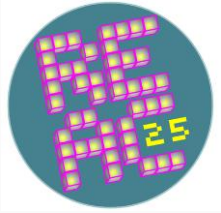
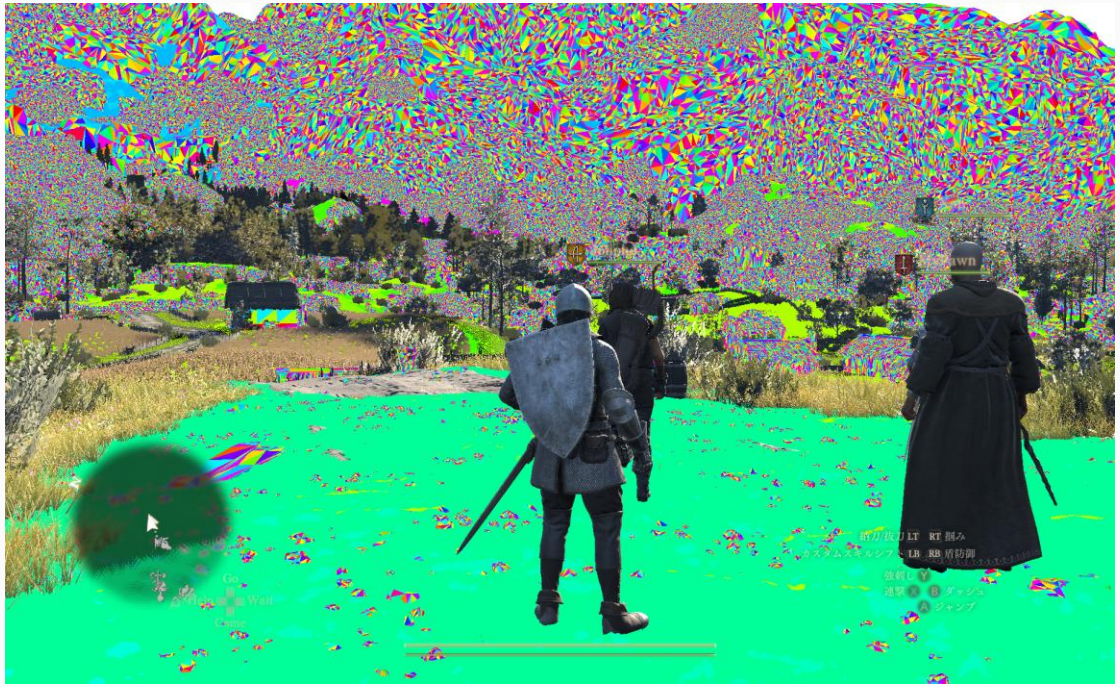


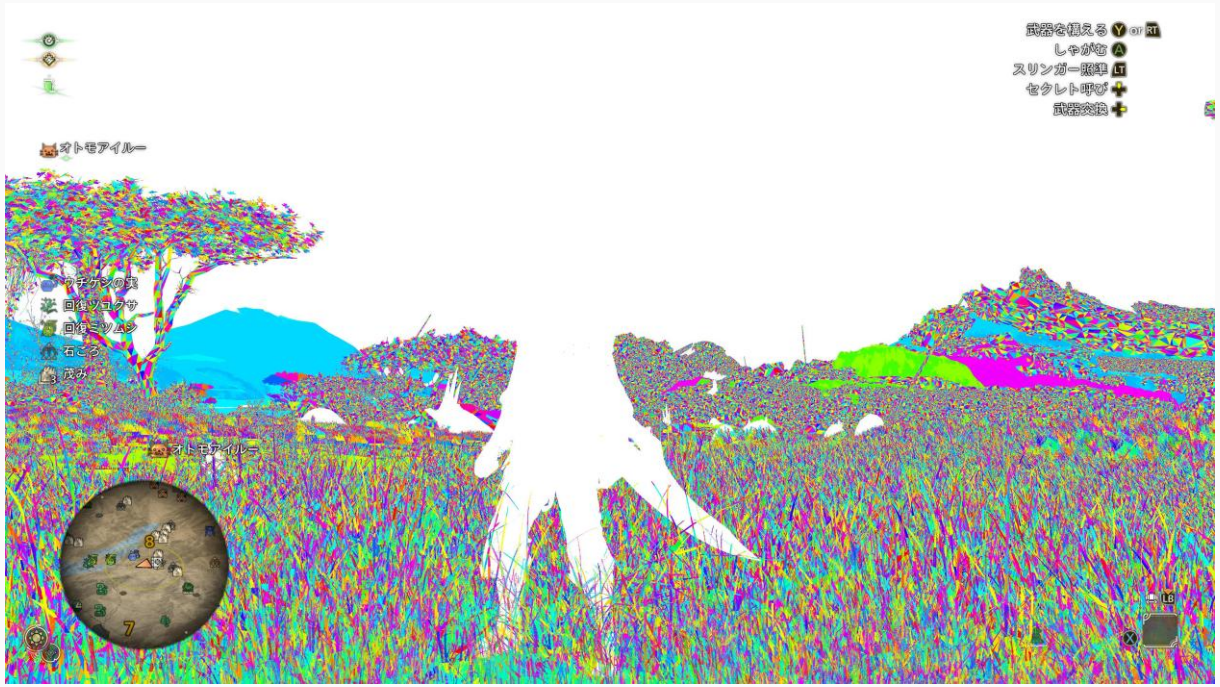
RE ENGINE
Meshlet Rendering Pipeline

A1

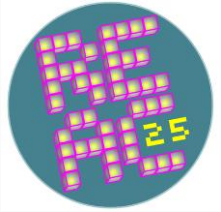
Games Utilizing the Meshlet Rendering Pipeline

Games Utilizing the Meshlet Rendering Pipeline





Agenda



RE ENGINE Team

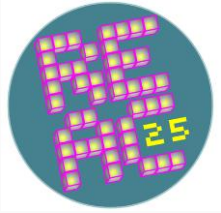
Previous Rendering Pipeline

Integrating the Meshlet Rendering Pipeline

Optimizing Time Changes

SL3

RE ENGINE Team



Independent from game development teams.

Supplies engine for each game.

Games prohibited from modifying engine.

A1

Handles feature additions, optimizations, and QA with game teams.

- Engine branch made for game when near completion.
- Revisions to engine are quickly reflected in game branches.

A2

SL2

Previous Rendering Pipeline

Used Multi-Draw Indirect and GPU-based occlusion culling.

- Small objects used instance-based occlusion culling.
- Large background meshes consisting of multiple objects divided into 256-triangle sections then occlusion culling performed.
  - Optimized for traditional single-body meshes with no extra steps needed.
  - Used occluders created by artists.

A1

Limitations of Instancing:

- Possible to render instances where bindless is applied and materials differ.
- Unable to handle differences between geometry and LODs due to CPU limitations.

Normal rendering

Occluder placed in front of camera

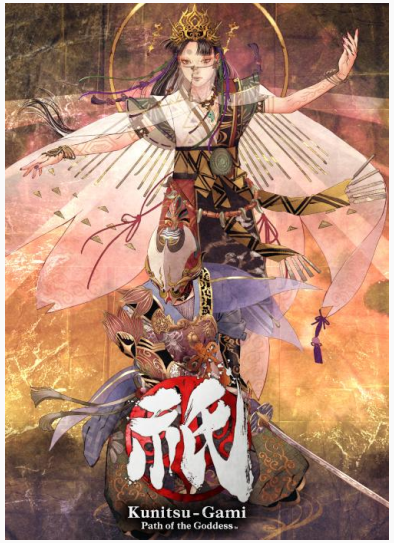
A3

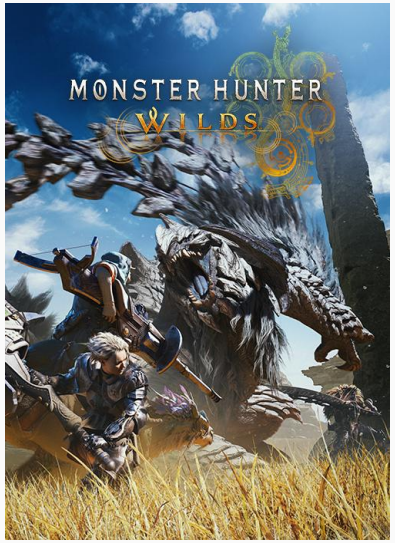
Rendering Vast Environments

Dragon's Dogma 2 and Monster Hunter Wilds

- Generate structures from separate models.
- Large amounts of geometry.
- Changing time of day.

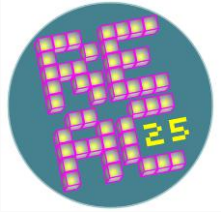
Productivity enhancements needed.





A1

Switching to the Meshlet Pipeline



Nanite is a rendering engineer's dream-come-true.

Automatic occlusion culling, auto-generated LODs, streaming, software rasterizer, visibility buffer, and more.

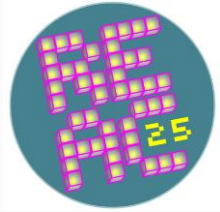
Problems to Solve:

CPU/GPU capability and efficiency enhancements:

1. Meshlets
2. Two-phase Occlusion Culling
3. Visibility Buffer
4. Software Rasterizer

Automatic LODs difficult to implement with current resources.

Dragon's Dogma 2



Background only supports meshlets for static meshes.

- Large meshes use a single hardware mesh shader or vertex shader.
- Small meshlets use software rasterizer.

Uses deferred rendering with visibility buffer.

- Same shader initialized only once.
  - All materials are processed as bindless.
- Some alpha tests and decals use special rasterizers.

A1

Meshlets

Meshlets are fragments of meshes divided by a defined granularity.

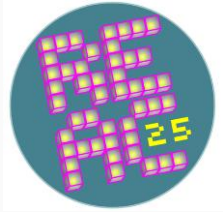
RE ENGINE can specify up to 128 vertices and 128 triangles.

Dragon's Dogma 2 and Monster Hunter Wilds specify both as 128.

Asset size changes depending on what is specified as shown below.

vrt32 tri32	vrt64 tri64	vrt64 tri126	vrt128 tri128
39,763 KB	35,386 KB	34,094 KB	33,231 KB

Meshlet Structure



Uses data converted offline.

Mesh represented with one ByteAddressBuffer.

The header is a 256-byte structure.

Determining LODs:

GUP determines LODs using LOD factor.

Clusters are referenced with lodMeshletOffset.

Type	Header
Float3	AABBMin
Int	lodNum
Float3	AABBMax
Int	validLodBits
Int[8]	lodMeshletsOffset
Int[8]	lodFactor
Int[8]	bindlessGeometryOffset

Supports Streaming Structure:

LODs are stored from least to most detailed.

Uses ReservedResource(TiledResource) for memory management.

Meshlet Clusters

Each cluster stores compression options and more in header.

Vertex data is stored directly within each cluster.

  Divided by index and vertex attributes for compression.

Two Types of Compression:

Vertex Quantization

 16bit, 10bit, or 8bit precision.

Index is 8-bit triangle list or triangle strip.

Attribute Compression

 Attributes include the normal, tangent, UV, vertex color, etc.

 Bit flag enables compression.

