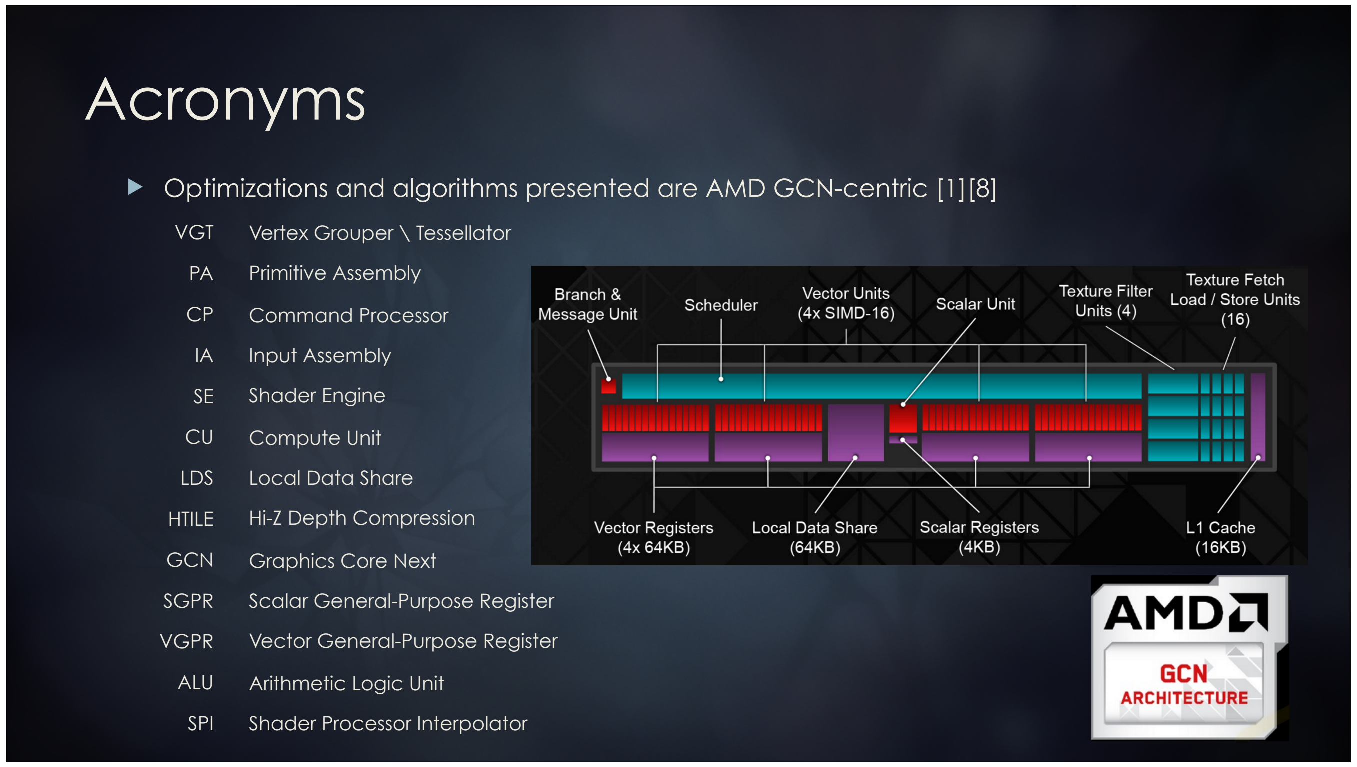
Optimizing the Graphics Pipeline with Compute

Graham Wihlidal
Sr. Rendering Engineer, Frostbite

GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 16–18, 2016 #GDC16

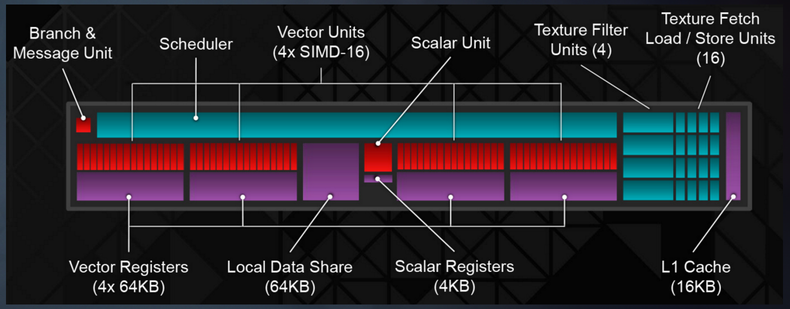
Hello, my name is Graham Wihlidal, a senior rendering engineer on the Frostbite team at EA. Today I am talking about optimizing the graphics pipeline with compute, or more specifically, how to render triangles fast, by not rendering so many triangles.



Acronyms

- Optimizations and algorithms presented are AMD GCN-centric [1][8]

VGT	Vertex Grouper \ Tessellator
PA	Primitive Assembly
CP	Command Processor
IA	Input Assembly
SE	Shader Engine
CU	Compute Unit
LDS	Local Data Share
HTILE	Hi-Z Depth Compression
GCN	Graphics Core Next
SGPR	Scalar General-Purpose Register
VGPR	Vector General-Purpose Register
ALU	Arithmetic Logic Unit
SPI	Shader Processor Interpolator



To start things off, I’d like to list a few acronyms. The optimizations and algorithms presented are specific to AMD GCN hardware, as this first version was aimed at “getting it right” on consoles and high end AMD PCs. The following definitions map to GCN hardware concepts, and are used throughout this presentation.

During an onsite at Microsoft with the Advanced Technology Group, while optimizing Dragon Age: Inquisition, it was clear that our displacement mapping was performing poorly on the hardware. Despite small improvements done to the shaders to reduce LDS usage, it was obvious that we were at the mercy of various bottlenecks.

I began experimenting with offloading the hull shader adaptive tessellation factor calculations to a compute shader and reading the results back in the hull shader. This prototype was quite successful, and included interesting approaches like HTILE sourced Hi-Z culling of the triangle patches.

```
CPU  11.863 ms, GPU  13.975 ms,  69 Hz, Latency:  29 ms
Engine Tuning
+ Application/...
[X]              Display Frame Rate
[-]              Display Profiler
- DrawBench/...
  [X]              Async Compute
  [X]              CPU Precull
  [-]              Depth Cull
  MDI              Draw Mode
  [X]              Index Cull
  10000            Object Count
+ Graphics/...
+ Timing/...
```

This then led me to build a new prototype for regular triangle and patch culling, as a spiritual successor to libEdge on PS3. This prototype was interesting; some scenes would be a win, and some would be a complete loss. I played around with various GCN instructions, memory optimizations, and asynchronous compute, and found scenes that showed a significant win using compute based triangle culling.

At a conference, I started chatting with Alex Nankervis and James Stanard from Microsoft, and learned they had been doing their own investigations for this [18]. We did a lot of excellent idea sharing back and forth, and my technology definitely improved thanks to their help.

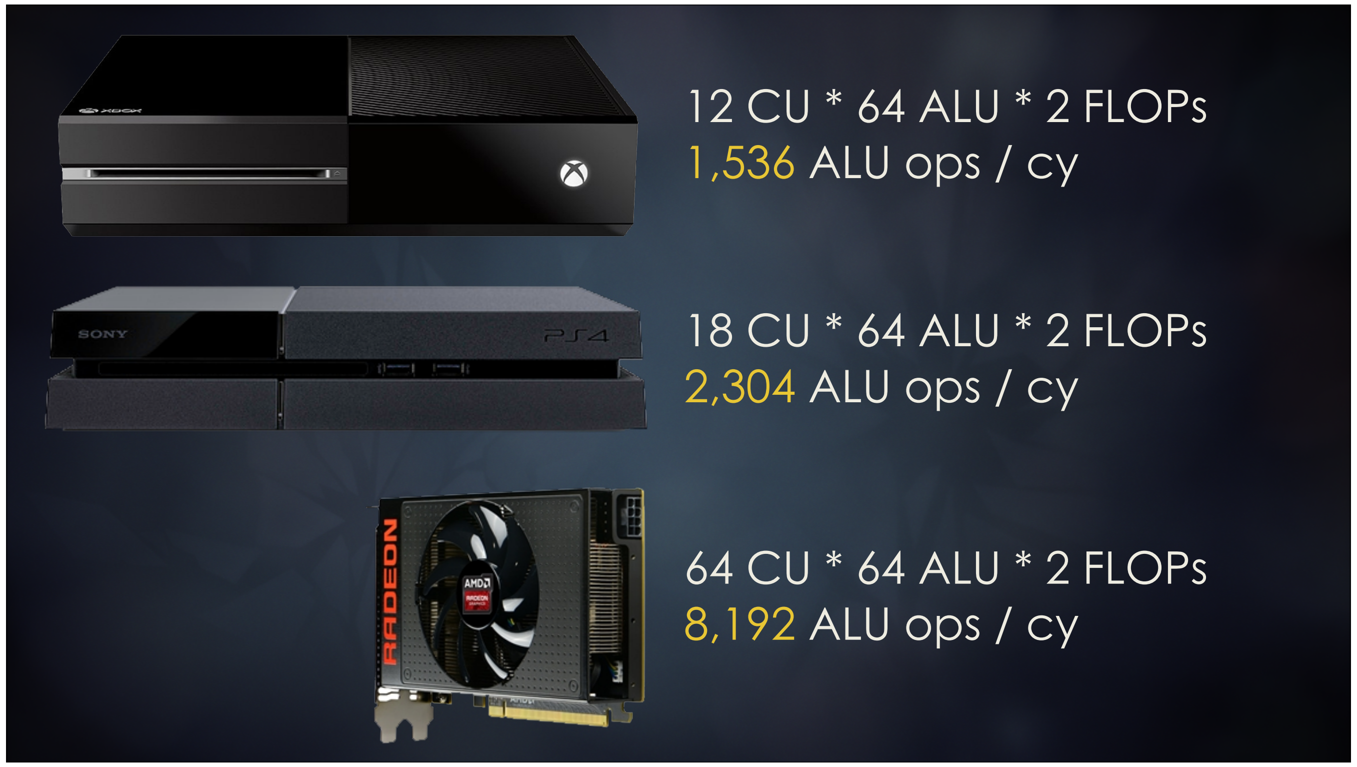
The final advancement, was when I met Matthäus Chajdas at AMD Munich, whom was also experimenting with compute triangle culling, now titled GeometryFX [20], and released open source as part of AMD GPUOpen. This sparked a great collaboration to push this technology further…

FROSTBITE™ 3

… which is now integrated into the Frostbite Engine, on the majority of our titles moving forward.

/ clock
/ clock
/ clock

Lets break down peak triangle rate for our target platforms, and prove how we can beat it. Both Xbox One and PlayStation 4 have 2 shader engines, where each can issue 1 triangle per clock. Newer AMD GPUs on PC have 4 shader engines, so a total of 4 triangles per clock.

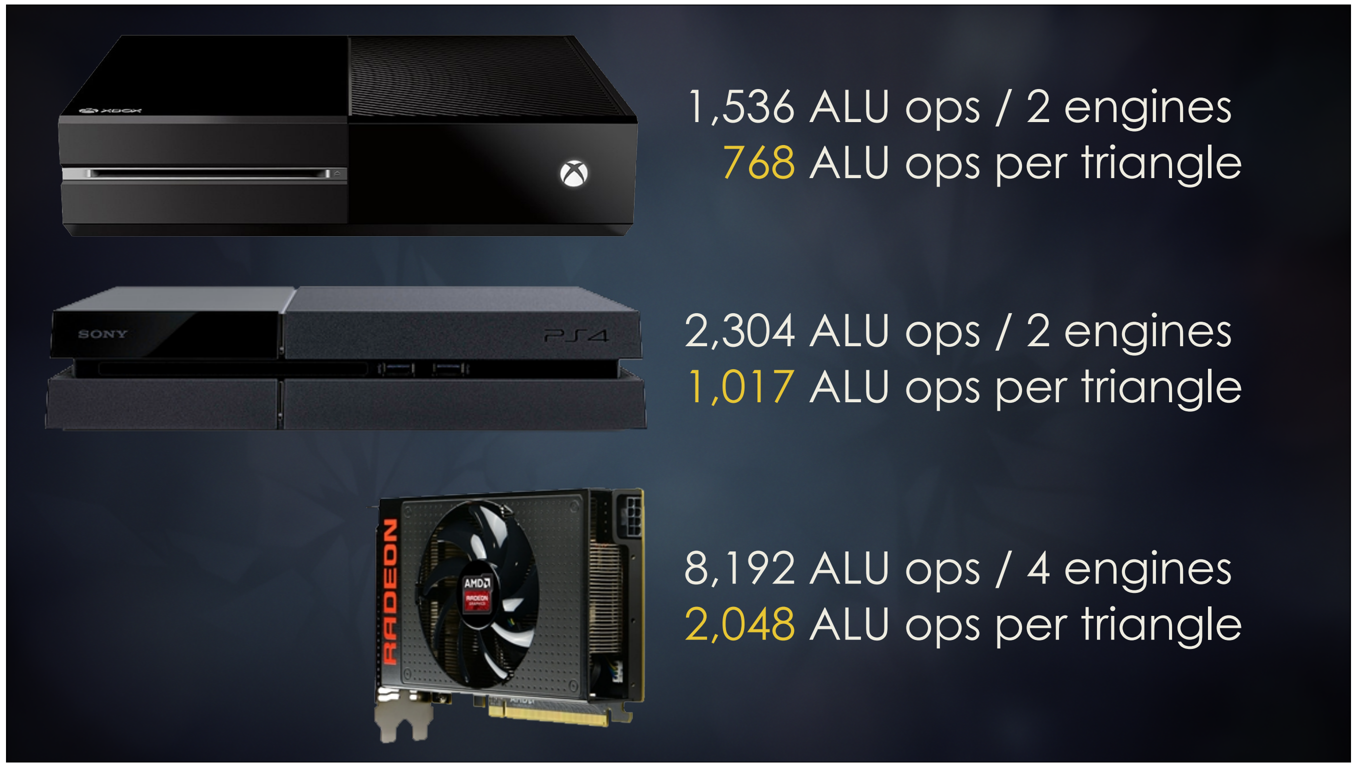


12 CU * 64 ALU * 2 FLOPs
1,536 ALU ops / cy

18 CU * 64 ALU * 2 FLOPs
2,304 ALU ops / cy

64 CU * 64 ALU * 2 FLOPs
8,192 ALU ops / cy

If we multiply the number of Compute Units (CUs) by the number of ALUs per CU, and multiply that value by 2 floating point operations, since one CU can execute 64 FMA in one cycle, we get the number of ALU ops that can be executed per cycle. There is a technical caveat to mention with this math. There are actually 4x as many waves running, but each ALU takes 4 clocks, so those two factors of 4 cancel out.

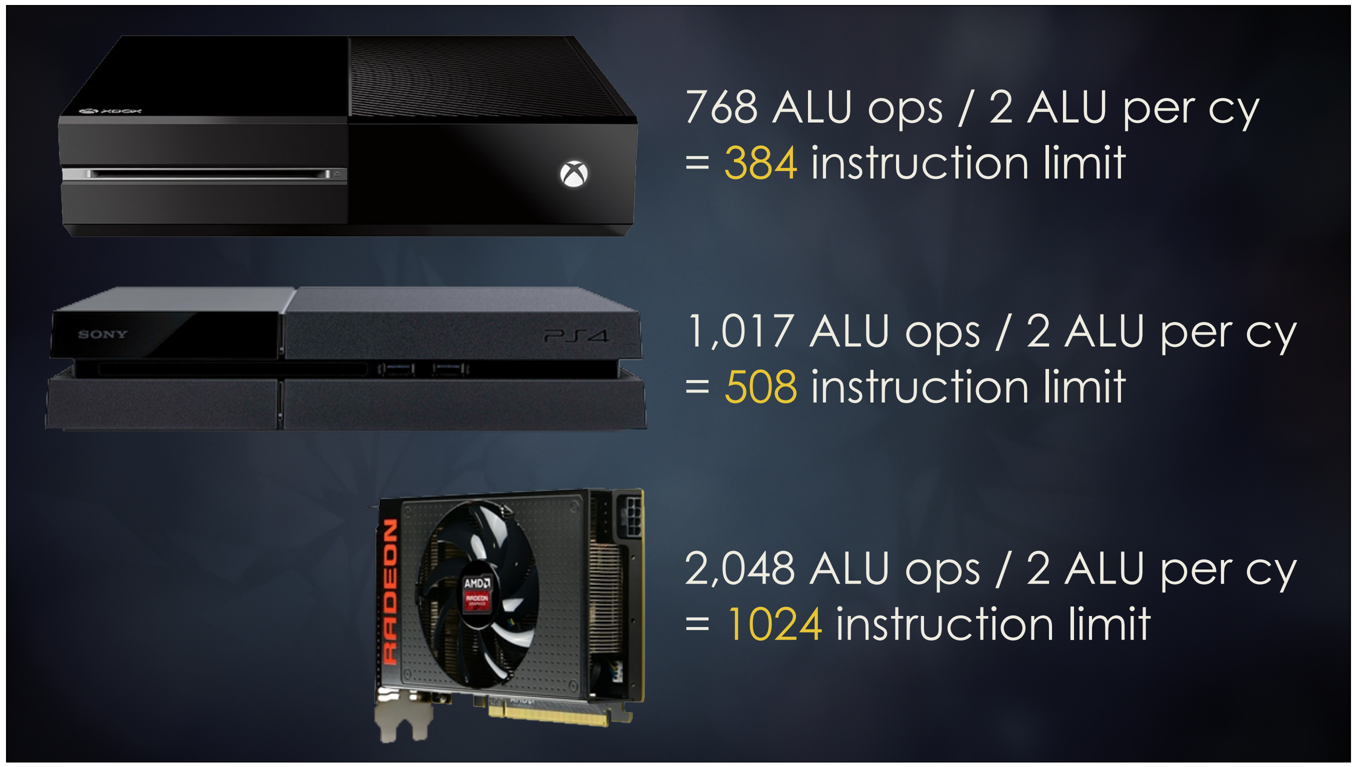


1,536 ALU ops / 2 engines
768 ALU ops per triangle

2,304 ALU ops / 2 engines
1,017 ALU ops per triangle

8,192 ALU ops / 4 engines
2,048 ALU ops per triangle

If we take the number of ALU ops that can be executed per cycle, and divide that by the number of available shader engines, we get the number of ALU ops that can be executed per triangle.

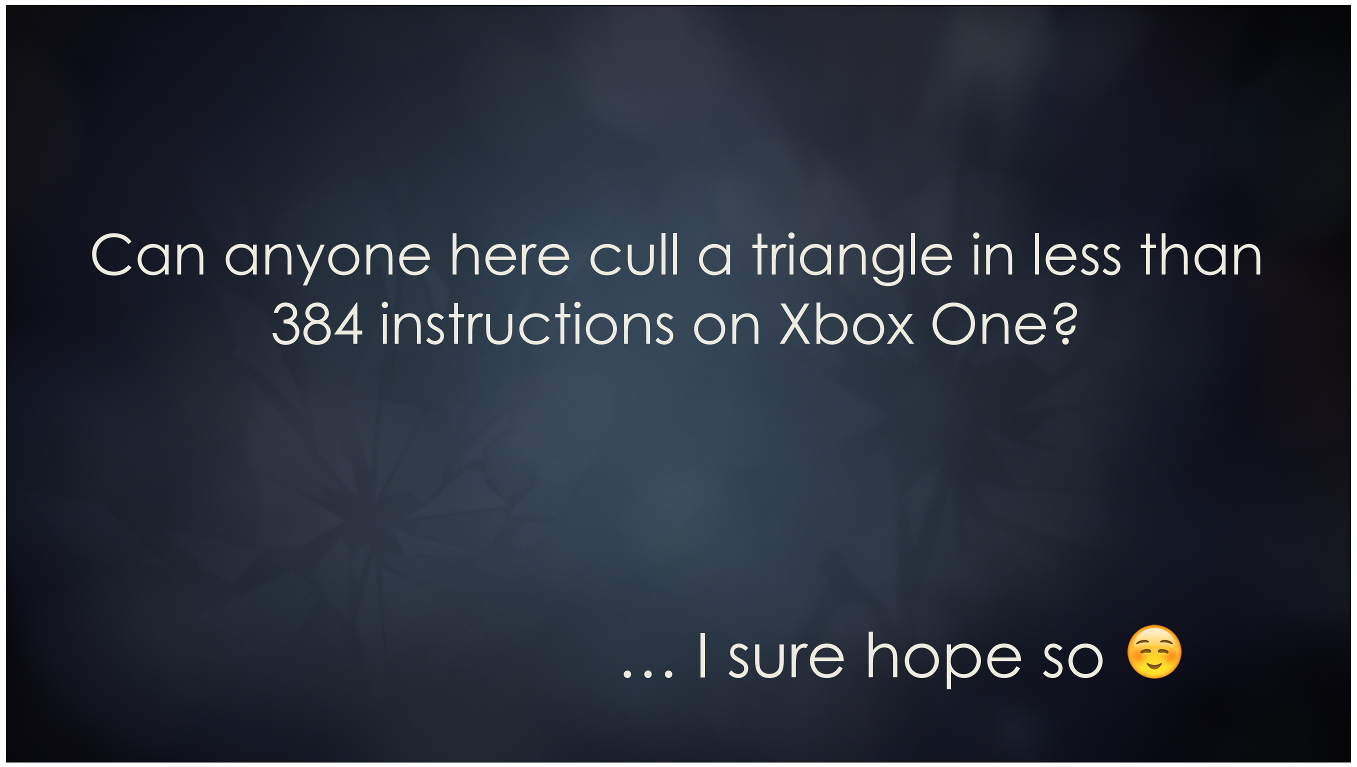


768 ALU ops / 2 ALU per cy
= 384 instruction limit

1,017 ALU ops / 2 ALU per cy
= 508 instruction limit

2,048 ALU ops / 2 ALU per cy
= 1024 instruction limit

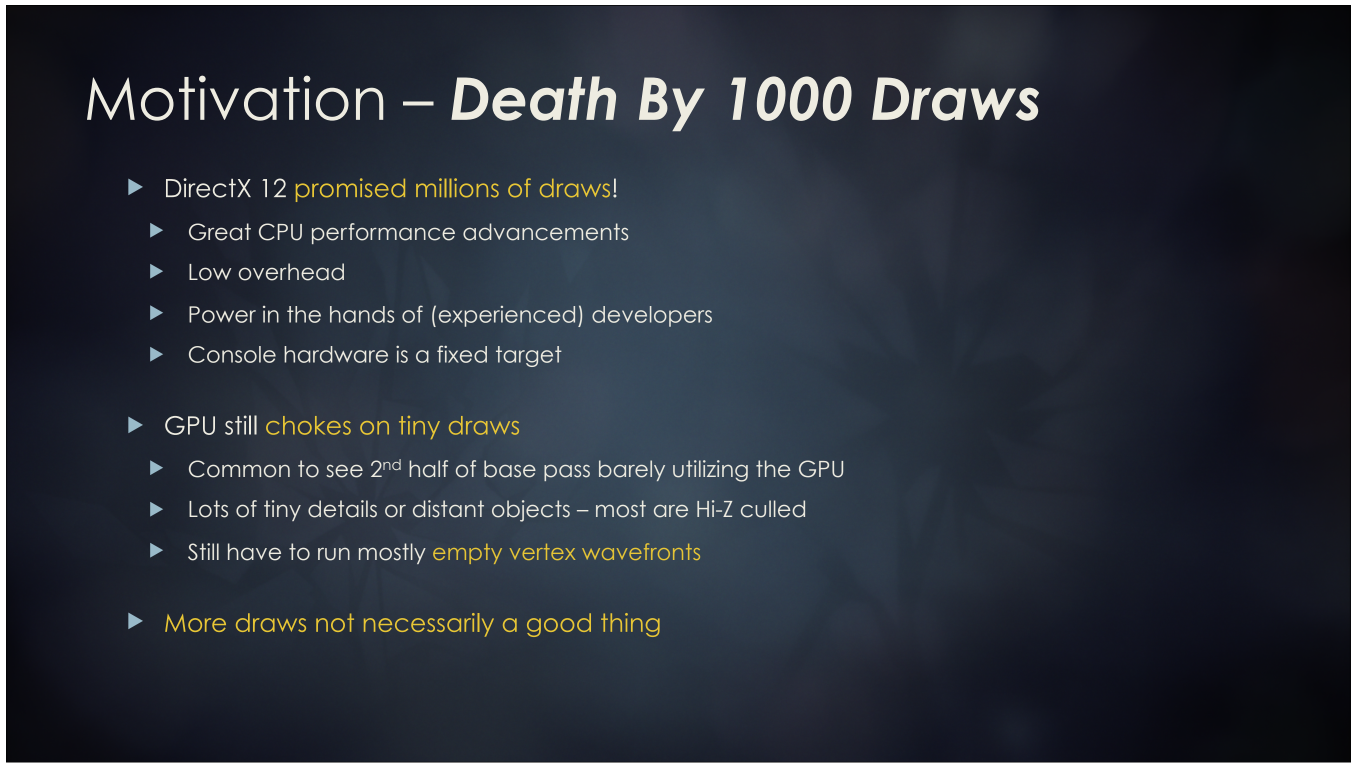
Finally, if we divide the number of ALU ops per triangle by the number of ALU ops per clock, we get a final instruction upper limit that we have to cull a triangle with, and still beat the fixed function primitive setup and scan converter.



Can anyone here cull a triangle in less than 384 instructions on Xbox One?

… I sure hope so ☺️

So the question I propose to the audience, is whether or not you can write a compute shader within 384 instructions that can efficiently cull a triangle. I sure hope so ☺️

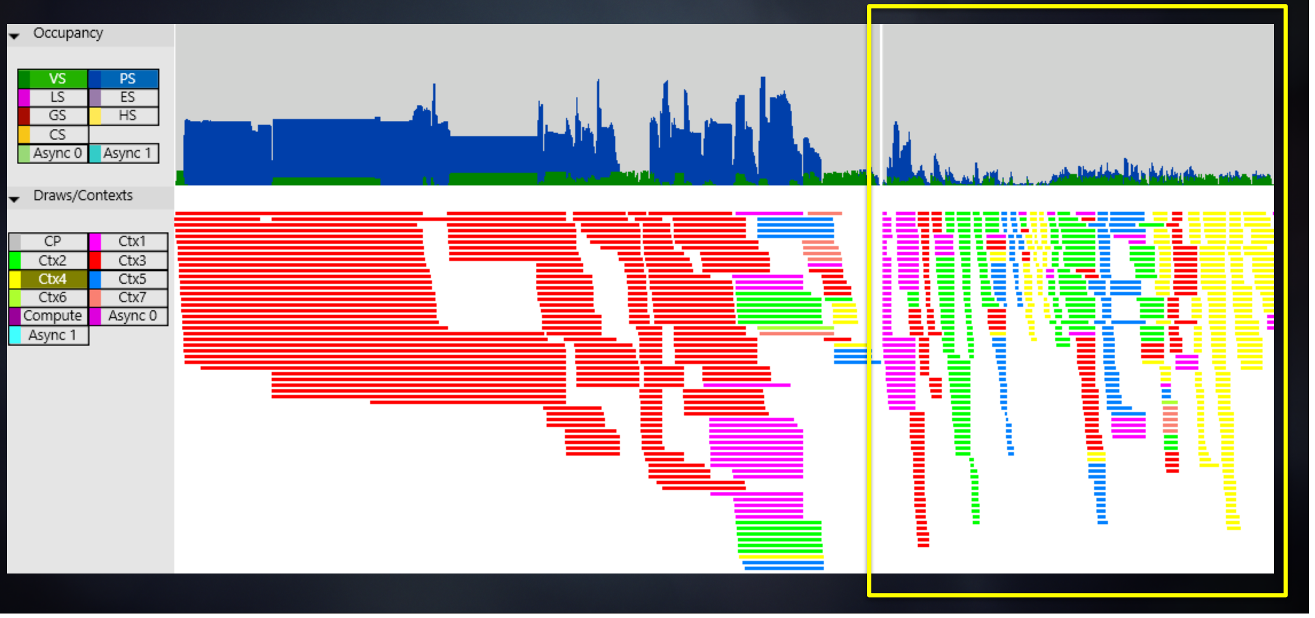


Motivation – ***Death By 1000 Draws***

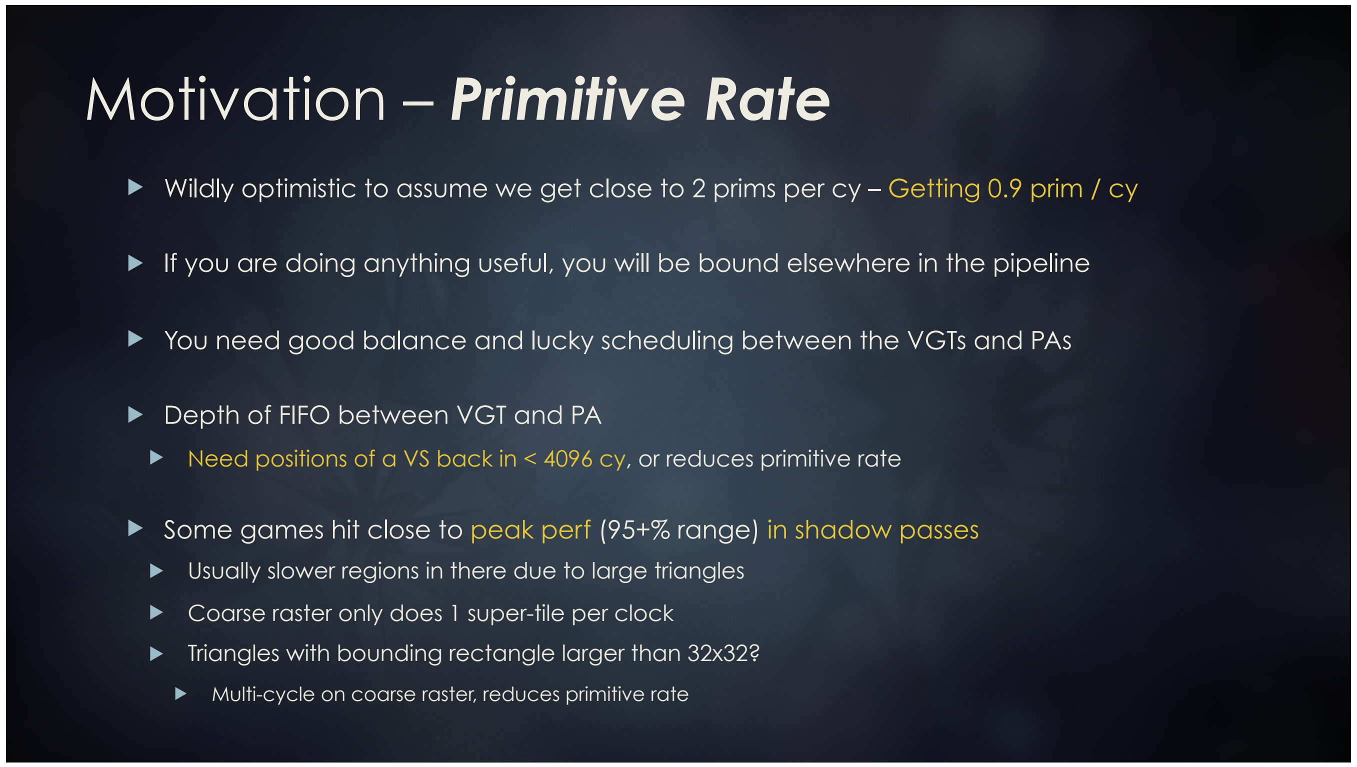
- DirectX 12 promised millions of draws!
  - Great CPU performance advancements
  - Low overhead
  - Power in the hands of (experienced) developers
  - Console hardware is a fixed target
- GPU still chokes on tiny draws
  - Common to see 2nd half of base pass barely utilizing the GPU
  - Lots of tiny details or distant objects – most are Hi-Z culled
  - Still have to run mostly empty vertex wavefronts
- More draws not necessarily a good thing

We now have DirectX 12, and we were promised millions of draws. The new API has given great CPU performance advancements through low overhead, and places the power in the hands of experienced developers. However, the GPU still chokes on tiny draws; it is quite common to see the 2nd half of the base pass barely utilizing the GPU. Typically there are lots of tiny details or distant objects, of which most are Hi-Z culled. The efficiency loss comes from the GPU still having to run mostly empty vertex wavefronts. More draws are not necessarily a good thing..

Motivation – ***Death By 1000 Draws***



In this GPU capture, you can see on the left that we start out alright, but very quickly on the right we end up spinning on vertex shader wavefronts that that don't result in any pixels.



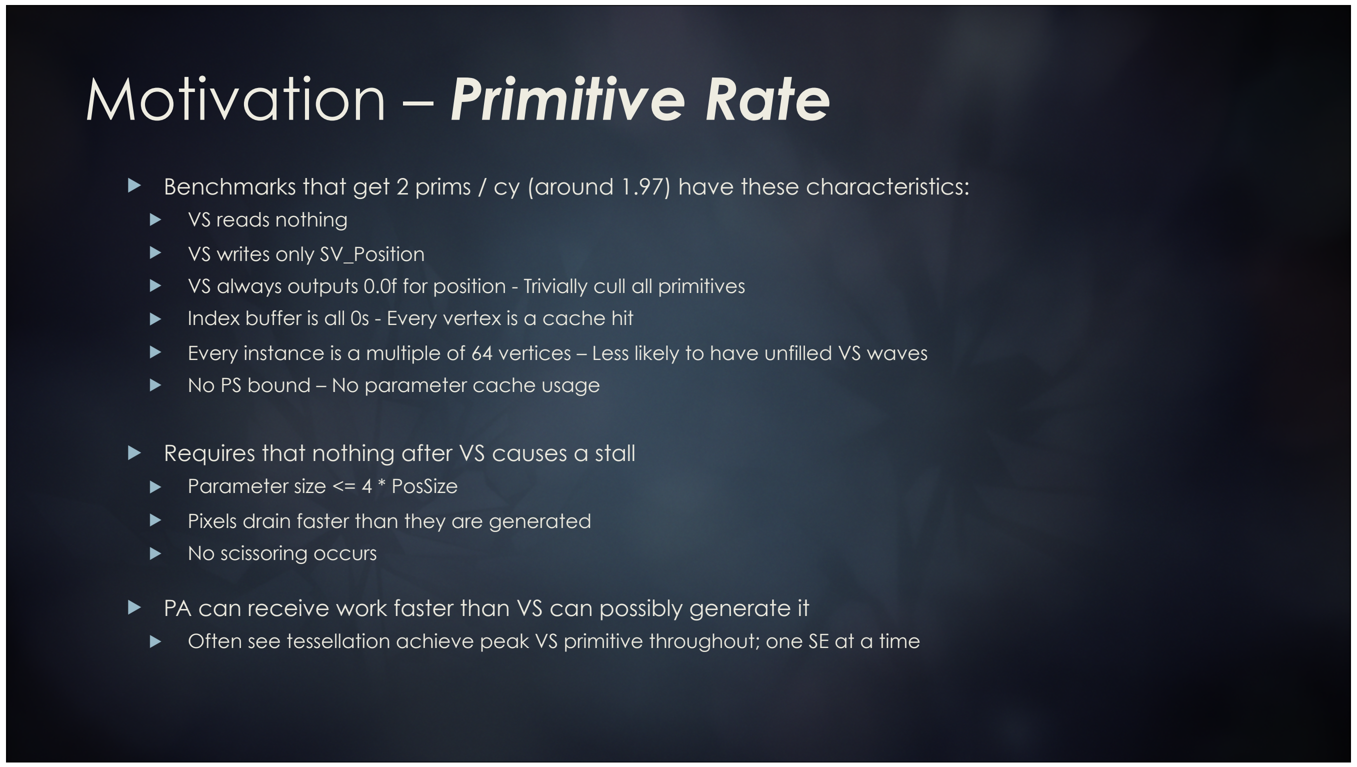
Motivation – ***Primitive Rate***

- Wildly optimistic to assume we get close to 2 prims per cy – Getting 0.9 prim / cy
- If you are doing anything useful, you will be bound elsewhere in the pipeline
- You need good balance and lucky scheduling between the VGTs and PAs
- Depth of FIFO between VGT and PA
  - Need positions of a VS back in < 4096 cy, or reduces primitive rate
- Some games hit close to peak perf (95+% range) in shadow passes
  - Usually slower regions in there due to large triangles
  - Coarse raster only does 1 super-tile per clock
  - Triangles with bounding rectangle larger than 32x32?
    - Multi-cycle on coarse raster, reduces primitive rate

I've shown how easy it can be to beat the peak primitive rate at a cursory glance. The GPU has a lot more going on, so we still have to profile and optimize our culling aggressively, especially bandwidth usage. A saving grace is that it is wildly optimistic to expect that we'll get 2 triangles per clock cycle on consoles, as the rasterizer is subjected to other pipeline bottlenecks; on Xbox One I measured an actual rate of 0.9 triangles per clock with regular rendering, which is really quite healthy, as prim rate is not where you want to be bound.

In practice, if you are actually submitting geometry this fast, and doing any useful rendering, then you will be bound elsewhere in the pipeline, at least during some intervals. Also, you need good balance and lucky scheduling between the two VGTs and PAs to get max rate on each. For instance, the same vertex in two different waves might have to be shaded twice, because the waves alternate between PAs, and the PAs operate independently.

Due to the depth of the FIFO between VGT and PA, you need to get the positions of a VS back in less than 4096 cycles, counting from the moment the vertex goes into the FIFO. This leaves you with slightly fewer cycles than that to compute your positions. If your VS takes longer, prim rate goes down linearly. Some games hit very close to peak perf (in the 95+% range) in shadow passes. There are usually some slower regions in there due to large triangles. The coarse rasterizer only does 1 super-tile per clock, so triangles with a bounding rectangle larger than 32x32 will need to multi-cycle on the coarse rasterizer, reducing primitive rate.



