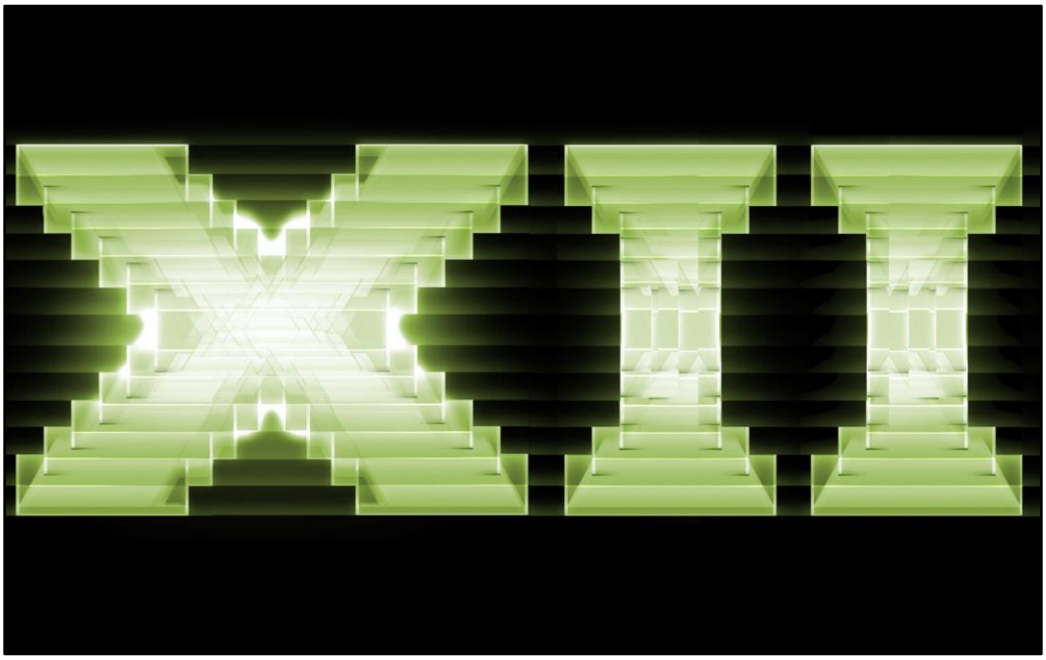
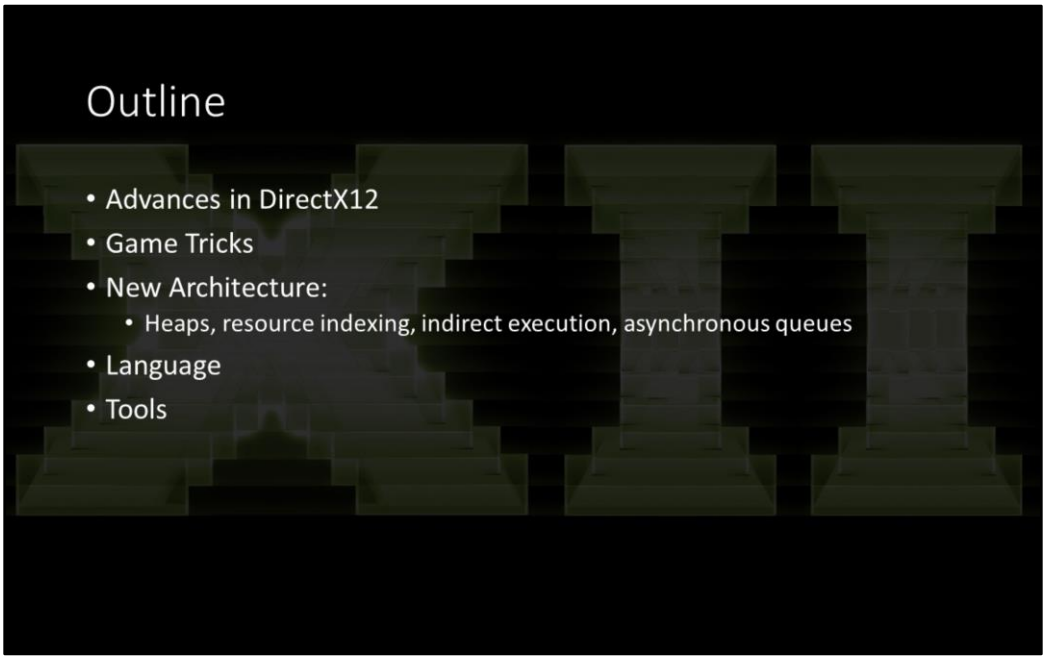


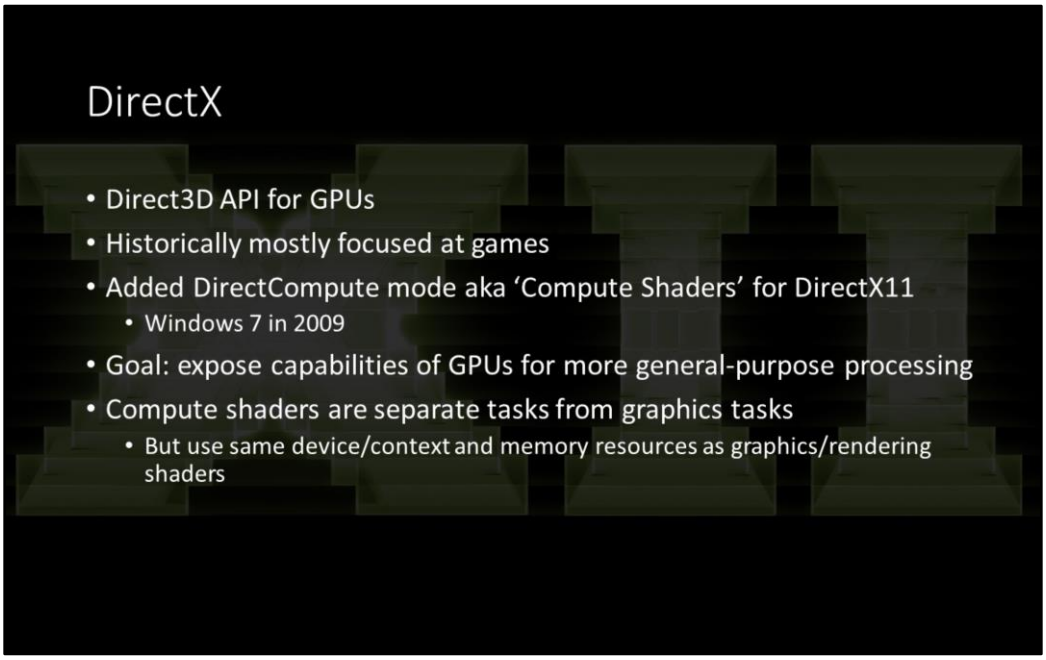
Day: Thursday, 03/19
Time: 16:00 - 16:50
Location: Room 212A
Level: Intermediate
Type: Talk
Tags: Developer - Tools & Libraries; Game Development

GTC Abstract

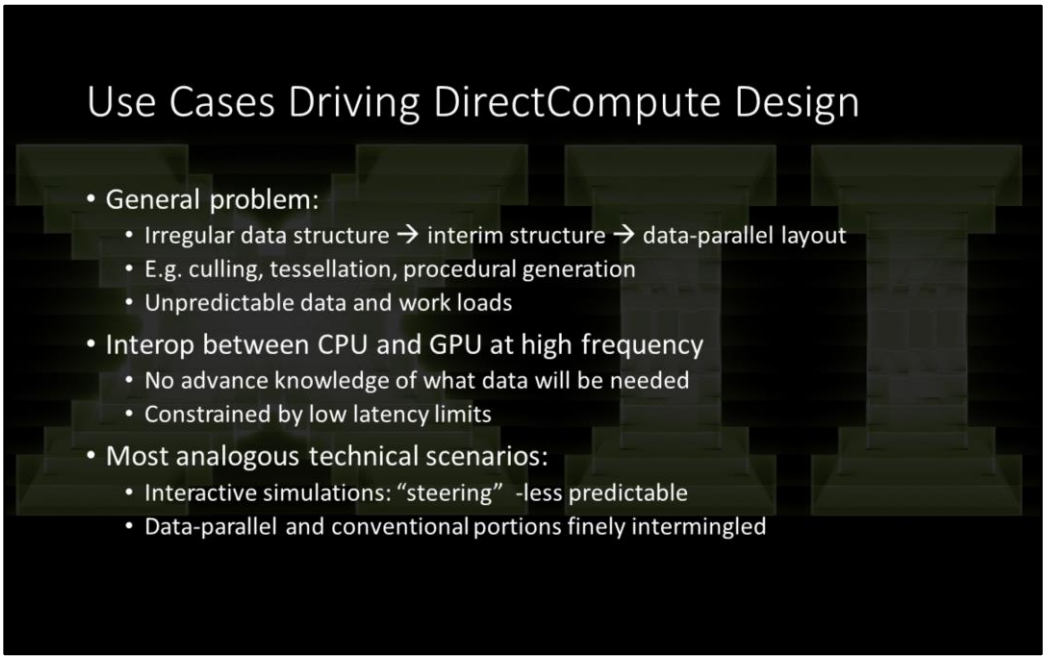
- DirectCompute, first introduced in Windows7/DirectX11 is currently used in many of the latest high-performance 3D games. Now, DirectX 12 adds new innovations that both improve performance-critical game scenarios and further broaden the applicability of GPU computing. Come and learn how game developers exploit current DirectCompute and new DirectX12 compute capabilities and find out which ones can be beneficial to your use case. Techniques covered include persistent mapped memory ranges, hardware image format conversion, default asynchronous resource access, and asynchronous task dispatch. We will also present the improvements to the GPU programming language which support these advancements, and the advanced IDE tools for performance profiling and development..



