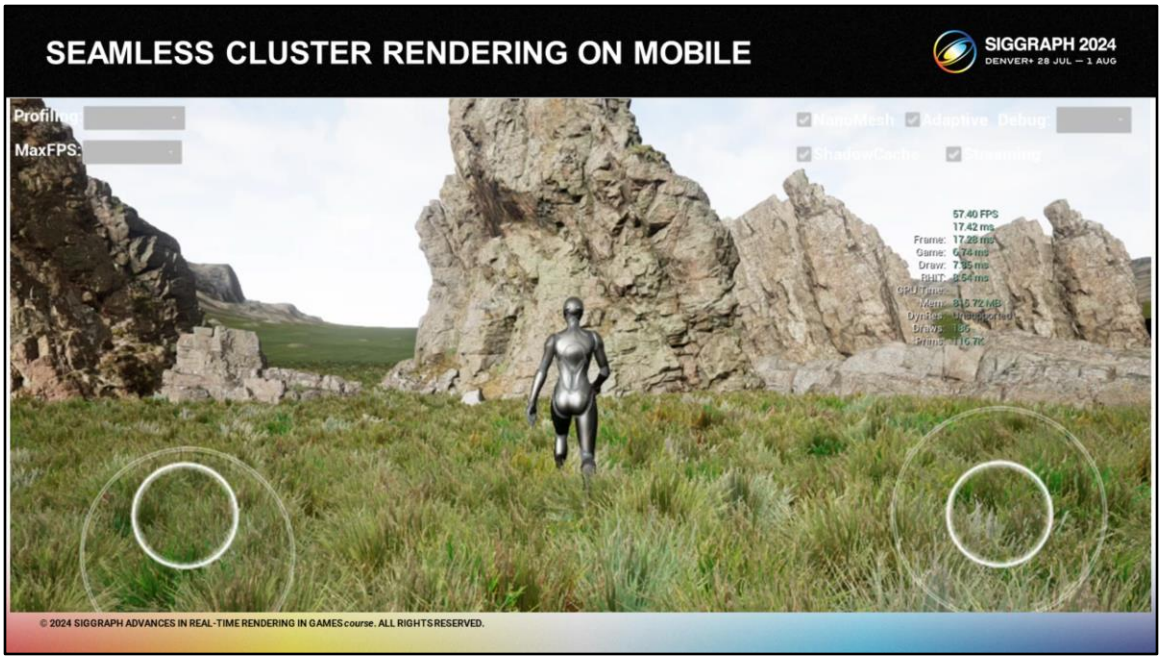
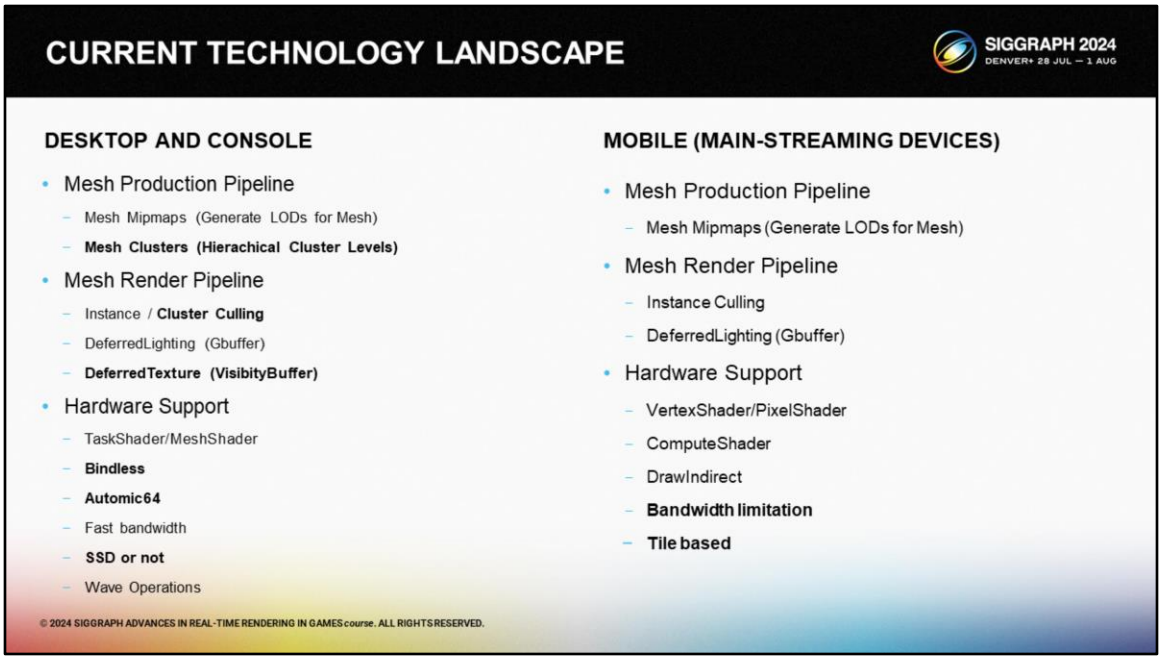
Hello everyone, I'm Shun from Tencent Games. My topic is seamless rendering on mobile.

We hope this rendering pipeline can simplify developers' work of customizing and tailoring mesh resources for different platforms, especially mobile, and reduce errors caused by manual operations. Of course, we also aim to ensure game graphics quality and interactive experience. It means refined rendering in areas that attract players' attention and no visual jumps or lags due to dynamic resource loading during gameplay.

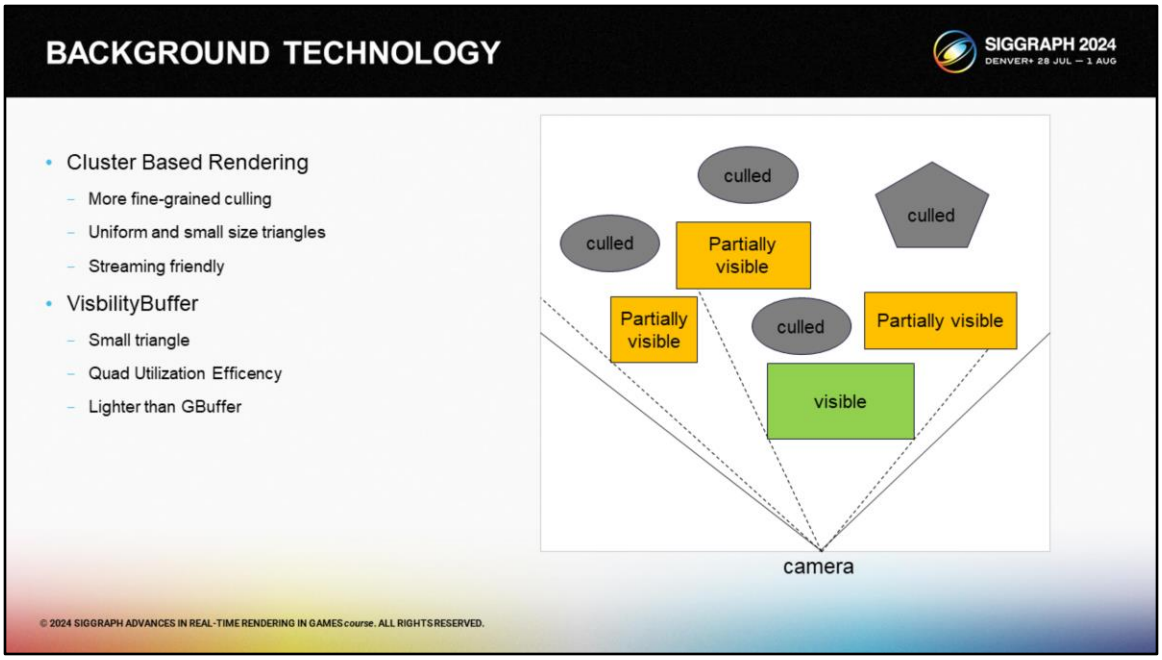


Talking alone might be abstract, so let's demonstrate with a test scenario. This is a rendering scene with 80 million triangles achievable on mobile. We only used high-precision assets in production, but during rendering, we control the granularity of clusters based on projection area and distance from the camera at the cluster level, ensuring optimal GPU resources for suitable effects.



However, compared to Desktop and console, mobile mainstream devices lack many hardware and software features necessary for seamless rendering, like rendering pipeline limitations, support for mesh shader and bindless features. Bandwidth and IO impacts on performance are also concerns.In addition, the mobile platform has its own tiled base GPU architecture.

So, how do we achieve seamless cluster rendering under these limitations? We aim for simplicity and efficiency, simplifying parts heavily reliant on hardware and bandwidth as much as possible.



Before diving into technical details, let's quickly review the background knowledge. First, Cluster-based Rendering: Occlusion culling is common in games, solving overdraw issues caused by direct object occlusion. But large objects often get fully rendered even if only part is visible. By dividing original meshes into small clusters and culling at the cluster level, GPUs can skip many invalid triangles. With fewer triangles per cluster, we can reduce vertex index precision, like using 8 bits. We can also load clusters on demand based on bounding boxes, improving GPU memory utilization.