Motivation – ***Primitive Rate***

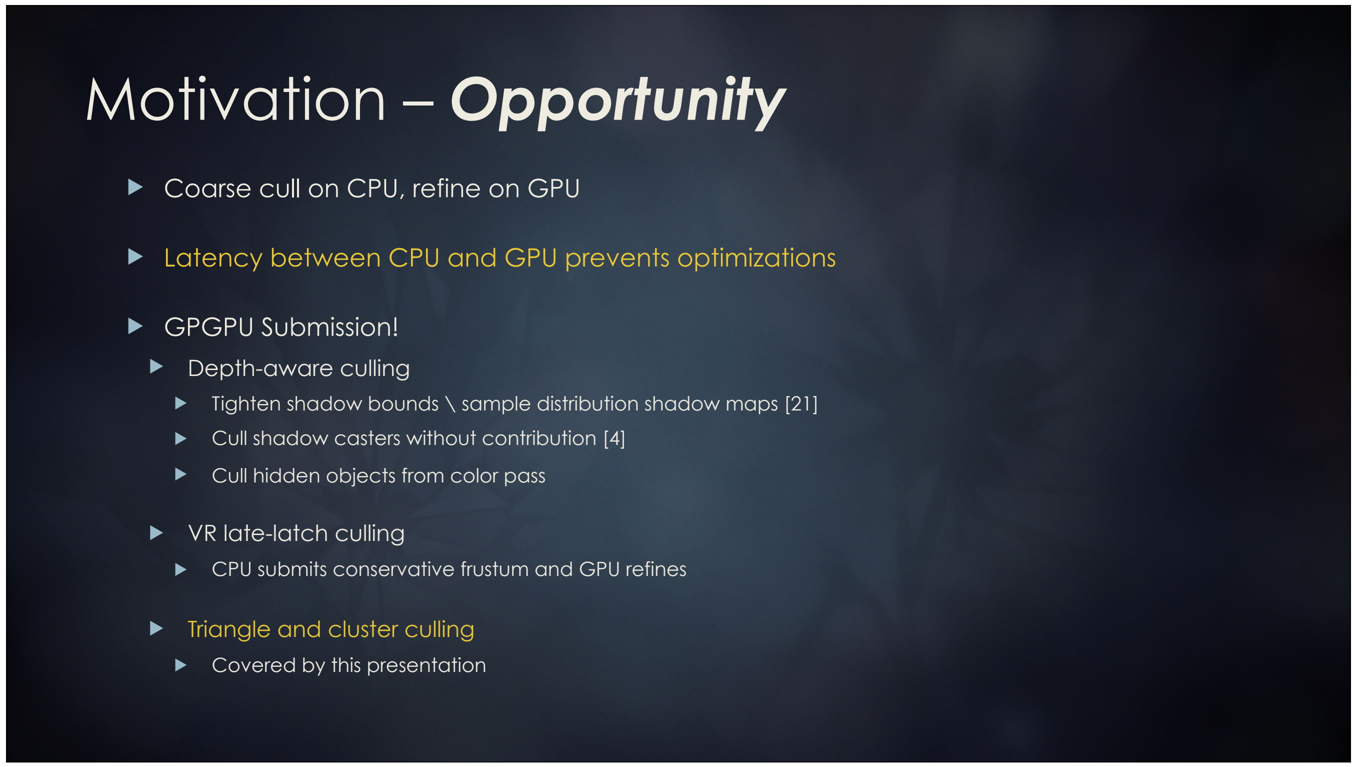
- Benchmarks that get 2 prims / cy (around 1.97) have these characteristics:
  - VS reads nothing
  - VS writes only SV_Position
  - VS always outputs 0.0f for position - Trivially cull all primitives
  - Index buffer is all 0s - Every vertex is a cache hit
  - Every instance is a multiple of 64 vertices – Less likely to have unfilled VS waves
  - No PS bound – No parameter cache usage
- Requires that nothing after VS causes a stall
  - Parameter size <= 4 * PosSize
  - Pixels drain faster than they are generated
  - No scissoring occurs
- PA can receive work faster than VS can possibly generate it
  - Often see tessellation achieve peak VS primitive throughout; one SE at a time

Benchmarks that get very close to 2 prims/clock (around 1.97) have these characteristics:

VS reads nothing
VS writes only SV_Position
VS always outputs 0.0f for position - So every primitive is trivially culled
Index buffer is all 0's
So every vertex is a cache hit.
Cache hits don't count as verts for purposes of peak vertex rate.
That's the only way we can get near 2 prims/clock without hitting 2 vertices/clock first.
Every instance is a multiple of 64 vertices - Makes unfilled VS waves less likely
No PS bound - So no parameter cache usage

The peak primitive rate also requires that nothing after VS causes a stall.
- ParamSize <= 4 * PosSize
- Pixels drain faster than they are generated
- No scissoring occurs

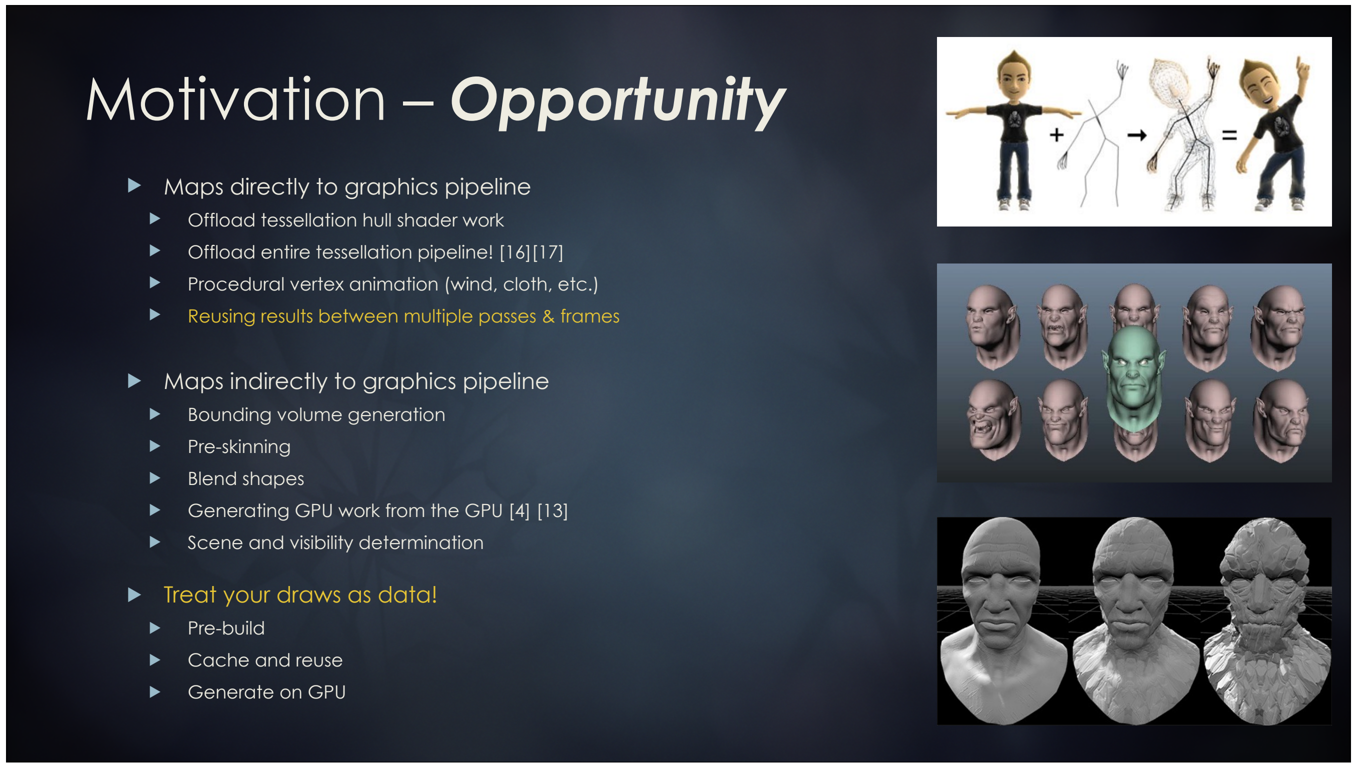
Apart from that, the PA can receive work faster than VS can possibly generate it. We often see tessellation achieve peak VS primitive throughput - for one SE at a time.


Motivation – ***Opportunity***

- Coarse cull on CPU, refine on GPU
- Latency between CPU and GPU prevents optimizations
- GPGPU Submission!
  - Depth-aware culling
    - Tighten shadow bounds \ sample distribution shadow maps [21]
    - Cull shadow casters without contribution [4]
    - Cull hidden objects from color pass
  - VR late-latch culling
    - CPU submits conservative frustum and GPU refines
  - Triangle and cluster culling
    - Covered by this presentation

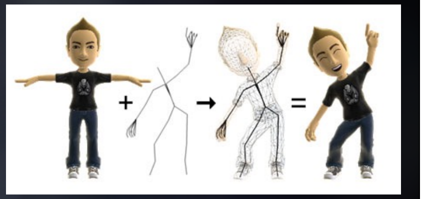
Engines typically do various methods of coarse culling on the CPU, prior to GPU submission. Due to latency between CPU and GPU, many optimizations are inappropriate, or It would mean tight lock stepping. The CPU is a limited resource on consoles, and this isn't a great use of a CPU core.

On PC you have to get the data over PCIE which would be prohibitive. Because of this, we want the culling to happen on the GPU's timeline, so the solution is to do GPGPU submission. GPU based approaches include depth-aware culling, VR late-latch culling, or triangle and cluster culling, which is covered by this presentation.



Motivation – ***Opportunity***

- Maps directly to graphics pipeline
  - Offload tessellation hull shader work
  - Offload entire tessellation pipeline! [16][17]
  - Procedural vertex animation (wind, cloth, etc.)
  - Reusing results between multiple passes & frames
- Maps indirectly to graphics pipeline
  - Bounding volume generation
  - Pre-skinning
  - Blend shapes
  - Generating GPU work from the GPU [4] [13]
  - Scene and visibility determination
- Treat your draws as data!
  - Pre-build
  - Cache and reuse
  - Generate on GPU

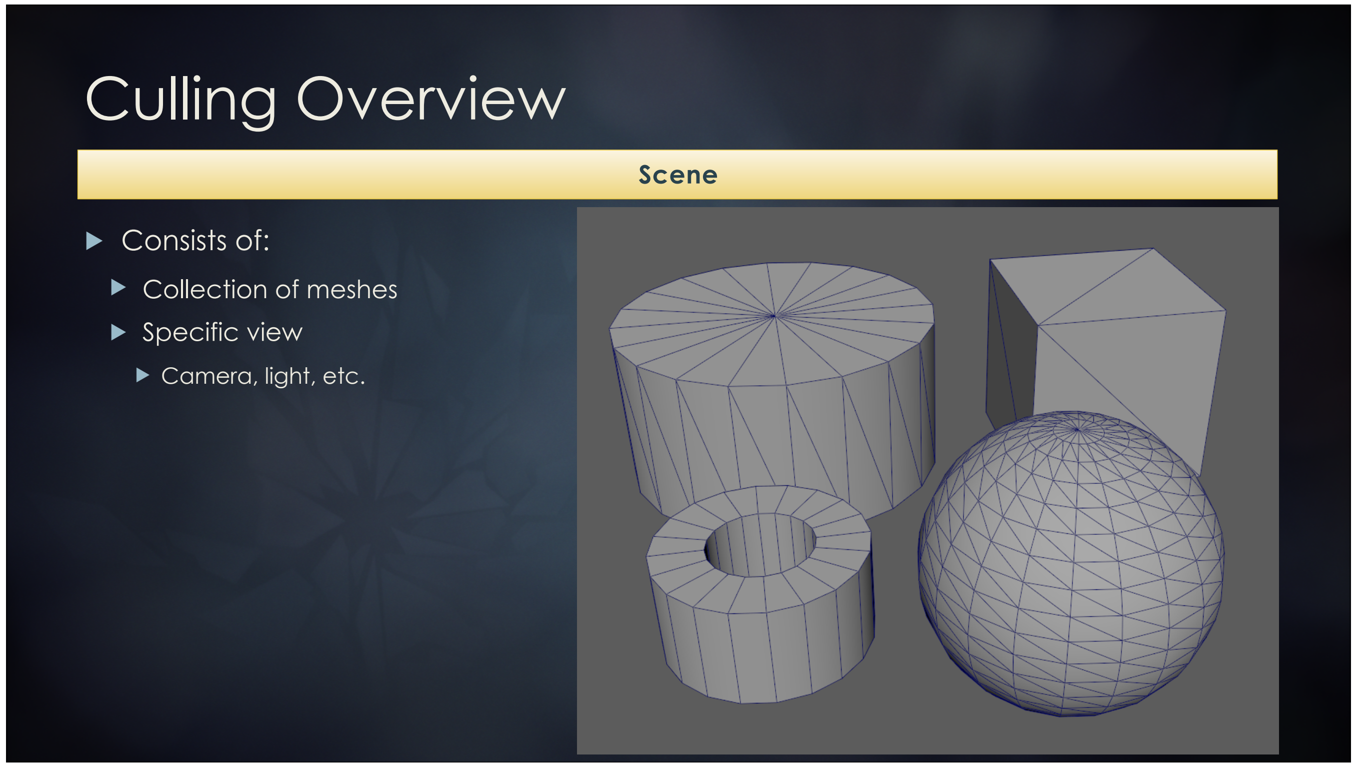


Compute shader mesh processing opens up opportunities for more efficiently supporting a variety of high fidelity features and improvements. Better yet, by reusing post-shader results between multiple passes, and doing less draw setup work on the CPU, there is increased optimization potential.

The mantra is to treat all your draws as regular data. The data can be pre-built, cached and reused, and generated on the GPU. This approach allows us increased flexibility, including the ability to work around various fixed function bottlenecks.

Culling Overview

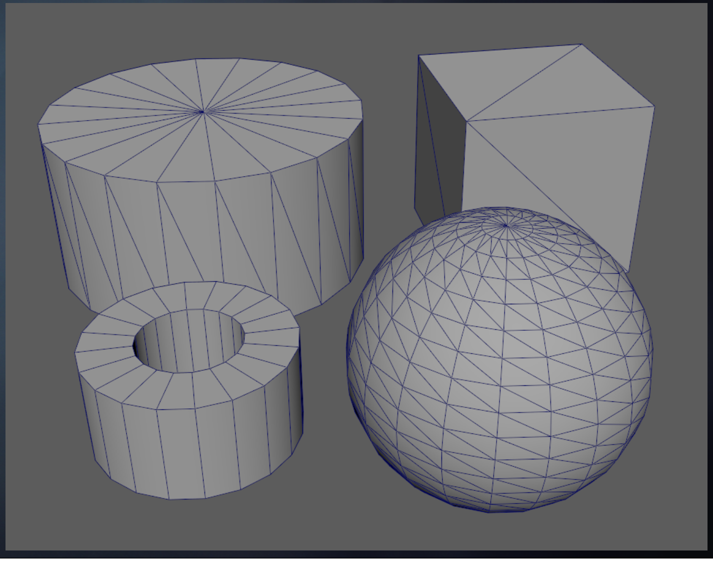
I’m going to start off by giving an overview of the culling methods



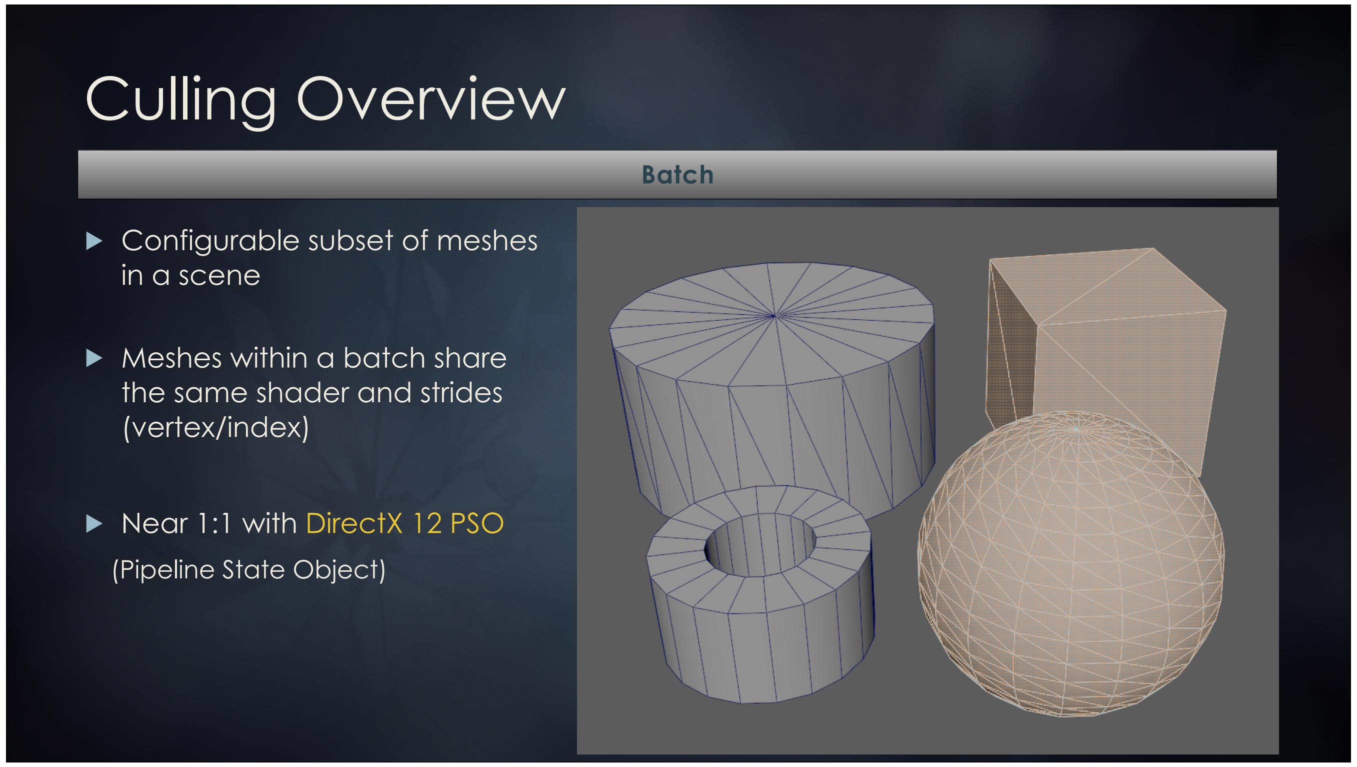
Culling Overview

Scene

- Consists of:
  - Collection of meshes
  - Specific view
    - Camera, light, etc.



Lets first define some terms to reduce confusion. A scene consists of a collection of meshes, displayed from a specific view

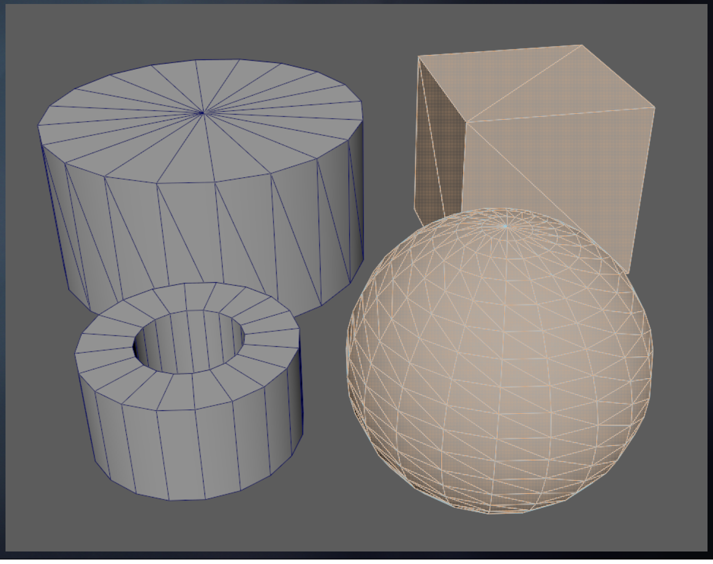


Culling Overview

Batch

- Configurable subset of meshes in a scene
- Meshes within a batch share the same shader and strides (vertex/index)
- Near 1:1 with DirectX 12 PSO

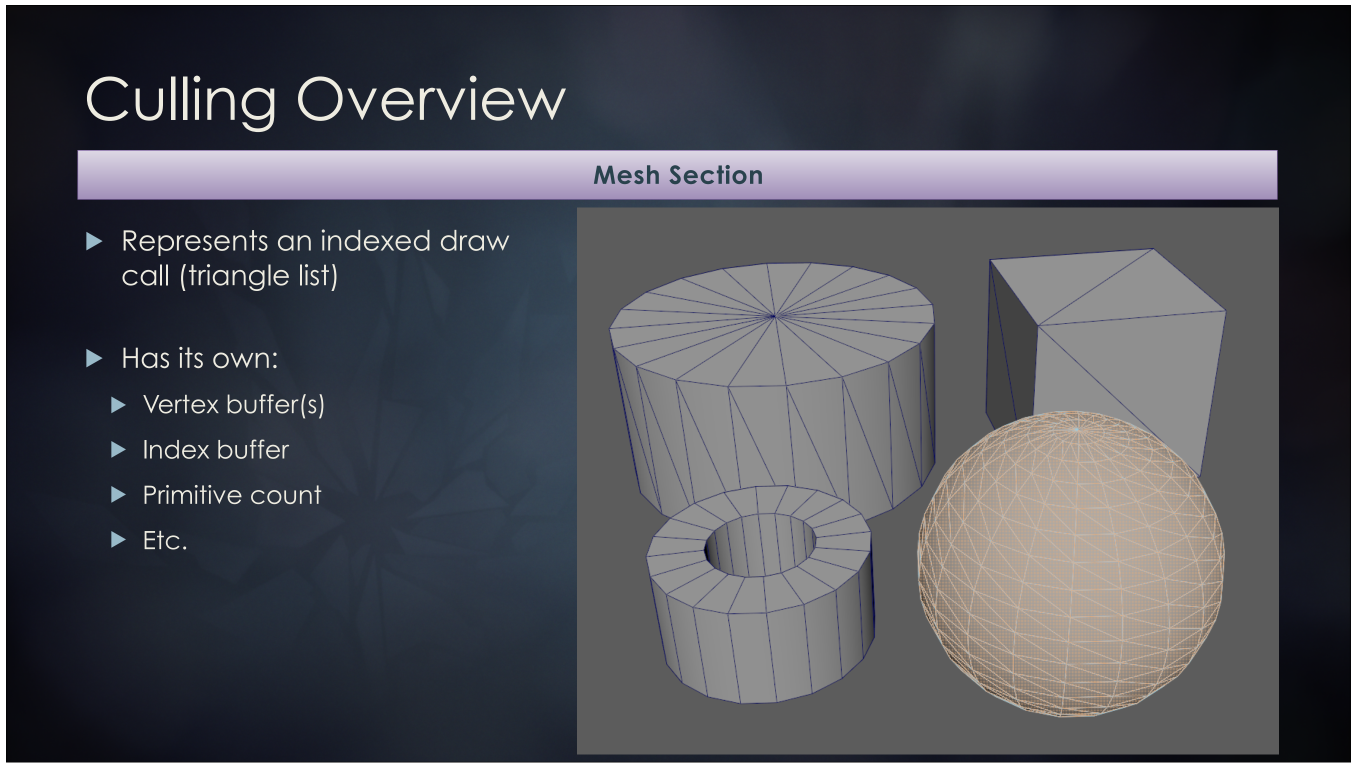
  (Pipeline State Object)



Then we have a batch, which is a configurable subset of meshes in a scene.

Except on Xbox One, we require all meshes in a batch to share the same shader, and also that all meshes share the same vertex and index strides.

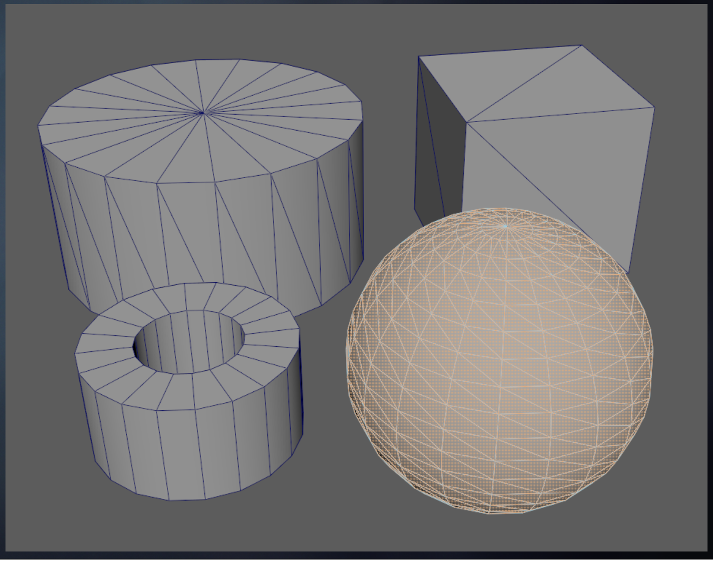
These requirements are due to the way that GPU driven rendering works currently, at least on PC. A batch here can be thought of as a near 1 to 1 with DirectX 12's Pipeline State Object concept, or PSO.



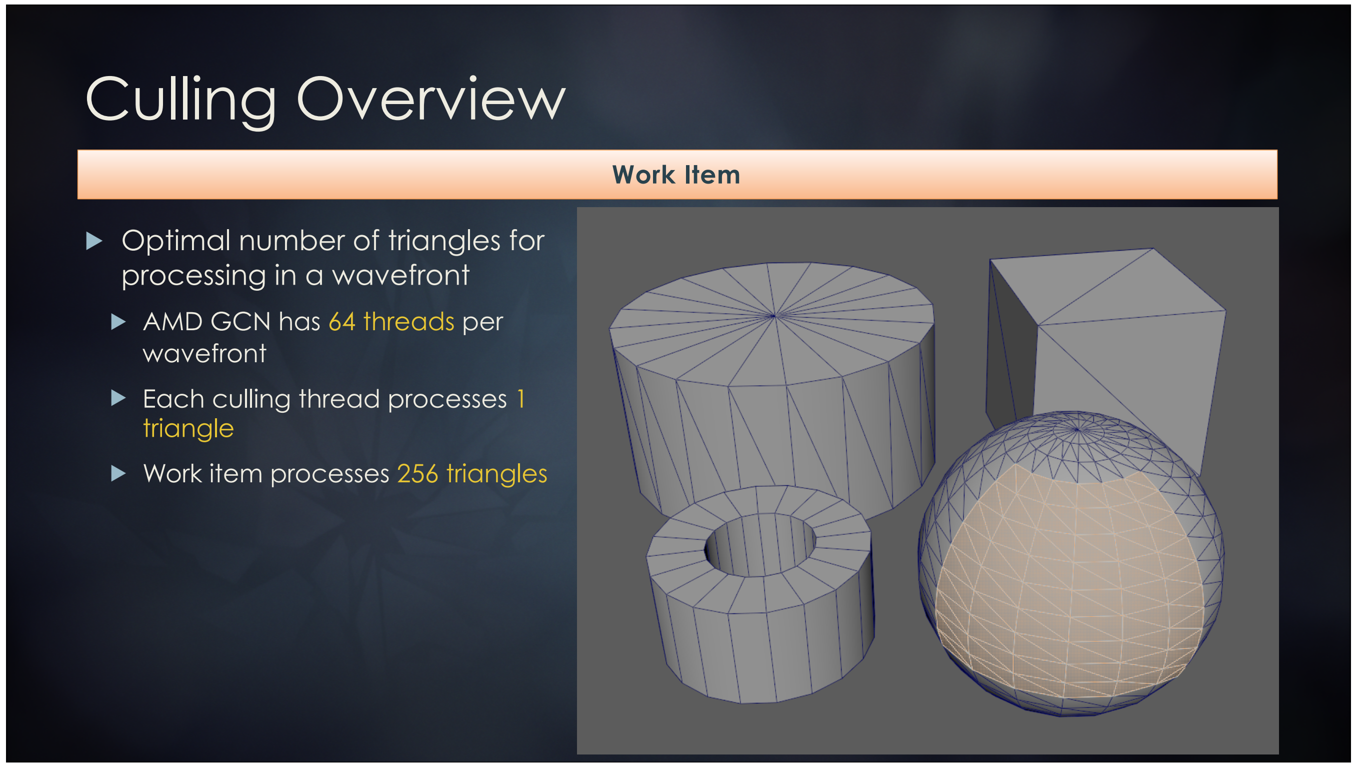
Culling Overview

Mesh Section

- Represents an indexed draw call (triangle list)
- Has its own:
  - Vertex buffer(s)
  - Index buffer
  - Primitive count
  - Etc.



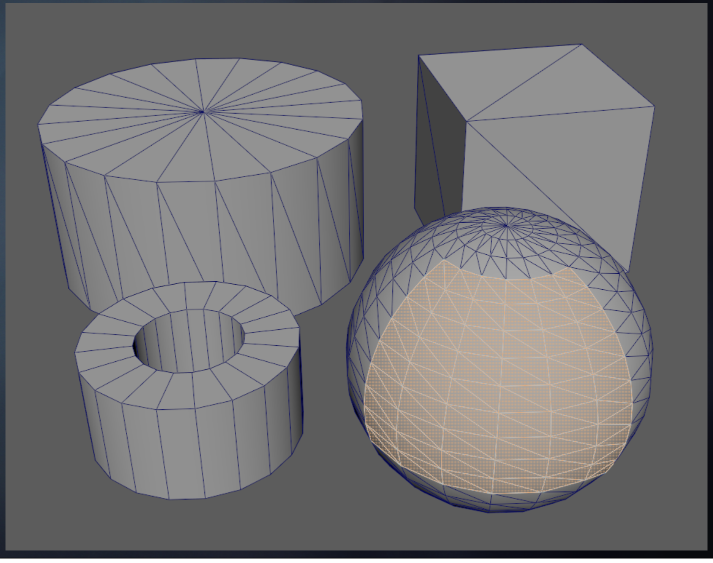
We also have a mesh section, which represents an indexed draw call. A mesh section has its own vertex buffers, index buffer, primitive count, and other properties.



Culling Overview

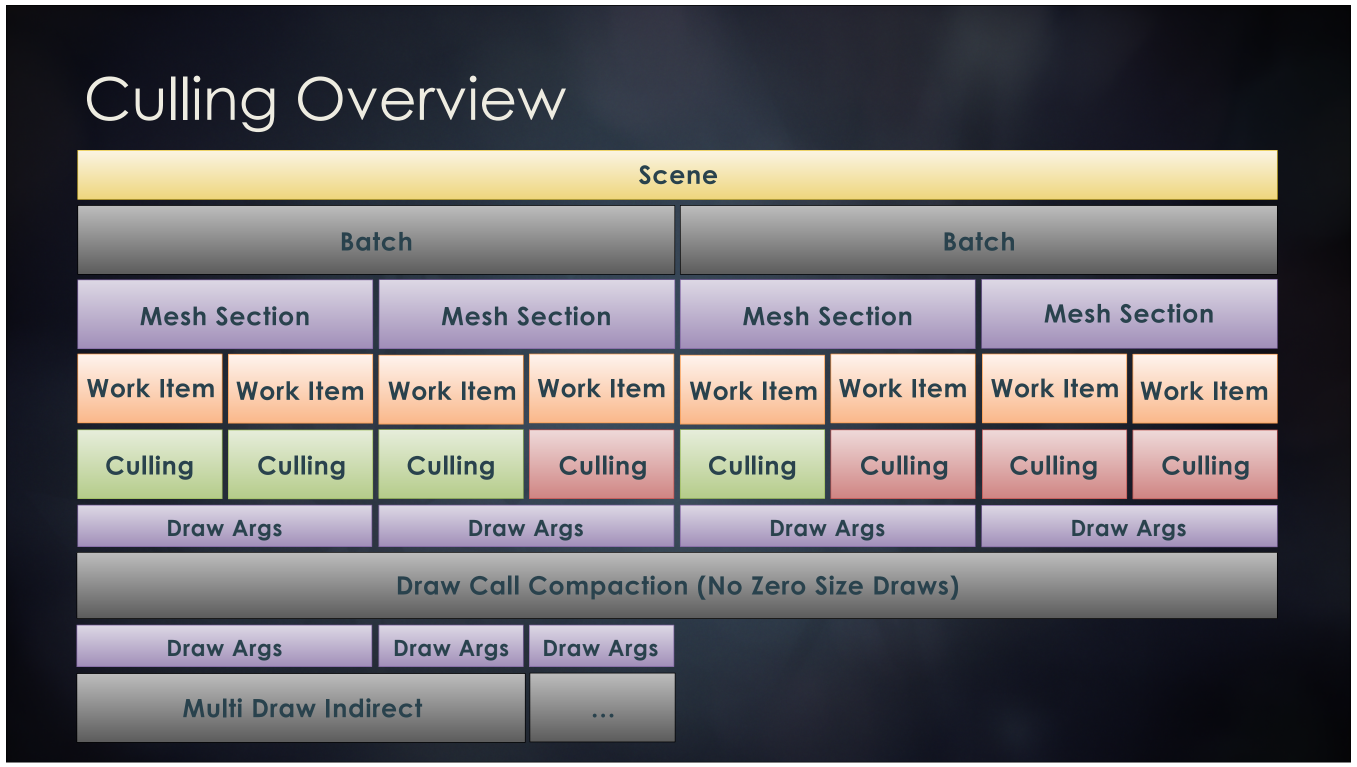
Work Item

- Optimal number of triangles for processing in a wavefront
  - AMD GCN has 64 threads per wavefront
  - Each culling thread processes 1 triangle
  - Work item processes 256 triangles



Lastly, we have a work item, which represents a subset of triangles in a batch that will be processed by a compute shader wavefront.

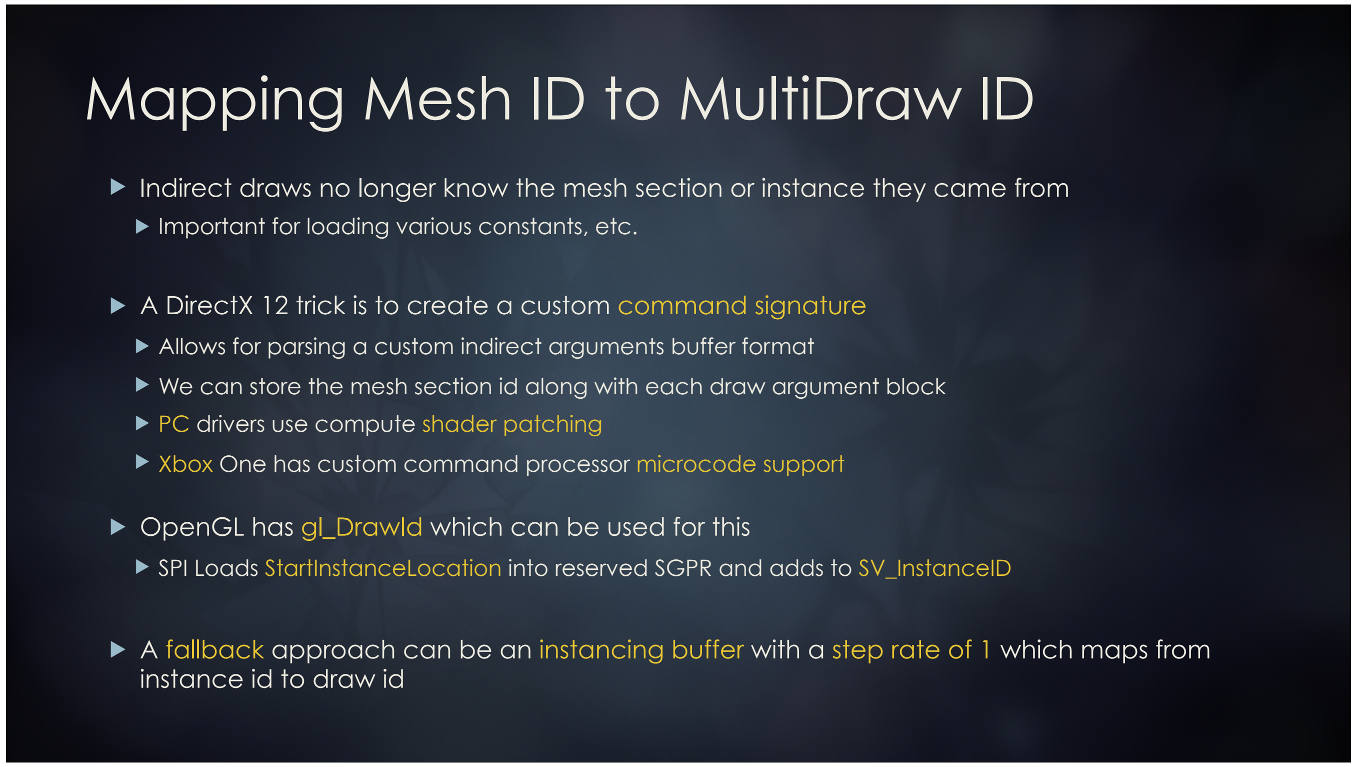
The number of triangles has been chosen based on the underlying hardware, and characteristics of the algorithm. AMD GCN has 64 threads per wavefront (which includes both consoles), each culling thread processes 1 triangle, and each work item processes 256 triangles.


Culling Overview

Here is a high level overview of how a scene breaks down into work items that first undergo coarse view culling, and then surviving clusters undergo triangle culling, with a variety of tests. We run a quick compaction pass that ensures we do not have zero size draws if a mesh section is entirely culled (like in the case of occlusion or frustum culling).

At the end of the pipeline, we have a group of indexed draw arguments that we can kick from the GPU with ExecuteIndirect on DirectX 12, or via the AMD_multi_draw_indirect extension on OpenGL.

On Xbox One, ExecuteIndirect has some incredible extensions where PSOs can be switched by indirect arguments, meaning we can issue a single ExecuteIndirect for our entire scene, regardless of state or resource changes.


Mapping Mesh ID to MultiDraw ID

- Indirect draws no longer know the mesh section or instance they came from
  - Important for loading various constants, etc.
- A DirectX 12 trick is to create a custom command signature
  - Allows for parsing a custom indirect arguments buffer format
  - We can store the mesh section id along with each draw argument block
  - PC drivers use compute shader patching
  - Xbox One has custom command processor microcode support
- OpenGL has gl_DrawId which can be used for this
  - SPI Loads StartInstanceLocation into reserved SGPR and adds to SV_InstanceID
- A fallback approach can be an instancing buffer with a step rate of 1 which maps from instance id to draw id

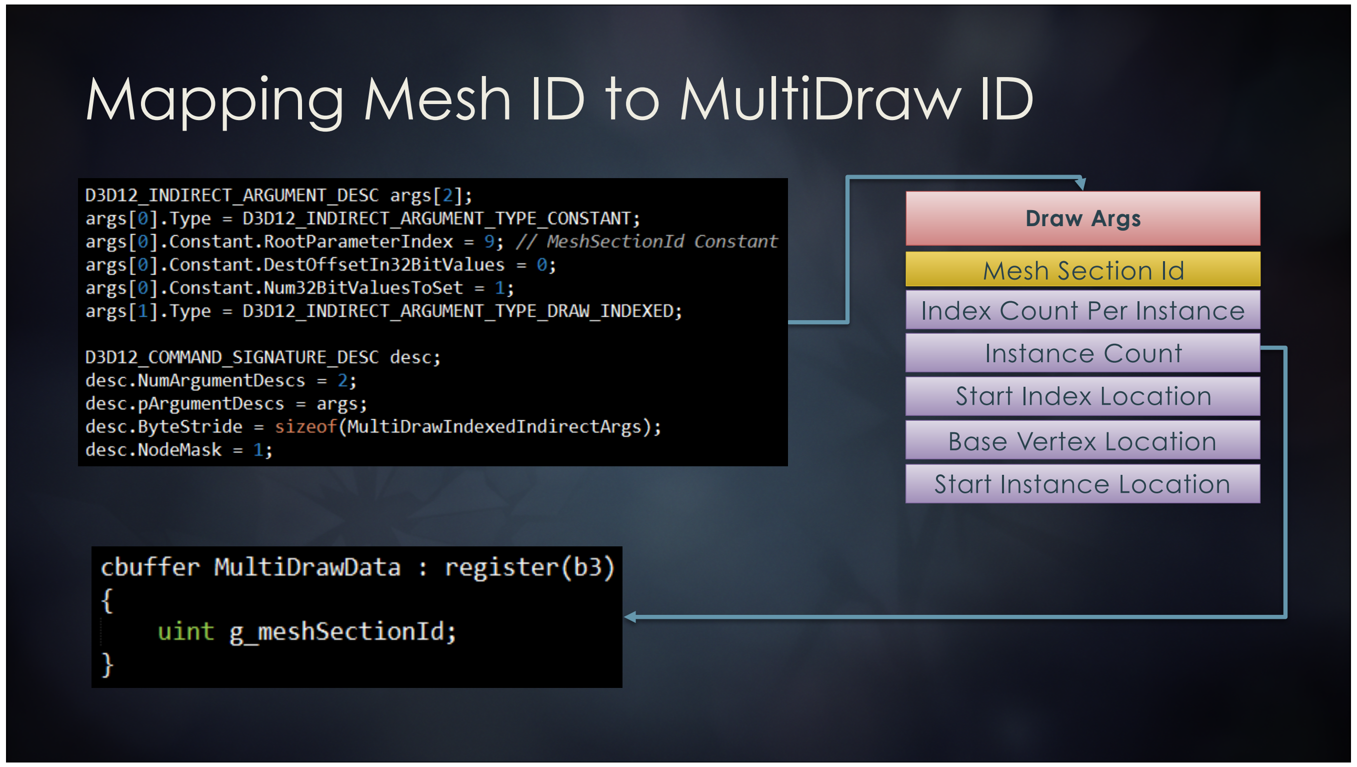
Constructing each draw argument block is fairly straight forward, it's mostly a matter of determining what starting index and count each block is responsible for during rendering.

However, things get complicated when you try to load constants or other resource data from a regular vertex or pixel shader, unaware that this culling pass has occurred. In order to avoid state changes, we have an instancing buffer that contains the transforms, colors, etc. per instance, but in this case it's no longer one to one with a draw call. Essentially, we need to add a custom 32 bit word to the argument buffer that tracks what original draw index it is associated with.

A DirectX 12 trick is to create a custom command signature. Doing so allows for parsing a custom indirect arguments buffer format, where we can store a custom id packed alongside the other hardcoded draw indexed argument values.

On PC, drivers use compute shader patching, where the id is loaded into a register for a shader to reference per invocation. On OpenGL, you can use gl_DrawId for this purpose. The command processor microcode on Xbox One handles indirect draws without intermediate steps or patching, which is extremely optimal.

An alternative would be to bind a buffer with per-instance step rate of 1 which maps from instance id to draw id. Depending on the driver implementation, this might be faster than the root constant approach for the time being while drivers mature.



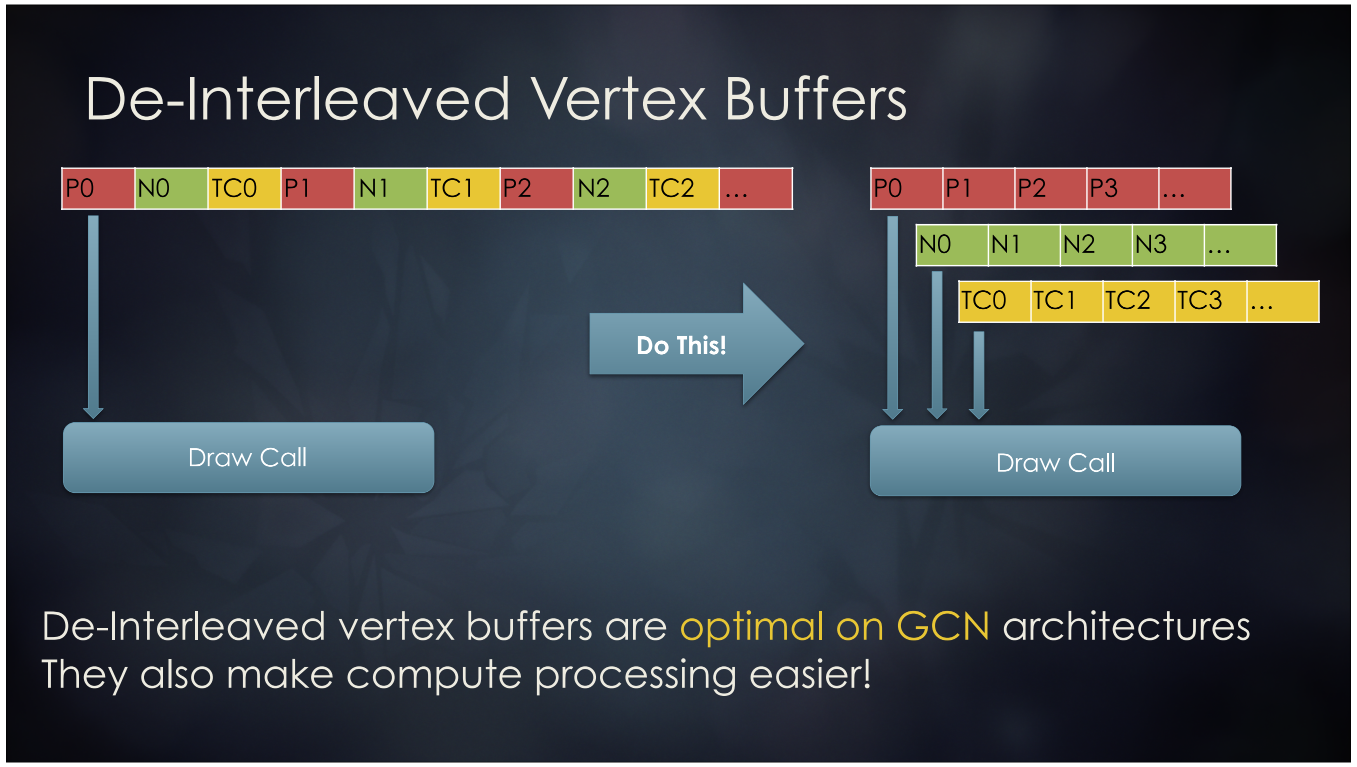
Mapping Mesh ID to MultiDraw ID

```
D3D12_INDIRECT_ARGUMENT_DESC args[2];
args[0].Type = D3D12_INDIRECT_ARGUMENT_TYPE_CONSTANT;
args[0].Constant.RootParameterIndex = 9; // MeshSectionId Constant
args[0].Constant.DestOffsetIn32BitValues = 0;
args[0].Constant.Num32BitValuesToSet = 1;
args[1].Type = D3D12_INDIRECT_ARGUMENT_TYPE_DRAW_INDEXED;

D3D12_COMMAND_SIGNATURE_DESC desc;
desc.NumArgumentDescs = 2;
desc.pArgumentDescs = args;
desc.ByteStride = sizeof(MultiDrawIndexedIndirectArgs);
desc.NodeMask = 1;
```

```
cbuffer MultiDrawData : register(b3)
{
    uint g_meshSectionId;
}
```

Here you can see the appropriate command signature description to define the custom format that's displayed on the right. Argument 0 defines the 32 bit mesh section id, including the parameter index into the root signature. Argument 1 then follows, which is the fixed list of 5 arguments that make up a draw indexed packet. This mapping will cause the 0th word of your argument block to be loaded into an SGPR register for use by the shader.