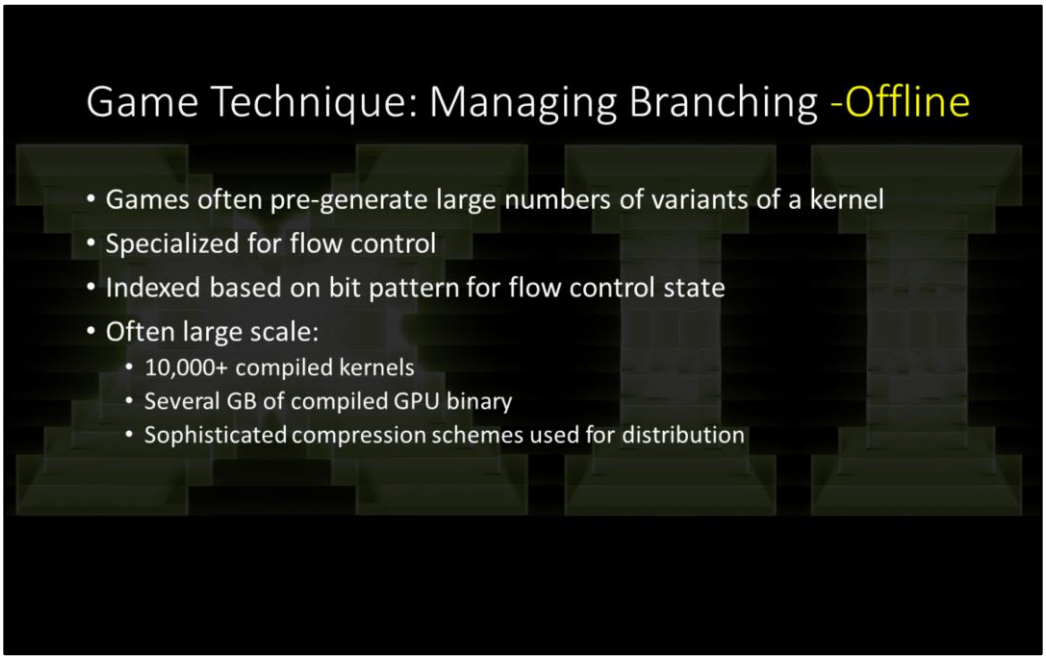
Talk about just some of the features of DX12 that are most relevant to technical computing or compute-scenarios.



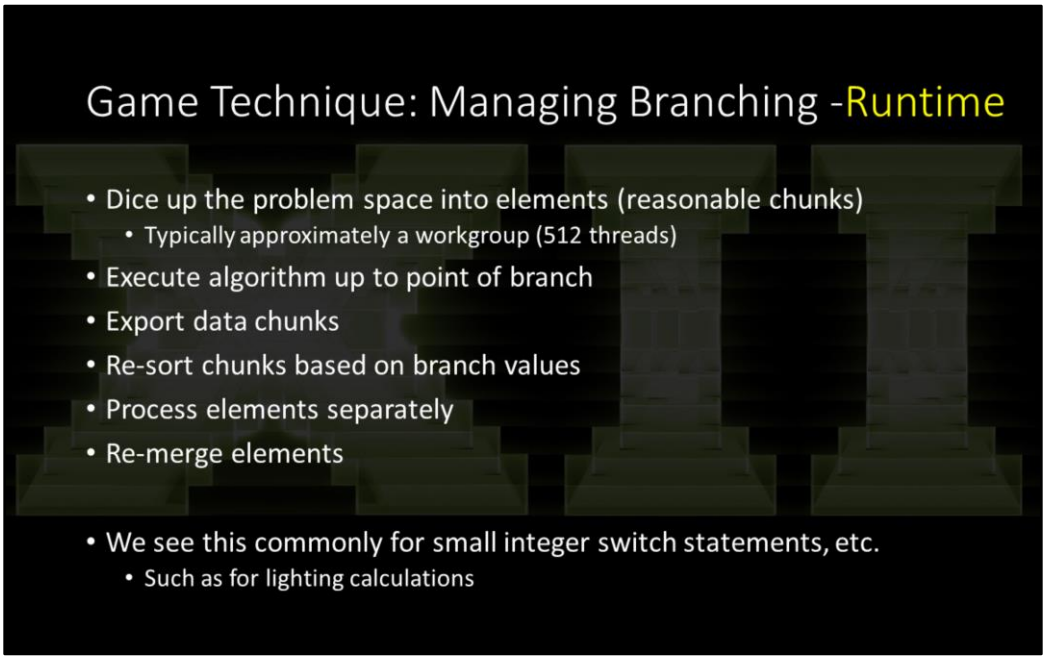
Compute shaders are not just another stage in the pipeline



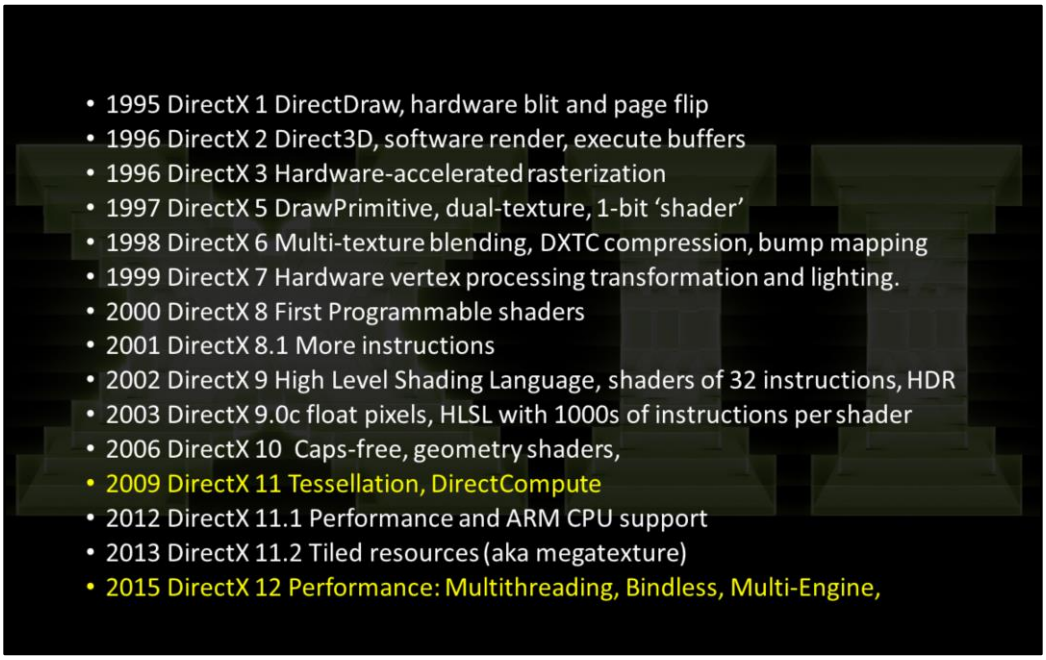
Games start with reality (very complex) and reduce it to an array of pixels
From unstructured (CPU) to data-parallel on GPU.
Compute shaders are a nice intermediate step in this pipeline.
Effectively enables an interim level of processing (not fully pointer chasing, but not burdened with graphics-specific semantics).
Ask: Why don't you just condition all the data beforehand (compile time).
Because it's a game, we don't know what data is going to be needed in a given frame.



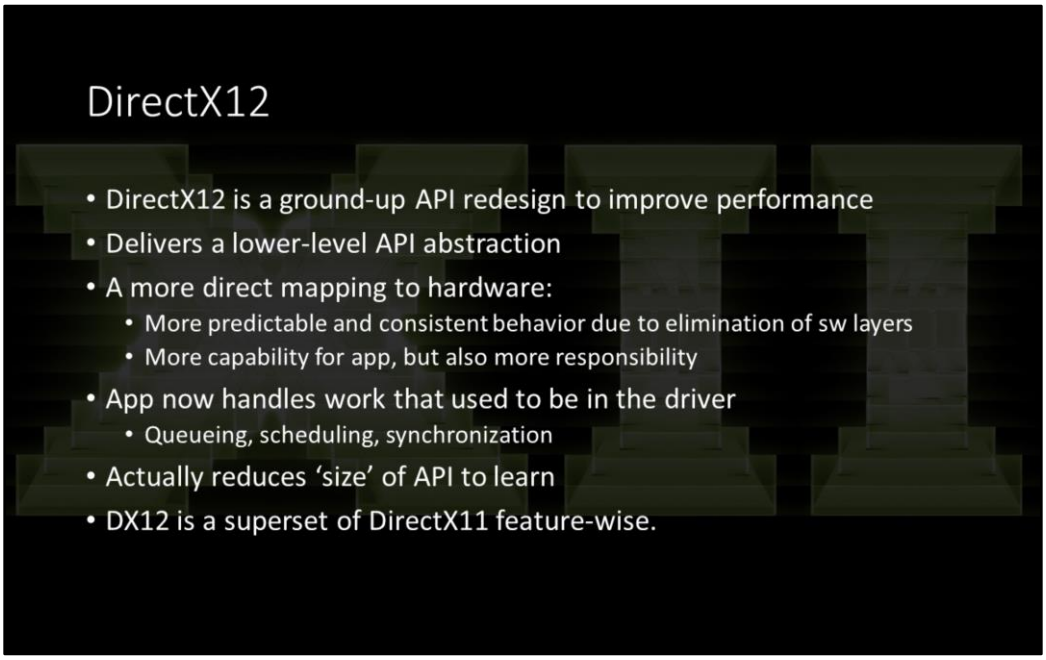
Maps to a kind of offline auto-tuning.
Used by Valve's Source engine