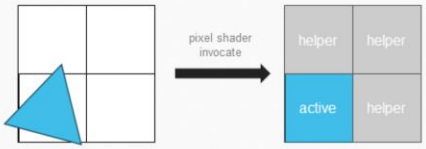
BACKGROUND TECHNOLOGY

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Cluster Based Rendering
  - More fine-grained culling
  - Uniform and small size triangles
  - Streaming friendly
- VisbilityBuffer
  - Small triangle
  - Quad Utilization Efficency
  - Lighter than Gbuffer

Triangle index	Instance index
22 bits	10 bits

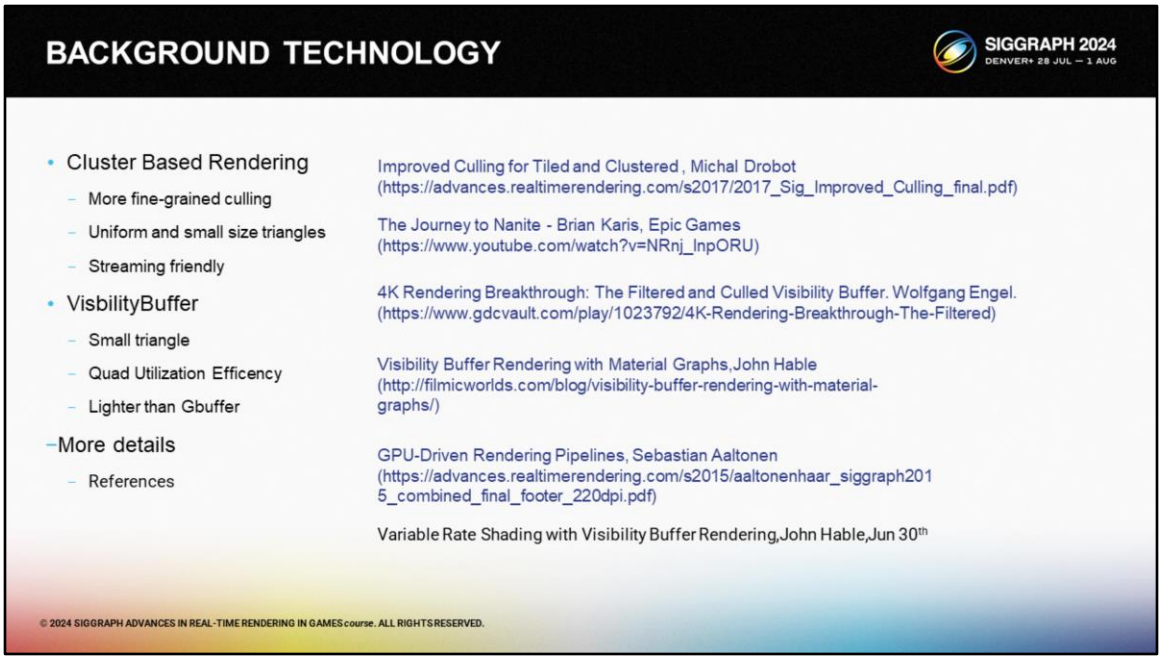
1 pixel triangle draw



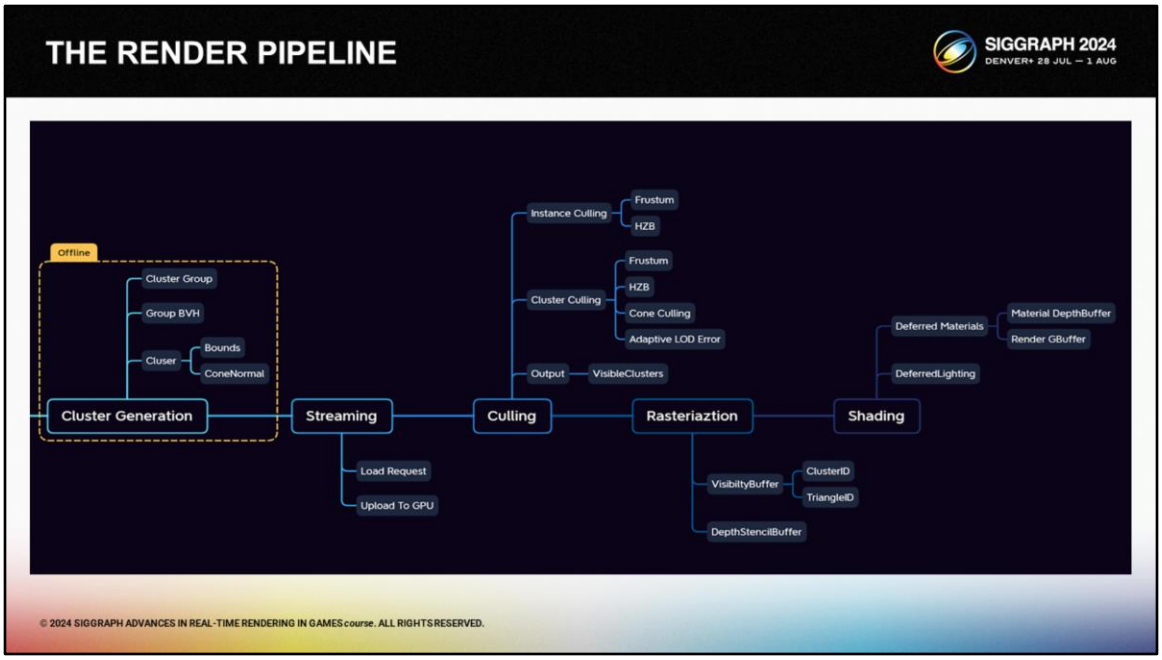
Shading invocations

	Material	Lighting
Forward	4x	
Deferred	4x	1x
Visibility	1x	1x

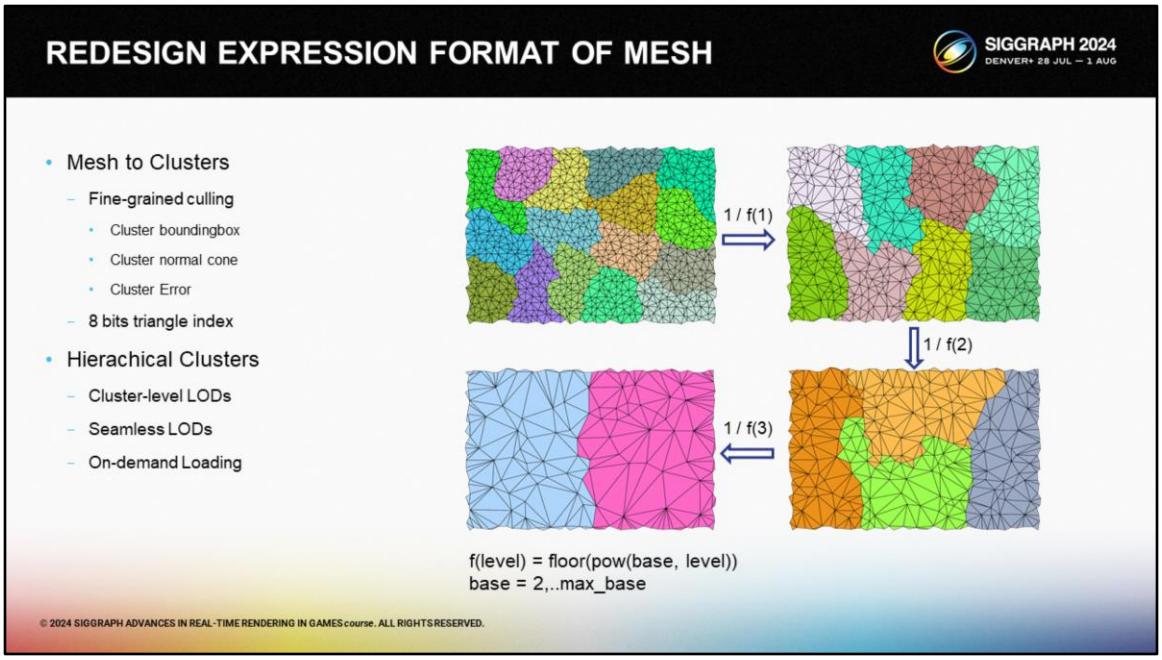
The second technology is Visbuffer. When rendering fine triangles, Visbuffer provides higher Quad Utilization than forward and deferred pipelines, with just 32 bits of extra overhead. John Hable will talk about vis buffer further in this 2024 advances.



