

优化Pipeline计算

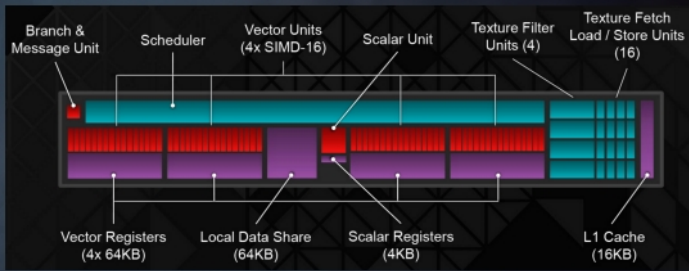
如何通过减少渲染那么多的三角形提高三角形的渲染速度

一些缩写所代表的意思，后文中会经常用到

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



AMD
GCN
ARCHITECTURE

分析目标平台的三角形发出速率

每个shader engines 在一个时钟周期内发出一个三角形，PC端较新的AMD显卡有四个shader engines，可以发出四个三角形每个时钟周期。

将计算单元(compute units)的数量 * 64 * ALU数量 * 2 即可得到一个时钟周期内ALU操作执行的数量。

使用ALU操作执行的数量 / engines的数量，可以得到一个时钟周期内，每个三角形中可以执行的ALU操作的数量

将每个三角形的ALU操作数 / 每个时钟周期ALU 的操作数，可以得到一个最终的指令上限。

乐观点假设Rasterizer每个时钟周期能处理两个三角形，实际上xb1是0.9个。因为从VGT->PA使用FIFO队列，所以在4096个时钟周期内填满FIFO队列，才不会使Rasterizer等待。

作者在得到每个时钟周期内接近 两个Prims的效果时有以下特征：

- VS 没有读任何东西
- VS仅输出SV_Position
- 每个VS仅输出0.0，都被剔除了

大部分的引擎都会在CPU端做剔除，再在GPU上refine。但是因为PCIe的传输速率低，所以我们会做一些GPU上的剔除，本篇主要讲cluster/triangleculling。

CS处理mesh带来的好处。最主要的思想就是把drawcall看作data，这些GPU生成的数据可以pre-built, cached, reused。

开始

Culling Overview

- Scenen包括
 - 网格集合
 - 特定的视角
 - camera light 等

将场景中的网格合并成一个Batch，即网格拥有相同的vertex index 步长，共享shader。

当然，batch划分为mesh section，即一次绘制需要的内容：

- vertex buffer
- index buffer
- primitive count
- 等

在swap中处理最佳的量：

- AMD中一个**wavefront(warp)**有64个线程
- 一个culling Thread处理一个三角形
- 一个work item处理256个三角形

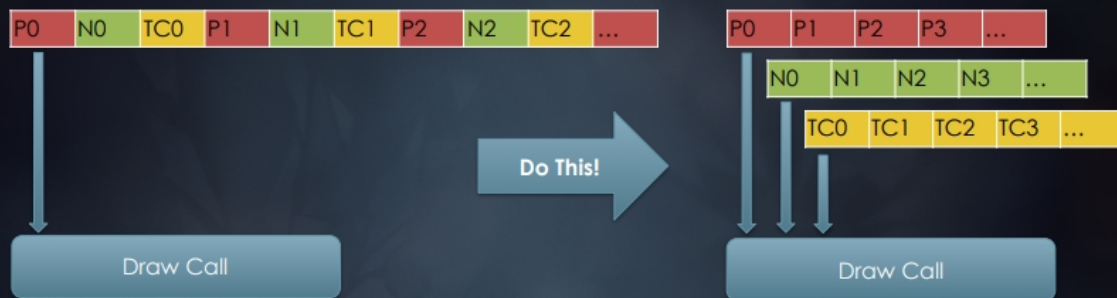


Mapping Mesh ID to MultiDraw ID

- 使用Indirect Draws时，着色器不再知道有关Mesh Section或者Instance的信息。着色器绘制的数据来源于各种各样的Constants Buffer。
- OpenGL 可以使用 gl_DrawId如SV_InstanceID