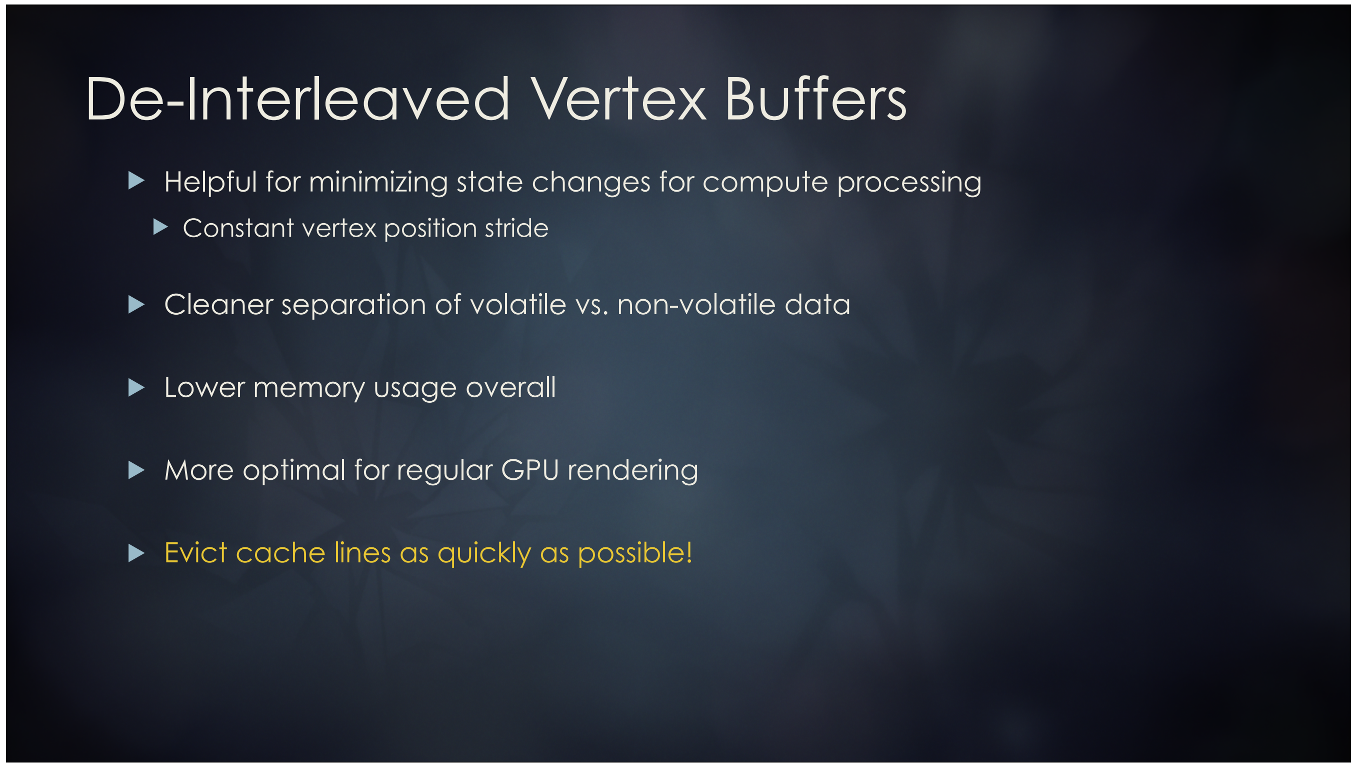
On PC, having a command signature with complex commands will cause ExecuteIndirect processing to go through a compute shader. However, having a single extra word to represent the draw id will remain on a fast path – similar to AGS MultiDrawIndirect or gl_DrawId.



De-Interleaved Vertex Buffers

De-Interleaved vertex buffers are optimal on GCN architectures
They also make compute processing easier!

Another architectural note is that we have de-interleaved our vertex buffers. This can be a substantial win on GCN architectures, and it also makes the task of compute mesh processing much easier.



De-Interleaved Vertex Buffers

- Helpful for minimizing state changes for compute processing
  - Constant vertex position stride
- Cleaner separation of volatile vs. non-volatile data
- Lower memory usage overall
- More optimal for regular GPU rendering
- Evict cache lines as quickly as possible!

There are a number of reasons that de-interleaving your vertex data is beneficial. In terms of compute processing performance, having culling data like position in its own stream away from other attributes like UVs, colors, TSB, etc. means that we have an almost never changing stride. The only time you would need to break batching would be 16 bit vs 32 bit precision.

Consoles and DirectX 12 placement resources can be spanned across all the geometry data, meaning that with a constant stride, and some pointer arithmetic to determine the right start vertex and index location for each draw, we can completely avoid binding varying buffers throughout rendering!

In addition to algorithmic benefits, de-interleaving your vertex data is more optimal for regular GPU rendering on GCN architectures, so there's really no excuse. We want to evict cache lines as quickly as possible. With interleaved data, the cache line needs to be kept between the first and the last read. With de-interleaved data and inlined fetch shaders, the wavefront fetches a cache line, consumes it, and it throws the cache line away immediately.

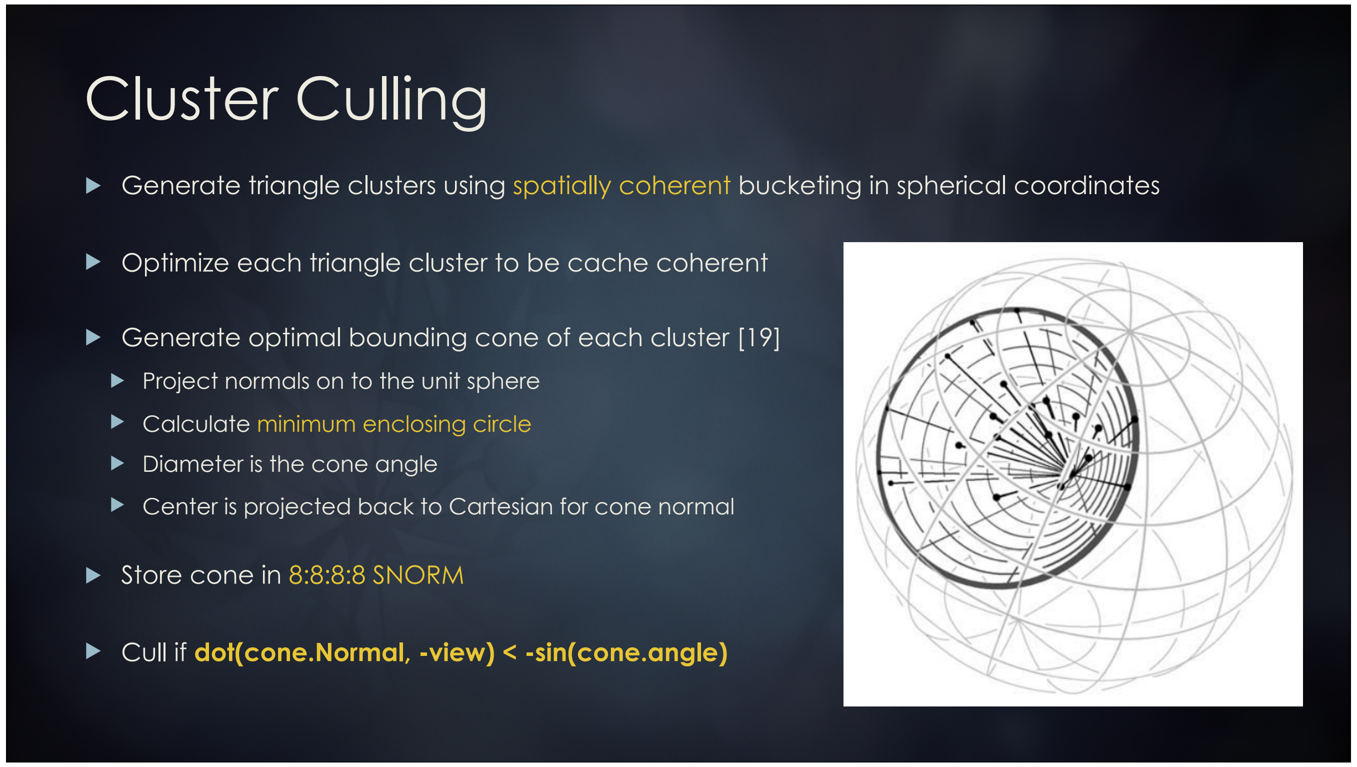
An additional benefit, is that de-interleaving delivers faster processing on the CPU, as the data will be SoA instead of AoS, making it easier to process with SSE/AVX, etc., and the same advantage applies on the GPU.

If you want to be the most optimal across mobile, AMD and other IHVs, it is common to at least have multiple interleaved streams of mutable vs immutable data, positions in their own streams (optionally with UVs in the case of alpha tested shadows), skinning data, and other common data grouped together.

Another advantage of de-interleaved vertex buffers is that you can create separate index buffers per pass. A depth-only pass (like for culling) can have more vertex re-use than a full pass, because you often need to duplicate vertices for full rendering (same position but different texcoord, or same position but different normal).

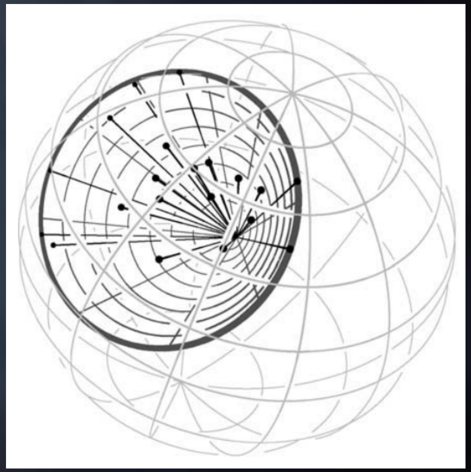
Cluster Culling

Before getting into the per triangle culling, it’s important to touch on coarse culling of triangle clusters



Cluster Culling

- Generate triangle clusters using spatially coherent bucketing in spherical coordinates
- Optimize each triangle cluster to be cache coherent
- Generate optimal bounding cone of each cluster [19]
  - Project normals on to the unit sphere
  - Calculate minimum enclosing circle
  - Diameter is the cone angle
  - Center is projected back to Cartesian for cone normal
- Store cone in 8:8:8:8 SNORM
- Cull if **dot(cone.Normal, -view) < -sin(cone.angle)**



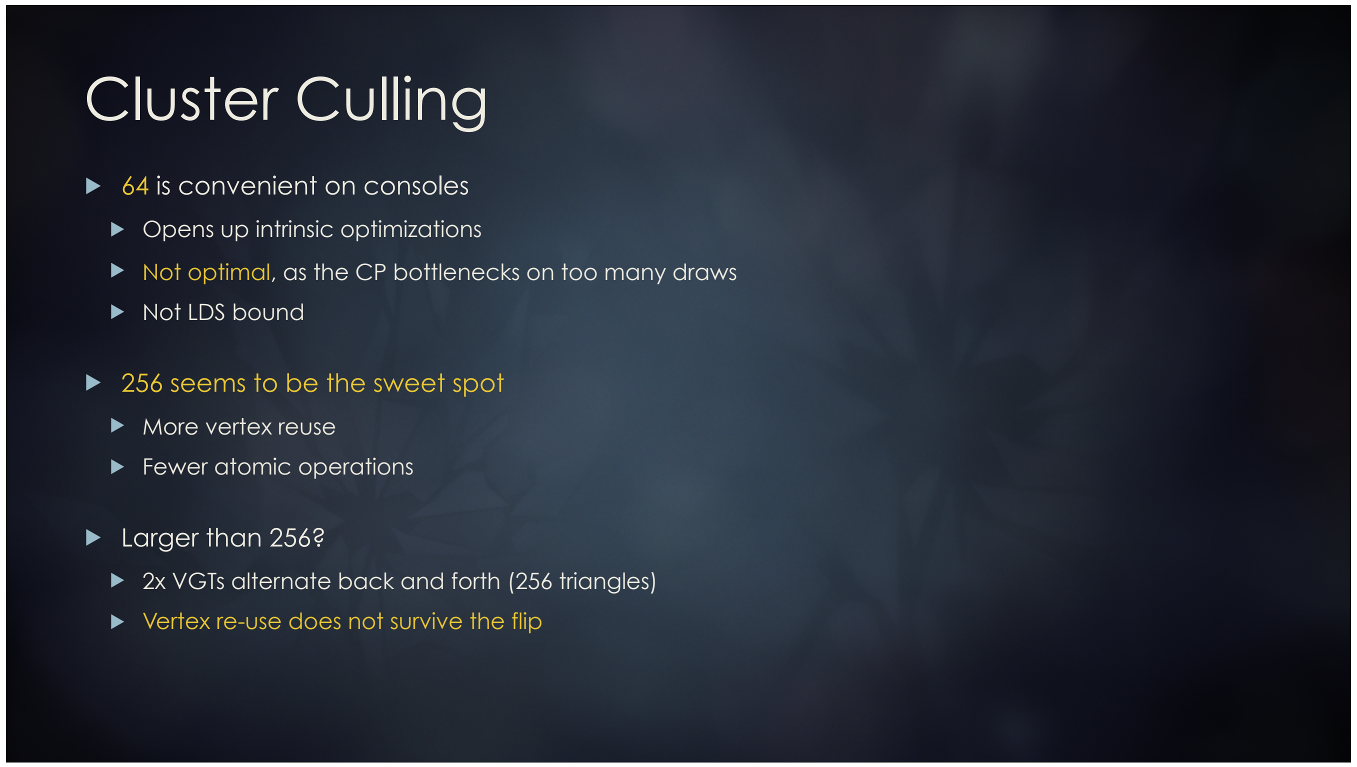
In order to make efficient use of the GPU, we first do a coarse GPU culling pass of our mesh data. An offline process partitions meshes into 256 triangle clusters using a greedy spatially and cache coherent bucketing algorithm. For each cluster, we generate an optimal bounding cone.

The general idea is to project each triangle normal on to the unit sphere, and take this 256 projected normal collection and calculate a minimum enclosing circle against the phi and theta pairs. Since the algorithm is operating on a difference of angles, we can use the circle diameter as the cone angle, and project the center back to Cartesian for the cone normal.

<Commence primary ignition>

4 component 8bit SNORM has enough precision to store this cone, which can be culled on the GPU by taking the dot product of the cone normal and a conservative cluster-centroid view vector and comparing it to the negative sine cone angle. The obvious optimization is to store the cone angle with the negative sine calculated in to the value.

You'll want to make an allowance for rounding, like slightly enlarging the cone angle to avoid false rejection. The cone normal will quantize as well. Any of the gbuffer encoding to improve normal accuracy would be applicable here, as long as they are not the ones that bias depth precision towards viewer facing.



Cluster Culling

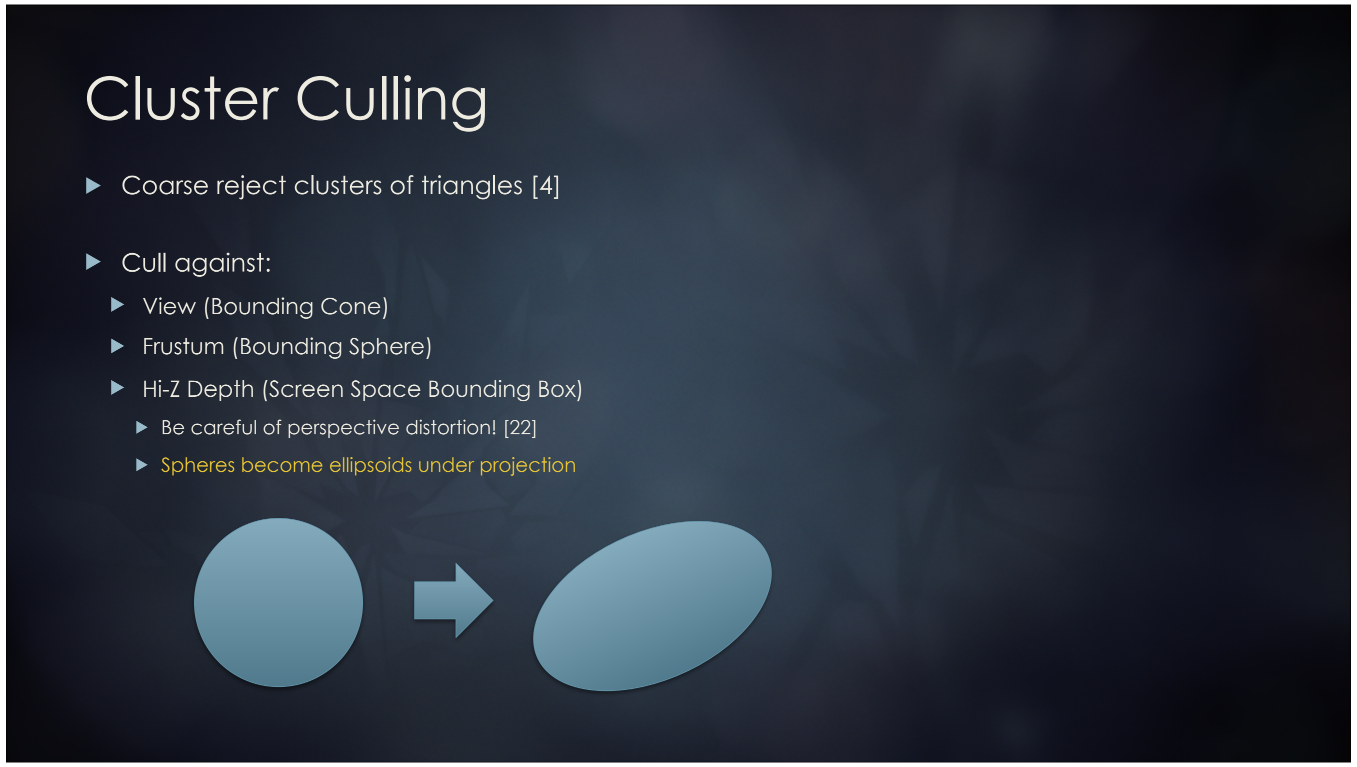
- 64 is convenient on consoles
  - Opens up intrinsic optimizations
  - Not optimal, as the CP bottlenecks on too many draws
  - Not LDS bound
- 256 seems to be the sweet spot
  - More vertex reuse
  - Fewer atomic operations
- Larger than 256?
  - 2x VGTs alternate back and forth (256 triangles)
  - Vertex re-use does not survive the flip

For determining the ideal cluster size, I did a lot of profiling of various configurations.

64 is a convenient size on consoles, as this opens up intrinsic optimizations. However, I found this to be sub-optimal, as the CP bottlenecks on too many draws, and we were never bound by LDS atomics.

Based on profiling, a cluster size of 256 seems to be the sweet spot.

The 2 VGTs flip back and forth every 256 triangles, and vertex re-use does not survive the flip, making a cluster size of 256 a wise choice.



Cluster Culling

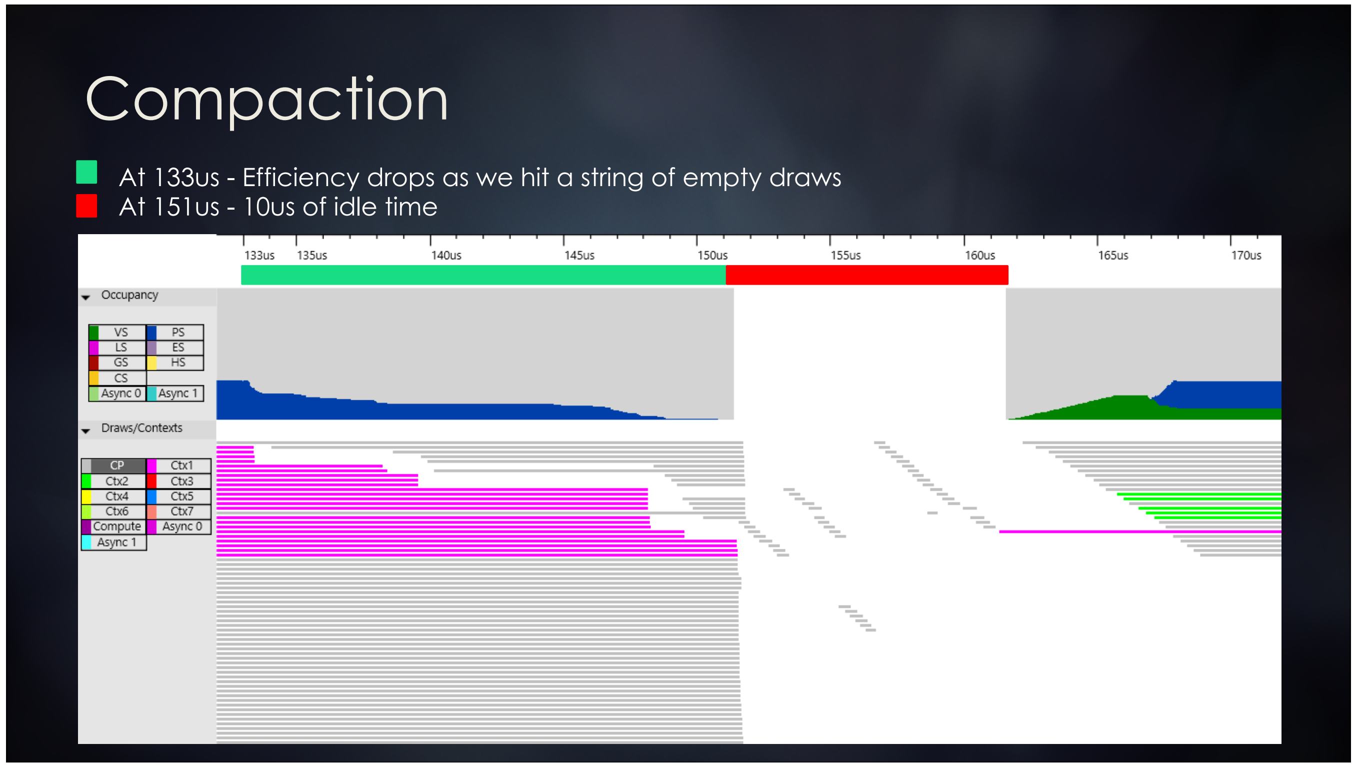
- Coarse reject clusters of triangles [4]
- Cull against:
  - View (Bounding Cone)
  - Frustum (Bounding Sphere)
  - Hi-Z Depth (Screen Space Bounding Box)
    - Be careful of perspective distortion! [22]
    - Spheres become ellipsoids under projection

This approach allows for us to coarse reject entire clusters of triangles prior to the per triangle culling pass. Additional per cluster tests include bounding sphere vs frustum, and testing the bounding sphere's screen space bounding box against a Hi-Z depth pyramid. Be careful that you account for perspective distortion, as spheres become ellipsoids under projection.

I won't go into further detail on cluster culling, as it's covered in great detail in the excellent GPU-Driven Rendering Pipelines presentation [4] from SIGGRAPH last year.

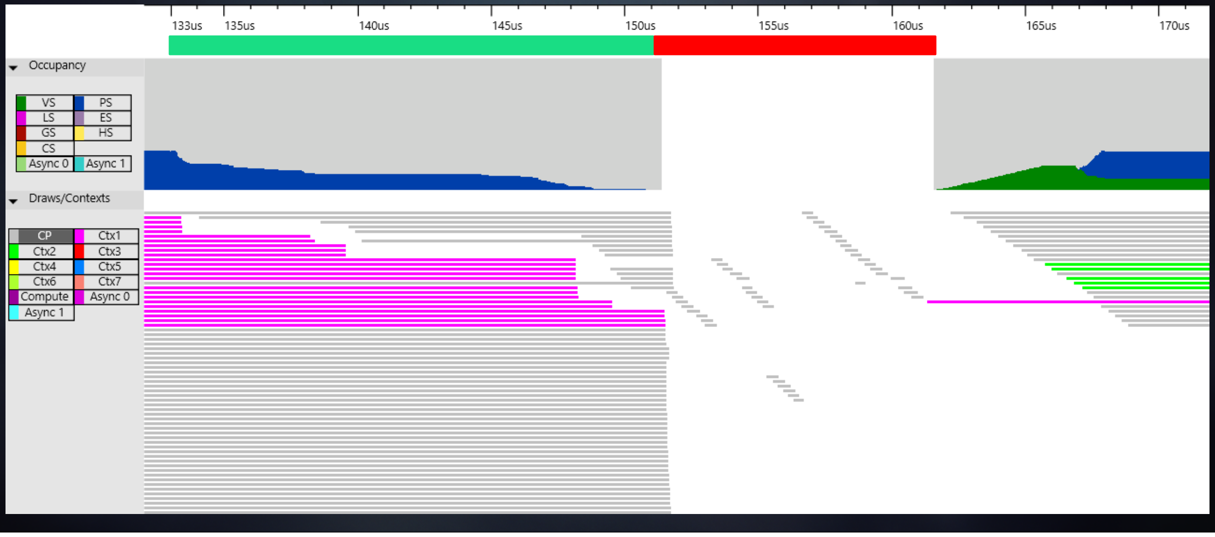
Draw Compaction

With cluster and triangle culling of draws on the GPU, it’s extremely important to remove zero sized draws from submission.



Compaction

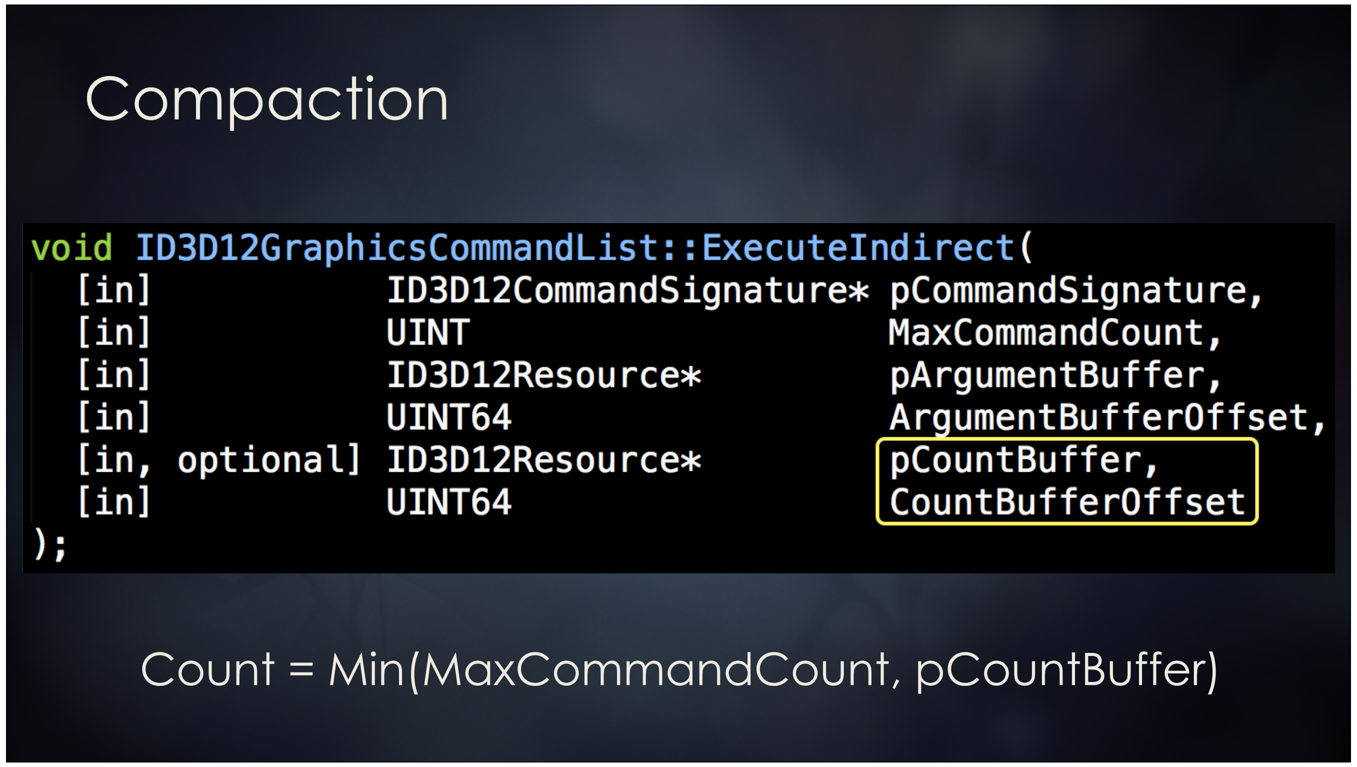
🟩 At 133us - Efficiency drops as we hit a string of empty draws
🟥 At 151us - 10us of idle time



The grey draws in this GPU capture are empty draw indirects. At first, the command processor (CP) cost is hidden by in-flight draws. Around 133us, the efficiency drops as we hit a string of empty draws. At 151us, we suffer around 10us of idle time.

However, the total impact is worse than 10us, since the GPU doesn't instantly resume 100% efficiency, as it takes time to fill the CUs with waves. Clusters of culled draws can easily overwhelm the command processor, which is potentially 1.5ms in a 60hz frame (seen in actual shipped games with real content).

While the savings from GPU culling still exceeds this cost, it is very important that we compact zero size draws in order to get the biggest gain. Even with 0 primitives, fetching indirect argument isn't free; there is a memory latency of ~300ns. The CP can hide a few of these in a row, but they add up. Additionally, the CP is consuming command buffer packets, and state changes aren't free.



Compaction

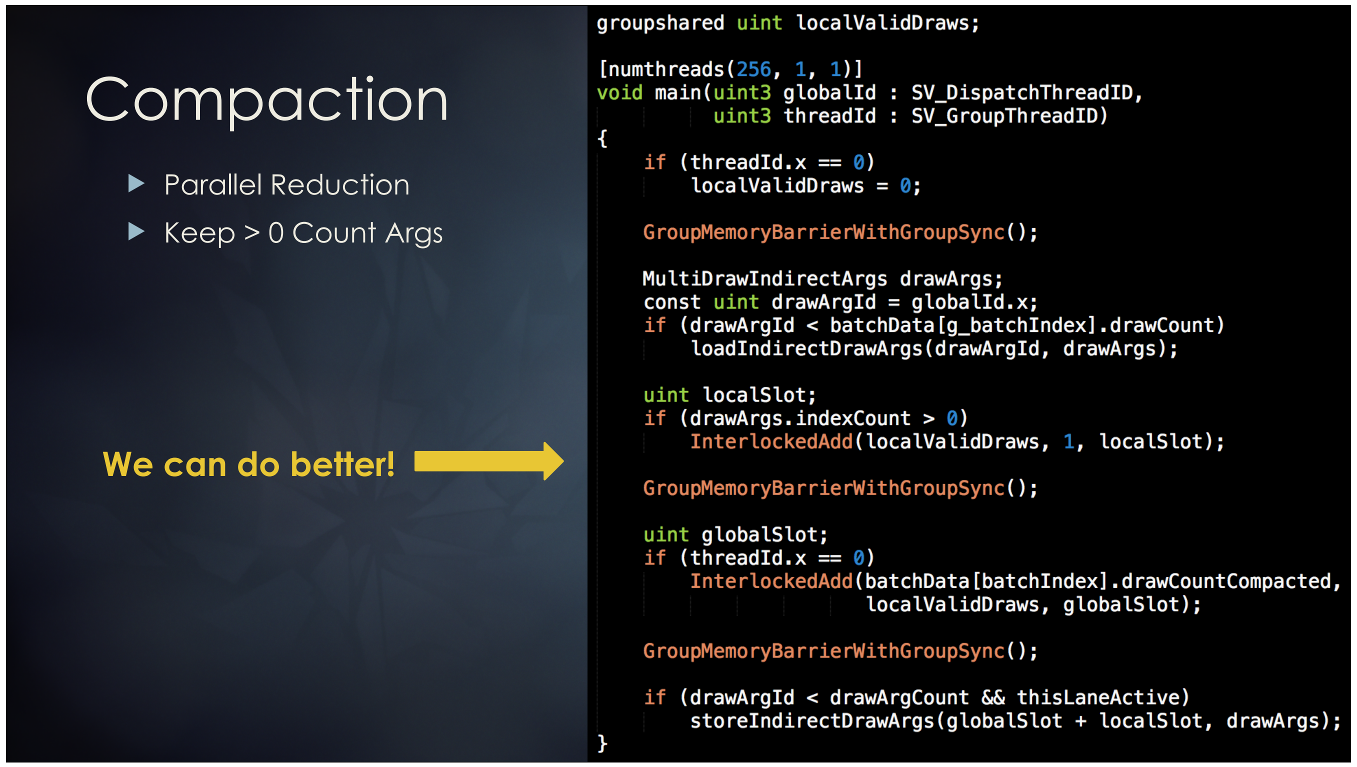
```
void ID3D12GraphicsCommandList::ExecuteIndirect(
  [in]            ID3D12CommandSignature* pCommandSignature,
  [in]            UINT                    MaxCommandCount,
  [in]            ID3D12Resource*         pArgumentBuffer,
  [in]            UINT64                  ArgumentBufferOffset,
  [in, optional]  ID3D12Resource*         pCountBuffer,
  [in]            UINT64                  CountBufferOffset
);
```

Count = Min(MaxCommandCount, pCountBuffer)

The CPU will issue the worst case number of draws, so zero size draws will cause the GPU to process indirect args even if they have zero surviving primitives. The GPU needs control over the draw count and state changes.

The ExecuteIndirect API in DirectX 12 has an optional count buffer and offset, which the GPU will use to clamp the upper bound of draws that the command processor will unroll.

Some IHVs currently patch this value with a compute shader, or run other sub-optimal paths. However, the feature is new, and widespread use will encourage IHVs to improve the drivers in this area.



Compaction

- Parallel Reduction
- Keep > 0 Count Args

We can do better!

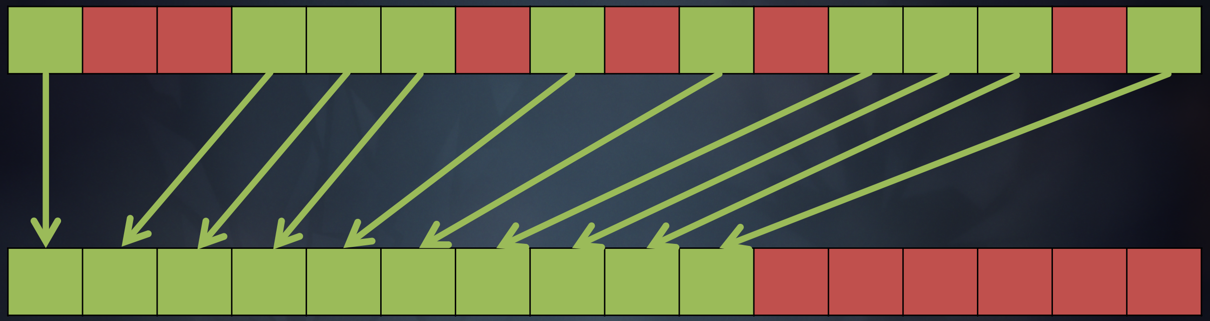
```
groupshared uint localValidDraws;

[numthreads(256, 1, 1)]
void main(uint3 globalId : SV_DispatchThreadID,
          uint3 threadId : SV_GroupThreadID)
{
    if (threadId.x == 0)
        localValidDraws = 0;

    GroupMemoryBarrierWithGroupSync();

    MultiDrawIndirectArgs drawArgs;
    const uint drawArgId = globalId.x;
    if (drawArgId < batchData[g_batchIndex].drawCount)
        loadIndirectDrawArgs(drawArgId, drawArgs);

    uint localSlot;
    if (drawArgs.indexCount > 0)
        InterlockedAdd(localValidDraws, 1, localSlot);

    GroupMemoryBarrierWithGroupSync();

    uint globalSlot;
    if (threadId.x == 0)
        InterlockedAdd(batchData[batchIndex].drawCountCompacted,
                       localValidDraws, globalSlot);

    GroupMemoryBarrierWithGroupSync();

    if (drawArgId < drawArgCount && thisLaneActive)
        storeIndirectDrawArgs(globalSlot + localSlot, drawArgs);
}
```

A cross platform approach to draw compaction is to do a parallel reduction with atomics in group shared memory. Each thread loads the indirect arguments for a draw and determines if the draw is worth keeping. A barrier allows all threads to complete and then the first thread in a group allocates output space for the surviving draw args.

Another barrier is performed so each thread gets the output location, and then the surviving draw args are written to the destination buffer. In this example, batchData is the ExecuteIndirect count buffer, and the offset is the location of drawCountCompacted.

With GCN intrinsics, and a thread group size of 64, we can do better!

Compaction

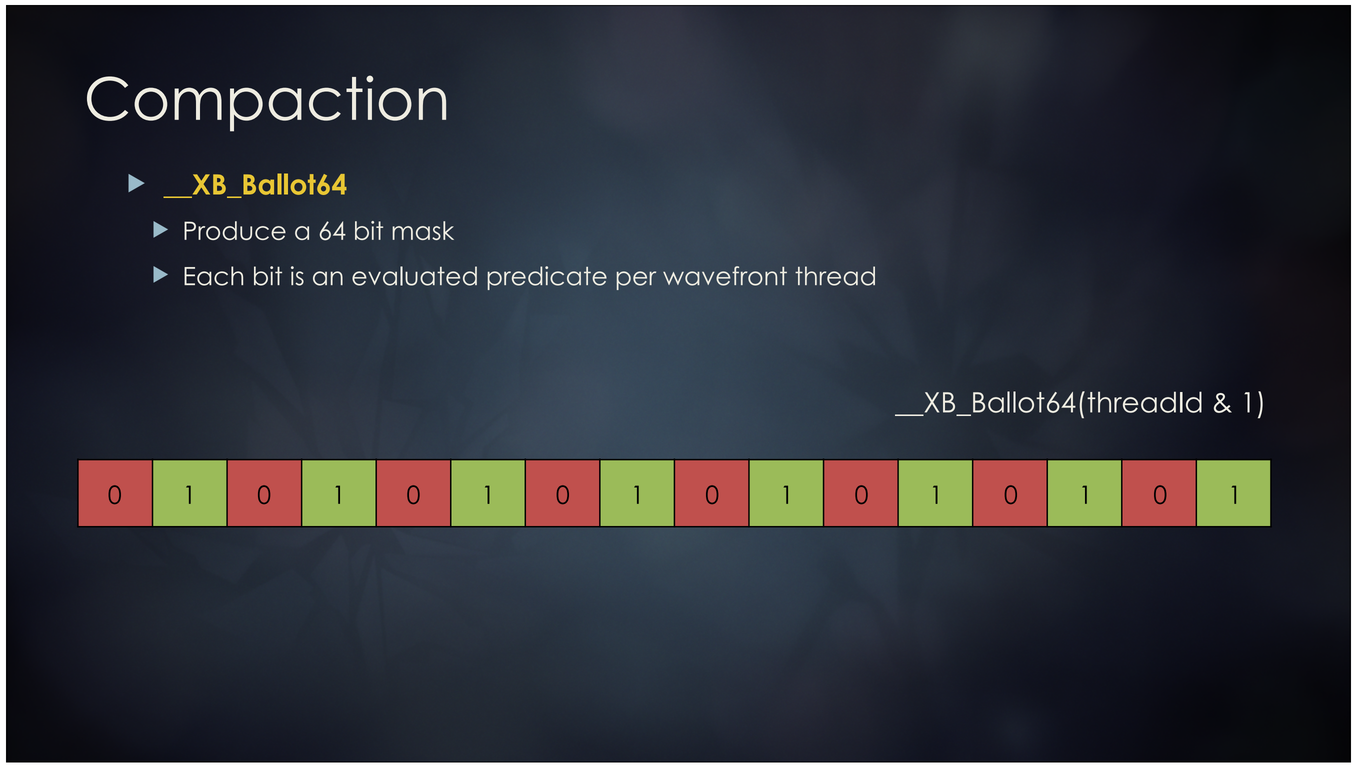


- Parallel prefix sum to the rescue!

🟩 0	🟥 0	🟥 0	🟩 1	🟩 2	🟩 3	🟥 3	🟩 4	🟥 4	🟩 5	🟥 5	🟩 6	🟩 7	🟩 8	🟥 8	🟩 9

The issue with optimizing the compaction is that each thread needs to write in a contiguous range, so using the thread id as the index wouldn't give us this, and we want to avoid global synchronization like the previous compaction algorithm.

This is where parallel prefix sum comes to the rescue! On the bottom range, you can see the indices we want computed per thread in order for each active thread to write into the correct contiguous slot.