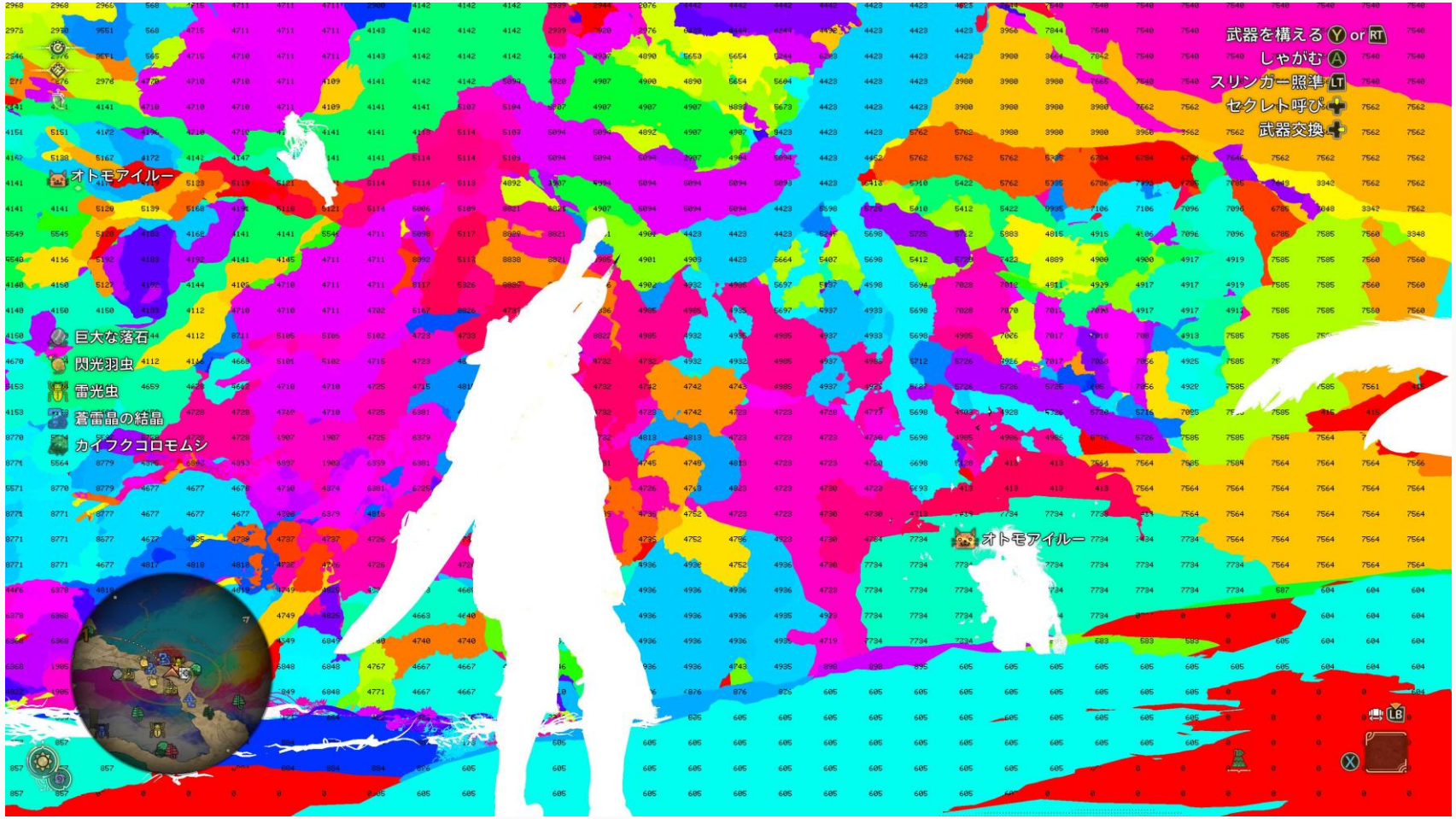
 Check if all elements in attribute are same value.

  If all attribute values the same, stores as a single value.

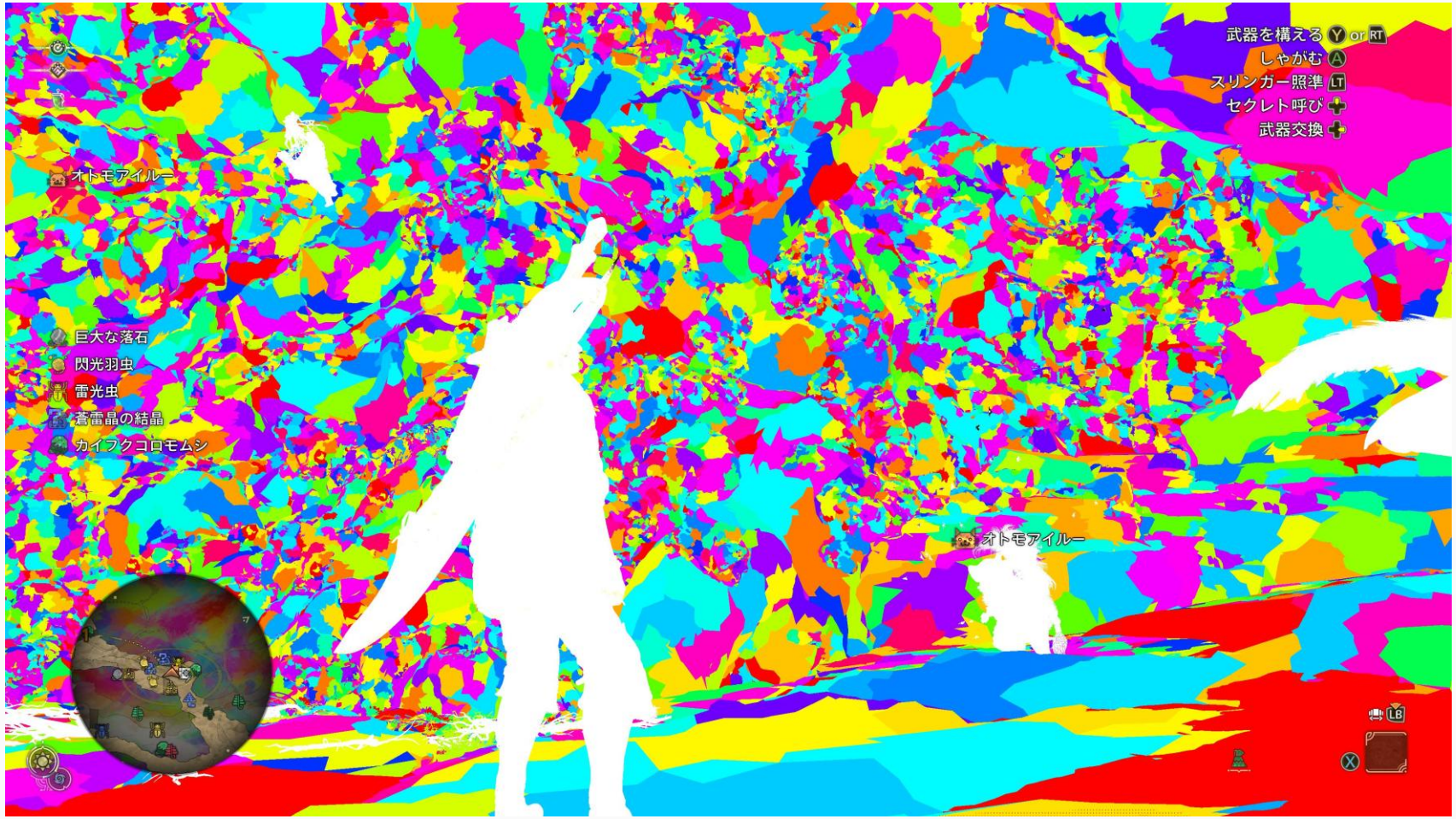
Type	MeshletHeader(32Bytes)
Float3	AABBCenter
bytes4	vertexNum(byte) primitiveNum(byte) materialID(byte) partsID(byte)
Float3 / ushort[6]	AABBExtent / AABBQuantizeCenter AABBQuantizeExtent
Uint / Bitflag	isMeshletCompressedNormal isMeshletCompressedTexcoord1 isMeshletCompressedTexcoord2 isMeshletCompressedTexcoord3 isMeshletCompressedVertexColor isMeshletCompressedSkined isMeshletNoTangent Reserved isMeshletUseVertexColor isMeshletUseTexcoord2 isMeshletUseTexcoord3 isMeshletUseSkinned .....

Compression Example: Vertex Colors

Instances


武器を構える Y or RT
しゃがむ A
スリンガー照準 LT
セクレト呼び
武器交換
オトモアイルー
巨大な落石
閃光羽虫
雷光虫
蒼雷晶の結晶
カイフクコロモムシ
オトモアイルー
LB
X