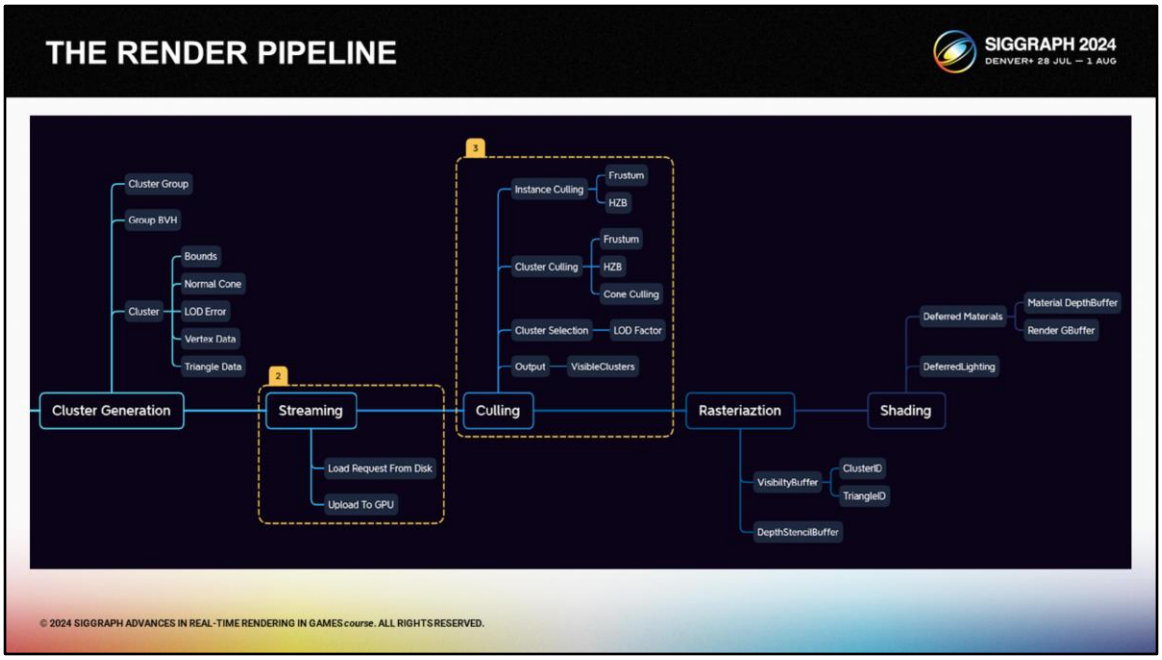
Thanks to Drobot‘s presentation in Advances, as well as Brian Karis’ talk,and Wolf.gang Engel‘s vis buffer original deck, also Sebastian’s SIGGRAPH 2015 talk . we can get more detail about those technologies . And John Hable will talk about vis buffer further in this 2024 advances



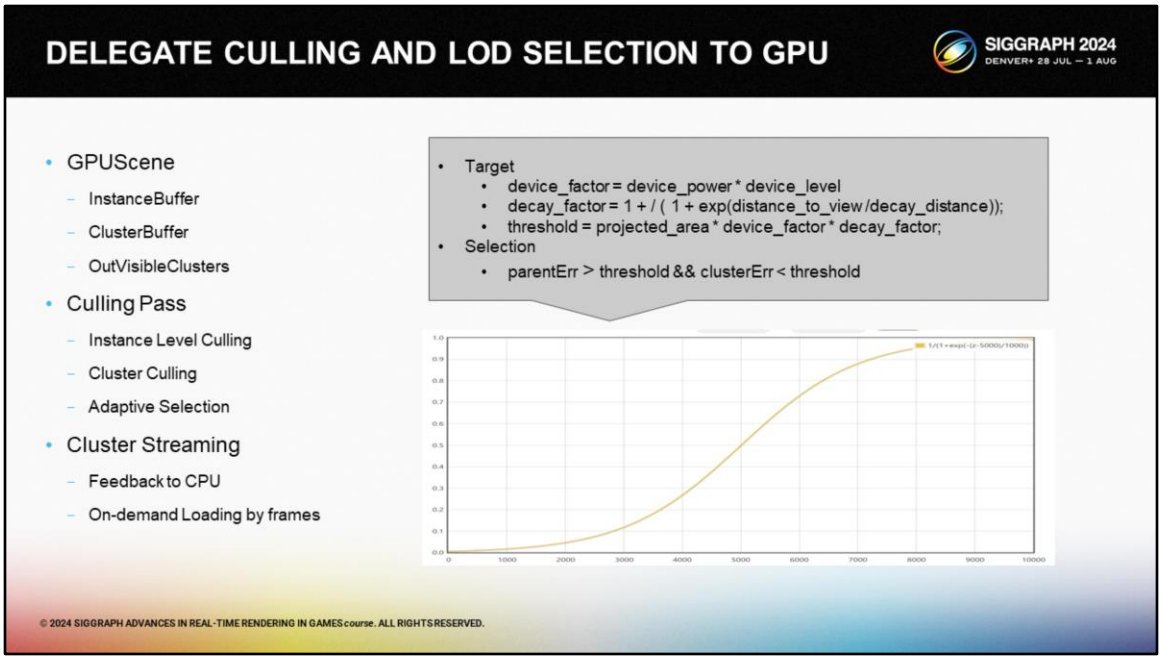
After introducing the background technologies, let's move on to the rendering pipeline architecture. The pipeline has two parts: offline and runtime. The offline stage imports and generates custom cluster data, while the runtime part includes streaming, culling, rasterization, and shading.



In the offline part, before anything else, we redesign the mesh storage structure. Instead of traditional multi-level LODs or simple clusters, we split meshes into many clusters and continuously recombine them into new, coarser-level clusters, iterating similarly. It's like UE5's Nanite, but we add a merge coefficient function to ensure controllability of assets and manually generated mesh LODs. This makes multi-level cluster data smaller, supporting data structures for culling and streaming. For each cluster, we store its bounding box and normal cone for efficient occlusion and backface culling. Each cluster has up to 128 triangles, so we can store indices with just 8 bits, saving space. For non-leaf nodes, we record errors from merging.



Next is the streaming and culling stages. Streaming starts with the roughest clusters from each instance, then loads on demand based on culling pass output.

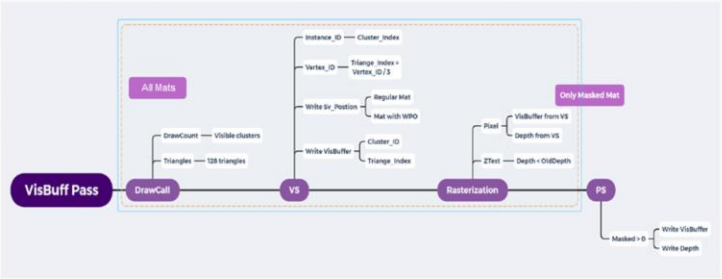


In culling, we let the GPU handle cluster control and selection. Objects are stored in an instance buffer, clusters in a cluster buffer, and IDs of successfully culled clusters in a visible cluster buffer. The cluster buffer loads dynamically based on GPU selection. Culling starts with fast instance-level culling using HZB, then applies various techniques like frustum culling, occlusion culling, and normal cone-based backface culling. We use LOD error similar to UE5's Nanite for cluster LOD culling, but we design a distance curve for LodFactor selection, ensuring enough rendering precision nearby and rougher triangles far away, mimicking manual mesh LOD factor control.

RENDER VISIBLE CLUSTERS TO VISIBILITYBUFFER

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

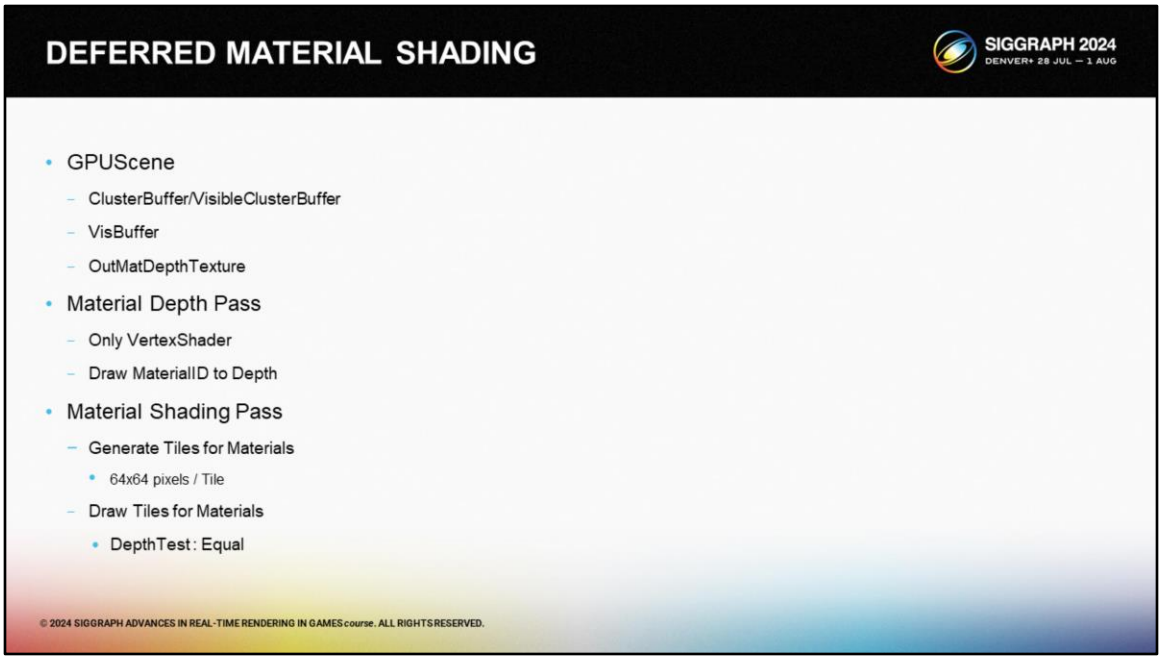
- Binning Pass
  - InVisibleClusters
  - OutBinningClusters
    - Regular Geometry
    - Deformed Geometry
    - Special Material Geometry
- Raster Pass
  - Simplest Indirect Draw
    - Only hardware rasterization
    - No extra depth buffer rewrite
  - Light-weight buffer and bandwdith
    - 32-bits VisBuffer
    - Cluster index and triangle index



