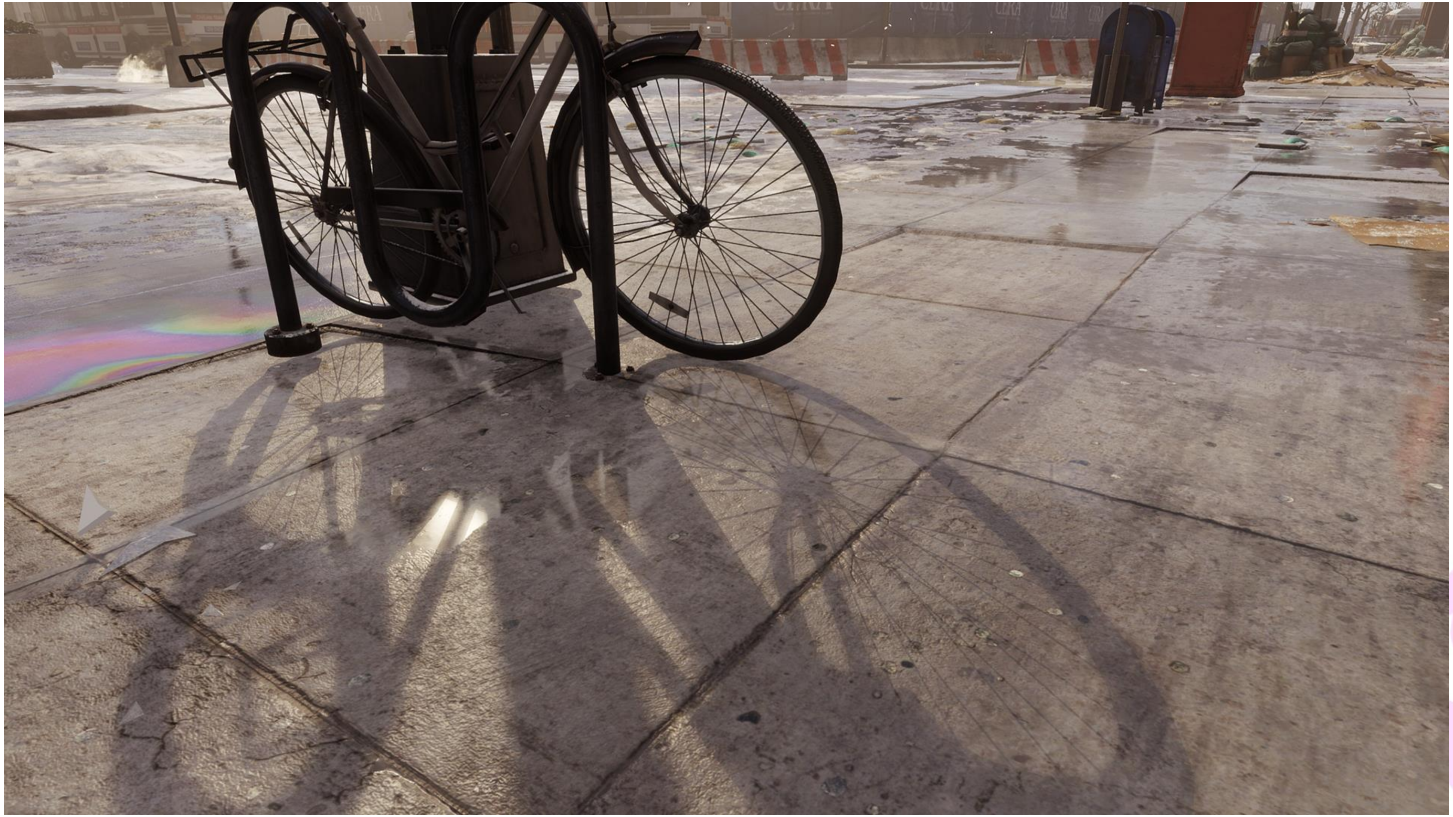
- See; Jon Story, "Hybrid Raytraced Shadows", D3D day - GDC 2015





Hybrid Raytraced Shadows

- Prim Buffer - Triangle vertices
- Prim Indices Map - Prim buffer indices of triangles
- Prim Count Map - # of tris per texel
- Raytrace triangles in a later pass



Ray traced shadows in, 'Tom Clancy's The Division', using conservative rasterization

Volume Tiled Resources

Tiled resources from DX11.2, now available for volume/3D resources

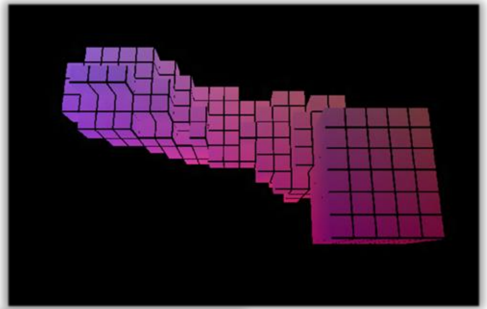
- Tier 3 tiled resources

Tiles are still 64kb

- And tile mapping still needs to be updated from the CPU

Extreme memory/performance benefits

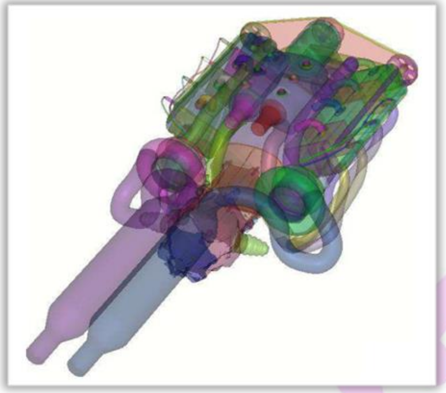
- Latency Resistant Sparse Fluid Simulation [Alex Dunn, D3D Day – GDC 2015]



Raster Ordered Views

Ordered writes

- Classic use case is OIT
  - See K+ Buffer OIT [Andreas A. Vasilakis, SIGGRAPH, 2014]
- Programmable blending
  - Completely custom blending, not bound by fixed HW



Use with care! Not free

- Watch the # conflicts

Typed UAV Loads

Finally, no more 32-bit restriction from the API

May allow you to remove console specific paths in engine

Loading from UAV slower than loading from SRV

- So still use SRV for read-only access

```
// Can do this e.g.
RWTexture2D<float4>

// and in conjunction with ROV :)
RasterizerOrderedTexture2D<float4>
```

Stencil Output

Implementations?

- N-ary algorithm using stencil?
  - Previous; clear + N passes
  - Now; Single pass

Performance considerations

- Comparable to using depth out

Questions?

adunn@nvidia.com
@AlexWDunn
#HappyGPU

gareth.thomas@amd.com
# DX12PerfTweet