使用各种缓冲区来传递绘制需要的信息，如verties, index, instance（位移，颜色。光照贴图等等）。

De-Interleaved Vertex Buffers



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

将vertex缓冲区去掉交错使之最高效的适应GVN架构。并且交错结构的Vertex缓冲也使CS 处理mesh 更加的简单。

去掉交错顶点缓冲结构好处：

- 将GPU计算时的状态变化降到最低
- 恒定的vertex position步长
- 更清晰的分离易丢失和非易丢失数据
- 内存使用更少
- 更理想的GPU渲染
- 尽快的清理管线

在CS中做剔除时，我们只需要操作位置信息即可，分离的vertex buffer意味着，我们可以在操作position时避开颜色，uv，法线等信息。

同时使用恒定的stride（步长），和一些类指针操作来确定每次绘制的起点和索引位置，我们可以避免在渲染过程中绑定不同的缓冲区。

同时这样结构的顶点缓冲区对Cache也很友好，可以re-use更多的顶点。同时CPU端的数据也由ASO变为SOA后更容易处理 SSE/AVX，同样的优势也适用于GPU。

SOA和AOS分别指阵列结构和结构列阵。

Cluster Culling

在更精细的针对三角形的剔除前，一个重要的部分是针对**Triangle Clusters**的粗剔除。

在球坐标系中，使用贪心算法将Mesh分为256个triangle一组的clusters，对于每个cluster预烘焙一个bounding cone。做法是将每个三角形的法线投影到单位球上，收集256个法线，并计算一个最小的闭包。4个8bit的snormal的精度就可以存储这个圆锥了。只需要使用圆锥的法线和视线的点乘与圆锥的张角的Sin值作比较就可以进行剔除了。为了避免false reject，张角可以扩大一些。

```
1  if(dot(cone.Normal,-view) < sin(cone.angle))
2  {
3      cull;
4  }
```

还可以对normal进行GBuffer 常用的encode。关于给Cluster做包围盒的操作，可以看[这个](#)。

Cluster 的大小在256是最优化的大小在PC端。可以最大化顶点重用和更少的原子操作。

关于Cluster Culling 的 Coares reject算法，[这篇文章详细介绍过了](#)。Occlusion用的是bounding sphere vs bounding box ([HiZ](#))，关于HiZ剔除还有[这个](#)。注意在透视投影中sphere会变成ellipsoid

Draw Compaction

更加紧致的IndexBuffer

从提交中删除0大小的绘制是非常重要的。



灰色的部分是捕捉的提交空绘制的GPU的部分，可以看到在133us时，效率开始下降，因为进行了一连串的空绘制。同时，在151us时，大约有10us的空闲时间。然而，消耗远比空10us更多，因为GPU不会立刻回复100%的效率。因为分配给warp任务-给GPU计算需要时间。大量被剔除的绘制画面容易使命令处理器被拖累，在60hz帧中，大约有1.5ms的延迟。

虽然使用GPU culling 对效率的提升超过了这个成本，但是为了获得最大收益，将Draw 中的0 size剔除是非常重要的。即使没有任何Primitives，获取间接参数也不是免费的。

综上所述 **compact index**是多么重要。

HOW TO Ccompact Index Buffer

CPU端，将会按照最坏的情况提交渲染，因此，即使一个三角新占据的面积是0，GPU照样会执行计算。当我们使用**ExecuteIndirect**命令时，将执行剔除，由GPU端来控制绘制计数和状态变化。

ExecuteIndirectDraw这个API有一个可配置的计数和offset缓冲区。GPU将使用它来使渲染的命令得到精简。

Compaction

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

$Count = \text{Min}(\text{MaxCommandCount}, pCountBuffer)$

ID3D12GraphicsCommandListExecuteIndirect API讲解：

如果pCountBuffer为null，则该API在绘制的时候，将使用MaxCommandCount参数来作为绘制调用参数；若其不为null，则取两个中的最小值来作为参数。