Compaction

- **__XB_Ballot64**
  - Produce a 64 bit mask
  - Each bit is an evaluated predicate per wavefront thread

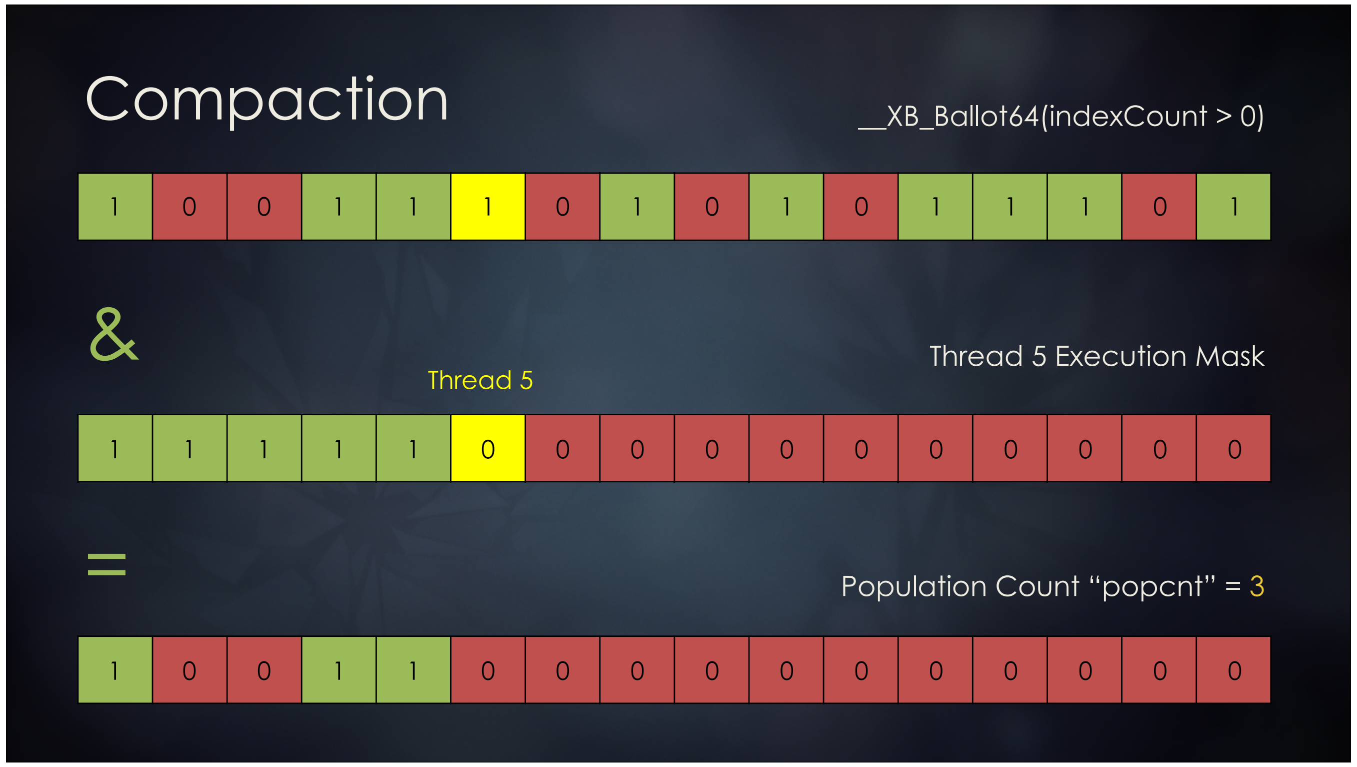
__XB_Ballot64(threadId & 1)

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The first thing to mention is a compiler intrinsic known as ballot. Ballot can be used to construct a 64 bit mask where each bit is an evaluated predicate per wavefront thread. For inactive threads, based on the execution mask, the bit will be 0 for these threads.

In this example, you can see a predicate that sets 0 for even threads, and 1 for odd threads.

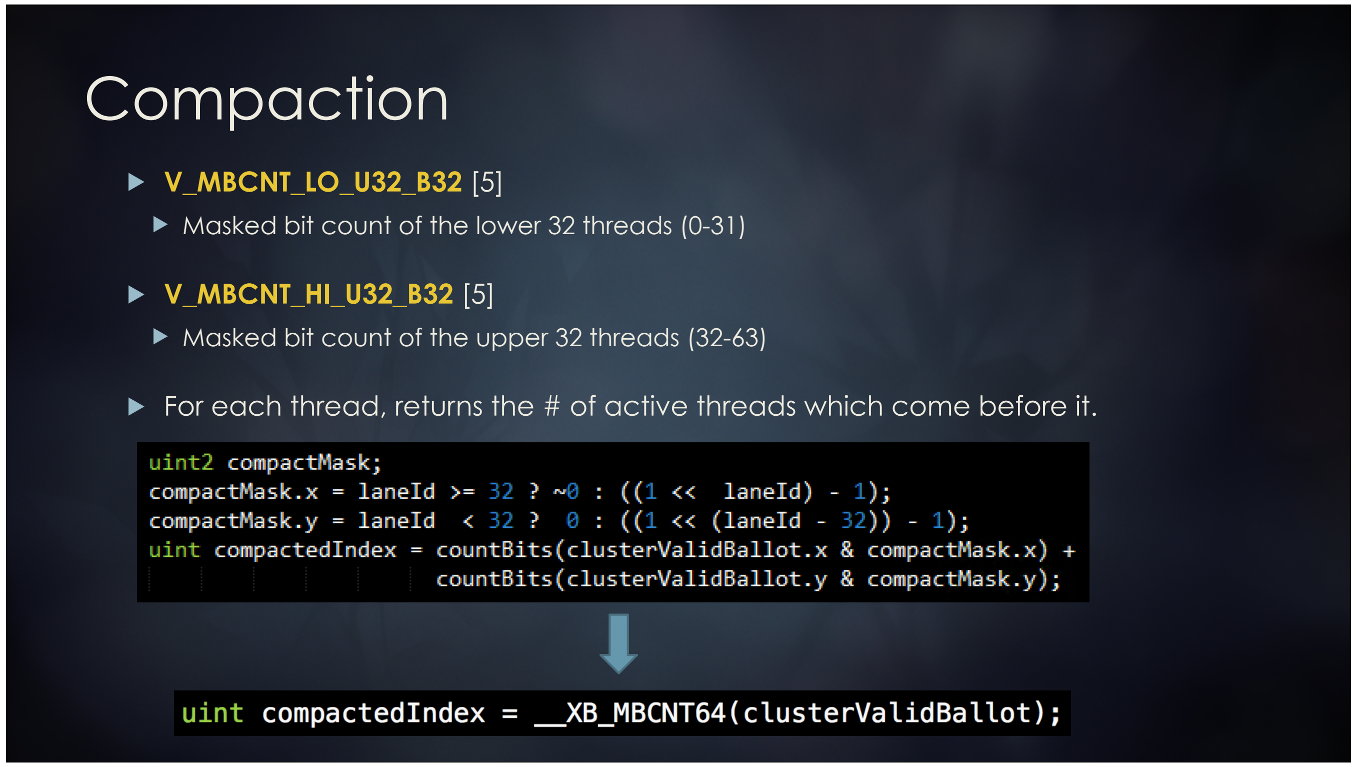
NV has a 32 wide NvBallot instruction available since Fermi:
https://www.opengl.org/registry/specs/NV/shader_thread_group.txt



Taking ballot further, we have this example showing thread 5. Before thread 5, we have three other threads that are valid, so we want to calculate the value 3 for thread 5's output slot.

By using ballot to generate a bit mask of surviving draw calls, we can & this mask against a thread execution mask where all bits are 0 except for threads lower than the current thread.

In this example, we can see the execution mask for thread 5, with only bits 0 to 4 set. Looking at the resultant bit range, one can see that a population count of the 1s will produce our expected output slot.



Compaction

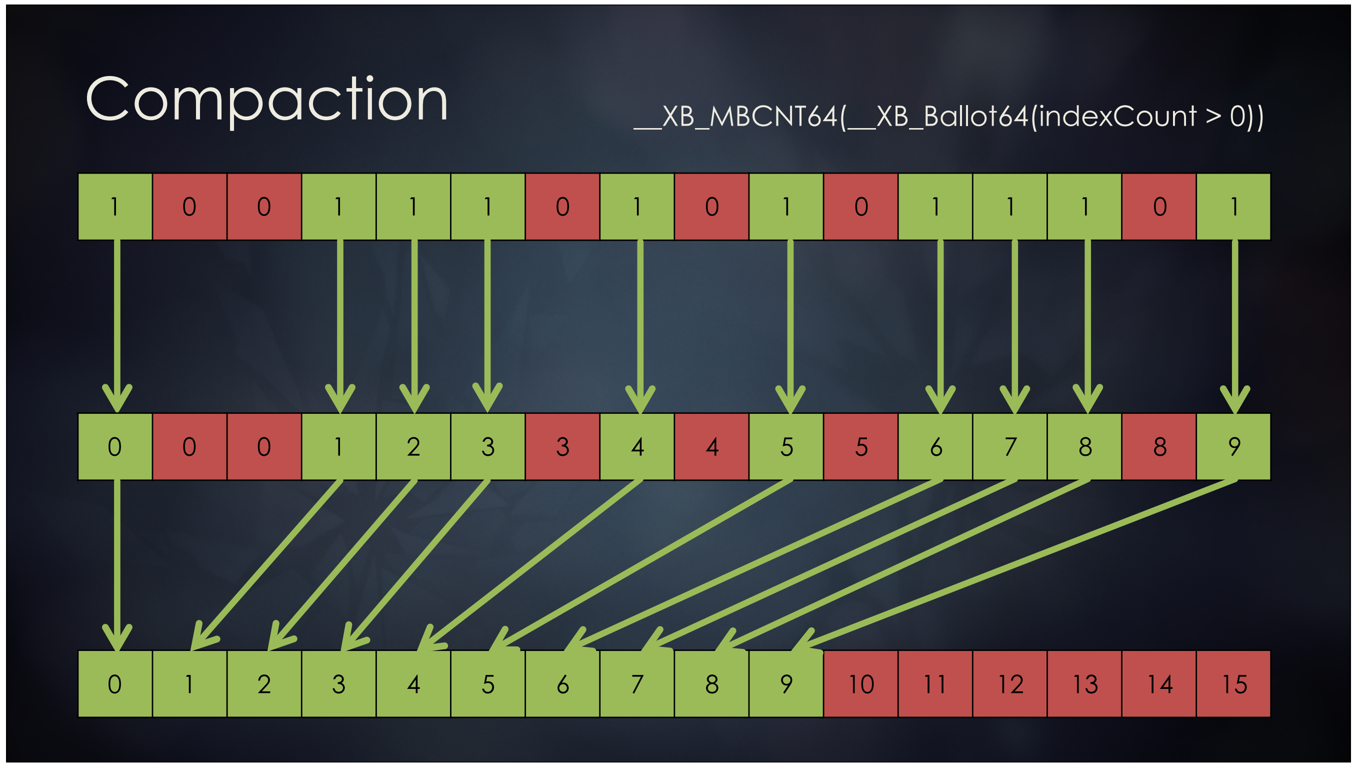
- **V_MBCNT_LO_U32_B32** [5]
  - Masked bit count of the lower 32 threads (0-31)
- **V_MBCNT_HI_U32_B32** [5]
  - Masked bit count of the upper 32 threads (32-63)
- For each thread, returns the # of active threads which come before it.

```
uint2 compactMask;
compactMask.x = laneId >= 32 ? ~0 : ((1 <<  laneId) - 1);
compactMask.y = laneId  < 32 ?  0 : ((1 << (laneId - 32)) - 1);
uint compactedIndex = countBits(clusterValidBallot.x & compactMask.x) +
                      countBits(clusterValidBallot.y & compactMask.y);
```

```
uint compactedIndex = __XB_MBCNT64(clusterValidBallot);
```

GCN has two instructions that can be paired with ballot to produce the correct compaction results. V_MBCNT_LO will produce a masked bit count of the lower 32 threads, and V_MBCNT_HI will produce a masked bit count of the upper 32 threads.

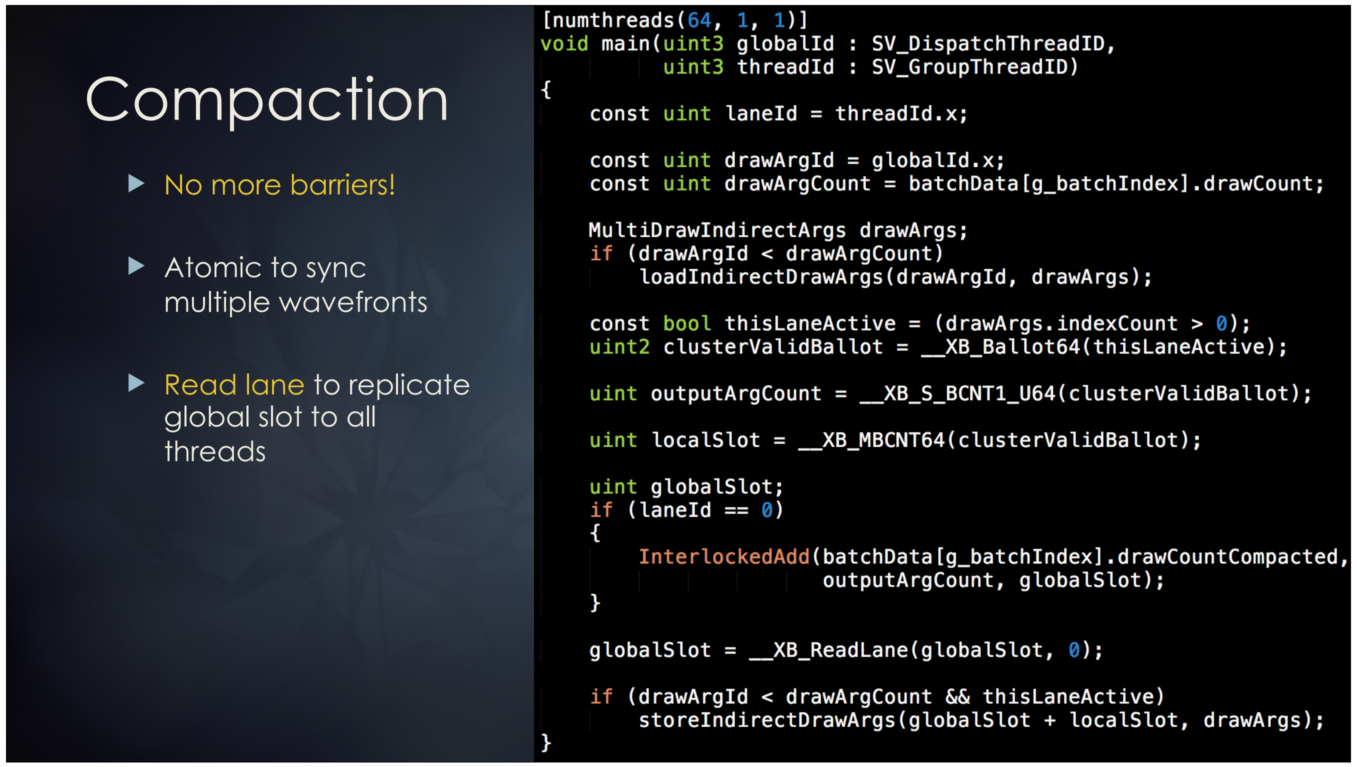
Chaining these instructions together will, for each thread, count the # of active threads which come before it, similar to the reference implementation listed here.



Compaction

```
__XB_MBCNT64(__XB_Ballot64(indexCount > 0))
```

Combining ballot and masked bit count will compact our surviving draw call stream within a wavefront without the need for any synchronization or group shared memory.



Compaction

- No more barriers!
- Atomic to sync multiple wavefronts
- Read lane to replicate global slot to all threads

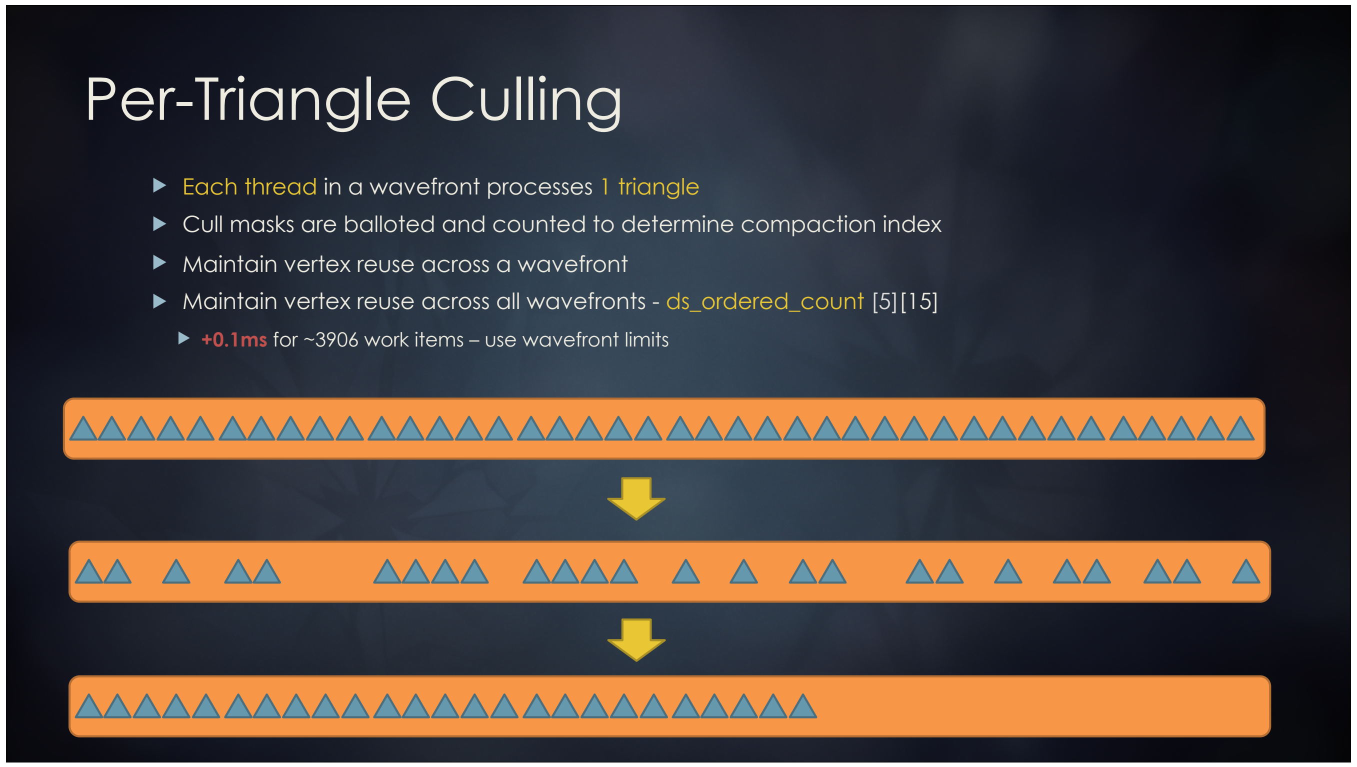
```
[numthreads(64, 1, 1)]
void main(uint3 globalId : SV_DispatchThreadID,
          uint3 threadId : SV_GroupThreadID)
{
    const uint laneId = threadId.x;

    const uint drawArgId = globalId.x;
    const uint drawArgCount = batchData[g_batchIndex].drawCount;

    MultiDrawIndirectArgs drawArgs;
    if (drawArgId < drawArgCount)
        loadIndirectDrawArgs(drawArgId, drawArgs);

    const bool thisLaneActive = (drawArgs.indexCount > 0);
    uint2 clusterValidBallot = __XB_Ballot64(thisLaneActive);

    uint outputArgCount = __XB_S_BCNT1_U64(clusterValidBallot);

    uint localSlot = __XB_MBCNT64(clusterValidBallot);

    uint globalSlot;
    if (laneId == 0)
    {
        InterlockedAdd(batchData[g_batchIndex].drawCountCompacted,
                       outputArgCount, globalSlot);
    }

    globalSlot = __XB_ReadLane(globalSlot, 0);

    if (drawArgId < drawArgCount && thisLaneActive)
        storeIndirectDrawArgs(globalSlot + localSlot, drawArgs);
}
```

The GCN optimized compaction shader looks like this. We no longer have any barriers. In order to compact across multiple wavefronts, we have a single atomic operation per wavefront that reserves the output space for all the surviving draw calls across each wavefront.

Instead of using a barrier so that all threads get globalSlot calculated correctly, we can read the value of globalSlot from the lane that computed it.

Triangle Culling

I’m now going to go over the per triangle culling filters performed on clusters that survive the initial coarse culling.



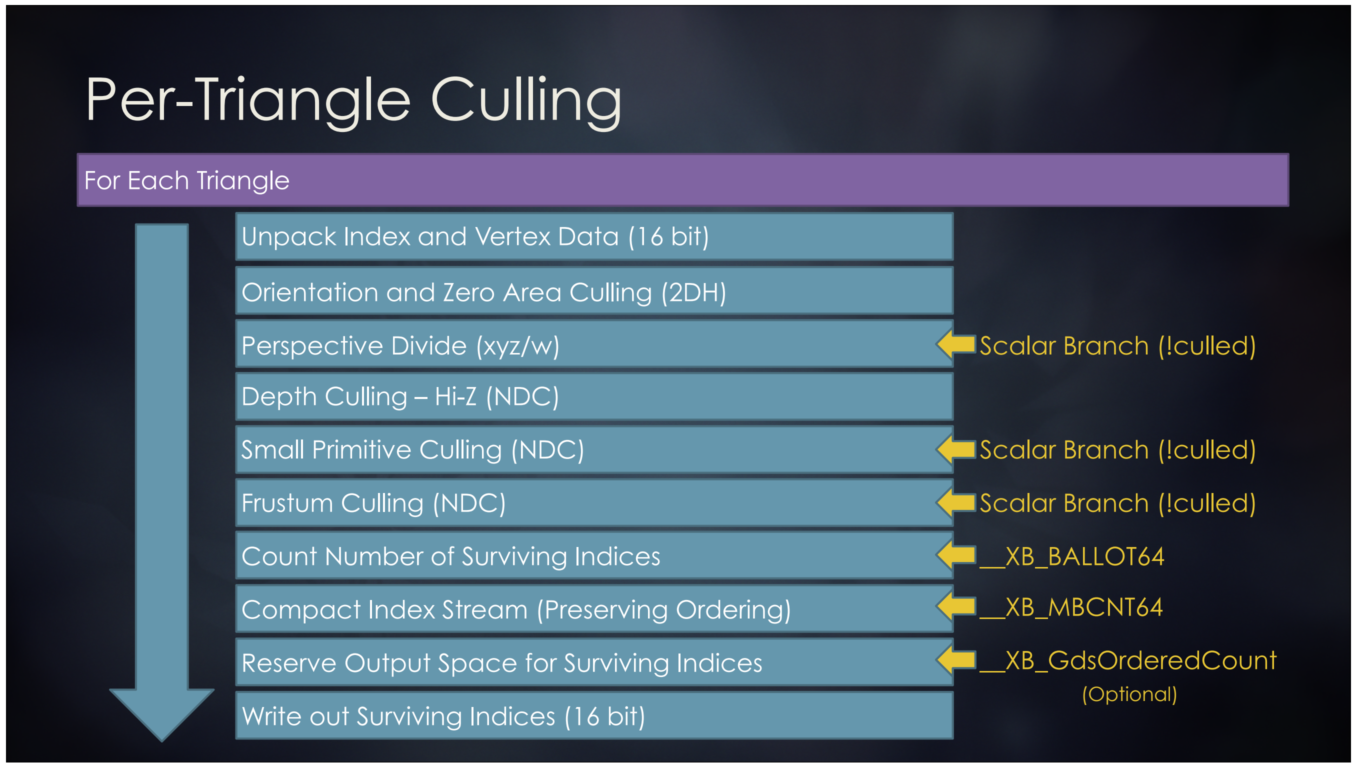
Per-Triangle Culling

- Each thread in a wavefront processes 1 triangle
- Cull masks are balloted and counted to determine compaction index
- Maintain vertex reuse across a wavefront
- Maintain vertex reuse across all wavefronts - ds_ordered_count [5][15]
  - **+0.1ms** for ~3906 work items – use wavefront limits

As mentioned already, each thread in a wavefront processes 1 triangle. Various culling operations are applied, and the surviving triangles across a wavefront need to be counted to determine the compaction index, or, the location in the resultant index buffer where the surviving indices will be written. This step is important for maintaining vertex reuse across a wavefront.

Each wavefront then writes out the block of surviving indices to its output location. If ordering across all wavefronts is important, such as with translucent or procedural rendering, then the block of surviving indices can be written out in wavefront creation order using ds_ordered_count. I found that using ds_ordered_count to maintain vertex reuse across an entire mesh was usually not worth the cost, as work items of 256 triangles gives perfect vertex reuse.

The factors contributing to the added cost are due to the way ds_ordered_count works under the hood, the size of the vertex cache, and what happens to vertex reuse when you start to remove parts of the mesh. If using ds_ordered_count, you can optimize it further through carefully tuned wavefront limits.



Per-Triangle Culling

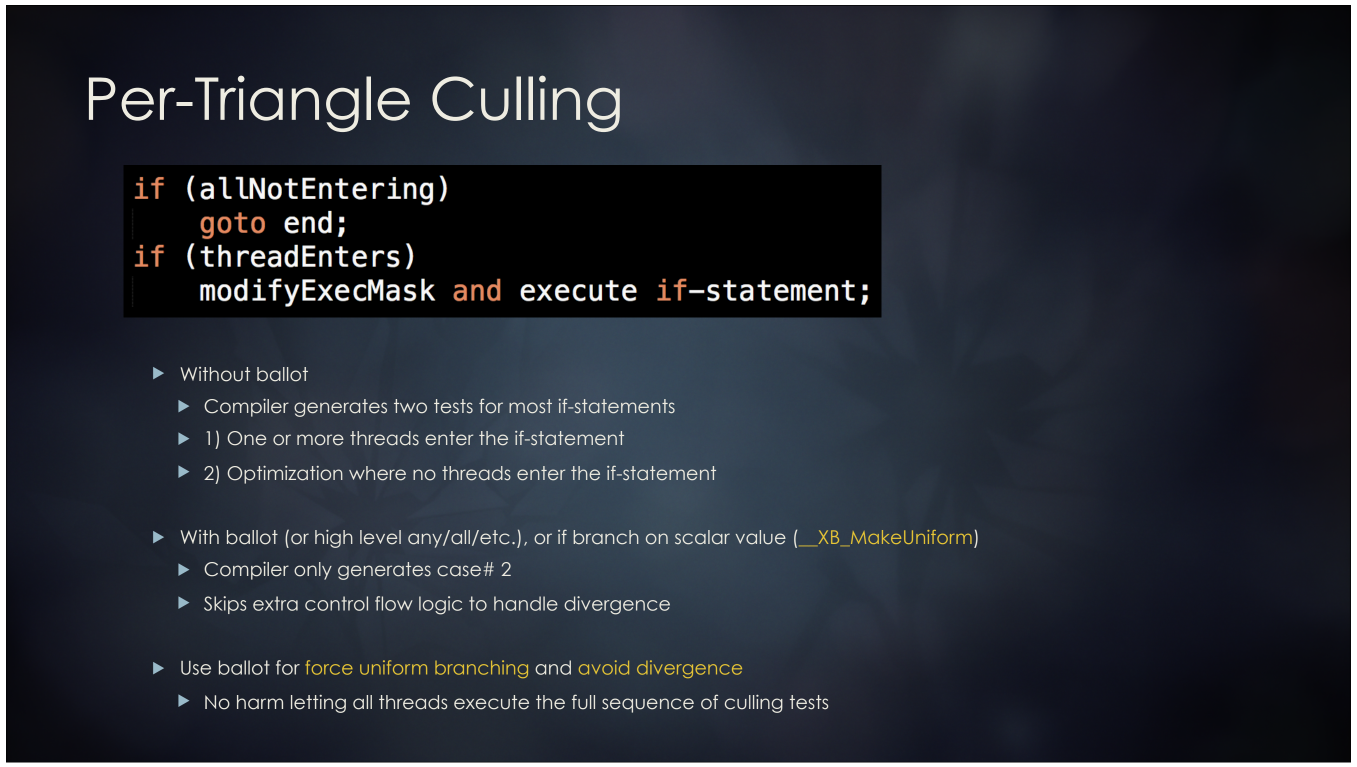
For Each Triangle

- Unpack Index and Vertex Data (16 bit)
- Orientation and Zero Area Culling (2DH)
- Perspective Divide (xyz/w) ← Scalar Branch (!culled)
- Depth Culling – Hi-Z (NDC)
- Small Primitive Culling (NDC) ← Scalar Branch (!culled)
- Frustum Culling (NDC) ← Scalar Branch (!culled)
- Count Number of Surviving Indices ← __XB_BALLOT64
- Compact Index Stream (Preserving Ordering) ← __XB_MBCNT64
- Reserve Output Space for Surviving Indices ← __XB_GdsOrderedCount (Optional)
- Write out Surviving Indices (16 bit)

This is an overview of operations that the cull shader is performing on 1 triangle per thread across each work item.

Triangle data is unpacked, the various culling filters are executed, count/compaction/reserve is performed, and then the indices are written out as 16 bit. Since compute cannot write out 16 bit types, I first zero the output buffer, use & 1 as a predicate on the thread id to determine low or high masking, and use InterlockedOr against the output location. This cleverly uses the L2 cache as a write combiner.

Another important optimization to mention, is that on consoles, you can use branch on a comparison with ballot to give the compiler a scalar branch uniformity hint in order to improve your code gen.



Per-Triangle Culling

```
if (allNotEntering)
    goto end;
if (threadEnters)
    modifyExecMask and execute if-statement;
```

- Without ballot
  - Compiler generates two tests for most if-statements
  - 1) One or more threads enter the if-statement
  - 2) Optimization where no threads enter the if-statement
- With ballot (or high level any/all/etc.), or if branch on scalar value (__XB_MakeUniform)
  - Compiler only generates case# 2
  - Skips extra control flow logic to handle divergence
- Use ballot for force uniform branching and avoid divergence
  - No harm letting all threads execute the full sequence of culling tests

Without ballot, the compiler will generate two tests for most if statements - one is for the case where one or more threads enter the if-statement, and the other is an optimization where the compiler will check to see if everyone *didn't* enter the if-statement, and if so it branches over the if-statement.

Really it's just a single comparison test and the compiler essentially checks to see if all lanes had the same result, so the compiler is basically generating a ballot for you. So you get code gen that looks like the top block.

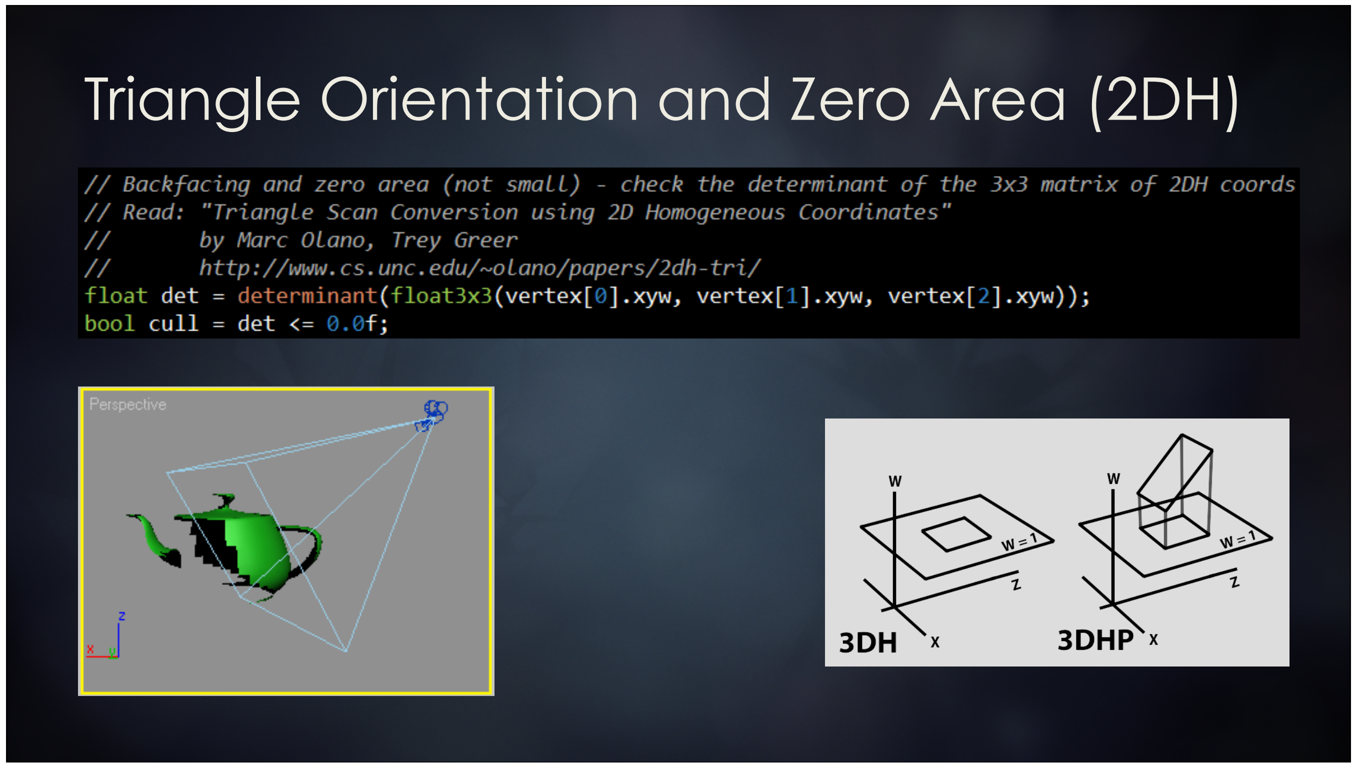
If you explicitly use ballot (or any/all/etc. which are high-level versions of ballot), or if you branch on a scalar value (i.e. __XB_MakeUniform), the compiler only generates the single "if (allNotEntering) goto end;" part and skips the extra control flow logic to handle divergence.

In the case of the culling work loop, I use ballot to force uniform branching and avoid divergence (including the slight codegen hit), because there's no harm in letting all threads execute the full sequence of culling tests. If any thread needs to run that code, then all threads end up running it because of the SIMD being 64-wide.

There is a case where you should use divergent branching - if any of the culling tests involve memory fetches or LDS ops, it is worth masking those out, such as with depth Hi-Z culling.

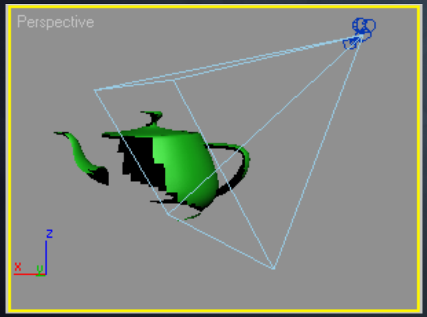
Orientation Culling

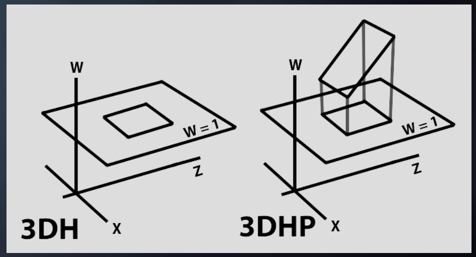
The first, and most important filter, is orientation culling.



Triangle Orientation and Zero Area (2DH)

```
// Backfacing and zero area (not small) - check the determinant of the 3x3 matrix of 2DH coords
// Read: "Triangle Scan Conversion using 2D Homogeneous Coordinates"
//       by Marc Olano, Trey Greer
//       http://www.cs.unc.edu/~olano/papers/2dh-tri/
float det = determinant(float3x3(vertex[0].xyw, vertex[1].xyw, vertex[2].xyw));
bool cull = det <= 0.0f;
```



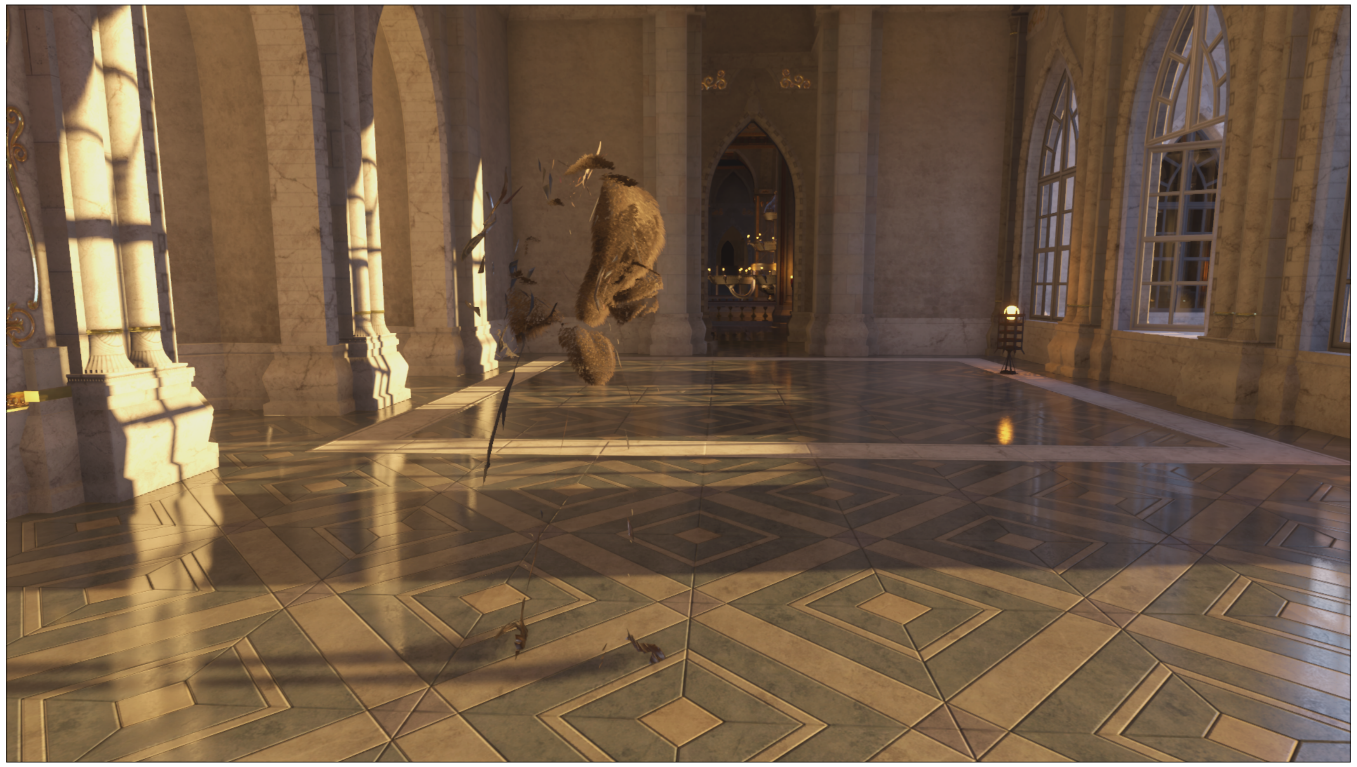


On average, 50% of a mesh will be culled from backface. So we need a test which is as cheap as possible. One of the cheapest tests is the one described in this paper [3], using the determinant of a 3x3 matrix with homogeneous coordinates [2]. This technique avoids clipping and projection, which includes ¼ rate reciprocal instructions coming from the perspective divide.

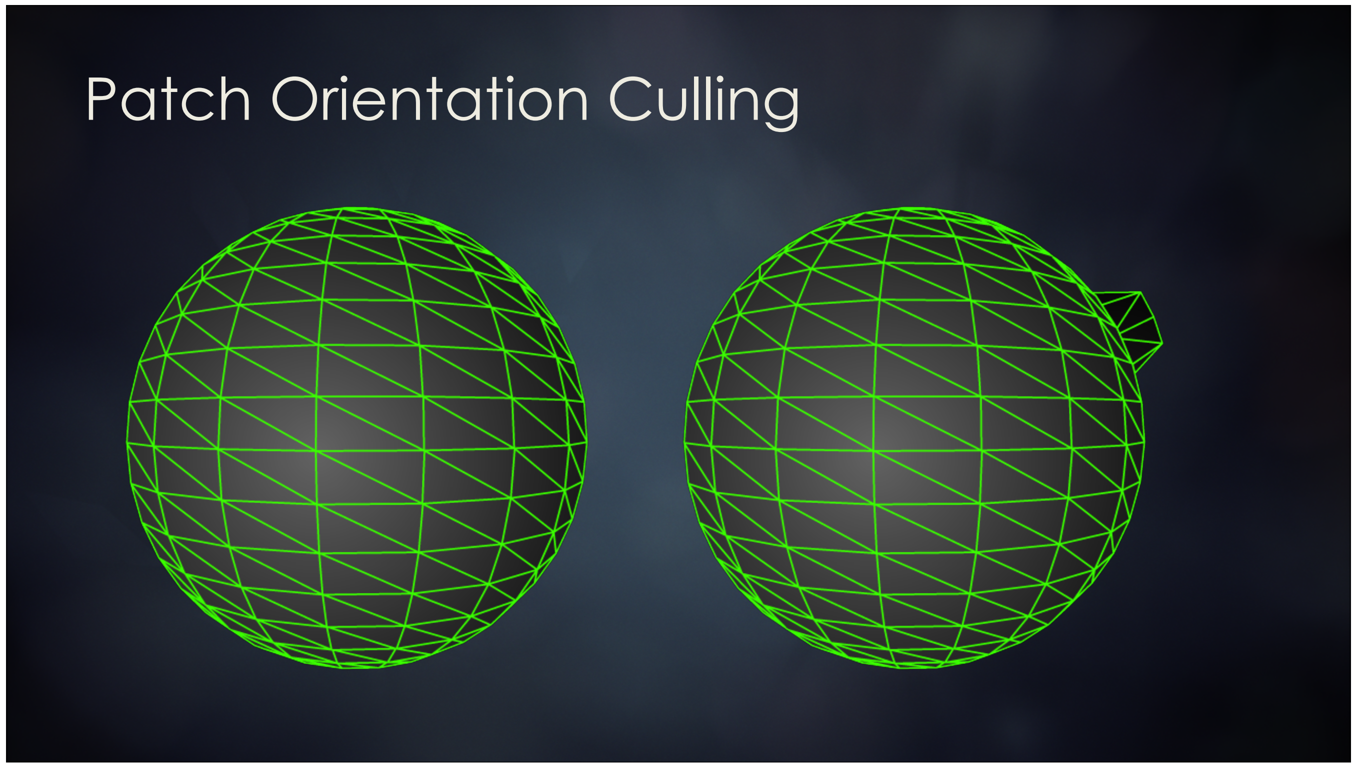
Using GCN specific optimizations, we can skip all the tests afterwards if backfacing already removed all the triangles within a wavefront. The direction of the determinant test is based on whether you are culling front or back facing triangles.