VisibilityBuffer(32bits)

cluster index	triangle index
25	7

During rasterization, to reduce power consumption and GPU usage, we store intermediate rendering results in a 32-bit visbuffer. Since all triangles are in clusters, ideally, one draw call renders them all to the visbuffer. For GPU parallelism, we categorize objects: type 1 records cluster and triangle IDs in the visbuffer in the vertex shader, skipping pixel shader computation. Type 2, like skinned meshes, recalculates vertex positions with bone info in the vertex shader. Type 3, like masked vegetation, filters visbuffer generation based on masked textures in the pixel shader. We avoid soft rasterization due to extra scene depth passes, lack of atomic64 support, and higher bandwidth from 64-bit visbuffers. Since we render by clusters, not instances, 7 bits for triangle indices suffice, leaving space for clusters.

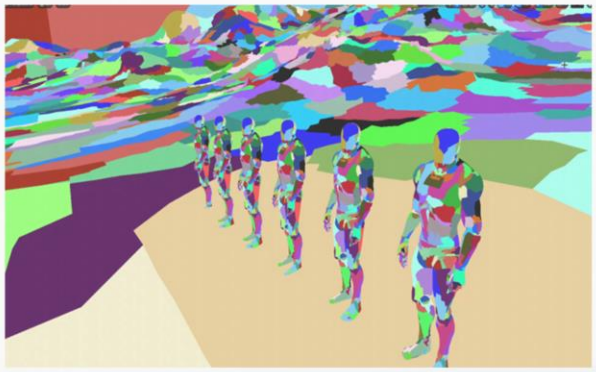


Since bindless is not yet supported on mobile devices, to achieve material shading, we need to render each material separately with a drawcall. Since it's rare for a single material to cover the entire screen, we divide the screen into many small tiles, and each material has its own list of tiles. This way, during rendering, we don't need to render each material across the entire screen. For each tile's rendering, we extract the materialID from the cluster information in the visbuffer and compare it with the current shading materialID. Only when they are equal will the material shading proceed.

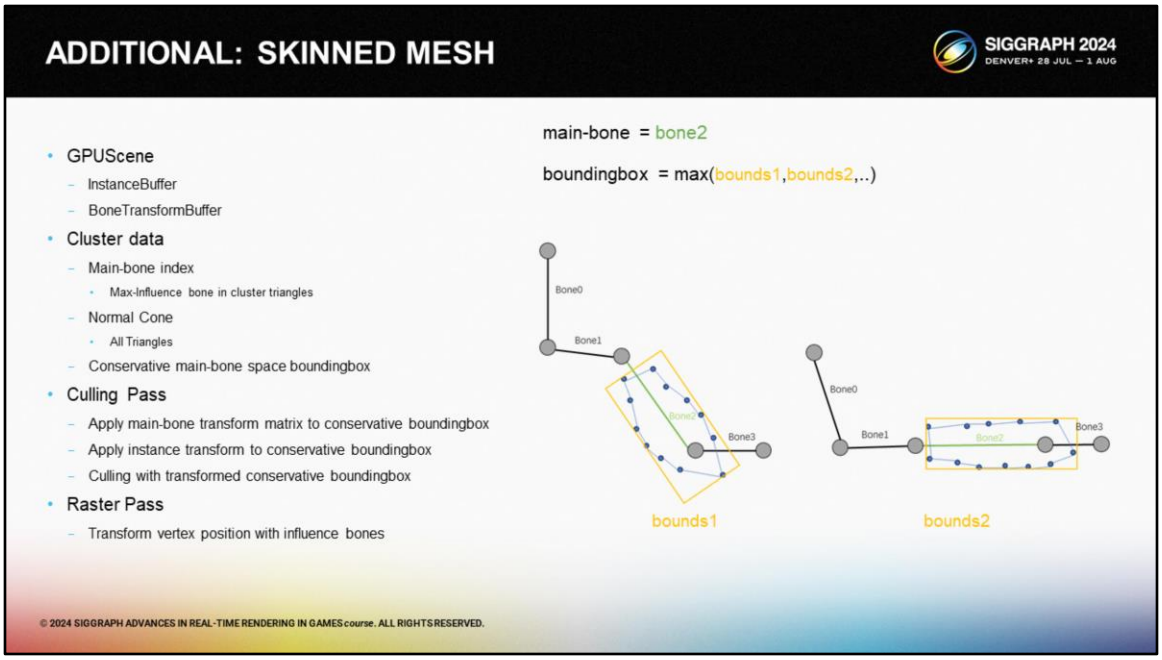
ADDITIONAL: SKINNED MESH

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- GPUScene
  - InstanceBuffer
  - BoneTransformBuffer
- Cluster data
  - Main-bone index
    - Max-Influence bone in cluster triangles
  - Normal Cone
    - All Triangles
  - Conservative main-bone space boundingbox
- Culling Pass
  - Apply main-bone transform matrix to conservative boundingbox
  - Apply instance transform to conservative boundingbox
  - Culling with transformed conservative boundingbox
- Raster Pass
  - Transform vertex position with influence bones



Rendering deformable meshes is inevitable in games. Take skinned meshes as an example. During animation, each cluster's bounds keep changing, making pre-calculated bounds for LOD and culling inaccurate. Real-time vertex-based bounds calculation is too time-consuming. So, we need special handling for skinned meshes. The approach is dynamic bound box calculation and culling based on clusters in the main bone space.

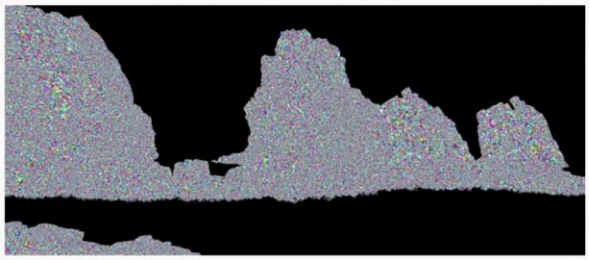


Offline, we compute each cluster's main bone, normal cone, and a conservative bounding box in the main bone space. The main bone is the one with the highest vertex weight in the cluster. The normal cone is the average normal of all cluster triangles, with a cone angle covering them all. The bound box covers the cluster's maximum range in the main bone space across all animations. Runtime, the CPU sends bone transform data to the GPU. During cluster culling, it reads the bone transform, transforms the conservative bound box to mesh space, then proceeds with normal cluster culling. Skinning rasterization follows normal skeletal animation calculations.

ADDITIONAL: GI FOR SEAMLESS RENDERING

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Lightmaps ?
  - Pixel level triangles
  - How to set Lightmap resolution ?
  - Is it easy to unwrap lightmap uv?
  - Memory / Storage controllable ?
- Hierarchical Voxelization Global Illumination



Atlas specification	width x height	mipmaps	format	Memory(MB)
Low Atlas	512 x 512	9	R8G8B8A8	> 1 MB
Medium Atlas	1024 x 1024	10	R8G8B8A8	> 4 MB
High Atlas	2048 x 2048	11	R8G8B8A8	> 16 MB

For Example, 100 instance will be from 100MB to 1600MB memory usage on mobile

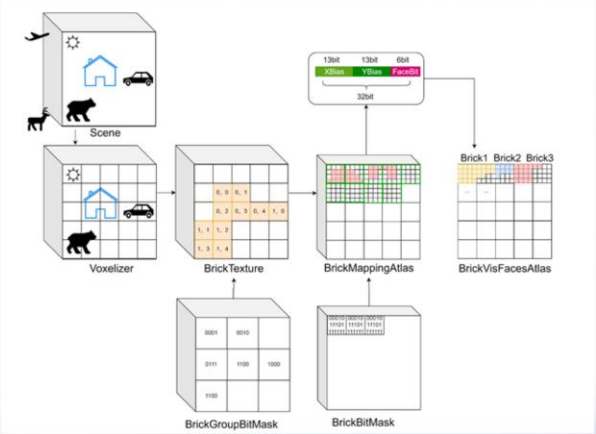
Seamless rendering, due to its inherent characteristics, employs special implementation methods and optimization techniques in lighting calculations. For instance, when calculating global illumination, we consider whether lightmaps can be utilized and whether their precision can cover pixel-level triangles. These factors significantly impact memory and storage. Hence, we've implemented a fully dynamic GI algorithm.