一个跨平台达到Compaction的做法是利用共享内存做parallel reduction算法。

在GPGPU中，有一些常用的并行算法。如Parallel Reduction，Parallel Scan等。具体参考[这个](#)和[这个](#)。

算法如下：

```
1 groupshared uint localValidDraws;
2 [numthreads(256,1,1)]
3 void main(uint3 globalID : SV_DispatchThreadID, uint3 threadID :
  SV_GroupThreadID)
4 {
5     //如果是当前Group内的第一个线程，则localValidDraws = 0;
6     if(threadID.x == 0)
7     {
8         localValidDraws = 0;
9     }
10    GroupMemoryBarrierWithGroupSync();
11    MultiDrawIndirectArgs drawArgs;
12    //globalID.x 是当前线程在整个发射的线程中的位置。
13    const uint drawArgId = globalID.x;
14    //如果当前线程的ID小于总的绘制长度，则获取当前线程负责处理的instance
15    if(drawArgId < batchData[g_batchIndex].drawCount)
16    {
17        loadIndirectDrawArgs(drawArgId, drawArgs);
18    }
19    uint localSlot;
20    //这里可以理解成，当前的这个Instance通过了剔除
21    if(drawArgs.indexCount > 0)
22        //让组内线程共享的localValidDraws+1
23        InterlockedAdd(localValidDraws, 1, localSlot);
24    //这里强制线程同步，localValidDraws里保存的既是有多少个Instance是可以被渲染的。
25    GroupMemoryBarrierWithGroupSync();
```



```

26     uint globalSlot;
27     //如果当前线程是线程组内第一个线程,drawCountCompacted是buffer的offset, 让他 +=
drawCountCompacted
28     //即可得出有多少个可用的Instance
29     if(threadId.x == 0)
30
31         InterlockedAdd(batchData[batchIndex].drawCountCompacted,localValidDraws,globalSlot);
32         GroupMemoryBarrierWithGroupSync();
33         //将可用的放到resultbuffer里。
34         if(drawArgId < drawArgCount && thisLaneActive)
35             storeIndirectDrawArgs(globalSlot+localSlot,drawArgs);
36     }

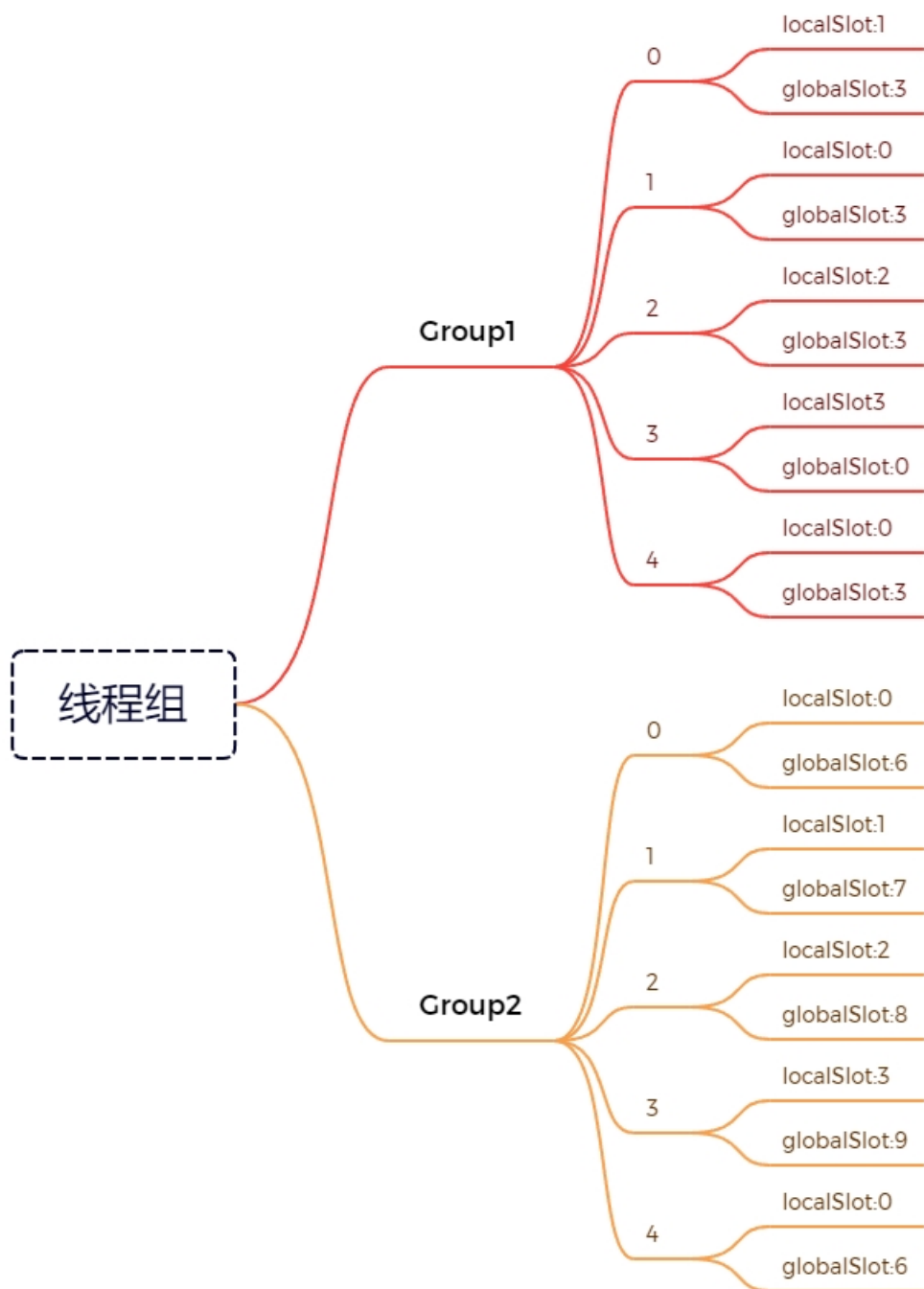
```