Meshlet Clusters

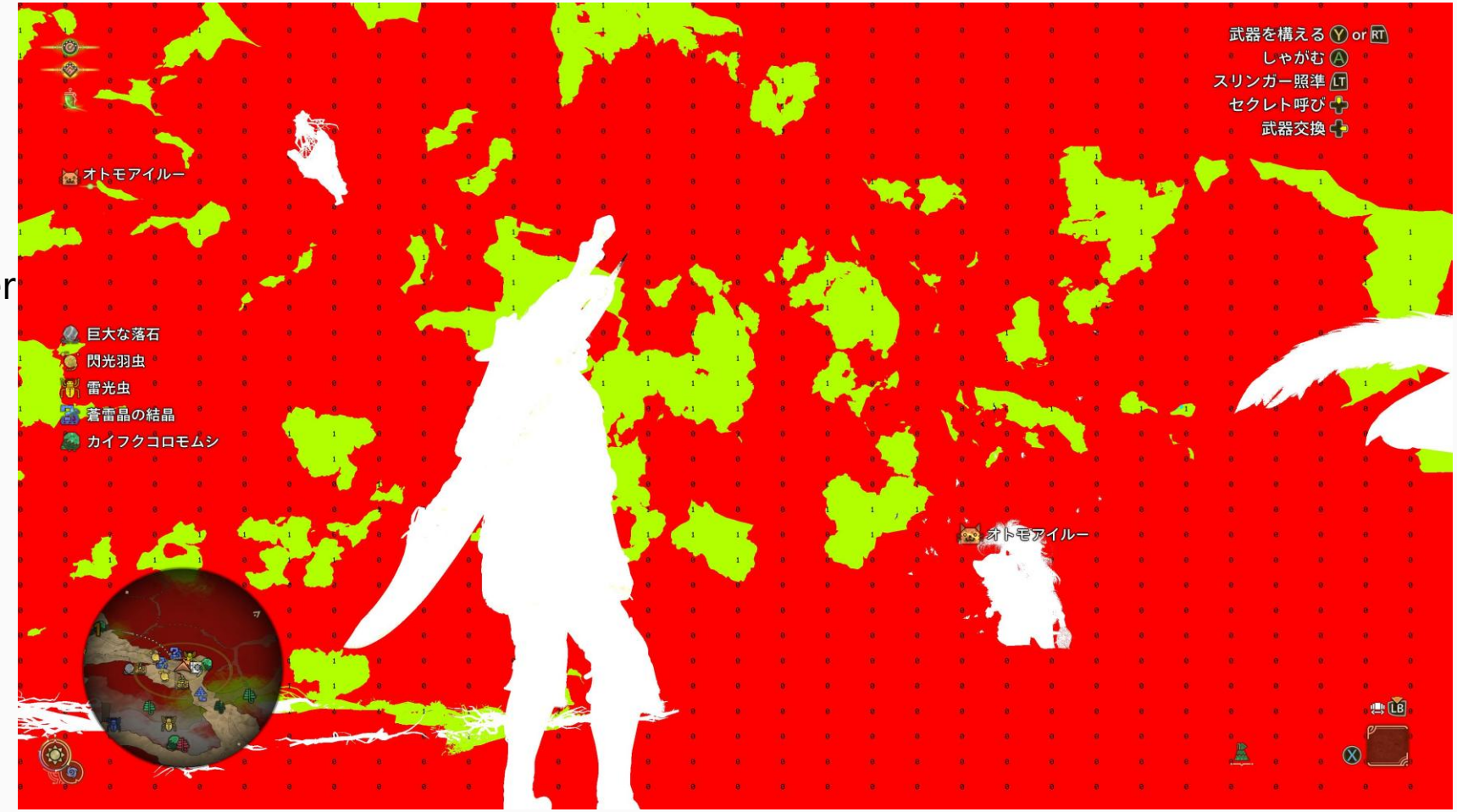


Vertex Color Compression

Red:
No vertex color

Yellow:
One-color cluster

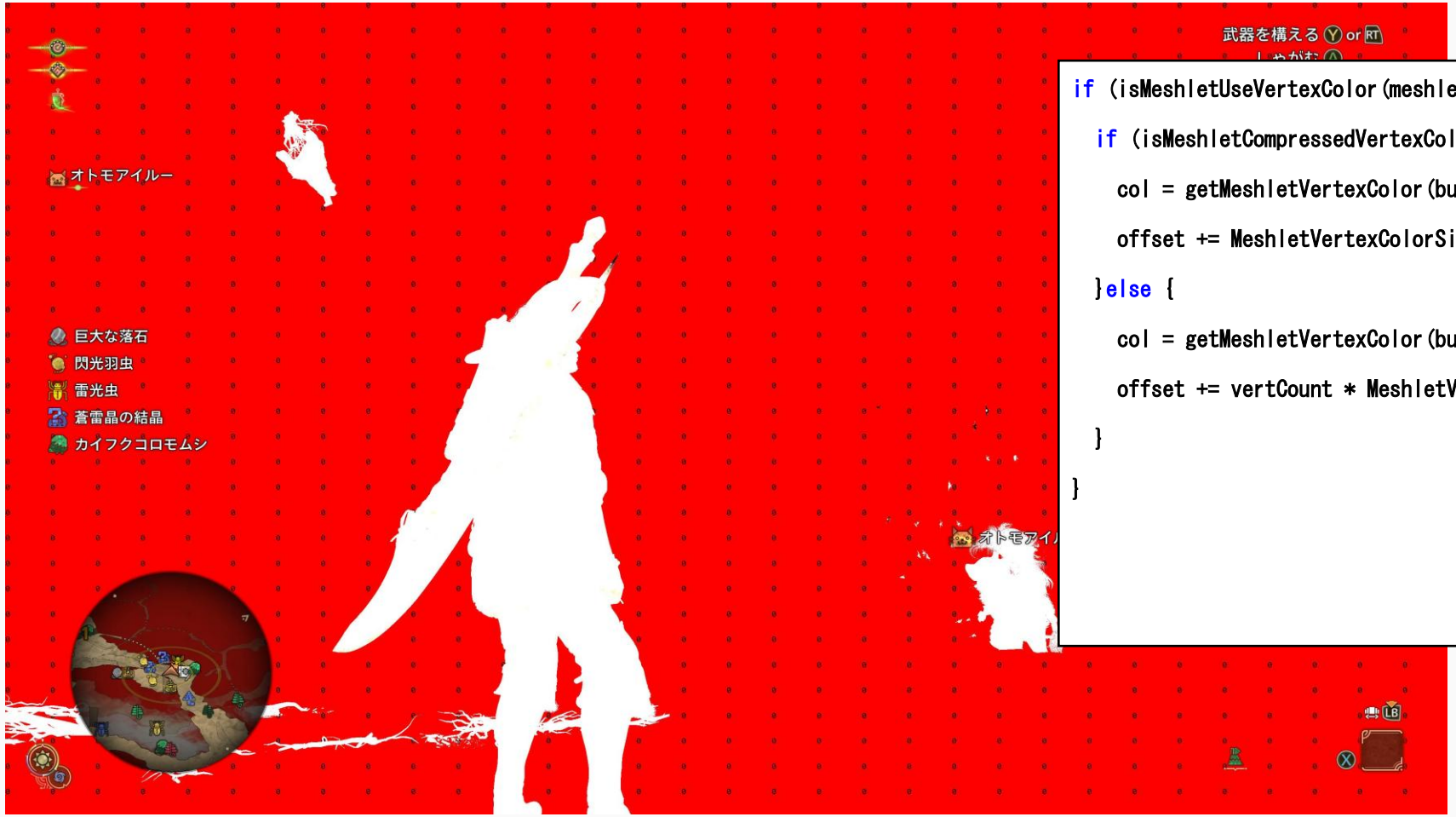
Blue:
Multiple-colors



SL2

VertexColorCompression

Yellow:
Multiple colors



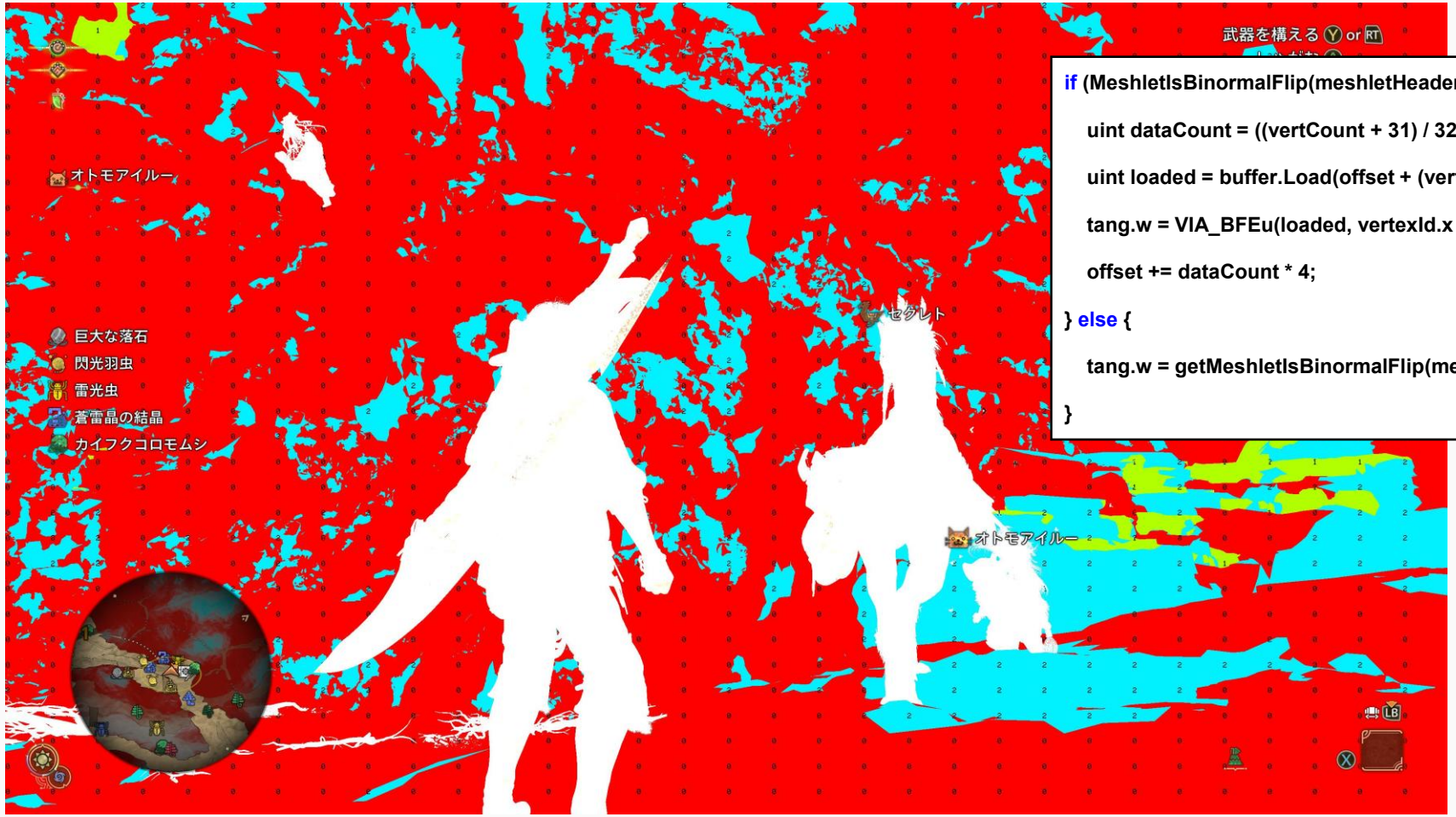

```
if (isMeshletUseVertexColor(meshletHeader)) {
  if (isMeshletCompressedVertexColor(meshletHeader)) {
    col = getMeshletVertexColor(buffer, offset, 0);
    offset += MeshletVertexColorSizeInBytes;
  }else {
    col = getMeshletVertexColor(buffer, offset, vertexIndex);
    offset += vertCount * MeshletVertexColorSizeInBytes;
  }
}
```

Flipping Binormals

Red:
Flip unneeded

Yellow:
Flip needed

Blue:
Mixed




```
if (MeshletIsBinormalFlip(meshletHeader) == 2) {
    uint dataCount = ((vertCount + 31) / 32);
    uint loaded = buffer.Load(offset + (vertexId.x / 32) * 4);
    tang.w = VIA_BFEu(loaded, vertexId.x & 31, 1) ? -1.0 : 1.0;
    offset += dataCount * 4;
} else {
    tang.w = getMeshletIsBinormalFlip(meshletHeader) ? -1.0 : 1.0;
}
```

Compression Effect on File Size

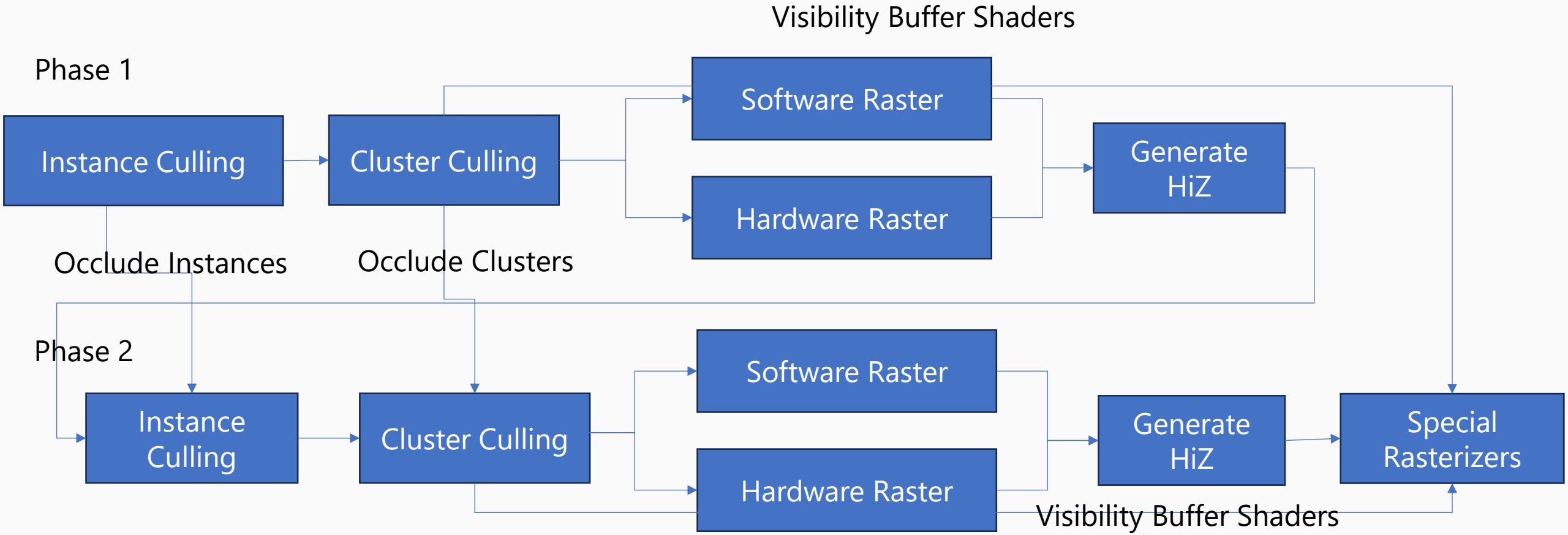
Real-life Result of Compression:

Reduction of 35MB of data.

Name	mesh	Meshlet	Meshlet + attribute compaction
RE ENGINE sample	75,791,760 bytes	56,275,516 bytes	40,309,144 bytes

Two-phase Occlusion Culling

Dragon Dogma 2's Visibility Buffer Rendering Pipeline



Instance Structure

Meshlet location information is 80-byte instance.

UserParam:

User-defined, referenced from shader.

PartsTableIndex:

Per-mesh 256-bit visibility flags.

Per-cluster visibility flags.

Large capacity per instance.

extendBindlessIndex:

Can access structures like SpeedTree.