ADDITIONAL: GI FOR SEAMLESS RENDERING

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- No Lightmap
- Hierarchical Voxelization Global Illumination
  - BrickGI
    - Low Memory Usage
    - GPU Friendly Algorithm

  Voxelization for Clusters



The GI algorithm utilizes a hierarchical voxel-based approach. It works by dividing the scene into blocks, voxelizing each block, and leveraging these blocks and refined voxels for rapid raymarching. BrickTexture stores BrickData offset in BrickMappingAtlas.BrickMappingAtlas stores voxels offset in VisFaceAtlas

ADDITIONAL: GI FOR SEAMLESS RENDERING

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- No Lightmap
- Hierarchical Voxelization Global Illumination
  - BrickGI
    - Low Memory Usage
    - GPU Friendly Algorithm
  - Voxelization for Clusters

- Gathering lighting data from Bricks
  - Tracing from BrickGroup => Bricks => Voxels
  - Get lighting data from BrickVisFacesAtlas

```
// Voxel cell size is 0.5 meters
BrickVoxelSize=0.5
// a virtual voxel group
VoxelGroupSize=2x2x2
// voxel group cell size is 2 x 0.5 = 1 meters
VoxelGroupCellSize=1
// 8x8x8 voxels in one brick
BrickSize=8x8x8
// 4 meters per brick,8 cells multiply 0.5 meters
BrickCellSize=4x4x4
// 4x4x4 bricks in one brickgroup,fast access for uint64
BrickGroupSize=4x4x4
// 16x16x16 meters,one brick is 4 meters ,four brick is 16 meters
BrickGroupCellSize=16x16x16
// one group 16 meters,32x32x8 brick groups multiply 16 meters is 512x512x128 meters
BrickGroupBitMask=32x32x8
```


Here are the default config variables.

ADDITIONAL: GI FOR SEAMLESS RENDERING

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- No Lightmap
- Hierarchical Voxelization Global Illumination
  - BrickGI
    - Low Memory Usage
    - GPU Friendly Algorithm
  - Voxelization for Clusters

- Gathering lighting data from Bricks
  - Tracing from BrickGroup => Bricks => Voxels
  - Get lighting data from BrickVisFacesAtlas

```
BrickGroupTexcoord = (SamplePoint - BrickGroupStartPos) / BrickGroupCellSize;
// check any brick exist or not in 4x4x4 bricks
BrickGroupMask = BrickGroupBitMask[BrickGroupTexcoord];
uint64 bAnyBrickExists = BrickGroupMask != 0;
if (bAnyBrickExists)
{
    // ray marching on bricks and check visibity with BrickGroupMask
    BrickTexcoord = RayMarchingInBricks(SamplePoint,RayDirection,BrickGroupTexcoord,BrickGroupMask);
    // get brick storage in BrickMappingAtlas
    uint2 AtlasBias = BrickTexture[BrickTexcoord];
    // check any voxel exist or not in 4x4x4 voxel groups
    BrickMask = BrickBitMask[AtlasBias];
    bAnyVoxelExists = BrickMask != 0;
    if(bAnyVoxelExists)
    {
        // raymarching with current brick voxels with BrickMask
        SampleVoxelTexcoord = RayMarchingInVoxels(SamplePoint,RayDirection,BrickGroupTexcoord,BrickTexcoord,BrickMask);
        SampleVoxelTexcoord += AtlasBias * BrickSize;
        Opacity = BrickOpacityAtlas[SampleVoxelTexcoord];
        if(Opacity != 0)
        {
            // 8x8x8 voxels / brick
            PackedVoxelVisFaceInfo = BrickMappingAtlas[SampleVoxelTexcoord];
            BiasX,BiasY,FaceIndex = UnpackVisFaceInfo(PackedVoxelVisFaceInfo);
            VisFaceAtlasTexcoord = BiasY * VisFaceAtlasSize.x + BiasX + FaceIndex;
            // we get lighting data finally.
            VoxelLighting = VoxelVisFaceLightingAtlas[VisFaceAtlasTexcoord];
        }
    }
}
```


Here is the Pseudocode how we get lighting data from brick and visfacelightingatlas.When performing ray tracing, the first step involves quickly querying brickgroupbitmask to identify which bricks within a group are valid. This involves a single uint64 sampling operation. If the mask is not zero, it indicates that the current brick group contains valid bricks, and we can determine which specific bricks are valid by checking which bits in the mask are set.
After locating the brickindex through the combination of the bitmask and the sampleposition, it is straightforward to convert this index into bricktexcoord, allowing us to query the actual bias stored for the brick in the bricktexture. Subsequently, we utilize both the brickbitmask and the sampleposition to continue with a rapid ray marching process, which identifies the valid voxelcoord within the brick. Then, in the brickmappingatlas, we locate the offset of the voxel within the visfaceatlas, and proceed to read the relevant lightingdata.

ADDITIONAL: GI FOR SEAMLESS RENDERING

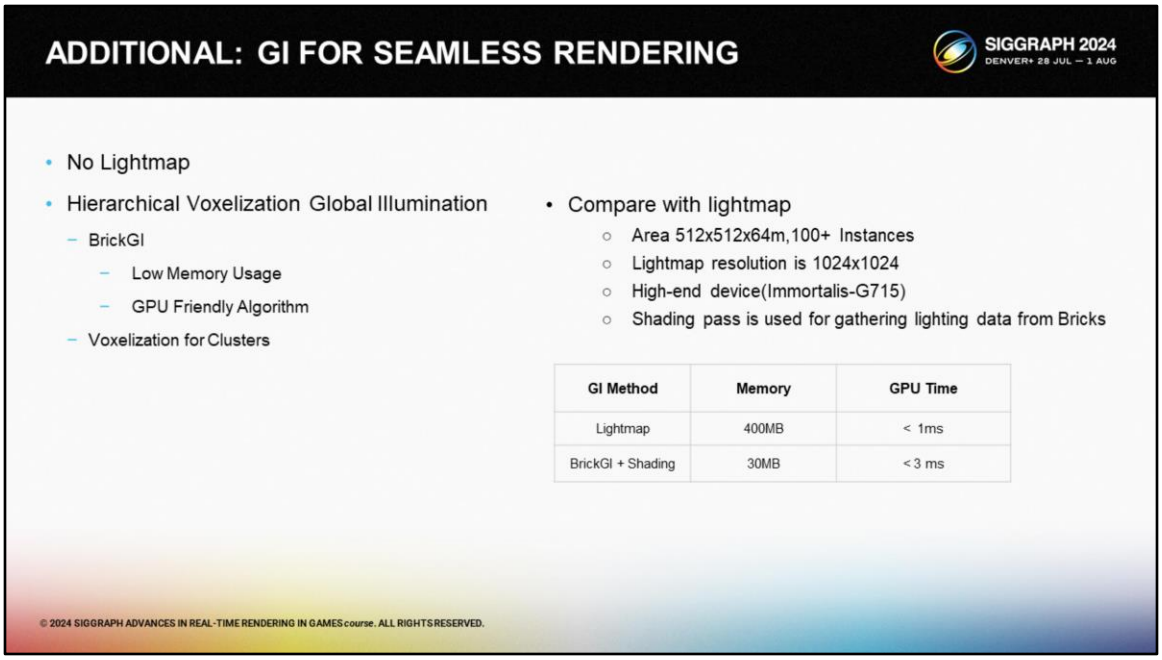
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- No Lightmap
- Hierarchical Voxelization Global Illumination
  - BrickGI
    - Low Memory Usage
    - GPU Friendly Algorithm
  - Voxelization for Clusters

