1

VI

Optimizing the Graphics Pipeline with Compute

Graham Wihlidal

1.1 Overview

Modern graphics cards have a large amount of generic and flexible computational power. However, regular rasterization-based rendering pipelines still require the usage of fixed function hardware in order to be efficient enough for real time. Due to the design constraints of fixed function hardware, numerous bottlenecks and limitations are present which hurt throughput. Often, when the fixed function units are active, there are inactive compute resources sitting idle. By scheduling compute units in parallel alongside the graphics units, we can optimize the overall pipeline, and ensure we are utilizing all the available hardware resources at any given time.

The initial version of this technology was aimed at current consoles and high end AMD PCs. As such, the optimizations and algorithms discussed are specific to AMD Graphics Core Next hardware (Figure 1.1) [Mah 2013].

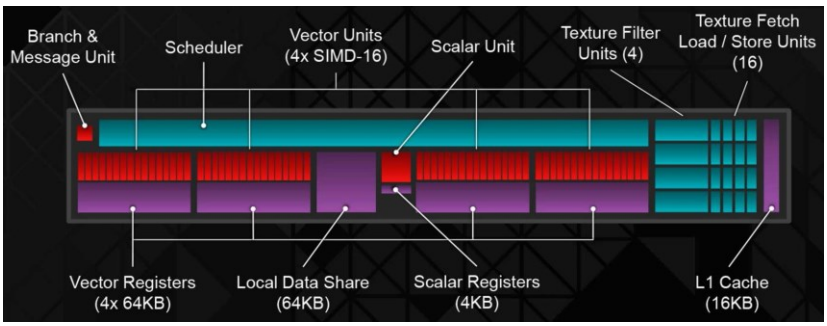


Figure 1.1. Diagram of an AMD GCN Compute Unit.

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

Table 1.1. AMD Graphics Core Next acronyms.

The definitions in Table 1.1 map to GCN hardware concepts, and are used throughout this chapter.

1.2 Introduction

While optimizing *Dragon Age*™: *Inquisition* (Figure 1.2), it was clear that displacement mapping was performing poorly on AMD hardware. Despite small improvements to the shaders to reduce LDS usage, it was obvious that performance was at the mercy of various bottlenecks. Experiments were done to offload

Figure 1.2. Screenshot of *Dragon Age*™: *Inquisition*.

Figure 1.3. Early triangle and patch-culling prototype.

the hull-shader adaptive tessellation factor calculations to a compute shader, and read the results back within the hull shader. This prototype was quite successful, and included interesting approaches like HTILE sourced Hi-Z culling of the triangle patches.

This then led to building a new prototype for regular triangle and patch culling, as a spiritual successor to LibEdge on PlayStation®3 (shown in Figure 1.3). This prototype was interesting; some scenes would be a win, and some would be a complete loss. By toying around with various GCN instructions, memory optimizations, and asynchronous compute, scenes were found which showed a significant win using compute-based triangle culling. These results motivated further extensive research and development into using the compute resources to supplement and optimize the fixed-function graphics pipeline and the integration of the technology into the cutting-edge Frostbite™ engine.

Engines typically use various methods for coarse culling on the CPU, prior to GPU submission. Due to latency between the CPU and GPU, many optimizations are inappropriate, or it would mean tight lock stepping. The CPU is a limited resource on consoles, and this is not a great use of a CPU core. On a PC, you have to get the data over the PCIE bus which would be prohibitive. Because of this limitation, it is ideal to have the culling happen on the GPU's timeline, so the solution is to do GPGPU submission.

Depth-aware culling can be performed, such as tightening shadow bounds or sample distribution shadow maps [Lauritzen et al. 2010]. Additionally, shadow casters without contribution, or hidden objects in the color pass, can be removed.

VR late-latch culling can also be performed by having the CPU submit a conservative frustum and having the GPU handle refinement. The idea of performing triangle and cluster culling on the GPU is covered in detail in this chapter.

Compute-shader mesh processing opens up additional opportunities for more efficiently supporting a variety of high-fidelity features and improvements, such as:

- offload tessellation hull-shader work;
- offload entire tessellation pipeline [Brainerd 2014];
- procedural vertex animation (wind, cloth, etc.);
- reusing results between multiple passes and frames;
- bounding volume generation;
- pre-skinning;
- generating GPU work from the GPU;
- scene and visibility determination.

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 GPU performs primitive assembly after the execution of multiple shader stages, so useless primitives will waste a lot of shader processing power. Reusing post-cull results between multiple passes will provide the GPU with the minimum amount of vertex data, resulting in performance amplification.

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

1.3 Motivation

Both Xbox One and the standard—as well as the PlayStation 4 Pro—have four shader engines, so a total of four triangles per clock.

By multiplying the number of compute units (CUs) by the number of VALUs per CU, and multiplying that value by two floating-point operations (FLOPS)—since one CU can execute 64 FMA in one cycle—we get the number of FLOPS that can be executed per cycle. There is a technical caveat to mention with this math. There are actually 4× as many waves running, but each VALU takes four clocks, so those two factors of four cancel out (see Table 1.2).

By taking the number of FLOPS that can be executed per cycle, and dividing that by the number of available shader engines, the result is the number of FLOPS that can be executed per triangle, shown in Table 1.3.

Finally, by dividing the number of FLOPS per triangle by the number of FLOPS per ALU, the result (shown in Table 1.4) is the final instruction upper-limit that is available for compute triangle culling, which still beats the fixed-function primitive setup and scan converter. Because of latency hiding, this is

Xbox One	12 CU * 64 ALU * 2 FLOPs **1,536** FLOPS / cy
PS4 Std	18 CU * 64 ALU * 2 FLOPs **2,304** FLOPS / cy
PS4 Pro	36 CU * 64 ALU * 2 FLOPs **4,608** FLOPS / cy
Fury X	64 CU * 64 ALU * 2 FLOPs **8,192** FLOPS / cy

Table 1.2. Number of FLOPs executed per cycle.

Xbox One	1,536 FLOPS / 2 shader engines **768** ALU ops per triangle
PS4 Std	2,304 FLOPS / 2 shader engines **1,152** FLOPS per triangle
PS4 Pro	4,608 FLOPs / 4 shader engines **1,152** FLOPS per triangle
Fury X	8,192 FLOPS / 4 shader engines **2,048** FLOPS per triangle

Table 1.3. Number of FLOPS executed per triangle.

not the actual duration, but rather, the VALU activity. It is also interesting to note that even though the PlayStation 4 Pro has 2× more compute units than the standard PlayStation 4, the addition of extra shader-engines results in the same instruction limit between both PlayStation 4 variants, as the ALU to geometry ratio is maintained.

Developers now have DirectX® 12 and Vulkan™, which promised thousands of draws, with very low overhead. 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 second 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

Xbox One	768 FLOPS / 2 FLOPS per ALU = **384 instruction limit**
PS4 Std	1,152 FLOPS / 2 FLOPS per ALU = **508 instruction limit**
PS4 Pro	1,152 FLOPS / 2 FLOPS per ALU = **508 instruction limit**
Fury X	2,048 FLOPS / 2 FLOPS per ALU = **1024 instruction limit**

Table 1.4. Compute culling instruction upper limit.

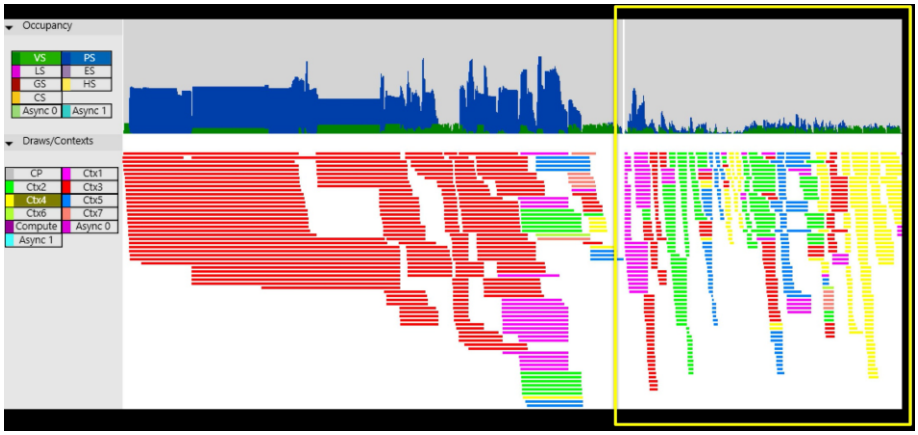


Figure 1.4. GPU spinning on empty vertex shader wavefronts.

vertex wavefronts. More draws are not necessarily a good thing. Figure 1.4 shows a GPU capture, where the left side starts out efficiently, but very quickly on the right, the GPU ends up spinning on vertex-shader wavefronts that that do not result in any pixels.