This particular test works under MSAA or EQAA conditions, as the zero area is not a small primitive test, but a degenerate triangle test (which any decent mesh pipeline should be removing offline, anyways).

Here is an example of the backface determinant test applied to Solas from a particular view.



Locking the current view, and then moving behind him shows that all backfacing triangles have been removed, as expected.



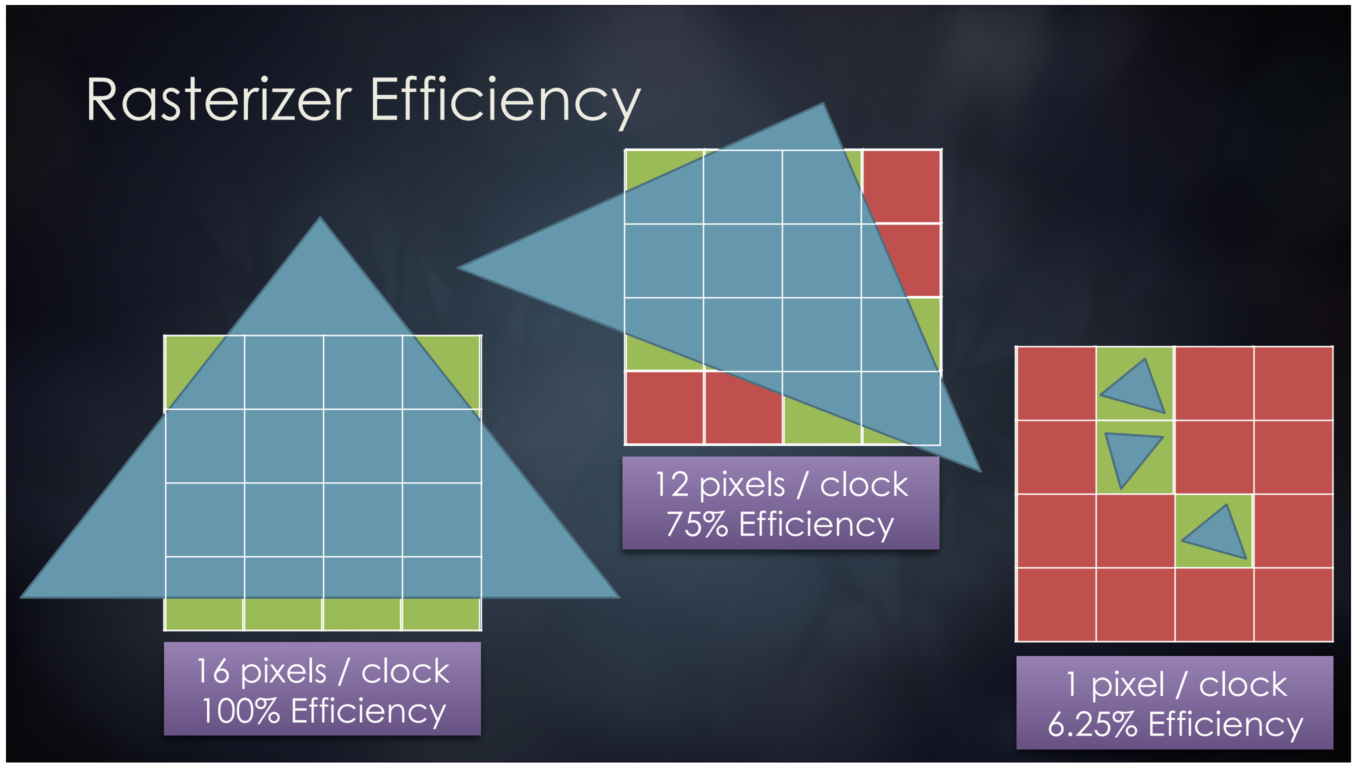
Patch Orientation Culling

When culling tessellated patches, it is also important to mention that the 2DH determinant test will not work correctly for back faces that displace into view. These faces would be culled pre-displacement, so you would lose a contributing portion of your silhouette.

For tessellated patches, we instead do back face culling in view space with a dot product bias that is determined by the max displacement amount.

Small Primitive Culling

Another effective filter is small primitive, or, culling triangles that do not generate pixel coverage.



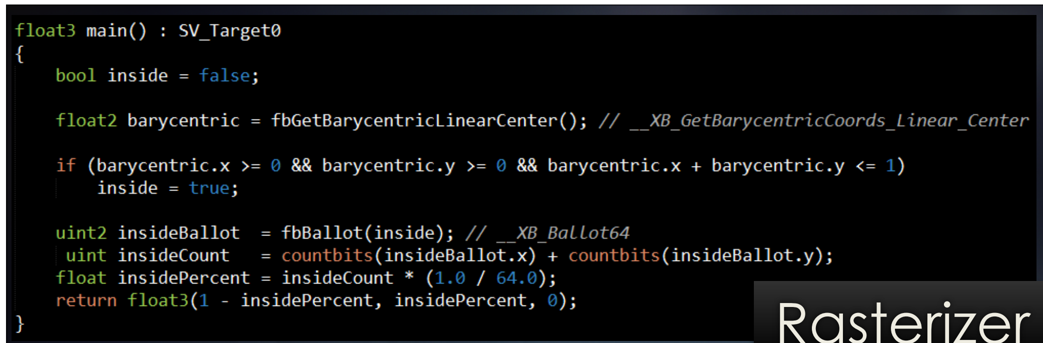
Rasterizer Efficiency

Each GCN rasterizer can read one triangle per clock and produce up to 16 pixels per clock. Because of this, small triangles are extremely inefficient to rasterize.

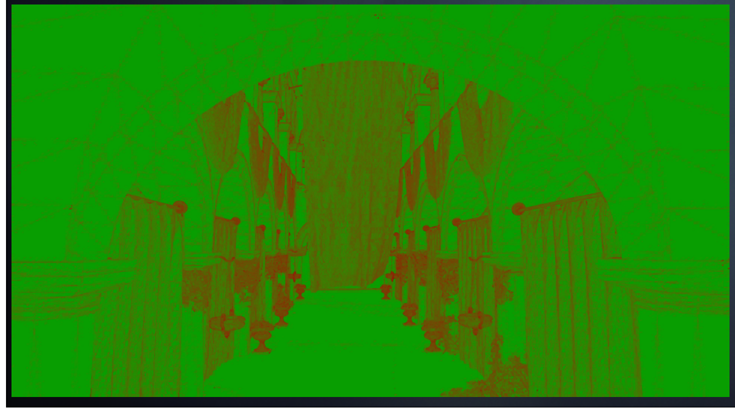
The left image produces 4 quads, 16 pixels, at peak efficiency. The middle image produces 4 quads, but only 12 pixels are valid. It consumes 16 threads in the pixel shader though, due to helper lanes. Helper lanes still take time to pack and prepare, so they actually hurt your pixel rate. Efficiency in the middle image is lost due to partially filled quads, as the GPU shades in blocks of 2x2 pixel quads.

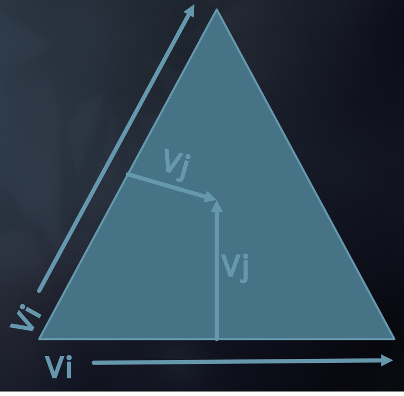
The right image has become bound by hitting primitive setup limits.

Rasterizer Efficiency



```
float3 main() : SV_Target0
{
    bool inside = false;

    float2 barycentric = fbGetBarycentricLinearCenter(); // __XB_GetBarycentricCoords_Linear_Center

    if (barycentric.x >= 0 && barycentric.y >= 0 && barycentric.x + barycentric.y <= 1)
        inside = true;

    uint2 insideBallot  = fbBallot(inside); // __XB_Ballot64
     uint insideCount   = countbits(insideBallot.x) + countbits(insideBallot.y);
    float insidePercent = insideCount * (1.0 / 64.0);
    return float3(1 - insidePercent, insidePercent, 0);
}
```

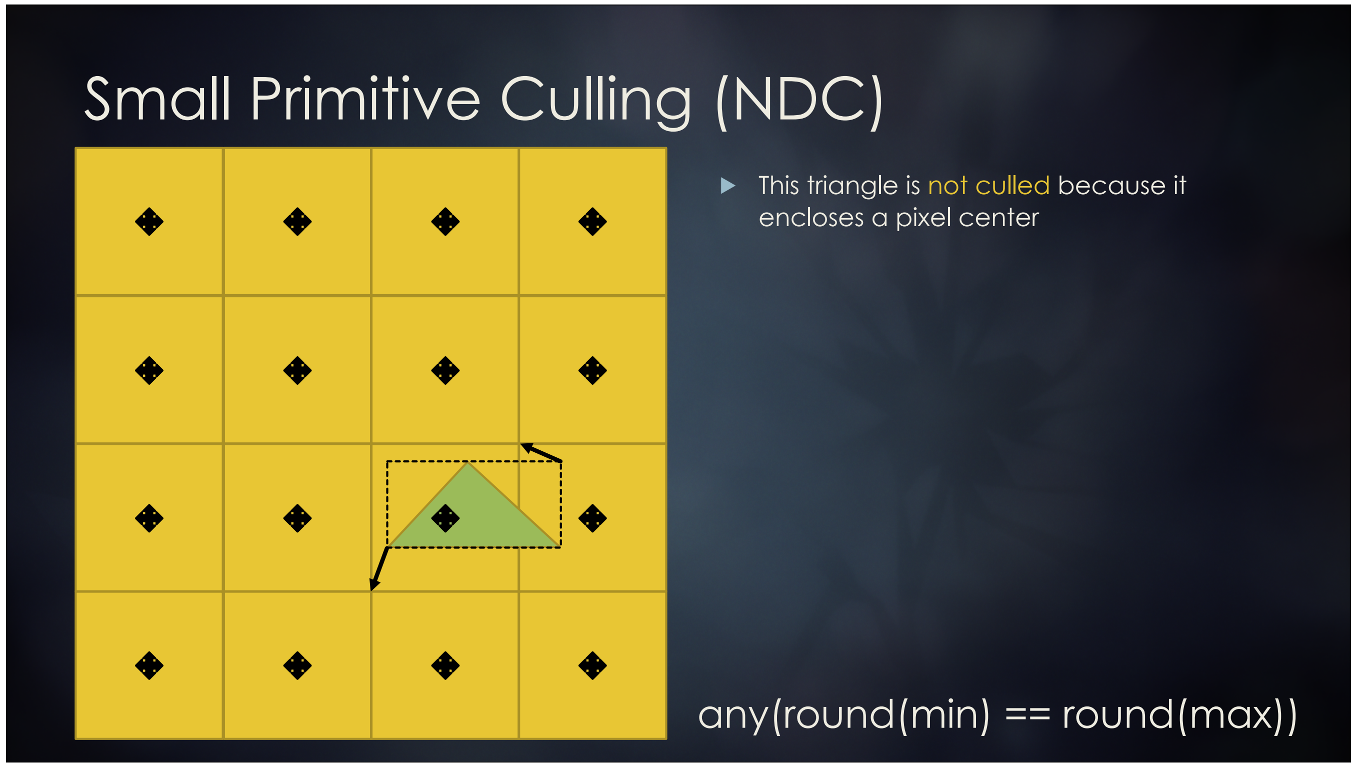




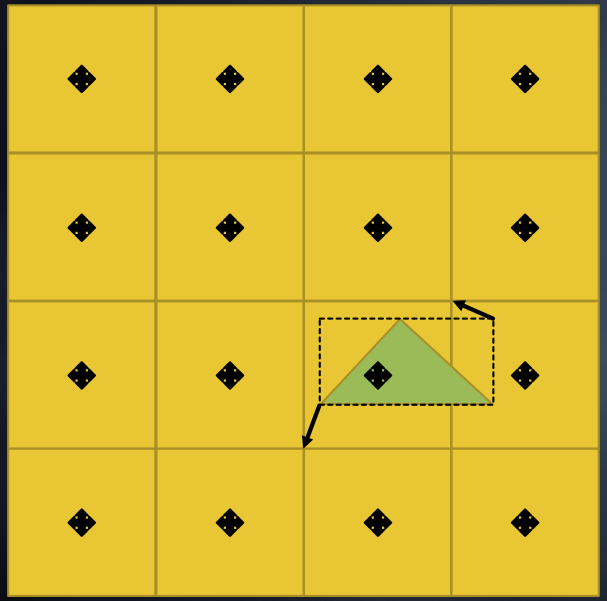
While not directly related to culling, this helpful pixel shader will identify meshes that are too dense, which will be affecting how many pixels are being delivered per clock. This is done by measuring the number of helper pixels. Or in other words, the number of covered pixels divided by the number of threads in a wavefront.

MSAA can have valid pixel threads with out-of-range barycentric coordinates, so switching from linear center to linear centroid will make it more accurate in this case.

I used this to get a rough idea of how prevalent small triangles were in our content, and whether or not a small primitive filter would be effective. As a bonus, this tool can now be used by artists to get a sense for how dense their meshes are given their LOD settings, and decimate accordingly.



Small Primitive Culling (NDC)



- This triangle is not culled because it encloses a pixel center

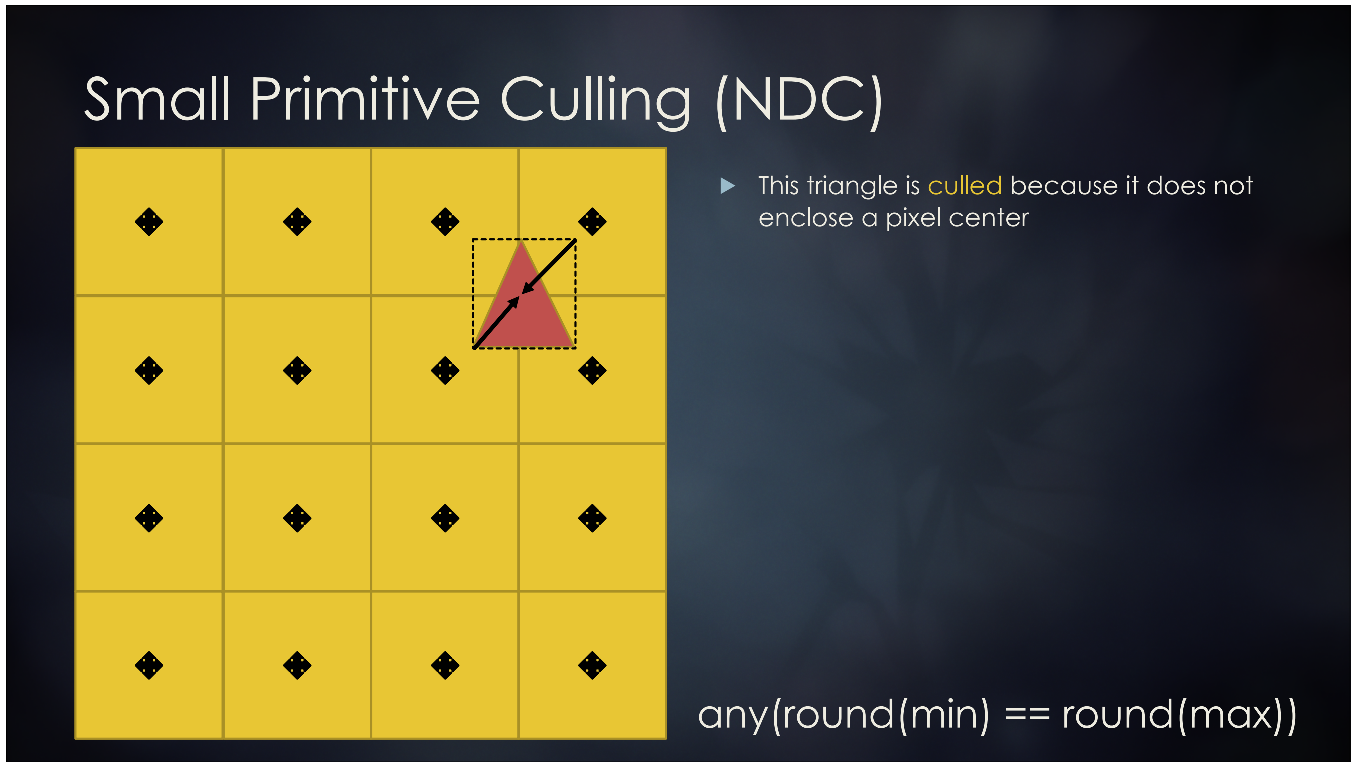
any(round(min) == round(max))

I originally started with a very exhaustive fixed point hardware precise small primitive filter, but later changed it to the approximation you see here, for non-MSAA targets.

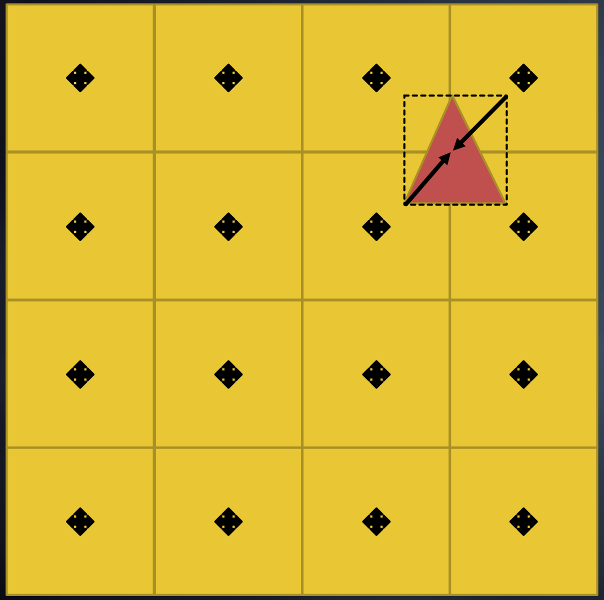
MSAA targets need to bias the test by enlarging based on sample count. If using custom programmable sample points, well, you're on your own. For MSAA, we need to essentially determine the maximum distance (in subpixels) between the pixel center and the outermost subpixel sample and use this to influence the test.

The general idea is that you take a screen space bounding box of a triangle, and snap min and max to the nearest pixel corner. If the min and max snap to either the same horizontal or vertical edge, the triangle does not enclose a pixel center, therefore not contributing pixel coverage.

In this example, this triangle is not culled because it encloses a pixel center.



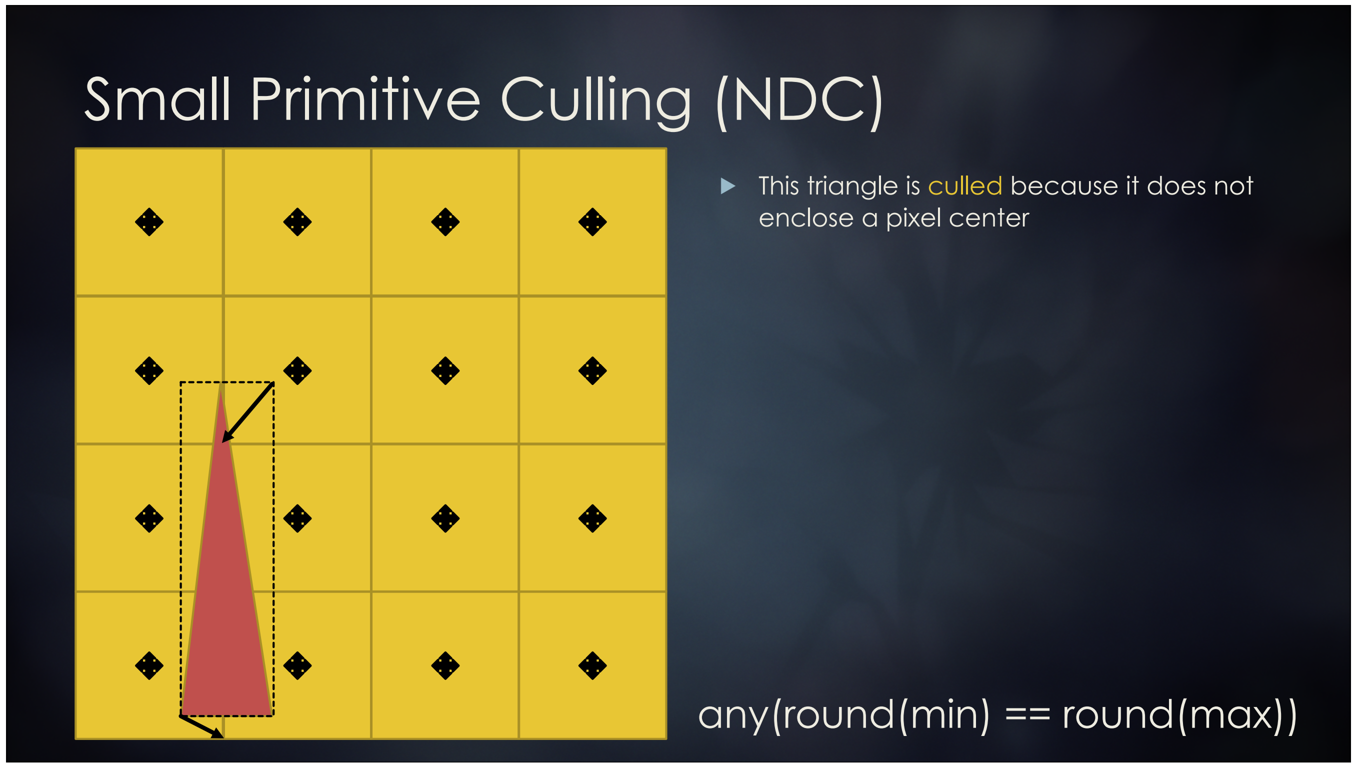
Small Primitive Culling (NDC)



- This triangle is culled because it does not enclose a pixel center

any(round(min) == round(max))

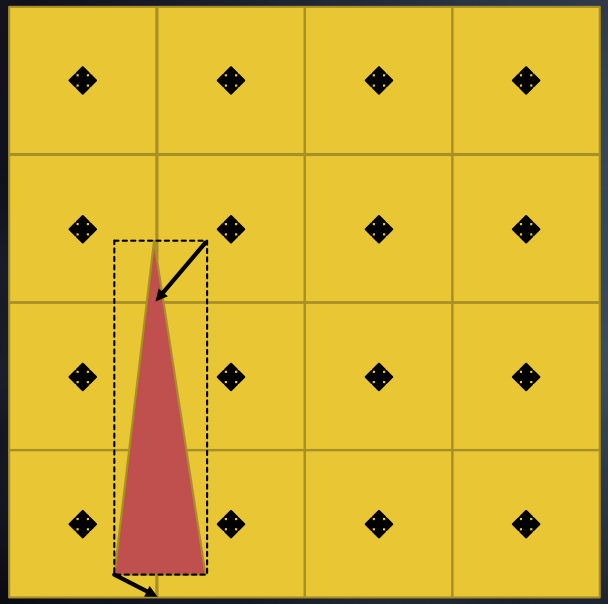
In a simple case, this triangle is culled because the min and max snap to the same location.



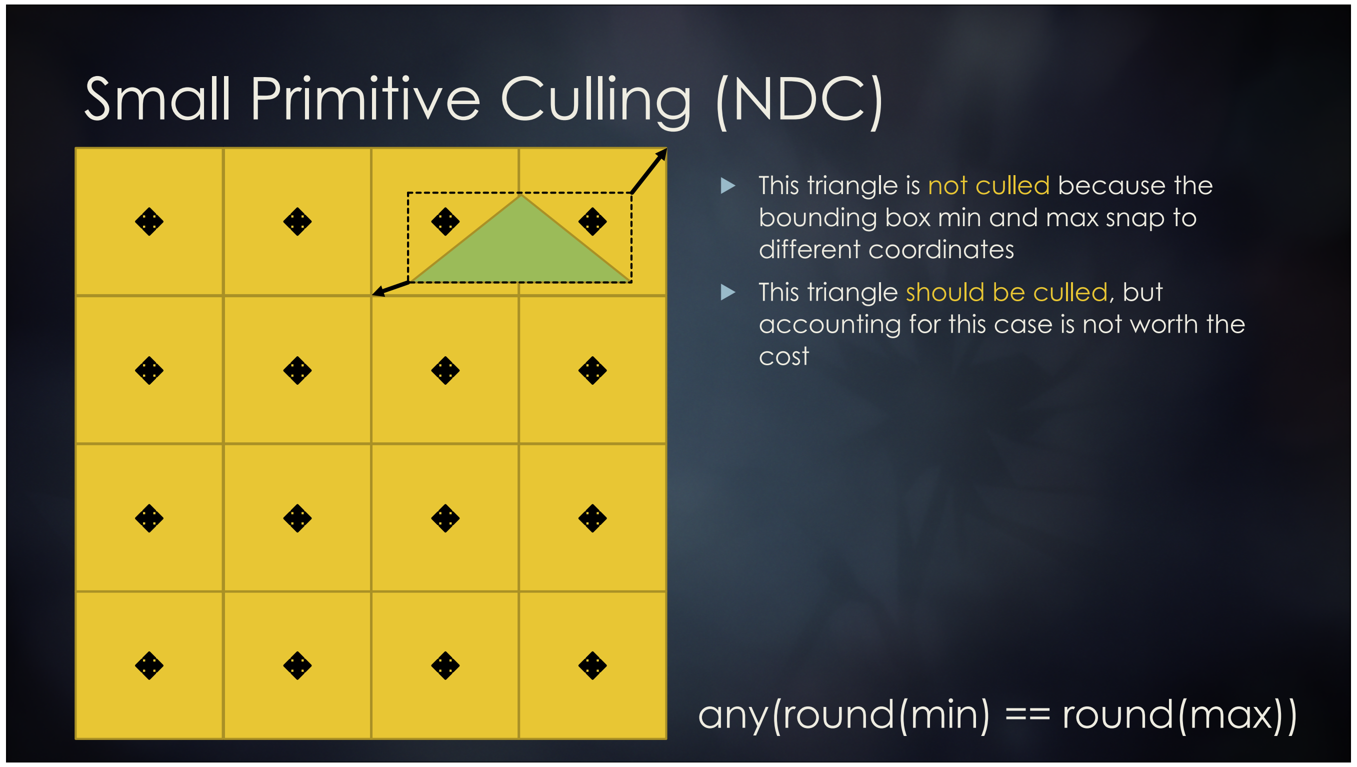
Small Primitive Culling (NDC)

- This triangle is culled because it does not enclose a pixel center

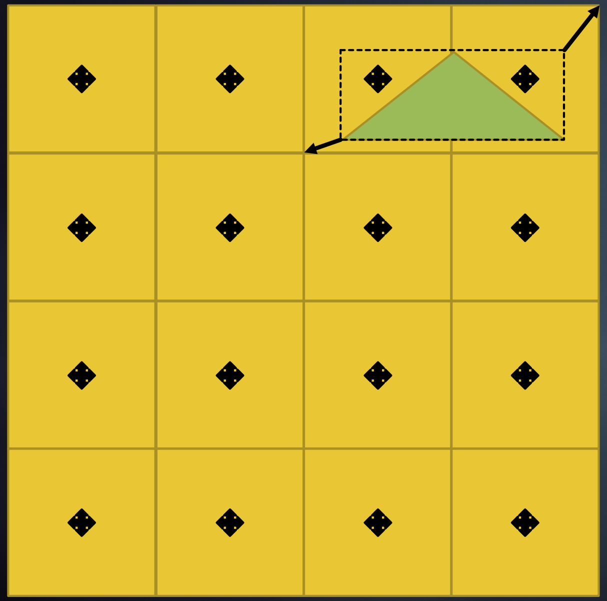
any(round(min) == round(max))



In a more complex case, this triangle is also culled. The min and max snap to different vertical coordinates, but the same horizontal coordinate.



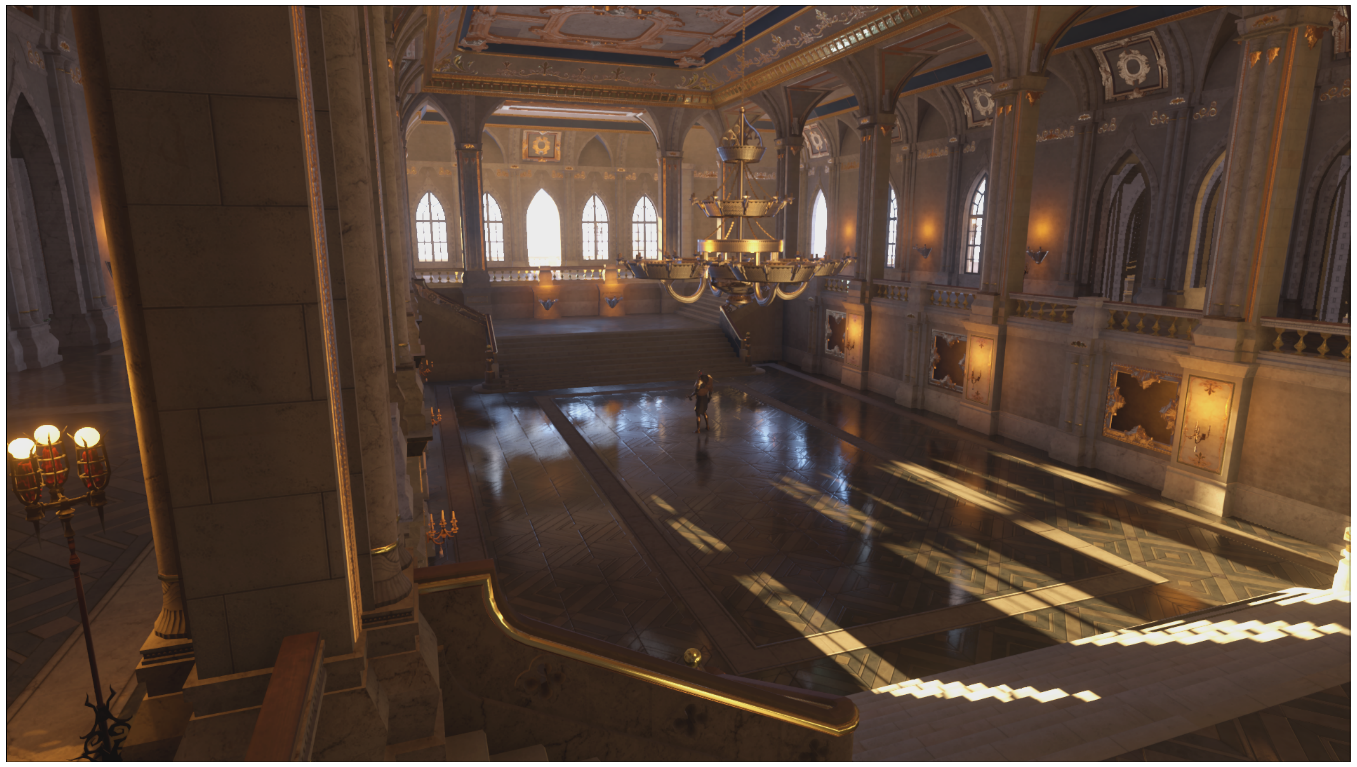
Small Primitive Culling (NDC)



- This triangle is not culled because the bounding box min and max snap to different coordinates
- This triangle should be culled, but accounting for this case is not worth the cost

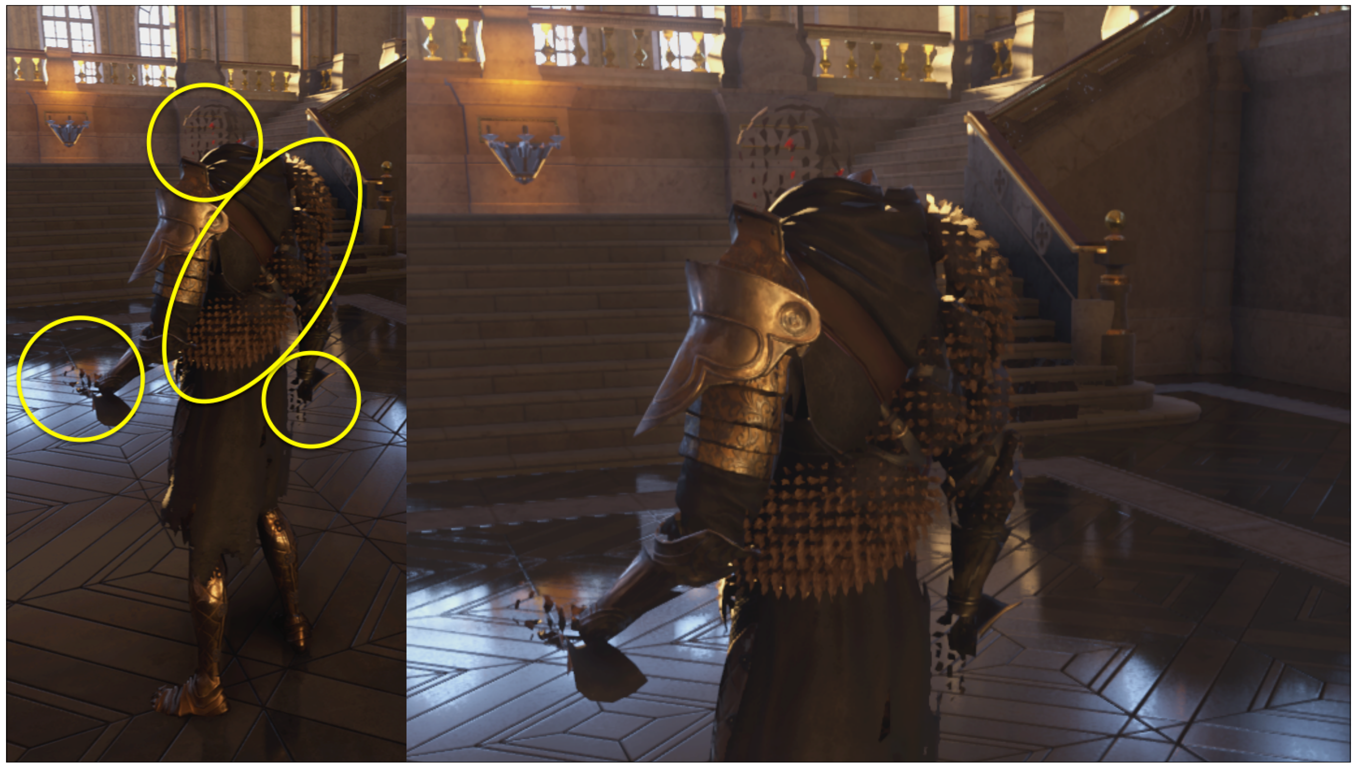
any(round(min) == round(max))

This test is conservative, so there is a case where triangles should be culled, but are not, as shown by this example. The bounding box min and max snap to different vertical and horizontal coordinates, yet the triangle does not enclose any pixel centers. Accounting for this case is not worth the cost, considering how cheap this test is.



Here is an example of the small primitive test applied to Solas, standing in the middle of the room, from a particular view.

Locking the view, and moving over to him shows quite a number of sub pixel triangles that have been removed with this filter.

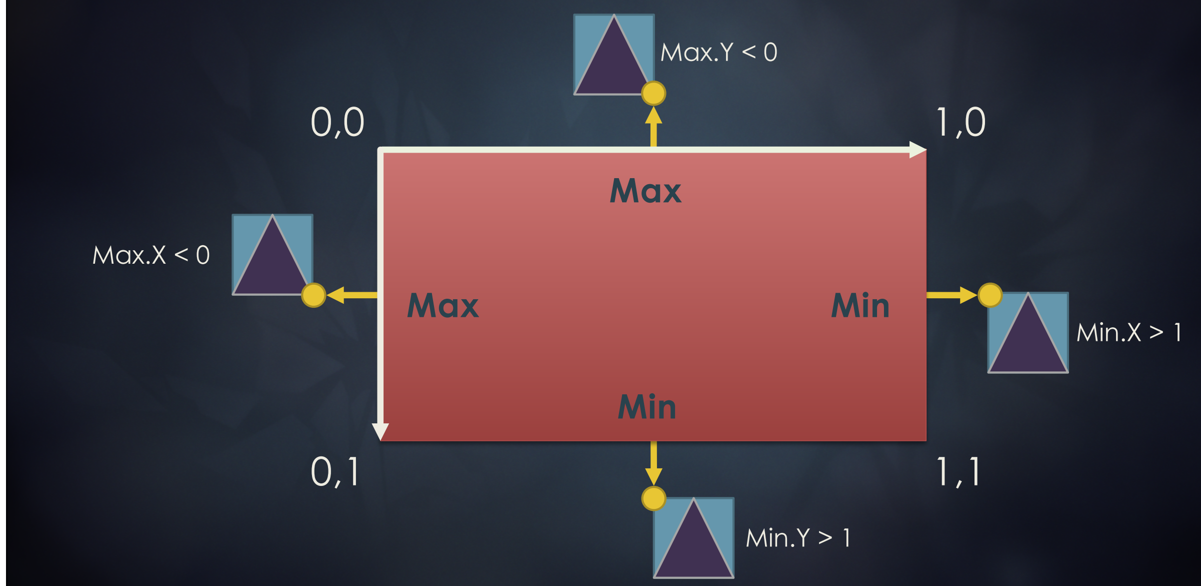


The projector may not show the removed triangles very well, so hopefully this enlarged version does a better job. Notice quite a number of removed triangles from the hands, head, and highly detailed pelt over his back. This extra concentration of triangles is typically due to importance of fidelity during close up cinematic shots during gameplay.

