Grappling With Performance:
Rendering Optimization Strategies In Rumbleverse

Jon Moore
Graphics Engineer
Iron Galaxy Studios

Hello everyone, I’m Jon Moore, I work as a Graphics Engineer at Iron Galaxy Studios, and I’d like to welcome you to my talk “Grappling With Performance: Rendering Optimization Strategies in Rumbleverse”

Rumbleverse

- 40-Player Battle Royale
- Melee Combat
- 1 km x 1 km Island Arena
- Performance target: 1080p 60 FPS on PS4
- Shipped on Unreal 4.27.1

If you're not familiar with the game, Rumblerse is a 40 player battle royale game featuring primarily melee combat on a 1km squared island arena.

Our gold standard performance target for the game is 1080p + 60 FPS on a base PS4 – with the game shipping across both generations of Sony and Microsoft consoles.

And an important detail to note – we are an Unreal licensee, and the game shipped using Unreal version 4.27.1, but using an engine that is not developed internally at IGS doesn't discourage me from finding plenty of optimization opportunities for the game, which is the motivation for this talk today

Optimization Philosophy

- Choice of Engine doesn't affect optimization potential – your content creates opportunities and exposes limitations of the engine
- The fastest wavefront is the one you never launch
- Everything worth doing in life is at least an 0.1 ms speedup

Before getting into specifics, I think I should preface with my personal philosophy on optimization, especially as it relates to work done on the GPU. These are ideas that formed before I started working on Rumbleverse, but my experiences here have only further solidified these beliefs.

First and foremost, I think there is optimization potential anytime an engine is not built from scratch for a particular game. It is always useful to reflect on how your specific content might allow the engine to be modified/configured to run it most efficiently. This is very true with a widely licensed engine like Unreal, but I've seen this with shared engine tech used internally between teams, or even just when building off the engine used for a previous game on a new project that is not a direct sequel.

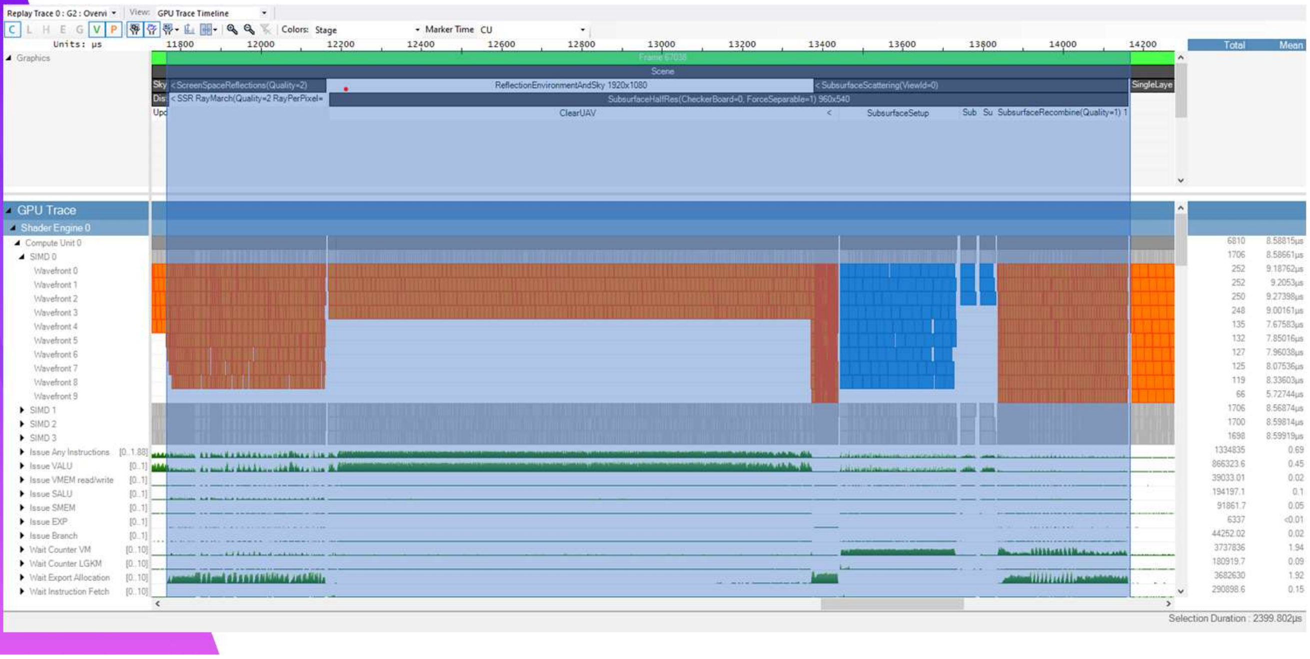
Secondly, I have over time come to appreciate that culling out redundant/unneeded work on the GPU is something that can be gone back to again and again when looking for gains. In the simplest sense, it's making sure that every wavefront running on the GPU is actually contributing to the final image.

Finally, whenever I am working on a 60 hz video game, I generally use 0.1 ms as my measuring stick for if an optimization is worth doing. Less than that, and I will sit on a change as not being worth the risk of modifying the engine. 0.1 ms is often my sweetspot of feeling like a change is worthwhile and enough 0.1 ms changes added together will eventually make a big impact.

Part 1:
Proactive Optimizations

From day 1 of my time on Rumbleverse – rendering was my primary focus, and I wanted to make an impact on the game to ensure we could push our look to the best it could be. This was in service of allowing artists to build richer environments, have more detailed materials, and avoid dynamic resolution drops as much as possible. This first section is focused on optimizations made proactively in pursuit of this goal as we set-up our initial configuration of rendering features across our target platforms.

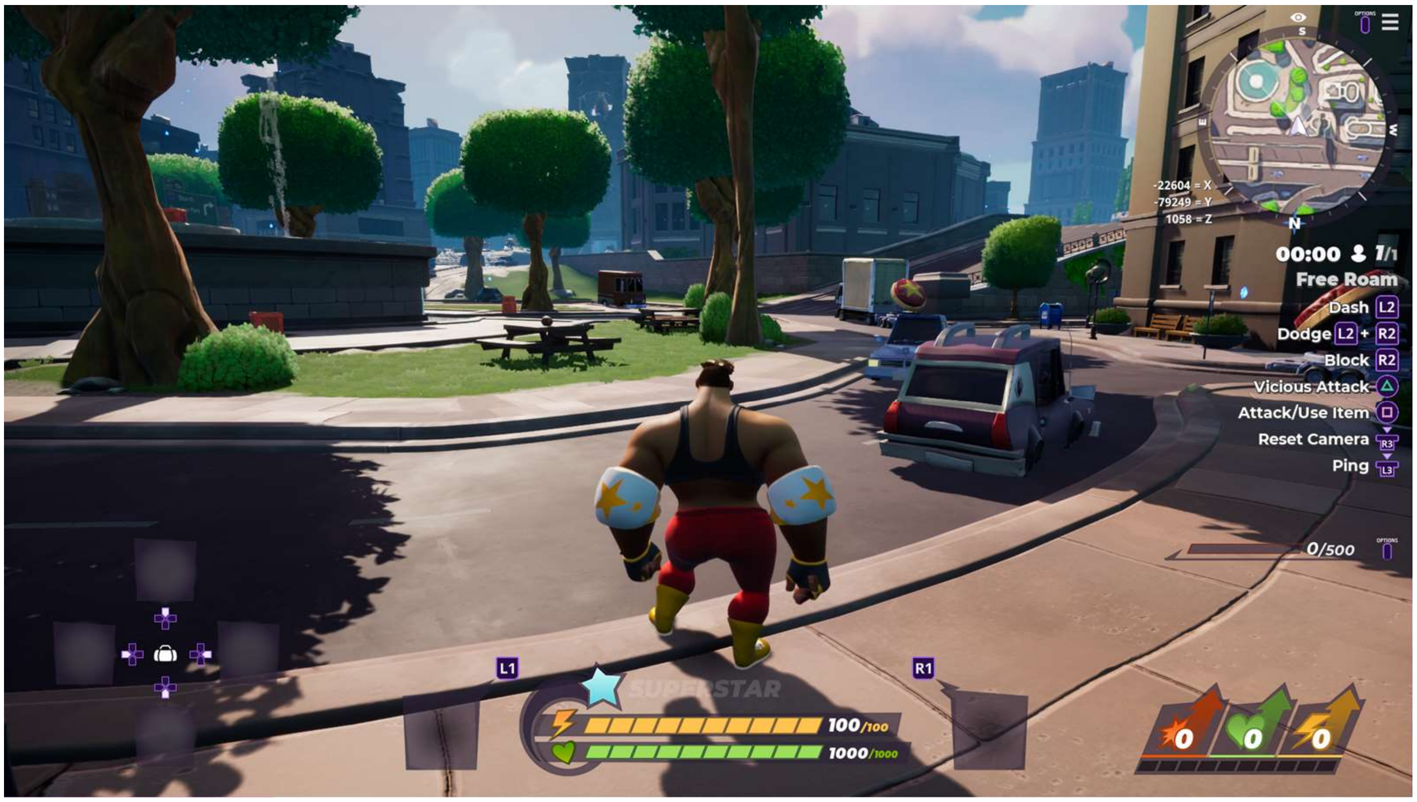
Reflections + Subsurface


Replay Trace 0 : G2 : Overvi
View: GPU Trace Timeline
Colors: Stage
Marker Time CU
Units: µs
11800
12000
12200
12400
12600
12800
13000
13200
13400
13600
13800
14000
14200
Graphics
Frame 67038
Scene
Sky
< ScreenSpaceReflections(Quality=2)
ReflectionEnvironmentAndSky 1920x1080
< SubsurfaceScattering(ViewId=0)
SingleLaye
Dis
< SSR RayMarch(Quality=2 RayPerPixel=
SubsurfaceHalfRes(CheckerBoard=0, ForceSeparable=1) 960x540
Upd
ClearUAV
<
SubsurfaceSetup
Sub
Su
SubsurfaceRecombine(Quality=1) 1
Total
Mean
GPU Trace
Shader Engine 0

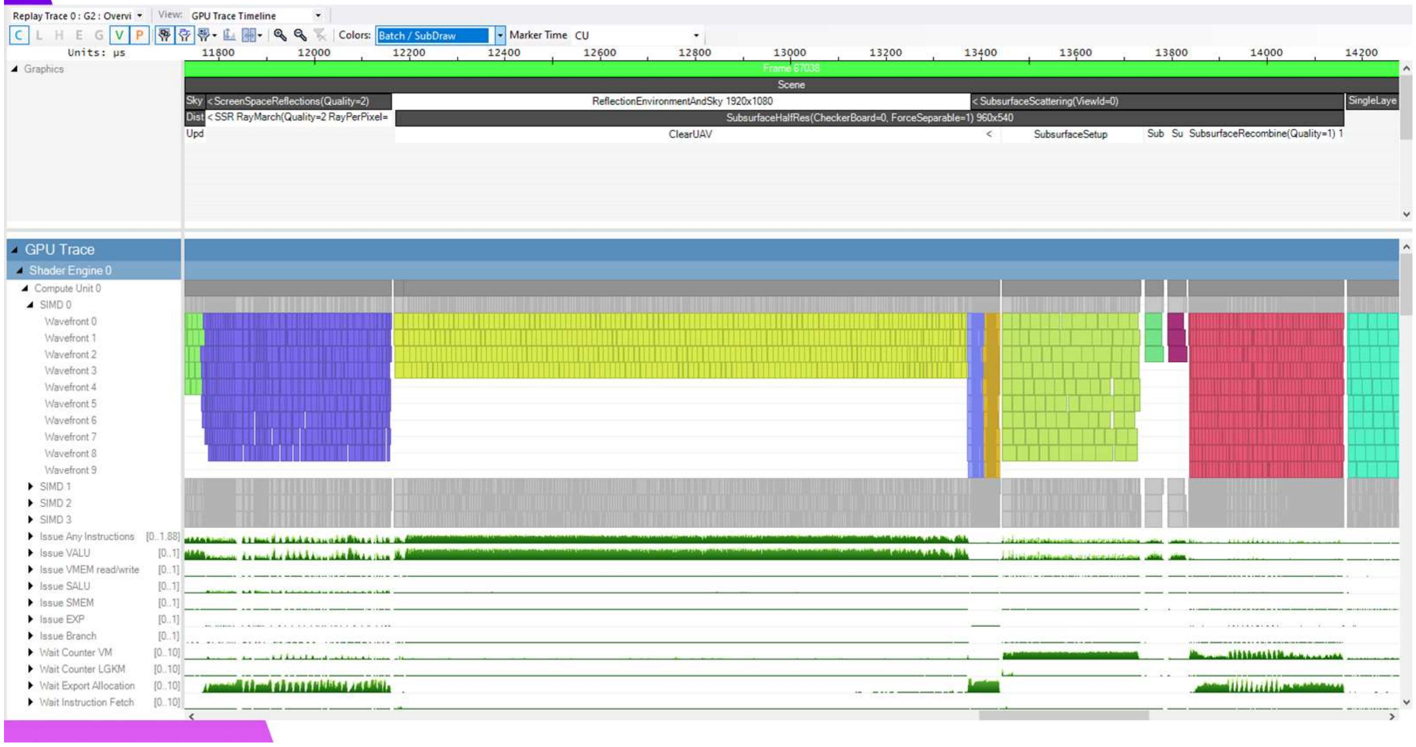
		Total	Mean
Compute Unit 0		6810	8.58815µs
SIMD 0		1706	8.58661µs
Wavefront 0		252	9.18762µs
Wavefront 1		252	9.2053µs
Wavefront 2		250	9.27398µs
Wavefront 3		248	9.00161µs
Wavefront 4		135	7.67583µs
Wavefront 5		132	7.85016µs
Wavefront 6		127	7.96038µs
Wavefront 7		125	8.07536µs
Wavefront 8		119	8.33603µs
Wavefront 9		66	5.72744µs
SIMD 1		1706	8.56874µs
SIMD 2		1700	8.59814µs
SIMD 3		1698	8.59919µs
Issue Any Instructions	[0..1.88]	1334835	0.69
Issue VALU	[0..1]	866323.6	0.45
Issue VMEM read/write	[0..1]	39033.01	0.02
Issue SALU	[0..1]	194197.1	0.1
Issue SMEM	[0..1]	91861.7	0.05
Issue EXP	[0..1]	6337	<0.01
Issue Branch	[0..1]	44252.02	0.02
Wait Counter VM	[0..10]	3737836	1.94
Wait Counter LGKM	[0..10]	180919.7	0.09
Wait Export Allocation	[0..10]	3682630	1.92
Wait Instruction Fetch	[0..10]	290898.6	0.15