Type	BindlessMeshInstance
Float3x4	World matrix
Int	Geometry index (bindless index)
Int	Material (byte offset)
Int	User Param
Int	Draw Flags
Int	Parts Table Index
int	Extend Bindless Index
Int2	reserved

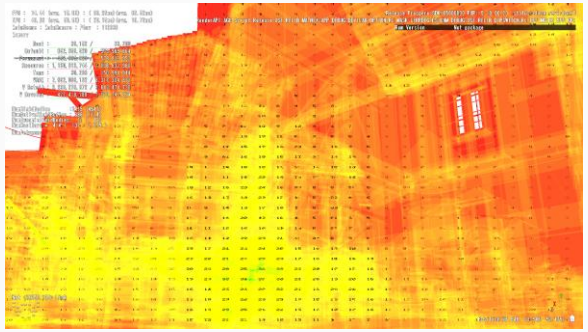
Culling

Instance Culling

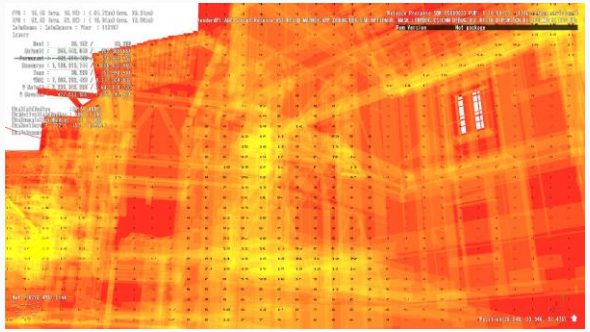
- Uses 32-bit draw flags to determine if target should be rendered.
- Viewport control, writing to cache when rending shadows, and more.

Instance/Cluster Culling target shape uses AABB

- View Frustum/Zero Area is handled mathematically.
- Runs HiZ culling if possible.
- If HiZ is a visibility buffer, sets LOD bias to 2 x 2, 4 x 4, 8 x 8.
- Increases culling test cost but can increase performance depending on meshlet footprint in screen-space.



2 x 2



8 x 8

A1

SL2

Distribution of GPU Processing Granularity for Cluster Culling

Meshlet amount changes depending on meshlet types and LODs.

- Meshlets made from many vertices generate many clusters.
- Single-thread and single-wave limits amount able to be processed.

VisibilityBuffer Pass	Cluster Culling
1.578 ms	0.543 ms

Tested on PlayStation5　　async compute off

Distribution of GPU Processing Granularity for Cluster Culling

Single instance divided into N waves and processed.

Implementation simple and can somewhat handle large amounts of clusters.

```
uint sharedInstance;
if (WaveIsFirstLane()){ globalAtomicInc(sharedInstance); }
shared_instance = WaveReadLaneFirst(sharedInstance);

uint div = INSTANCE_SUBDIV_COUNT; // 16 sub-division
uint targetIdx = sharedInstance / div;
while(targetIdx< getInstanceCount()){
    ....
    uint MeshletNum = getMeshletNum( getMeshlet(targetIdx), getLOD(targetIdx));
    ....
    uint subMeshletNum = MeshletNum / div;              // Sub-dividing
    uint index = sharedInstance & (div-1);              // Get subdivision index
    uint offset = subMeshletNum * index;                // Calculate offset using subdivision index
    if (index == div-1) subMeshletNum = MeshletNum - offset;    // Reassigning number of subdivision
    if (MeshletNum < WaveSize * div){                   // Optimizing for small number of clusters
        subMeshletNum = min((uint)max(int(MeshletNum) - int(index * WaveSize), 0), WaveSize);
        offset = WaveSize * index;
    }

    for (uint instanceID = WavePrefixCountBits(1); instanceID < subMeshletNum; instanceID += WaveSize){
        culling(targetIdx, offset + instanceID);
    }

    if (WaveIsFirstLane()){ globalAtomicInc(sharedInstance);}
    shared_instance = WaveReadLaneFirst(sharedInstance);
    targetIdx = sharedInstance / div;
}
```

Clusters	VisibilityBuffer Pass	Cluster Culling
1	1.578 ms	0.543 ms
16	1.159 ms	0.051 ms

Tested on PlayStation5 async compute off

Structure of Culling Results

64-bit Output Structure of Cluster Culling:

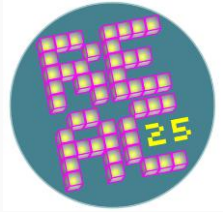
Type	CulledMeshletInfo (64-bit)
Uint 24-bit	Instance Index
Uint 8-bit	Fixed point float, LOD transition value
Uint	Meshlet Cluster Offset

CulledMeshletInfo:

Acquires material index of cluster from cluster offset.

Accesses bindless material structure from instance index.

Hardware Raster



GPU switches between mesh shaders and vertex shaders.
Depth test enabled, 64bitAtomic to PrimitiveID or output as R32G32Uint RenderTarget.

Signature	InstanceIndex	Triangle Index
1-bit (Always 0)	24-bit	7-bit

PrimitiveID

Triangle Index

7-bit due to maximum triangle output of 128.

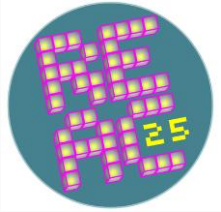
Instance Index

24-bit for accessing CulledMeshletInfo.

Signature

Signature can change size and used to switch visibility buffer types.
Meshlets are 1-bit value 0b, terrain is 2-bit value 10b.

Mesh Shader



Tested Amount of Output Vertices and Triangles

- **Low amount of vertices and triangles cause increased file size and cluster culling cost.**
- **Clustering performance rapidly degrades when looking at overall times.**

	v128/t128	v64/t126	v64/t84	v64/t64
Visibility pass	882 us	920 us	935 us	1002 us
Cluster Culling	168 us	208 us	229 us	300 us
HW Raster	158 us	150 us	141 us	139 us

Tested on GeForce RTX 4090

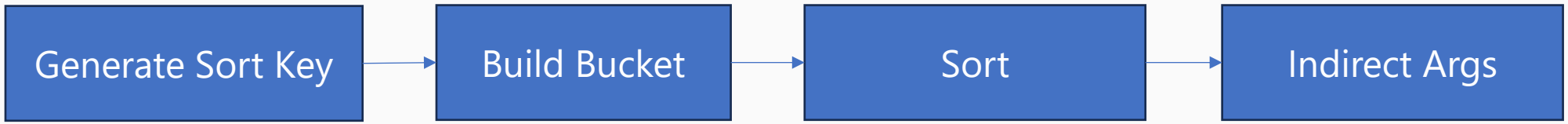
Not Using Triangle.

- **In testing it turns on and off, but clearly not used to its fullest.**

Output of primitive IDs use Primitive Attribute (DX12) and more.

Special Rasterizers

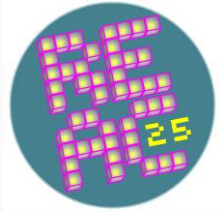
Uses information of primitive output from two-phase occlusion culling
AlphaTestGBuffer/GBufferDecal/TwoSideGBuffer
Sorts based on IDs then renders.



Generate Key	Bucket	Sort	indirectArgs
24 us	11.29 us	5.95 us	1.05 us

PlayStation 5

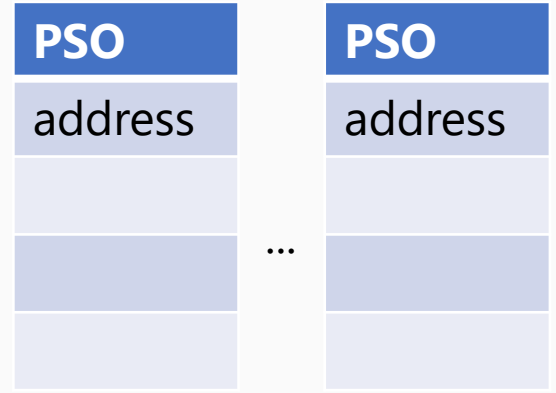
GenerateSortKey



Retrieve shader IDs from materials of clusters from cluster culling.

- Shader IDs as index for indirect table of shaders for each material.
- Index of sorted PSO used as key.
  - Shading Shader used for displaying materials.

Shader ID	Shading Shader	Visibility Buffer	Geometry Transform	Shadow Cast
0	0	0	NA	0
1	1	NA	NA	0
2	2	NA	0	2
3	0	1	NA	0



Bucket and Sort

Fast and parallel sort is highly desirable.

bitonicsort? radixsort?

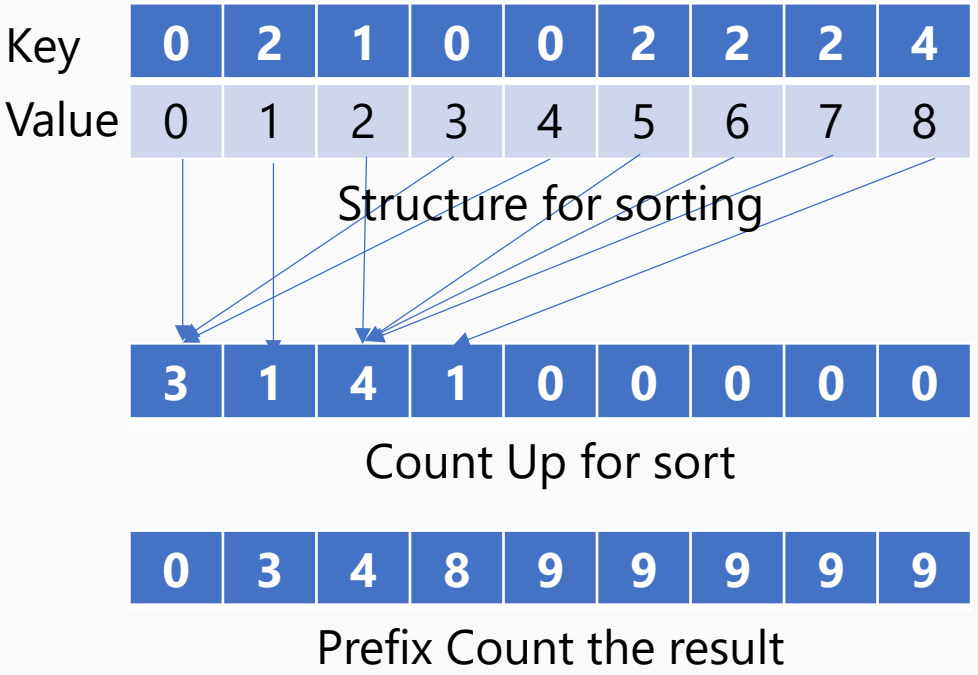
Uses a simpler method:

Restrict key to 11 bits.
Make bucket.
- Bucket is shader ID's key bit length.
- Prefix Count what is counted up.

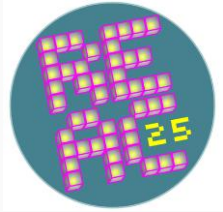
Prefix Count determined offset of write destination.

Sort

・Elements atomic add bucket Prefix Count.



Sort



Atomic add offset of key bucket and store data in return value.
- **Atomic cost high and value continuity for parallel processing not guaranteed.**
- **Solved using Compact Value if key is the same within a wave.**

Unoptimized Code

```
uint write_offset = 0;
RWBucket.InterlockedAdd( bucket_index * 4 , 1, write_offset);
RWSortedKey.Store( write_offset * 4, key);
RWSortedData.Store( write_offset * 4, data);
```

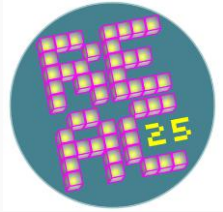
Data Count	Not Optimized	Optimized
59019	13.05 us	1.05 us
473397	177.99 us	28.42 us

Tested on PlayStation5

Optimized Code

```
uint write_offset = 0;
bool active = true;
[loop]
while (WaveActiveAnyTrue(active)) {
        if (!active) continue;
        [branch]
        if (WaveReadLaneFirst( bucket_index ) == bucket_index ){
                uint wave_local_count = WaveActiveCountBits(1);
                uint wave_local_offset = WavePrefixCountBits(1);
                if (wave_local_offset == 0)
                        RWBucket.InterlockedAdd( bucket_index * 4, wave_local_count, write_offset);
                write_offset = WaveReadLaneFirst(offset);
                write_offset+= wave_local_offset;
                active = false;
        }
}
RWSortedKey.Store( write_offset * 4, key);
RWSortedData.Store( write_offset * 4, data);
```

IndirectArgs



Created using sort results.

Already evident from bucket sort result.

Software Rasterizer

Software Rasterizer announced to be 3 times faster.

