Game Developers Conference®

February 28 - March 4, 2011
Moscone Center, San Francisco

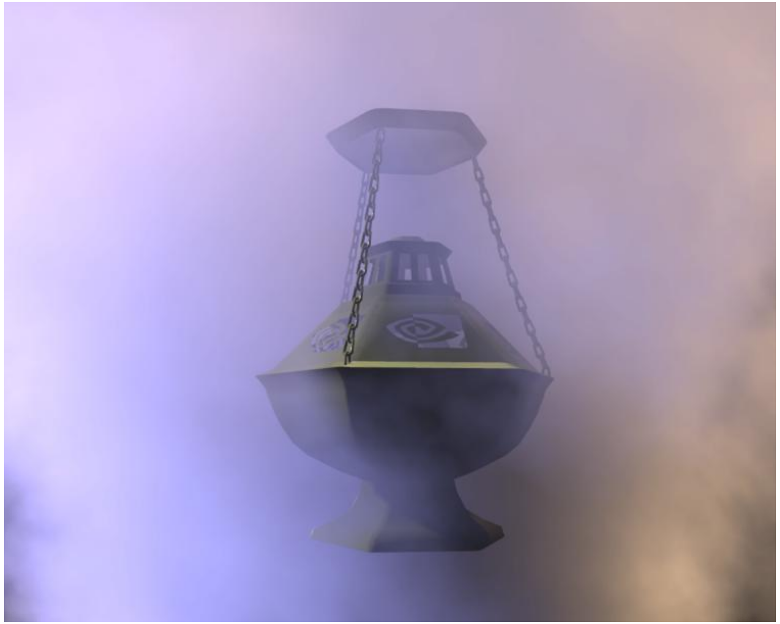
www.GDConf.com

DX11 Performance Gems

(Or: "DX11 – unpacking the box")

Jon Jansen - Developer Technology Engineer, NVIDIA

Topics Covered



Case study: Opacity Mapping

– Using tessellation to accelerate lighting effects
– Accelerating up-sampling with GatherRed()
– Playing nice with AA using SV_SampleIndex
– Read-only depth for soft particles

Topics Covered (cont)

Deferred Contexts

– How DX11 can help you pump the API harder than you thought possible

– Much viscera - very gory!

Not Covered

‘Aesthetic’ tessellation

DirectCompute

!!! Great talks on these topics coming up !!!

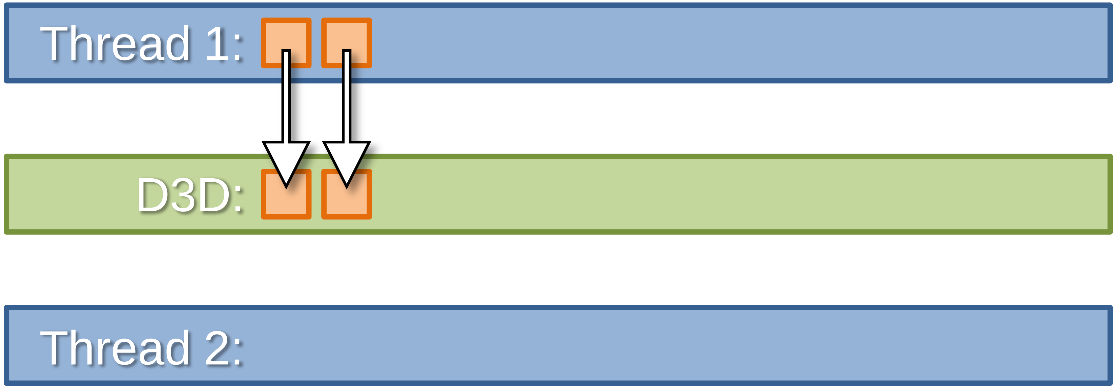
DEFERRED CONTEXTS

Deferred Contexts

- What are your options when your API submission thread is a bottleneck?
- What if submission could be done on multiple threads, to take advantage of multi-core?
  – This is what Deferred Contexts solve in DX11

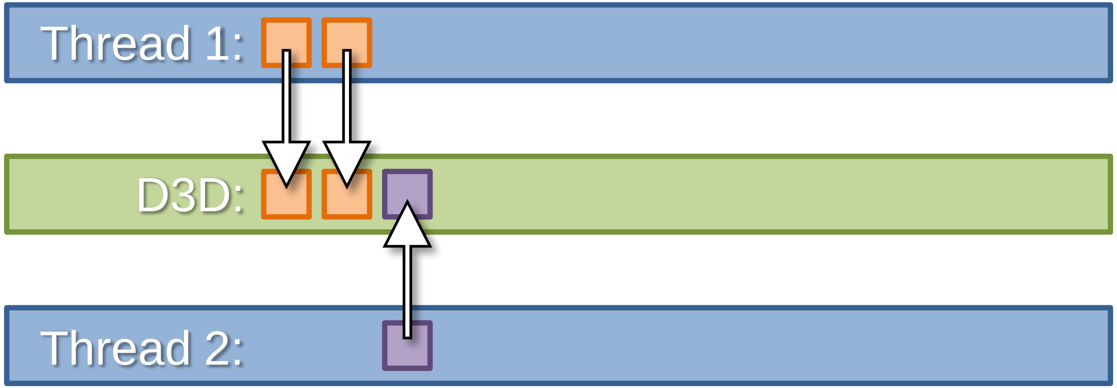
Deferred Contexts

- So why not just submit directly to an API from multiple threads and be done?



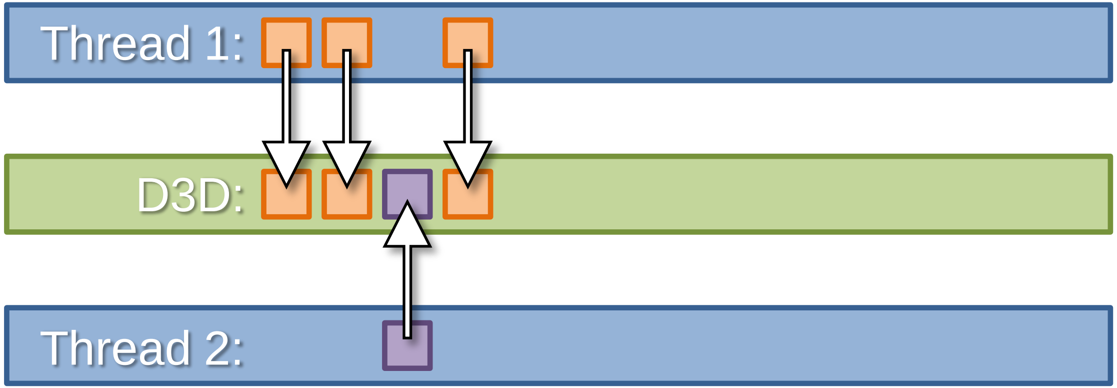
Deferred Contexts

- So why not just submit directly to an API from multiple threads and be done?



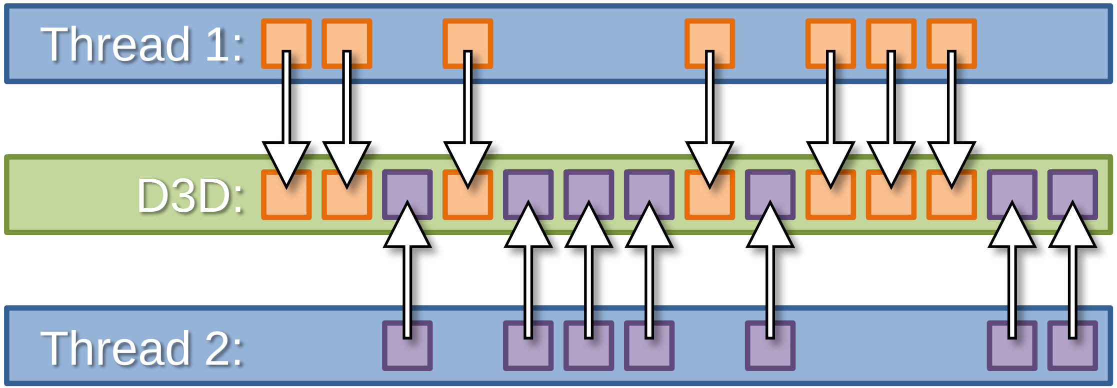
Deferred Contexts

- So why not just submit directly to an API from multiple threads and be done?



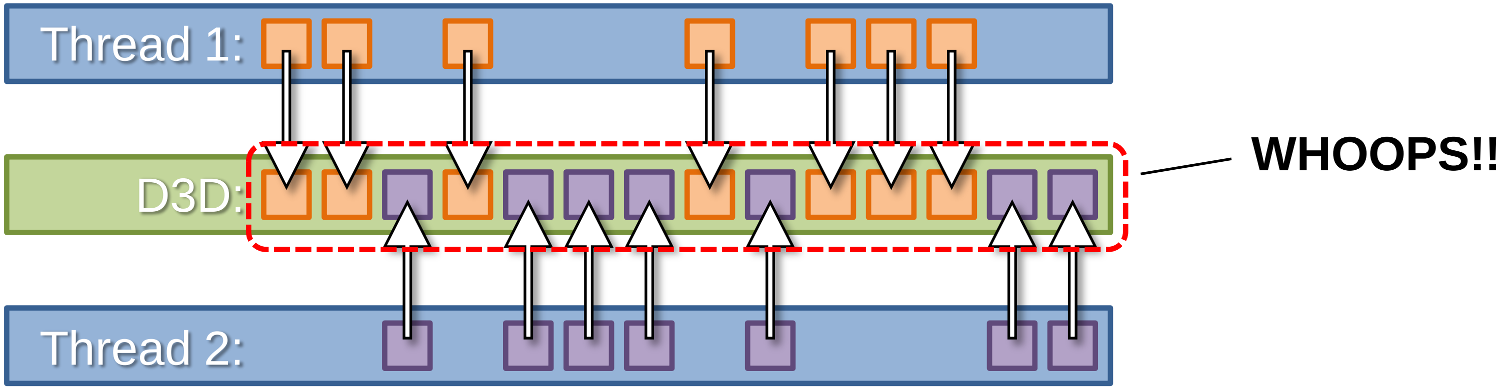
Deferred Contexts

- So why not just submit directly to an API from multiple threads and be done?