Selection Duration : 2399.802µs

Let’s start by looking at the 2.4 ms stretch of frame time in Razor on PS4 that I found myself looking at early on in development, where reflections and subsurface scattering are handled in stock UE4. All parts of this time were decided to be pretty important to the look of our game, and I wanted to try to reduce its cost as that is 14% of a 16.6 ms frame time.



Here is the scene in question that the trace is from. I'll revisit this in a bit, but it's pretty standard for us: we have some foliage, a character, some sky, some shiny metallic objects, and some reflective windows.


Replay Trace 0 : G2 : Overvi

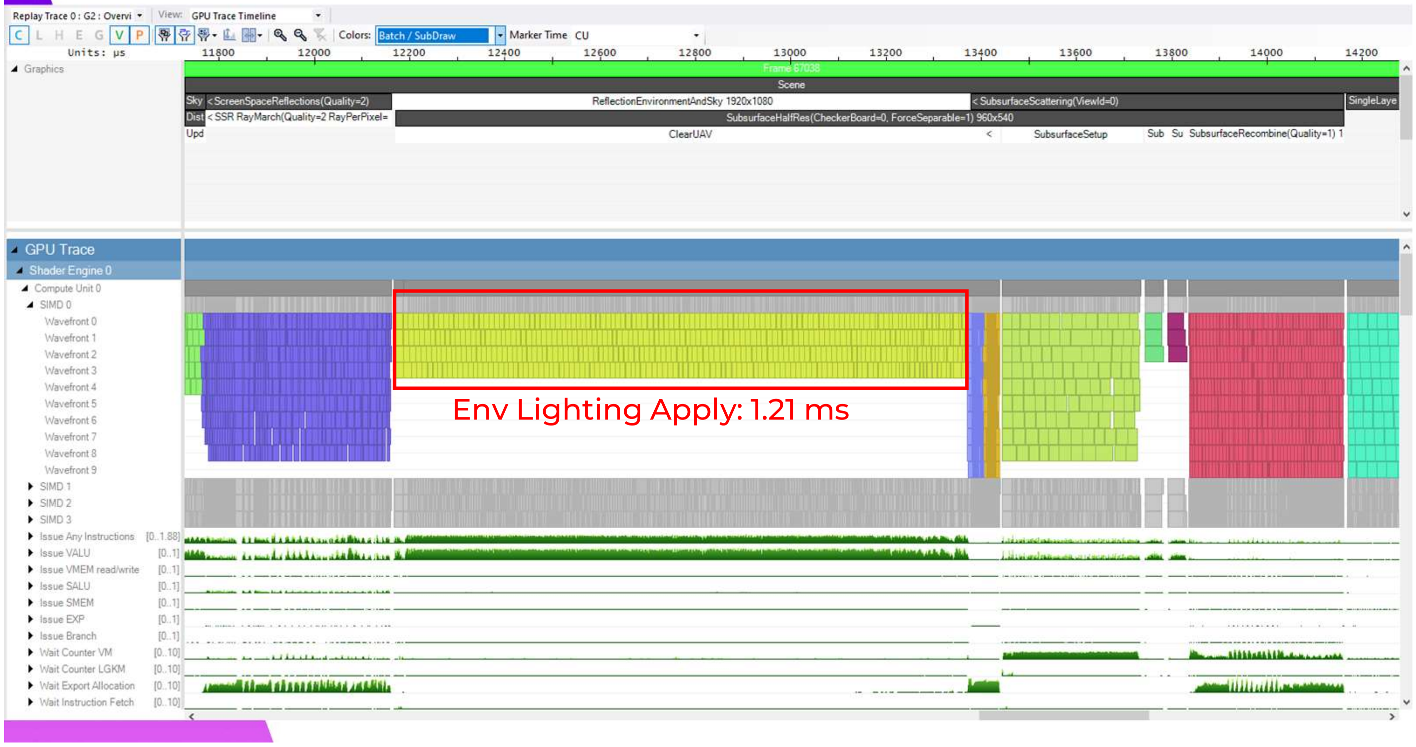
View: GPU Trace Timeline

Colors: Batch / SubDraw

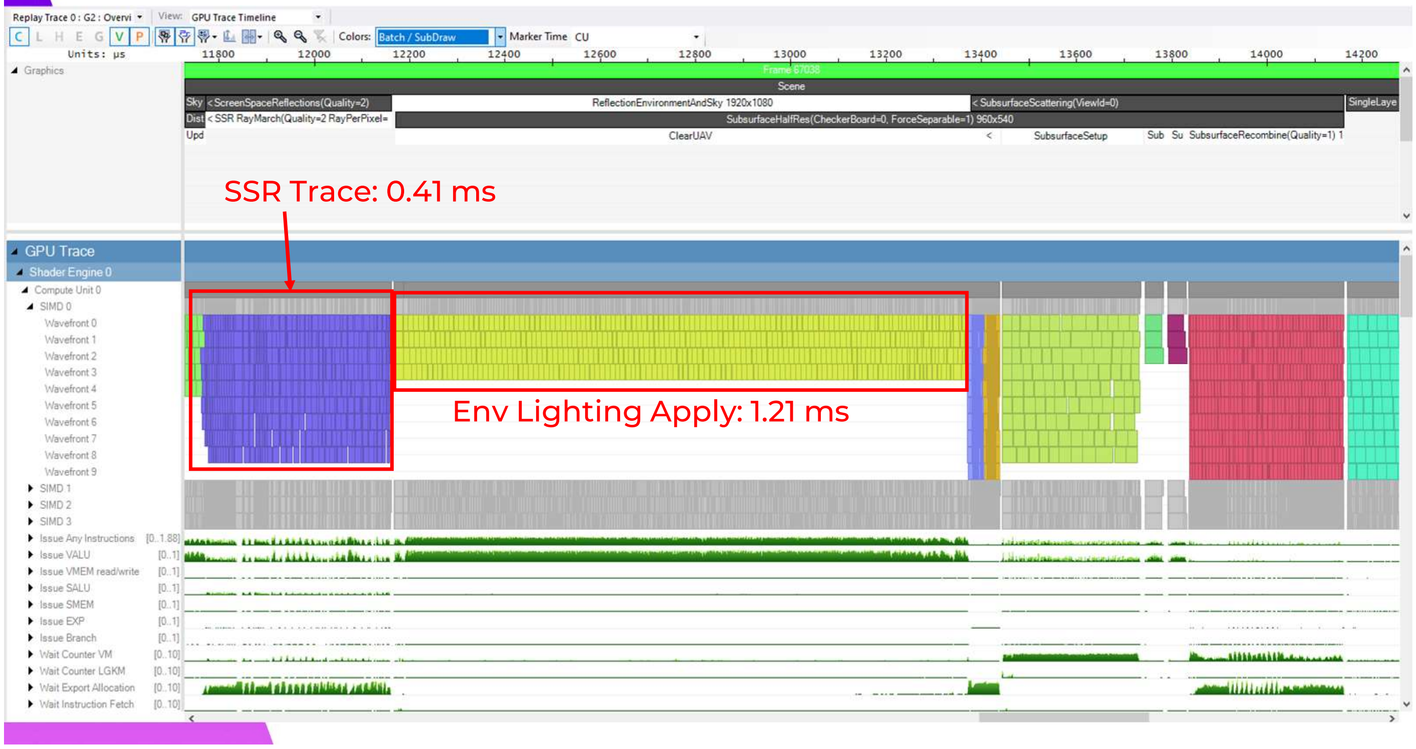
Marker Time CU

Units: µs

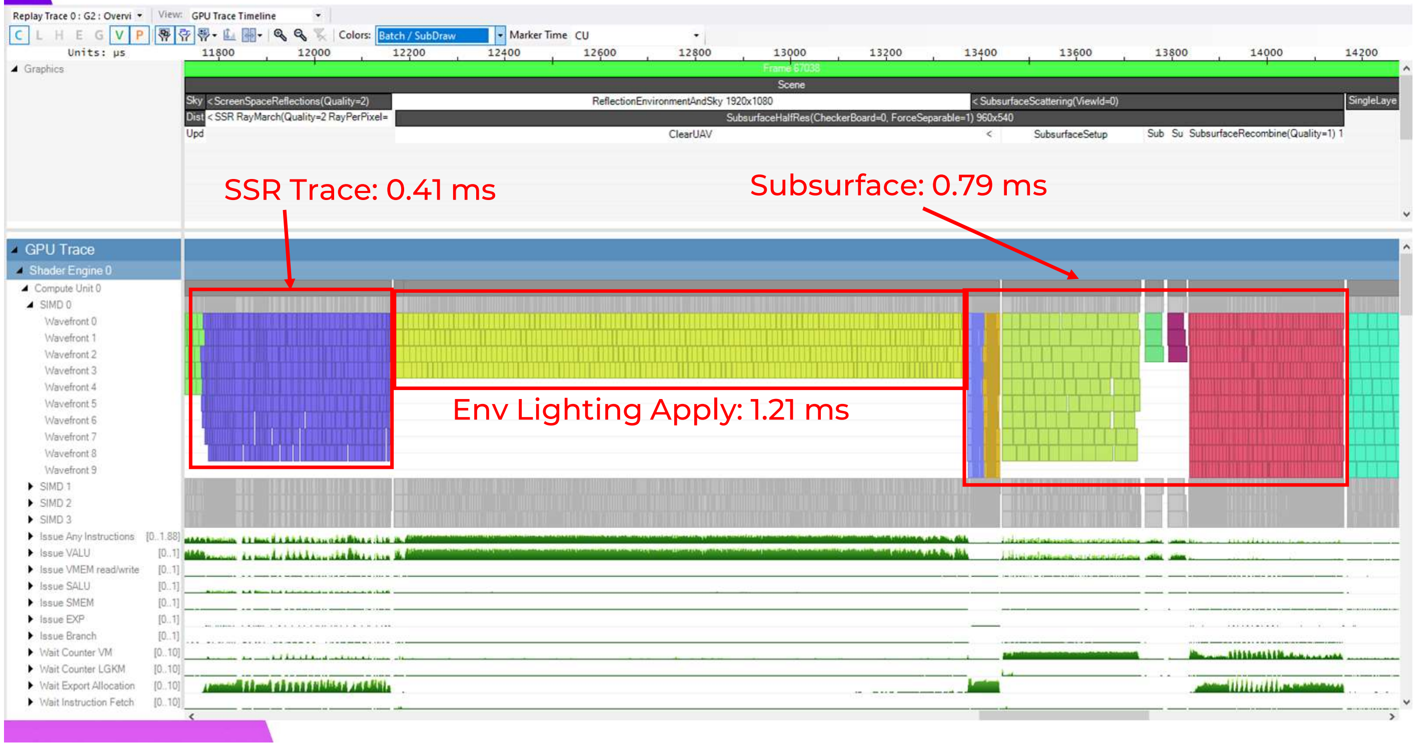
Let’s go back to that stretch of time in Razor with the wavefronts colored by batch.