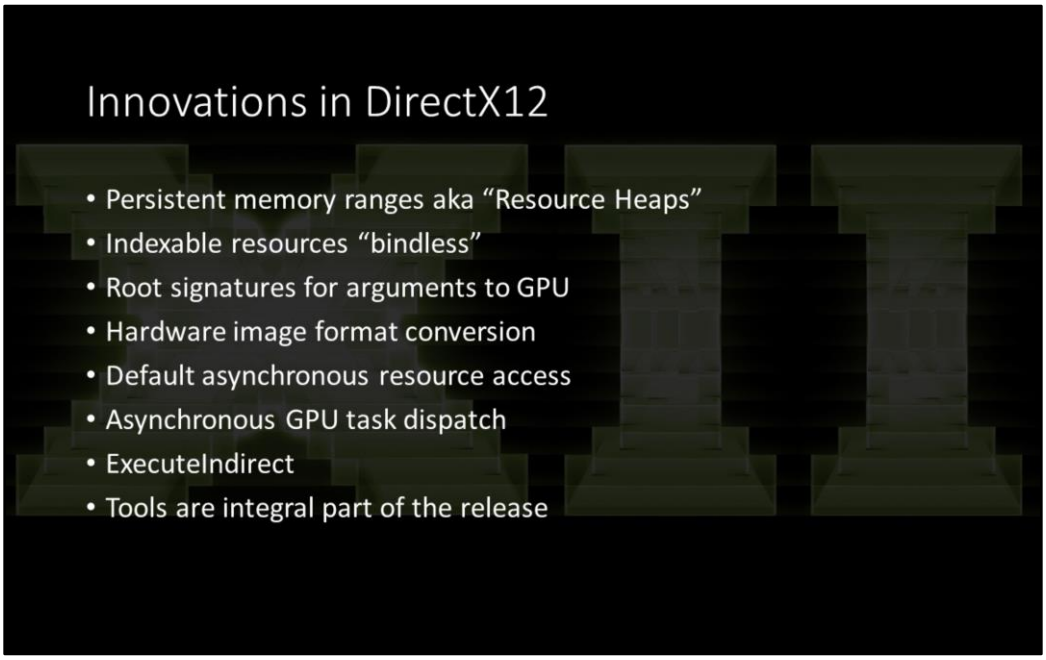
Branching performance is a fundamental characteristic of data-parallel processors.
Some are attempting this on a per warp/wavefront basis now too, although this is difficult with DX due to absence of warp-level operations.
Visualizing workgroups on the dataset is helpful, or at least stats on where the divergence is happening.
Used by DICE's FrostBite engine in Battlefield 3 and later.



Evolution of DirectX releases and key features of each version



Classic example of synchronization
DX11 forcibly serializes accesses on resource boundaries. Even in compute tasks.
This is overkill if you know you aren't writing to the same areas of that resource.



These are the innovations in DX12 that are relevant to Compute-related workloads
This is the outline of the rest of the talk.

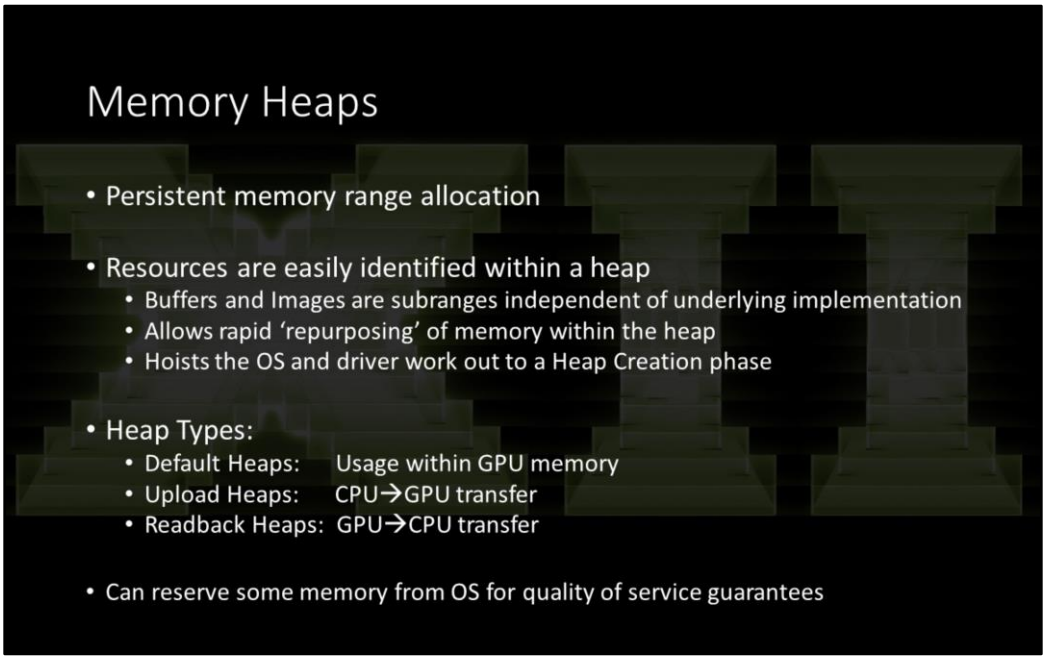
Command List-based API

- Create a Device object
- Create a Command Queue for use on main thraed
- Create Command Lists on separate threads/cores
  - Using a command allocator per thread
- Execute Command Lists via the Command Queue
  - On the original thread
- Non blocking by default

Graphics State

- Blend modes
- Shaders
- Etc.

- All packed into a single PSO “Pipeline State Object”
- Provided with each command list



In Earlier APIs, memory was virtualized. A 1GB video card would fill up and we would swap resource objects back to system memory using an LRU policy
You couldn't really know how much memory was actually present.

In DX12, Resources can now be allocated easily within these heaps and changed in-place or dynamically to different types with no overhead
(the driver and kernel aren't even aware of these changes).

Old model of DirectX was that app would specify expected usage of a resource, then the OS would define appropriate cache policies (write combine, writeback, etc). Now these map directly to those cache policies, they are basically just shortcuts for specifying the memory type yourself. Another example of a lower-level API abstraction.

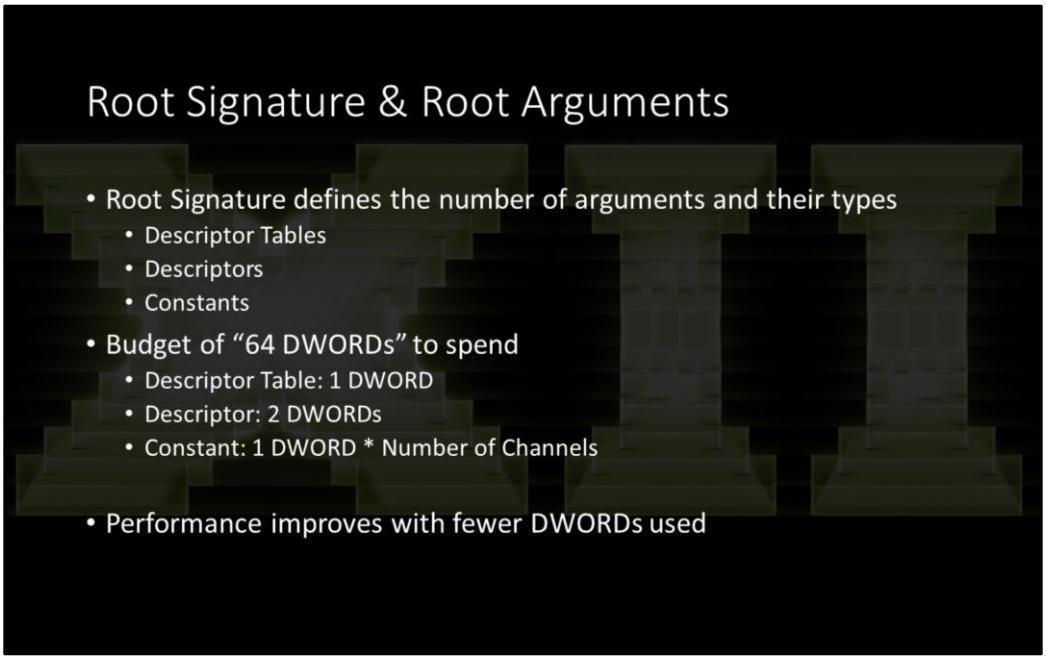
Bindless Resources

- 1M resources can be nominated for use and indexed at runtime
- Each has a header called a ‘resource descriptor’
- These are stored in an array (descriptor heap) for convenience
  - Which also minimizes fragmentation
- Concept of a ‘resource table’ is a subrange within this array
  - Convenient for passing to shaders

Flexible Pipeline Parameterization

- Two parts: Root Signature and Root Arguments
- Contains constants, descriptors, and descriptor tables
- Leverage hardware specific registers and pipelined renaming paths for highest frequency parameters
- Remove indirection from a constant descriptor index to an explicit descriptor

No need to reset entire set of bindings for a few high-frequency descriptor changes
Like a function call for the PSO
Pass Values and Pointers



Analog of function signature and function call arguments
The (argc, argv[]) for your GPU code
main(int argc, char *argv[]) {};

API – Root Signature

```
struct D3D12_ROOT_SIGNATURE_SLOT
{
    D3D12_ROOT_ARGUMENT_TYPE ArgumentType;

    union
    {
        D3D12_DESCRIPTOR_TABLE_LAYOUT  DescriptorTable;
        D3D12_ROOT_CONSTANTS           Constants;
        D3D12_ROOT_DESCRIPTOR          Descriptor;
    }
    …
}
```

API – Root Signature Creation

```
D3D12_ROOT_SIGNATURE_SLOT SigSlots[4];
ID3D12RootSignature*      pSig;

SigSlots[0].ArgumentType = D3D12_ROOT_ARGUMENT_32BIT_CONSTANTS;
SigSlots[1].ArgumentType = D3D12_ROOT_ARGUMENT_CBV;
SigSlots[2].ArgumentType = D3D12_ROOT_ARGUMENT_DESCRIPTOR_TABLE;
SigSlots[3].ArgumentType = D3D12_ROOT_ARGUMENT_DESCRIPTOR_TABLE;
…

pDevice->CreateRootSignature(SigSlots, sizeof(SigSlots), &pSig);
```