Deferred Contexts

- So why not just submit directly to an API from multiple threads and be done?



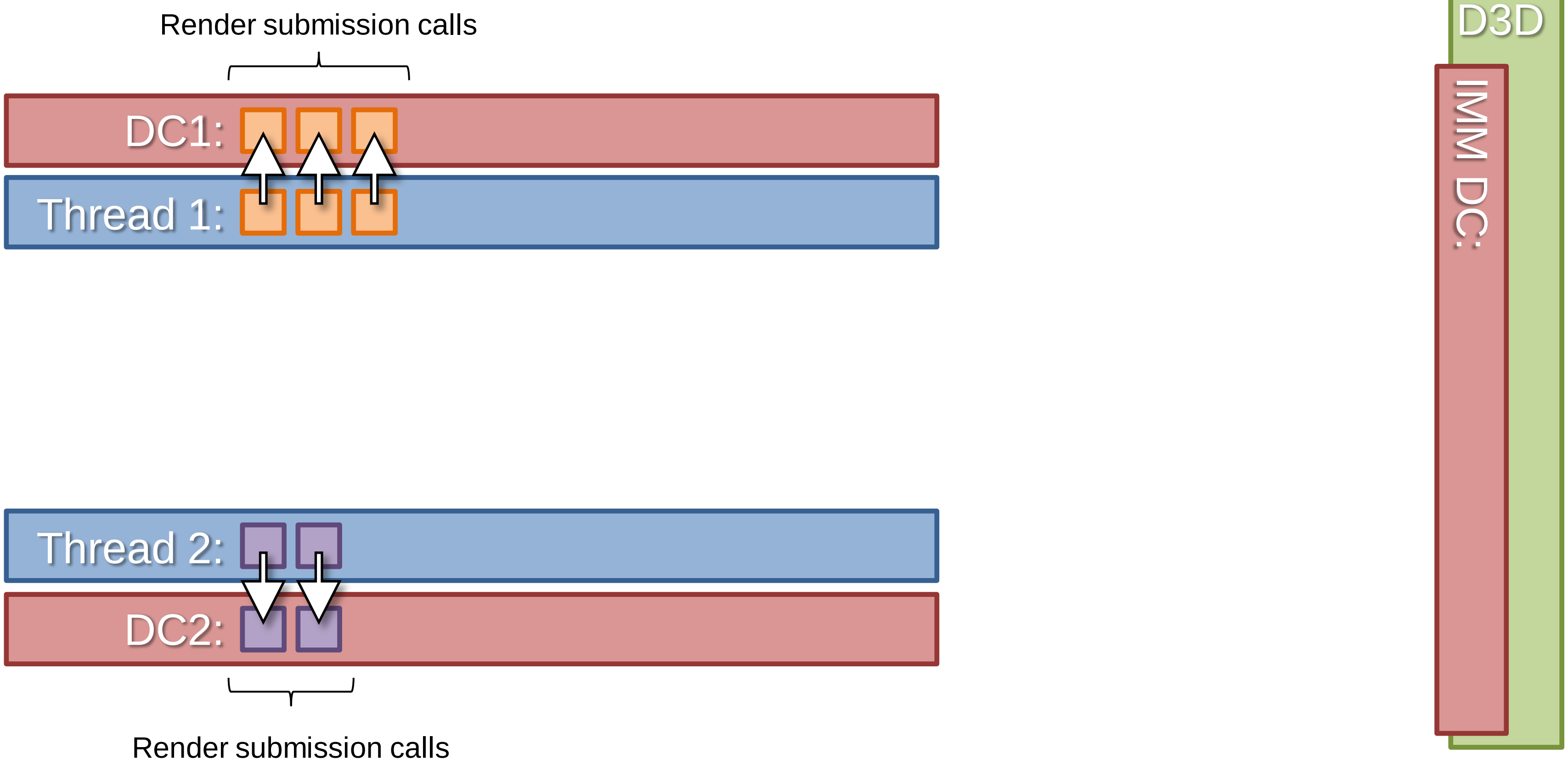
Deferred Contexts

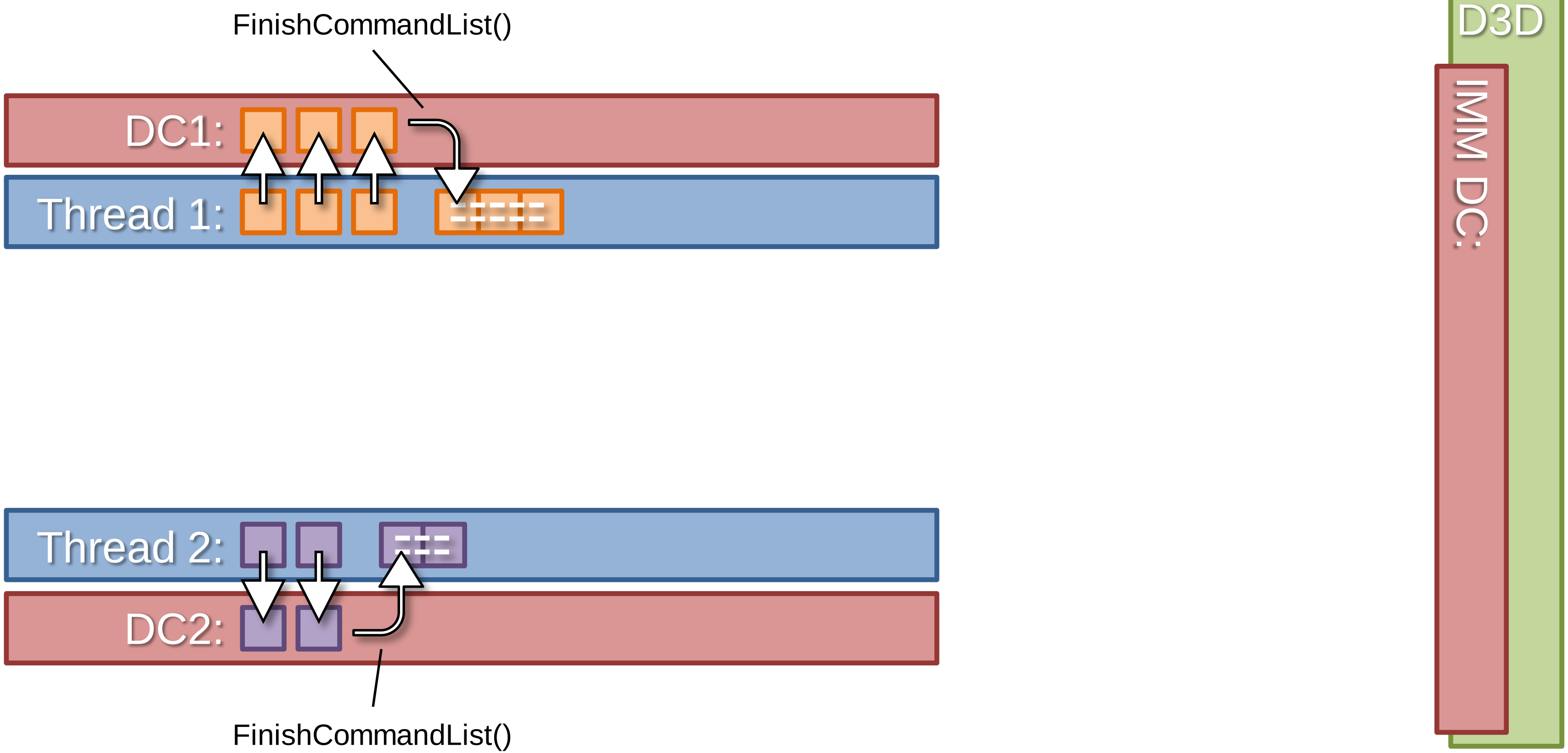
- A Deferred Context is a device-like interface for building command-lists