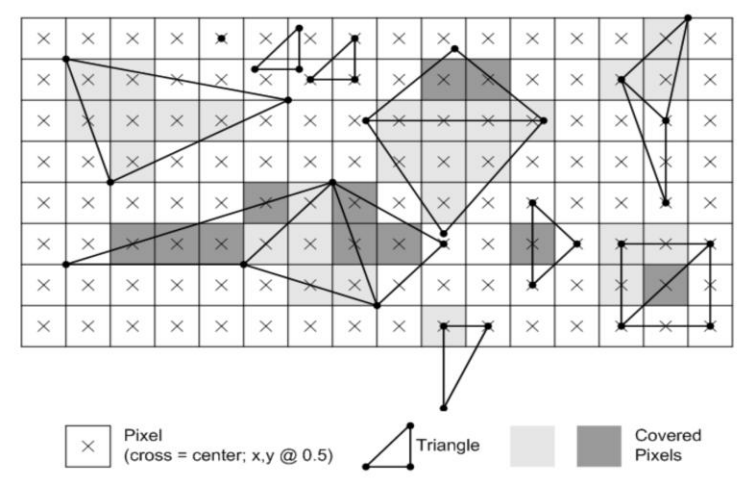
We tested ComputeShader on older hardware, results 2.4 times faster.

Scanline version was fast, but not used due to artifacting.

Software Rasterizer requires same behavior as Hardware Rasterizer.

Fulfills DirectX rasterization rules test.

Visibility buffer output as 64-bit Atomic, shadow map output as 32-bit Atomic.



[Rasterization Rules - Win32 apps | Microsoft Learn]

Sort GBuffer of Visibility Buffer

Sort after normal meshes of GBuffer rendered.

- If visibility buffer depth and GBuffer depth same, convert shader ID to 16-bit and output.
  - Create 32-bit mask for subsequent rendering.
    - Covert SV_Position to bits mapped in 8 x 4 render areas.
    - Use WaveActiveBitOr in wave, representative output as InterlockedOr.
  - Limit area the PixelShader can run.

Game Screen

Visualization of results separated by shaders

Writing to GBuffer in Visibility Buffer

Render area with DrawInstanced

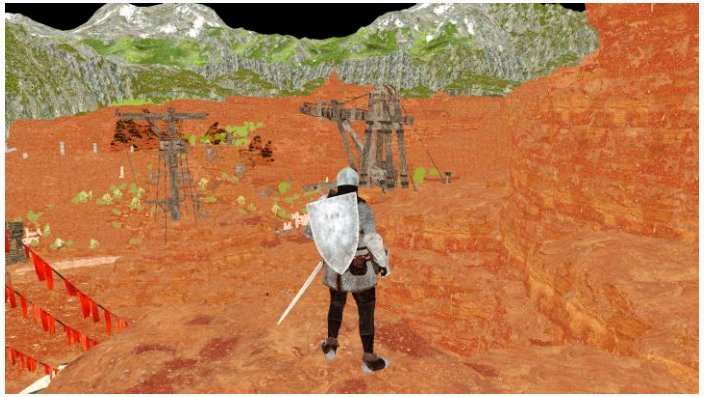
- Read internal VS 32-bit data, set unneeded to NaN and mask.

Recover structures needed for shading with PixelShader

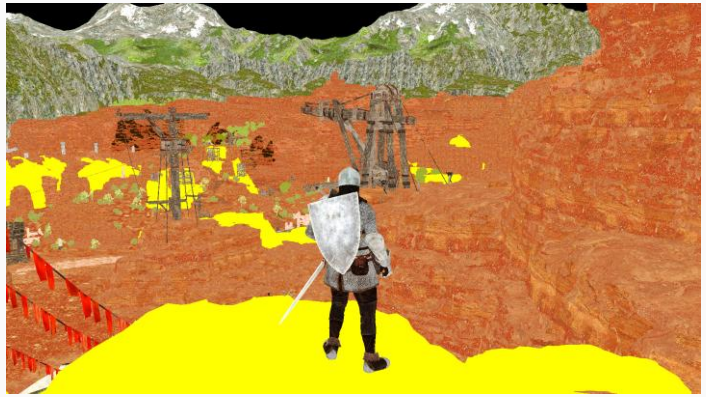
- Restore material paraments/textures from bindless structures with StructuredBuffer or ConstantBufferView.
- Restore vertex structures using CulledInstanceInfo from primitive IDs of visibility buffer.

Shader Initialization Amount

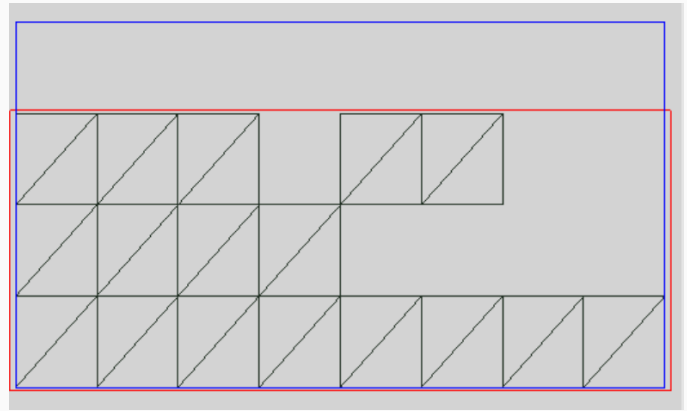
- Dragon's Dogma 2 depends on the scene, GBuffer rendered with about 50 DrawInstanced.
- Monster Hunter Wilds executes about 60 GBuffer of visibility buffer.



GBuffer

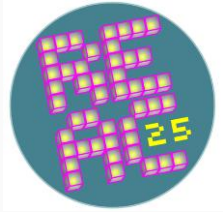


A shader's effect area in yellow



Vertex shader output

Derivatives of Visibility Buffer GBuffer Shaders



RE ENGINE Uses Shader Graphs

- Cases where UVs not modified are automatically resolved. A1
- UV scale, rotation, translation have dedicated texture sampler node.

Complicated and Specialized Code

- Each title handles code written by the game team.
  - Triplanars and heightmaps particularly difficult for them to handle. A2
  - Represented using SampleLevel.

Special Rasterizers

Use Special Rasterizers to recreate partial decals.

Maintains compatibility with traditional mesh representations.

Dragon's Dogma 2 Results

Development build, meshlet rendering pipeline on/off performance test.

Meshlet Rendering	Rendering Commands	Rendering CPU Time	GPU Time	Video Memory
Disable	19510	5.502 ms	27.05 ms	6.821 GB
Enable	8172	3.891 ms	19.90 ms	5.960 GB

Tested on PlayStation 5

A1

Monster Hunter Wilds

Handling more complicated scenes.

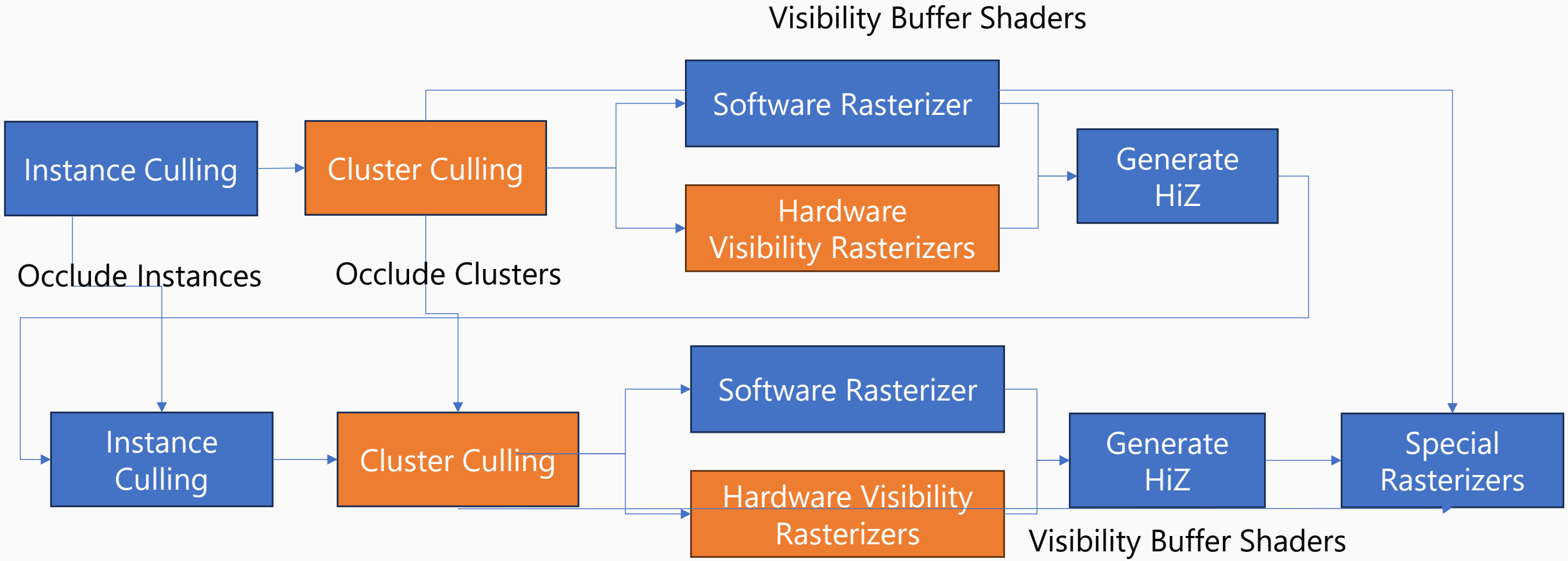
Almost everything rendered uses visibility buffer.

Game Screen

Meshlet + Visibility Buffer Rendering Disabled

Two-phase Occlusion Culling

Visibility Buffer Rendering Pipeline of Monster Hunter Wilds



Monster Hunter Wilds

Handling Visibility Buffers for Alpha Testing and Vertex Transform

- Added support for two-phase occlusion culling.
- Added support for SpeedTree version 8 middleware.

A1

Everything Rendered

Meshlet Special Rasterizers Disabled

SL2

Hardware Visibility Rasters

Merge alpha testing and vertex transforms into Visibility Buffer.

DepthTest + RTV R32G32Uint

Use same sort algorithm as Special Rasterizers.

Key = Shader ID: 8 || Depth: 3

More occlusion culling with improved system.

	Visibility Pass	G Buffer Pass	Total
Dragon's Dogma 2	1.705 ms	7.977 ms	9.632 ms
Monster Hunter Wilds	3.127 ms	5.937 ms	9.064 ms

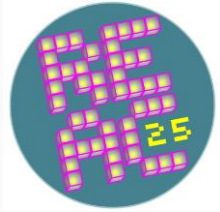
Async Compute Disabled @ PlayStation5 1664p

Improved occlusion culling

Hardware Raster occlusion culling before improvement

Game screen of created occluders

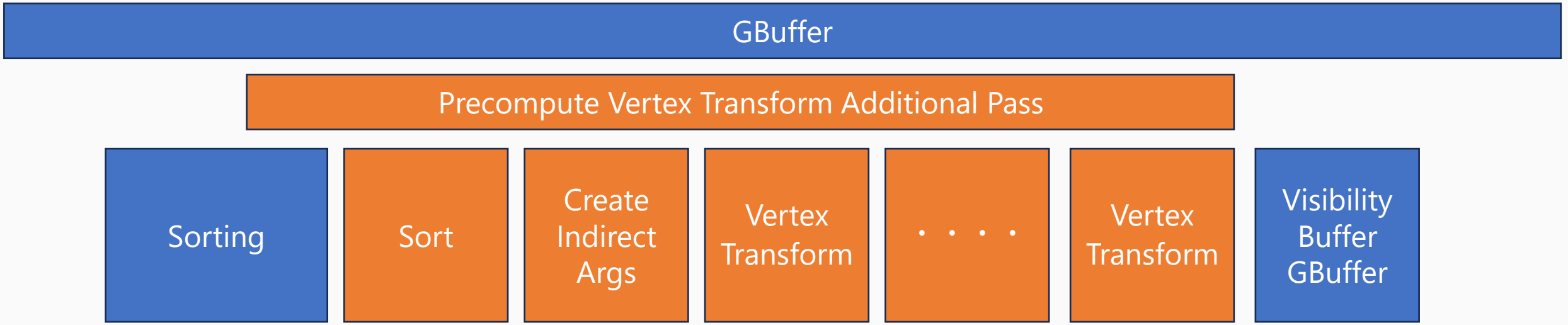
Pre-compute Vertex Transforms



Tried visibility buffer without separate vertex transform pass.