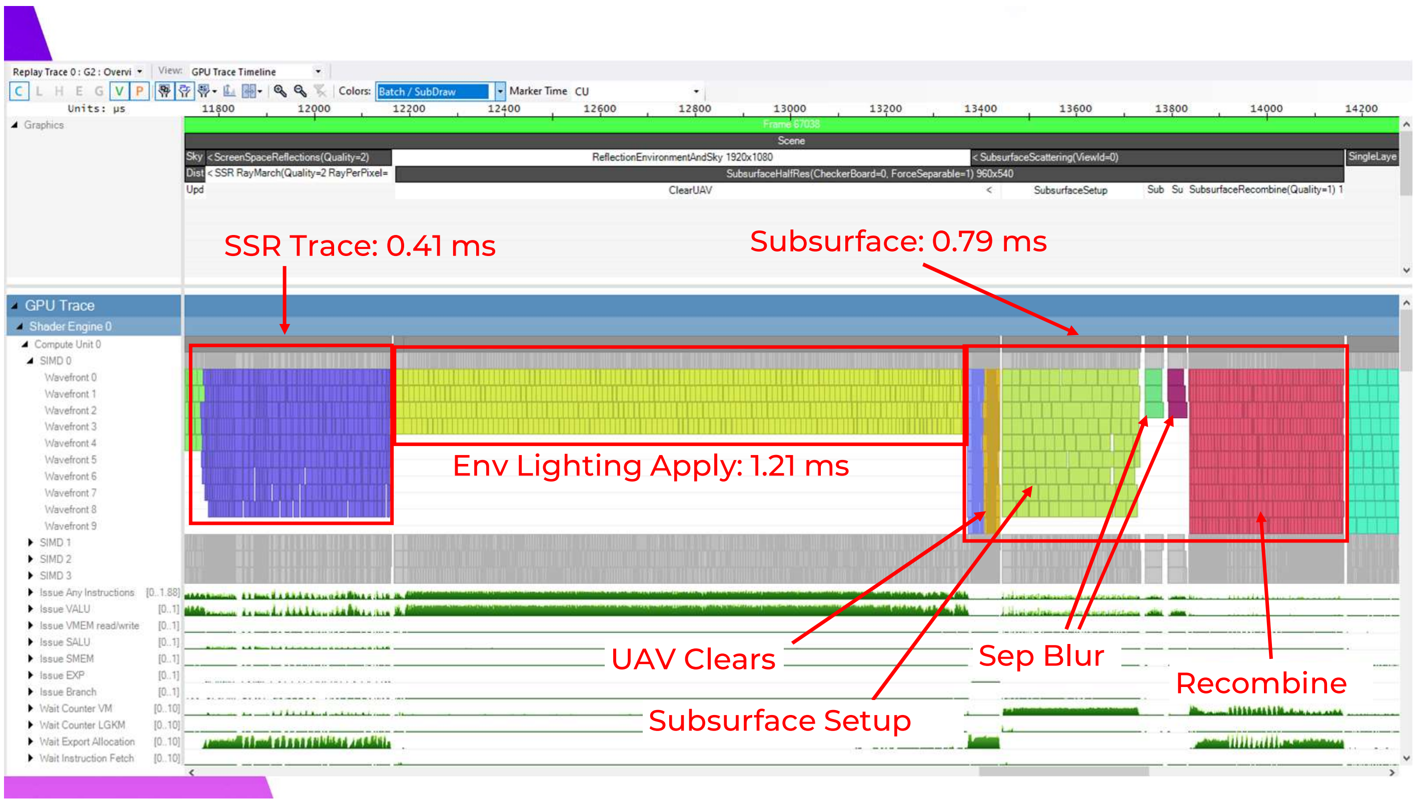
This long section in the middle is the Env Lighting Apply pixel shader, clocking in at 1.21 ms. This is where indirect diffuse, specular, and skylighting is combined together after the direct lighting is handled by the deferred lighting loop.


SSR Trace: 0.41 ms
Env Lighting Apply: 1.21 ms
GPU Trace
Shader Engine 0
Compute Unit 0
SIMD 0
ScreenSpaceReflections(Quality=2)
ReflectionEnvironmentAndSky 1920x1080
SubsurfaceScattering(ViewId=0)
SSR RayMarch(Quality=2 RayPerPixel=
SubsurfaceHalfRes(CheckerBoard=0, ForceSeparable=1) 960x540
ClearUAV
SubsurfaceSetup
SubsurfaceRecombine(Quality=1) 1

Before that, Screen Space Reflection tracing takes 4 tenths of a ms



And this time after Env Lighting Apply is 0.79 ms on subsurface scattering


SSR Trace: 0.41 ms
Subsurface: 0.79 ms
Env Lighting Apply: 1.21 ms
UAV Clears
Sep Blur
Recombine
Subsurface Setup
Sky <ScreenSpaceReflections(Quality=2)
ReflectionEnvironmentAndSky 1920x1080
<SubsurfaceScattering(ViewId=0)
SubsurfaceHalfRes(CheckerBoard=0, ForceSeparable=1) 960x540
ClearUAV
SubsurfaceSetup
SubsurfaceRecombine(Quality=1)
GPU Trace
Shader Engine 0

If you're not familiar, SSS in this version of Unreal clears some UAVs, does SubsurfaceSetup which does tile classification and downsamples the subsurface to half res, and then dispatches compute shader batches for tiles based on the subsurface algorithm selected by tile classification.

And then finally there is a recombine with the Scene Color after the blur. Classifying by algorithm was added to Unreal after our character look was developed, so our tiles all fall into the path using Screen Space Separable Subsurface – based on the technique developed by Jorge Jimenez

Separable Subsurface Scattering and Eye Rendering by Jorge Jimenez: http://advances.realtimerendering.com/s2012/activision/Jimenez-Separable_Subsurface_Scattering_and_Eye_Rendering(Siggraph2012).pptx

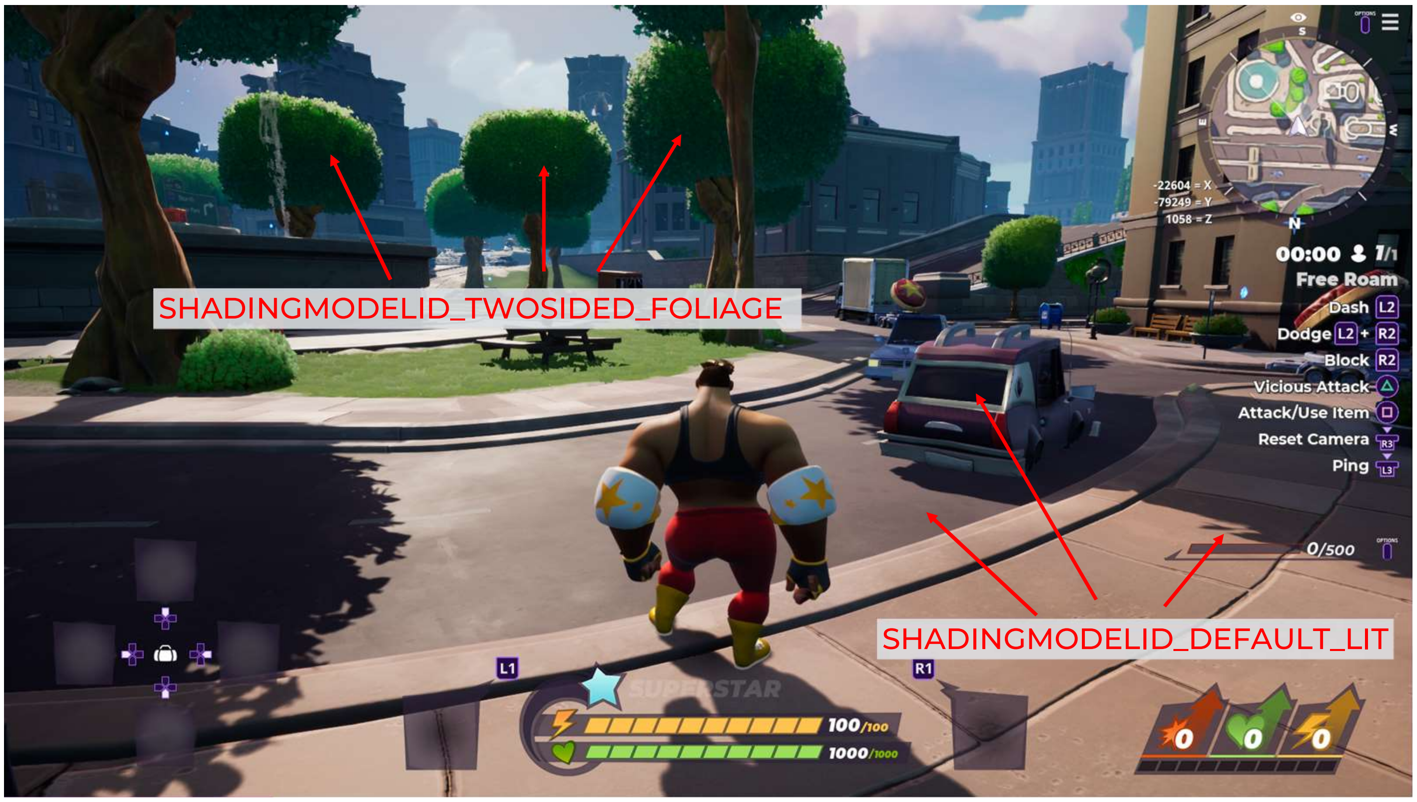
Tile Classification

- Original Idea: improve occupancy in reflection apply by using tile classification
- Inspired by Ramy El Garawany's presentation: *Deferred Lighting in Uncharted 4*
- Algorithm:
  1. analyze GBuffer
  2. build lists of tiles based on material properties
  3. render each using different shader permutations + DispatchIndirect

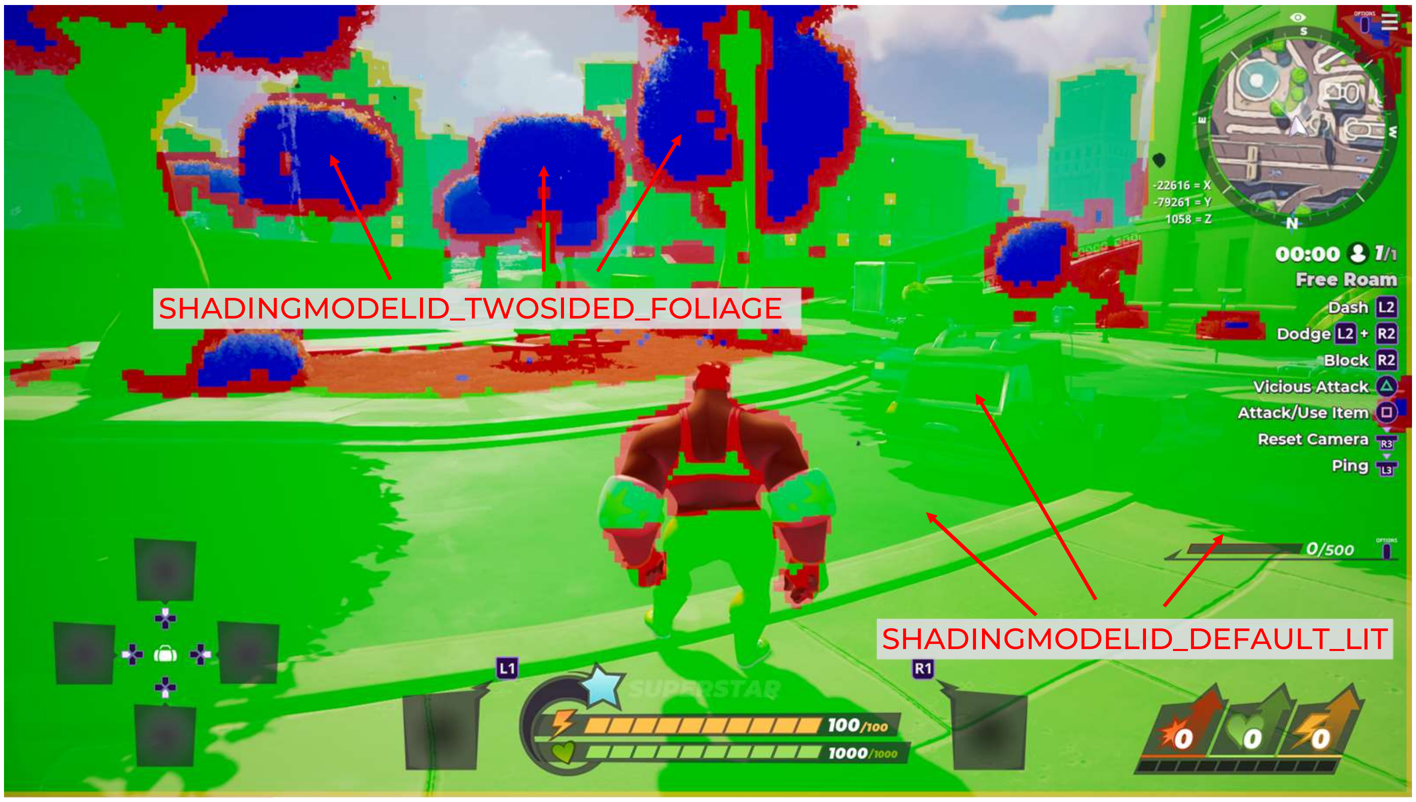
Mentioning tile classification on SSS is an interesting launching point to how I approached improving these systems. Naughty Dog first presented a version of the tile classification technique for reducing per-tile cost of their lighting in the talk *Deferred Lighting in Uncharted 4* from Siggraph 2016.