Frustum Culling

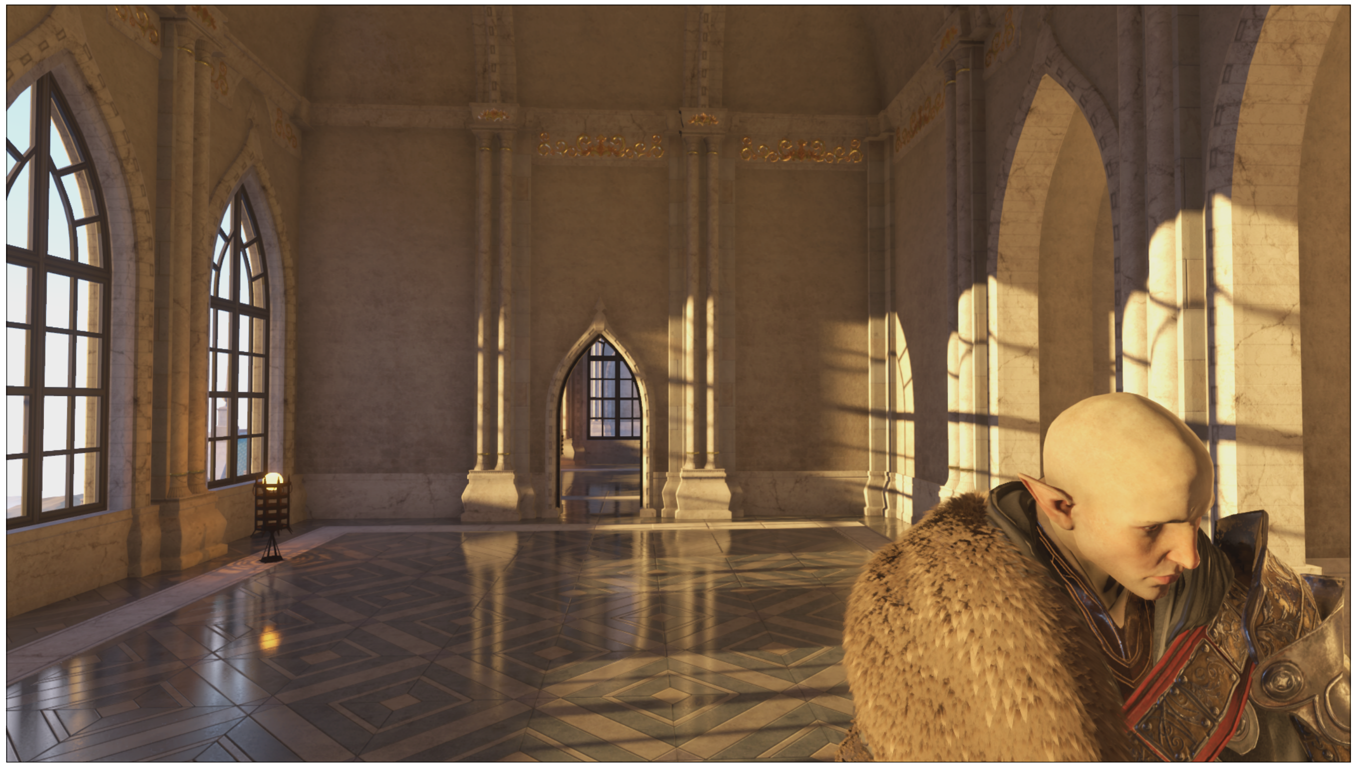
The next per triangle filter to cover is frustum culling

Frustum Culling (NDC)

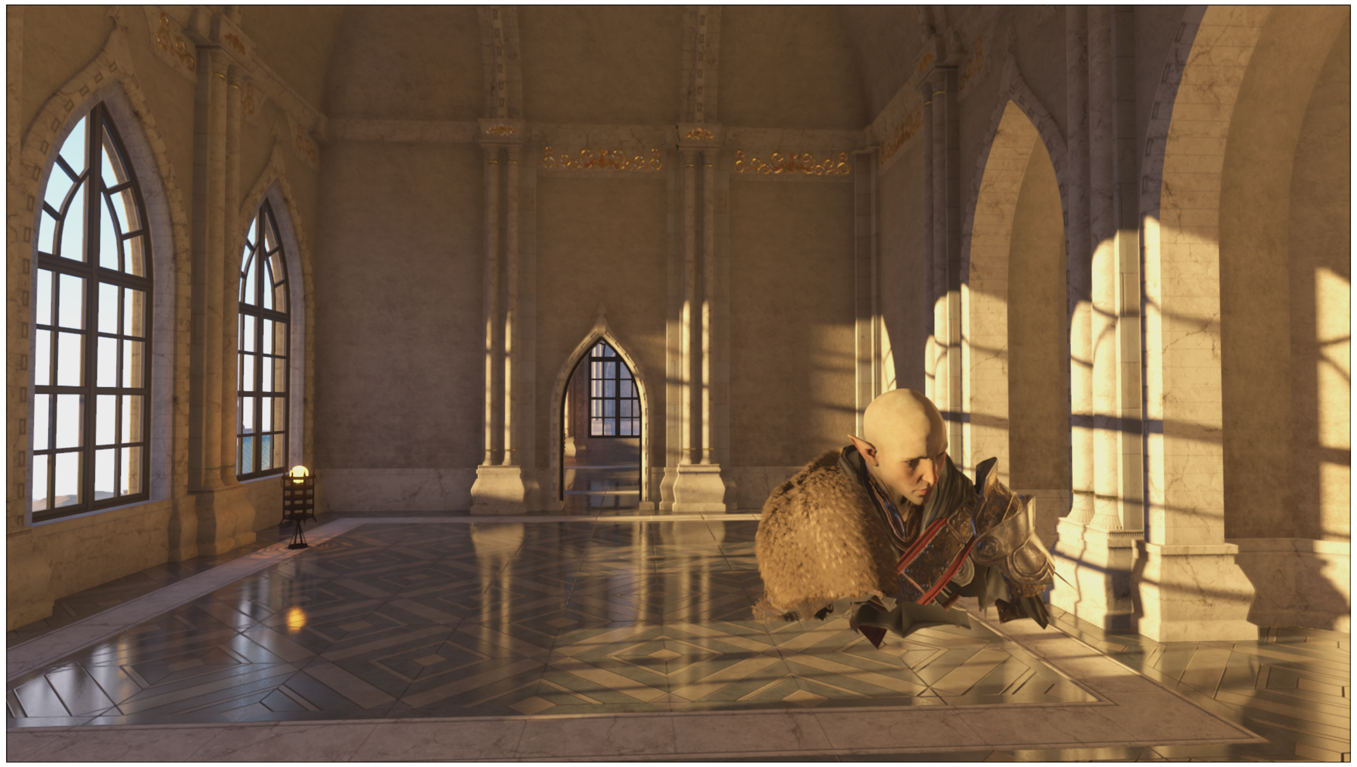


Most engines have whole object frustum culling on the CPU, making per cluster or triangle GPU frustum culling only effective when these objects intersect the planes. After the earlier culling filters, we now have post-projection vertices, and a huge budget of available ALU, so we do trivial frustum culling of 4 planes in 4 cycles, which still does provide some benefit in fringe cases, especially for composite objects which are made up of many parts.

Near and far plane culling is usually not worth the ALU for most titles. Similar to back face culling, it is important to mention that tessellated patches also require some form of tolerance values, in order to prevent incorrect culling of patches which tessellate from outside to inside of the view.



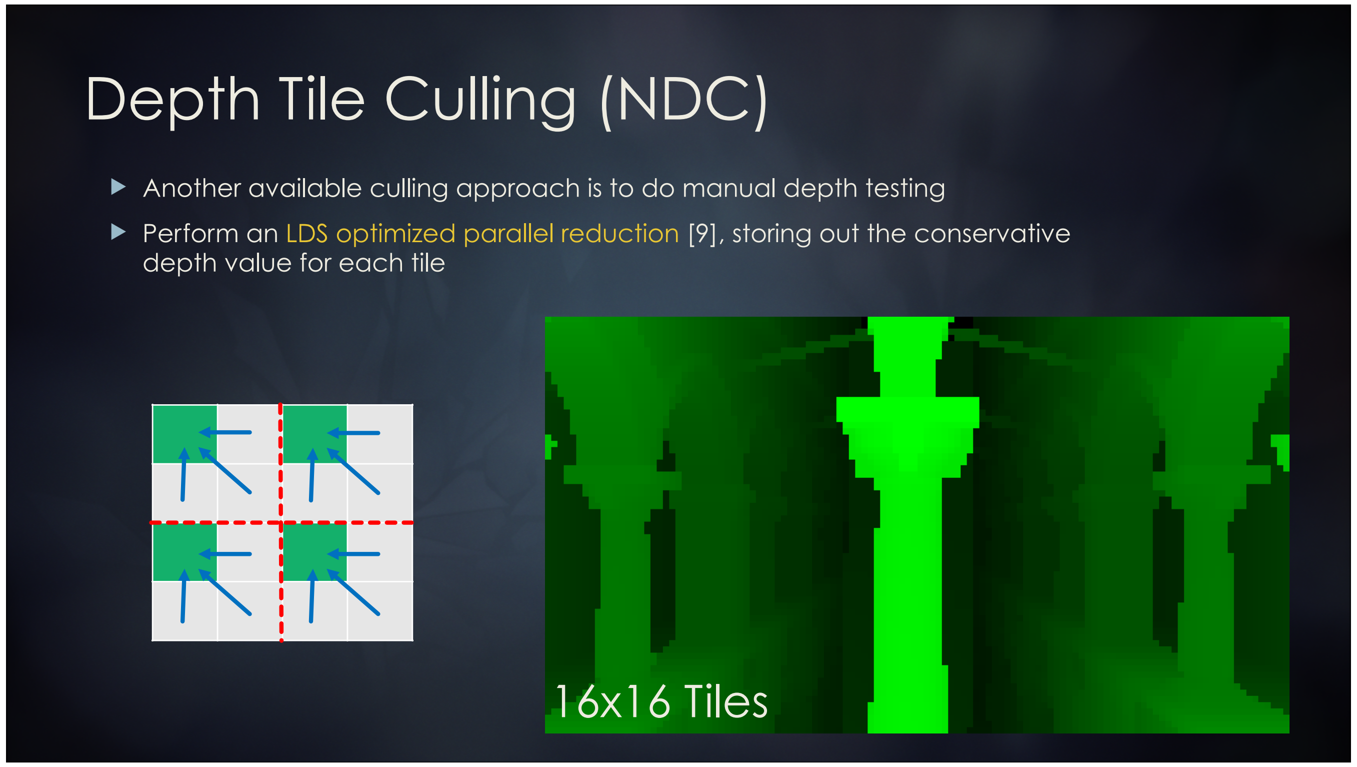
Here is an example of the frustum culling filter. This is the current view with just frustum culling enabled. We have an object which survives CPU frustum culling, but there are still quite a lot of triangles that could be removed.



After locking the view, moving backwards shows us how many triangles were removed using this filter.

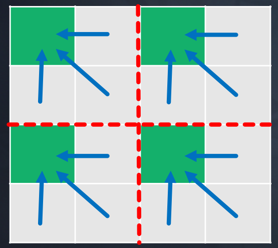
Depth Culling

The last filter, and the one which is the most involved to implement, is depth culling.



Depth Tile Culling (NDC)

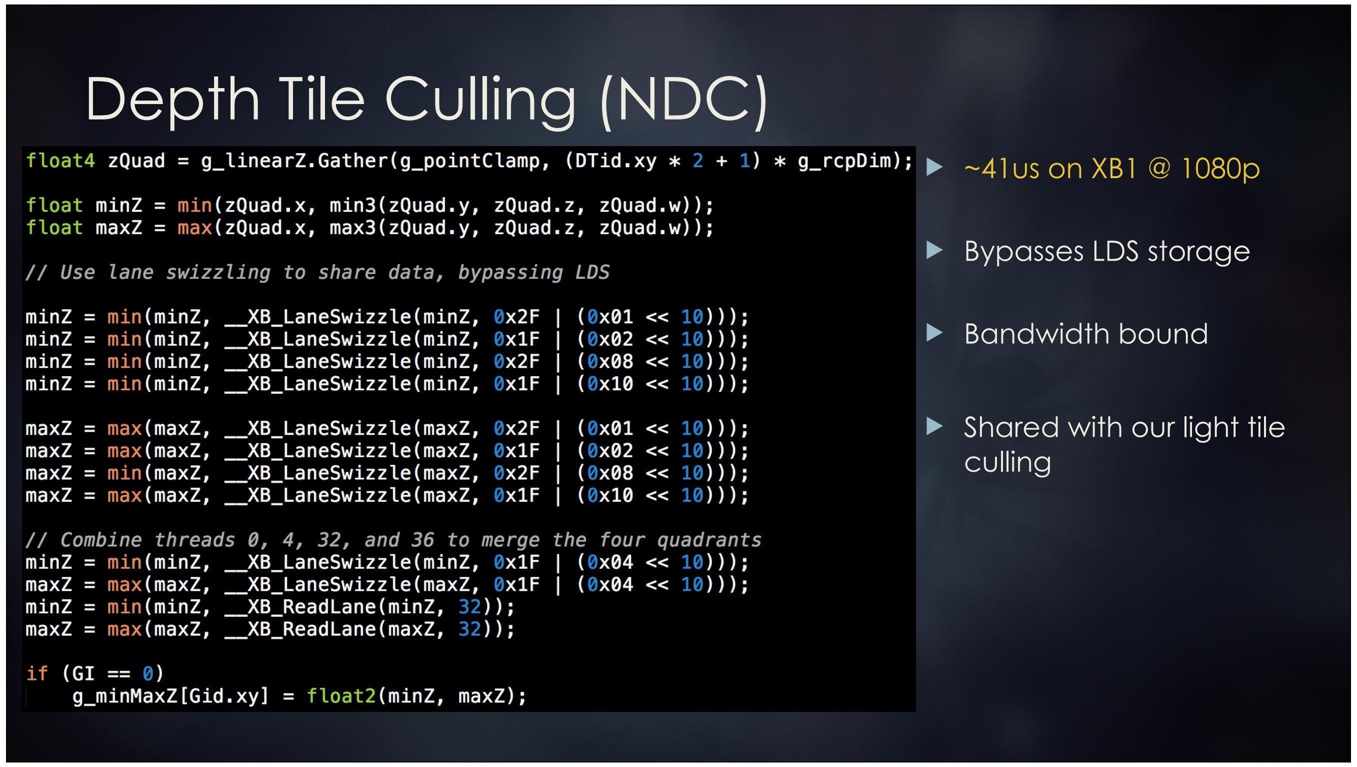
- Another available culling approach is to do manual depth testing
- Perform an LDS optimized parallel reduction [9], storing out the conservative depth value for each tile



16x16 Tiles

Another available triangle culling approach is to do manual depth testing. However it is worth noting that directly reading depth for cluster or triangle culling is extremely scene dependent due to availability, and the quality of occluders at any given time. The general technique is to take the depth buffer and perform an LDS optimized parallel reduction [9], storing out the conservative depth min or max value for each tile.

In my initial tests, I ran a full z pre-pass that produced a 16x16 depth tile grid, which I then tested against a screen space bounding box of the triangle or cluster. If the box was fully contained within a single tile, I would do a fast depth test and reject it. This approach, while fast, would only remove a fraction of triangles; any occluded triangles that straddled a tile border wouldn't be rejected. Modifying the filter to cull larger triangles spanning multiple tiles would be extremely expensive, and not worth the cost.

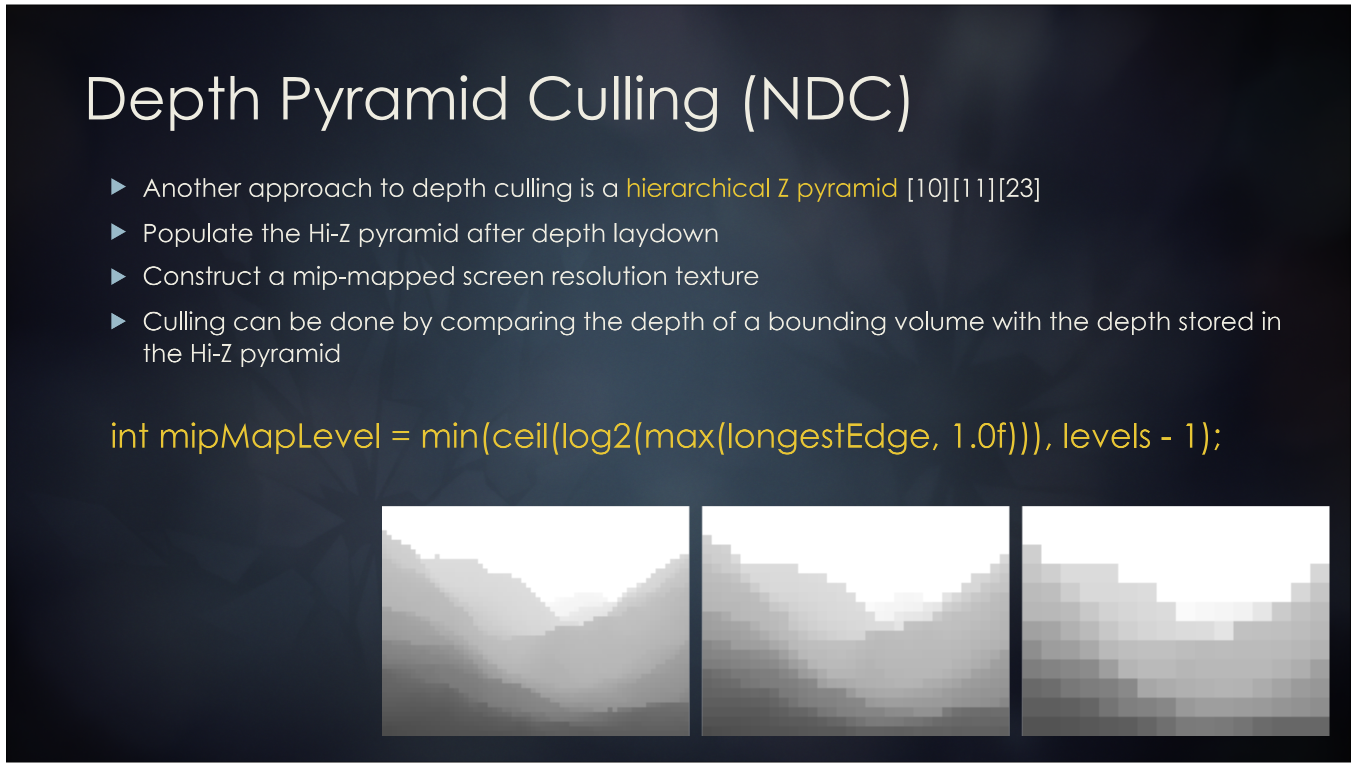


Depth Tile Culling (NDC)

```
float4 zQuad = g_linearZ.Gather(g_pointClamp, (DTid.xy * 2 + 1) * g_rcpDim);

float minZ = min(zQuad.x, min3(zQuad.y, zQuad.z, zQuad.w));
float maxZ = max(zQuad.x, max3(zQuad.y, zQuad.z, zQuad.w));

// Use lane swizzling to share data, bypassing LDS

minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x2F | (0x01 << 10)));
minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x1F | (0x02 << 10)));
minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x2F | (0x08 << 10)));
minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x1F | (0x10 << 10)));

maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x2F | (0x01 << 10)));
maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x1F | (0x02 << 10)));
maxZ = min(maxZ, __XB_LaneSwizzle(maxZ, 0x2F | (0x08 << 10)));
maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x1F | (0x10 << 10)));

// Combine threads 0, 4, 32, and 36 to merge the four quadrants
minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x1F | (0x04 << 10)));
maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x1F | (0x04 << 10)));
minZ = min(minZ, __XB_ReadLane(minZ, 32));
maxZ = max(maxZ, __XB_ReadLane(maxZ, 32));

if (GI == 0)
    g_minMaxZ[Gid.xy] = float2(minZ, maxZ);
```

- ~41us on XB1 @ 1080p
- Bypasses LDS storage
- Bandwidth bound
- Shared with our light tile culling

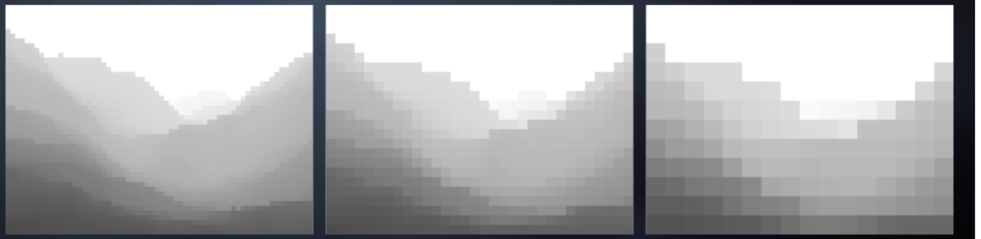
Here is a variant of parallel depth reduction which uses GCN lane swizzling to share data, bypassing LDS storage. With a 16 bit ESRAM depth buffer already decompressed, this computation runs in approximately 41 microseconds on the XB1 @ 1080p, and is completely bandwidth bound. We use the results from this reduction for other parts of our rendering including light tile culling.



Depth Pyramid Culling (NDC)

- Another approach to depth culling is a hierarchical Z pyramid [10][11][23]
- Populate the Hi-Z pyramid after depth laydown
- Construct a mip-mapped screen resolution texture
- Culling can be done by comparing the depth of a bounding volume with the depth stored in the Hi-Z pyramid

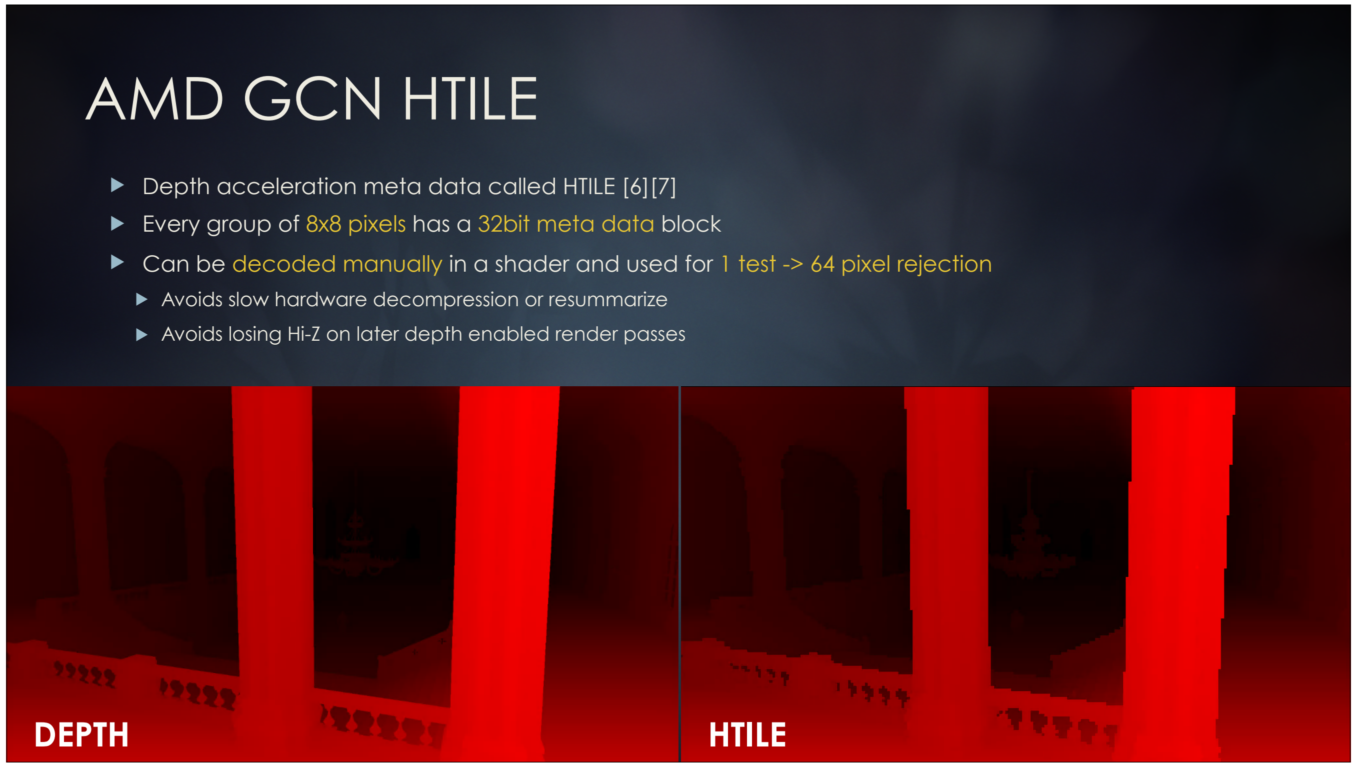
```
int mipMapLevel = min(ceil(log2(max(longestEdge, 1.0f))), levels - 1);
```



Another approach to depth culling is a hierarchical Z pyramid [10][11], which starts at the resolution of the depth buffer, and goes all the way to a single pixel. The first level of the pyramid is populated after depth laydown, similar to the depth tiles method. After which, we populate the remaining mip levels in the pyramid through a custom downsample pass.

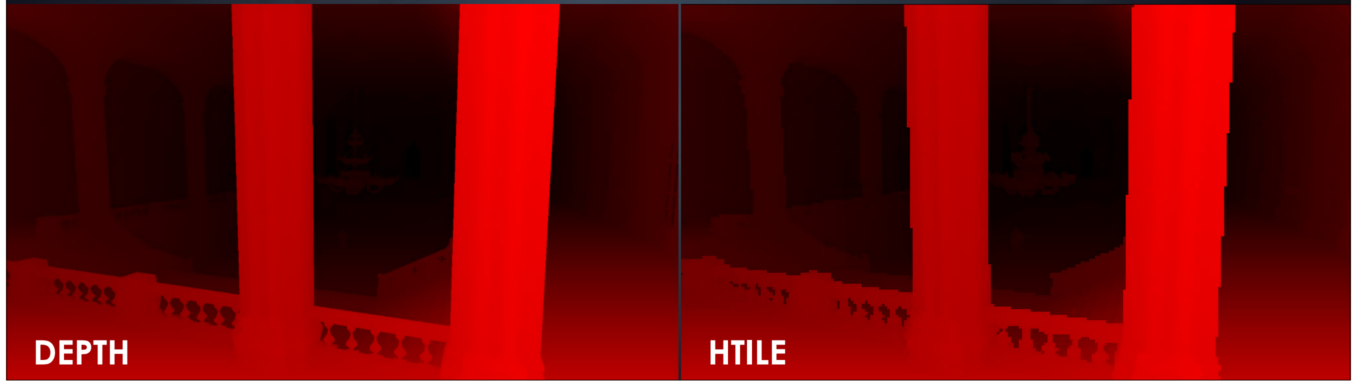
Each texel in mip level N contains the min or max depth of all corresponding texels in mip level N-1. Culling can be done by comparing the depth of a bounding volume's longest edge with the depth stored in the Hi-Z pyramid. Because the pyramid goes down to a single level, we can very easily get a single mip level to fetch, instead of using multiple fetches to handle overlapping quads.

This is the approach I ended up using for depth based culling, except I also accelerated it with HTILE.



AMD GCN HTILE

- Depth acceleration meta data called HTILE [6][7]
- Every group of 8x8 pixels has a 32bit meta data block
- Can be decoded manually in a shader and used for 1 test -> 64 pixel rejection
  - Avoids slow hardware decompression or resummarize
  - Avoids losing Hi-Z on later depth enabled render passes

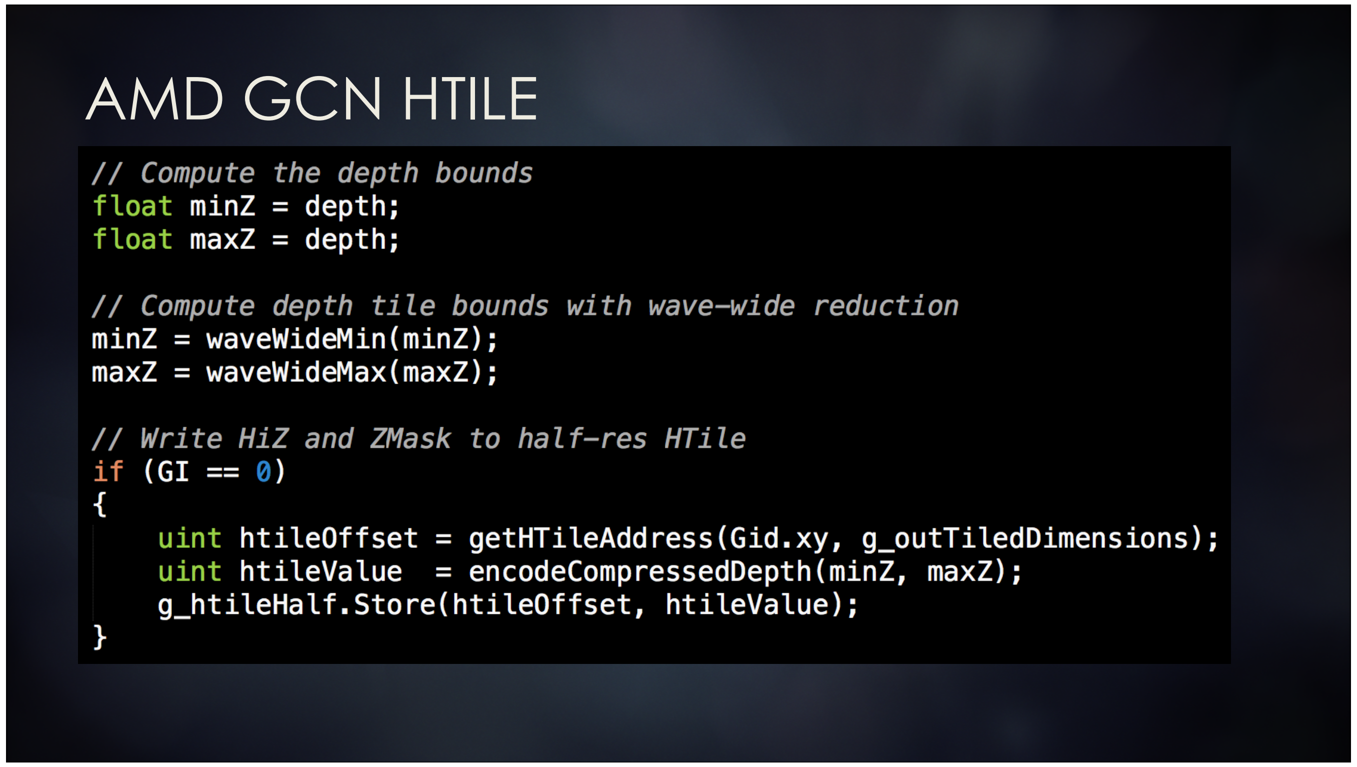


GCN has a depth acceleration meta data called HTILE [7] which accelerates regular GPU depth operations. Every 8x8 group of pixels has a corresponding 32 bit meta data block. While this meta data accelerates regular GPU depth operations, it can be decoded manually in a shader and used for early rejection of 64 pixels with a single test, or for any other relevant purpose.

HTILE is usually imprecise, and the bounds must be conservative. Additionally, the bounds can only grow until you “resummarize”, where every depth value must be read in order to recompute the bounds.

On consoles, HTILE is used to give us conservative depth testing without having to decompress the depth buffer for testing in a shader, or disabling Hi-Z on subsequent depth enabled render passes. We have a decompression compute shader which binds the HTILE surface as an R32 UINT texture, manually decodes the tile information, and produces a depth texture.

There are some gotchas with using HTILE, but manual HTILE decoding or encoding can be a big performance win in a variety of scenarios. Currently, HTILE is only directly accessible to console developers.



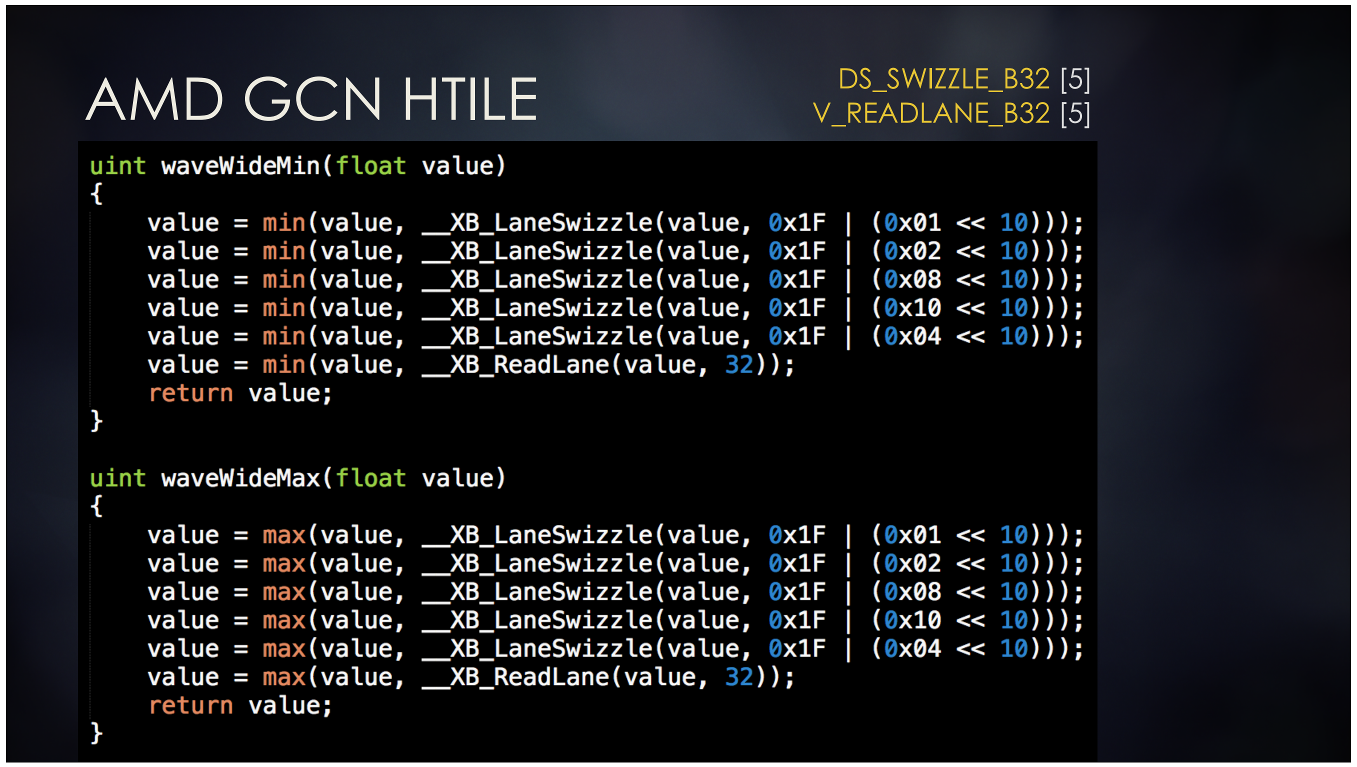
AMD GCN HTILE

```
// Compute the depth bounds
float minZ = depth;
float maxZ = depth;

// Compute depth tile bounds with wave-wide reduction
minZ = waveWideMin(minZ);
maxZ = waveWideMax(maxZ);

// Write HiZ and ZMask to half-res HTile
if (GI == 0)
{
    uint htileOffset = getHTileAddress(Gid.xy, g_outTiledDimensions);
    uint htileValue  = encodeCompressedDepth(minZ, maxZ);
    g_htileHalf.Store(htileOffset, htileValue);
}
```

When computing the first downsampled mip level of our Hi-Z pyramid, we can leverage the fact we've already read the input depth values. So we can also perform full and\or half res linearization of the depth values, and we can also write out half resolution HTILE so other passes like particles can use Hi-Z culling against that mip level, without needing to resummarize the half resolution depth buffer.

Since we need to build each HTILE meta data block from 64 pixels, we can't just use our already reduced 4 to 1 min\max values. We need to parallel reduce all pixels in an 8x8 tile to produce the correct min and max values for HTILE.

You could do a parallel reduction in LDS like all the cool kids do, or you could be even more awesome and do it with lane swizzling.



AMD GCN HTILE

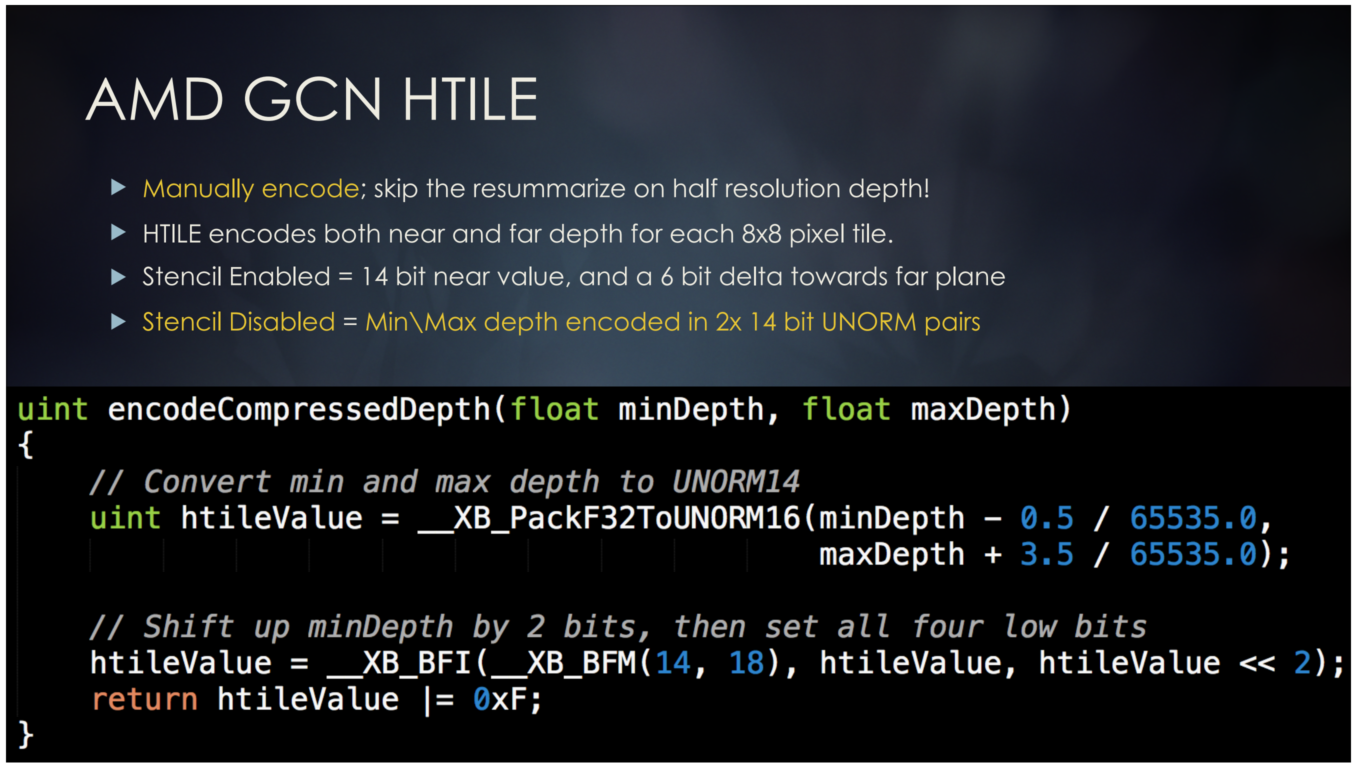
DS_SWIZZLE_B32 [5]
V_READLANE_B32 [5]

```
uint waveWideMin(float value)
{
    value = min(value, __XB_LaneSwizzle(value, 0x1F | (0x01 << 10)));
    value = min(value, __XB_LaneSwizzle(value, 0x1F | (0x02 << 10)));
    value = min(value, __XB_LaneSwizzle(value, 0x1F | (0x08 << 10)));
    value = min(value, __XB_LaneSwizzle(value, 0x1F | (0x10 << 10)));
    value = min(value, __XB_LaneSwizzle(value, 0x1F | (0x04 << 10)));
    value = min(value, __XB_ReadLane(value, 32));
    return value;
}

uint waveWideMax(float value)
{
    value = max(value, __XB_LaneSwizzle(value, 0x1F | (0x01 << 10)));
    value = max(value, __XB_LaneSwizzle(value, 0x1F | (0x02 << 10)));
    value = max(value, __XB_LaneSwizzle(value, 0x1F | (0x08 << 10)));
    value = max(value, __XB_LaneSwizzle(value, 0x1F | (0x10 << 10)));
    value = max(value, __XB_LaneSwizzle(value, 0x1F | (0x04 << 10)));
    value = max(value, __XB_ReadLane(value, 32));
    return value;
}
```

Because each HTILE entry represents an 8x8 pixel block, we can use a wave wide min and max operation across 64 depth values in a tile using lane swizzle.

The DS_SWIZZLE_B32 instruction swizzles input thread data based on an offset mask and returns, without reading or writing DS memory banks.

Lane swizzle only works on 32 lanes, not 64, so we need to do a final combine which merges the first 32 lanes with the last 32 lanes. This is done with the read lane instruction, allowing us to grab the reduced value from another lane.



AMD GCN HTILE

- Manually encode; skip the resummarize on half resolution depth!
- HTILE encodes both near and far depth for each 8x8 pixel tile.
- Stencil Enabled = 14 bit near value, and a 6 bit delta towards far plane
- Stencil Disabled = Min\Max depth encoded in 2x 14 bit UNORM pairs

```
uint encodeCompressedDepth(float minDepth, float maxDepth)
{
    // Convert min and max depth to UNORM14
    uint htileValue = __XB_PackF32ToUNORM16(minDepth - 0.5 / 65535.0,
                                            maxDepth + 3.5 / 65535.0);

    // Shift up minDepth by 2 bits, then set all four low bits
    htileValue = __XB_BFI(__XB_BFM(14, 18), htileValue, htileValue << 2);
    return htileValue |= 0xF;
}
```

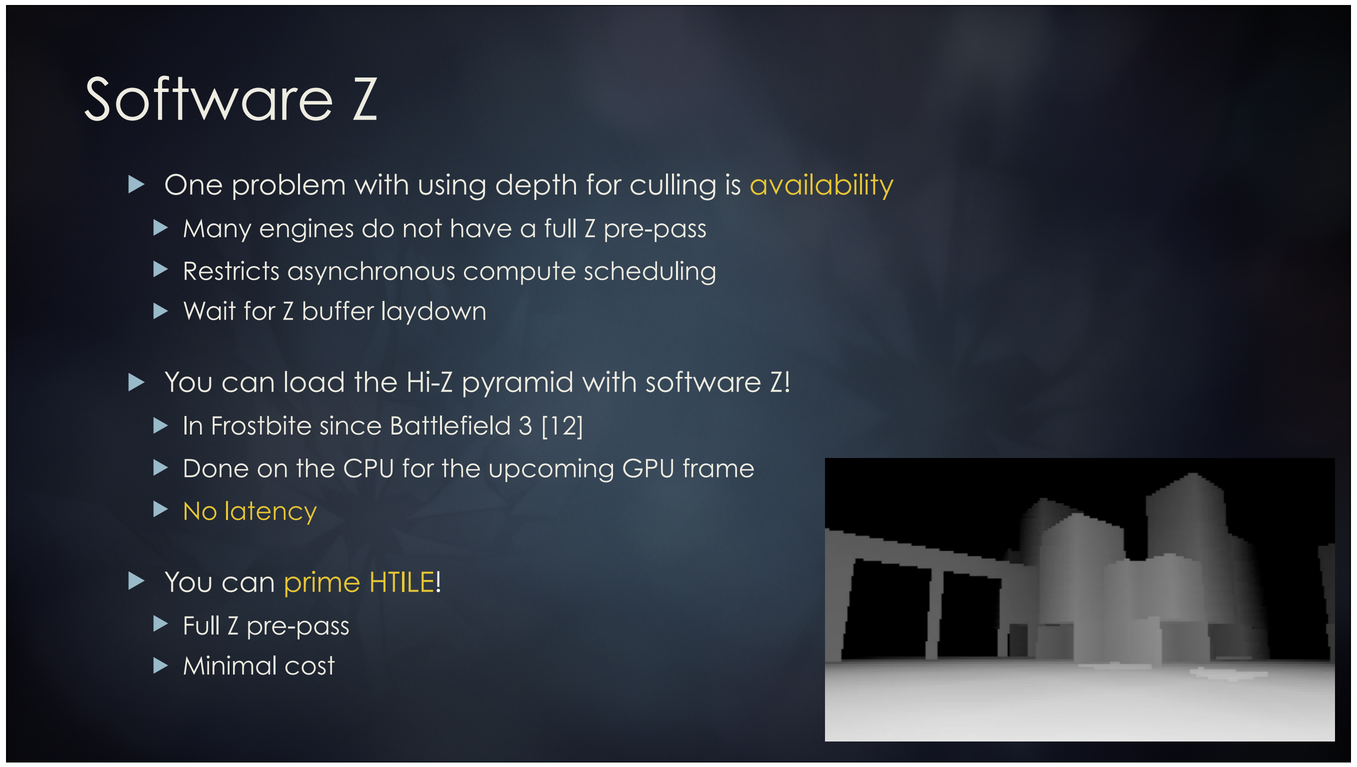
Rather than paying the cost of a depth read back during a resummarize, we can manually encode HTILE during the downsample operation.