At a cursory glance, it seems quite easy to beat the peak primitive rate. However, the GPU has a lot more going on, so it is still important to profile and optimize the culling aggressively, especially bandwidth usage. A saving grace is that it is wildly optimistic to expect two triangles per clock cycle on consoles, as the rasterizer is subjected to other pipeline bottlenecks; on Xbox One, an actual rate of 0.9 triangles per clock was measured with regular rendering, which is really quite healthy, as primitive 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 vertex shader 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 vertex shader takes longer, primitive 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 due to large triangles. The coarse rasterizer only does one super-tile per clock, so triangles with a bounding rectangle larger than 32×32 will need to multi-cycle on the coarse rasterizer, reducing primitive rate.

Benchmarks that get very close to two primitives per clock (around 1.97) have these characteristics:

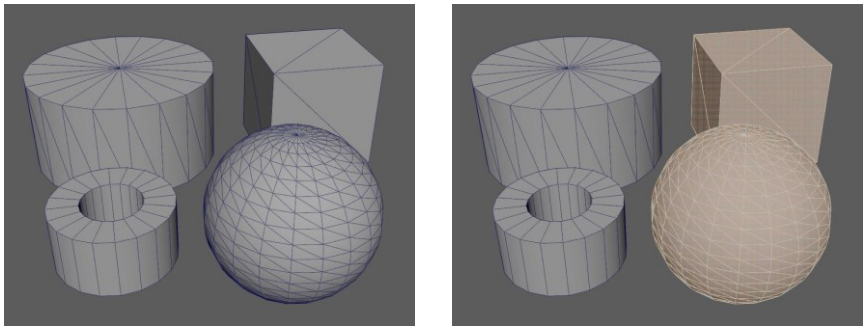
- Vertex shader reads nothing;
- Vertex shader writes only SV_Position;
- Vertex shader always outputs 0.0 for position
  - so every primitive is trivially culled;
- Index buffer is all 0
  - so every vertex is a cache hit;
  - cache hits do not count as vertices for purposes of peak vertex rate;
  - that is the only way to get near two primitives per clock without hitting two vertices per clock first;
- Every instance is a multiple of 64 vertices;
  - unfilled vertex shader waves are less likely;
- No pixel shader bound;
  - no parameter cache usage;
- The peak primitive rate also requires that nothing after the vertex shader causes a stall;
  - parameter size <= 4 × position size;
  - pixels drain faster than they are generated;
  - no scissoring occurs.

Apart from that, the PA can receive work faster than the vertex shader can possibly generate it. It is common to see tessellation achieve peak vertex-shader primitive throughput—for one shader engine at a time.

1.4 Draw Association

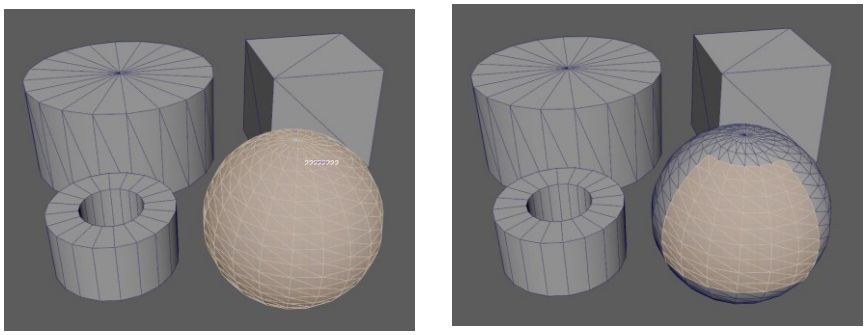
We review some terms in order to reduce ambiguity and confusion. A ***scene*** consists of a collection of meshes, displayed from a specific view (shown in Figure 1.5(a)).

A ***batch*** is a configurable subset of meshes in a scene (shown in Figure 1.5(b)). Except on Xbox One, all meshes in a batch are required to share the same shader; also 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 a PC.



(a) (b)

Figure 1.5. (a) A scene is a collection of meshes displayed from a particular view; (b) a batch is a configurable subset of meshes in a scene.



(a) (b)

Figure 1.6. (a) A mesh section represents an indexed draw call; (b) a work item represents a subset of triangles in a batch.

A batch here can be thought of as a near 1 : 1 with DirectX OR 12's Pipeline State Object concept (PSO).

A ***mesh section*** represents an indexed draw call. A mesh section has its own vertex buffers, index buffer, primitive count, and other properties (shown in Figure 1.6(a)).

Finally, there is a ***work item*** (shown in Figure 1.6(b)), which represents a subset of triangles in a batch that will be processed by the culling compute shader. 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 one triangle, and each work item processes 256 triangles.

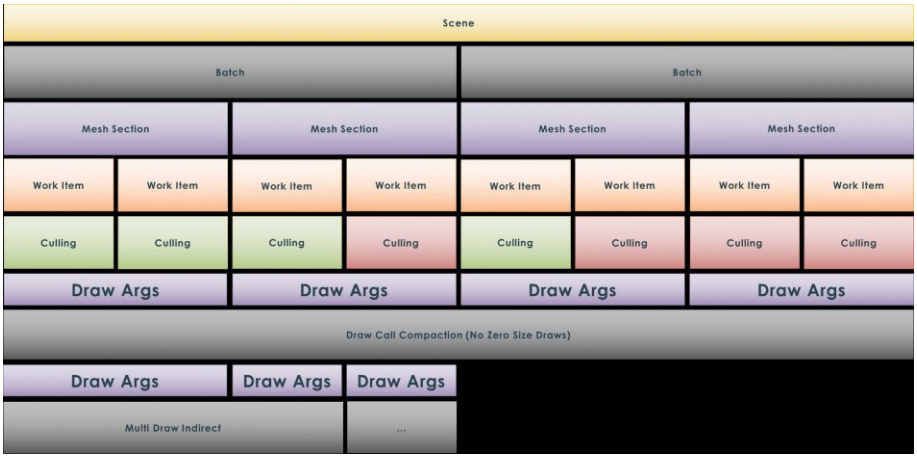


Figure 1.7. Overview of scene processed by culling system.

Figure 1.7 describes 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. A quick compaction pass is performed that ensures there are no zero-size draws if a mesh section is entirely culled (as in the case of occlusion or frustum culling).

At the end of the pipeline, there is a group of indexed draw arguments that can be kicked from the GPU with **ExecuteIndirect** on DirectX® 12, the **VK_AMD_draw_indirect_count** extension on Vulkan™, 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 a single **ExecuteIndirect** can be issued for the entire scene, regardless of state or resource changes.

Constructing each draw argument block is fairly straightforward, as it is 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, there is an instancing buffer that contains the transforms, colors, etc. per instance, but in this case it is no longer 1 : 1 with a draw call. Essentially, a custom 32-bit word needs to be added 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 a custom id can be packed alongside the other hardcoded draw indexed argument values.

On a 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

Figure 1.8. Multi-draw indirect structure layout.

gl_DrawId for this purpose. The command processor microcode on Xbox One handles indirect draws without intermediate steps or patching, which is the most optimal approach. An alternative would be to bind a buffer with a per-instance step rate of one, 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.

Listing 1.1 shows the appropriate command signature description to configure the dispatch to load the draw id into a register. 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 five arguments that make up a draw indexed packet (format shown in Figure 1.8).

```
D3D12_INDIRECT_ARGUMENT_DESC args[2];
args[0].Type = D3D12_INDIRECT_ARGUMENT_TYPE_CONSTANT;
args[0].Constant.RootParameterIndex = 9; // Multi Draw Id 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;
```

Listing 1.1. Example DirectX® 12 command signature for multi-draw id.

This mapping will cause the 0th word of your argument block to be loaded into an SGPR register for use by the shader (shown in Listing 1.2). On a 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, processing will remain on a fast path—similar to AGS MultiDrawIndirect or gl_DrawId.

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

Listing 1.2. HLSL syntax for accessing multi-draw id.