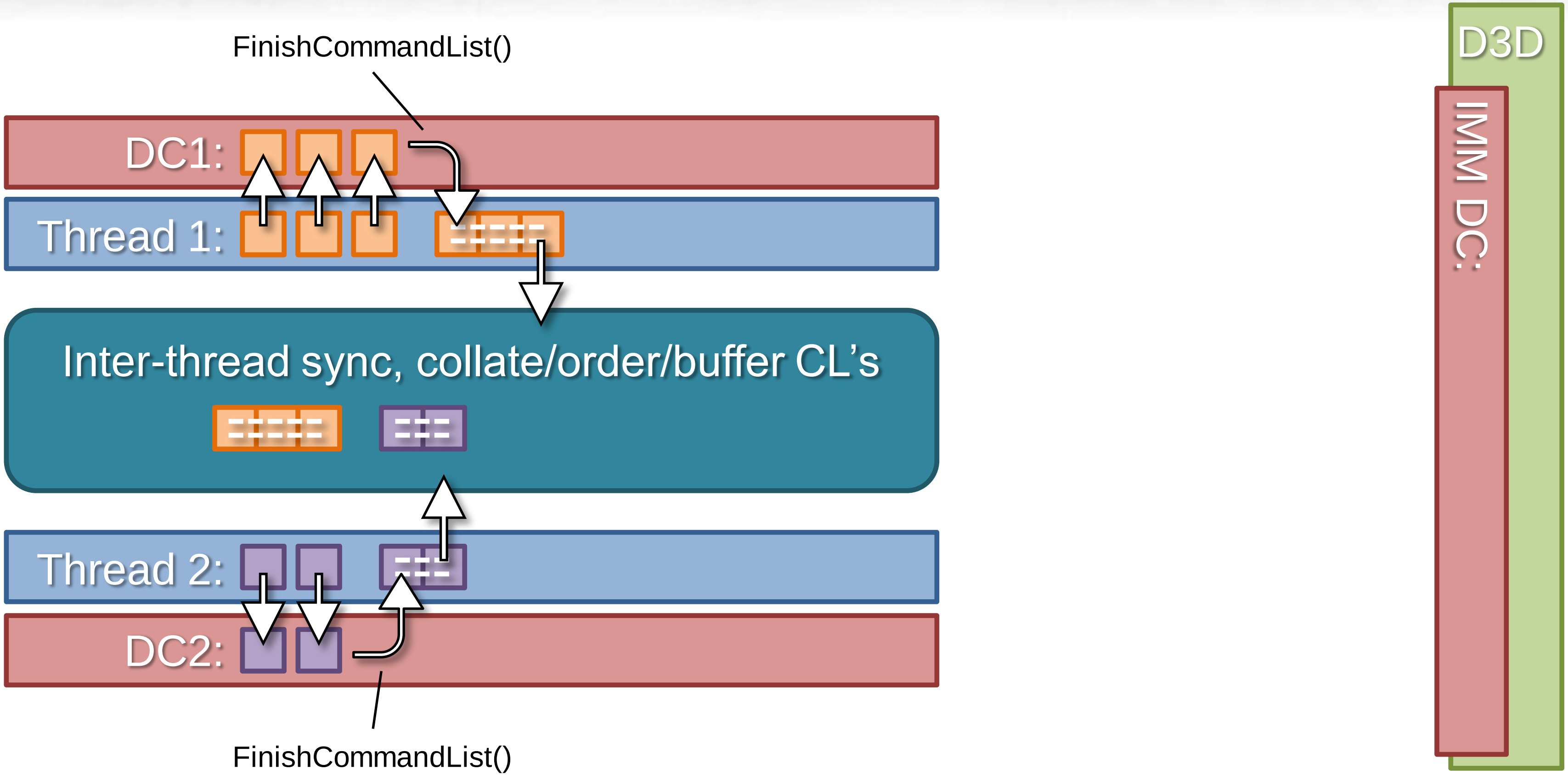
```
// Creation is very straightforward
ID3D11DeviceContext* pDC = NULL;
hr = pD3DDevice->CreateDeferredContext(0,&pDC);
```

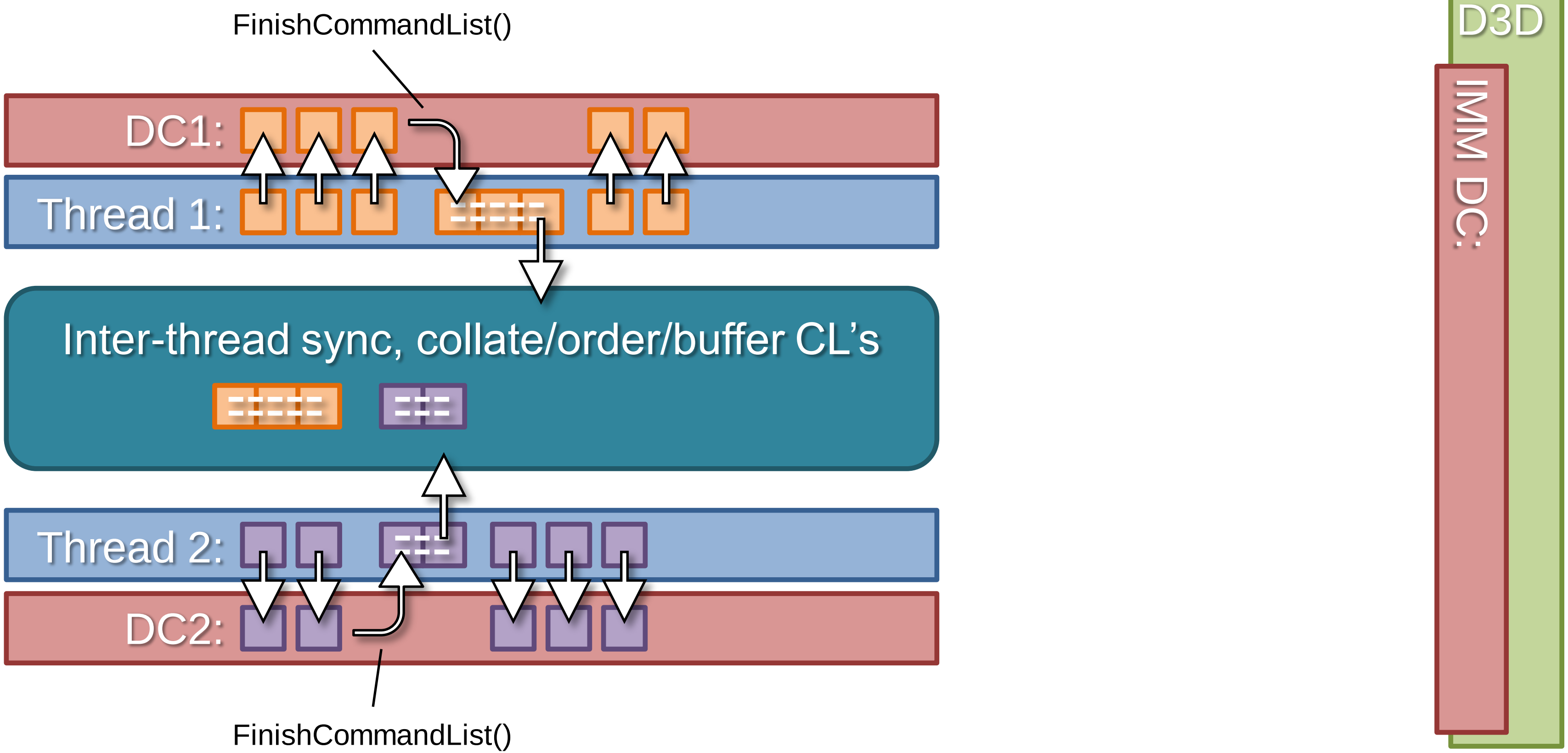
Deferred Contexts

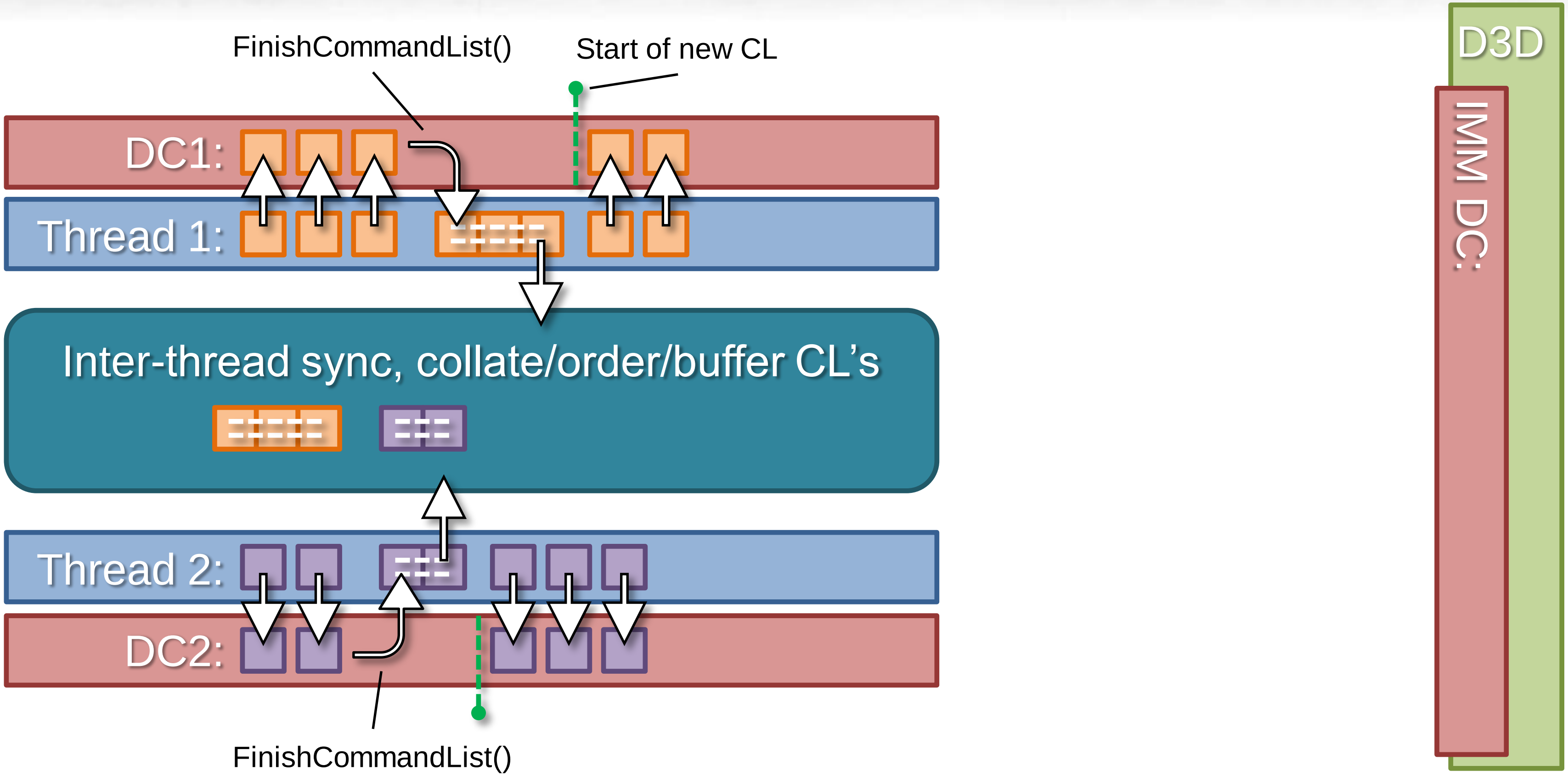
- DX11 uses the same ID3D11DeviceContext interface for ‘immediate’ API calls
- Immediate context is the <u>only</u> way to finally submit work to the GPU
    - Access it via ID3D11Device::GetImmediateContext()
    - ID3D11Device has <u>no</u> submission API


