



DirectCompute in DirectX12: Innovations from the Game Space

Chas. Boyd

Principle PM Microsoft OSG Graphics

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..

Outline

- Advances in DirectX12
- Game Tricks
- New Architecture:
 - Heaps, resource indexing, indirect execution, asynchronous queues
- Language
- Tools

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

DirectX

- Direct3D API for GPUs
- Historically mostly focused at games
- Added DirectCompute mode aka 'Compute Shaders' for DirectX11
 - Windows 7 in 2009
- Goal: expose capabilities of GPUs for more general-purpose processing
- Compute shaders are separate tasks from graphics tasks
 - But use same device/context and memory resources as graphics/rendering shaders

Compute shaders are not just another stage in the pipeline

Use Cases Driving DirectCompute Design

- General problem:
 - Irregular data structure → interim structure → data-parallel layout
 - E.g. culling, tessellation, procedural generation
 - Unpredictable data and work loads
- Interop between CPU and GPU at high frequency
 - No advance knowledge of what data will be needed
 - Constrained by low latency limits
- Most analogous technical scenarios:
 - Interactive simulations: “steering” -less predictable
 - Data-parallel and conventional portions finely intermingled

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.

Game Technique: Managing Branching -Offline

- Games often pre-generate large numbers of variants of a kernel
- Specialized for flow control
- Indexed based on bit pattern for flow control state
- Often large scale:
 - 10,000+ compiled kernels
 - Several GB of compiled GPU binary
 - Sophisticated compression schemes used for distribution

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

Game Technique: Managing Branching -Runtime

- Dice up the problem space into elements (reasonable chunks)
 - Typically approximately a workgroup (512 threads)
 - Execute algorithm up to point of branch
 - Export data chunks
 - Re-sort chunks based on branch values
 - Process elements separately
 - Re-merge elements
-
- We see this commonly for small integer switch statements, etc.
 - Such as for lighting calculations

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.

- 1995 DirectX 1 DirectDraw, hardware blit and page flip
- 1996 DirectX 2 Direct3D, software render, execute buffers
- 1996 DirectX 3 Hardware-accelerated rasterization
- 1997 DirectX 5 DrawPrimitive, dual-texture, 1-bit 'shader'
- 1998 DirectX 6 Multi-texture blending, DXTC compression, bump mapping
- 1999 DirectX 7 Hardware vertex processing transformation and lighting.
- 2000 DirectX 8 First Programmable shaders
- 2001 DirectX 8.1 More instructions
- 2002 DirectX 9 High Level Shading Language, shaders of 32 instructions, HDR
- 2003 DirectX 9.0c float pixels, HLSL with 1000s of instructions per shader
- 2006 DirectX 10 Caps-free, geometry shaders,
- 2009 DirectX 11 Tessellation, DirectCompute
- 2012 DirectX 11.1 Performance and ARM CPU support
- 2013 DirectX 11.2 Tiled resources (aka megatexture)
- 2015 DirectX 12 Performance: Multithreading, Bindless, Multi-Engine,

Evolution of DirectX releases and key features of each version

DirectX12

- DirectX12 is a ground-up API redesign to improve performance
- Delivers a lower-level API abstraction
- A more direct mapping to hardware:
 - More predictable and consistent behavior due to elimination of sw layers
 - More capability for app, but also more responsibility
- App now handles work that used to be in the driver
 - Queueing, scheduling, synchronization
- Actually reduces 'size' of API to learn
- DX12 is a superset of DirectX11 feature-wise.

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.

Innovations in DirectX12

- Persistent memory ranges aka “Resource Heaps”
- Indexable resources “bindless”
- Root signatures for arguments to GPU
- Hardware image format conversion
- Default asynchronous resource access
- Asynchronous GPU task dispatch
- ExecuteIndirect
- Tools are integral part of the release

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 thread
- 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

Memory Heaps

- Persistent memory range allocation
- Resources are easily identified within a heap
 - Buffers and Images are subranges independent of underlying implementation
 - Allows rapid 'repurposing' of memory within the heap
 - Hoists the OS and driver work out to a Heap Creation phase
- Heap Types:
 - Default Heaps: Usage within GPU memory
 - Upload Heaps: CPU→GPU transfer
 - Readback Heaps: GPU→CPU transfer
- Can reserve some memory from OS for quality of service guarantees