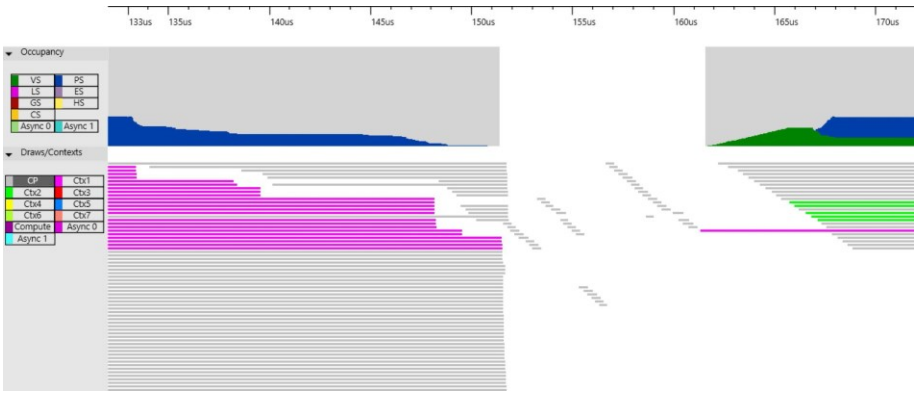


Figure 1.9. GPU zero-size draw inefficiency.

1.5 Draw Compaction

With cluster and triangle culling of draws on the GPU, it is extremely important to remove zero-sized draws from submission. The grey draws in the GPU capture shown in Figure 1.9 are empty indirect draws. At first, the command-processor (CP) cost is hidden by in-flight draws. Around 133 μs, the efficiency drops as the GPU hits a string of empty draws. At 151 μs, the GPU suffers around 10 μs of idle time. However, the total impact is worse than 10 μs, since the GPU does not instantly resume 100% efficiency, since it takes time to fill the CUs with waves. Clusters of culled draws can easily overwhelm the command processor, which is potentially 1.5 ms in a 60 Hz frame (seen in actual shipped games with real content).

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

The CPU will issue the worst-case number of draws, so zero-size draws will cause the GPU to process indirect arguments even if they have zero surviving primitives. The GPU needs control over the draw count and state changes. The `ExecuteIndirect` API in DirectX OR 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 (Listing 1.3). Some independent hardware vendors (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.

```
Count = min(MaxCommandCount, pCountBuffer)

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

Listing 1.3. Function prototype of DirectX® 12 `ExecuteIndirect`.

In order to optimize for empty draws, compaction is needed so that only draws with surviving triangles reach the GPU command processor. A cross-platform approach to draw compaction is to do a parallel reduction with atomics in group shared memory (shown in Listing 1.4); 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 arguments. Another barrier is performed so each thread gets the output location, and then the surviving draw arguments are written to the destination buffer.

In this example, batchData is the ExecuteIndirect count buffer, and the offset is the location of drawCountCompacted.

```
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);
}
```

Listing 1.4. Cross-platform compaction compute shader.

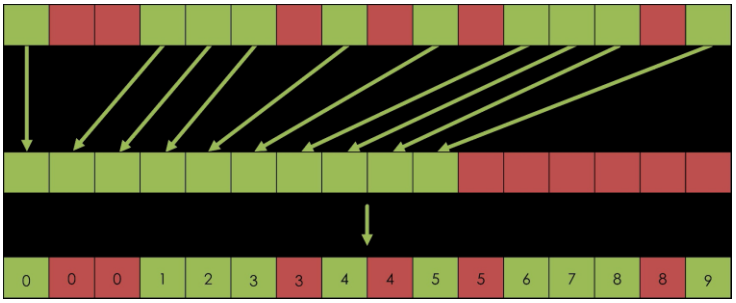


Figure 1.10. Expected result of index compaction.

With GCN intrinsics, and a thread group size of 64, we can do better! 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 would not be correct, and it is ideal to avoid global synchronization like the previous compaction algorithm. This is where parallel prefix sum comes to the rescue! In Figure 1.10, you can see the indices we want computed per thread in order for each active thread to write into the correct contiguous slot.

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

Figure 1.11 shows the results of `XB_Ballot64(threadId & 1)`, which is a predicate that sets 0 for even threads, and 1 for odd threads.

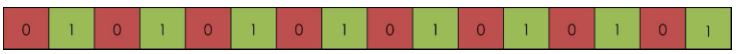


Figure 1.11. Result of simple ballot predicate.

Taking `ballot` further, Figure 1.12 is an example showing the results of a predicate running on thread 5. Before thread 5, there are three other threads that are valid, so the goal is to calculate the value three for thread 5's output slot.

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

In this example, the execution mask for thread 5 is shown, 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.

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 (0 – 31), and `V_MBCNT_HI` will produce a masked bit count of the

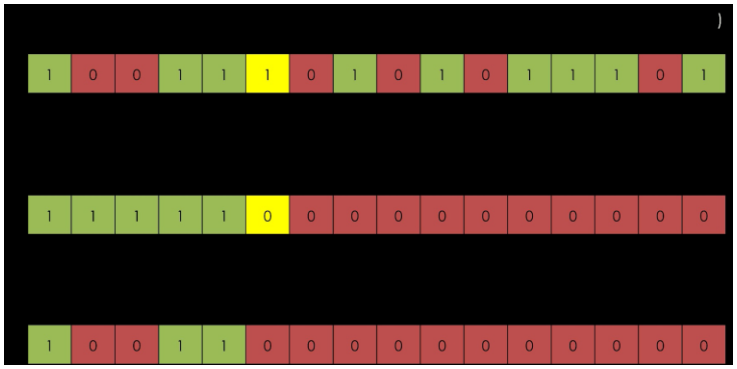


Figure 1.12. GPU population count example.

upper 32 threads (32 – 63). Chaining these instructions together will, for each thread, count the number of active threads which come before it, similar to the reference implementation in Listing 1.5.

```
uint2 mask;
mask.x = laneId >= 32 ? ~0 : ((1 << laneId) - 1);
mask.y = laneId < 32 ? 0 : ((1 << (laneId - 32)) - 1);
uint compactIndex = countbits(ballot.x & mask.x) +
                    countbits(ballot.y & mask.y);
```

Listing 1.5. Reference implementation of masked bit count.

The reference implementation can be completely replaced by the following:

```
uint compactIndex = XB_MBCNT64(ballot.xy);
```

Combining **ballot** and masked-bit count will compact the surviving draw-call stream within a wavefront without the need for any synchronization or group shared memory (shown in Figure 1.13).

The GCN-optimized compaction shader is shown in Listing 1.6; there are no longer any barriers. In order to compact across multiple wavefronts, there is 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, the value of **globalSlot** can be read directly from the lane that computed it.

```
[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;

```

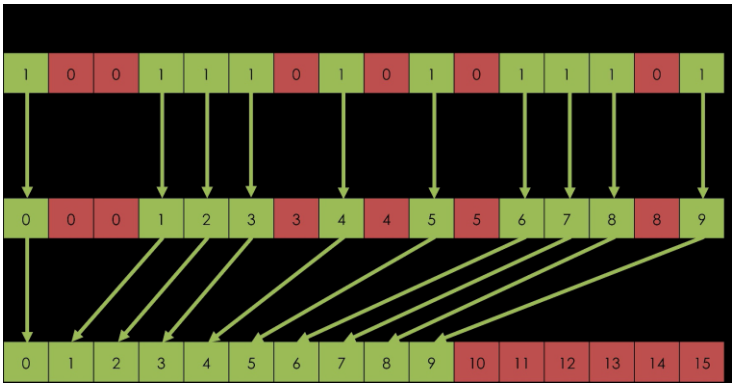


Figure 1.13. Masked bit count compaction.

```
10      MultiDrawIndirectArgs drawArgs;
11      if (drawArgId < drawArgCount)
12          loadIndirectDrawArgs(drawArgId, drawArgs);
13
14      const bool thisLaneActive = (drawArgs.indexCount > 0);
15      uint2 clusterValidBallot = __XB_Ballot64(thisLaneActive);
16
17      uint outputArgCount = __XB_S_BCNT1_U64(clusterValidBallot);
18
19      uint localSlot = __XB_MBCNT64(clusterValidBallot);
20
21      uint globalSlot;
22      if (laneId == 0)
23      {
24          InterlockedAdd(batchData[g_batchIndex].drawCountCompacted,
25                         outputArgCount, globalSlot);
26      }
27
28      globalSlot = __XB_ReadLane(globalSlot, 0);
29
30      if (drawArgId < drawArgCount && thisLaneActive)
31          storeIndirectDrawArgs(globalSlot + localSlot, drawArgs);
32  }
```

Listing 1.6. GCN-optimized compaction compute shader.

1.6 Cluster Culling

In order to make efficient use of the GPU, a coarse GPU culling pass is first performed on the mesh data [Haar and Aaltone 2015]. 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 [Barequet and Elber 2005]. 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 (shown

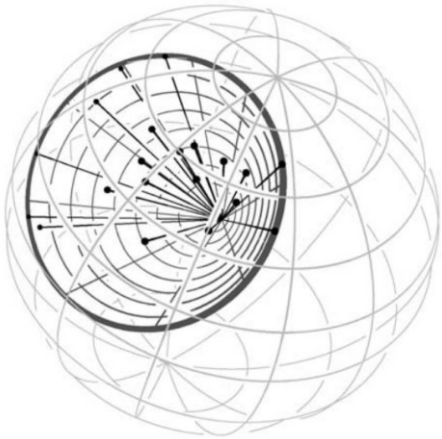


Figure 1.14. Minimum enclosing circle of unit sphere coordinates.

in Figure 1.14). 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.

The four-component 8-bit 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.