使用上述算法举个例子：

1	0	1	1	0	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---

这是一个有十个元素的IndexBuffer，1代表其可以被渲染，0代表其不可以被渲染，我们需要用上面的算法将可以被渲染的放到一起，减少GPU处理不可以渲染的数据的时间。

准备两个线程组，每个线程组五个线程。只关注一下每个线程在三次同步后的localSlot，和globalSlot值。

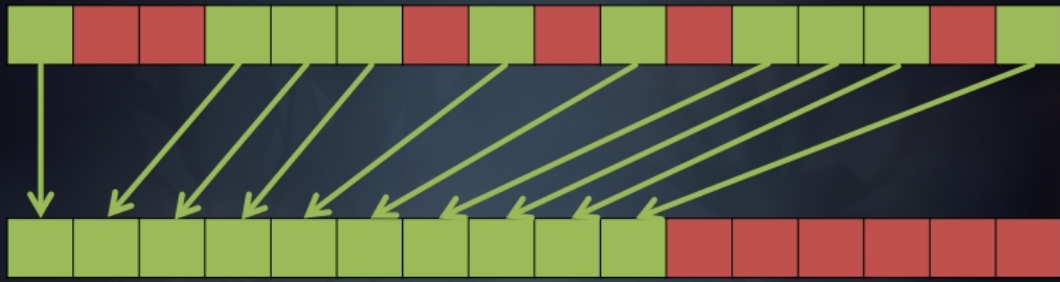


可以得到最后的结果：

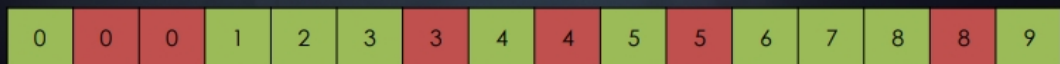
0	0	0	0	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---

将Offset值设置为4，即可得到一个连续的CountBuffer。

Compaction



► Parallel prefix sum to the rescue!



回顾一下刚才的算法，我们需要将结果写回一个连续的缓冲区内，做到这点仅靠ThreadId，是做不到的，所以我们做了两次同步，第一次是线程组内下标同步，第二次是全局下标同步，怎么做可以避免这么多次同步呢？

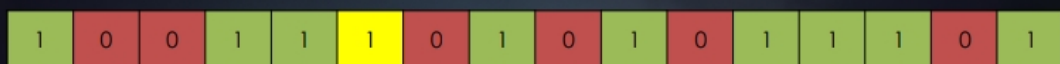
前缀和算法

实现这个算法，我们需要用到一个GPU内置变量，**Ballot**，AMD和英伟达的还不一样，AMD是64位掩码，英伟达是32位。这个的介绍可以看[介绍一些GPU内部变量](#)