API – HLSL Remains Unchanged

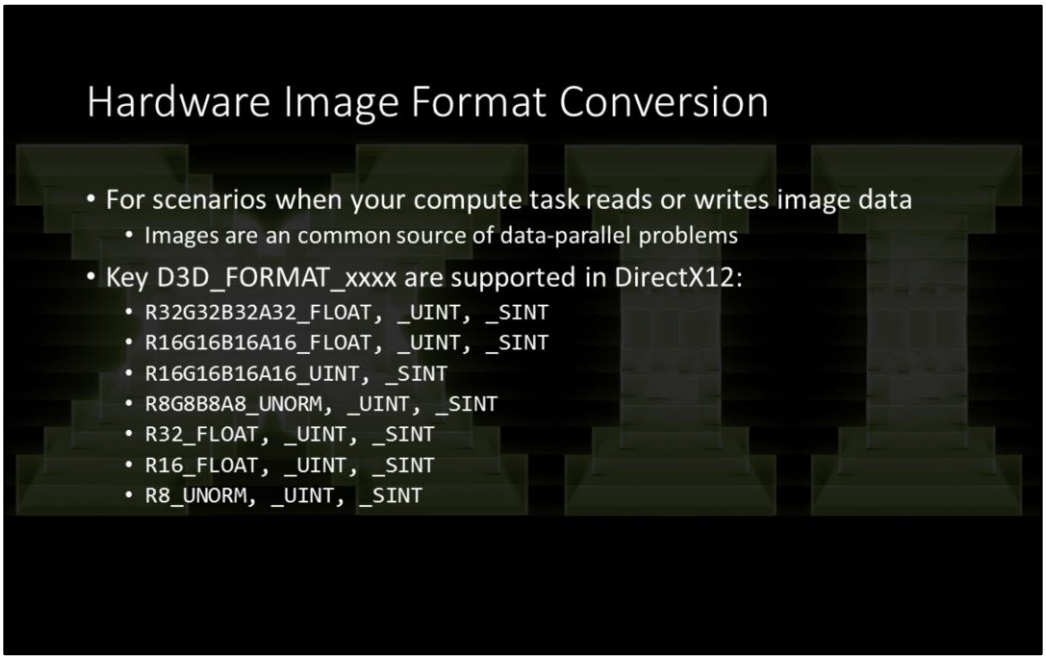
```
cbuffer DrawConstants
{
    UINT ConstantBufferOffset;
} : register(b0)

Buffer    ObjectPerDrawParams   : register(t7);
Texture2D ObjectTextureArray[5] : register(t2);
Sampler   ObjectSamplers[2]     : register(s0);
```

API – CreatePSO

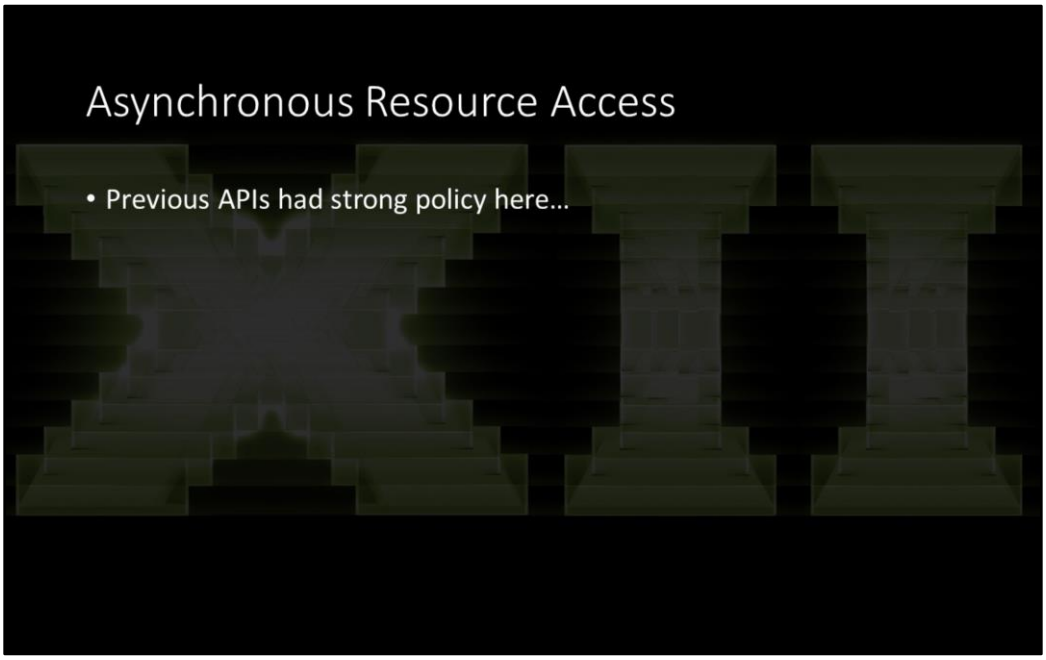
```
ID3D12Blob*             pShaderBytecode;
ID3D12RootSignature*    pRootSignature;
ID3D12PipelineState*    pPipelineState;

…

pDevice->CreatePSO(pShaderBytecode, pRootSignature, &pPipelineState);
```

API – Initializing Root Arguments

```
pCommandList->SetGraphicsRootSignature(pSignature);

pCommandList->SetGraphicsRoot32bitConstant(0, BaseOffsetInCBV);
pCommandList->SetGraphicsRootConstantBufferView(1, CBVDescriptorHandle);
pCommandList->SetGraphicsDescriptorTable(2, SamplerDescriptorTable);
pCommandList->SetGraphicsDescriptorTable(3, TextureDescriptorTable);
```



Before now, this sort of image pixel format conversion was considered part of the graphics use case for the chip.
and was absent from the data i/o pathways used in compute tasks. That has been fixed.
Float, signed, unsigned, and unorm variants of all the 4-channel and single-channel pixel formats are options.
Should improve performance substantially vs writing your own pack/unpack code.
Some implementations will support many more than this, but this is the guaranteed set for DX12 devices.


Asynchronous Resource Access
• Previous APIs had strong policy here…

GPU Efficiency: Explicit resource transitions

- Modern GPUs require resources to be in different ‘states’ for different use cases, and knowledge of when these transitions need to occur
- In DirectX 12, app is responsible for identifying when these transitions need to occur.
- Making these transitions explicit makes it clear when operations are expensive..

GPU Efficiency: Explicit resource transitions *(cont'd)*

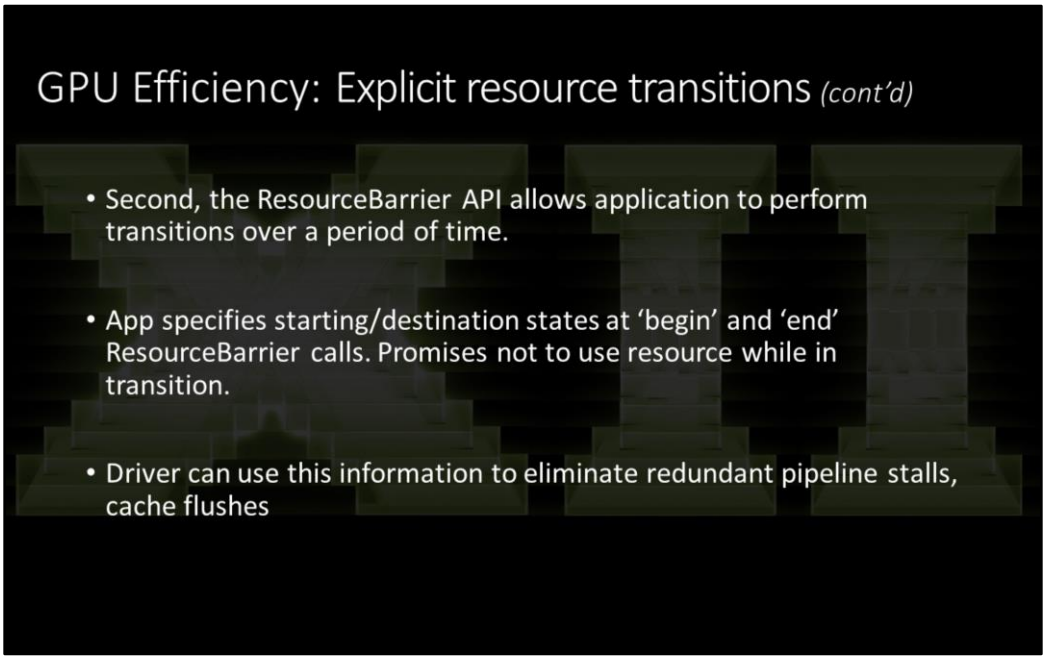
- .. but also gives games the opportunity to eliminate unnecessary transitions. Two key opportunities:
- First, UAV synchronization is now exposed as an explicit resource barrier.
- Previously, driver would ensure all writes to a UAV were in order of dispatch by inserting "Wait for Idle" commands after each dispatch.

GPU Efficiency: Explicit resource transitions *(cont'd)*

- If app has high-level knowledge that dispatches can run out of order, WaitForIdle's can be removed

- But more importantly, dispatches can then run in parallel to achieve higher GPU occupancy

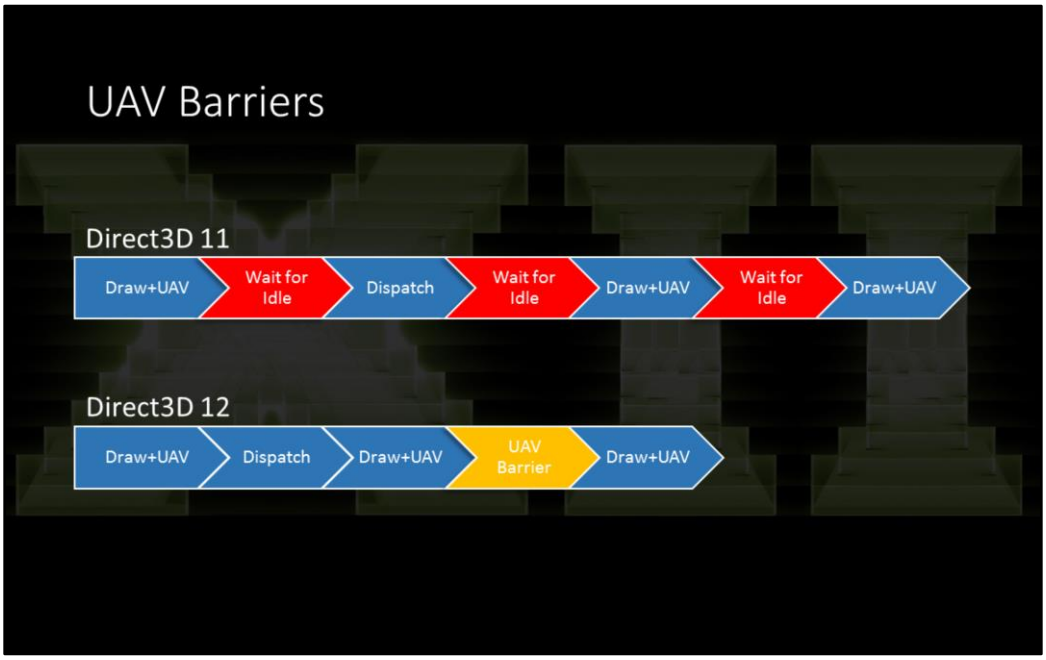
- Particularly beneficial for large numbers of dispatches with low thread counts