I thought that the heavy time spent on reflection apply could be optimized in a similar way here. I would write a tile classification shader that looked at the Gbuffer properties of an 8x8 group of pixels and build lists based on the materials present. Then each list can be rendered using DispatchIndirect with the different shader permutations bound to each dispatch.

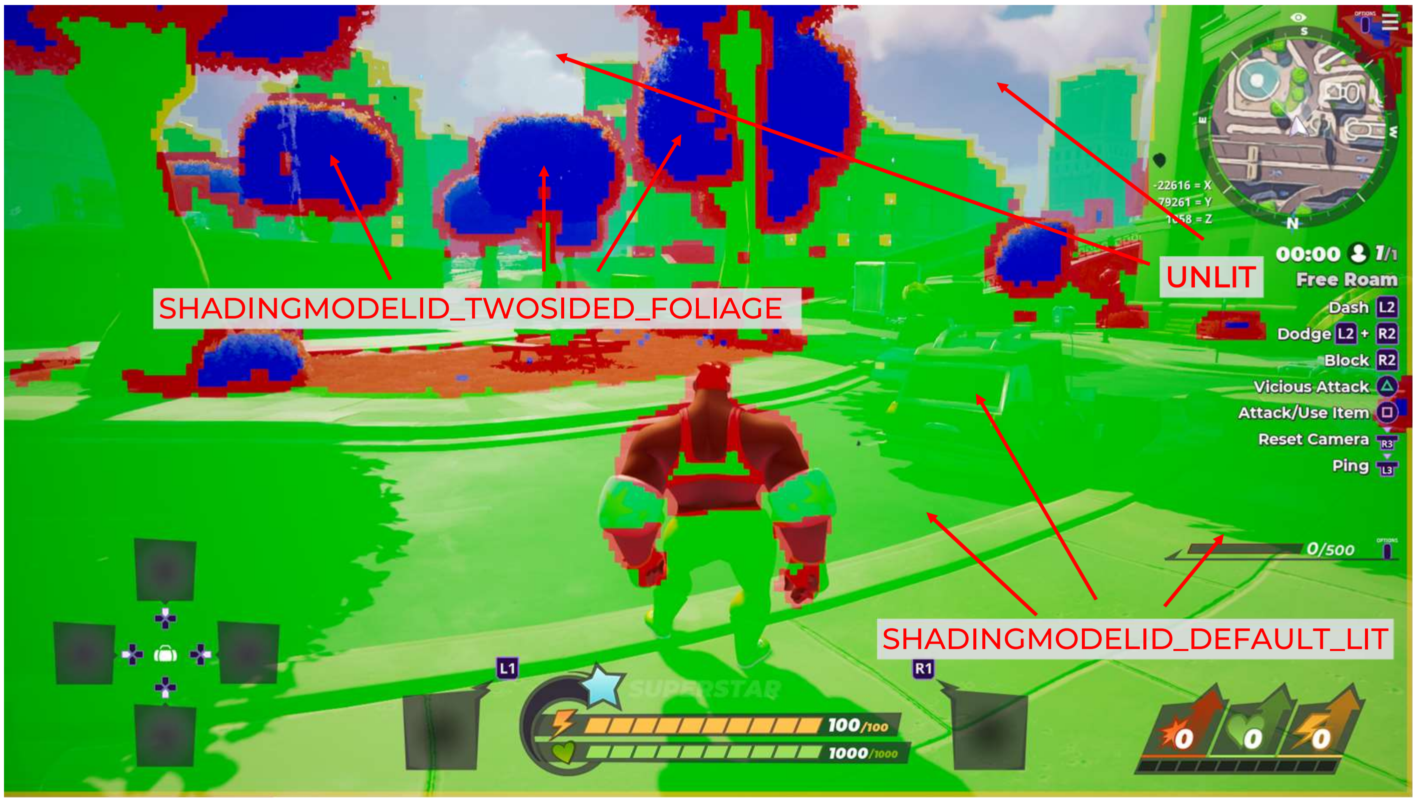
Deferred Lighting in Uncharted 4 by Ramy El Garawany:
http://advances.realtimerendering.com/s2016/s16_ramy_final.pptx



So for example in this frame we can pretty clearly see groupings of pixels that are all default lit or twosided foliage.



And here you can see a visualization of the tile classification in practice. The green tiles are default lit, the blue is all foliage, and the red contains a shading paths that are "complex" and run the full shader



However, the most important thing this work took me to was the unlit tiles here, tiles that are culled entirely had the biggest impact on the running time. This got me thinking about how the tile classification could be used for culling workload from SSR + SSS as well, and that cost of running the classification would be shared across multiple steps of our rendering.

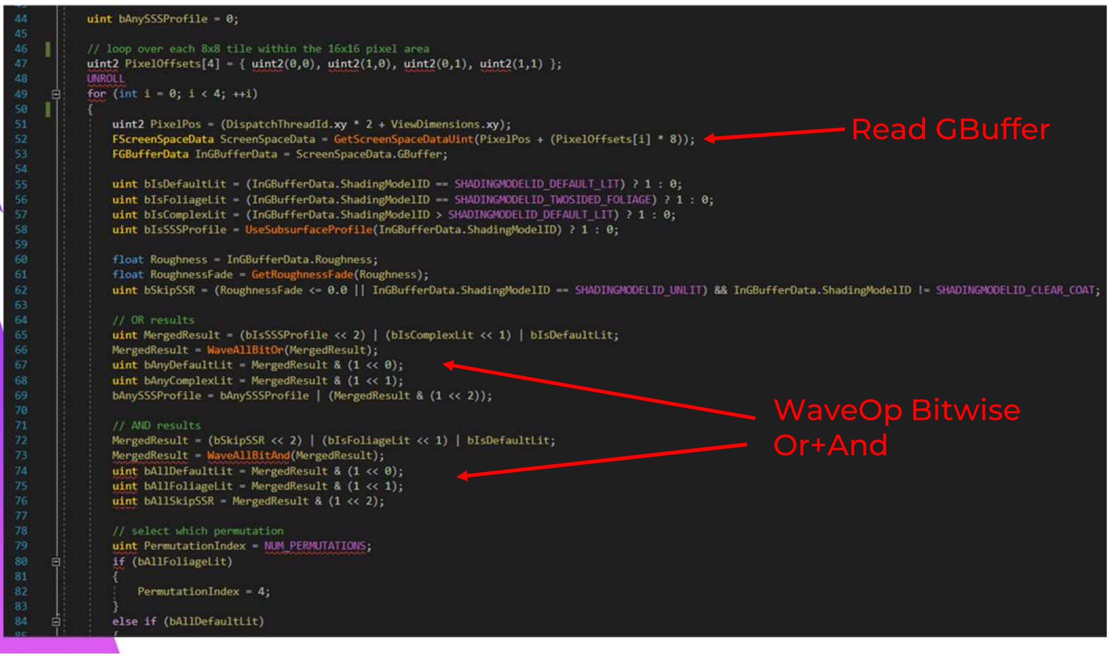
Remember, the fastest wavefront is the one you never launch!

Tile Classification 2.0

- Better Idea: skip empty waves in multiple passes based on a single classify step
- Cull unlit (skybox tiles) from everything
- Skip SSR trace on tiles above roughness threshold
- Skip SSS on tiles with no skin materials in them, run required clears with simplified shader permutation

So here’s the updated plan: cull unlit tiles from all of the passes, add a classification for if all the pixels are too rough to trigger SSR traces, and skip SSS on tiles with no skin materials on them, and run a simplified clear on tiles that need to be cleared but don’t need the full SSS set-up.

Tile Classify Shader



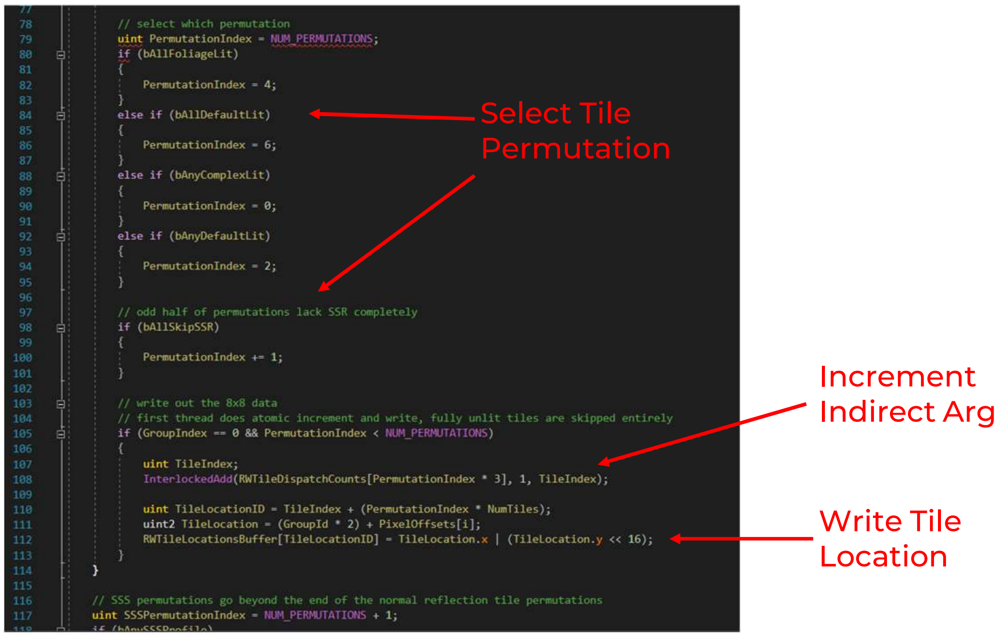
```
    uint bAnySSSProfile = 0;

    // loop over each 8x8 tile within the 16x16 pixel area
    uint2 PixelOffsets[4] = { uint2(0,0), uint2(1,0), uint2(0,1), uint2(1,1) };
    UNROLL
    for (int i = 0; i < 4; ++i)
    {
        uint2 PixelPos = (DispatchThreadId.xy * 2 + ViewDimensions.xy);
        FScreenSpaceData ScreenSpaceData = GetScreenSpaceDataUint(PixelPos + (PixelOffsets[i] * 8));
        FGBufferData InGBufferData = ScreenSpaceData.GBuffer;

        uint bIsDefaultLit = (InGBufferData.ShadingModelID == SHADINGMODELID_DEFAULT_LIT) ? 1 : 0;
        uint bIsFoliageLit = (InGBufferData.ShadingModelID == SHADINGMODELID_TWOSIDED_FOLIAGE) ? 1 : 0;
        uint bIsComplexLit = (InGBufferData.ShadingModelID > SHADINGMODELID_DEFAULT_LIT) ? 1 : 0;
        uint bIsSSSProfile = UseSubsurfaceProfile(InGBufferData.ShadingModelID) ? 1 : 0;

        float Roughness = InGBufferData.Roughness;
        float RoughnessFade = GetRoughnessFade(Roughness);
        uint bSkipSSR = (RoughnessFade <= 0.0 || InGBufferData.ShadingModelID == SHADINGMODELID_UNLIT) && InGBufferData.ShadingModelID != SHADINGMODELID_CLEAR_COAT;

        // OR results
        uint MergedResult = (bIsSSSProfile << 2) | (bIsComplexLit << 1) | bIsDefaultLit;
        MergedResult = WaveAllBitOr(MergedResult);
        uint bAnyDefaultLit = MergedResult & (1 << 0);
        uint bAnyComplexLit = MergedResult & (1 << 1);
        bAnySSSProfile = bAnySSSProfile | (MergedResult & (1 << 2));

        // AND results
        MergedResult = (bSkipSSR << 2) | (bIsFoliageLit << 1) | bIsDefaultLit;
        MergedResult = WaveAllBitAnd(MergedResult);
        uint bAllDefaultLit = MergedResult & (1 << 0);
        uint bAllFoliageLit = MergedResult & (1 << 1);
        uint bAllSkipSSR = MergedResult & (1 << 2);

        // select which permutation
        uint PermutationIndex = NUM_PERMUTATIONS;
        if (bAllFoliageLit)
        {
            PermutationIndex = 4;
        }
        else if (bAllDefaultLit)
```

let’s walk through a little bit of what is happening in the Tile Classify shader code. Classification happens on 8x8 tiles, but each group covers a 16x16 area because subsurface scattering is happening at half res. After sampling the gbuffer properties with UE4’s GetScreenSpaceDataUint function, I use wave ops to merge the bitmasks for each 8x8 tile together.