```
dot(cone.Normal, -view) < -sin(cone.angle)
```

You will 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 typical g-buffer encodings to improve normal accuracy would be applicable here, as long as they are not the ones that bias depth precision towards viewer facing.

Various configurations were profiled in order to determine the ideal cluster size. Using a cluster size of 64 is convenient on consoles, as doing so opens up intrinsic optimizations. However, this was found to be sub-optimal, since the CP bottlenecks on too many draws, and the culling was never bound by LDS atomics. Based on profiling, a cluster size of 256 seems to be ideal. 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.

This approach allows 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

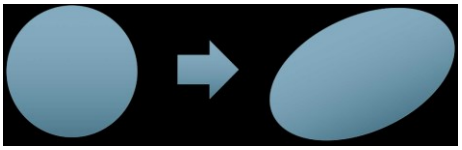


Figure 1.15. Spheres become ellipsoids under projection.

Hi-Z depth pyramid. Be careful to account for perspective distortion, as spheres become ellipsoids under projection [Mara and McGuire 2013], shown in Figure 1.15.

1.7 Triangle Culling

There are a number of filters applied to the mesh during the triangle-culling phase. Each thread in a wavefront processes one triangle. Various culling operations are applied, and the surviving triangles across a wavefront need to be balloted and 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 (Figure 1.16).

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` [Advanced Micro Devices 2012]. Profiling determined that using ordered append 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 ordered append 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 ordered append, you can optimize it further through carefully tuned wavefront limits.

Figure 1.17 shows an overview of operations that the cull shader is performing on one triangle per thread across each work item.

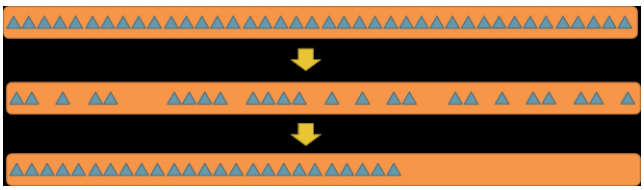


Figure 1.16. Triangles in a stream undergoing compaction after culling.

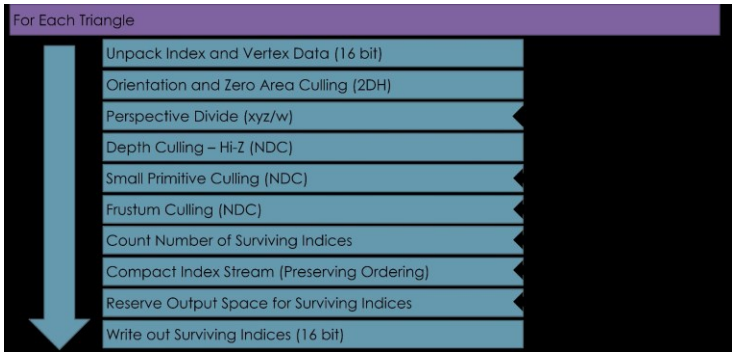


Figure 1.17. Overview of triangle cull shader operations.

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, the output buffer is first zeroed, the predicate & 1 is used on the thread id to determine low or high masking, and InterlockedOr is used on the output location. This cleverly uses the L2 cache as a write combiner.

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

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 all threads did not enter the if statement, and if so the compiler branches over the if statement.

```
1  if (allNotEntering)
2      Goto end;
3
4  if (threadEnters)
5      modifyExecMask and execute if-statement
```

Listing 1.7. Typical compiler-generated conditional tests.

Really, this is 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. This results in generated code that looks like Listing 1.7.

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 should only generate the single "if (allNotEntering) goto end;" part and skip the extra control-flow logic to handle divergence.

In the case of the culling work loop, ballot is used to force uniform branching and avoid divergence (including the slight code-gen hit), because there is 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.

1.7.1 Orientation Culling

On average, 50% of a mesh will be culled due to winding order. Therefore, we need a test which is as cheap as possible. One of the cheapest tests is the one shown in Listing 1.8 [Olano and Greer 1997], using the determinant of a 3 × 3 matrix with homogeneous coordinates [Blinn and Newell 1978]. This technique avoids clipping and projection, which includes quarter-rate reciprocal instructions coming from the perspective divide.