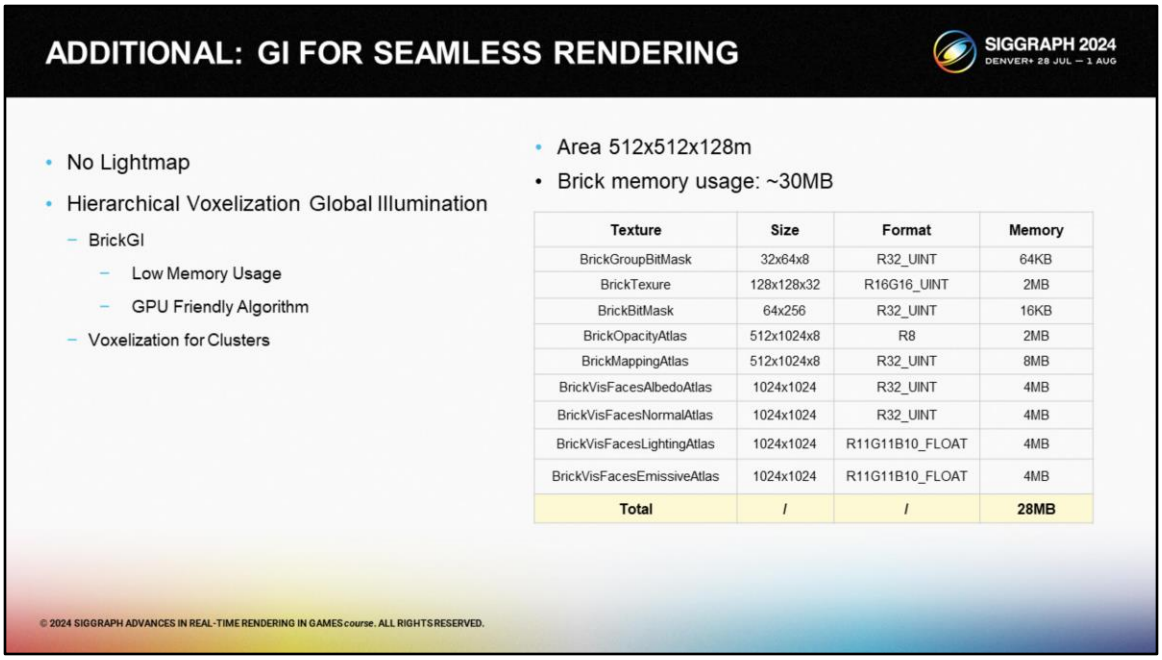
- Gathering lighting data from Bricks
  - Tracing from BrickGroup => Bricks => Voxels
  - Get lighting data from BrickVisFacesAtlas

```
int3 RayMarchingInBricks(SamplePoint,SampleDir,BrickGroupTexcoord,BrickGroupMask)
{
    step = 0;
    while(step < 64)
    {
        SampleBrickTexcoord = (SamplePoint - BrickGroupTexcoord * BrickGroupCellSize) / BrickCellSize;
        BrickIndex = BrickTexcoord.x + (BrickTexcoord.y + BrickTexcoord.z * 4) * 4;
        if( (1 << BrickIndex) & BrickGroupMask)
        {
            return SampleBrickTexcoord;
        }
        else
        {
            SamplePoint += SampleDir * BrickCellSize;
        }
    }
    return int3(-1,-1,-1);
}
```

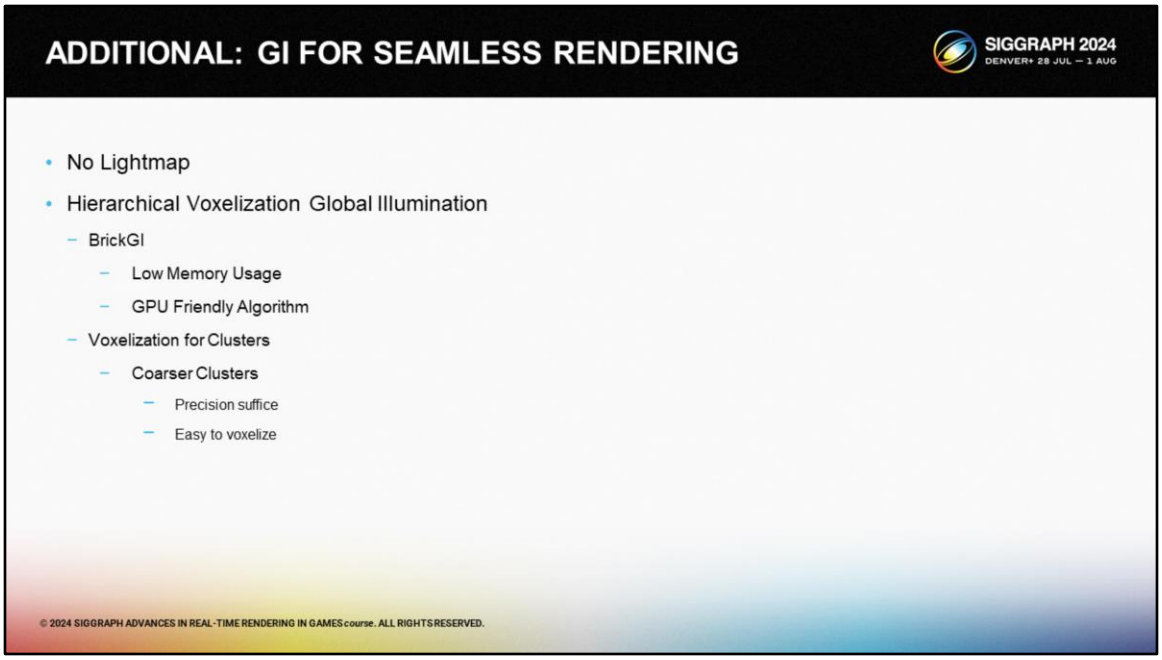

Here is the Pseudocode how we get valid brick from brickmask.



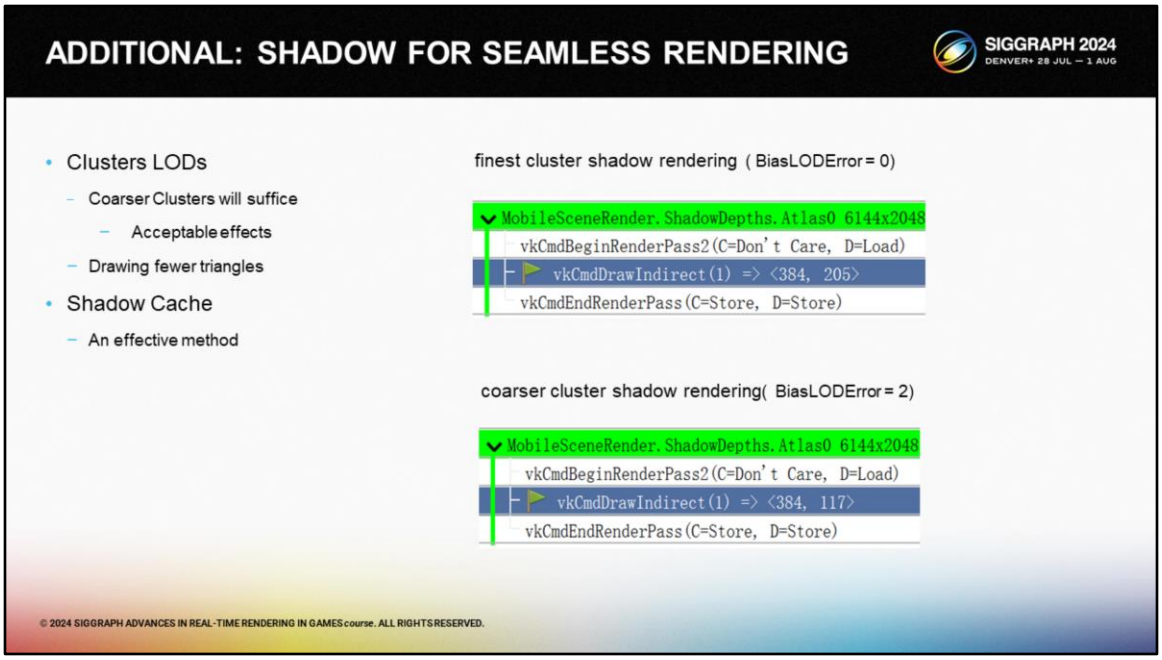
Let's compare the performance between brickgi and lightmap on mobile devices. In this test, we added over 100 objects, each with a lightmap resolution of 1024x1024, within a scene area of 512x512 meters. Our findings show that lightmap requires at least 160MB of memory overhead, whereas brickgi only needs 30MB. Additionally, while the GPU time for brickgi is slightly higher than that of lightmap, 3ms is still considered an acceptable latency for most mobile games.



This table shows the details of 30MB memory allocated to cover an area within 512 meters. Moreover, high-precision clusters aren't necessary for voxelization calculations, keeping the overall voxelization cost low.



Moreover, high-precision clusters aren't necessary for voxelization calculations, keeping the overall voxelization cost low. Voxelization inherently has limited precision, and clusters with lower precision are more easily voxelized through rasterization



Regarding direct lighting calculations, our tests indicate that high-precision clusters for generating shadow depths aren't essential. Using coarser approximations has minimal impact on visual quality but significantly enhances performance. For example, the upper image uses a more detailed rendering requiring over 200 drawcalls, while the lower image achieves similar results with around 100 drawcalls.

ADDITIONAL: SHADOWS FOR SEAMLESS RENDERING

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Clusters LODs
  - Coarser Clusters will suffice
    - Acceptable effects
  - Drawing fewer triangles
- Shadow Cache
  - An effective method

finest cluster shadow rendering (BiasLODError = 0)

coarser cluster shadow rendering(BiasLODError = 2)

The upper image here shows the effect with 200 drawcalls, and the lower image displays the shadowmap with 100 drawcalls. The difference is barely noticeable. Additionally, one of the most effective methods is shadow caching.