You can see that in the code this happening with the UE4 shader API commands WaveAllBitOr and WaveAllBitAnd. After these wave ops, each thread in the wavefront is going to hold the same mask value in MergedResult.

One benefit to using wave ops is that the logic following the wave commands is all going to be scalar ALU, as the compiler knows MergedResult is going to be uniform across the wave

Tile Classify Shader

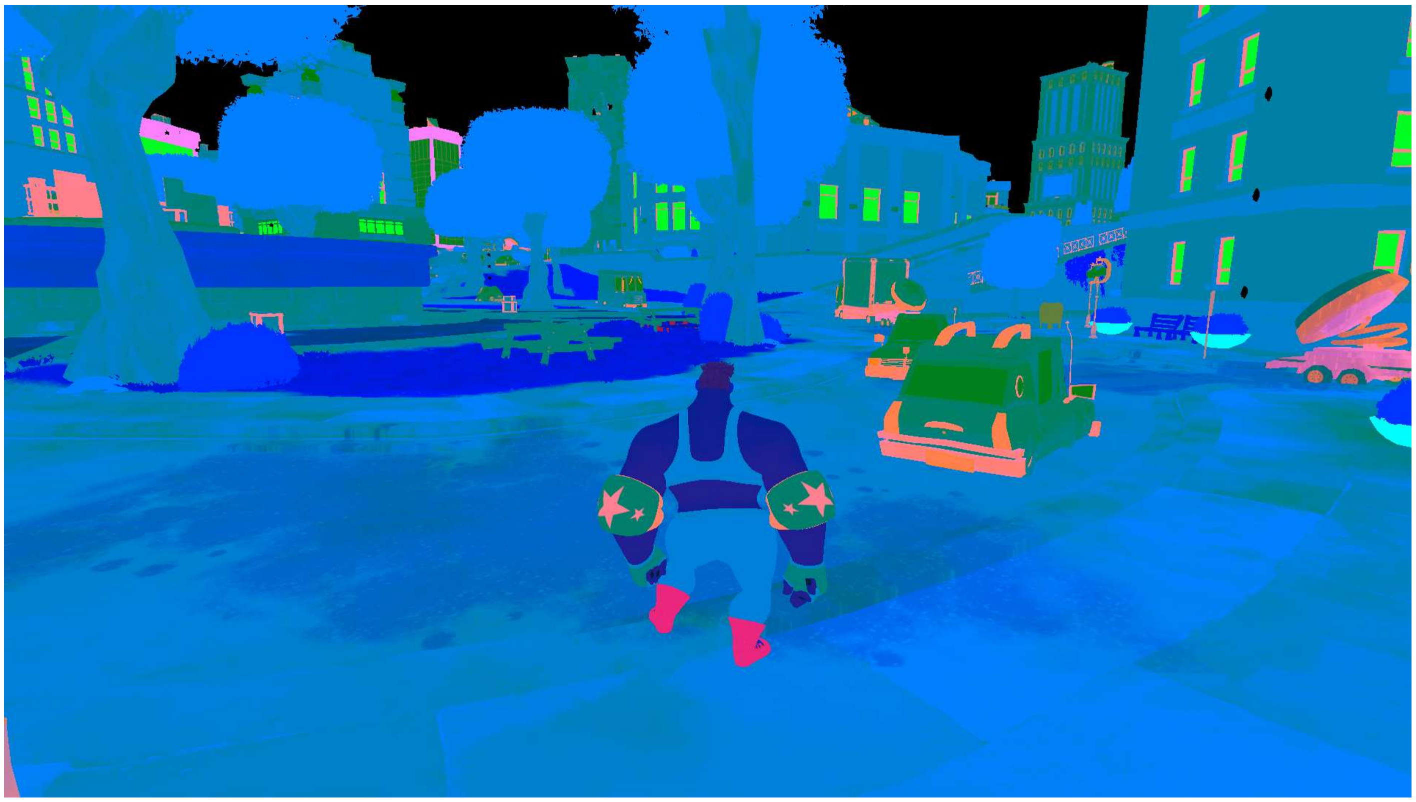


```
        // select which permutation
        uint PermutationIndex = NUM_PERMUTATIONS;
        if (bAllFoliageLit)
        {
            PermutationIndex = 4;
        }
        else if (bAllDefaultLit)
        {
            PermutationIndex = 6;
        }
        else if (bAnyComplexLit)
        {
            PermutationIndex = 0;
        }
        else if (bAnyDefaultLit)
        {
            PermutationIndex = 2;
        }

        // odd half of permutations lack SSR completely
        if (bAllSkipSSR)
        {
            PermutationIndex += 1;
        }

        // write out the 8x8 data
        // first thread does atomic increment and write, fully unlit tiles are skipped entirely
        if (GroupIndex == 0 && PermutationIndex < NUM_PERMUTATIONS)
        {
            uint TileIndex;
            InterlockedAdd(RWTileDispatchCounts[PermutationIndex * 3], 1, TileIndex);

            uint TileLocationID = TileIndex + (PermutationIndex * NumTiles);
            uint2 TileLocation = (GroupId * 2) + PixelOffsets[i];
            RWTileLocationsBuffer[TileLocationID] = TileLocation.x | (TileLocation.y << 16);
        }
    }

    // SSS permutations go beyond the end of the normal reflection tile permutations
    uint SSSPermutationIndex = NUM_PERMUTATIONS + 1;
```

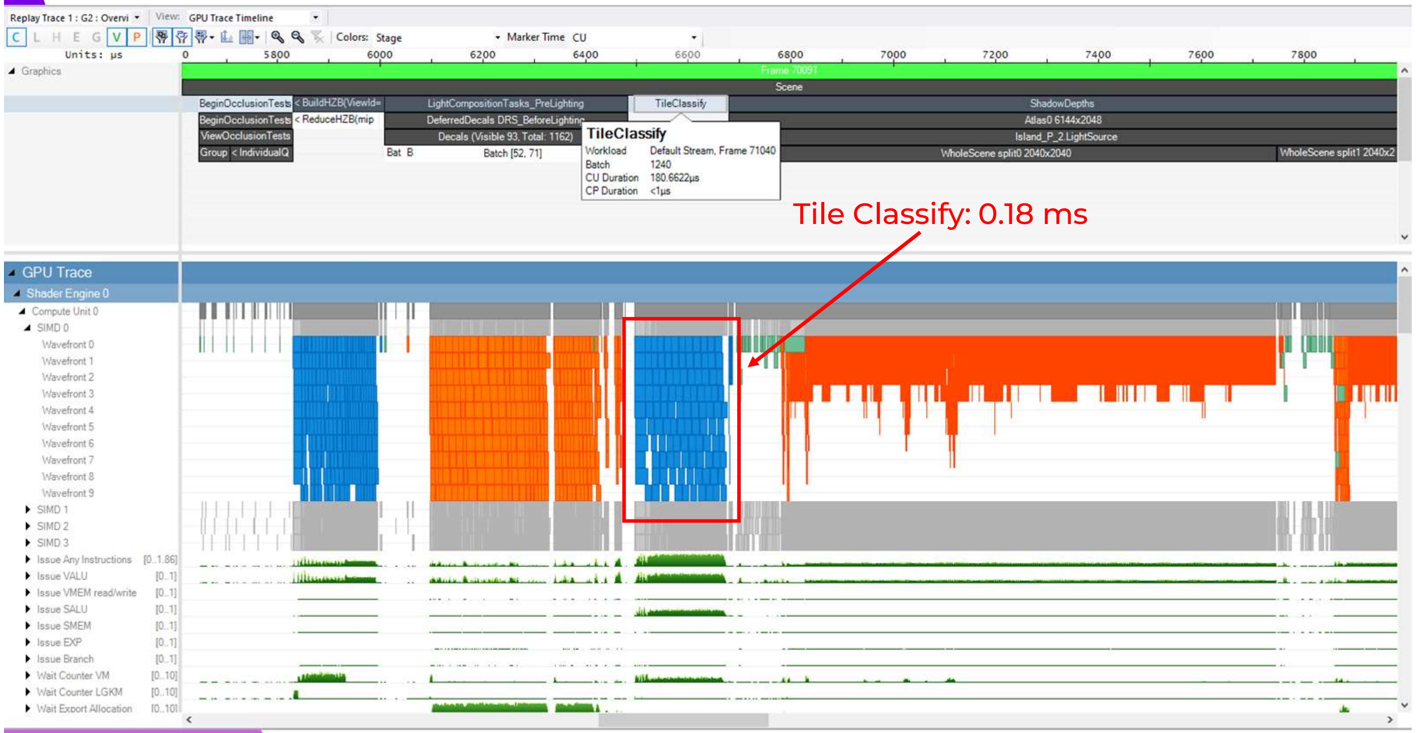
Then the shader permutation for the tile is selected based on the bits held in the MergedResult across the whole wave, and the result is written out by the first thread. An interlocked add occurs on the counts which gets a unique index for the tile that maps to the Tile Locations Buffer, which holds the pixel location for a given tile on the screen and will be used to reconstruct the pixel locations for each tile in the apply shaders.

Note that I selected 8x8 tiles because those are the size of 1 wavefront (64 threads) on GCN. Uncharted 4 used 16x16 tiles – which cuts the memory needed for the tile lists to 25%. Tile location lists require memory equal to max tile count * permutation count. 8x8 tiles will allow tighter bounds on expensive material paths. I've opted for 8x8 partly because our permutation count is more limited. For example, it's in my backlog to try adding a path for tiles that contain foliage and default lit, as that is our most common boundary case falling into the complex path, but that would require more memory.

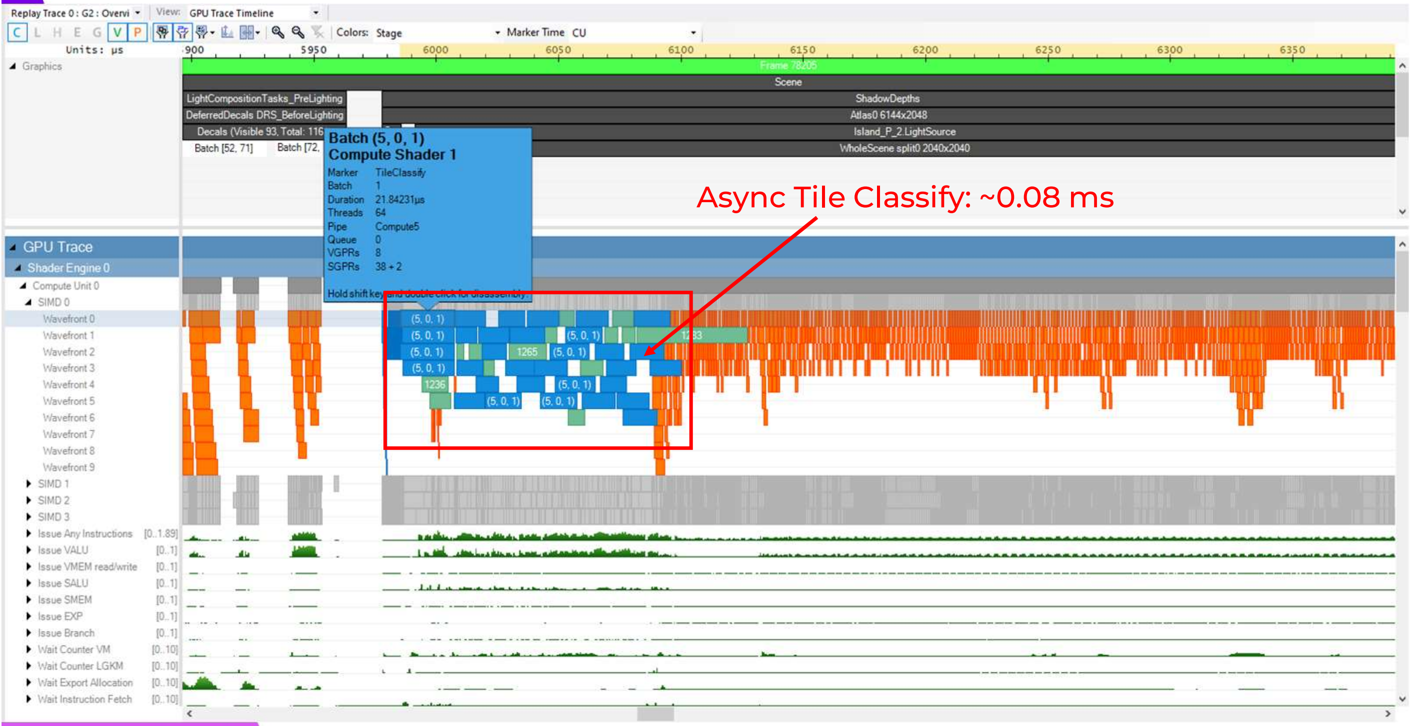
We have 10 shader permutations currently – which results in a 1.296 mb tile locations buffer for 8x8 tiles at 1080p. I should be able to reclaim 48 kb if I didn't over-allocate for the half-res tile lists, but most of the memory is coming from the 8 permutations used for SSR+Reflection Apply



It’s worth noting – one really great thing about this shader is that those calls to sample the gbuffer properties all map back to just a single texture read. Epic has conveniently already packed all that information into just one Gbuffer target for us, which holds both Roughness and Material ID, which I’m showing for my shot here.