```
// Orientation and zero area (not small) test.
// Check the determinant of the 3x3 matrix of 2DH coordinates.
float det = determinant(float3x3(vertex[0].xyw,
                                 vertex[1].xyw,
                                 vertex[2].xyw));
bool cull = det <= 0.0f;
```

Listing 1.8. 2DH orientation and zero area test.

Using GCN specific optimizations, we can skip all the tests afterwards if winding order has 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.

Figure 1.18. Back-face determinant test scene.

Figure 1.19. Results of back-face determinant test.

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.

Figure 1.18 is an example of the back-face determinant test applied to a character (Solas) from a particular view. Locking the current view, and then moving behind the character shows (in Figure 1.19) that all back-facing triangles have been removed, as expected.

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 (Figure 1.20).

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.

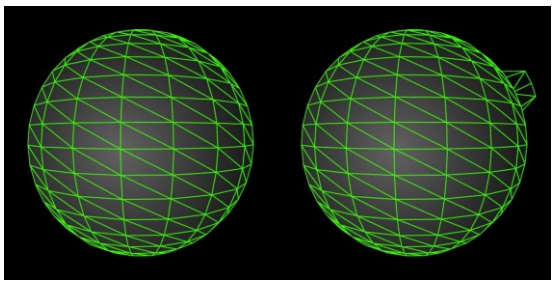


Figure 1.20. Back faces displacing into view.

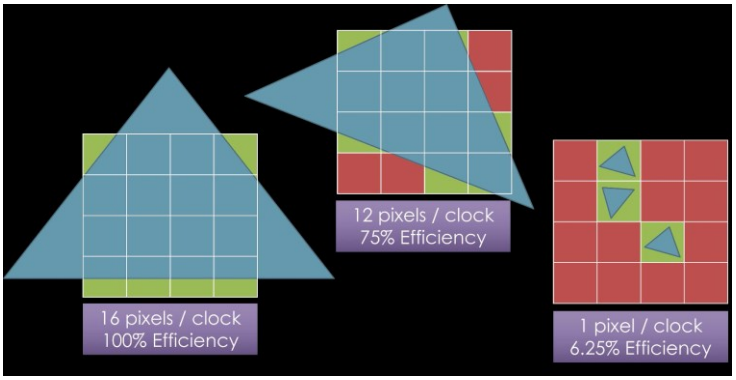


Figure 1.21. Small triangles are inefficient to rasterize.

1.7.2 Small Primitive Culling

Another effective filter is small primitive, or, culling triangles that do not generate pixel coverage. 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 (Figure 1.21).

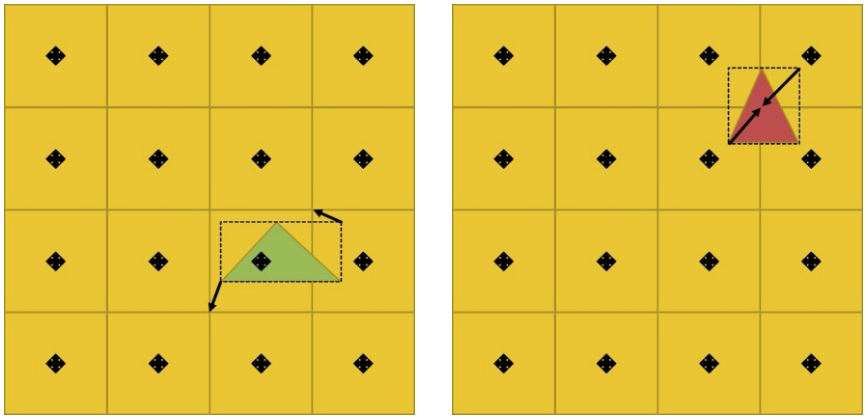
The left image in Figure 1.21 produces four quads, 16 pixels, at peak efficiency. The middle image produces four quads, but only 12 pixels are valid. It consumes 16 threads in the pixel shader though, due to helper lanes. Helper lanes still take time to pack and prepare, so they actually hurt your pixel rate. Efficiency in the middle image is lost due to partially filled quads, since the GPU shades in blocks of 2 × 2 pixel quads. The right image has become bound by hitting primitive setup limits.

Originally, a very exhaustive fixed-point hardware-precise small primitive filter was used, but later this was changed to the approximation you see below, for non-MSAA targets.

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

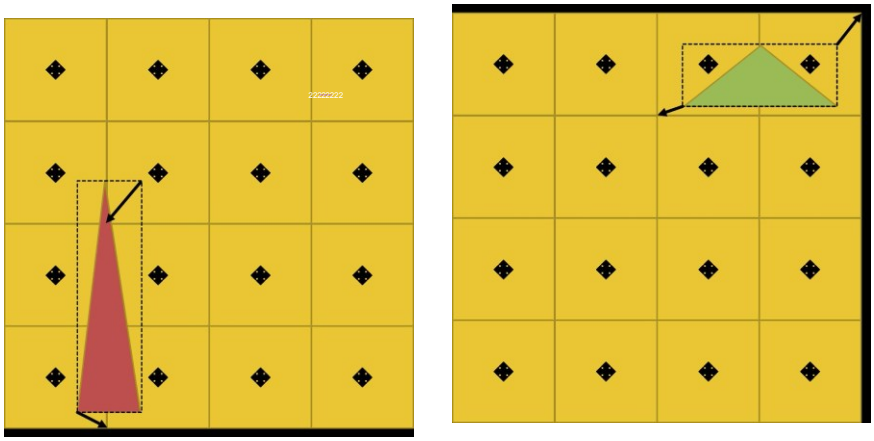
MSAA targets need to bias the test by enlarging based on sample count. If using custom-programmable sample points, a different solution will be needed. For MSAA, we need to essentially determine the maximum distance (in sub-pixels) between the pixel center and the outermost sub-pixel sample and use this to influence the test.

The general idea is to 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 Figure 1.22(a), the triangle is not culled because it encloses a pixel center. In a simple case (Figure 1.22(b)), the triangle is culled because the min and max snap to the same location.



(a) (b)

Figure 1.22. (a) Triangle is not culled—no pixel coverage; (b) triangle is culled—min and max snap to same location



(a) (b)

Figure 1.23. (a) Triangle is culled; min and max snap to same horizontal coordinate; (b)Triangle is not culled; conservative test misses case.

In a more complex case (Figure 1.23(a)), the triangle is also culled. The min and max snap to different vertical coordinates, but to the same horizontal coordinate. This test is conservative, so there is a case where triangles should be culled, but are not, as shown by Figure 1.23(b). 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.

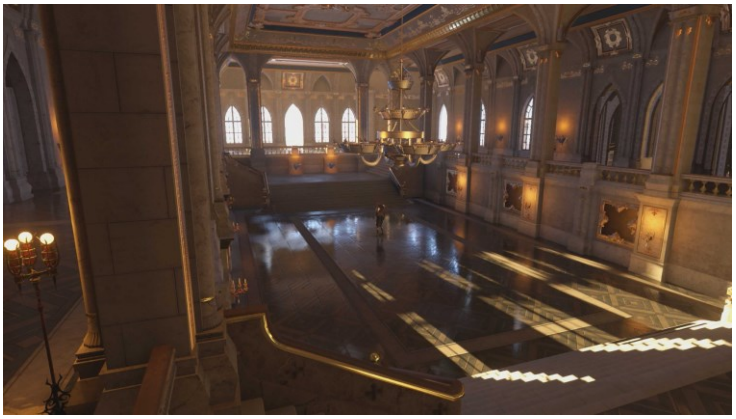


Figure 1.24. Small primitive test scene.

Figure 1.24 shows the small primitive test applied to the character 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 (Figure 1.25). 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.

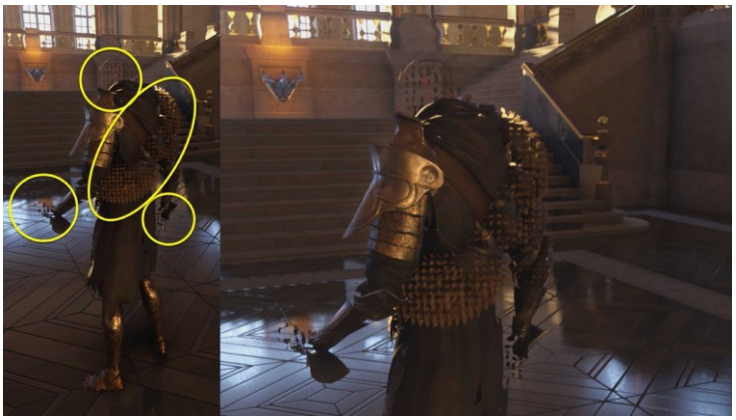


Figure 1.25. Results of small primitive test.

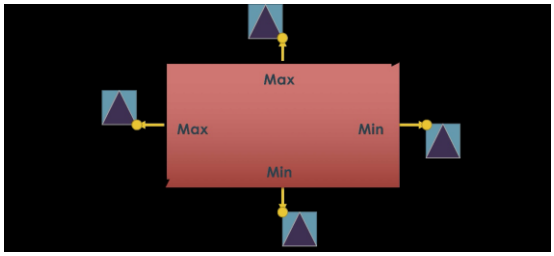


Figure 1.26. Fast frustum test.

1.7.3 Frustum Culling

The next per-triangle filter to cover is frustum culling. 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 four planes in four cycles (Figure 1.26), 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 games. 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.

Figure 1.27 is an example of the frustum-culling filter; this is the current view with just frustum culling enabled. We have a mesh which survives CPU frustum culling, but there are still quite a lot of triangles that could be removed.

Figure 1.27. Frustum culling test scene.

Figure 1.28. Results of frustum-culling test.

After locking the view, moving backwards shows us how many triangles were removed using this filter (Figure 1.28).

1.7.4 Depth Culling

Another available triangle-culling approach is to do manual depth-testing [Hill and Collin 2011]. 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 [Harris 2007] (Figure 1.29), storing out the conservative depth min or max value for each tile.

In the initial implementation, a full z pre-pass was run that produced a 16 × 16 depth-tile grid (Figure 1.30), which was then tested against a screen-space bounding box of a triangle or cluster. If the box was fully contained within a single tile, a fast depth test was performed to reject it. This approach, while fast, would only remove a fraction of triangles; any occluded triangles that

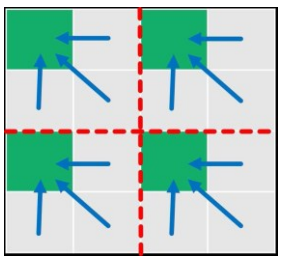


Figure 1.29. Depth reduction.

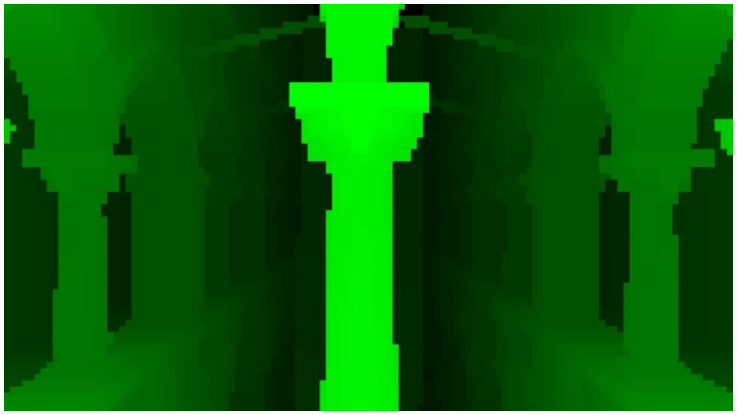


Figure 1.30. Generated depth tiles.

straddled a tile border would not be rejected. Modifying the filter to cull larger triangles spanning multiple tiles would be extremely expensive and not worth the cost.

Listing 1.9 shows a variant of parallel depth reduction which uses GCN lane swizzling to share data, bypassing LDS storage.

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

With a 16-bit ESRAM depth buffer already decompressed, this computation runs in approximately 41 μs on the Xbox One @ 1080p and is completely bandwidth bound. The result from this reduction is used for other parts of the rendering including compute light-tile culling.