我们可以利用这个变量，将被剔除的线程的掩码为设置为0，保留的设置为1。

Compaction

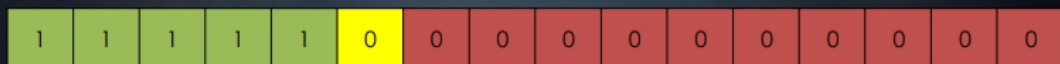
__XB_Ballot64(indexCount > 0)



&

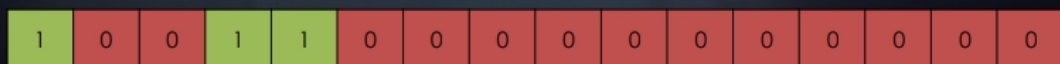
Thread 5

Thread 5 Execution Mask



=

Population Count "popcnt" = 3



第一行中可以看到，在线程5之前有三个有效的线程。

直接看代码吧：

```
1 groupshaded uint localValidDraws;
2 [numthreads[256,1,1]]
3 void main(uint3 globalID : SV_DispatchThreadID,unit3 threadID :
  SV_GroupThreadID)
```



```

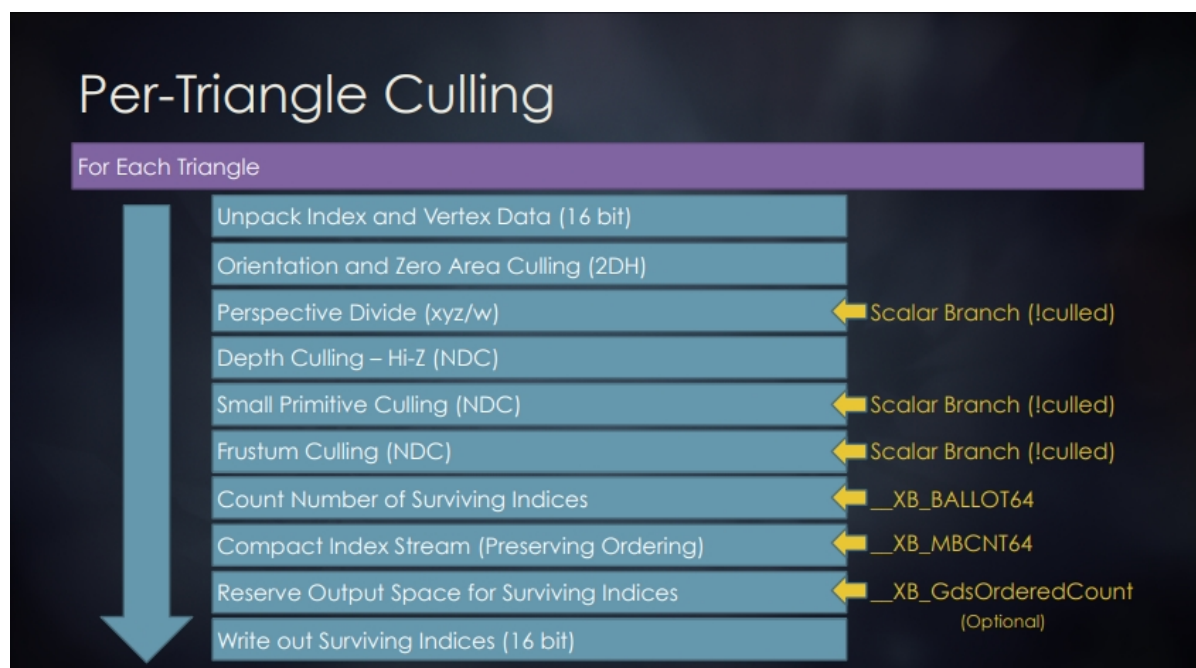
4  {
5      //当前线程在线程组内的ID
6      const uint laneID = threadId.x;
7      const uint drawArgId = globalId.x;
8      const uint drawArgCount = batchData[g_batchIndex].drawCount;
9      MultiDrawIndirectArgs drawArgs;
10     if(drawArgId<drawArgCount)
11         loadIndirectDrawArgs(drawArgId,drawArgs);
12     const bool thisLaneActive = (drawArgs.indexCount > 0);
13     uint2 clusterValidBallot = __XB_Ballot64(clusterValidBallot);
14     uint outputArgCount = __XB_SBCNT1_U64(clusterValidBallot);
15     uint localSlot = __XB_MBCNT64(clusterValidBallot);
16     uint globalSlot;
17     if(laneId == 0)
18     {
19
20         InterlockedAdd(batchData[batchIndex].drawCountCompacted,outputArgCount,globalSlot);
21     }
22     globalSlot = __XB_ReadLane(globalSlot,0);
23     if(drawArgId < drawArgCount && thisLaneActive)
24         storeIndirectDrawArgs(globalSlot + localSlot,drawArgs)
25 }