3-vertex transform in GBuffer costly.

Before rendering GBuffer, vertex transformations calculated using separate ComputeShader.



Preparing the Precompute Vertex Transform

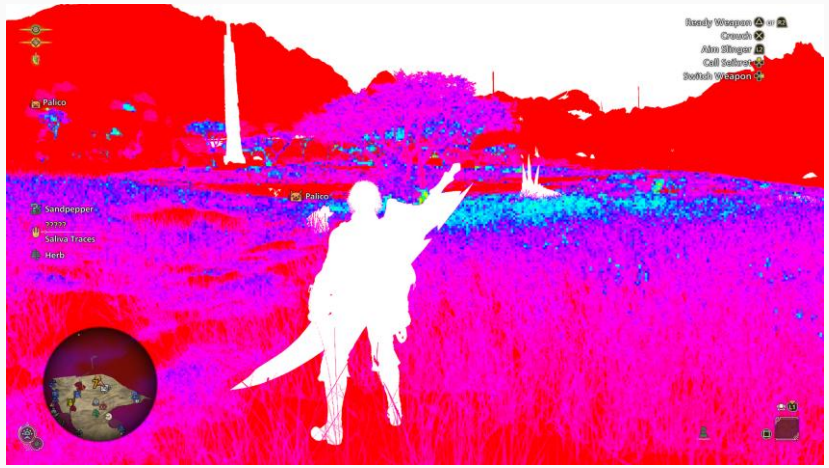
Visibility Buffer GBuffer sorting runs at same time as pixel shaders.

Output requires vertex transform x 3 vertex calculation resources.

If PrimitiveID same within Wave, make Compact to reduce computations.

Console executed with wave64, PC wave amount depends on Hardware/Driver.

Game screen

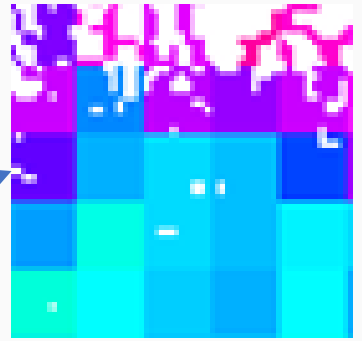


Allocation count

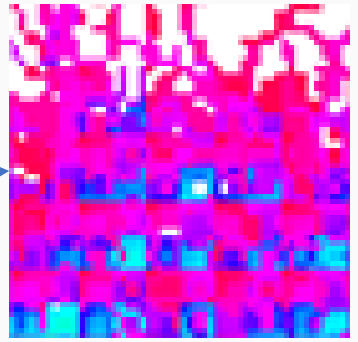
Allocation offset

Compaction of Wave Vertex Transforms

```
uint2 WaveCompactBufferWrite(RWByteAddressBuffer Counter,uint CounterOffset, RWStructuredBuffer<uint> WriteBuffer,uint WriteValue) {
    bool active = true, owner = false;
    uint count = 0, offset = 0;
    while (WaveActiveAnyTrue(active)) {
        count++; //uniform count
        if (!active) continue;
        if (WaveReadFirstLane(writeValue) == writeValue) {
            owner = WavePrefixCountBits( true ) == 0;
            offset = count - 1;  //offset count.
            active = false;  //in-active
        }
    };
    uint allocate_offset = 0;
    if (WaveIsFirstLane()) // compact write
        Counter.InterlockedAdd(0, CounterOffset, allocate_offset);
    offset += WaveReadFirstLane( allocate_offset) ;
    if (owner)
        WriteBuffer[offset] = WriteValue;
    return offset;
}
```

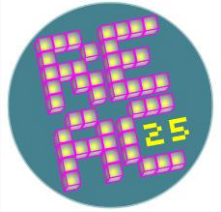


Wave Output Sum



Returned offset

Post-processing of Vertex Transform Primitive IDs



Does not use wider compaction.

64-threads waves on consoles.

Sort and IndirectArgs creation method same as Special Rasterizers.

Compute Shaders of Vertex Transform

Splits 1 primitive into 3 vertices and create 3 threads.

Outputs current frame HPOS, last frame HPOS, normal, and optional tangent.

Precompute Vertex Transform Storage Format

Screen-base vertex transforms use lots of memory.

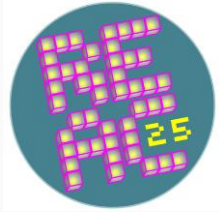
- Render resolution * 3 vertices * vertex size
- Memory outside execution interval auto retrieved due to set as alias memory.

Monster Hunter Wilds:

- 16 bytes for each vertex.

name	format
HPOS(xyw)	16-bit(half) + 16-bit(half) + 32-bit(float)
Previous HPOS(xyw) – HPOS(xyw)	10-bit(half) + 10-bit(half) + 12-bit(half)
Normal	10-bit(snorm) + 10-bit(snorm) + 10-bit(snorm) + 2(unused)

Precompute Vertex Transform Performance



Fast at sorting, slow at vertex transforms.
Faster than directly calculating with GBuffer.

Consoles perform async precompute vertex transform before rendering meshes.
Before sorting, threads increased as there are no impedances from other meshes.
This works great in Monster Hunter Wilds.

Generate Key	Bucket	Sort	indirectArgs
48 us	12.28 us	28.42 us	1.8 us

Vertex Transform
487.03 us

Tested on PlayStation 5

Improving Cluster Culling

Distant HLODs had many clusters, while foliage had only few.
- Performance could not be maintained with a single distribution method.
- Distribution of fixed granularity was inadequate.

Implemented new system that can distribute jobs amongst waves.
- Ring Buffer structure using AtomicMin and InterlockedCompareExchange.

Scene 1

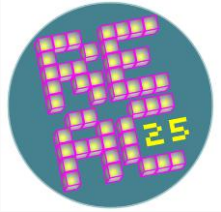
Scene 2

	Scene 1	Scene 2
Ring buffer	0.4 ms	0.241 ms
1 instance 16 waves	0.945 ms	0.197 ms
1 instance 128 waves	4.42142 ms	0.456 ms

Phase 1 cluster culling process time

PlayStation 5 1664p

GPU Ring Buffer



Overview

JOB_EMPTY defined as 0. Can input jobs that are 32-bit and not a value of 0.

If unable to read/write, waits to be handled at next loop timing or by another wave.

Will continue until all jobs completed.

```
bool JobScattering(uint job){
    if (job != JOB_EMPTY){
        uint counts = WaveActiveCountBits(1);
        uint writeoffset = 0;
        if (WaveIsFirstLane()){
            RingBuffer.InterlockedAdd(JOB_RING_TAIL,counts,writeoffset);
        }
        writeoffset = WaveReadLaneFirst(writeoffset) + WavePrefixCountBits(1);
        writeoffset&= WORK_BUFFER_SIZE_IN_DWORD-1;
        uint ret;
        RingBuffer.InterlockedCompareExchange(writeoffset*4,JOB_EMPTY,job,ret);
        if (ret == JOB_EMPTY){
            uint counts = WaveActiveCountBits(1);
            if (WaveIsFirstLane())
                RingBuffer.InterlockedAdd(JOB_COUNT,counts);
            return true;
        }
    }
    return false;
}
```

```
bool jobGathering(out uint job){
    uint counts = WaveActiveCountBits(1);
    uint readoffset = 0;
    if (WaveIsFirstLane()){
        RingBuffer.InterlockedAdd(JOB_RING_HEAD,counts,readoffset);
    }
    readoffset = WaveReadLaneFirst(readoffset) + WavePrefixCountBits(1);
    readoffset&= WORK_BUFFER_SIZE_IN_DWORD-1;
    RingBuffer.InterlockedMin(readoffset*4,JOB_EMPTY, job);
    if (job != JOB_EMPTY){
        int counts = (int)WaveActiveCountBits(1);
        if (WaveIsFirstLane())
            RingBuffer.InterlockedAdd(JOB_COUNT,-counts);
        return true;
    }
    return false;
}
```

Monster Hunter Wilds Results

Meshlet rendering pipeline on/off performance comparison in dev build.

Meshlet Rendering	Rendering Commands	Rendering CPU Time	GPU Time	Video Memory
Disable	8589	8.981ms	34.212 ms	5.290 GB
Enable	6940	5.228ms	27.605 ms	5.177 GB

Tested on PlayStation 5

A1

Shadow Casting

RE ENGINE formerly used precompressed shadow maps for directional shadows.

Dragon's Dogma 2 and Monster Hunter Wilds have a dynamic 24-hour day/night cycle.

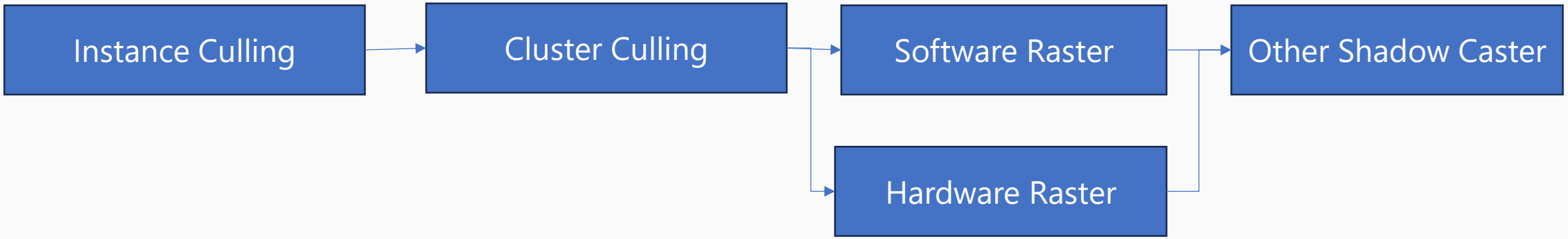
Shadow Casting

Shadow Caster for meshlets implemented as single-pass culling.

Each culling stage uses frustum culling and zero-area-size culling.

Functions for static structures affected by shadow caches like spotlights.

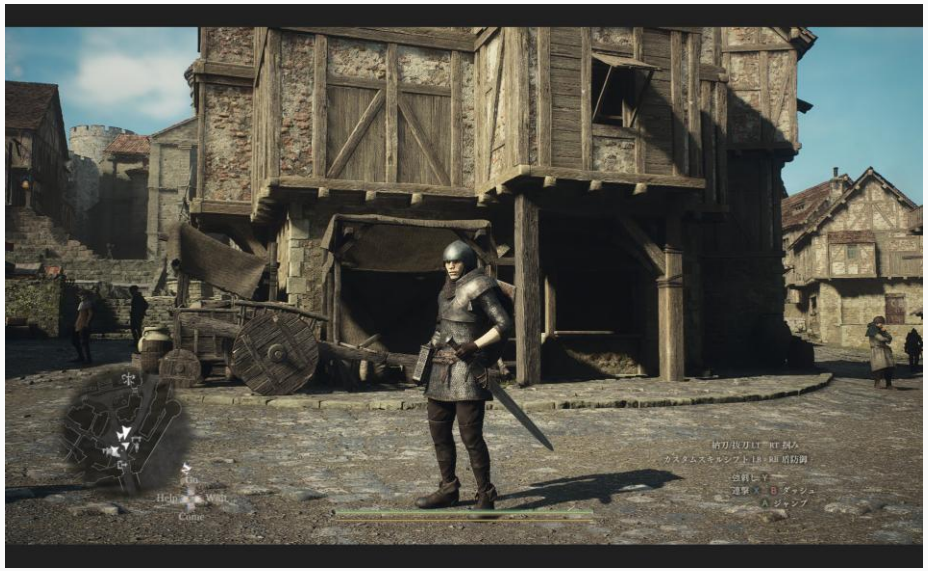
Inadequate performance for fully dynamic lights like Directional Light.



Generate Shadow Map Occluders from Depth

Uses camera depth reprojection from GPU-driven rendering.