```
1  float4 zQuad = g_linearZ.Gather(g_pointClamp (DTid.xy * 2 + 1) * g_rcpDim);
2
3  float minZ = min(zQuad.x, min3(zQuad.y, zQuad.z, zQuad.w));
4  float maxZ = max(zQuad.x, max3(zQuad.y, zQuad.z, zQuad.w));
5
6  // Use lane swizzling to share data, bypassing LDS
7
8  minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x2F | (0x01 << 10)));
9  minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x1F | (0x02 << 10)));
10 minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x2F | (0x08 << 10)));
11 minZ = min(minZ, __XB_LaneSwizzle(minZ, 0x1F | (0x10 << 10)));
12
13 maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x2F | (0x01 << 10)));
14 maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x1F | (0x02 << 10)));
15 maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x2F | (0x08 << 10)));
16 maxZ = max(maxZ, __XB_LaneSwizzle(maxZ, 0x1F | (0x10 << 10)));
17
```

```
// 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);
```

Listing 1.9. GCN parallel depth reduction.

Another approach to depth culling is using a hierarchical Z pyramid [Greene et al. 1993], 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 that, we populate the remaining mip levels in the pyramid through a custom downsample pass (Figure 1.31).

Figure 1.31. Depth pyramid example.

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.

```
int mip = min(ceil(log2(max(edge, 1.0))), levels 1);
```

This is the approach used for the depth-based culling, except it has also been accelerated with HTILE.

GCN has a depth acceleration meta-data called HTILE [Advanced Micro Devices 2011] which accelerates regular GPU depth operations. Every 8×8 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.

Figure 1.32. Visualized depth buffer of a scene.

Figure 1.33. Visualized HTILE of a scene.

There are some complexities with using HTILE, but manual decoding or encoding can be a big performance-win in a variety of scenarios. Currently, HTILE is only directly accessible to console developers (see Figures 1.32 and 1.33).

When computing the first downsampled mip level of the Hi-Z pyramid, you can leverage the fact that the shader already read the input depth values. So that shader can also perform full and or half-resolution linearization of the depth values; we can also write out half-resolution HTILE. This has the benefit that 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 cannot just use the already reduced 4 : 1 min and max values. We need to parallel reduce all pixels in an 8 × 8 tile to produce the correct min and max values for HTILE. You could do a parallel reduction in LDS, or you could take advantage of lane swizzling on GCN (shown in Listing 1.10).