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

Root Signature & Root Arguments

- Root Signature defines the number of arguments and their types
 - Descriptor Tables
 - Descriptors
 - Constants
- Budget of “64 DWORDs” to spend
 - Descriptor Table: 1 DWORD
 - Descriptor: 2 DWORDs
 - Constant: 1 DWORD * Number of Channels
- Performance improves with fewer DWORDs used

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;

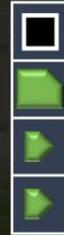
    union
    {
        D3D12_DESCRIPTOR_TABLE_LAYOUT DescriptorTable;
        D3D12_ROOT_CONSTANTS Constants;
        D3D12_ROOT_DESCRIPTOR Descriptor;
    }
    ...
}
```

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);
```

Hardware Image Format Conversion

- For scenarios when your compute task reads or writes image data
 - Images are a common source of data-parallel problems
- Key D3D_FORMAT_xxxx are supported in DirectX12:
 - R32G32B32A32_FLOAT, _UINT, _SINT
 - R16G16B16A16_FLOAT, _UINT, _SINT
 - R16G16B16A16_UINT, _SINT
 - R8G8B8A8_UNORM, _UINT, _SINT
 - R32_FLOAT, _UINT, _SINT
 - R16_FLOAT, _UINT, _SINT
 - R8_UNORM, _UINT, _SINT

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.

Dispatch

WaitForIdle

Dispatch

WaitForIdle

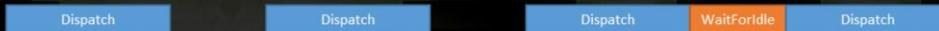
Dispatch

WaitForIdle

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

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

- Second, the ResourceBarrier API allows application to perform transitions over a period of time.
- App specifies starting/destination states at 'begin' and 'end' ResourceBarrier calls. Promises not to use resource while in transition.
- Driver can use this information to eliminate redundant pipeline stalls, cache flushes

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

UAV Barriers

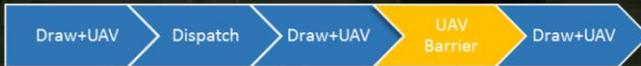
- In D3D11 all UAV accesses in 1 Draw/Dispatch must complete before any UAV accesses in a subsequent Draw/Dispatch
- This results in idle GPU shader cores for small Draw/Dispatch
- In D3D12 UAV accesses in multiple Draw/Dispatch are truly unordered, applications must use an explicit barrier to enforce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