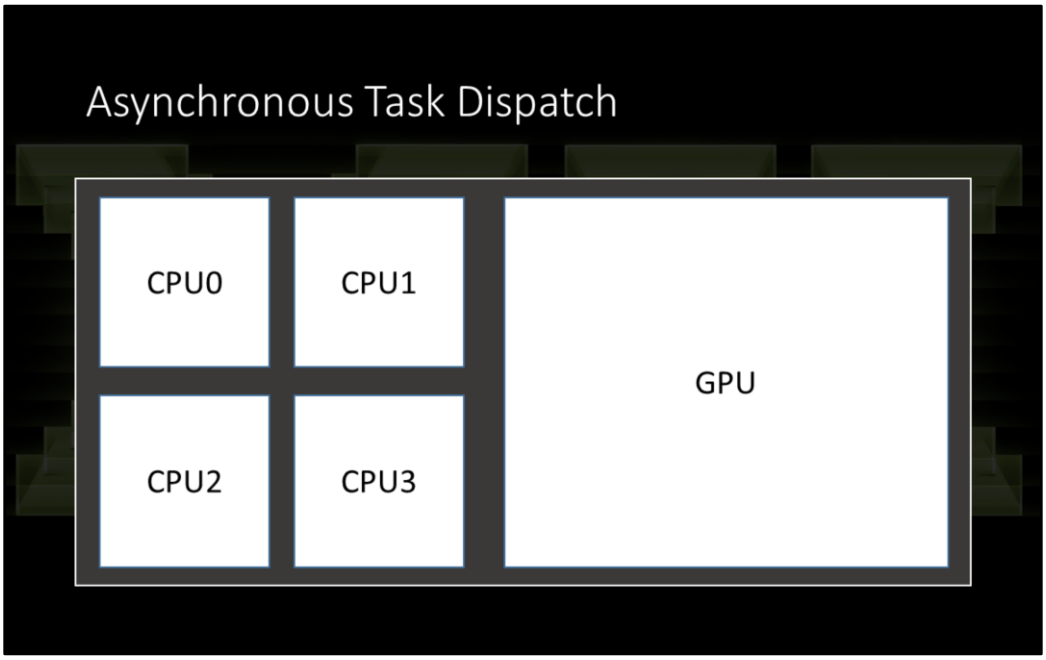


Spreading out this notification lets the implementation distribute work across time to avoid a sudden glitch

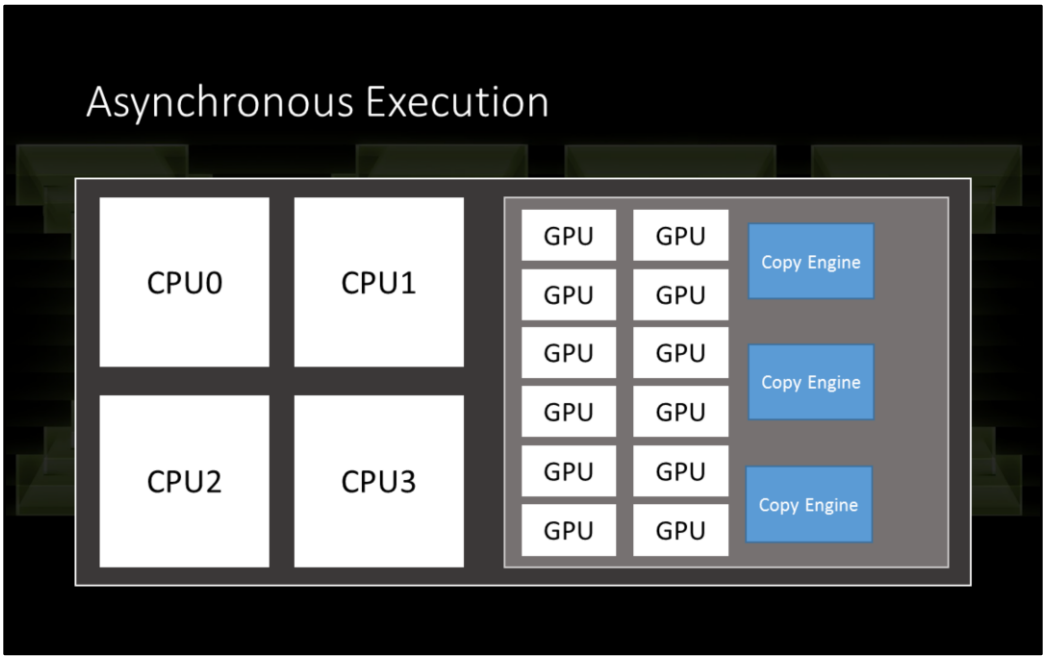
UAV Barriers

- In D3D11 all UAV accesses in 1 Draw/Dispatch must complete before any UAV accesses in a subsequent Draw/Dispatch
- This results in idle GPU shader cores for small Draw/Dispatch
- In D3D12 UAV accesses in multiple Draw/Dispatch are truly unordered, applications must use an explicit barrier to enforce ordering


UAV Barriers
Direct3D 11
Draw+UAV
Wait for Idle
Dispatch
Wait for Idle
Draw+UAV
Wait for Idle
Draw+UAV
Direct3D 12
Draw+UAV
Dispatch
Draw+UAV
UAV Barrier
Draw+UAV



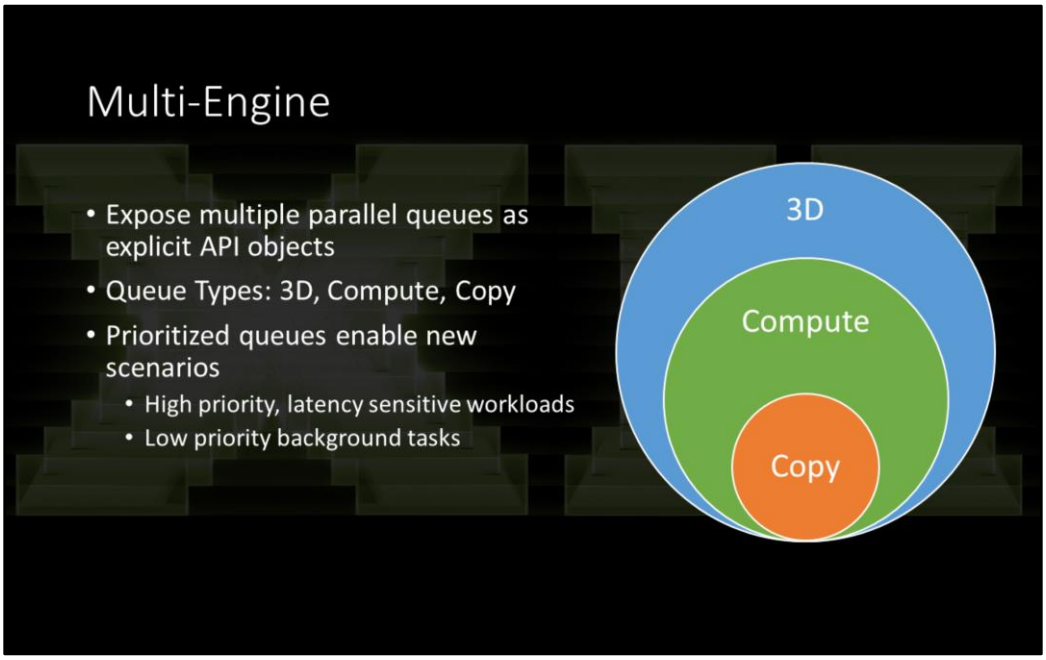
DX11 model was that the GPU was a single monolithic core.



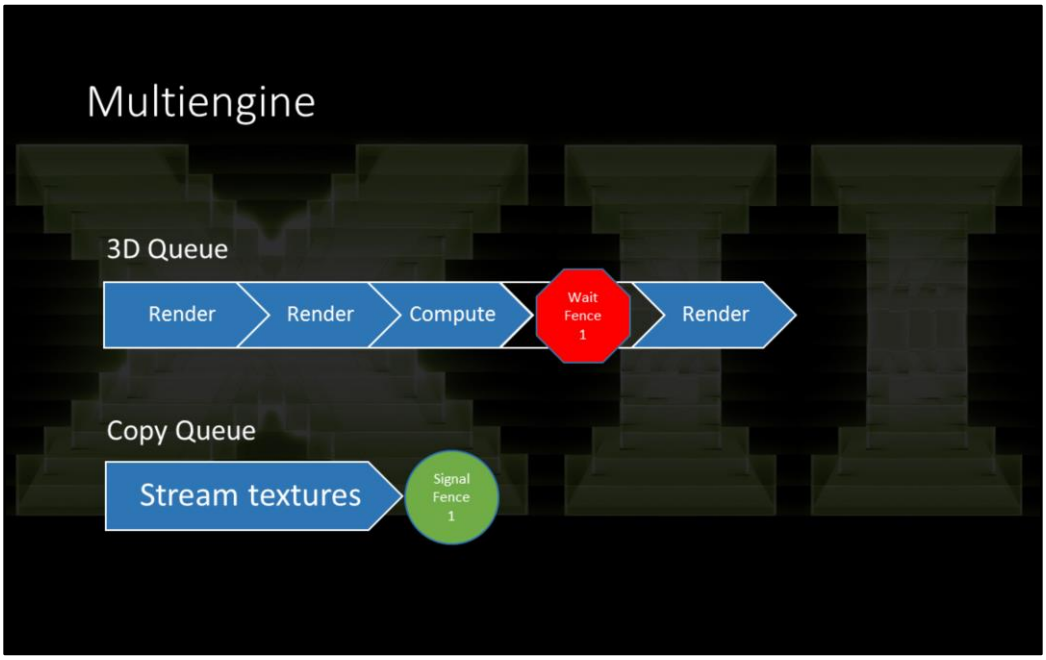
But in reality, there are other components on there like the encoders and decoder and the display scan-out engines, etc.
DX12 enables this