```
// 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_outTileDim);
    uint htileValue = encodeCompressedDepth(minZ, maxZ);
    g_htileHalf.Store(htileOffset, htileValue);
}
```

Listing 1.10. Lane swizzle HTILE generation.

Because each HTILE entry represents an 8 × 8 pixel block, we can use a wave-wide min and max operation across 64 depth values in a tile using lane swizzle (see Listing 1.11).

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

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

Listing 1.11. Lane swizzle wave-wide min and max.

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 8 × 8 pixel tile. Near depth is used for trivial accept, whereas far depth 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 z-mask, which we set to zero for clear.

The Hi-Z pyramid does not need stencil, so the encoding routine in Listing 1.12 is for the non-Hi-stencil format.

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

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

Listing 1.12. HTILE encoding routine.

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*** (Figure 1.34) [Collin 2011], which is generated on the CPU for the upcoming GPU frame; the results of this operation can be used to load the 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 reprojecting your previous depth buffer and testing with that.

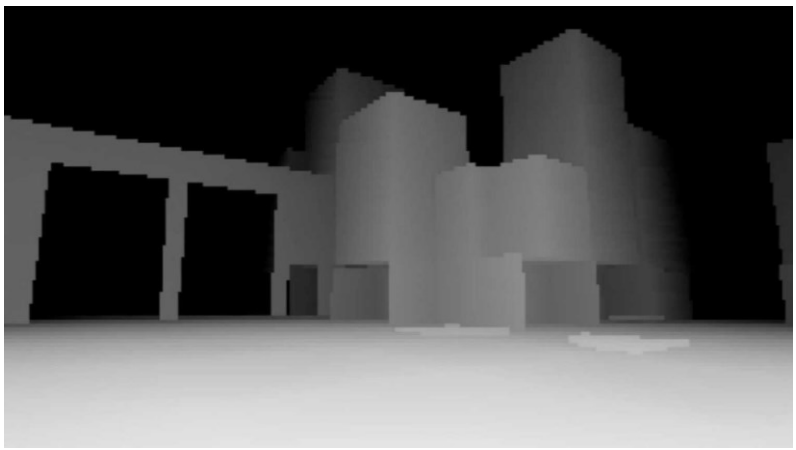


Figure 1.34. Software occlusion raster.

Figure 1.35. Depth pyramid test scene.

Figure 1.35 shows the character 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. Figure 1.36 shows the visualized occluder geometry used to produce the software occlusion buffer for this scene.

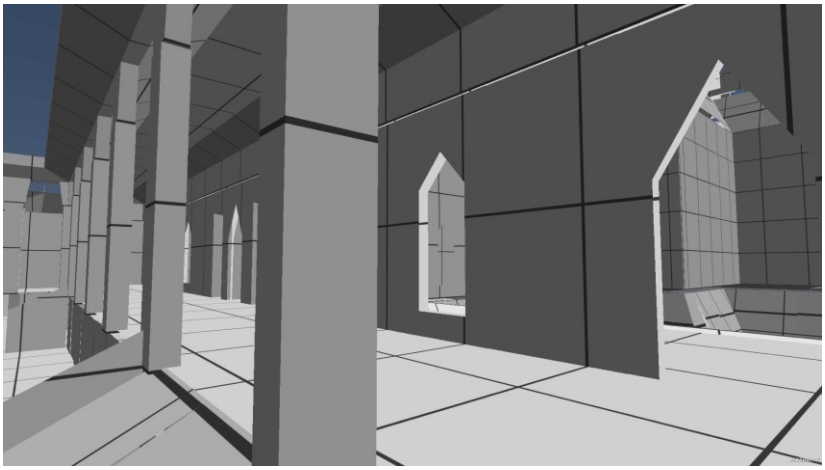


Figure 1.36. Visualized occluder geometry.

Locking the view, and moving to the other side of the pillar, we see the surviving triangles for Solas and what was rejected by Hi-Z culling (Figure 1.37).

Figure 1.37. Results of depth pyramid test.

1.8 Batch Scheduling

It is important to discuss how the batching is structured to make the overlap of culling and rendering efficient. 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 × 128 k triangles, where N is large enough to get decent overlap between culling and render; 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 (Figure 1.38).

Figure 1.38. Fixed memory budget of batches.

Assuming 16-bit unsigned short, 384 k triangle indices is 786 Kb of memory. Four buffers is approximately 3 MB, which allows for up to a half-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.

In the example shown in Figure 1.39, we have four buffers, which gives us a total surviving triangle capacity of 512 k. We have to calculate output require-

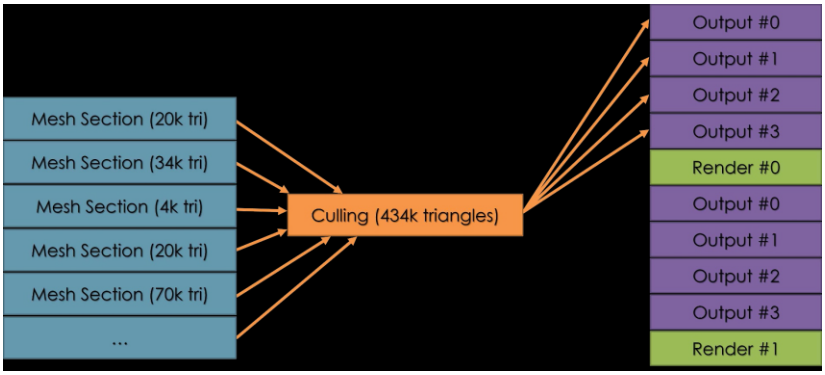


Figure 1.39. Batching within budget.

ments before culling, in case all triangles survive. An initial thought was to do a rough heuristic against 50% back-face culled, but certain projections could cause problems. You can see that culling is processing 434 k triangles, which fits well within our 512 k limit. Render #0 will occur, and then the next pass can reuse the output buffers. This leads into a more complex case.

In the example shown in Figure 1.40, the culling is processing more triangles than we have capacity for. When we determine that the buffers are exhausted, 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.

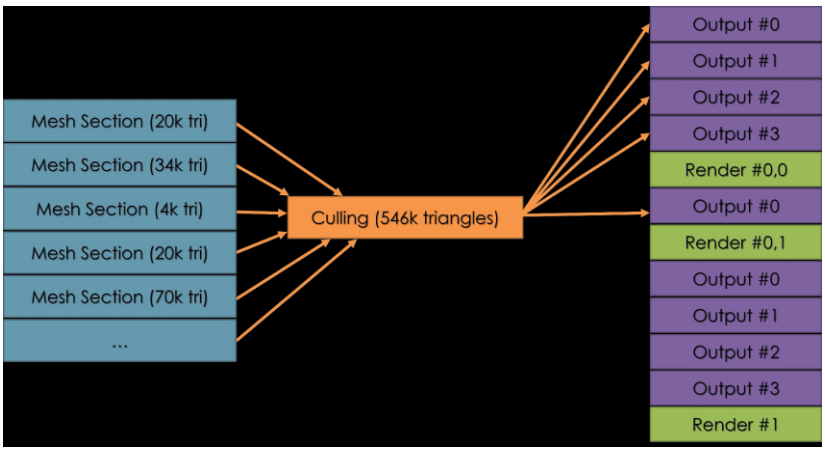


Figure 1.40. Batching over budget.

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 (Figure 1.41).

Figure 1.41. Culling on graphics pipe.

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 (Figure 1.42).

Figure 1.42. Culling on compute pipe.

You can use inexpensive wait on label operations to ensure that dispatch and render are pipelined correctly. On a PC, try to aim for fewer batches at a larger size, due to the inability of DirectX OR 12 to issue efficient mid-command buffer fences.

In general, you want to schedule your asynchronous compute to happen at the same time as low-intensity rendering work, like a depth pre-pass or shadows. Use fences to bracket the dispatches so they do not start early or late on the GPU, and make sure to flush the asynchronous compute command buffer so it does not 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 asynchronous compute to do the work during the last stages of post-processing on the previous frame.

1.9 De-interleaved Vertex Buffers

Another architectural note is that the vertex buffers have been de-interleaved (Figure 1.43). This can be a substantial win on GCN architectures, and it also makes the task of compute mesh-processing much easier.

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 texture coordinates, vertex colors, tangent space basis, etc. means that there is an almost never

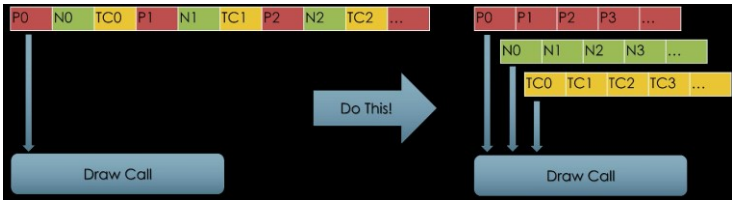


Figure 1.43. De-interleaved vertex buffers.

changing stride. The only time you would need to break batching would be 16-bit vs 32-bit precision.

Consoles and DirectX OR 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 is really no excuse. The goal is 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 the cache line is thrown 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 texture coordinates 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 texture coordinate, or same position but different normal).

1.10 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.

Figure 1.44. *Dragon Age™ Inquisition* displacement mapping.

Tessellation is not for everyone, as there are games that cannot afford the overhead, but there are some strategies that can be used to improve performance when hardware tessellation is used, such as *Dragon Age™ Inquisition* (Figure 1.44), and *Star Wars™ Battlefront*.

When using tessellation, the goal is to produce vertex waves at peak rate per shader engine. 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 (Figure 1.45), 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 the max tessellation factor.

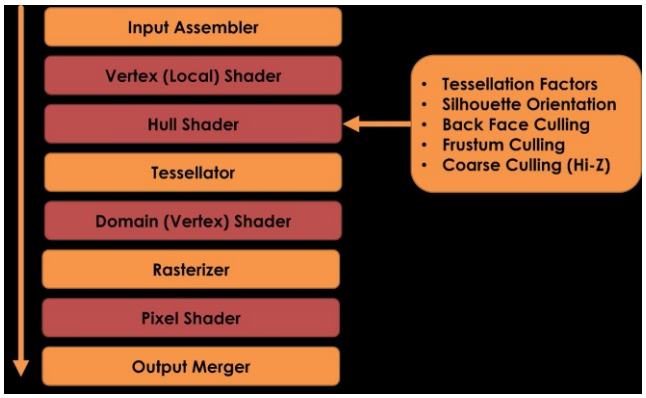


Figure 1.45. Hardware-tessellation pipeline.

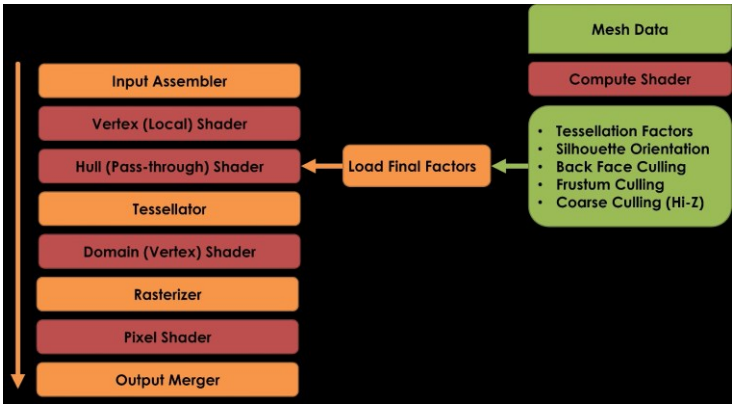


Figure 1.46. Offloaded tessellation factors calculation.

There are 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 that the GPU can only fit so much control-point data into LDS. The other issue is that the shader compiler implements the patch constant function in the case of three vertex triangle patches by turning off two out of the three active threads and only running code on the remaining thread. With 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.

A first step of 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 (Figure 1.46). The results are then stored into a factors buffer that the hull shader can index with SV_PrimitiveID. This optimization makes the hull shader stay active for the bare minimum amount of time, which is nice, but this method 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 doing initial profiling of tessellation on GCN, some overhead was expected, but it was shocking 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 three times more vertices! Additionally, these factor-1 draws suffer from the same parallelism constraints that were just mentioned.

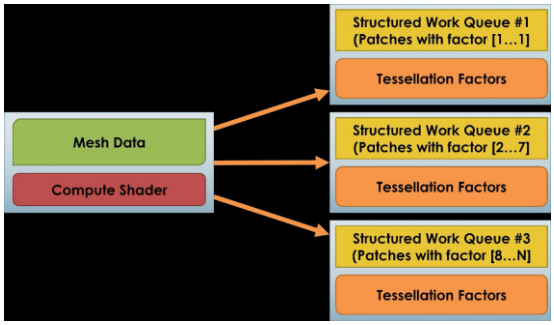


Figure 1.47. Compute tessellation work queues.

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 (Figure 1.47). Culled patches with a factor of 0 are not processed further and do not get inserted to any work queue.

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.

The general goal here is to produce small patches, so that we can parallelize more of the mesh across more CUs. All of the vertices heading into the domain stage need to be processed on the same CU, since tessellation-patch constants are 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 four smaller patches and push them into the tessellation work queue with $\frac{1}{4}$ of the original tessellation factor (Figure 1.48).

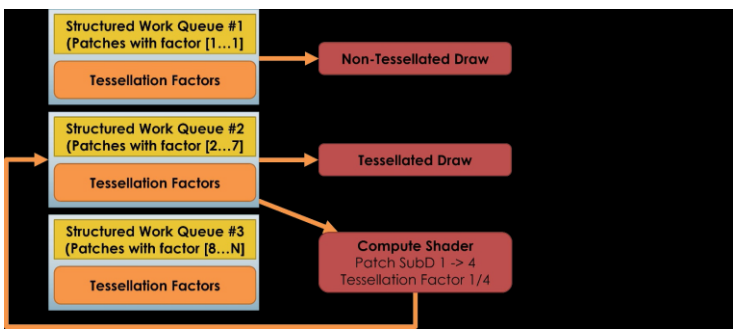


Figure 1.48. Compute tessellation pipeline.

One thing you need to handle is accounting for T-junctions between varying patch levels. Using an algorithm like PN-AEN [McDonald and Kilgard 2010] will give you triangle patches which include edge-adjacency information, which is helpful for solving this issue.