Let's look at seamless rendering's performance, comparing effects, framerates, power consumption, package size, and production efficiency.

TEST ON MOBILE ENVIROMENT

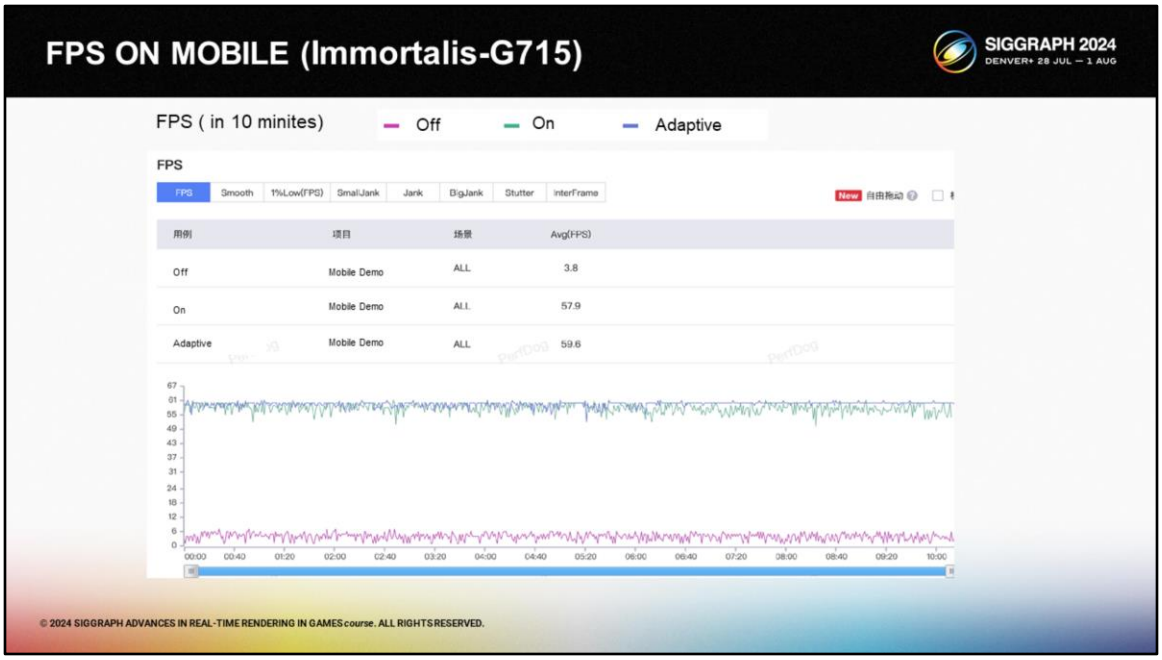
SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Benefits
  - Quality
    - More than 80 million triangles
  - FPS
    - 60 FPS On Mobile
  - Power
    - ~ 128 mW
  - Package Size
    - Only 1 / 4 vs Manual Mesh Level LOD
  - Productivity
    - More 4x efficency

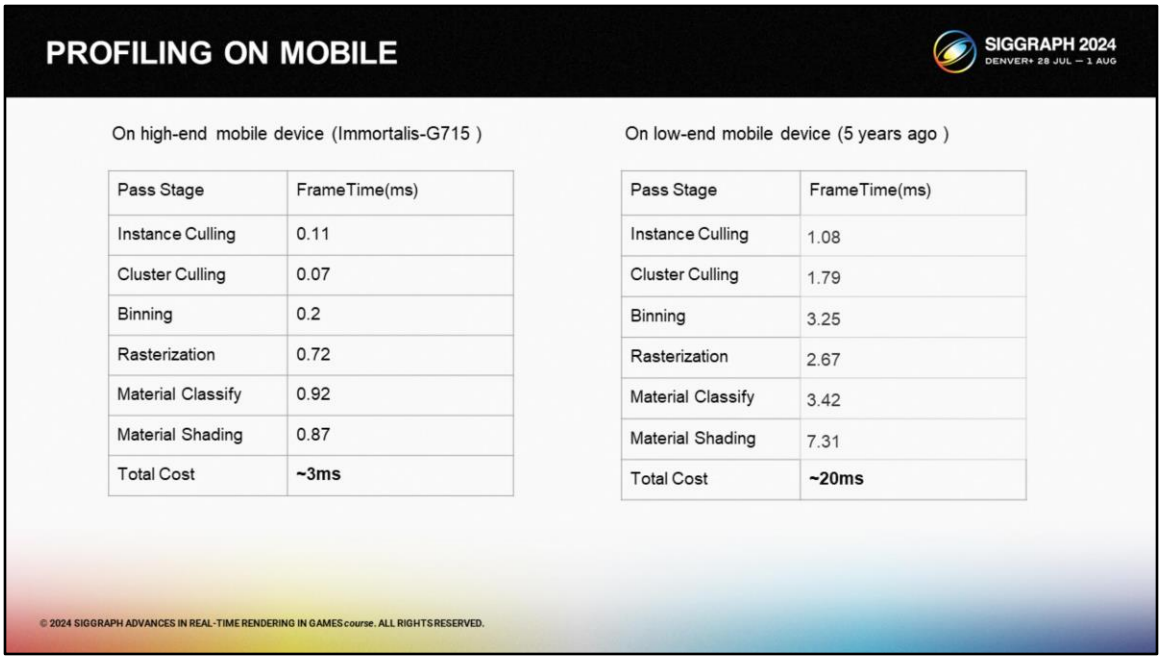
Scene

Type	Mesh count	Triangles of Mesh	Total
Rock	40	2 million	80 million
Landscape	4096	1024	4 million
Grass	10000+	20	2 million

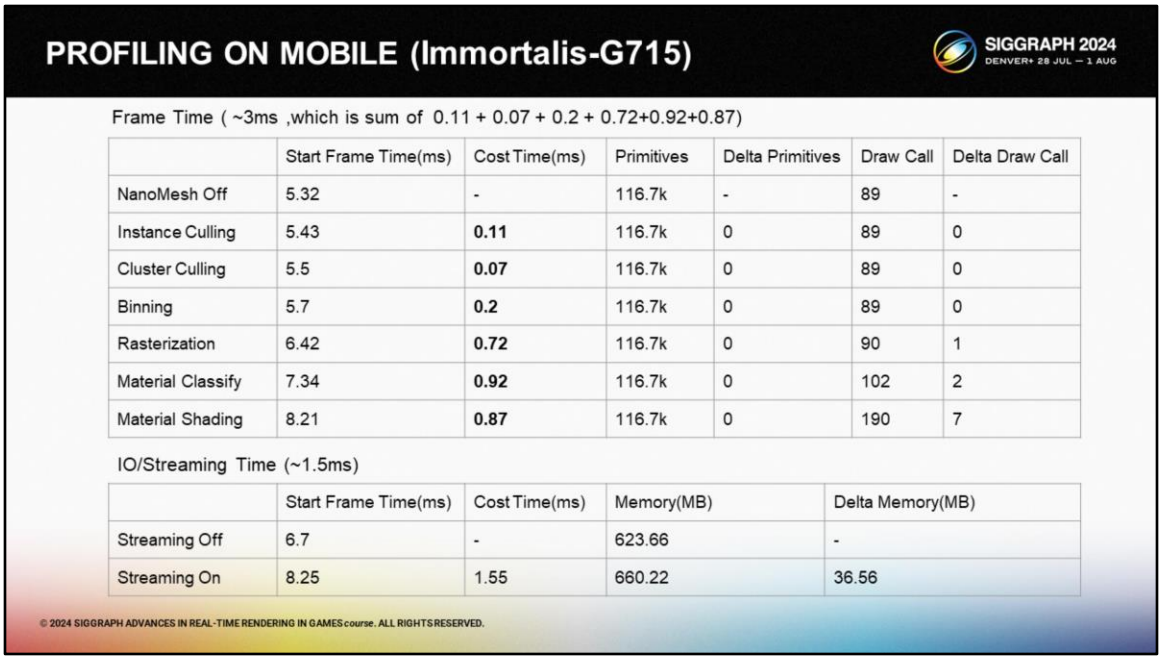
Quality-wise, we can render 80 million triangles simultaneously, unimaginable on mobile, especially retail devices. With this many triangles, we maintain stable framerates and acceptable power consumption. We also save on package size. Plus, we don't need to manually create LODs; just LOD0 assets suffice.