Now back in razor let's look at the cost of running this classification shader – it takes a modest 0.18 ms on a base PS4 at 1080p, and can execute as soon as the decals are done modifying the gbuffer



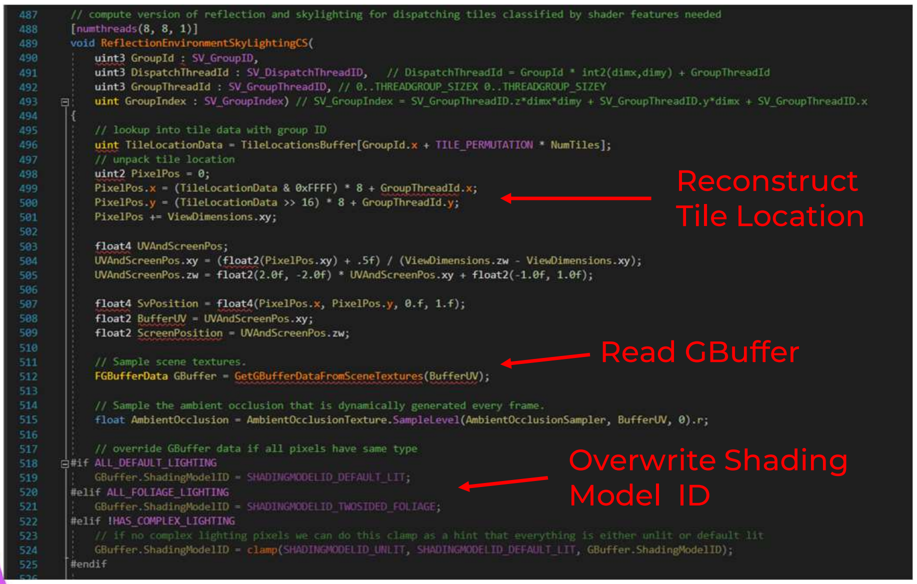
Batch (5, 0, 1)
Compute Shader 1

Marker	TileClassify
Batch	1
Duration	21.84231µs
Threads	64
Pipe	Compute5
Queue	0
VGPRs	8
SGPRs	38 + 2

Hold shift key and double click for disassembly.

And we can actually do better, this classification job very nicely overlaps with shadow depth rendering using async compute, here you can see it overlapping with some vertex shading work before some pixel shader waves for masked materials run. This is frame dependent, but I generally see running it async save ~0.1 ms of frame time which makes the cost approximately 0.08 ms on a PS4.

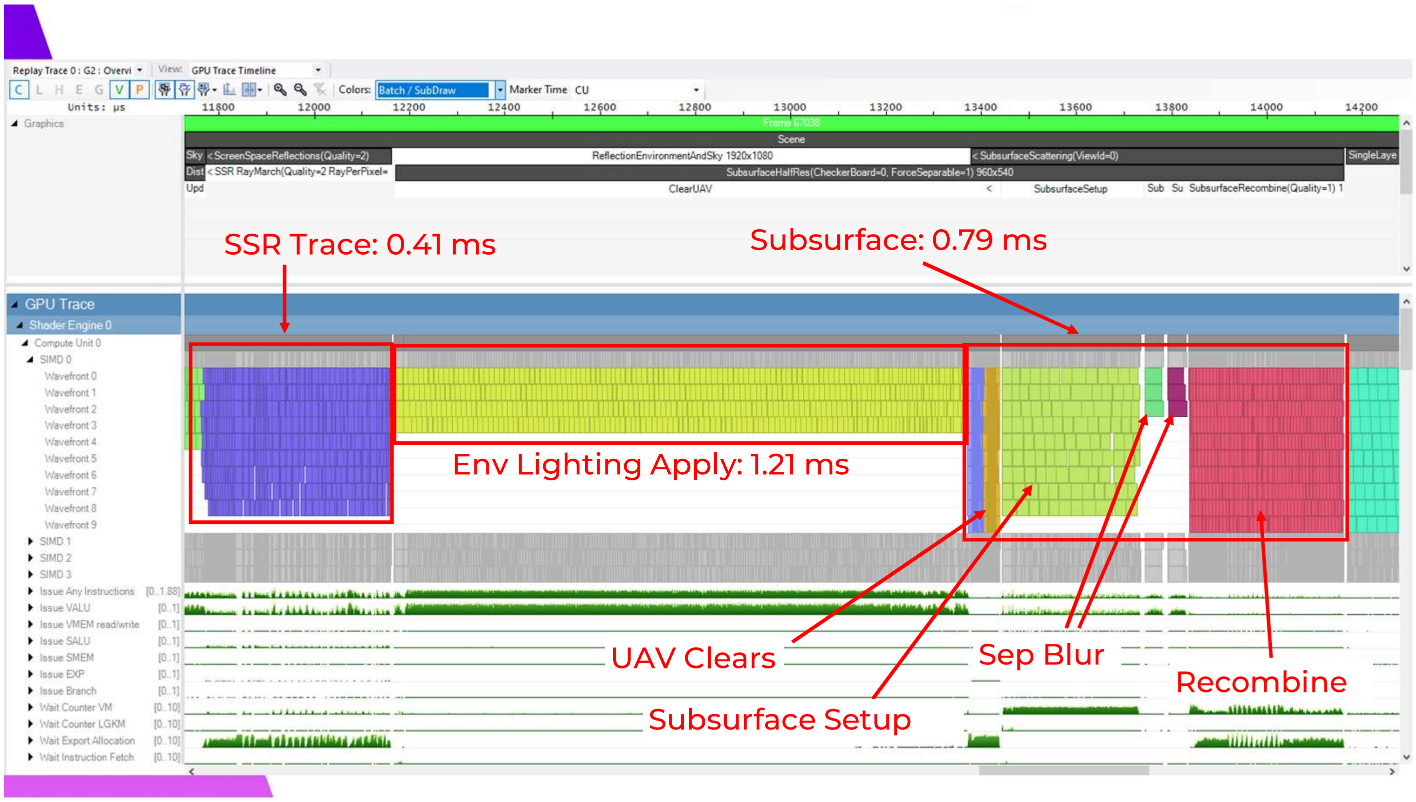
Env Apply Shader



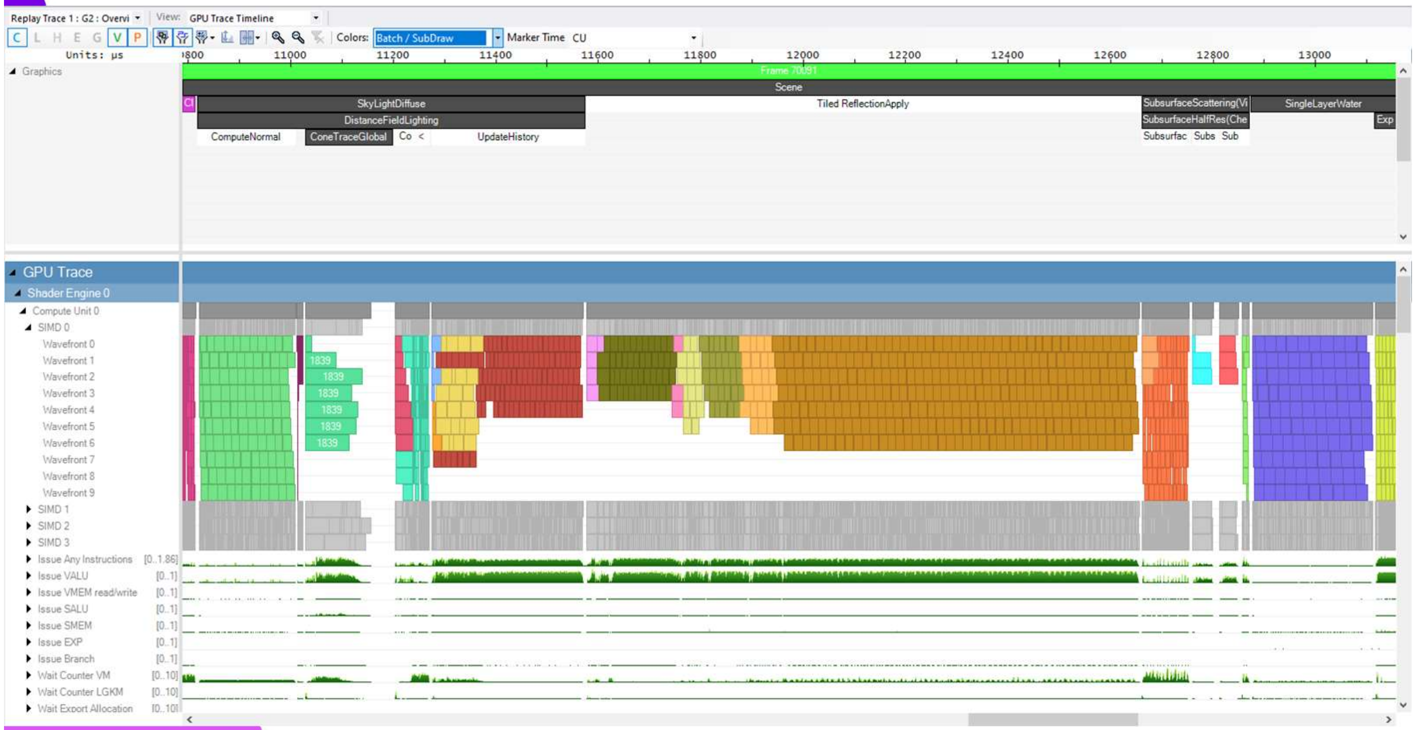
```
// compute version of reflection and skylighting for dispatching tiles classified by shader features needed
[numthreads(8, 8, 1)]
void ReflectionEnvironmentSkyLightingCS(
    uint3 GroupId : SV_GroupID,
    uint3 DispatchThreadId : SV_DispatchThreadID,   // DispatchThreadId = GroupId * int2(dimx,dimy) + GroupThreadId
    uint3 GroupThreadId : SV_GroupThreadID, // 0..THREADGROUP_SIZEX 0..THREADGROUP_SIZEY
    uint GroupIndex : SV_GroupIndex) // SV_GroupIndex = SV_GroupThreadID.z*dimx*dimy + SV_GroupThreadID.y*dimx + SV_GroupThreadID.x
{
    // lookup into tile data with group ID
    uint TileLocationData = TileLocationsBuffer[GroupId.x + TILE_PERMUTATION * NumTiles];
    // unpack tile location
    uint2 PixelPos = 0;
    PixelPos.x = (TileLocationData & 0xFFFF) * 8 + GroupThreadId.x;
    PixelPos.y = (TileLocationData >> 16) * 8 + GroupThreadId.y;
    PixelPos += ViewDimensions.xy;

    float4 UVAndScreenPos;
    UVAndScreenPos.xy = (float2(PixelPos.xy) + .5f) / (ViewDimensions.zw - ViewDimensions.xy);
    UVAndScreenPos.zw = float2(2.0f, -2.0f) * UVAndScreenPos.xy + float2(-1.0f, 1.0f);

    float4 SvPosition = float4(PixelPos.x, PixelPos.y, 0.f, 1.f);
    float2 BufferUV = UVAndScreenPos.xy;
    float2 ScreenPosition = UVAndScreenPos.zw;

    // Sample scene textures.
    FGBufferData GBuffer = GetGBufferDataFromSceneTextures(BufferUV);

    // Sample the ambient occlusion that is dynamically generated every frame.
    float AmbientOcclusion = AmbientOcclusionTexture.SampleLevel(AmbientOcclusionSampler, BufferUV, 0).r;

    // override GBuffer data if all pixels have same type
#if ALL_DEFAULT_LIGHTING
    GBuffer.ShadingModelID = SHADINGMODELID_DEFAULT_LIT;
#elif ALL_FOLIAGE_LIGHTING
    GBuffer.ShadingModelID = SHADINGMODELID_TWOSIDED_FOLIAGE;
#elif !HAS_COMPLEX_LIGHTING
    // if no complex lighting pixels we can do this clamp as a hint that everything is either unlit or default lit
    GBuffer.ShadingModelID = clamp(SHADINGMODELID_UNLIT, SHADINGMODELID_DEFAULT_LIT, GBuffer.ShadingModelID);
#endif
```

Before we look at the performance of the apply steps – let's look at just the compute shader apply for Environment Lighting apply. This was just a full screen pixel shading pass in the original implementation, and this compute shader path runs using repeated calls to DispatchIndirect with different shader permutations.