Render submission calls
DC1:
Thread 1:
Thread 2:
DC2:
Render submission calls
D3D
IMM DC:

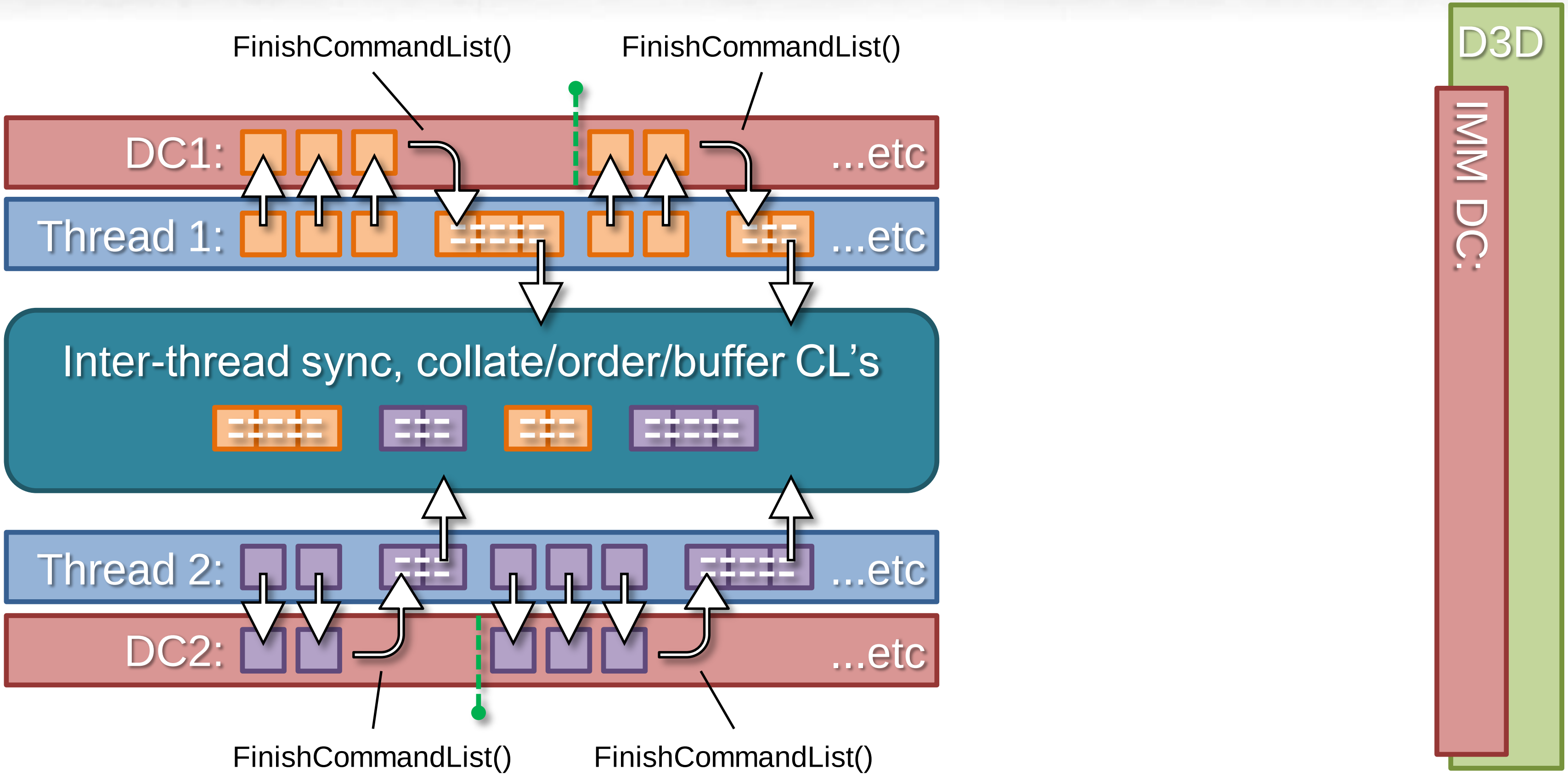

FinishCommandList()
DC1:
Thread 1:
Thread 2:
DC2:
FinishCommandList()
D3D
IMM DC:

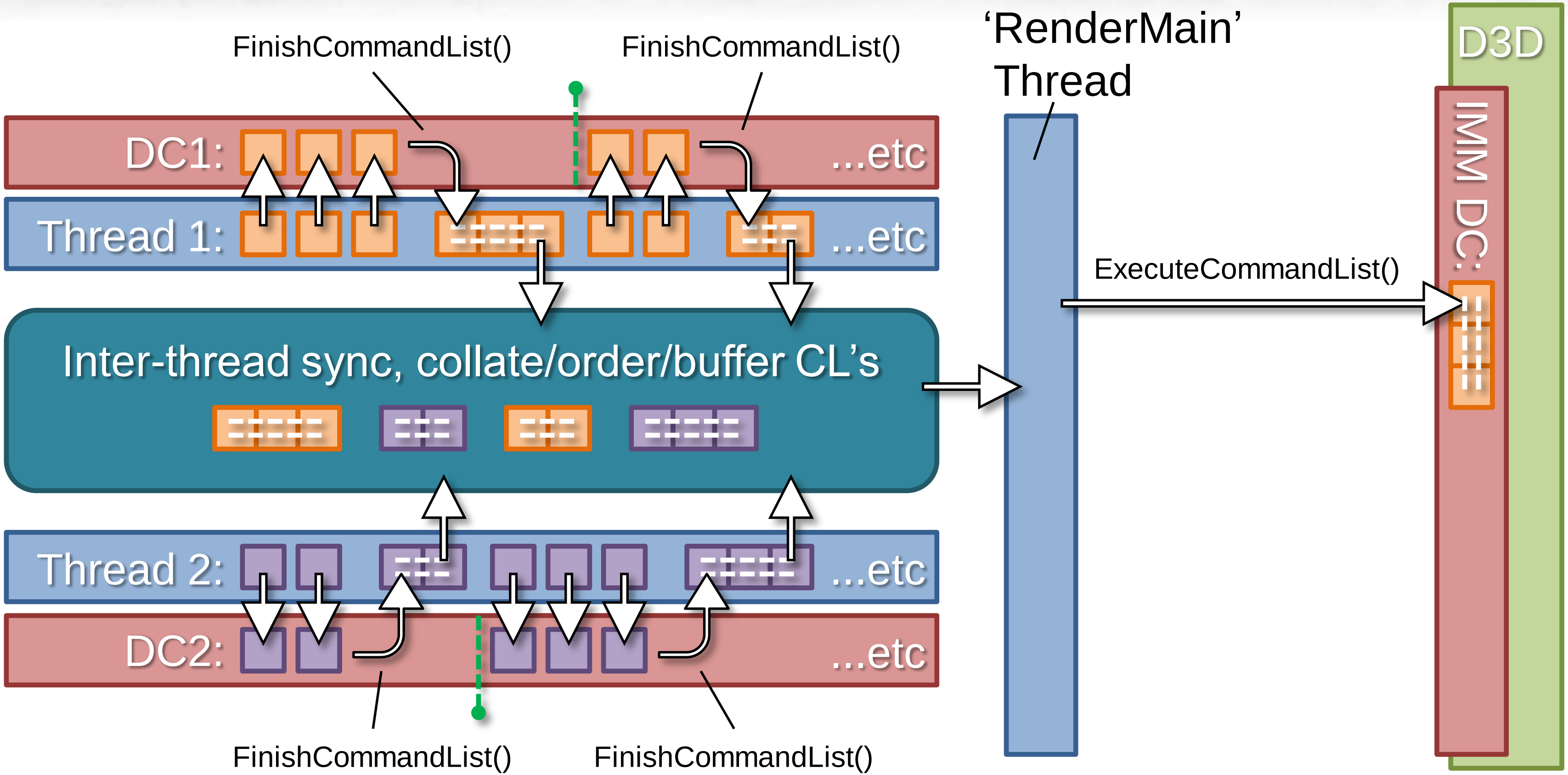

FinishCommandList()
DC1:
Thread 1:
Inter-thread sync, collate/order/buffer CL's
Thread 2:
DC2:
FinishCommandList()
D3D
IMM DC:

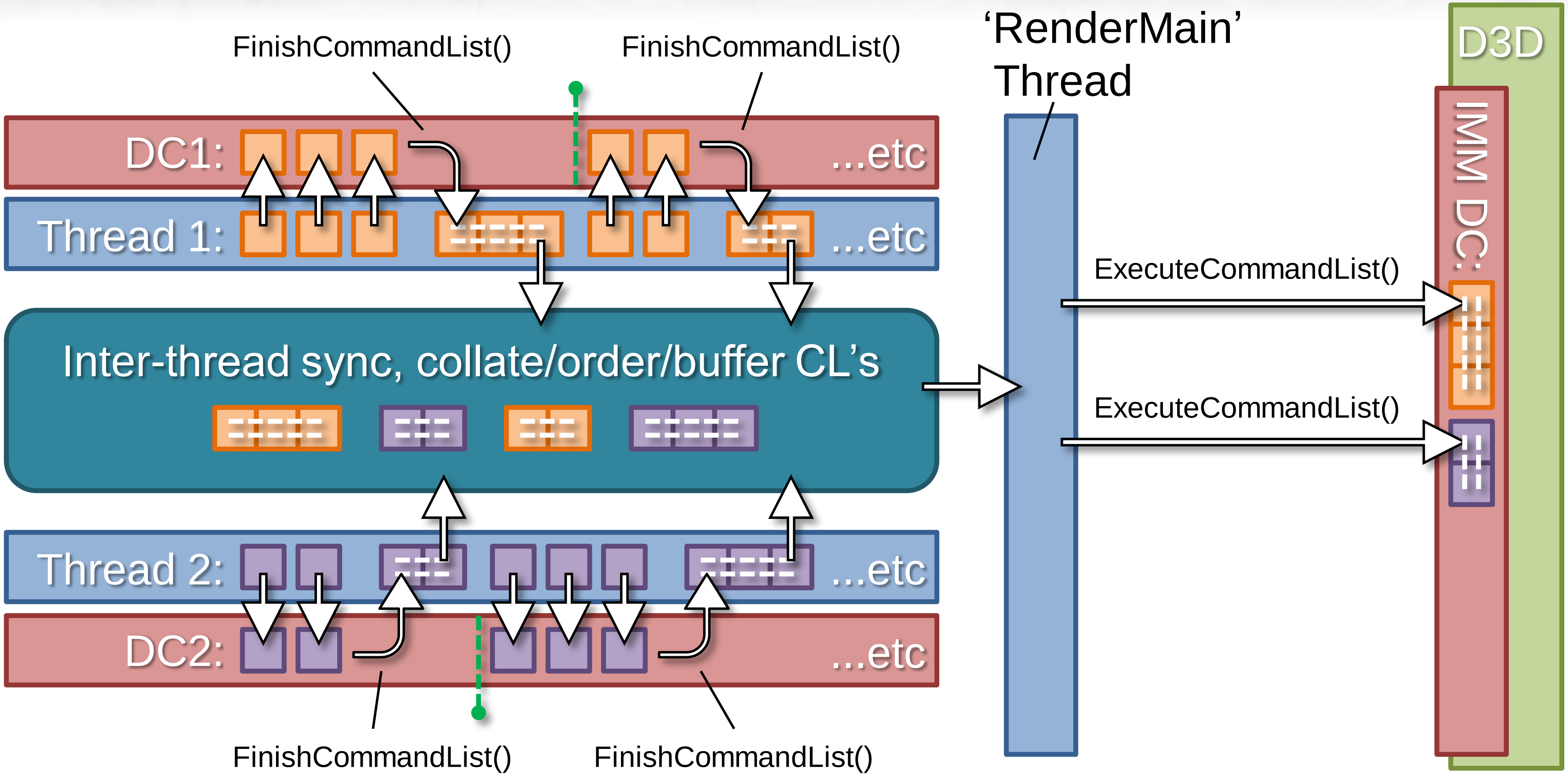

FinishCommandList()
DC1:
Thread 1:
Inter-thread sync, collate/order/buffer CL's
Thread 2:
DC2:
FinishCommandList()
D3D
IMM DC:

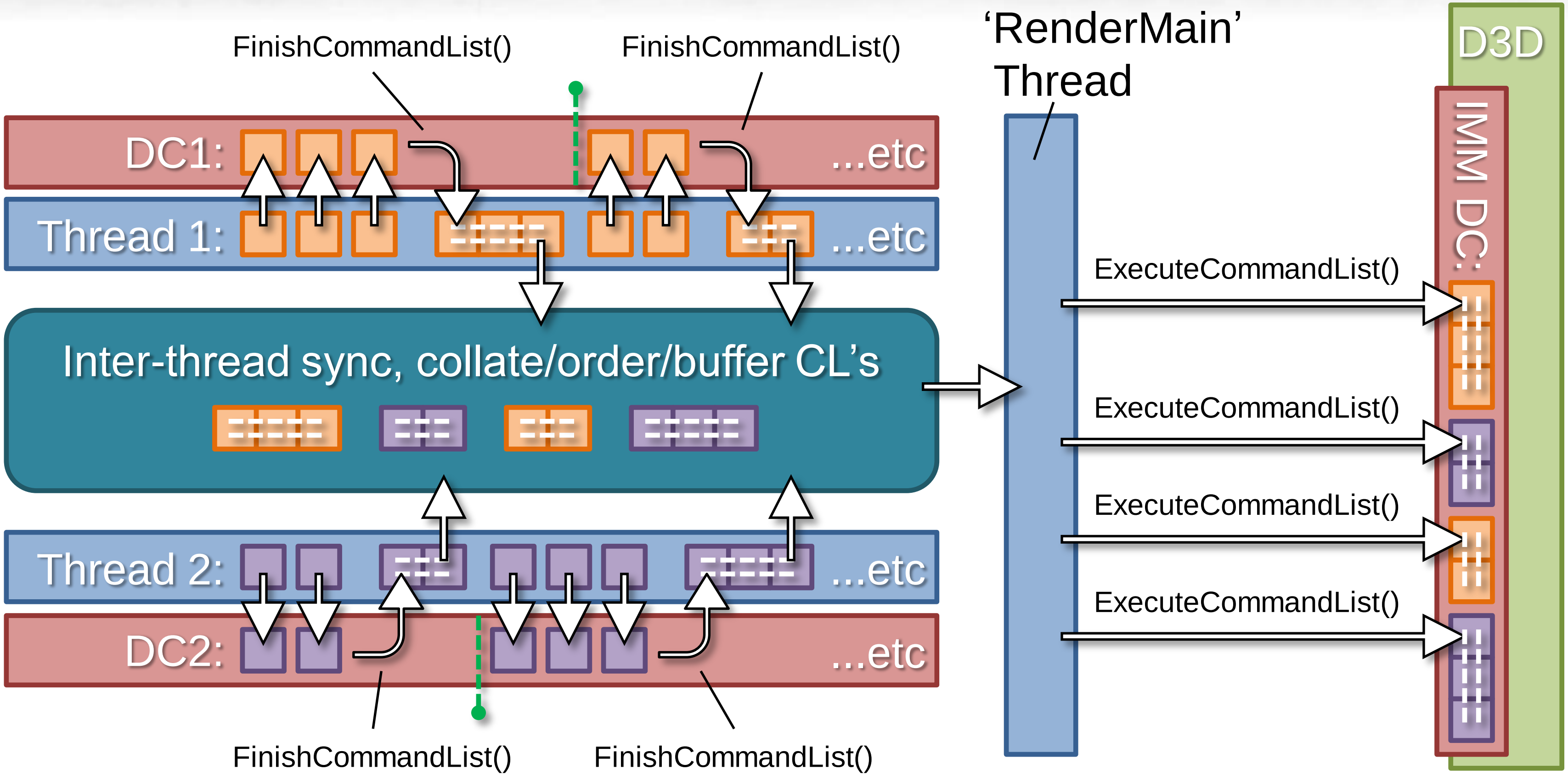


FinishCommandList()
Start of new CL
DC1:
Thread 1:
Inter-thread sync, collate/order/buffer CL's
Thread 2:
DC2:
FinishCommandList()
D3D
IMM DC:


FinishCommandList()
FinishCommandList()
DC1:
...etc
Thread 1:
...etc
Inter-thread sync, collate/order/buffer CL's
Thread 2:
...etc
DC2:
...etc
FinishCommandList()
FinishCommandList()
D3D
IMM DC:


FinishCommandList()
FinishCommandList()
'RenderMain' Thread
D3D
DC1:
...etc
Thread 1:
...etc
IMM DC:
ExecuteCommandList()
Inter-thread sync, collate/order/buffer CL's
Thread 2:
...etc
DC2:
...etc
FinishCommandList()
FinishCommandList()