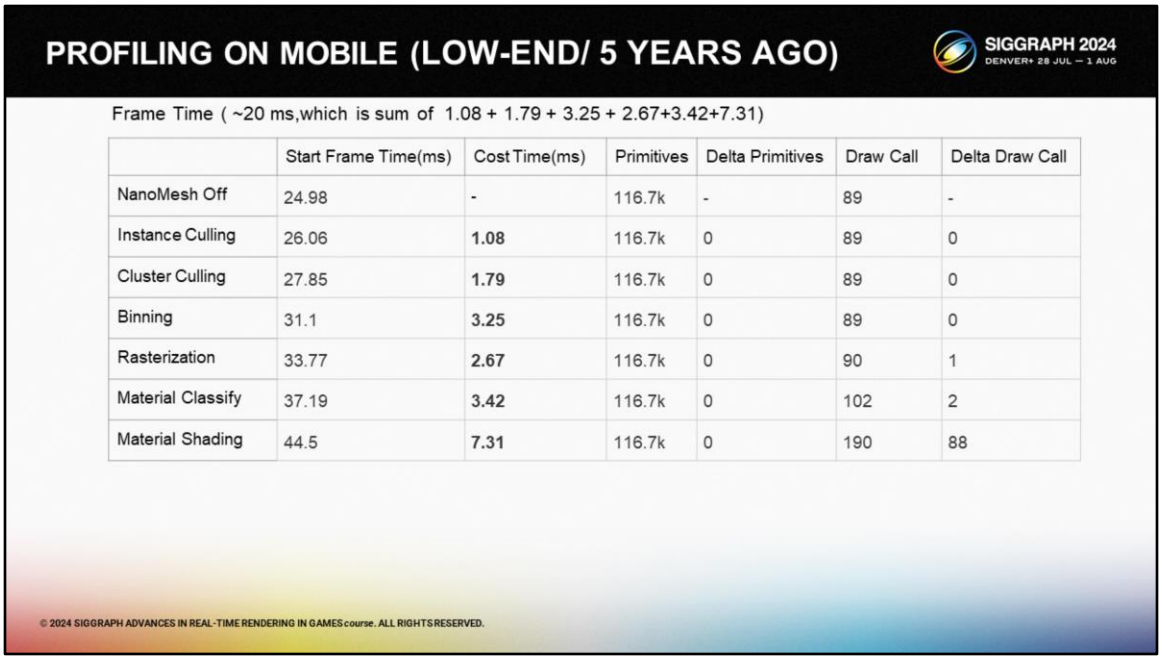
Here's the framerate on a mobile device. Blue and green lines show the rendering effect after using seamless mobile rendering. Notice the blue curve, maintaining around 60fps with minimal fluctuations.



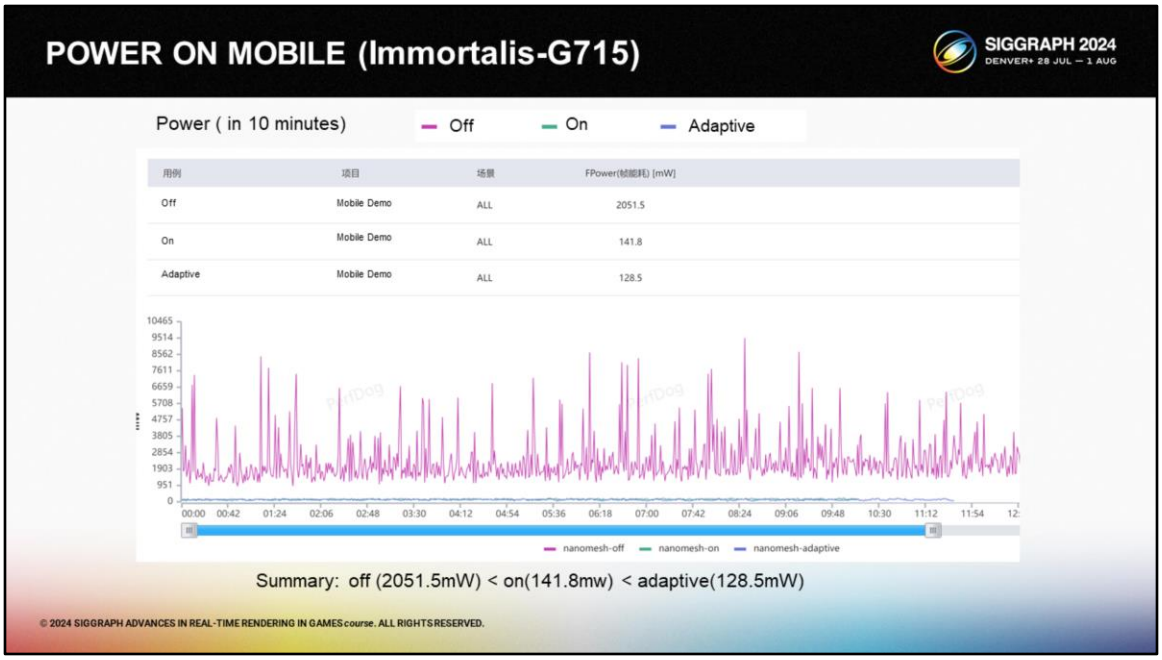
Take a look at the detailed performance data on different level mobile device. On high-end device ,it only take about 3ms gpu time,while on low-end mobile device which is 5 years ago,it will be about 20ms.



We can also take a look at the detailed performance data on high-end mobile device. In the table above, I've broken down each rendering phase and monitored the changes in GPU time consumption step by step. If enable instance culling,it will take 0.11 ms, and then turn on cluster culling,it take 0.07ms , and then turn on binning pass ,it take 0.2 ms ,then rasterization pass which take 0.72ms ,then material classify pass take 0.92ms ,the last pass is material shading which take 0.87ms. We can observe that the time consumption for culling is not significant, and in this scenario, rasterization and material shading is the more time-consuming part. The table below shows the time consumption of IO Streaming, and we can see that the GPU's IO performance is also acceptable.

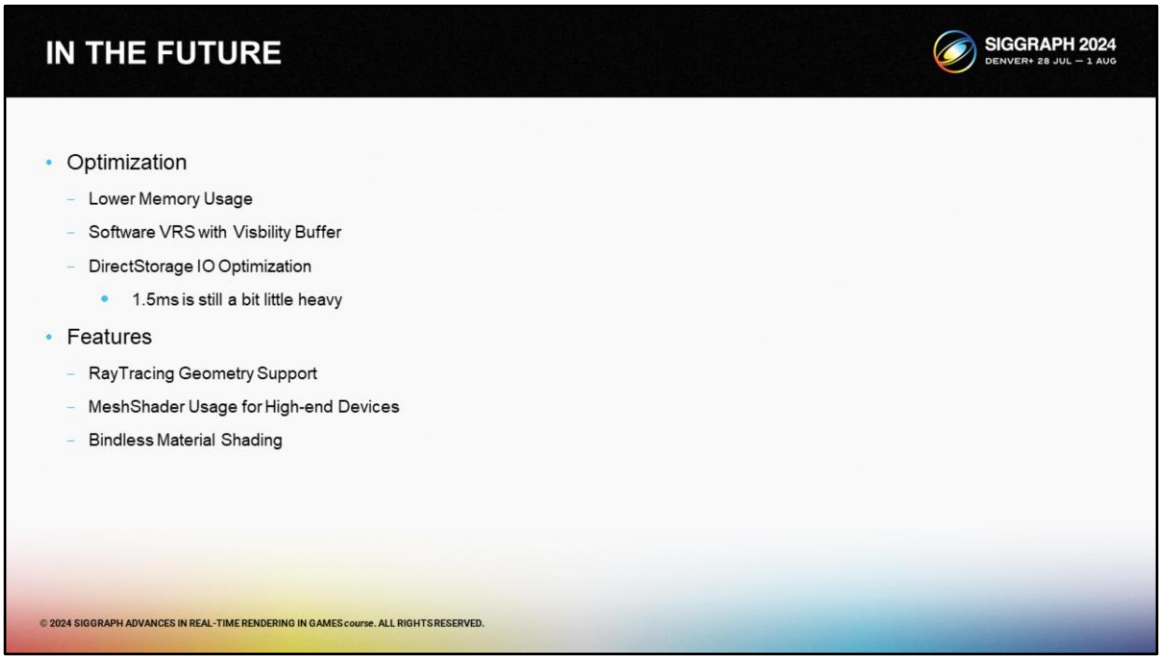


We can also take a look at the detailed performance data on low-end mobile device which is 5 year ago. We can observe that the time consumption for culling is not significant, and in this scenario, material shading is the most time-consuming part.



This is power consumption over 10 minutes on a mobile device. Seamless rendering on mobile (blue) performs well. We found that after enabling this, the power consumption is significantly lower than that of traditional mesh rendering mode.

These are our research and validation results, but there's much room for improvement. This is just the beginning.



We'll continue optimizing performance and iterating features based on advancements in rendering technology and hardware capabilities.such as memory usage,software vsr with buffer,io optimization,1.5ms is still a bit little heavy. For features,We are not supporting raytracing now.And with mobile hardware capabilities development,we will add meshshader and bindless material shading support.

Thank you for your time. Feel free to reach out to me if you have any further questions.