1.11 Performance

For the performance metrics, the test scene used is shown in Figure 1.49. There are quite a number of render passes, and with 171 unique PSOs, but the results shown are for a single g-buffer pass of 450 k triangles, rendered at 1080p on both platforms (without cluster culling or compute tessellation). For the Xbox One, the depth buffer and a few of the g-buffer color targets are in ESRAM, with everything else in DRAM.

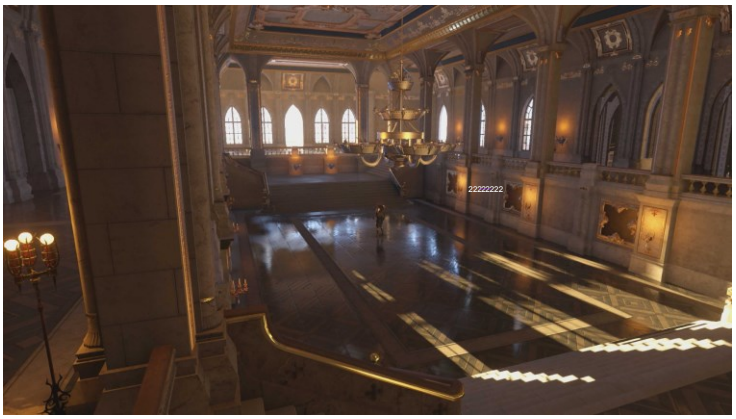


Figure 1.49. Performance profiling test scene.

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 four cycles for four planes, so it doesn't hurt to leave it.

After orientation culling, the depth filter is the second most effective technique, but it 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 will notice that for this scene, we have managed to cull enough that we are only left with 22% of the original triangle count. Now imagine feeding the

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

Table 1.5. Ratio of culled vs rendered triangles.

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

Table 1.6. Primitive culling amounts for the various filters.

Platform	Base	Cull	Draw	Total
XB1 (DRAM)	5.47ms	0.24ms	4.54ms	4.78ms
PS4 Std (GDDR5)	4.56ms	0.13ms	3.76ms	3.89ms
Fury X (HBM)	1.79ms	0.06ms	1.40ms	1.46ms

Table 1.7. Synchronous culling performance, no tessellation.

Platform	Base	Cull	Draw	Total
XB1 (DRAM)	5.47ms	0.26ms	4.56ms	4.56ms
PS4 Std (GDDR5)	4.56ms	0.15ms	3.80ms	3.80ms
Fury X (HBM)	1.79ms	0.06ms	1.40ms	1.40ms

Table 1.8. Asynchronous culling performance, no tessellation.

resultant culled index buffer into related passes, where we do not need to worry about the cost of culling.

Tables 1.5-1.8 are the performance figures for this scene, on both consoles and on an AMD Fury X on PC. Even with mesh data in DRAM, the Xbox One culling is slowest, yet needs barely any time at all to process one-half million triangles in a g-buffer pass. Synchronously, we are saving 15-30% of our rendering cost, and, asynchronously, we are saving a bit more. The draw and cull times get a little bit longer when running asynchronously, but you will 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, likely by an additional 10–15%, but it is important to show the effectiveness of per-triangle culling even in a color pass with varying PSO changes.

Tables 1.9 and 1.10 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 seven. Here you will 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

Platform	Base	Cull	Draw	Total
XB1 (DRAM)	19.3ms	0.24ms	11.1ms	11.3ms
PS4 Std (GDDR5)	12.8ms	0.13ms	8.08ms	8.21ms
Fury X (HBM)	3.35ms	0.06ms	1.46ms	1.52ms

Table 1.9. Synchronous culling performance, with tessellation.

Platform	Base	Cull	Draw	Total
XB1 (DRAM)	19.3ms	0.26ms	11.2ms	11.2ms
PS4 Std (GDDR5)	12.8ms	0.15ms	8.10ms	8.10ms
Fury X (HBM)	3.35ms	0.06ms	1.46ms	1.46ms

Table 1.10. Asynchronous culling performance, with tessellation.

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

1.12 Conclusion

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 is not much of an issue for that under DirectX® 12 or Vulkan™, except for the unnecessary memory pressure of each instance reloading the same vertex data and less wavefront packing. However, this system is getting incredible L2$ hits for the instanced data, when running synchronously. With uninstanced data, profiling shows approximately 20 bytes of bandwidth usage per triangle, but with instancing due to the batch chunk size and near perfect L2$ residency, profiling shows approximately 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.

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. It will also be important to further investigate careful tuning of wavefront limits and also CU masking.

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.

In summary, small and inefficient draws are a problem. DirectX® 12 and Vulkan™ provide an API to submit tons of draws at great performance from the CPU, but the GPU can still choke on these. Therefore, it is important to

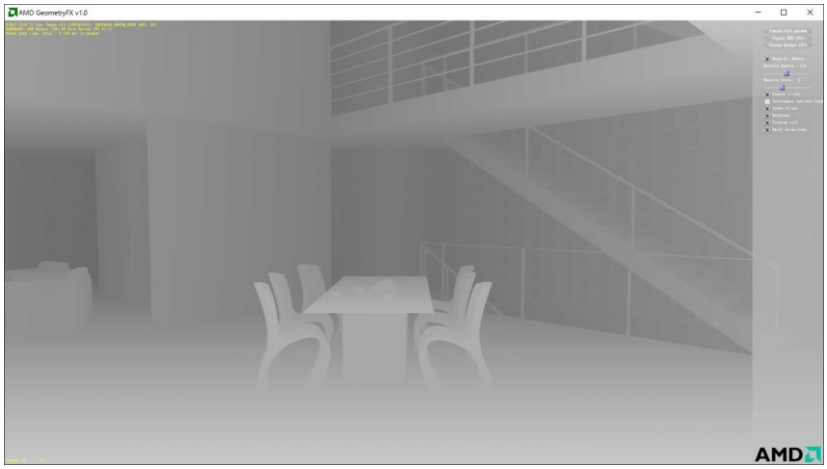


Figure 1.50. AMD GPUOpen GeometryFX.

realize that compute and rasterization are friends; treat your draws as data and have both compute and graphics help each other out. You typically only want to use idle GPU resources to remove fixed-function bottlenecks; otherwise you may impact performance instead. Asynchronous compute is extremely powerful—be sure to schedule compute wavefronts alongside the rest of your frame, but do not forget that you can overlap compute and graphics work on the same pipe; many engineers do not realize this.

If you are interested in implementing something similar to this technology, you should look at AMD GeometryFX (Figure 1.50), released as part of GPUOpen. While not the exact same technology as discussed in this chapter, it serves as a great example on how to implement such compute-based triangle culling.

Acknowledgements

I would like to thank Johan Andersson, Christina Coffin, Mark Cerny, and the Frostbite Rendering team for their support of this research. I also want to thank the numerous people that offered guidance, ideas, improvements, support, and friendly conversation over a beer. Your help was very much appreciated! Thanks to Chris Brennan, 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 "IHatePolygons" 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 (@post-

goodism), Colin Barré-Brisebois (@ZigguratVertigo), 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).

Bibliography

Advanced Micro Devices, 2011. Radeon evergreen / northern islands acceleration. Evergreen Programming Guide. URL: http://developer.amd.com/.

Advanced Micro Devices, 2012. Southern islands series instruction set architecture. Reference Guide. URL: http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf.

Barequet, G., and Elber, G. 2005. Optimal bounding cones of vectors in three dimensions. ***Inf. Process. Lett. 93***, 2, 83 – 89.

Blinn, J. F., and Newell, M. E. 1978. Clipping using homogeneous coordinates. In ***Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques***, ACM, New York, SIGGRAPH '78, 245 – 251.

Brainerd, W. 2014. Tessellation in ***Call of Duty: Ghosts***. In ***Advances in Real-Time Rendering in Games, SIGGRAPH Course***, ACM, New York.

Collin, D. 2011. Culling the battlefield: Data oriented design in practice. In ***Game Developer's Conference***. URL: http://www.gdcvault.com/play/1014492/Culling-the-Battlefield-Data-Oriented.

Greene, N., Kass, M., and Miller, G. 1993. Hierarchical z-buffer visibility. In ***Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques***, ACM, New York, SIGGRAPH '93, 231 – 238.

Haar, U., and Aaltone, S. 2015. Gpu-driven rendering pipelines. In ***Advances in Real-Time Rendering in Games, SIGGRAPH Course***, ACM, New York.

Harris, M., 2007. Optimizing parallel reduction in cuda. http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

Hill, S., and Collin, D. 2011. Practical, dynamic visibility for games. In ***GPU Pro 2***, W. Engel, Ed. A K Peters, 329 – 347.

Lauritzen, A., Salvi, M., and Lefohn, A. 2010. Sample distribution shadow maps. In ***Advances in Real-Time Rendering***, ACM, New York, SIGGRAPH 2010 course. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/sampleDistributionShadowMaps_SIGGRAPH2010_notes.pdf.

Mah, L., 2013. The AMD GCN architecture: A crash course. AMD Developer Summit. URL: https://www.slideshare.net/DevCentralAMD/gs4106-the-amd-gcn-architecture-a-crash-course-by-layla-mah.

Mara, M., and McGuire, M. 2013. 2d polyhedral bounds of a clipped, perspective-projected 3d sphere. ***Journal of Computer Graphics Techniques (JCGT) 2***, 2, 70 – 83.

McDonald, J., and Kilgard, M., 2010. Crack-free point-normal triangles using adjacent edge normals. Nvidia Report. URL: http://developer.download.nvidia.com/whitepapers/2010/PN-AEN-Triangles-Whitepaper.pdf.

Olano, M., and Greer, T. 1997. Triangle scan conversion using 2d homogeneous coordinates. In ***Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware***, ACM, New York, HWWS' 97, 89 - 95.