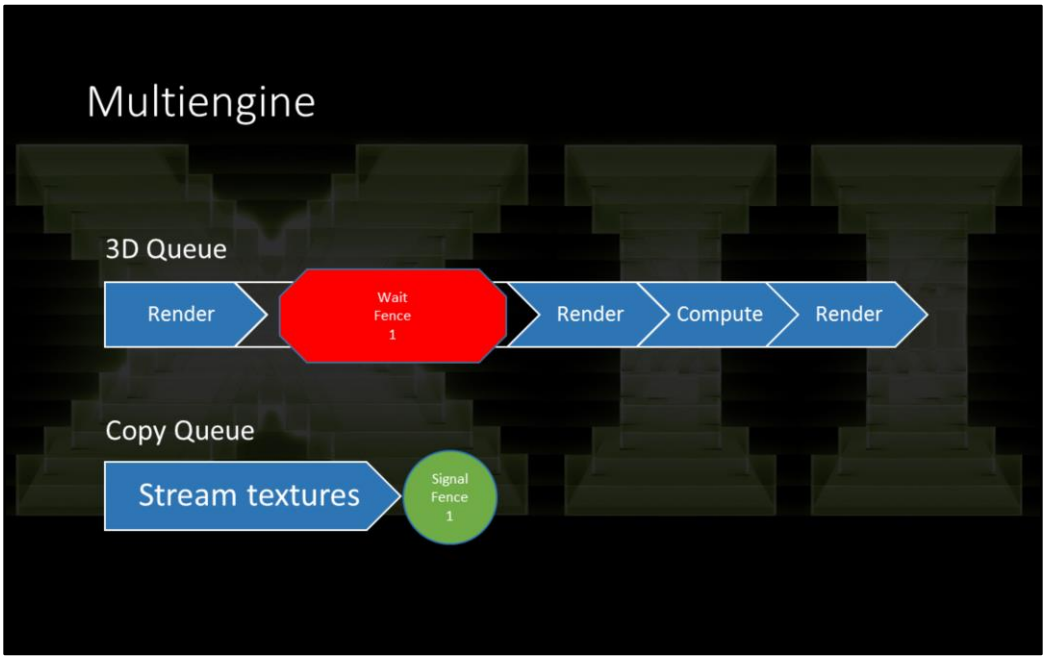
Execution Model

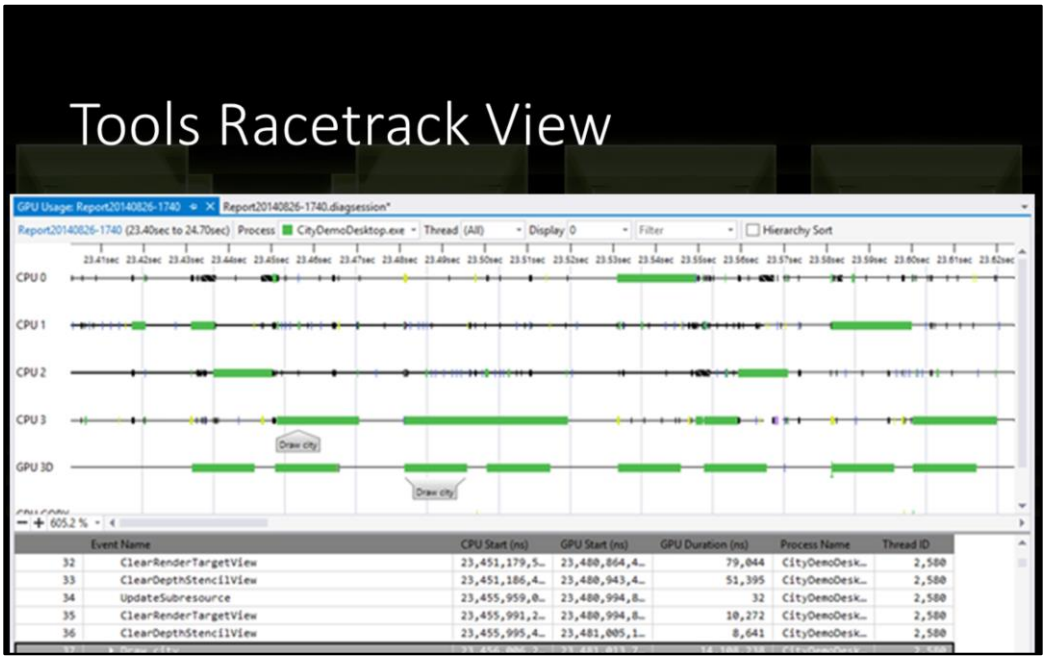
- All of these are just cores aka ‘engines’
  - They can be invoked asynchronously
- Model is a queue per core for independent async operation
- A queue guarantees serial order of execution on a single engine
- Can specify priorities between queues
  - Enables background processing in ‘idle’ clock cycles
- And also implement semaphores across queues



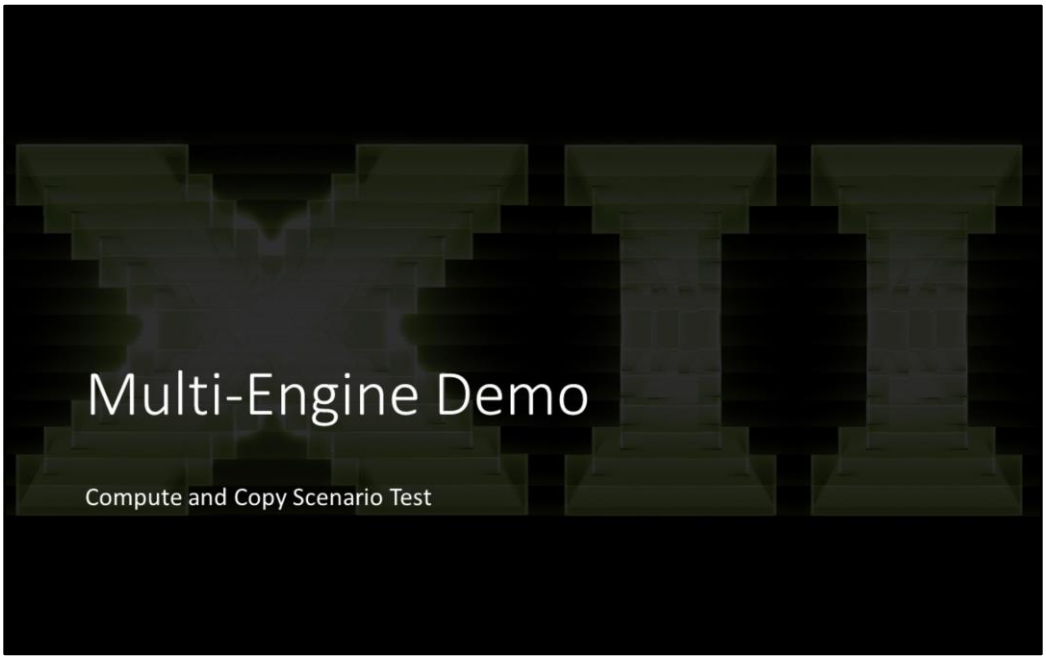
Extract all the parallelism out of the hardware that's available
Why do we have these nested? Because that's how the hardware actually works:
Really the 3D engine can do anything. It can do compute tasks and also the highest bandwidth copy tasks.
A compute queue is just using the 3D engine when you know you can power down the graphics-specific portions of that core.
A copy queue can be done on a separate blitter core aka DMA engine.


Multiengine
3D Queue
Render
Render
Compute
Wait Fence 1
Render
Copy Queue
Stream textures
Signal Fence 1

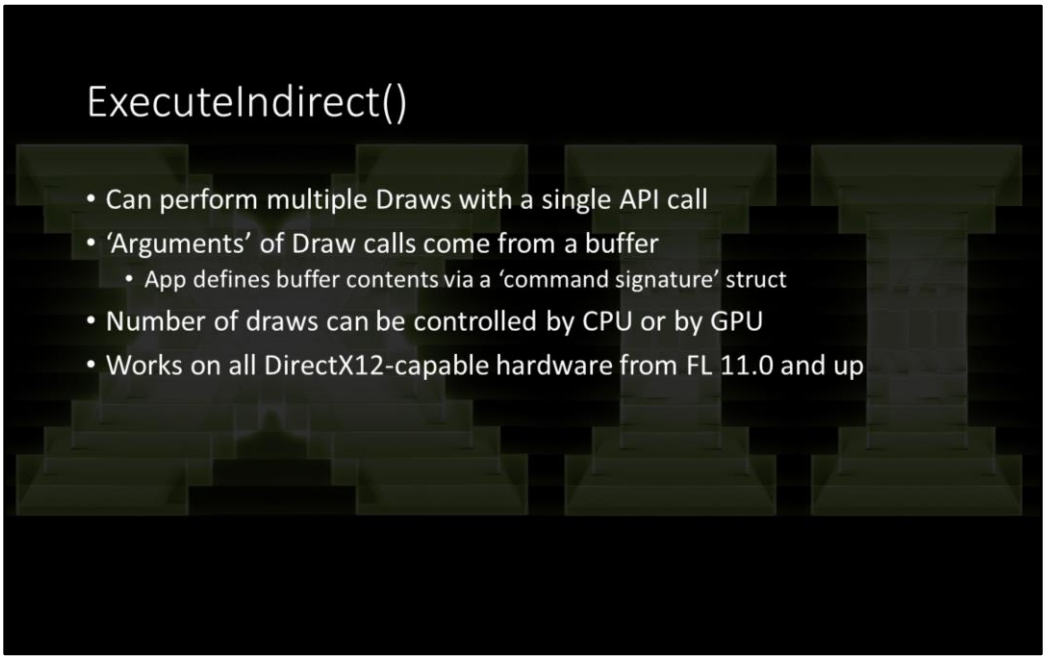

Multiengine
3D Queue
Render
Wait
Fence
1
Render
Compute
Render
Copy Queue
Stream textures
Signal
Fence
1



This shows how the model is even expressed in the tools.
You can see that the GPU engines (3D, and Copy) are peers to the CPU cores in the model.