UAV Barriers

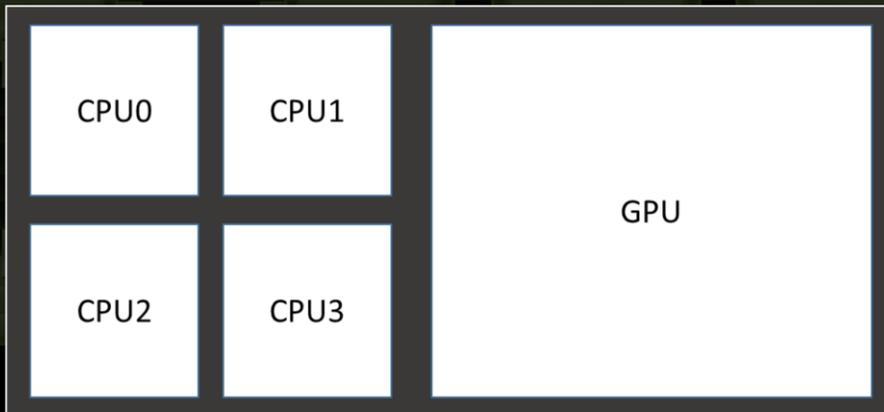
Direct3D 11



Direct3D 12

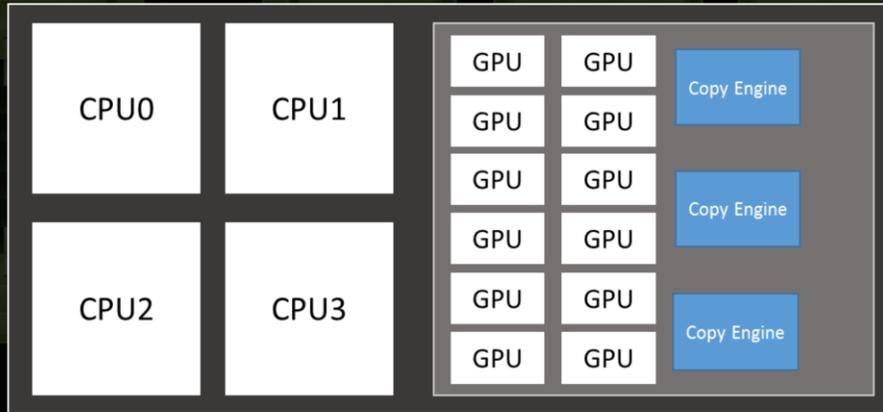


Asynchronous Task Dispatch



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

Asynchronous Execution



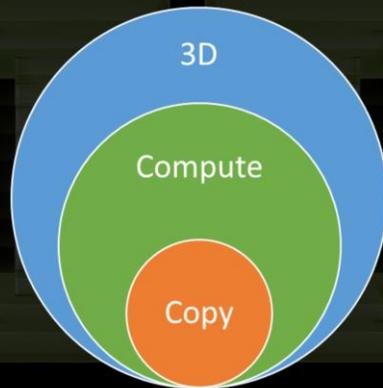
But in reality, there are other components on there like the encoders and decoder and the display scan-out engines, etc.
DX12 enables this

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

Multi-Engine

- Expose multiple parallel queues as explicit API objects
- Queue Types: 3D, Compute, Copy
- Prioritized queues enable new scenarios
 - High priority, latency sensitive workloads
 - Low priority background tasks



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.

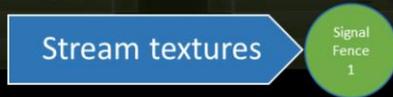
A copy queue can be done on a separate blitter core aka DMA engine.