HTILE encodes both near and far depth for each 8x8 pixel tile. Near is used for trivial accept, whereas far is used for trivial reject; anything in between these planes must do hi-resolution testing.

If stencil is enabled, we have a 14 bit near value, and a 6 bit delta towards the far plane.

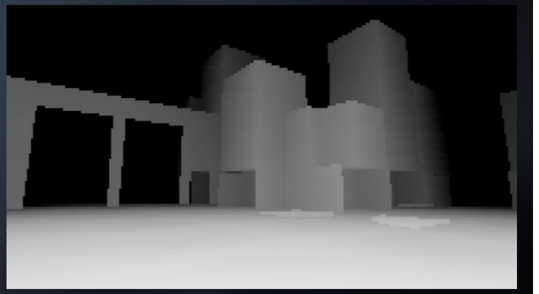
If stencil is not enabled, min and max depth is encoded into two 14 bit pairs. The bottom 4 bits in both cases is zMask, which we set to zero for clear.

Our Hi-Z pyramid doesn't need stencil, so this encoding routine is for the non Hi-Stencil format.



Software Z

- One problem with using depth for culling is availability
  - Many engines do not have a full Z pre-pass
  - Restricts asynchronous compute scheduling
  - Wait for Z buffer laydown
- You can load the Hi-Z pyramid with software Z!
  - In Frostbite since Battlefield 3 [12]
  - Done on the CPU for the upcoming GPU frame
  - No latency
- You can prime HTILE!
  - Full Z pre-pass
  - Minimal cost

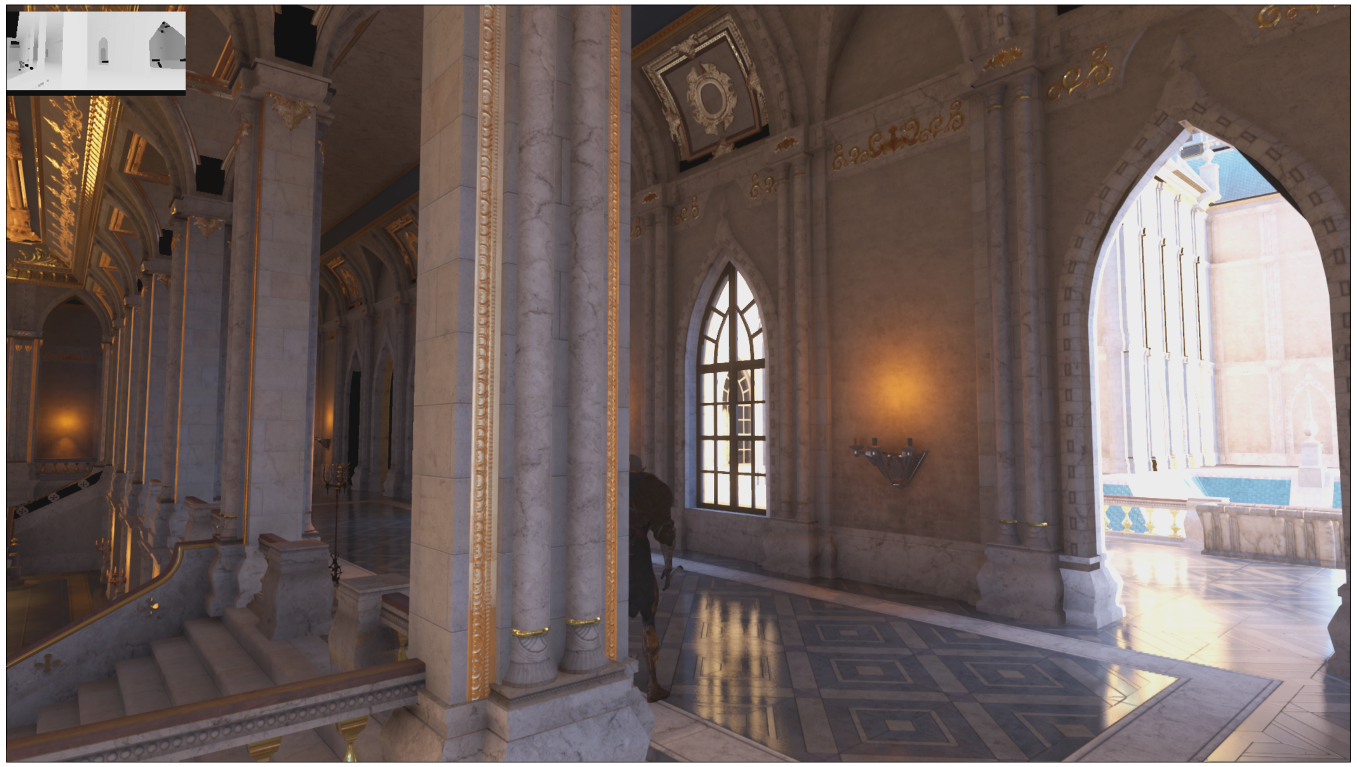


One problem with using depth for culling is availability. Many engines only have a partial z pre-pass, or none at all. This restricts how early you can kick off asynchronous compute work. You need to wait for Z buffer laydown before performing the depth test for culling.

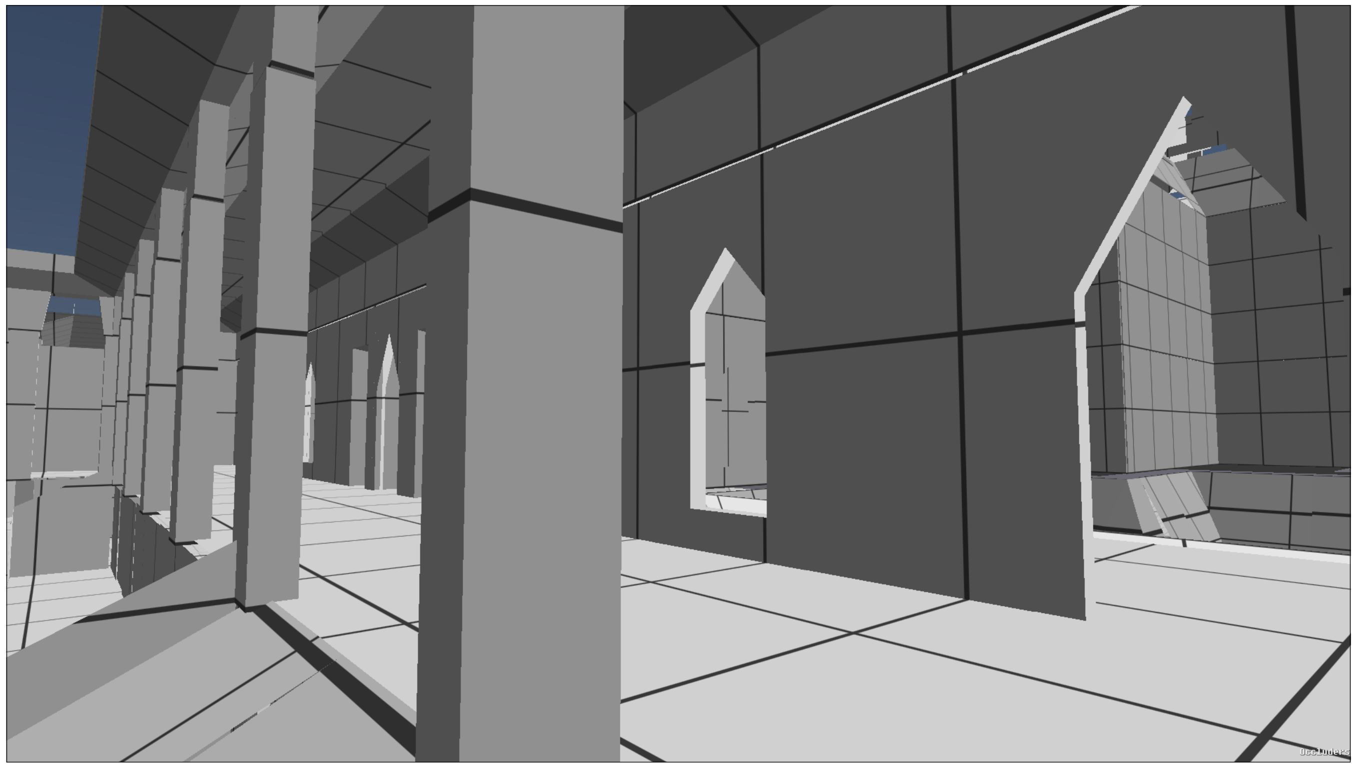
Frostbite has had a software rasterizer for occluders since Battlefield 3 [12], which is generated on the CPU for the upcoming GPU frame; the results of which can be used to the load Hi-Z pyramid prior to any related rendering passes, with no latency.

In addition to loading the Hi-Z pyramid, you can also use your software raster to conservatively prime your HTILE buffer as if you had a full pre-pass!

Without a software rasterizer or a full Z pre-pass, you can use a trick like re-projecting your previous depth buffer and testing with that.



This image shows Solas behind a pillar, and the results of the CPU rasterized occlusion buffer in the top left. Using this buffer, a Hi-Z pyramid was constructed, and the triangles for Solas are being depth tested against the appropriate mip level in this texture.

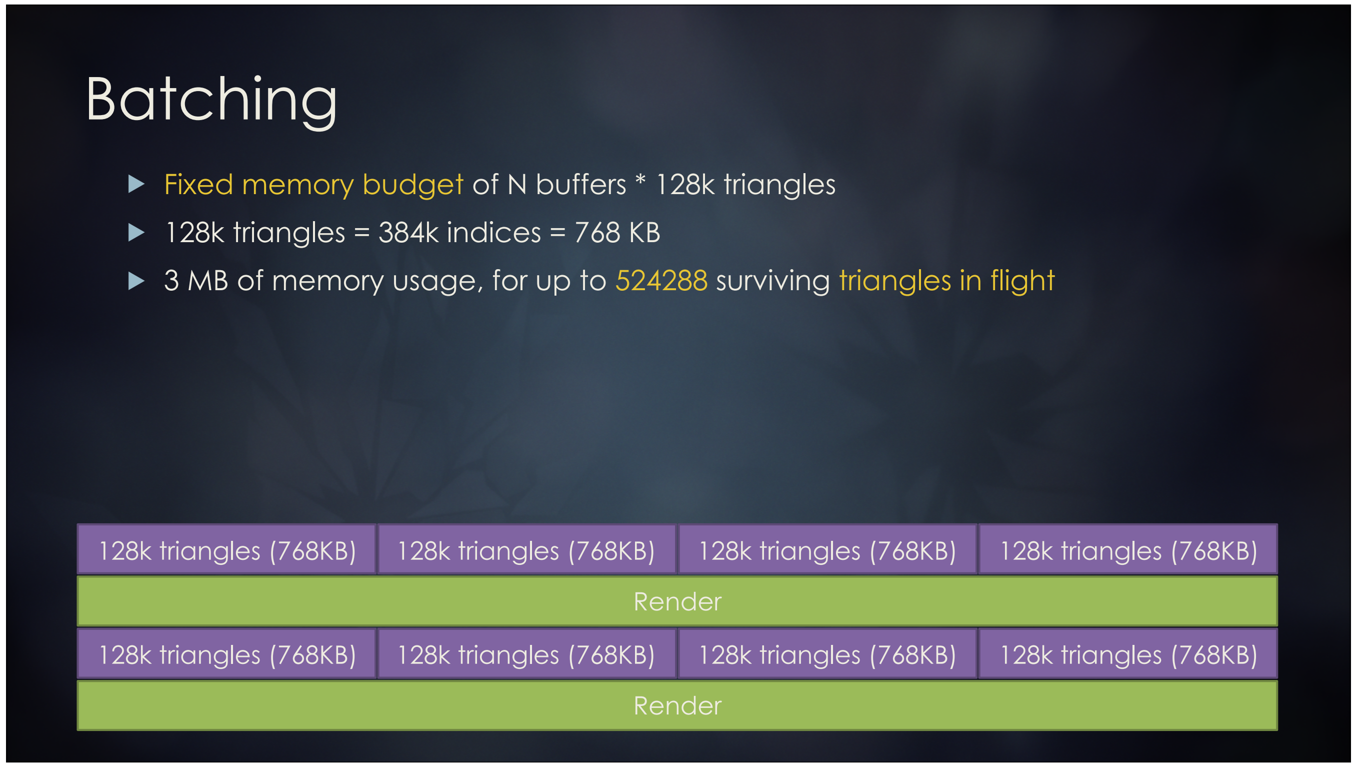


This image visualizes the occluder geometry used to produce the software occlusion buffer for this scene.

Locking the view, and moving to the other side of the pillar, shows the surviving triangles for Solas, and what was rejected by Hi-Z culling.

Batching and Perf

Lastly, I'm going to go over how the batching is structured to make the overlap of culling and rendering efficient.



Batching

- Fixed memory budget of N buffers * 128k triangles
- 128k triangles = 384k indices = 768 KB
- 3 MB of memory usage, for up to 524288 surviving triangles in flight

128k triangles (768KB)
128k triangles (768KB)
128k triangles (768KB)
128k triangles (768KB)
Render
128k triangles (768KB)
128k triangles (768KB)
128k triangles (768KB)
128k triangles (768KB)
Render

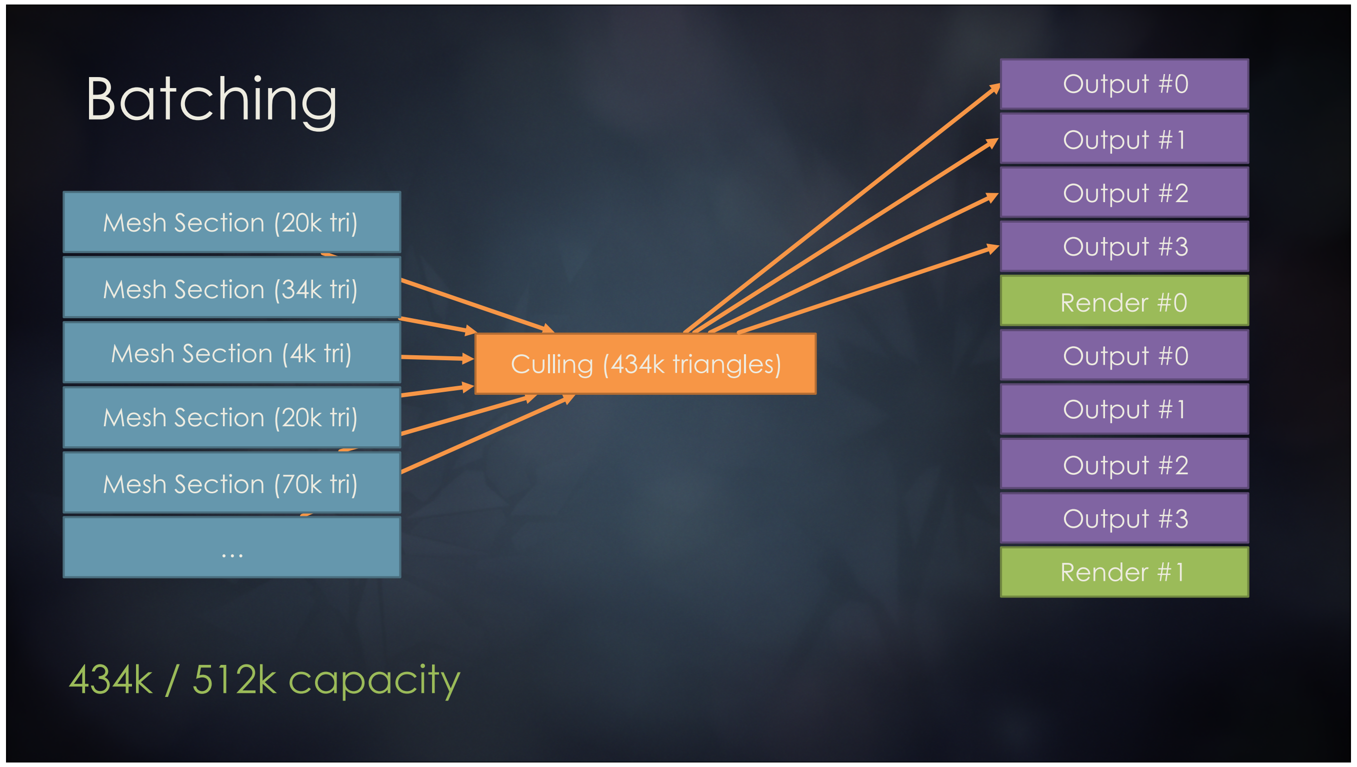
In order to efficiently run all the culling filters against a scene and render the results, the batching had to be carefully architected. The number of triangles in a scene can wildly vary between game teams or even different views, and predictable memory usage is desirable.

We start with a fixed memory budget of N buffers * 128k triangles, where N is high enough to get decent overlap between culling and render, and N should be at least 4. Doing a dispatch, wait, draw, loop would be bad, as that would cause the CP to stutter. We want to go a couple of dispatches ahead of render to account for this efficiently.

Assuming 16bit unsigned short, 384k triangle indices is 786Kb of memory.

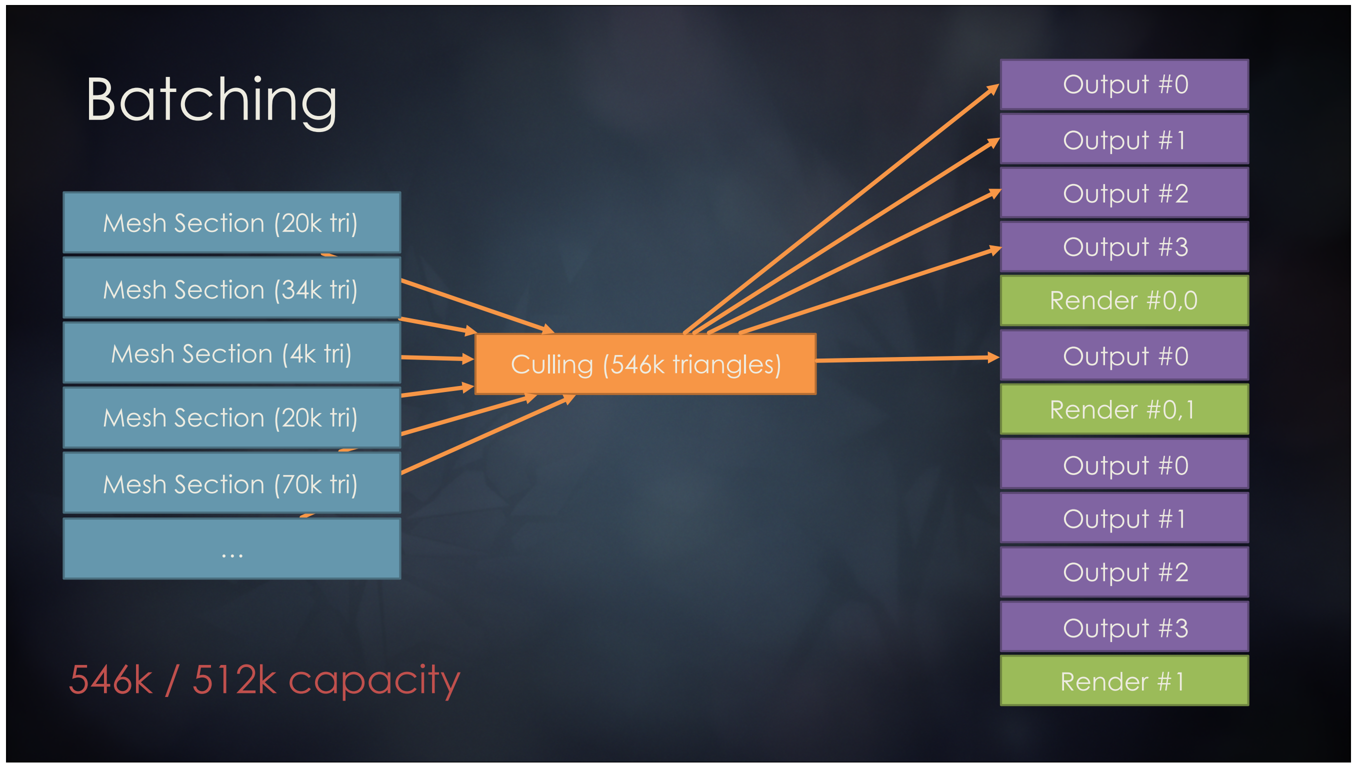
4 buffers is approximately 3MB, which allows for up to half a million triangles in flight. By sizing the buffers this way, and with careful scheduling, the data stays resident in the L2 cache when the vertex wavefronts execute.

Batching



In this example, we have 4 buffers, which gives us a total surviving triangle capacity of 512k. We have to calculate output requirements before culling, in case all triangles survive. I thought of doing a rough heuristic against 50% backface culled, but certain projections could cause problems.

You can see that culling is processing 434k triangles, which fits well within our 512k limit. Render #0 will occur, and then the next pass can reuse the output buffers. This leads into a more complex case…



Batching

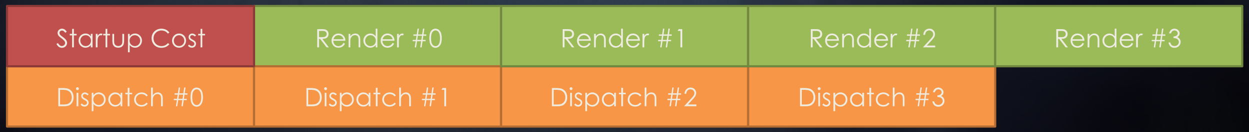
546k / 512k capacity

In this example, culling is processing more triangles than we have capacity for. When we determine that we've exhausted our buffers, we can do a mid-dispatch flush of the rendering. This will free up our output buffers for rendering the remaining triangles.

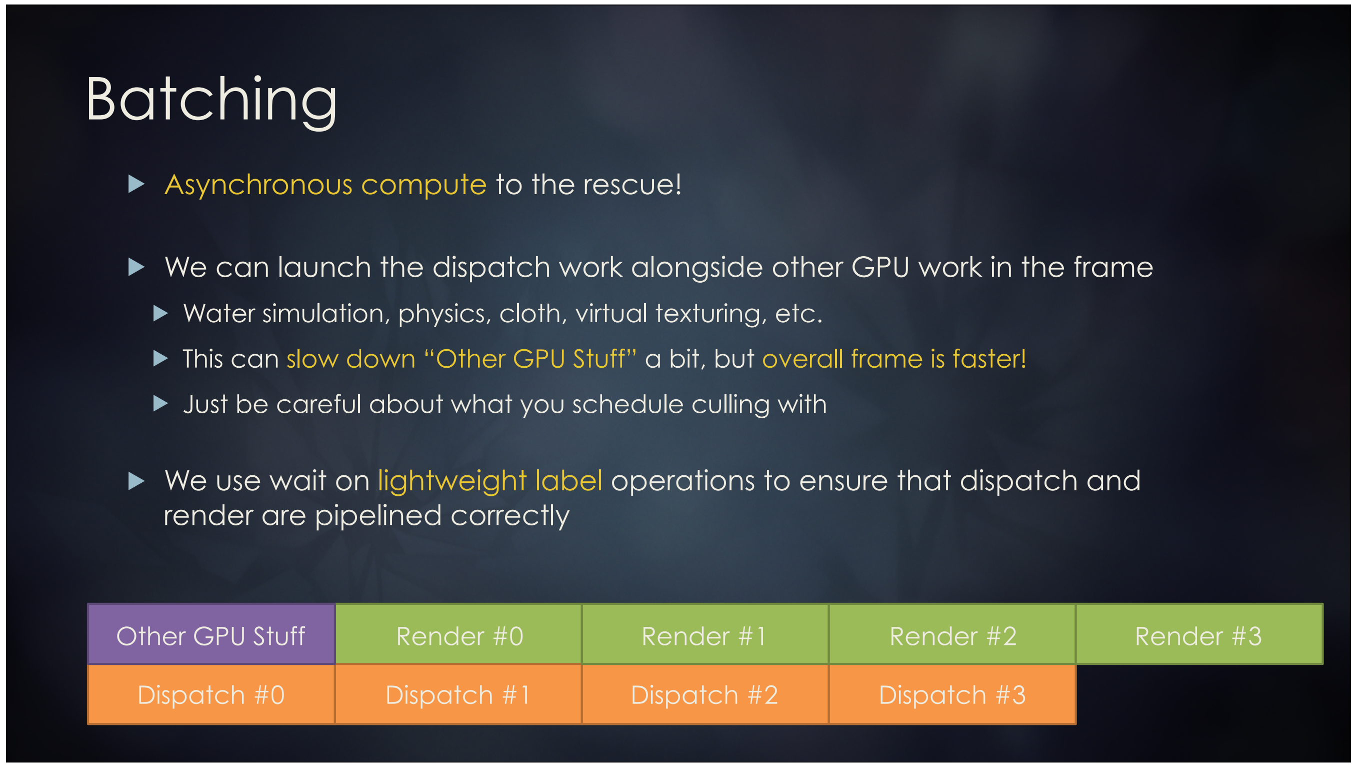
Using triangle lists is nice, because we can trivially cut up a mesh without concern, as long as we maintain ordering for optimal vertex reuse, or translucent objects.

Batching

- Overlapping culling and render on the graphics pipe is great
- But there is a high startup cost for dispatch #0 (no graphics to overlap)
- If only there were something we could use….



Overlapping culling and render wavefronts on the graphics pipe is great, but there is a high startup cost for the initial dispatch, when there is no graphics work to overlap. If only there were something we could use…



Batching

- Asynchronous compute to the rescue!
- We can launch the dispatch work alongside other GPU work in the frame
  - Water simulation, physics, cloth, virtual texturing, etc.
  - This can slow down "Other GPU Stuff" a bit, but overall frame is faster!
  - Just be careful about what you schedule culling with
- We use wait on lightweight label operations to ensure that dispatch and render are pipelined correctly

Other GPU Stuff	Render #0	Render #1	Render #2	Render #3
Dispatch #0	Dispatch #1	Dispatch #2	Dispatch #3	

Asynchronous compute to the rescue! We can launch the dispatch work alongside other GPU work in the frame, such as water simulation, physics, cloth, virtual texturing, etc. This can slow down some of the graphics pipe work a bit, but overall frame time is faster. Just be careful about "what" you schedule culling to run with.

We use inexpensive wait on label operations to ensure that dispatch and render are pipelined correctly. On PC we aim for fewer batches at a larger size due to the inability of DirectX12 to issue efficient mid command buffer fences.

In general, you want to schedule your async compute to happen at the same time as low-intensity rendering work, like a depth prepass or shadows. Use fences to bracket the dispatches so they don't start early or late on the GPU, and make sure to flush the async compute command buffer so it doesn't stall the GPU waiting on the auto-kickoff.

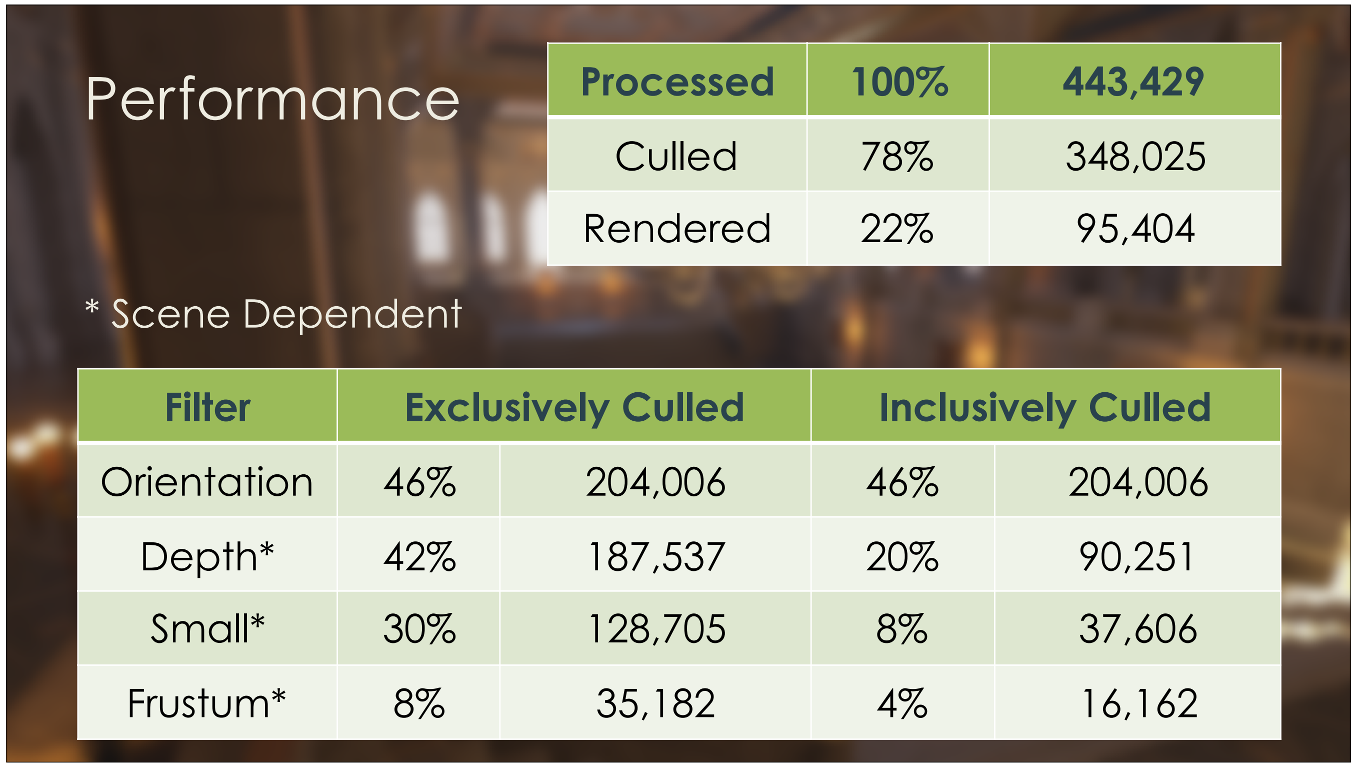
After that, you can use compute shader limit APIs to restrict the total number of thread groups per CU allowed or disable some CUs from either compute or graphics.

You can also kick off async compute to do the work during the last stages of post processing on the previous frame.

Performance

443,429 triangles @ 1080p
171 unique PSOs

For the performance figures, this is the test scene used. There are quite a number of render passes, and with 171 unique PSOs, but we’ll look at a single gbuffer pass of 450k triangles, rendered at 1080p on both platforms. For the Xbox One, the depth buffer and a few of the gbuffer color targets are in ESRAM, with everything else in DRAM.



Performance

Processed	100%	443,429
Culled	78%	348,025
Rendered	22%	95,404

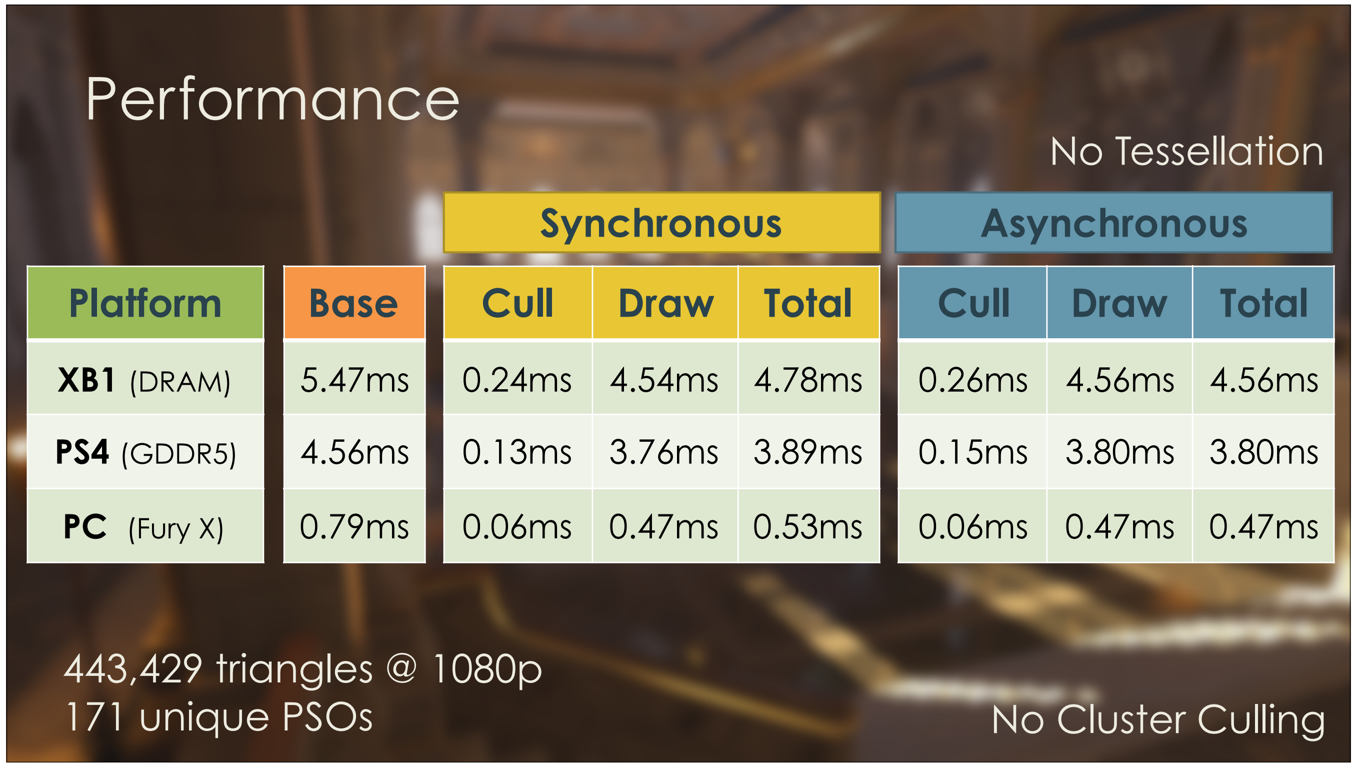
* Scene Dependent

Filter	Exclusively Culled		Inclusively Culled	
Orientation	46%	204,006	46%	204,006
Depth*	42%	187,537	20%	90,251
Small*	30%	128,705	8%	37,606
Frustum*	8%	35,182	4%	16,162

Aside from orientation culling, the other filters are very scene dependent. If you have a lot of dense meshes, the small primitive filter can be very effective, especially in the case of dense shadow maps. If you have aggressive view culling on the CPU, or in the coarse cluster culling pass, then the frustum culling may be less useful. However, once you have projected your vertices for the other filters, doing frustum culling is 4 cycles for 4 planes, so it doesn't hurt to leave it.

Next to orientation culling, the depth filter is the 2nd most effective, but is completely dependent on the quality of your depth buffer prior to culling. If you have a full z pre-pass, or can load it from software occluders or re-projected previous frame depth, then it may do wonders.

You'll notice that for this scene, we've managed to cull enough that we are only left with 22% of the original triangle count. Now imagine feeding the resultant culled index buffer into related passes, where we don't need to worry about the cost of culling.



Performance

No Tessellation

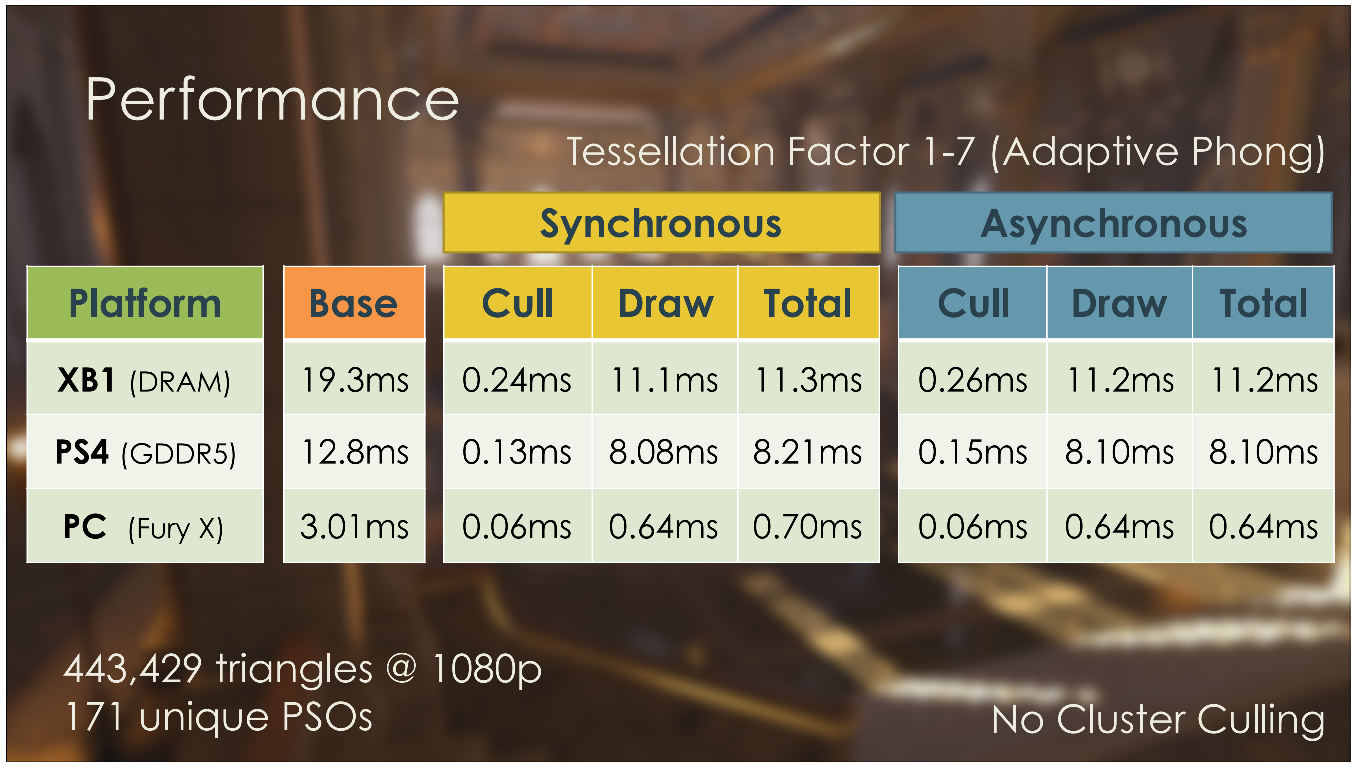
Platform	Base	Synchronous Cull	Synchronous Draw	Synchronous Total	Asynchronous Cull	Asynchronous Draw	Asynchronous Total
XB1 (DRAM)	5.47ms	0.24ms	4.54ms	4.78ms	0.26ms	4.56ms	4.56ms
PS4 (GDDR5)	4.56ms	0.13ms	3.76ms	3.89ms	0.15ms	3.80ms	3.80ms
PC (Fury X)	0.79ms	0.06ms	0.47ms	0.53ms	0.06ms	0.47ms	0.47ms

443,429 triangles @ 1080p
171 unique PSOs

No Cluster Culling

Here are the performance figures for this scene, on both consoles, and an AMD Fury X on PC. Even in DRAM, the XB1 culling is slowest, and barely any time at all to process half a million triangles in a gbuffer pass. Synchronously, we're saving 15-30% of our rendering cost, and asynchronously we're saving a bit more. The draw and cull times get a little bit longer when running asynchronously, but you'll notice that the overall cost goes down. This is due to some resource contention between compute and graphics.

A shadow or depth pass would improve performance even further than this, likely but an additional 10-15%, but I wanted to show the effectiveness of per triangle culling even in a color pass with varying PSO changes.



Performance

Tessellation Factor 1-7 (Adaptive Phong)

Platform	Base	Synchronous Cull	Synchronous Draw	Synchronous Total	Asynchronous Cull	Asynchronous Draw	Asynchronous Total
XB1 (DRAM)	19.3ms	0.24ms	11.1ms	11.3ms	0.26ms	11.2ms	11.2ms
PS4 (GDDR5)	12.8ms	0.13ms	8.08ms	8.21ms	0.15ms	8.10ms	8.10ms
PC (Fury X)	3.01ms	0.06ms	0.64ms	0.70ms	0.06ms	0.64ms	0.64ms

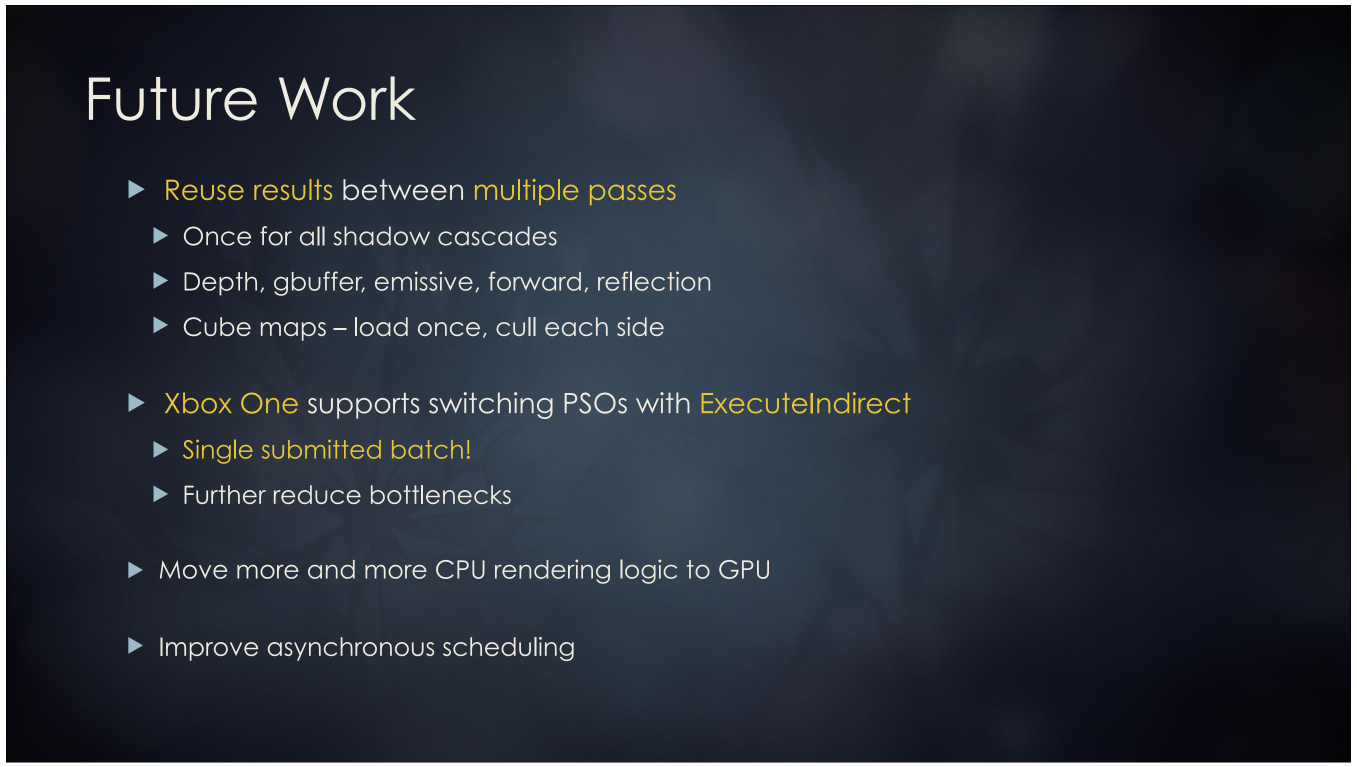
443,429 triangles @ 1080p
171 unique PSOs

No Cluster Culling

And here are the performance figures when we add a complex tessellation expansion factor to all the triangles. Specifically, a screen space adaptive phong tessellation with a factor no larger than 7. Here you'll see a massive increase in initial rendering cost, due to numerous hardware bottlenecks.