```

Triangle Culling

对Clusters做过粗糙剔除后，便要对三角形做更精细的剔除了。

一个线程处理一个三角形，流程与之前的很相似：



每个thread处理一个triangle，通过的culling的会使用刚才的技巧来获得当前triangle的compact index。对于wavefront之间如果想做半透排序等，需要使用ds_ordered_count来保证wavefront之间的输出顺序。

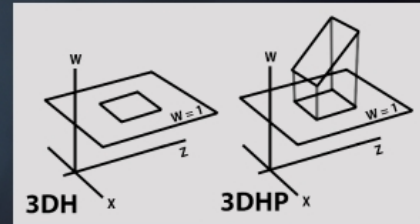
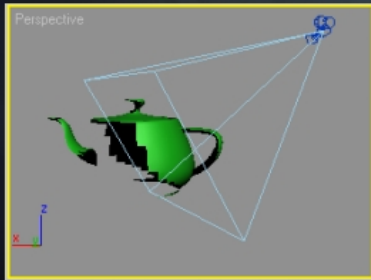
优化If分支太极限了。

Orientation Culling

包含背面剔除和过小面积的三角形剔除

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;
```



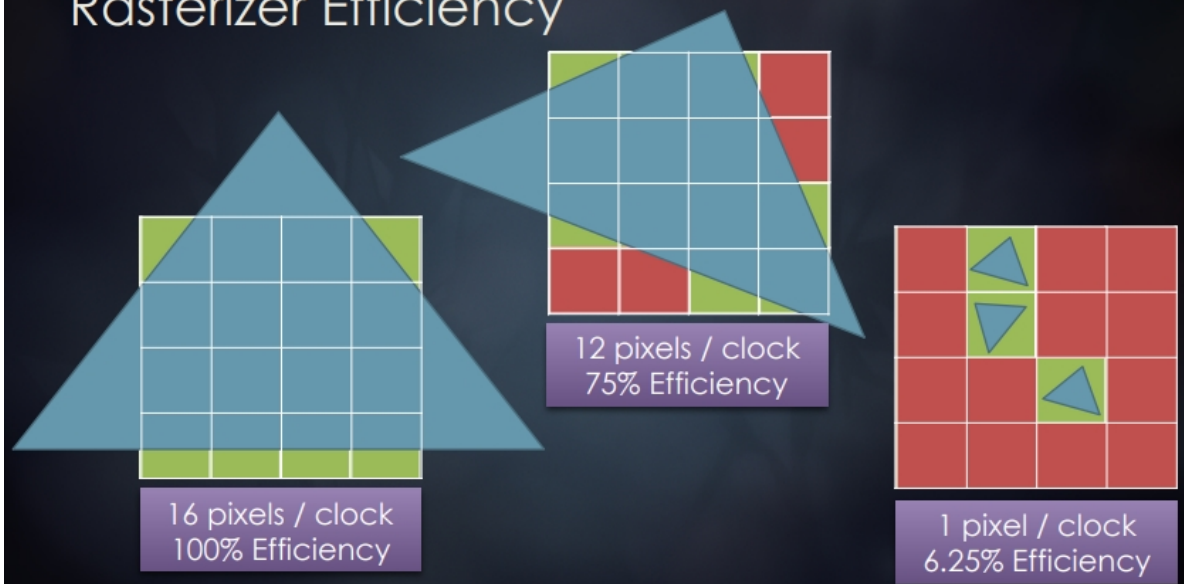
Tessellation patch back face cull略，没接触过。

Small Primitive Culling

对于不产生像素的三角形执行剔除。

每个时钟周期内，GPU可以处理一个三角形，产生16个像素。正因为如此，小的三角形会严重影响效率。可以看到最右边的效率最低

Rasterizer Efficiency



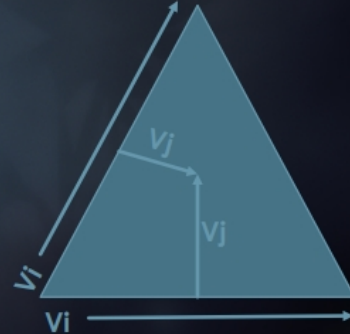
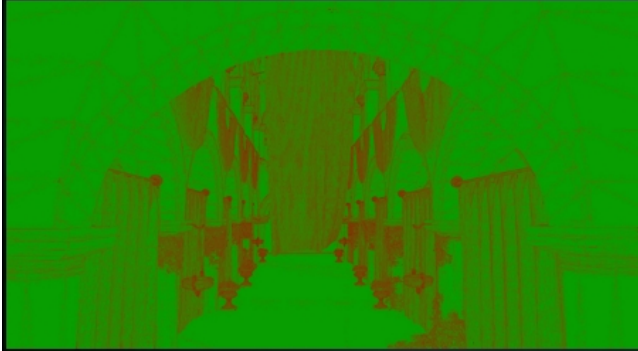
```
float3 main() : SV_Target0
{
    bool inside = false;