Multiengine

3D Queue



Copy Queue

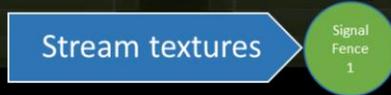


Multiengine

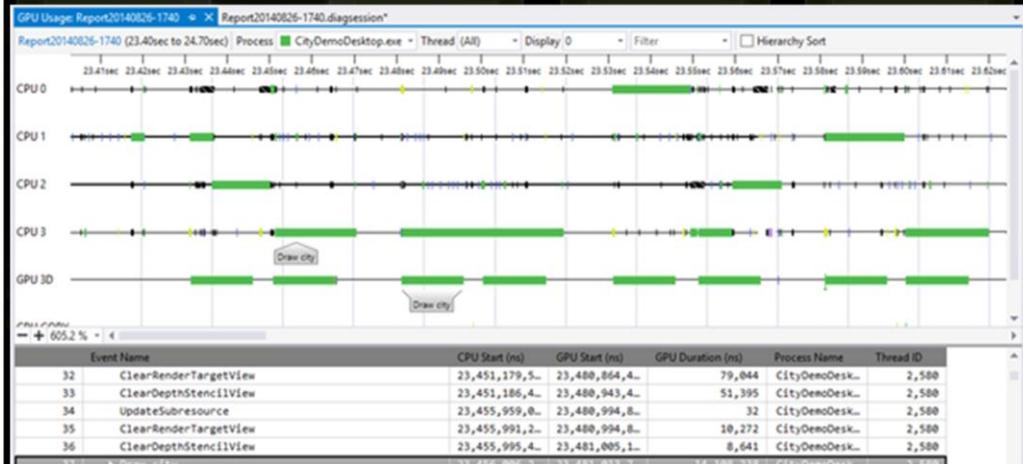
3D Queue



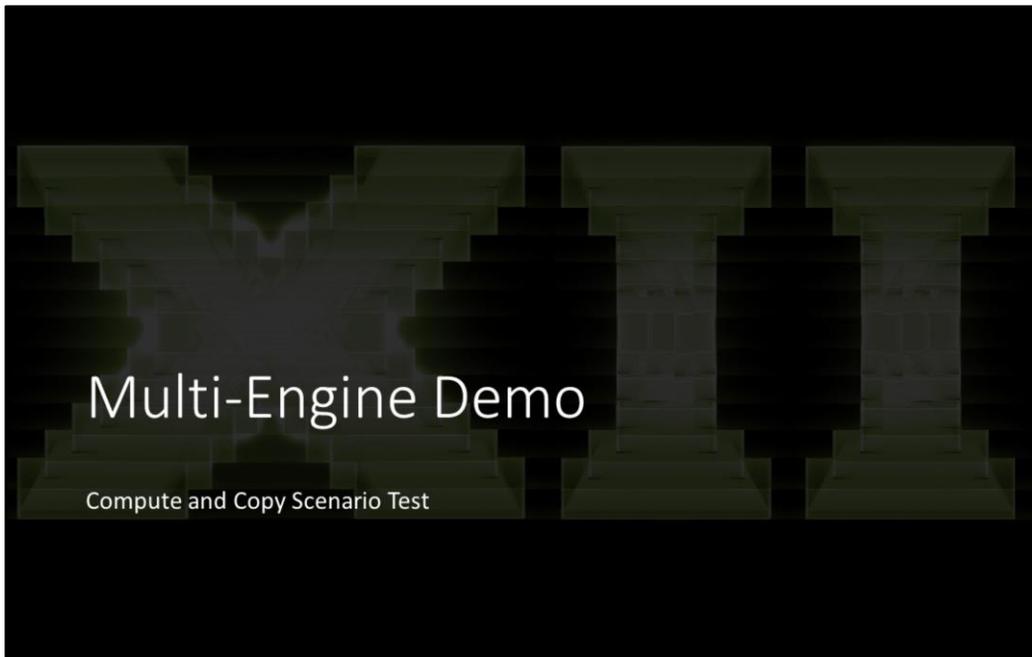
Copy Queue



Tools Racetrack View



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.



Mandelbrot

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.

ExecuteIndirect()

- Can perform multiple Draws with a single API call
- 'Arguments' of Draw calls come from a buffer
 - App defines buffer contents via a 'command signature' struct
- Number of draws can be controlled by CPU or by GPU
- Works on all DirectX12-capable hardware from FL 11.0 and up

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
    mCmdLst->SetGraphicsRootConstantBufferView(RT_CBV,
        constantsPointer);
    constantsPointer += sizeof(DrawConstantBuffer);