Mandlebrot
This laptop gets 37% speedup by pushing the copy task on to a separate queue.
which means a cheaper core is now the one blocked on PCIe bandwidth, and the ALUs can go full rate.



Much like everything else in DirectX, we've abstracted the nuances of all the hardware and enabled this feature on every 12 GPU

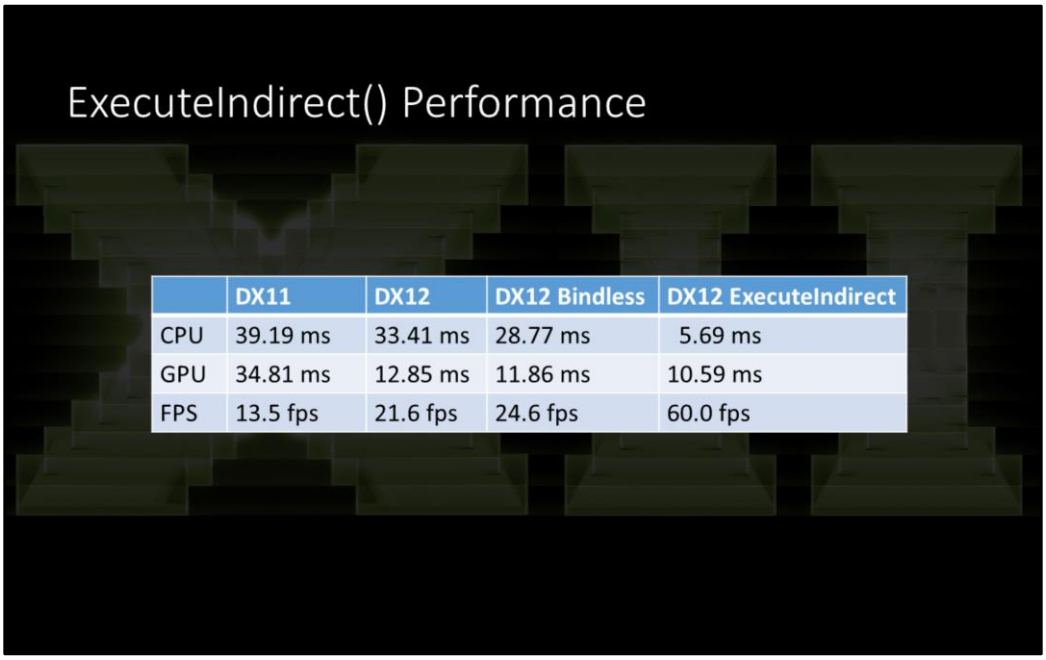
ExecuteIndirect Command Signature

- Operations performed by ExecuteIndirect described by a ‘command signature’
- Describes the layout of the argument buffer and the set of commands
- Operations include:
  - Set vertex or index buffer
  - Change root constants
  - Set root resource views (SRV, UAV, CBV)
  - Draw, DrawIndexed, or Dispatch

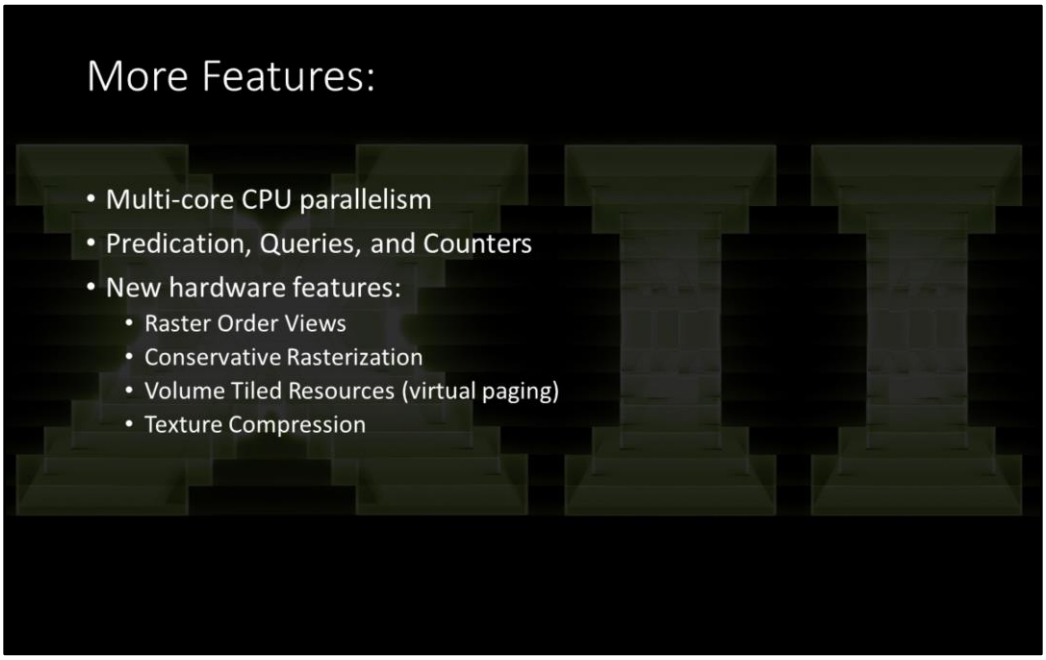
ExecuteIndirect versus Draw Loop

```
for (UINT drawIdx = drawStart; drawIdx < drawEnd;
++drawIdx)
{
    // Set bindings
    cmdLst->SetGraphicsRootConstantBufferView(RT_CBV,
constantsPointer);
    constantsPointer += sizeof(DrawConstantBuffer);

    auto textureSRV =
textureStartSRV.MakeOffsetted(staticData-
>textureIndex, handleIncrementSize);
    cmdLst->SetGraphicsRootDescriptorTable(RT_SRV,
textureSRV);

    cmdLst->DrawIndexedInstanced(dynamicData-
>indexCount, 1, dynamicData->indexStart, staticData-
>vertexStart, 0);
}
```

```
mCmdLst->SetGraphicsRootDescriptorTable(RT_SRV,
mTextureStart);

mCmdLst->ExecuteIndirect(mCommandSignature,
settings.numAsteroids, frame->mIndirectArgBuffer-
>Heap(), 0, nullptr, 0);
```



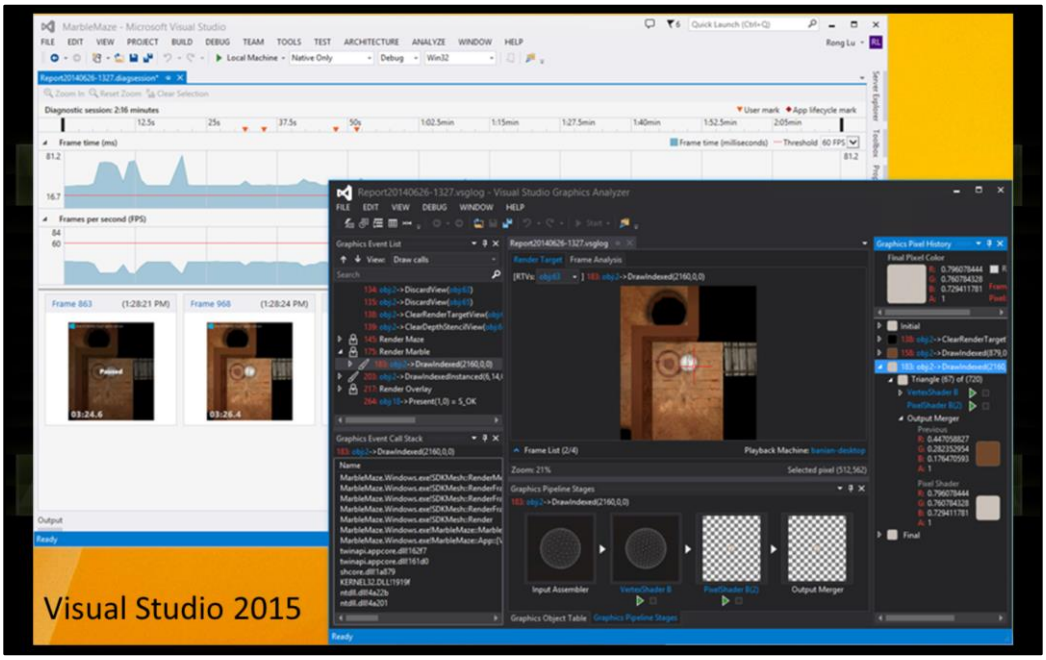
Total CPU time



Most New Hardware Features are more interesting for graphics-related workloads
ROVs enable spatial random access, but temporal serialization.
Useful when starting from a graphcis tasks and writing to a general datastructure (UAB)
E.g. for when you sort input triangles beforehand and want to retain that, or other algorithms where order matters.