FinishCommandList()
FinishCommandList()
'RenderMain' Thread
D3D
DC1:
...etc
Thread 1:
...etc
IMM DC:
ExecuteCommandList()
Inter-thread sync, collate/order/buffer CL's
ExecuteCommandList()
Thread 2:
...etc
DC2:
...etc
FinishCommandList()
FinishCommandList()


FinishCommandList()
FinishCommandList()
'RenderMain' Thread
D3D
DC1:
...etc
Thread 1:
...etc
IMM DC:
ExecuteCommandList()
Inter-thread sync, collate/order/buffer CL's
ExecuteCommandList()
ExecuteCommandList()
Thread 2:
...etc
ExecuteCommandList()
DC2:
...etc
FinishCommandList()
FinishCommandList()

Deferred Contexts

- Flexible DX11 internals
  - DX11 runtime has built-in implementation
  - BUT: the driver can take charge, and use its own implementation
    - e.g. command lists could be built at a lower level, moving more of the CPU work onto submission threads

Deferred Contexts: Perf

- Try to balance workload over contexts/threads
  - but submission workloads are seldom predictable
  - granularity helps (if your submission threads are able to pick up work dynamically)
  - if possible, do heavier submission workloads first
  - ~12 CL's per core, ~1ms per CL is a good target

Deferred Contexts: Perf

- Target reasonable command list sizes
  - think of # of draw calls in a command list much like # triangles in a draw call
  - i.e. each list has overhead
    - ~equivalent to a few dozen API calls

Deferred Contexts: Perf

- Leave some free CPU time!
  - having all threads busy can cause CPU saturation and prevent “server” thread from rendering*
  - ‘busy’ includes busy-waits (i.e. polling)

*this is good general advice: never use more than N-1 CPU cores for your game engine. Always leave one for the graphics driver

Deferred Contexts: Perf

- Mind your memory!
  - each Map() call associates memory with the CL
  - releasing the CL is the only way to release the memory
  - could get tight in a 2GB virtual address space!

Deferred Contexts: wrapping up

- DC's + multi-core = pump the API **HARD**
- Real-world specifics coming up in Dan's talk

CALL TO ACTION: If you wish you could submit more batches and you're not already using DC's, then experiment!

CASE STUDY: OPACITY MAPPING

Case study: Opacity Mapping

Case study: Opacity Mapping

GOALS:

- Plausible lighting for a game-typical particle system (16K largish translucent billboards)
- Self-shadowing (using opacity mapping)
  - Also receive shadows from opaque objects
- 3 light sources (all with shadows)

Case study: Opacity Mapping

+ opacity mapping*

=

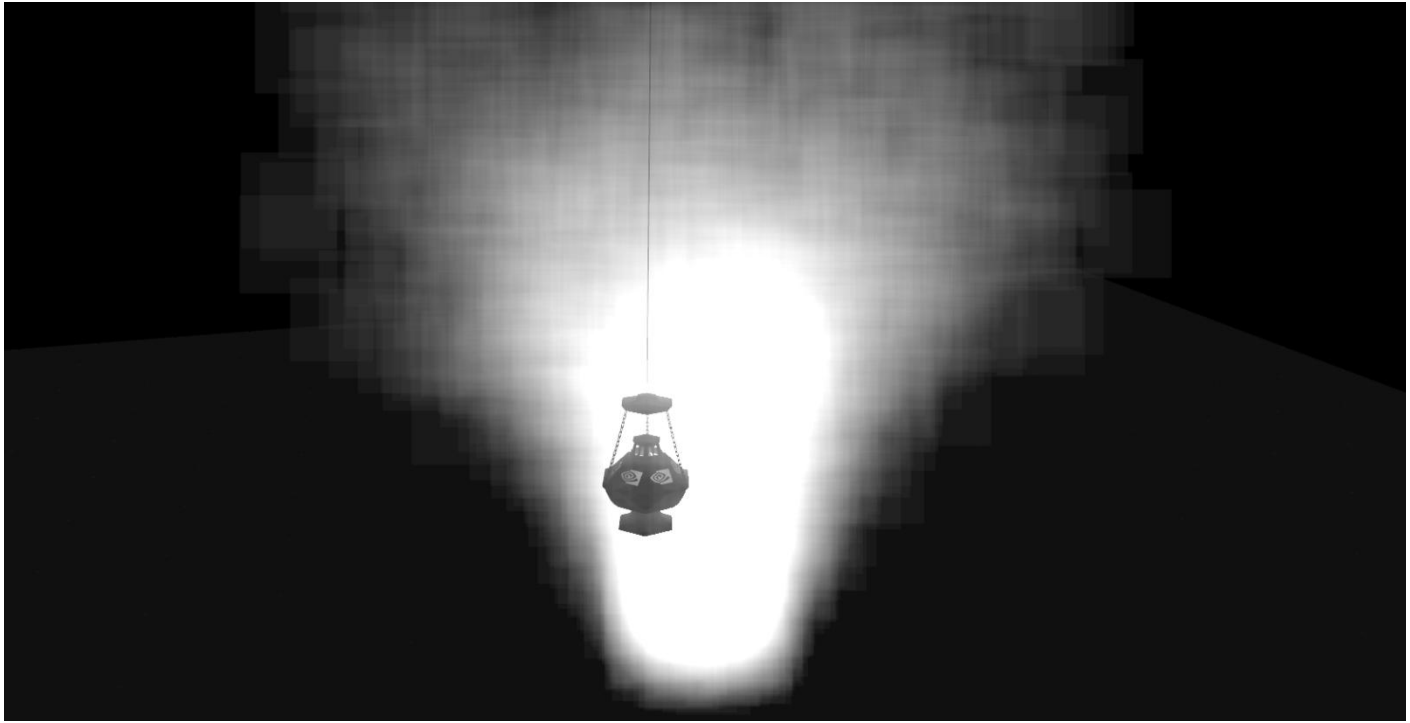
*e.g. [Jansen & Bavoil, 2010]

Case study: Opacity Mapping

- Brute force (per-pixel lighting/shadowing) is not performant
  - 5 to 10 FPS on GTX 560 Ti* or HD 6950*
- Not surprising considering amount of overdraw...

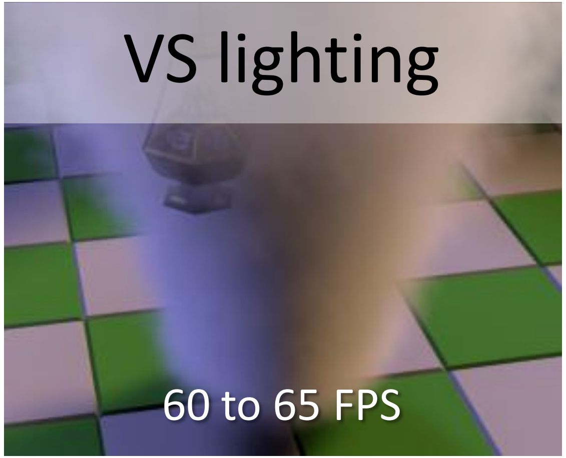
*1680x1050, 4xAA

Case study: Opacity Mapping



Case study: Opacity Mapping

– Vertex lighting? Faster, but shows significant delta from 'ground truth' of per-pixel lighting...