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

    mCmdLst->DrawIndexedInstanced(dynamicData-
        >indexCount, 1, dynamicData->indexStart, staticData-
        >vertexStart, 6);
}
```

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

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

ExecuteIndirect() Performance

	DX11	DX12	DX12 Bindless	DX12 ExecuteIndirect
CPU	39.19 ms	33.41 ms	28.77 ms	5.69 ms
GPU	34.81 ms	12.85 ms	11.86 ms	10.59 ms
FPS	13.5 fps	21.6 fps	24.6 fps	60.0 fps

Total CPU time

More Features:

- Multi-core CPU parallelism
- Predication, Queries, and Counters
- New hardware features:
 - Raster Order Views
 - Conservative Rasterization
 - Volume Tiled Resources (virtual paging)
 - Texture Compression

Most New Hardware Features are more interesting for graphics-related workloads

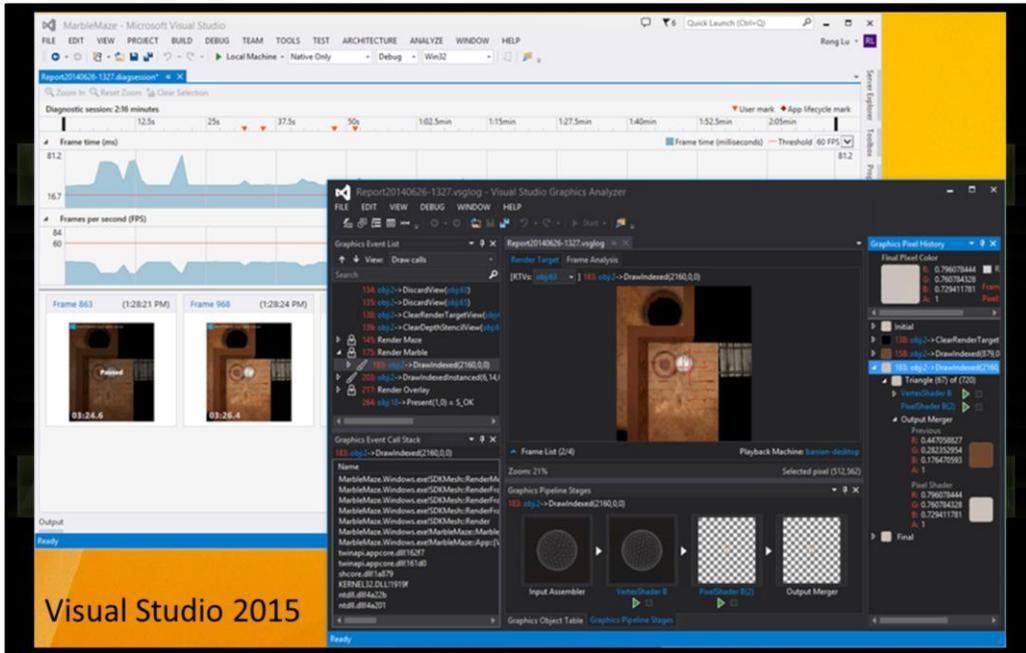
ROVs enable spatial random access, but temporal serialization.

Useful when starting from a graphics 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

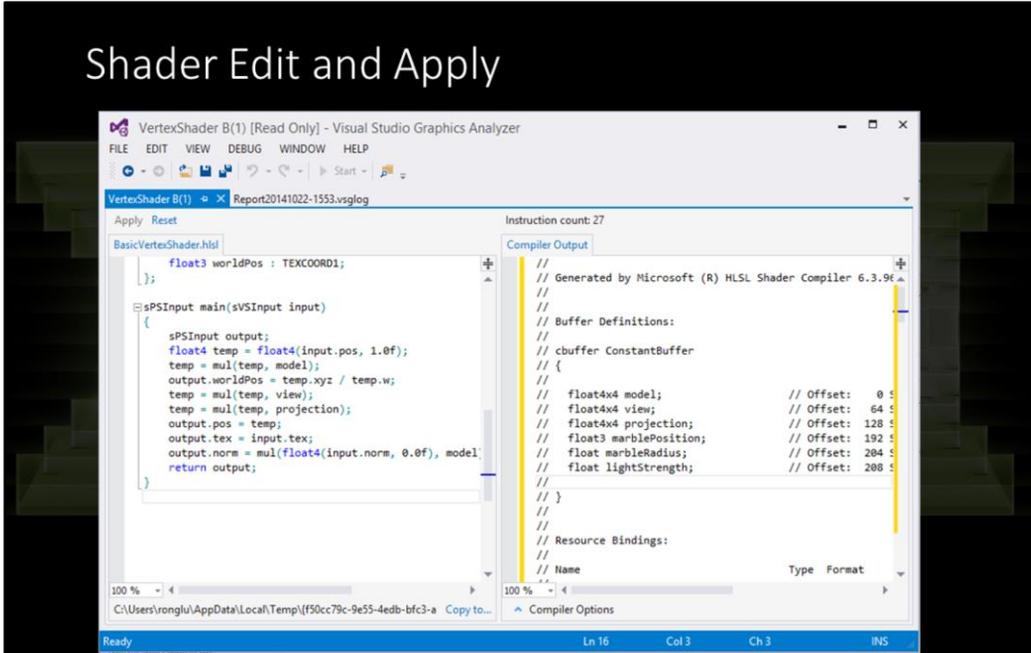
- SDK layer can be enabled for detailed validation
- Tools are now built in concert with the API
 - Capture/Playback
 - Timing Analysis
 - Visualization of intermediate results
- New instrumentation has been added to drivers
 - Detailed stats on internal registers



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

Summary

- DirectX12 will help your Compute scenarios with
 - Direct control of hardware and resources
 - Great multithreaded scalability for CPU efficiency
 - Multiple asynchronous queues: 3D, compute, and copy
 - GPU side work preparation via ExecuteIndirect
 - Hardware accelerated pixel format conversion to/from arbitrary buffers
- More to come

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	