Projects depth buffer onto shadow map to generate occluders.

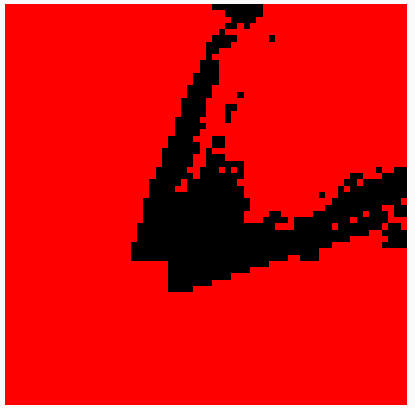


Shadow Occluder

64 x 64 occlusion culling structure with mipmaps from depth buffer.
Red indicates foreground objects, anything behind it likely occluded.

Depth Buffer

Cascade Occluder Map 0

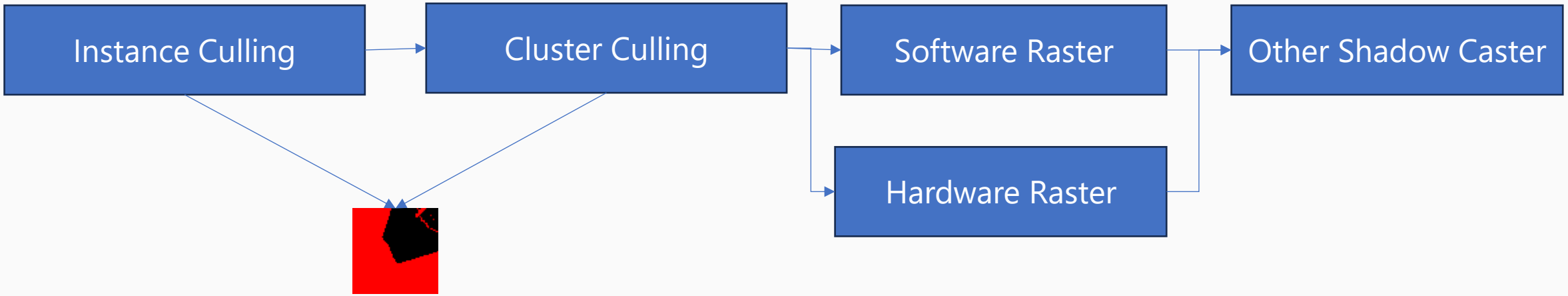


Cascade Occluder Map 1

Cascade Occluder Map 2

Shadow Casting

Only directional lights use occlusion culling.



Shadow Map

Shadow Maps Using Occluders:

Depth Buffer

Cascade Shadow Map 0

Cascade Shadow Map 1

Cascade Shadow Map 2

Issue with Shadow Maps Outside Field of Vision

Combined with ray tracing, indoor areas became brighter.

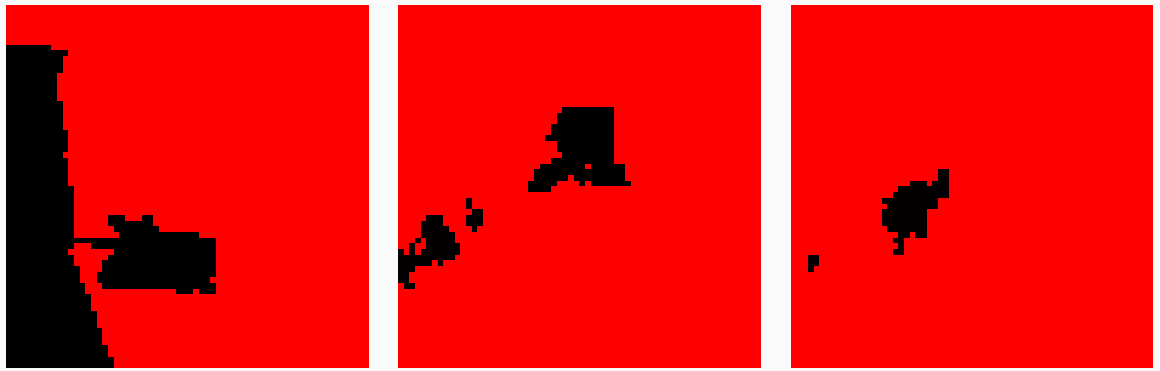
Dragon's Dogma 2 uses raytracing for global illumination.

GI lighting reuses shadow map for shadows.
Unable to represent shadows outside field of vision.

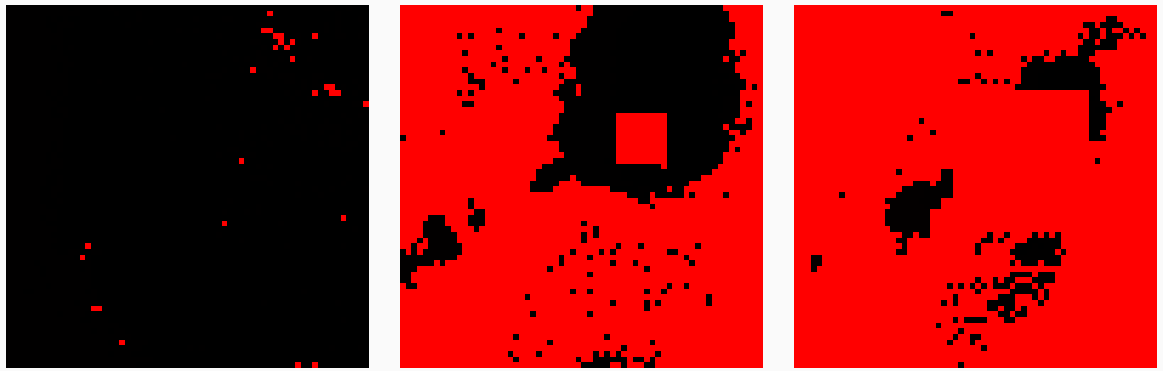
Expanding Shadow Maps of Shadow Casts

Add previous frame's ray hit positions to render target of shadow map.

By inserting ray hit positions, visible area (black) expands.



Depth Reprojection Occlusion Map



Ray Hit Position Injected Occlusion Map

Result After Change

Natural lighting after changes.

Depth Reprojection

Depth Reprojection + Ray Hit Position

Results of Shadow Casts

3 Cascade Shadow with AsyncCompute

	Primitives Input PA	Time
Frustum Culling	13.54M Tris	6.71 ms
Depth Reprojection	7.93M Tris	4.83 ms

Processing time for shadow cast of one frame.

	Software Raster	Hardware Raster
Frustum Culling	0.381 ms	0.824 ms
Depth Reprojection	0.094 ms	0.304 ms

Meshlet processing time for Cascade Shadow Map 2.

Tested on PlayStation 5

Deep World Shadows

Monster Hunter Wilds features vast vertical spaces.
Unsuitable for camera-based depth processes.
Render all mesh shadows above.

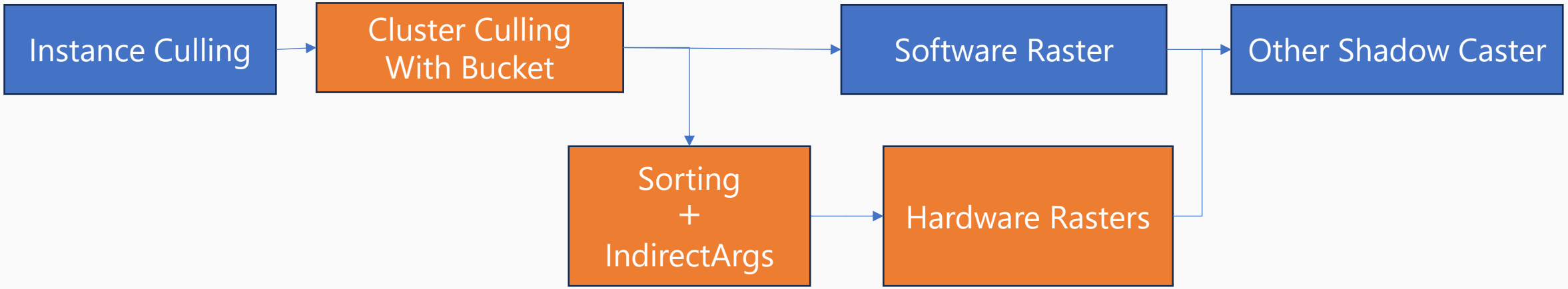
Severe geometry over-drawing.

Shadow Cast Update

Support vertex transformation and alpha testing.

Create buckets for sorting during cluster culling.

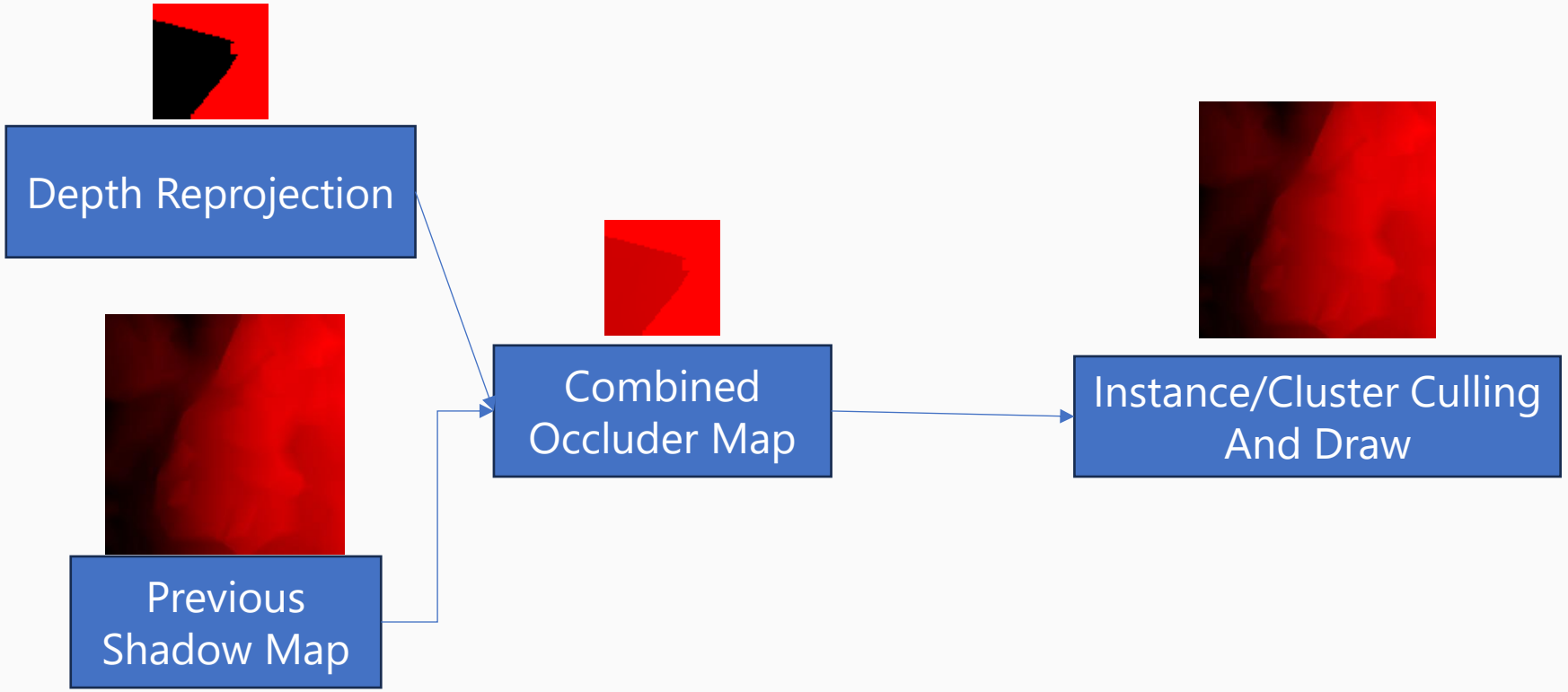
Maximum of 256 shader entries, return Prefix Count after culling.