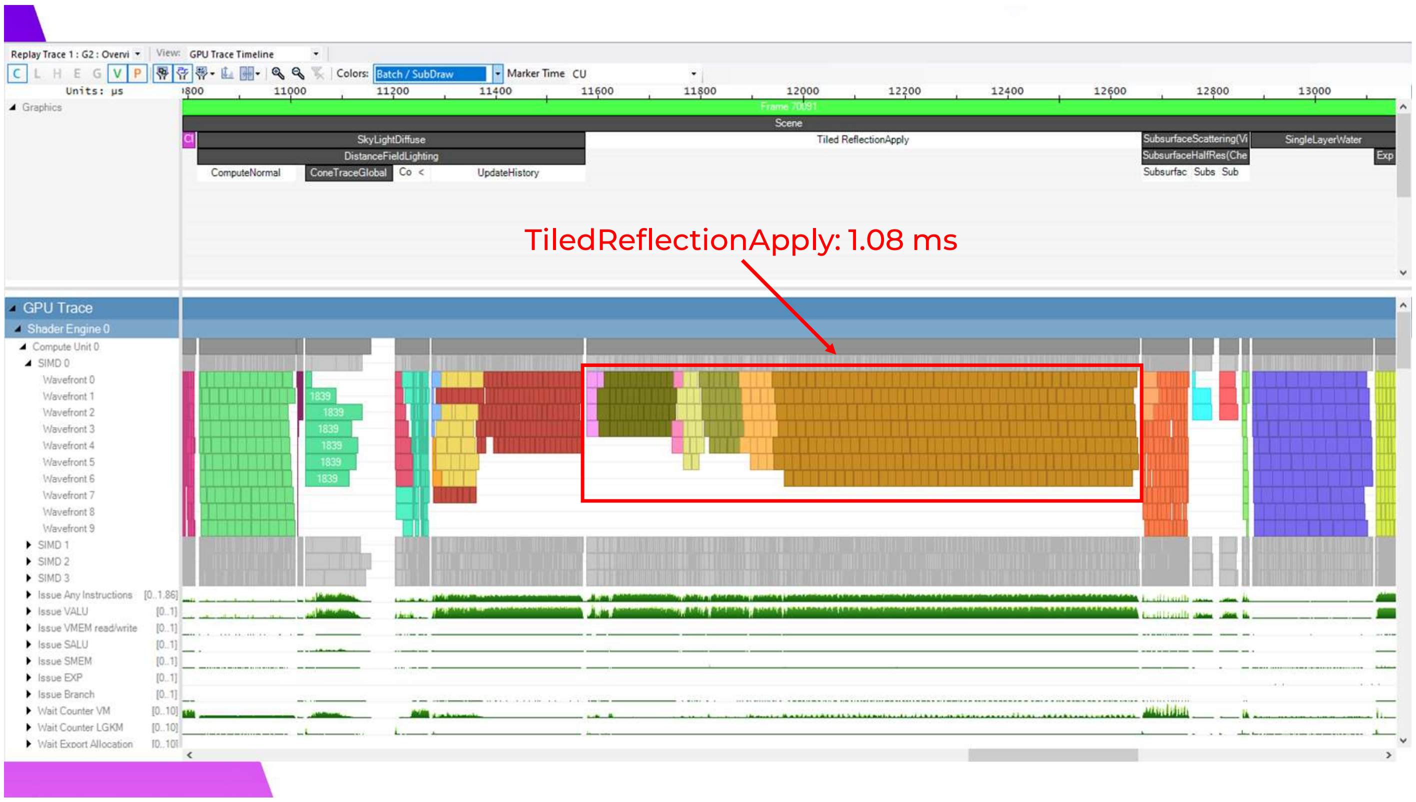
The shader begins by using the GroupId to look up into the tile locations buffer and then unpacking to an individual pixel location based on GroupThreadId. You can see here that after the Gbuffer is read, an overwrite of the ShadingModelID will allow the optimizer to perform dead code elimination based on the preprocessor macros defined based on the shader permutation.


SSR Trace: 0.41 ms
Subsurface: 0.79 ms
Env Lighting Apply: 1.21 ms
UAV Clears
Sep Blur
Recombine
Subsurface Setup
ScreenSpaceReflections(Quality=2)
ReflectionEnvironmentAndSky 1920x1080
SubsurfaceScattering(ViewId=0)
SSR RayMarch(Quality=2 RayPerPixel=
SubsurfaceHalfRes(CheckerBoard=0, ForceSeparable=1) 960x540
ClearUAV
SubsurfaceSetup
SubsurfaceRecombine(Quality=1) 1
GPU Trace
Shader Engine 0
Compute Unit 0

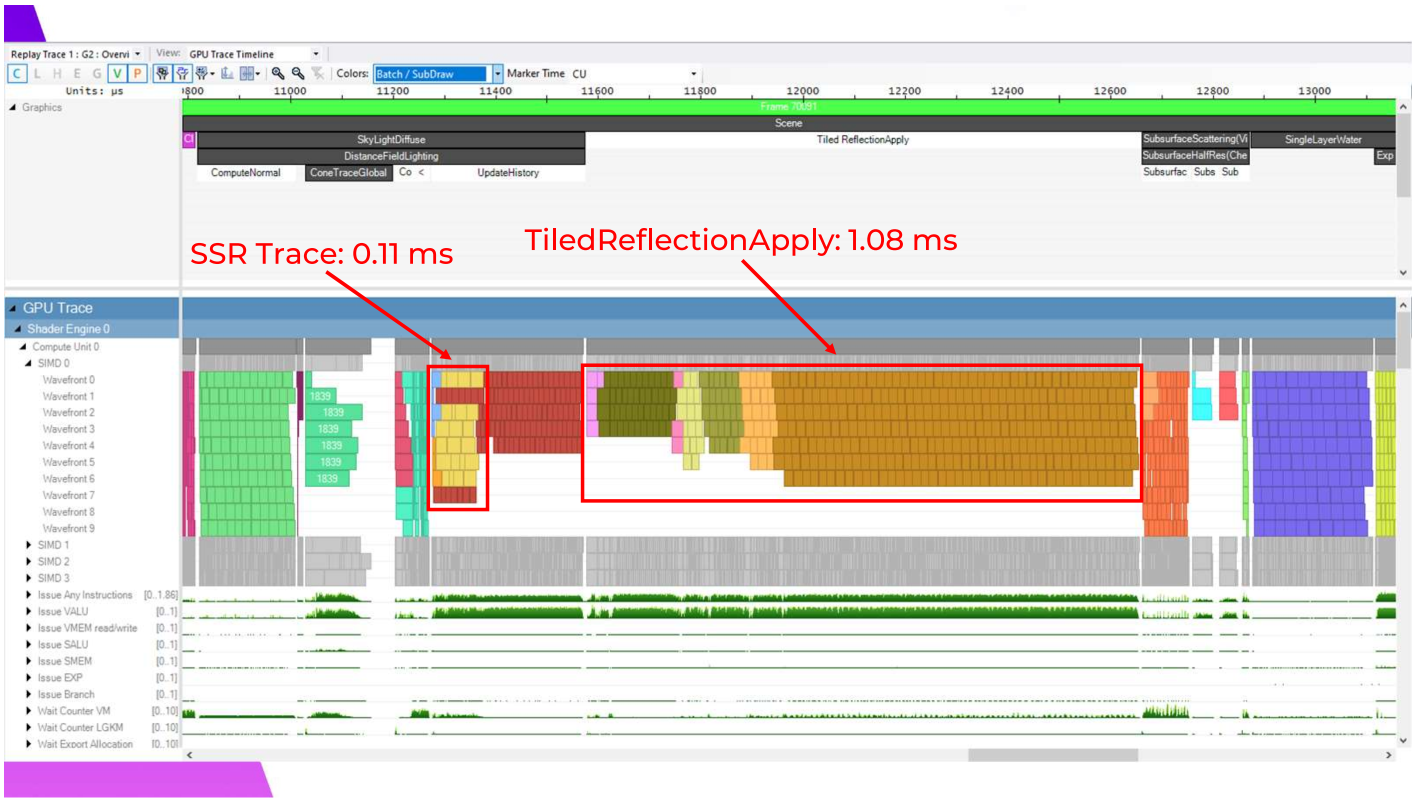
Now we need to look at what this 0.08 ms is buying is in the application of SSR, Reflection Environment, and SSS. Here is the original sequence of shading that I showed before.



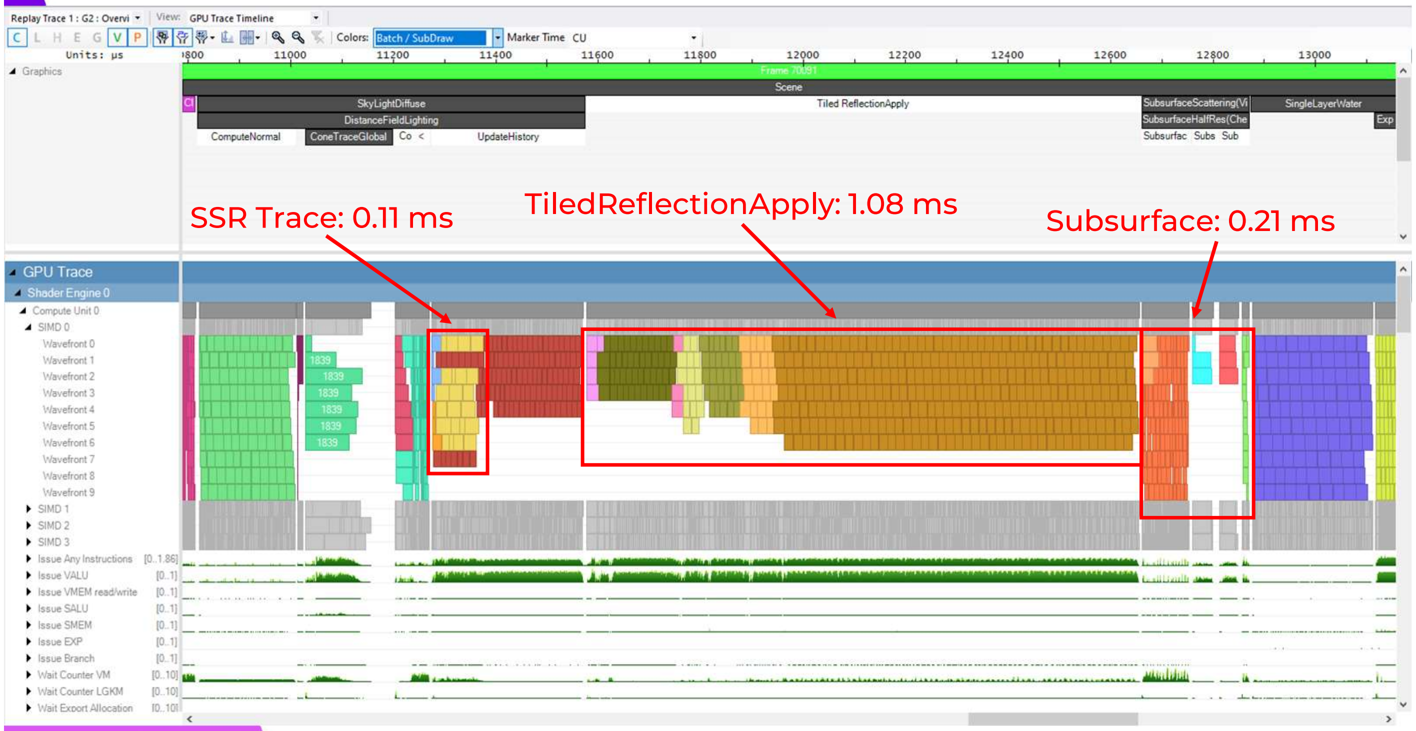
And here is our new frame.