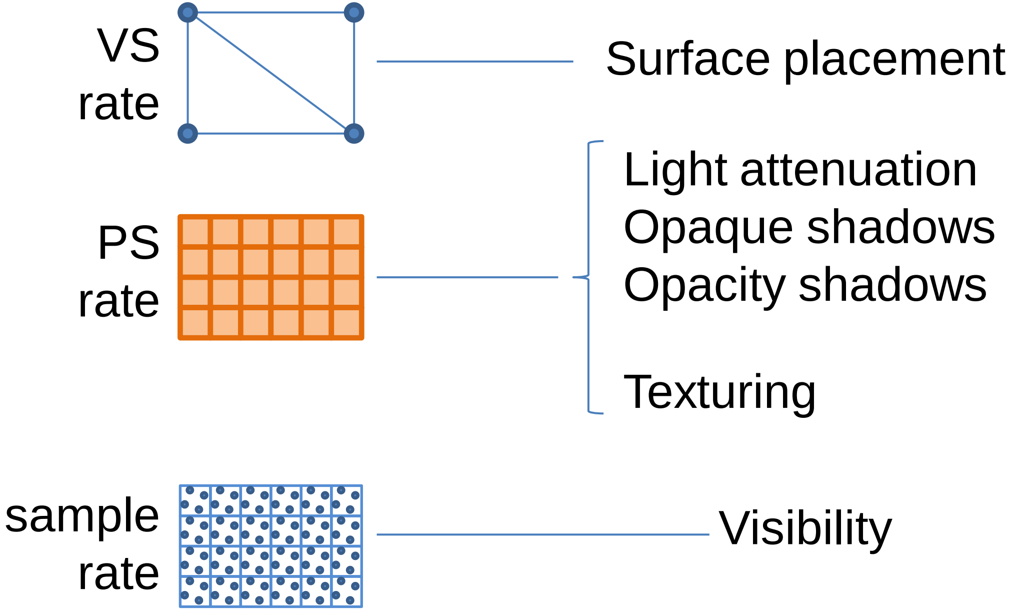


Case study: Opacity Mapping

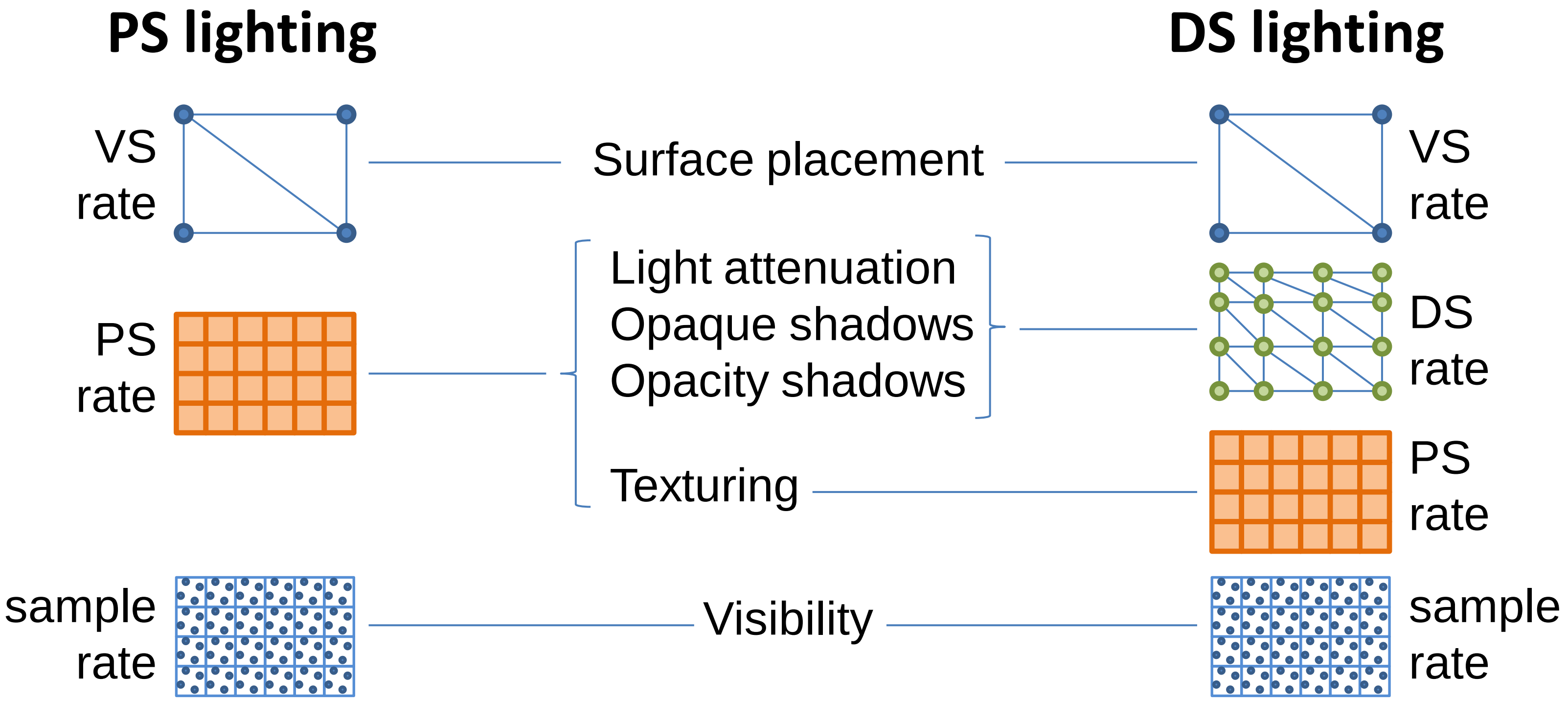
- Use DX11 tessellation to calculate lighting at an intermediate ‘sweet-spot’ rate in the DS
- High-frequency components can remain at per-pixel or per-sample rates, as required
  - opacity
  - visibility

Case study: Opacity Mapping

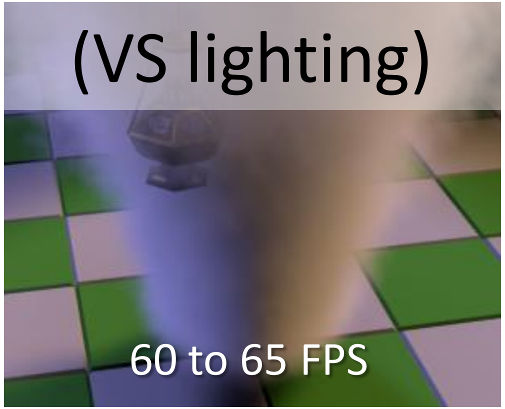
PS lighting

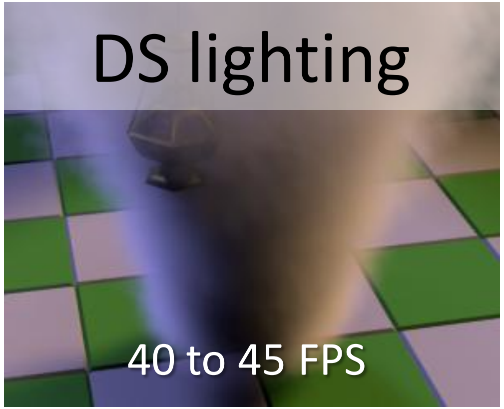


Case study: Opacity Mapping



Case study: Opacity Mapping





Case study: Opacity Mapping

- Adaptive tessellation gives best of both worlds
  - VS-like calculation frequency
  - PS-like relationship with screen pixel frequency
    - 1:15 works well in this case
- Applicable to any slowly-varying shading result
  - GI, other volumetric algos

Case study: Opacity Mapping

- Main bottleneck is fill-rate following tess-opt
- So... render particles to low-res offscreen buffer*
  - significant benefit, even with tess opt (1.2x to 1.5x for GTX 560 Ti / HD 6950)
  - **<u>BUT:</u>** simple bilinear up-sampling from low-res can lead to artifacts at edges...

*[Cantlay, 2007]

Case study: Opacity Mapping

Ground truth (full res)

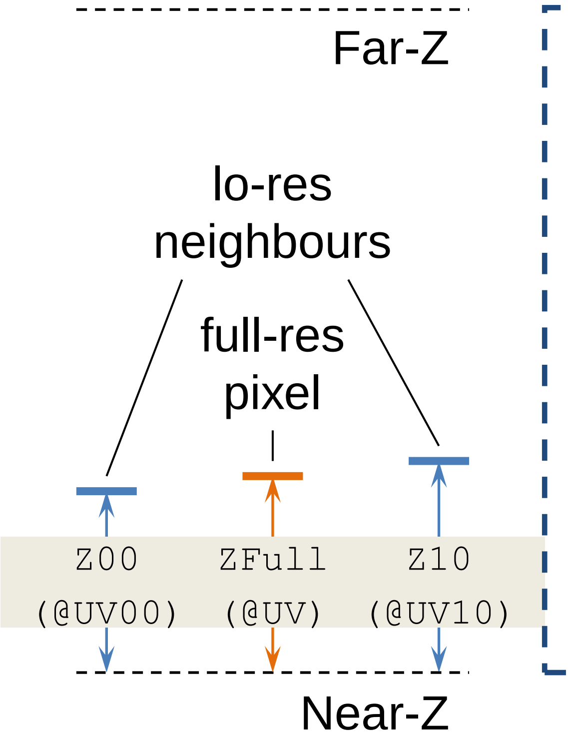
Bilinear up-sample (half-res)

Case study: Opacity Mapping

- Instead, we use nearest-depth up-sampling*
  - conceptually similar to cross-bilateral filtering**
  - compares high-res depth with neighbouring low-res depths
  - samples from closest matching neighbour at depth discontinuities (bilinear otherwise)

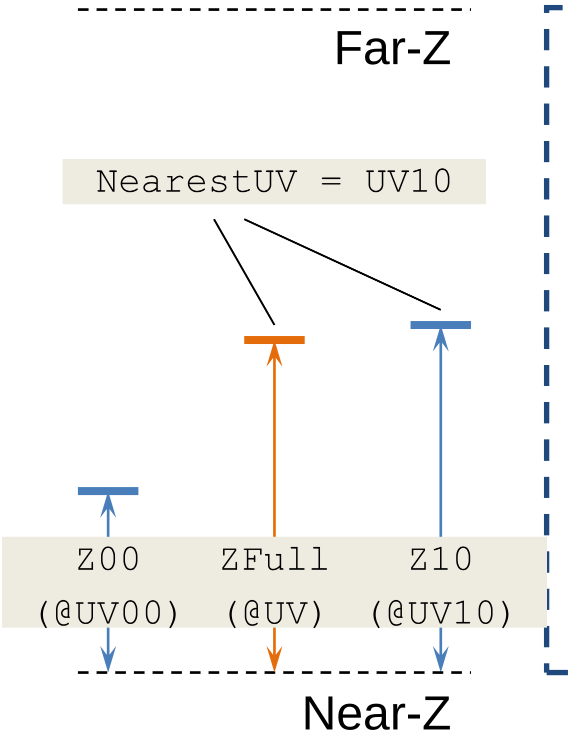
*[Bavoil, 2010] **[Eisemann & Durand, 2004] [Petschnigg et al, 2004]