    float2 barycentric = fbGetBarycentricLinearCenter(); // __XB_GetBarycentricCoords_Linear_Center

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

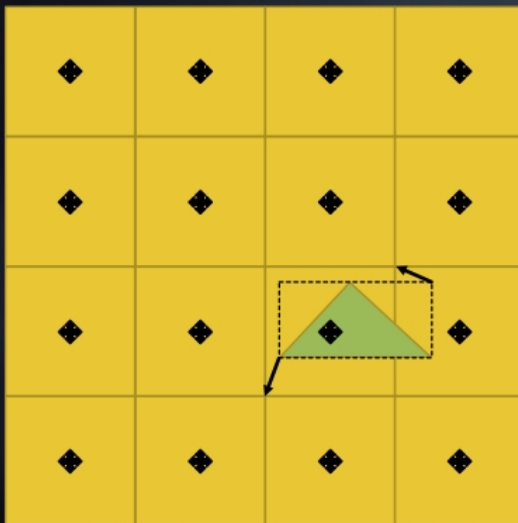
    uint2 insideBallot = fbBallot(inside); // __XB_Ballot64
    uint insideCount = countbits(insideBallot.x) + countbits(insideBallot.y);
    float insidePercent = insideCount * (1.0 / 64.0);
    return float3(1 - insidePercent, insidePercent, 0);
}
```

Rasterizer Efficiency



这个像素着色器可以可视化raster的效率，对于每个triangle会对应一个warp来处理其raster后的像素，那么每个warp中真正有效的pixel占thread的比值就可以看出是否有太多琐碎的triangle，另外也可以测试LOD的设置是否合理。

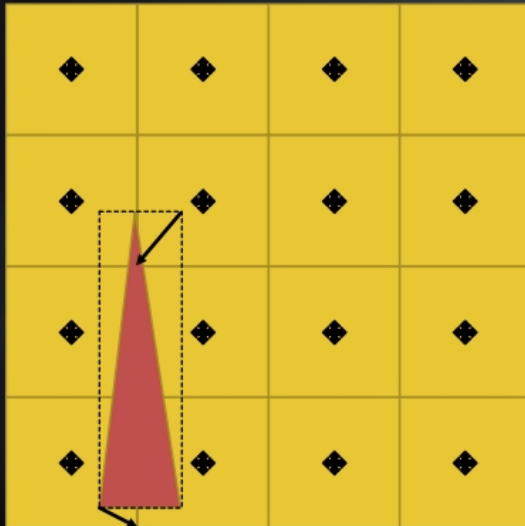
Small Primitive Culling (NDC)



► This triangle is **not culled** because it encloses a pixel center

$\text{any}(\text{round}(\text{min}) == \text{round}(\text{max}))$

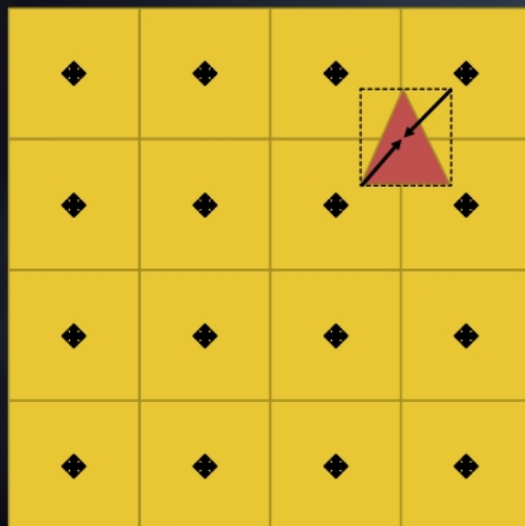
Small Primitive Culling (NDC)



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

$\text{any}(\text{round}(\text{min}) == \text{round}(\text{max}))$