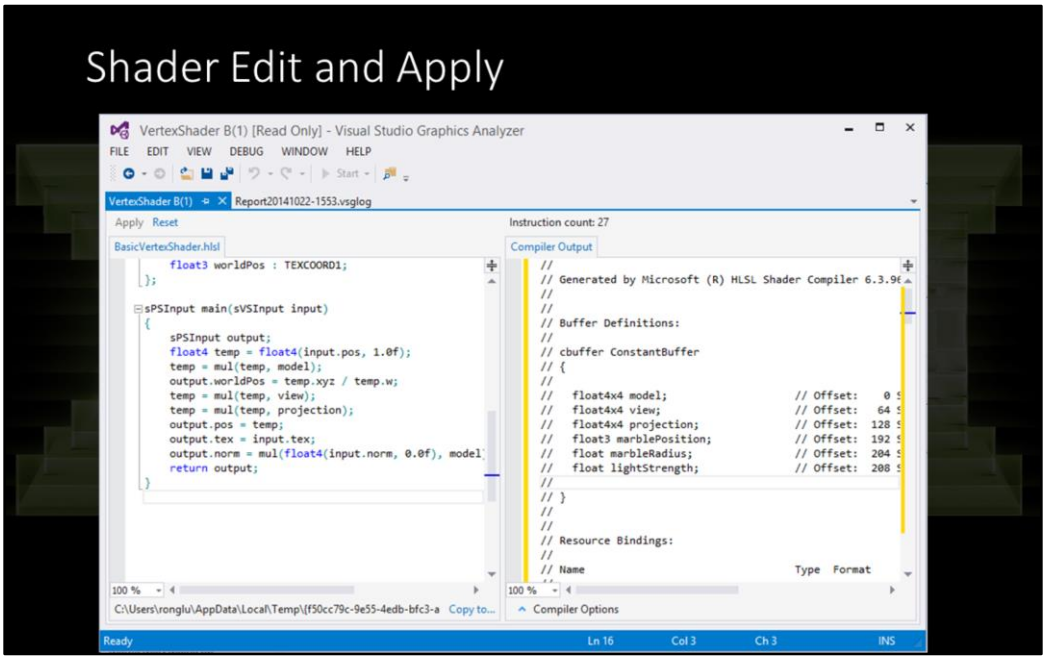
Tools

- SDK layer can be enabled for detailed validation
- Tools are now built in concert with the API
  - Capture/Playback
  - Timing Analysis
  - Visualization of intermediate results
- New instrumentation has been added to drivers
  - Detailed stats on internal registers

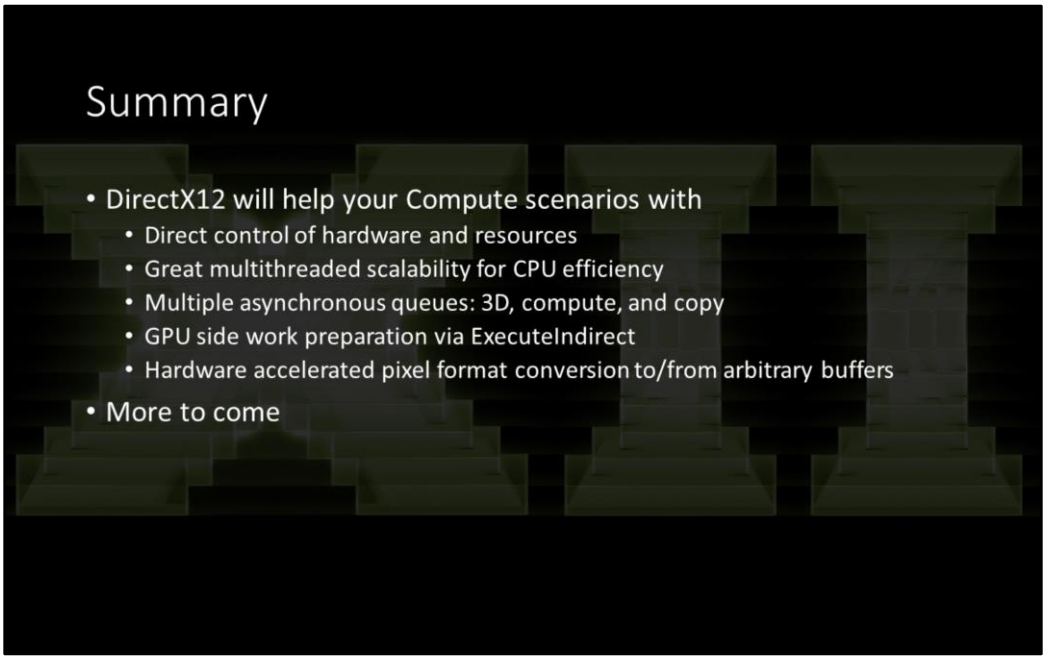


VS 2015
Unified CPU, GPU, System profiling and debugging tool for the Universal App Platform and full breadth of Windows devices

Shader Edit and Apply



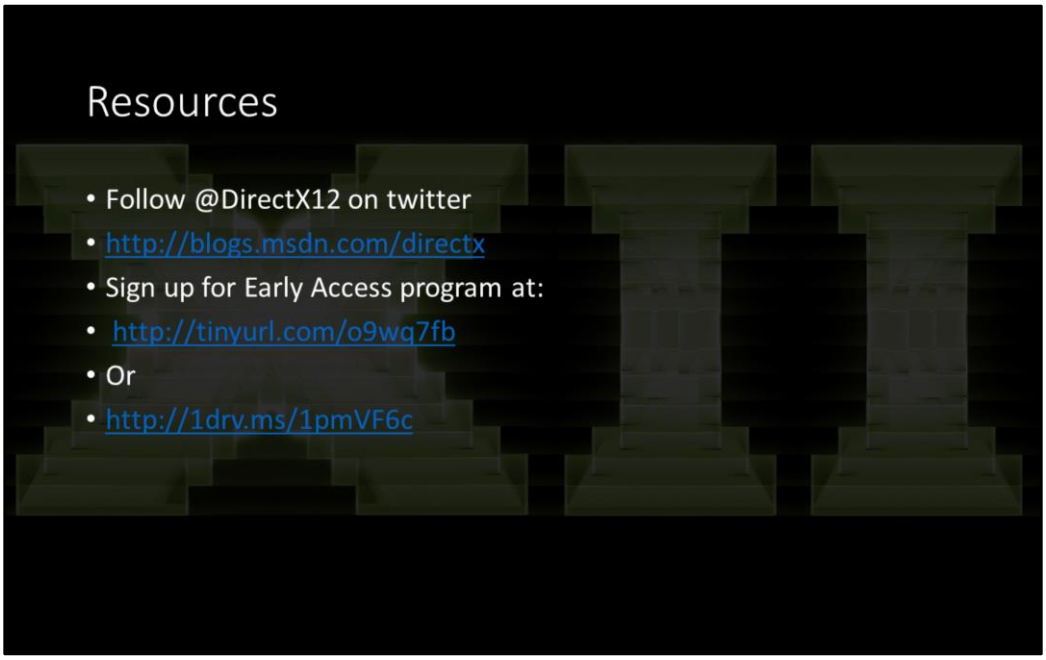
Side by side windows for HLSL source code and shader compiler output
Edit shader code and apply changes to the log file to view impacts

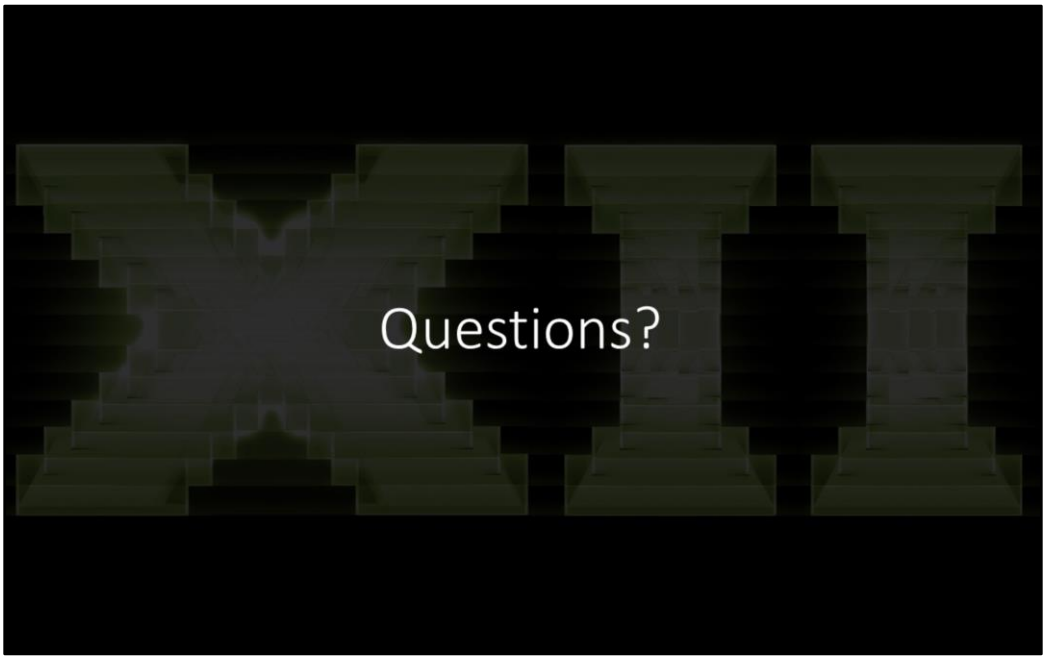


GPGPU was not the main focus of DX12, yet there are several that massively improve the DirectCompute capabilities and performance
Support for multi-GPU, and for VR/Stereo.

DirectX12 on http://channel9.msdn.com

- http://channel9.msdn.com/events/GDC/GDC-2015/Advanced-DirectX12-Graphics-and-Performance
- http://channel9.msdn.com/events/GDC/GDC-2015/Better-Power-Better-Performance-Your-Game-on-DirectX12
- http://channel9.msdn.com/events/GDC/GDC-2015/Solve-the-Tough-Graphics-Problems-with-your-Game-Using-DirectX-Tools
- Search terms: “DirectX GDC”


Resources
• Follow @DirectX12 on twitter
• http://blogs.msdn.com/directx
• Sign up for Early Access program at:
• http://tinyurl.com/o9wq7fb
• Or
• http://1drv.ms/1pmVF6c


Questions?

Compute Glossary of Terms (WIP)

DCompute	CUDA	OpenCL	Vector CPU	
shader	kernel	kernel	routine	
thread	thread	work item	SIMD lane	
wavefront	warp	-	Thread	
threadgroup	threadblock	work group	-	
CU/EU/SIMD	SMP	ComputeUnit	core	
Execution	Grid	N-D range	task?	
PSO	samplers?	??		all register state plus compiled GPU code
Cmdlist	?	cmdQueue		
constant	constant	constant		R/O across entire kernel (‘uniform’ in OpenGL)
groupshared	shared	local		R/W shared across thread/work group
registers	local	private		local to given thread
resource	global	global		R/W across entire machine (Buffers)
sampler	sampler	sampler		
tex resource	tex res	image object		
buffer res	buf res	buffer object		
fence		marker		