Case study: Opacity Mapping




```
if( abs(Z00-ZFull) < kDepthThreshold &&
    abs(Z10-ZFull) < kDepthThreshold &&
    abs(Z01-ZFull) < kDepthThreshold &&
    abs(Z11-ZFull) < kDepthThreshold )
{
   return loResColTex.Sample(sBilin,UV);
}
    else
{
   return loResColTex.Sample(sPoint,NearestUV);
}
```

Case study: Opacity Mapping




```
if( abs(Z00-ZFull) < kDepthThreshold &&
    abs(Z10-ZFull) < kDepthThreshold &&
    abs(Z01-ZFull) < kDepthThreshold &&
    abs(Z11-ZFull) < kDepthThreshold )
{
   return loResColTex.Sample(sBilin,UV);
}
    else
{
   return loResColTex.Sample(sPoint,NearestUV);
}
```

Case study: Opacity Mapping

Ground truth (full res)

Nearest-depth up-sample

Case study: Opacity Mapping

- Use SM5 GatherRed() to efficiently fetch 2x2 low-res depth neighbourhood in one go

```
float4 zg = g_DepthTex.GatherRed(g_Sampler,UV);
float z00 = zg.w;    // w: floor(uv)
float z10 = zg.z;    // z: ceil(u),floor(v)
float z01 = zg.x;    // x: floor(u),ceil(v)
float z11 = zg.y;    // y: ceil(uv)
```

Case study: Opacity Mapping

- Nearest-depth up-sampling plays nice with AA when run per-sample
  - and surprisingly performant! (FPS hit < 5%)

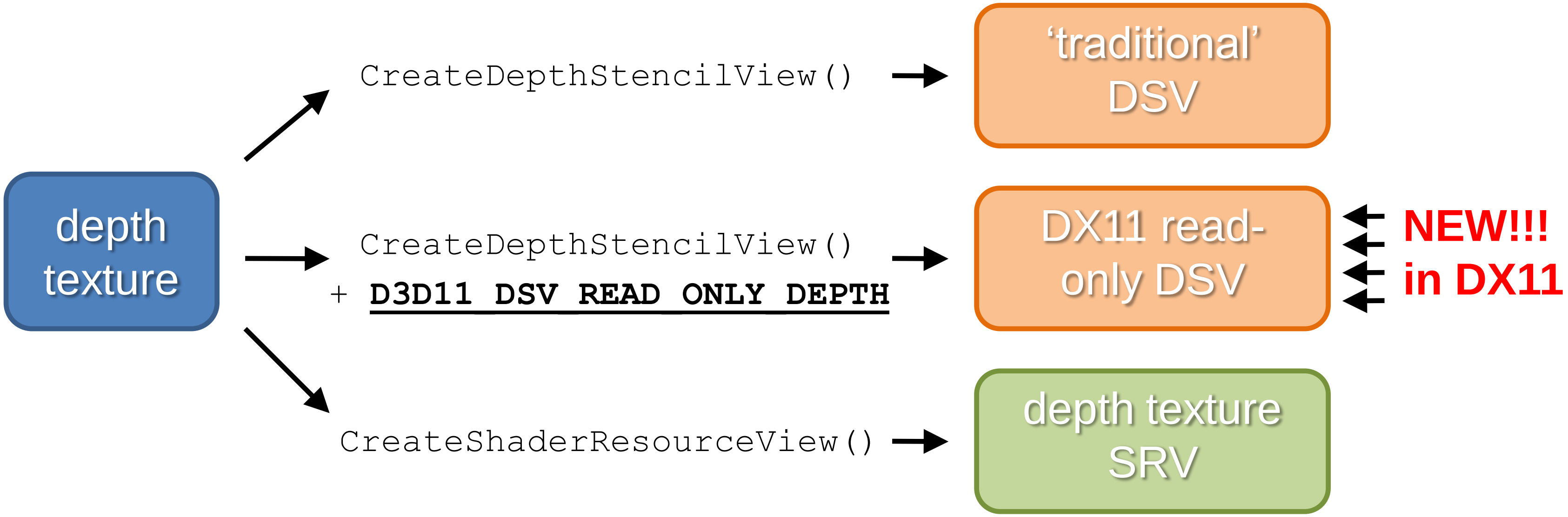
```
float4 UpsamplePS(  VS_OUTPUT In,
                    uint uSID : SV_SampleIndex
                    ) : SV_Target
```

Case study: Opacity Mapping

- Soft particles (depth-based alpha fade)
  - requires read from scene depth
  - for < DX11, this used to mean...
    - **EITHER:** sacrificing depth-test (along with any associated acceleration)
    - **OR**: maintaining two depth surfaces (along with any copying required)

Case study: Opacity Mapping

DX11 solution:



Case study: Opacity Mapping

STEP 1: render opaque objects to depth texture

‘traditional’ DSV

DX11 read-only DSV

depth texture SRV

```
pDC->OMSetRenderTargets(...)

// Render opaque objects
```

Case study: Opacity Mapping

STEP 2: render soft particles with depth-test

‘traditional’ DSV

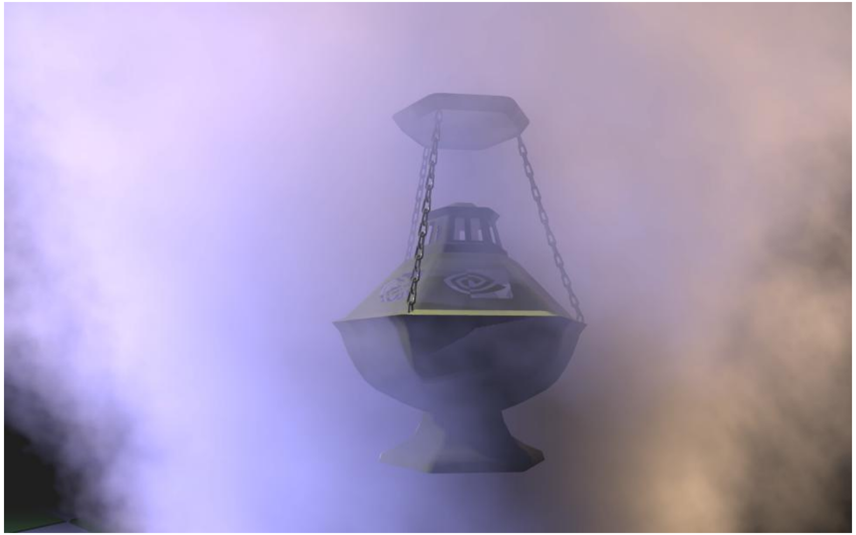
DX11 read-only DSV

depth texture SRV

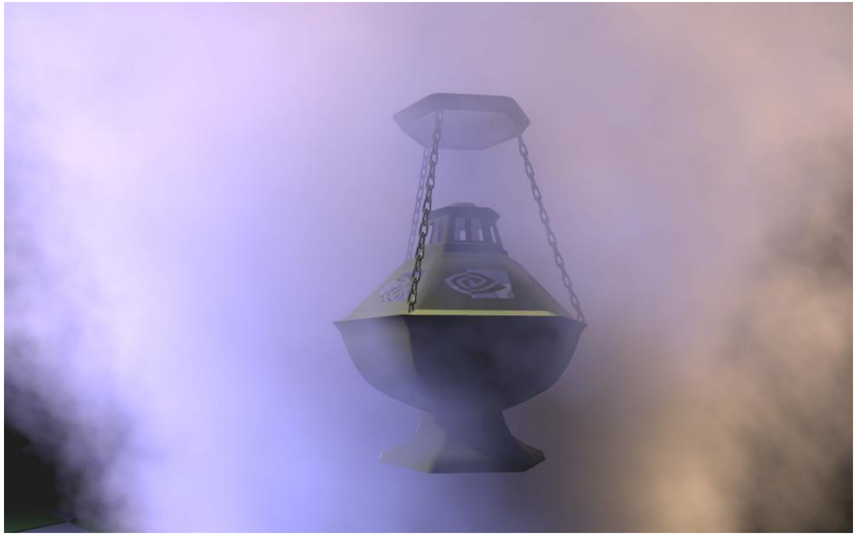
```
pDC->OMSetRenderTargets(...)
pDC->PSSetShaderResources(...)
// (Valid D3D state!)

// Render soft particles
```

Case study: Opacity Mapping

'Hard' particles



Soft particles



Case study: wrapping up

- 5x to 10x overall speedup
- DX11 tessellation gave us most of it
- But rendering at reduced-res alleviates fill-rate and lets tessellation shine thru
- GatherRed() and RO DSV also saved cycles

Case study: wrapping up

CALL TO ACTION: *Go light some particles!*

End of tour!

- Questions?

jjansen at nvidia dot com

References

BAVOIL, L. 2010. Modern Real-Time Rendering Techniques. From *Future Game On Conference, September 2010.*

CANTLAY, I. 2007. High-speed, off-screen particles. In *GPU Gems 3.* 513-528

EISEMANN, E., & DURAND, F. 2004. Flash Photography Enhancement via Intrinsic Relighting. In *ACM Trans. Graph. (SIGGRAPH)* 23, 3, 673–678.

JANSEN, J., AND BAVOIL, L. 2010. Fourier opacity mapping. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 165–172.

PETSCHNIGG, G., AGRAWALA, M., HOPPE, H., SZELISKI, R., COHEN, M., & TOYAMA, K. 2004. Digital photography with flash and no-flash image pairs. In *ACM Trans. Graph. (SIGGRAPH)* 23, 3, 664–672.