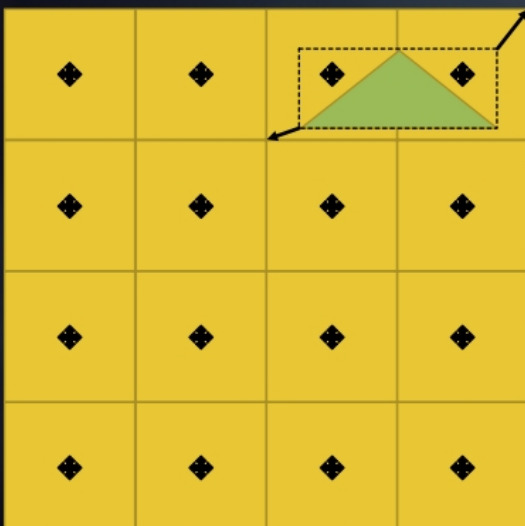
Small Primitive Culling (NDC)



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

$\text{any}(\text{round}(\text{min}) == \text{round}(\text{max}))$

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

$\text{any}(\text{round}(\text{min}) == \text{round}(\text{max}))$

上面几幅图，对了各种不同情况的三角形剔除。

具体做法是在屏幕空间构建一个三角形的包围盒，并将包围盒的最大值和最小值对齐到最近的像素点中心，如果最大值和最小值捕捉的水平方向和垂直方向一致，则保留。

Frustum Culling


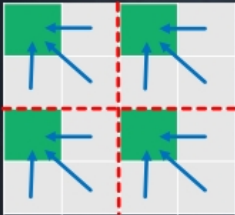
可以在NDC空间对三角形做视锥剔除。

Depth Culling

最难的一个。

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

这是算法：

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

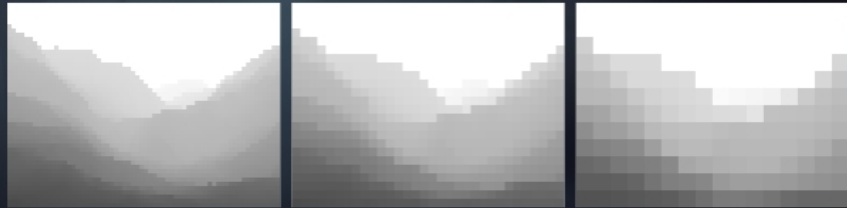
该算法通过Z—Pre Pass 来事先生成一个16X16的 深度图，通过三角形的包围盒与深度图对比来实现剔除，只能剔除很少一部分。

另一种算法就是使用HI-Z.

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

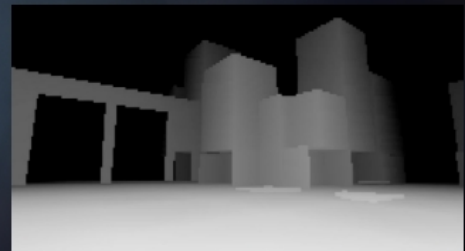
```
int mipMapLevel = min(ceil(log2(max(longestEdge, 1.0f))), levels - 1);
```



使用软光栅进行深度测试也是可行的

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



Batching and Perf

不看了，一些优化手段。