Handling Geometry Over-draw

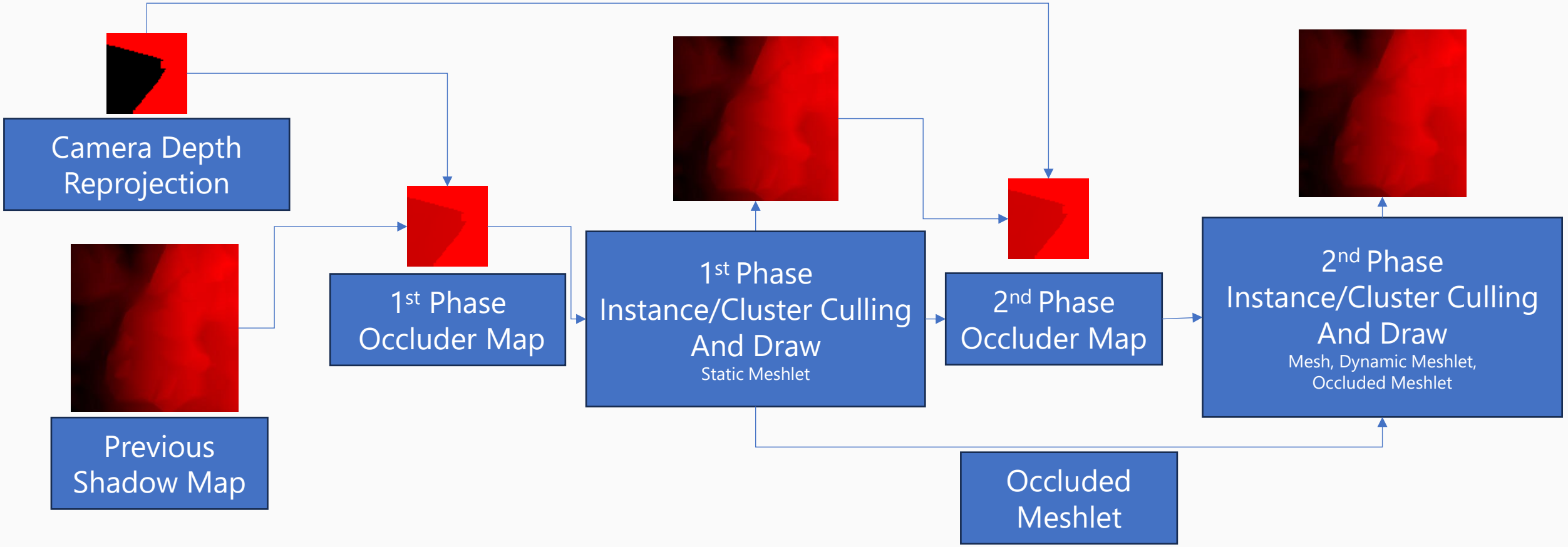
Use previous frame's shadows as occluder?

- **Reuse with creates artifacts and limited to single pass.**
- **However, static camera resulted in high execution speed.**



Two-phase Occlusion Culling

Change to two-phase occlusion culling for directional lights.



Shadow Cast Time

3 cascade shadow maps and 1 spotlight shadow map:

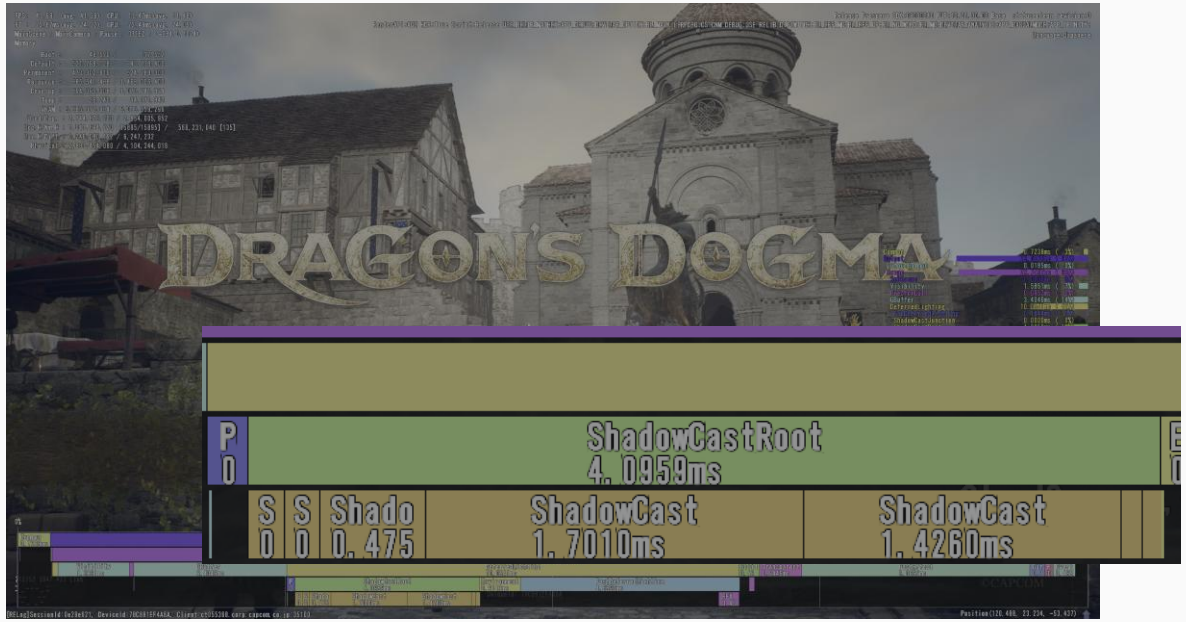
	Primitives Input PA	Processing Time
Depth Reprojection	10.65M Tris	3.19 ms
DR + SMI (artifact)	2.71M Tris	1.91 ms
DR+SMI + TPOC	3.30M Tris	2.28 ms

Tested on PlayStation 5

Optimizations for Shadow Casters in Dragon's Dogma 2

RE ENGINE maintains compatibility with all engine versions.

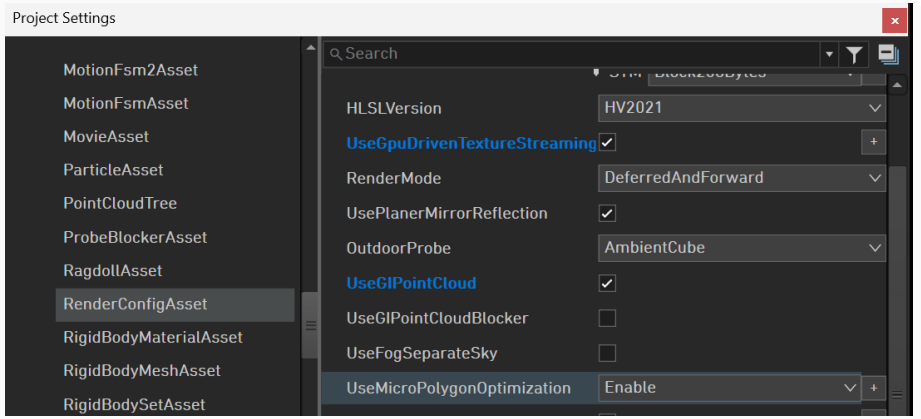
Optimization Features ON



Optimization Features OFF

Summary

Meshlets
Two-phase Occlusion Culling
Visibility Buffer
Software Rasterizer



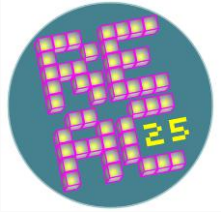
Improving performance and memory usage.

Features per-mesh or system-wide flags

that can be toggled easily.

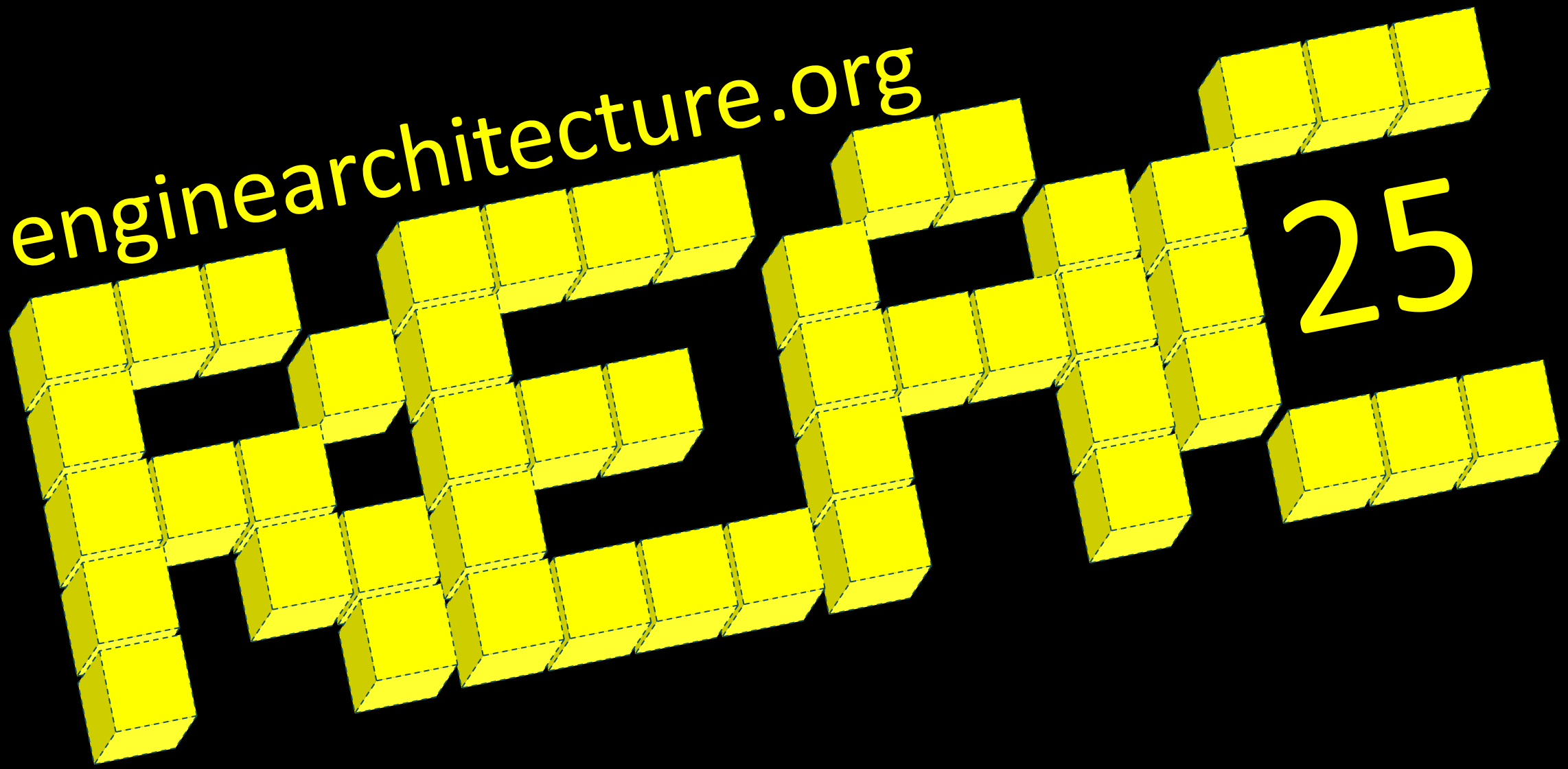
A1

Next Steps

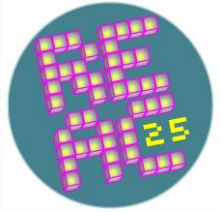


Generate automatic seamless LODs to reduce workload.
Support subdivision surfaces and tessellation to improve product quality.
Implement automatic solutions for shader derivative issues.

Further investigate possible optimizations.


enginearchitecture.org
25

References



- A Deep Dive into Nanite Virtualized Geometry
- GPU-Driven Rendering Pipelines
- Adventures with Deferred Texturing in Horizon Forbidden West
- Visibility Buffer Rendering with Material Graphs
- Improved Culling for Tiled and Clustered Rendering in Call of Duty: Infinite Warfare
- DirectX 12 Optimization Techniques on Biohazard RE:2 and Devil May Cry 5
- 最新タイトルのグラフィックス最適化事例 [Examples of Graphics Optimizations in the Latest Titles], CEDEC 2018