Because of this, our culling cost stays the same, as we are doing culling prior to tessellation, but the performance improvement to the final draw time is much higher, as the cost of rendering a surviving triangle is much more extreme.

In this scene, synchronously culling saves between 40-80% of the rendering time, and asynchronously it saves a bit more.

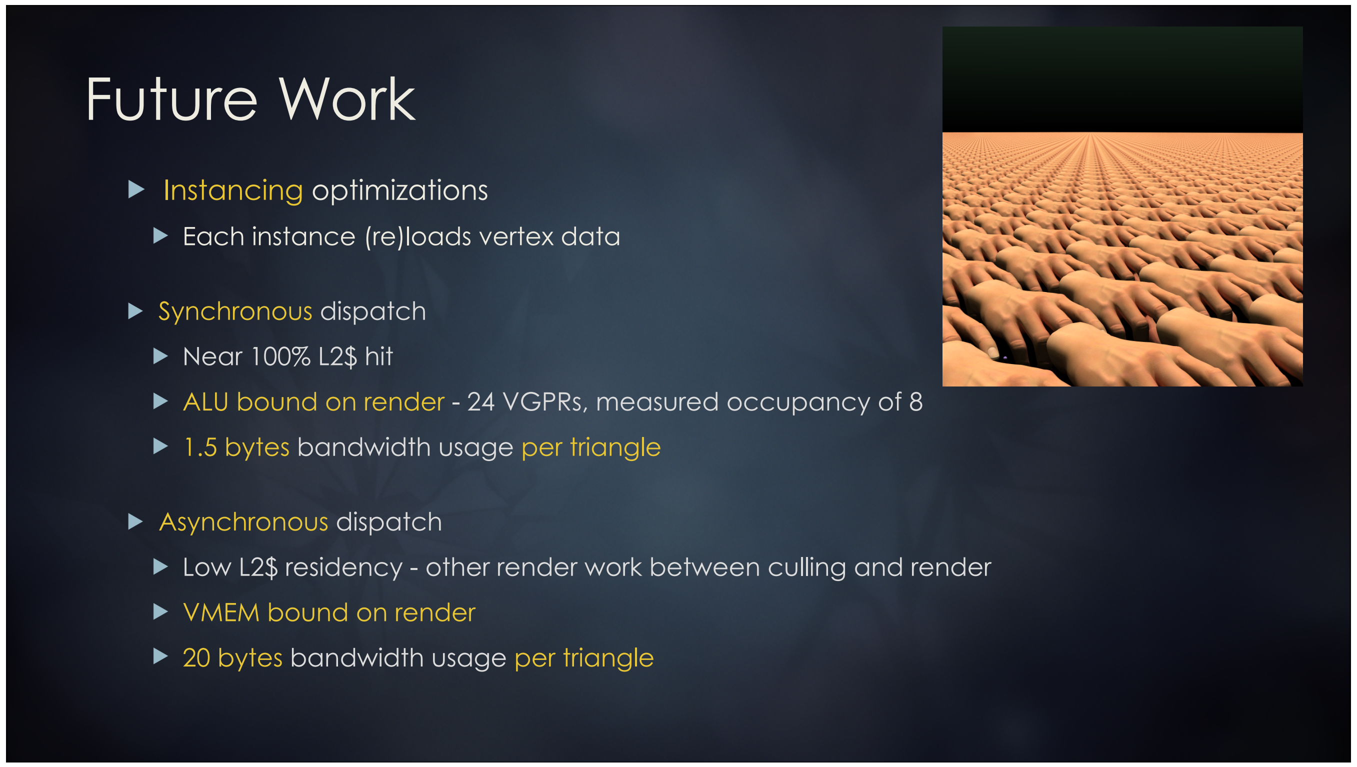

Future Work

- Reuse results between multiple passes
  - Once for all shadow cascades
  - Depth, gbuffer, emissive, forward, reflection
  - Cube maps – load once, cull each side
- Xbox One supports switching PSOs with ExecuteIndirect
  - Single submitted batch!
  - Further reduce bottlenecks
- Move more and more CPU rendering logic to GPU
- Improve asynchronous scheduling

It can be argued that traditional triangle processing in compute may not be the most effective use of the silicon, though aside from performance improvements, especially for shadow maps, this system serves as a platform for chaining other passes using the filtered index buffer for source triangles, instead of the unfiltered original index buffer.

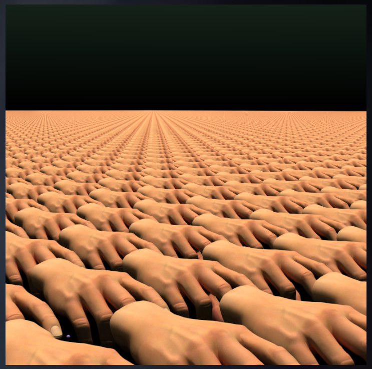
Additionally, the results from the culling can be resubmitted into subsequent passes from the same view, giving a performance amplification by skipping culling and reusing results.

The Xbox One supports switching PSOs per multi draw packet with ExecuteIndirect! This means we can submit a single batch, regardless of PSO differences, and further reduce bottlenecks. I can't stress how awesome this feature is, and we will definitely be using it going forward.



Future Work

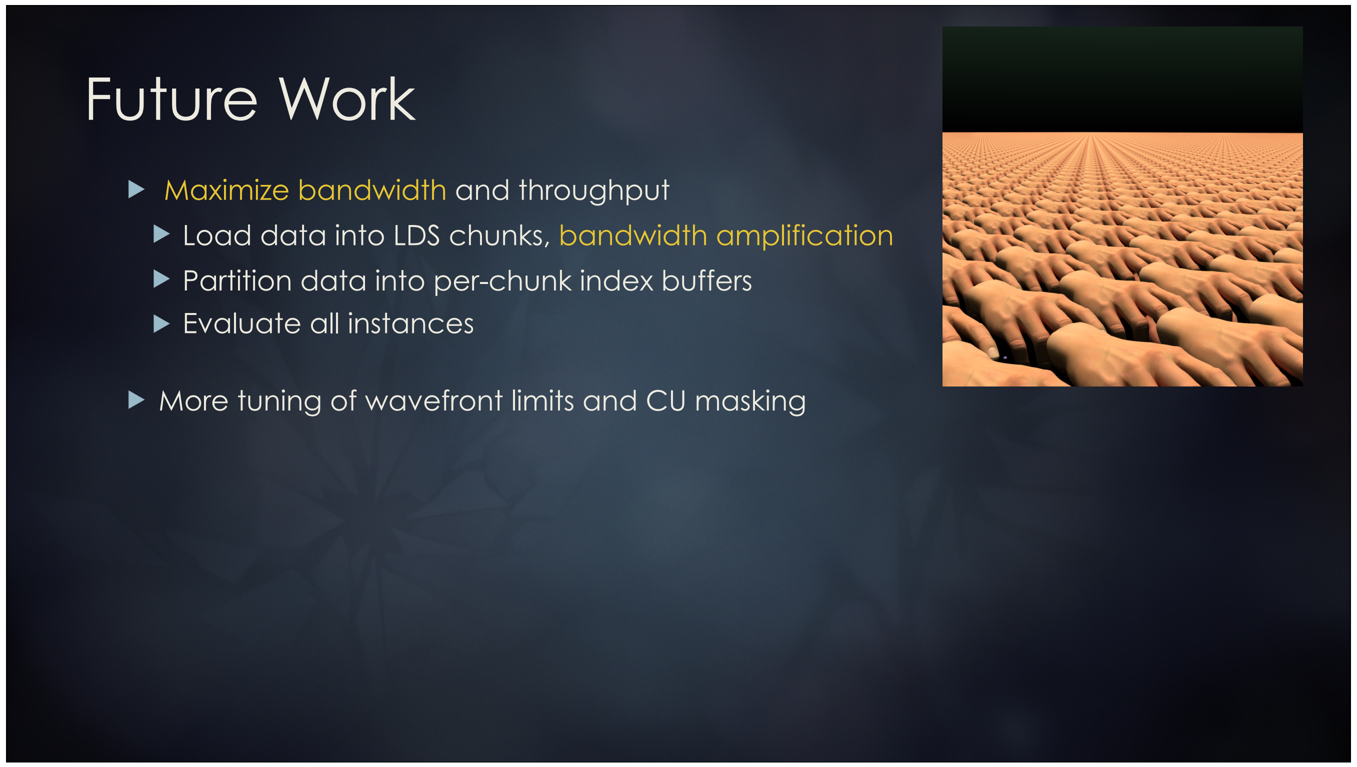
- Instancing optimizations
  - Each instance (re)loads vertex data
- Synchronous dispatch
  - Near 100% L2$ hit
  - ALU bound on render - 24 VGPRs, measured occupancy of 8
  - 1.5 bytes bandwidth usage per triangle
- Asynchronous dispatch
  - Low L2$ residency - other render work between culling and render
  - VMEM bound on render
  - 20 bytes bandwidth usage per triangle



For future improvements:

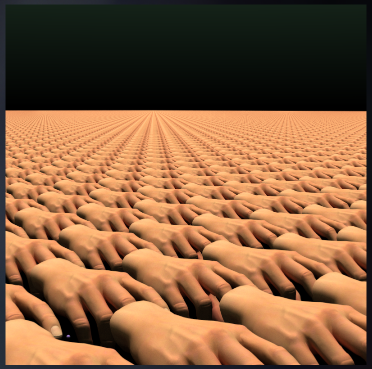
Each instanced draw is unrolled into multiple draws, since each instanced draw needs its own culled index buffer range. Instancing is primarily a CPU win, so the unrolling isn't an issue for that under DX12, except for the unnecessary memory pressure of each instance reloading the same vertex data. However, this system is getting incredible L2$ hits for the instanced data, when running synchronously.

With un-instanced data, I've measured about 20 bytes of bandwidth usage per triangle, but with instancing due to the batch chunk size and near perfect L2$ residency, I'm measuring 1.5 bytes of bandwidth usage per triangle, which is excellent. So nothing needs to be done in the synchronous case, but the asynchronous case can be improved a lot.



Future Work

- Maximize bandwidth and throughput
  - Load data into LDS chunks, bandwidth amplification
  - Partition data into per-chunk index buffers
  - Evaluate all instances
- More tuning of wavefront limits and CU masking



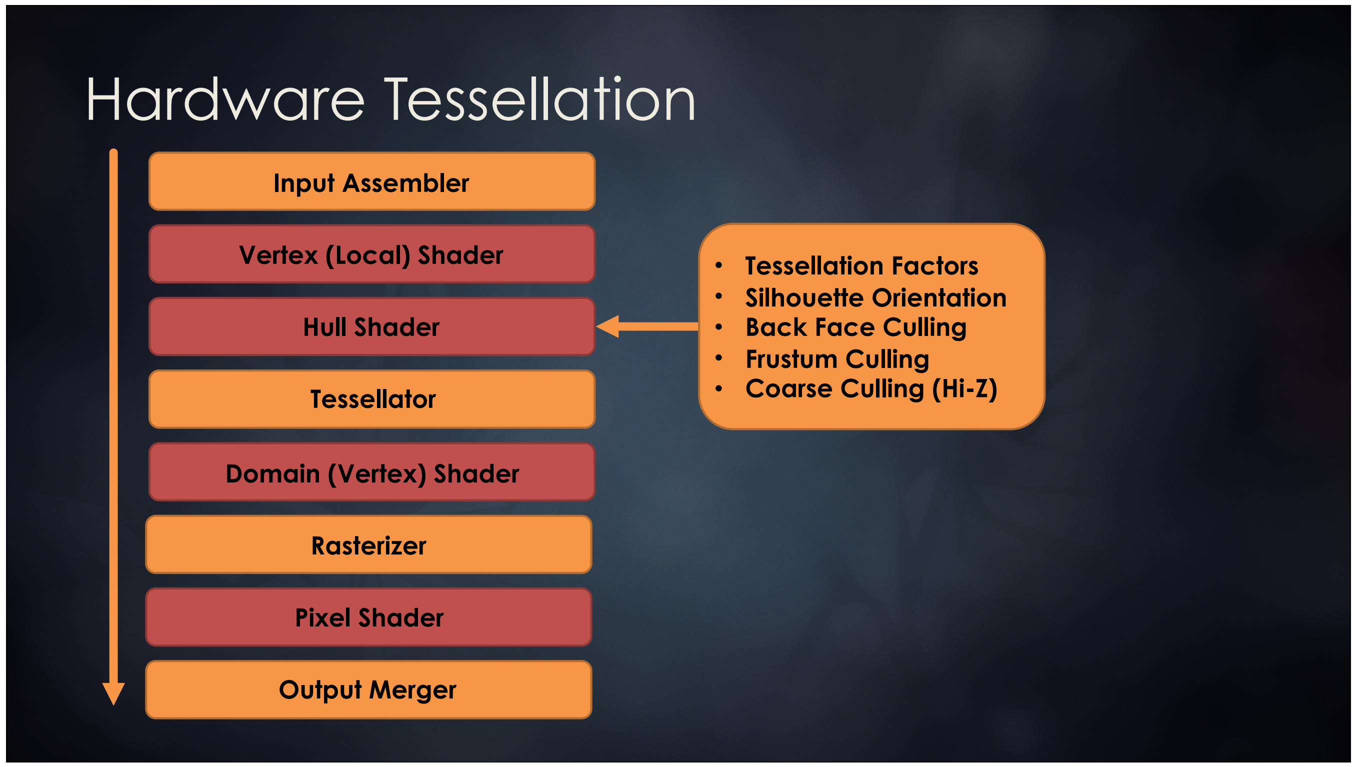
An improvement to instancing would be to load the vertex data once into chunks of LDS for bandwidth amplification, as each instance would perform culling against LDS loaded data.

We also want to investigate more at careful tuning of wavefront limits, and also CU masking.

Hardware Tessellation

Another interesting use case for compute mesh processing is to optimize hardware tessellation GPU bottlenecks. There are a number of cases where hardware tessellation can be extremely beneficial, especially when you are looking at optimizing content creation, procedural algorithms, or offloading CPU level of detail to the GPU.

It isn't for everyone, as even internally there are some titles that cannot afford the overhead, but I'm going to briefly mention some strategies I did to further improve performance when hardware tessellation is used, such as on Dragon Age: Inquisition, and Star Wars: Battlefront.



Hardware Tessellation

- Tessellation Factors
- Silhouette Orientation
- Back Face Culling
- Frustum Culling
- Coarse Culling (Hi-Z)

When using tessellation, the goal is to produce vertex waves at peak rate per SE. If not, then you want the reason to be “pixel waves are not draining fast enough”, i.e. the tessellation itself is not getting in your way.

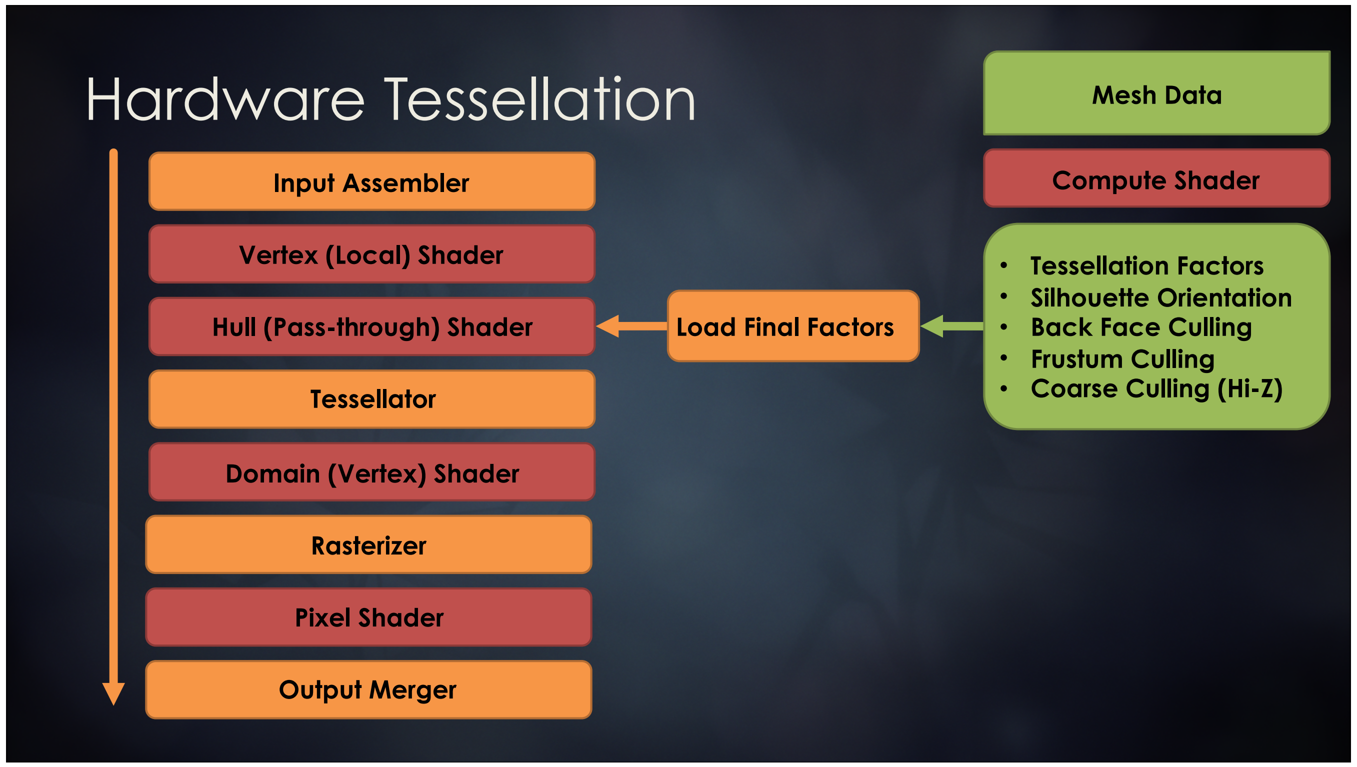
In a traditional hardware tessellation pipeline, the hull shader would do the heavy lifting of calculating adaptive screen space tessellation factors, as well as various patch level culling techniques. The calculated factors would range between 0 and whatever your max tessellation factor is set to.

Let me explain the two main reasons why hull shaders are so bad and why we want to move the work over to compute. Hull shaders tend to have very few active threads out of the 64 per wave. One issue is because the GPU can only fit so much control point data into LDS.

The other issue is because the shader compiler implements the patch constant function by, in the case of 3 vertex triangle patches, turning off 2 out of the 3 active threads, and only running code on the remaining thread.

Between these two problems, you are getting very low parallelism in what tends to be a very complex shader.

In general, the recommendation for small tessellation factors is to load as much data as late as possible so it happens after expansion, i.e. in the domain shader.



Hardware Tessellation

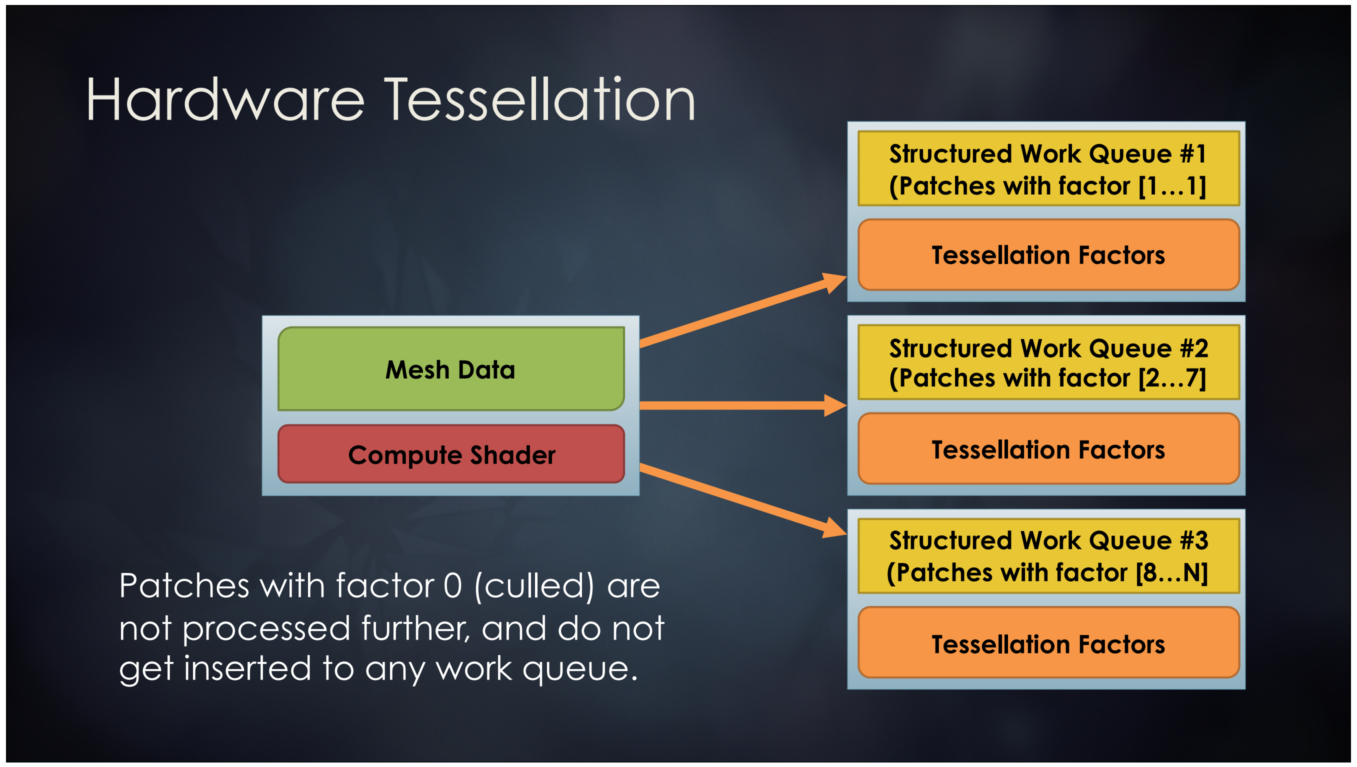
A first step optimization is to offload the work that the hull shader is doing, by moving these costly calculations to a compute dispatch earlier in the frame. The results would be stored into a factors buffer that the hull shader could then index with SV_PrimitiveId.

This optimization makes the hull shader stay active for the bare minimum amount of time, which is nice, but still suffers from high expansion bottlenecks and other inefficiencies. A factor of 0 would tell the hardware to cull the patch, and anything else would do a tessellated draw, including a factor of 1.

When I first started trying out tessellation on GCN, I expected some overhead, but I was shocked to find such a disparity between the cost of rendering a regular draw vs. a tessellated draw with a factor of 1. Low tessellation factors would perform reasonably well, but high tessellation factors performed very poorly.

Digging into it more, it turns out that vertex reuse is disabled at the vertex shader stage, and is instead enabled at the domain shader stage when the tessellation factor is greater than 1. This equates to about 3x more vertices!

Additionally, these factor 1 draws suffer from the same parallelism constraints that I just discussed.

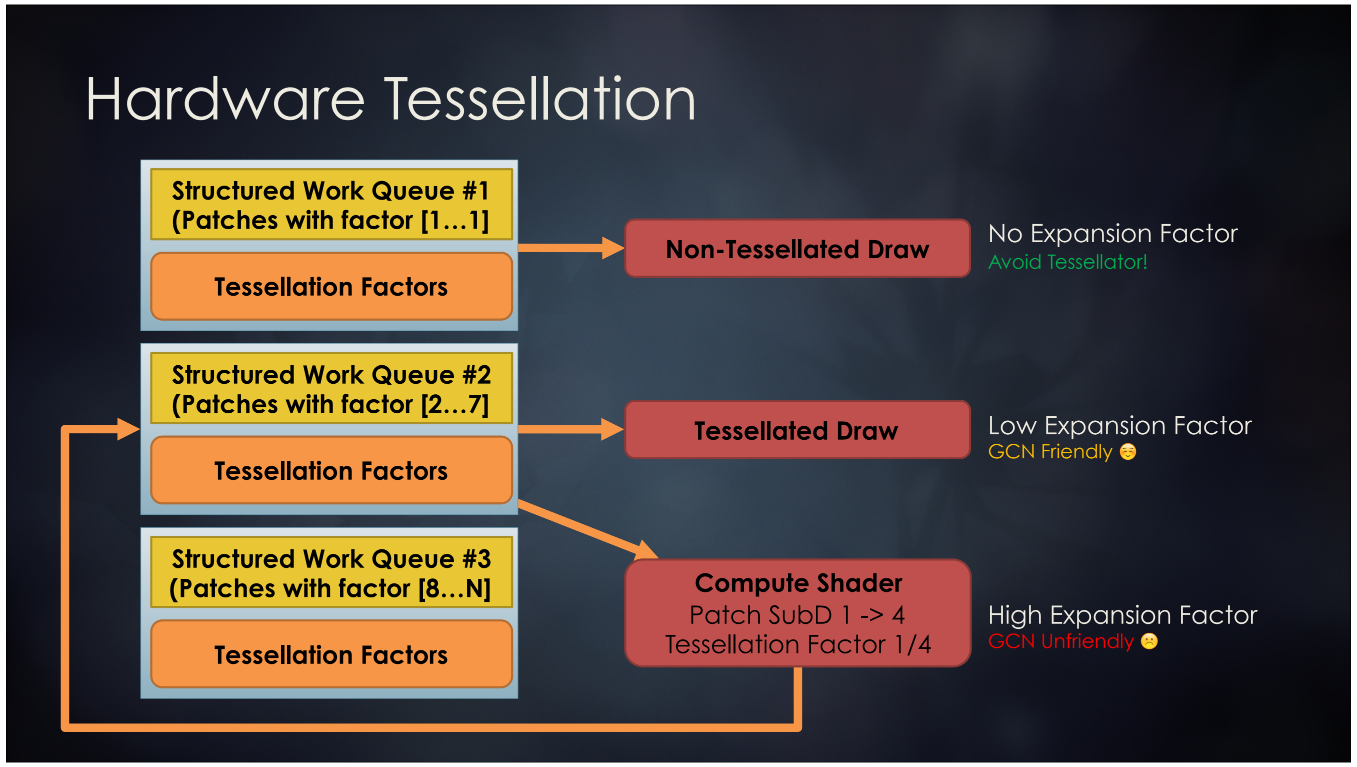


Hardware Tessellation

Mesh Data
Compute Shader
Structured Work Queue #1 (Patches with factor [1…1]
Tessellation Factors
Structured Work Queue #2 (Patches with factor [2…7]
Tessellation Factors
Structured Work Queue #3 (Patches with factor [8…N]
Tessellation Factors

Patches with factor 0 (culled) are not processed further, and do not get inserted to any work queue.

The improved optimization is to have a compute dispatch that, based on the compute tessellation factors, buckets the patches into one of three structured work queues. Culled patches with a factor of 0 are not processed further, and do not get inserted to any work queue.



Hardware Tessellation

Patches with a factor of 1 get placed into a queue that will be rendered without tessellation.

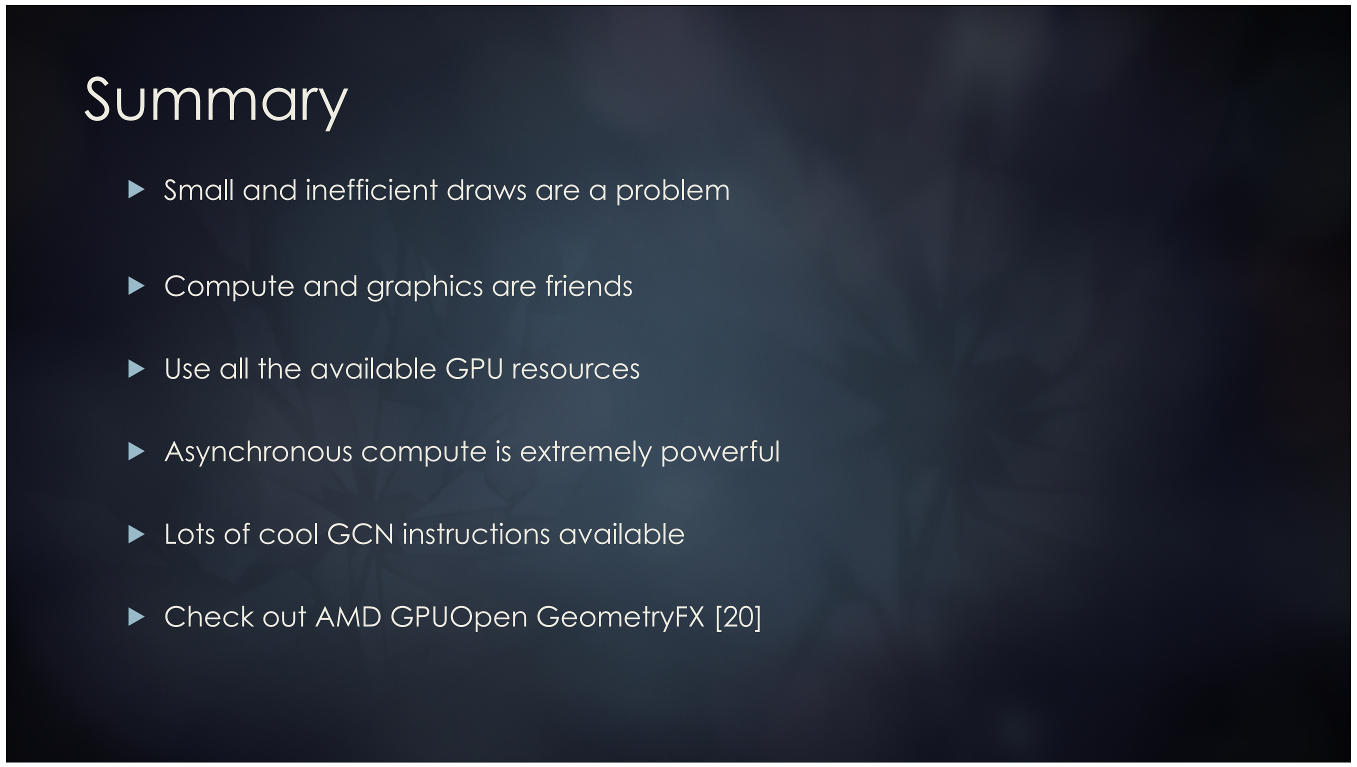
Patches with a factor of 2…7 get placed into a queue to be rendered with tessellation.

Patches with higher factors get placed into a queue that will undergo coarse refinement prior to tessellation [14].

The general goal here is to produce small patches, so that we can parallelize more of the mesh across more CUs. All the vertices heading into the domain stage need to be processed on the same CU, due to tessellation patch constants being stored in LDS. So the larger a patch, the less parallelism is achieved.

The compute shader will do a coarse subdivision of the patch into 4 smaller patches, and push them into the tessellation work queue with ¼ of the original tessellation factor.

One thing you need to handle is accounting for T-junctions between varying patch levels. Using an algorithm like PN-AEN will give you triangle patches which include edge adjacency information. This is helpful for solving this issue.



Summary

- Small and inefficient draws are a problem
- Compute and graphics are friends
- Use all the available GPU resources
- Asynchronous compute is extremely powerful
- Lots of cool GCN instructions available
- Check out AMD GPUOpen GeometryFX [20]

In summary, small and inefficient draws are a problem. DirectX 12 gives us an API to submits tons of draws at great performance from the CPU, but the GPU can still choke on these.

Compute and rasterization are friends; treat your draws as data, and have both compute and graphics help each other out.

Use idle GPU resources to remove fixed function bottlenecks.

Asynchronous compute is extremely powerful - be sure to schedule compute wavefronts alongside the rest of your frame, but don’t forget that you can overlap compute and graphics work on the same pipe, many developers do not realize this.

Remember that there are lots of cool GCN intrinsics available to optimize with. Grab a coffee, sit on a comfortable couch with your laptop, and read through the entire GCN instruction set documentation. You’ll find all sorts of crazy things you can exploit. Also be on the lookout for AMD GPUOpen. Many of the intrinsics I covered today will be exposed soon on PC for you to utilize!

Lastly, if you are interested in implementing something similar to this tech, be sure to check out GPUOpen GeometryFX; which is much easier than reverse engineering Frostbite to get at our custom solution ☺️

MAKE RASTERIZATION GREAT AGAIN!

★★★★★★

Lets work together and make rasterization great again!

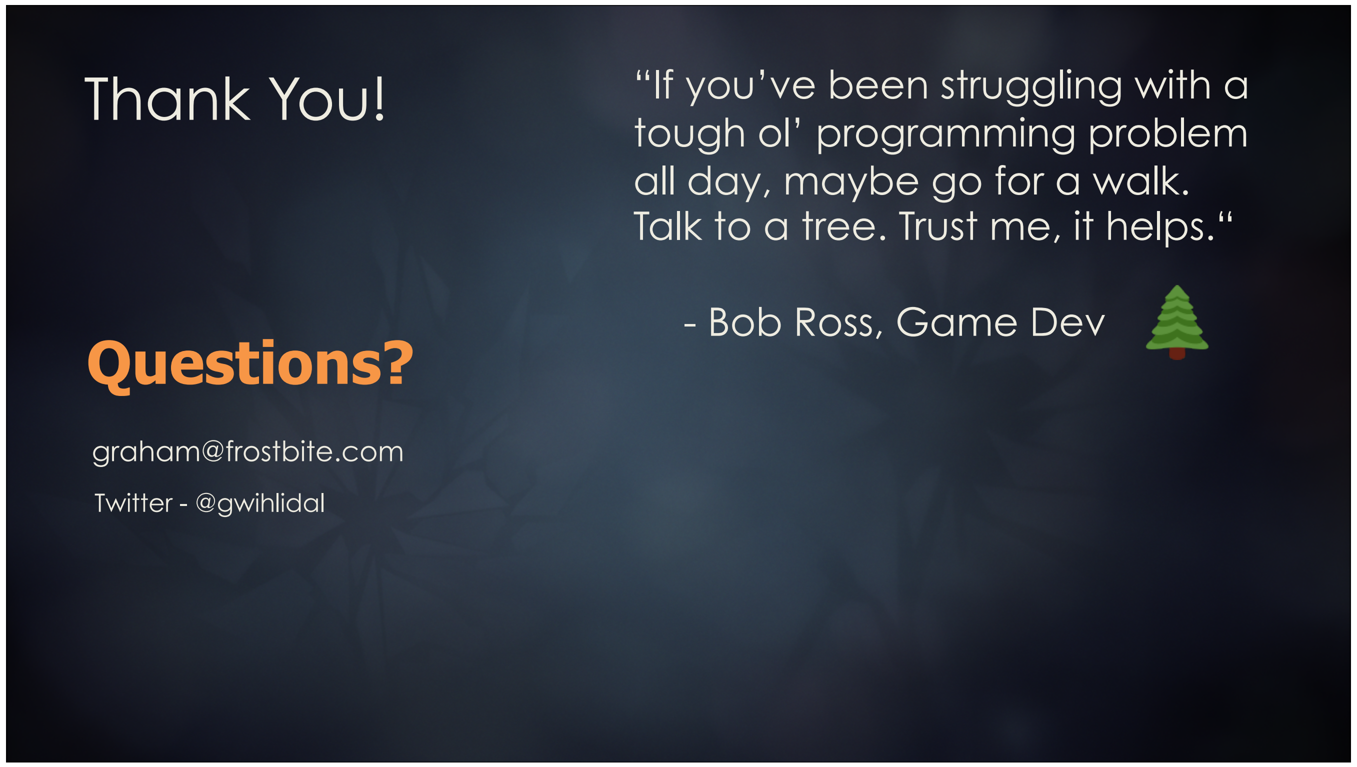
Acknowledgements

- Matthäus Chajdas (@NIV_Anteru)
- Ivan Nevraev (@Nevraev)
- Alex Nankervis
- Sébastien Lagarde (@SebLagarde)
- Andrew Goossen
- James Stanard (@JamesStanard)
- Martin Fuller (@MartinJIFuller)
- David Cook
- Tobias "GPU Psychiatrist" Berghoff (@TobiasBerghoff)
- Christina Coffin (@ChristinaCoffin)
- Alex "I Hate Polygons" Evans (@mmalex)
- Rob Krajcarski
- Jaymin "SHUFB 4 LIFE" Kessler (@okonomiyonda)
- Tomasz Stachowiak (@h3r2tic)
- Andrew Lauritzen (@AndrewLauritzen)
- Nicolas Thibieroz (@NThibieroz)
- Johan Andersson (@repi)
- Alex Fry (@TheFryster)
- Jasper Bekkers (@JasperBekkers)
- Graham Sellers (@grahamsellers)
- Cort Stratton (@postgoodism)
- David Simpson
- Jason Scanlin
- Mike Arnold
- Mark Cerny (@cerny)
- Pete Lewis
- Keith Yerex
- Andrew Butcher (@andrewbutcher)
- Matt Peters
- Sebastian Aaltonen (@SebAaltonen)
- Anton Michels
- Louis Bavoil (@LouisBavoil)
- Yury Uralsky
- Sebastien Hillaire (@SebHillaire)
- Daniel Collin (@daniel_collin)

I want to thank Christina Coffin and Mark Cerny for their mentoring of this presentation, Johan Andersson for allowing me to work on this research, Ivan Nevraev for fielding numerous driver and compiler requests, and I also want to thank the numerous people that offered guidance, ideas, improvements, support, and friendly conversation over a beer. Your help was much appreciated!

References

- [1] “The AMD GCN Architecture – A Crash Course” – Layla Mah
- [2] “Clipping Using Homogenous Coordinates” – Jim Blinn, Martin Newell
- [3] "Triangle Scan Conversion using 2D Homogeneous Coordinates“ - Marc Olano, Trey Greer
- [4] “GPU-Driven Rendering Pipelines” – Ulrich Haar, Sebastian Aaltonen
- [5] “Southern Islands Series Instruction Set Architecture” – AMD
- [6] “Radeon Southern Islands Acceleration” – AMD
- [7] “Radeon Evergreen / Northern Islands Acceleration” - AMD
- [8] “GCN Architecture Whitepaper” - AMD
- [9] “Optimizing Parallel Reduction In CUDA” – Mark Harris
- [10] “Hierarchical-Z Map Based Occlusion Culling” – Daniel Rákos
- [11] “Hierarchical Z-Buffer Occlusion Culling” – Nick Darnell
- [12] “Culling the Battlefield: Data Oriented Design in Practice” – Daniel Collin
- [13] “The Rendering Pipeline – Challenges & Next Steps” – Johan Andersson
- [14] “GCN Performance Tweets” – AMD
- [15] “Learning from Failure: … Abandoned Renderers For Dreams PS4 …” – Alex Evans
- [16] “Patch Based Occlusion Culling For Hardware Tessellation” - Matthias Nießner, Charles Loop
- [17] “Tessellation In Call Of Duty: Ghosts” – Wade Brainerd
- [18] “MiniEngine Framework” – Alex Nankervis, James Stanard
- [19] “Optimal Bounding Cones of Vectors in Three Dimensions” – Gill Barequet, Gershon Elber
- [20] “GPUOpen GeometryFX” – AMD
- [21] “Sample Distribution Shadow Maps” – Andrew Lauritzen
- [22] “2D Polyhedral Bounds of a Clipped, Perspective-Projected 3D Sphere” – Mara and McGuire
- [23] “Practical, Dynamic Visibility for Games” - Stephen Hill



Thank You!

“If you’ve been struggling with a tough ol’ programming problem all day, maybe go for a walk. Talk to a tree. Trust me, it helps.“

- Bob Ross, Game Dev 🌲

Questions?

graham@frostbite.com

Twitter - @gwihlidal

I’d also like to thank GDC for having me present today, Microsoft and AMD for allowing me to show a lot of “secret sauce”, and everyone here that attended my talk.

As a reminder, please fill out the electronic evaluation that was mailed to you.

And with that, I’d like to open this up to any questions you may have.

Instancing Optimizations

- Can do a fast bitonic sort of the instancing buffer for optimal front-to-back order
  - Utilize DS_SWIZZLE_B32
    - Swizzles input thread data based on offset mask
    - Data sharing within 32 consecutive threads
  - Only 32 bit, so can efficiently sort 32 elements
  - You could do clustered sorting
    - Sort each cluster's instances (within a thread)
    - Sort the 32 clusters