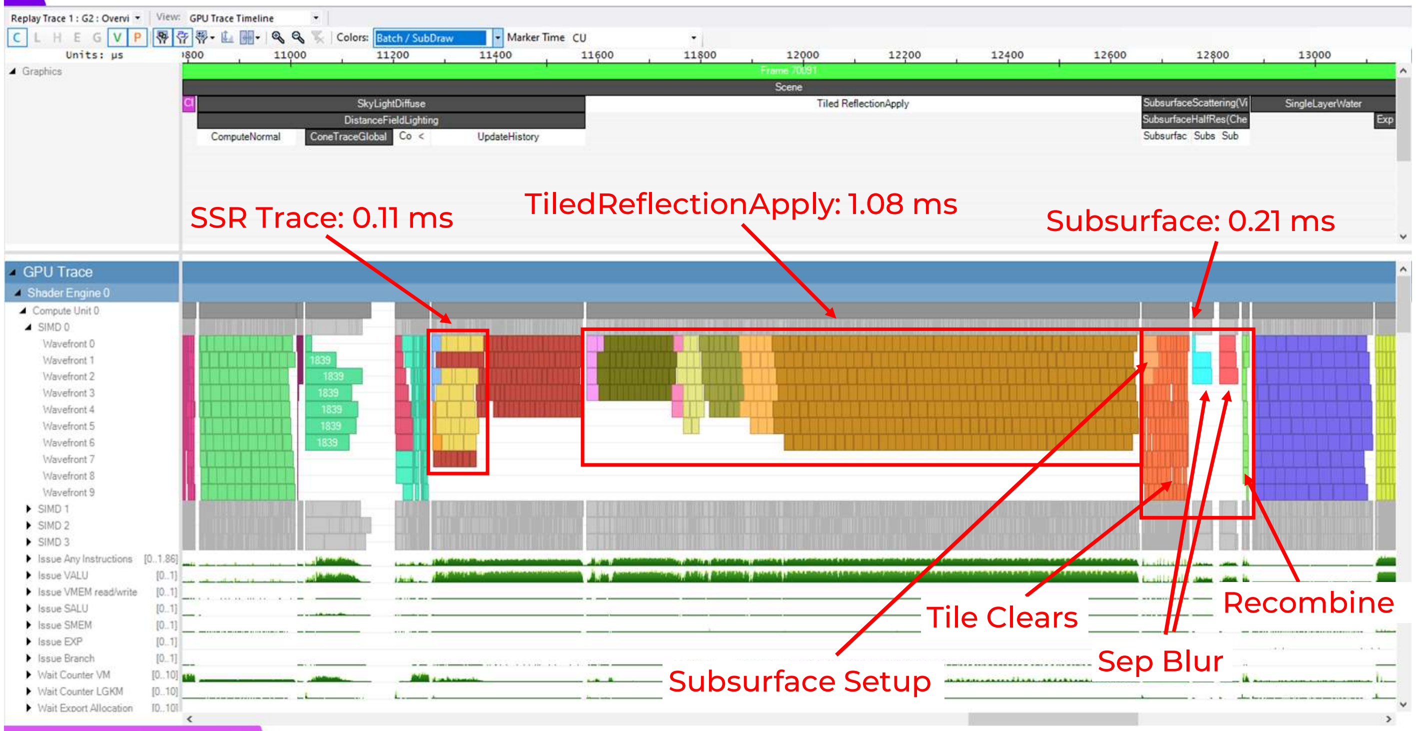
Here we have the Tiled Reflection apply shader at 1.08 ms, an 0.13 ms improvement. About half of this benefit is from culling sky pixels in this frame, so not really worthwhile in frames without sky pixels. One micro-optimization I would point out here is that I have slower waves with lower occupancy ordered first after the initial barrier and then have the fastest waves at the end. This helps reduce cracks between the batches – believe this is from lower occupancy waves waiting on more registers to become available. The lower occupancy waves also tend to be longer running, so putting the fastest batch last helps to ensure work drains quickly before the next barrier.



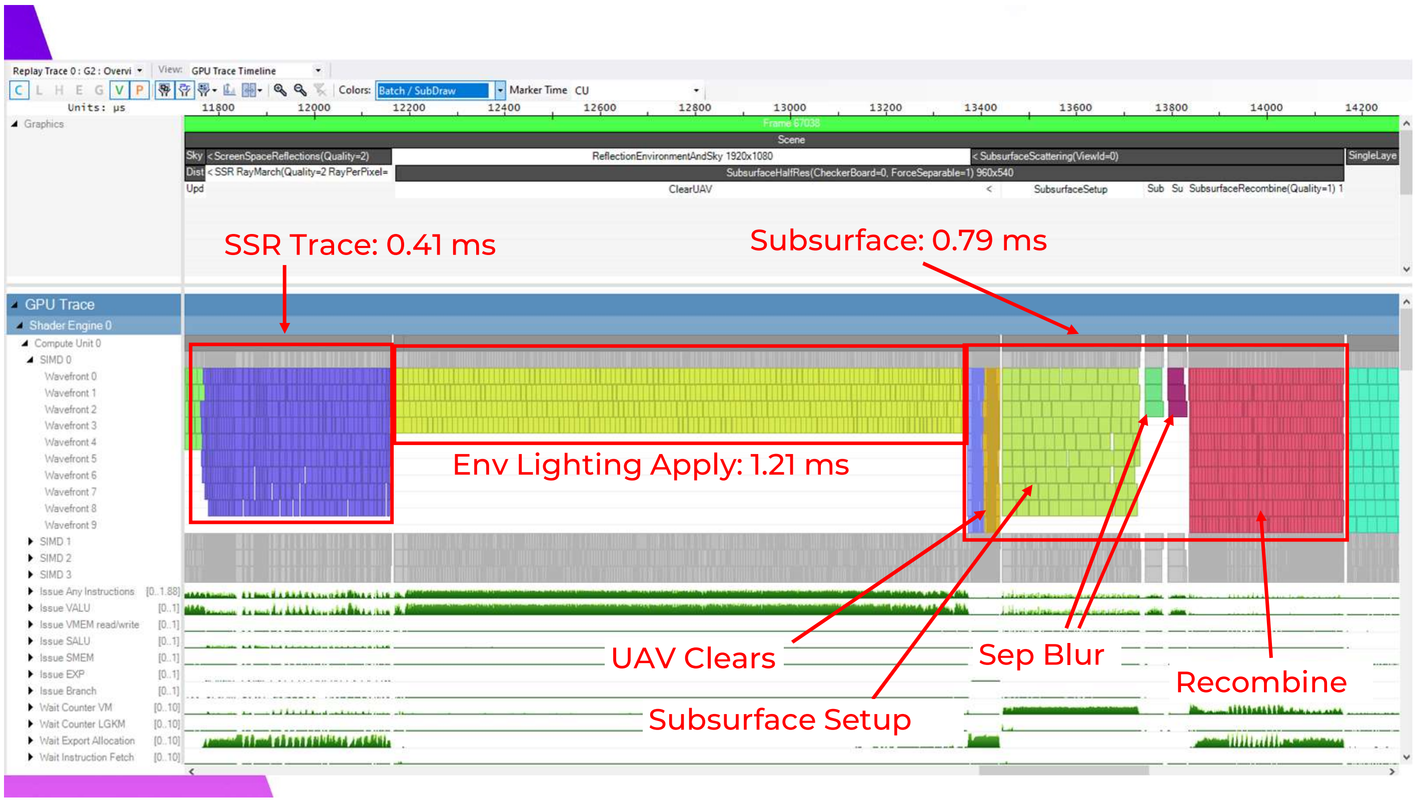
While TiledReflection apply is getting a small benefit, Screen Space Reflections is conversely seeing a really good benefit. 0.3 ms better and that is actually a conservative improvement because the waves are getting better overlap with DFAO history update. There is no need for a barrier as these are writing to separate buffers that both feed into the reflection apply. These results with SSR were what first gave me confidence that this was going to be a worthwhile optimization



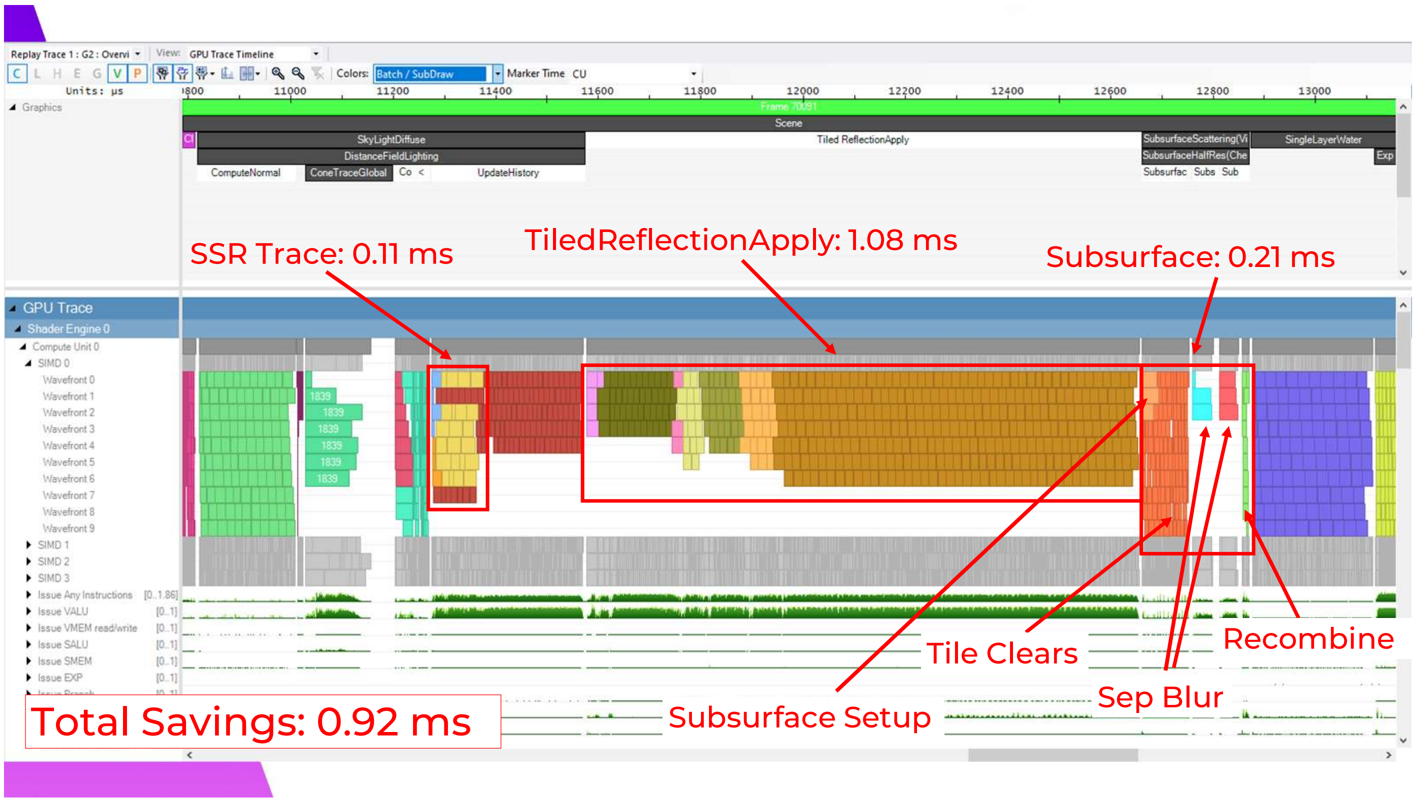
And then Subsurface after the reflection apply, which also sees a huge improvement, 0.58 ms better.



Here you can see the set-up and tile clears get really nice overlap with the tile clears having much shorter waves than the full setup shader. The blur steps are similar from the stock tile classify and the recombine is very fast as only tiles with skin in them are run.


SSR Trace: 0.41 ms
Subsurface: 0.79 ms
Env Lighting Apply: 1.21 ms
UAV Clears
Sep Blur
Recombine
Subsurface Setup
ScreenSpaceReflections(Quality=2)
ReflectionEnvironmentAndSky 1920x1080
SubsurfaceScattering(ViewId=0)
SSR RayMarch(Quality=2 RayPerPixel=
SubsurfaceHalfRes(CheckerBoard=0, ForceSeparable=1) 960x540
ClearUAV
SubsurfaceSetup
SubsurfaceRecombine(Quality=1) 1
GPU Trace
Shader Engine 0
Compute Unit 0

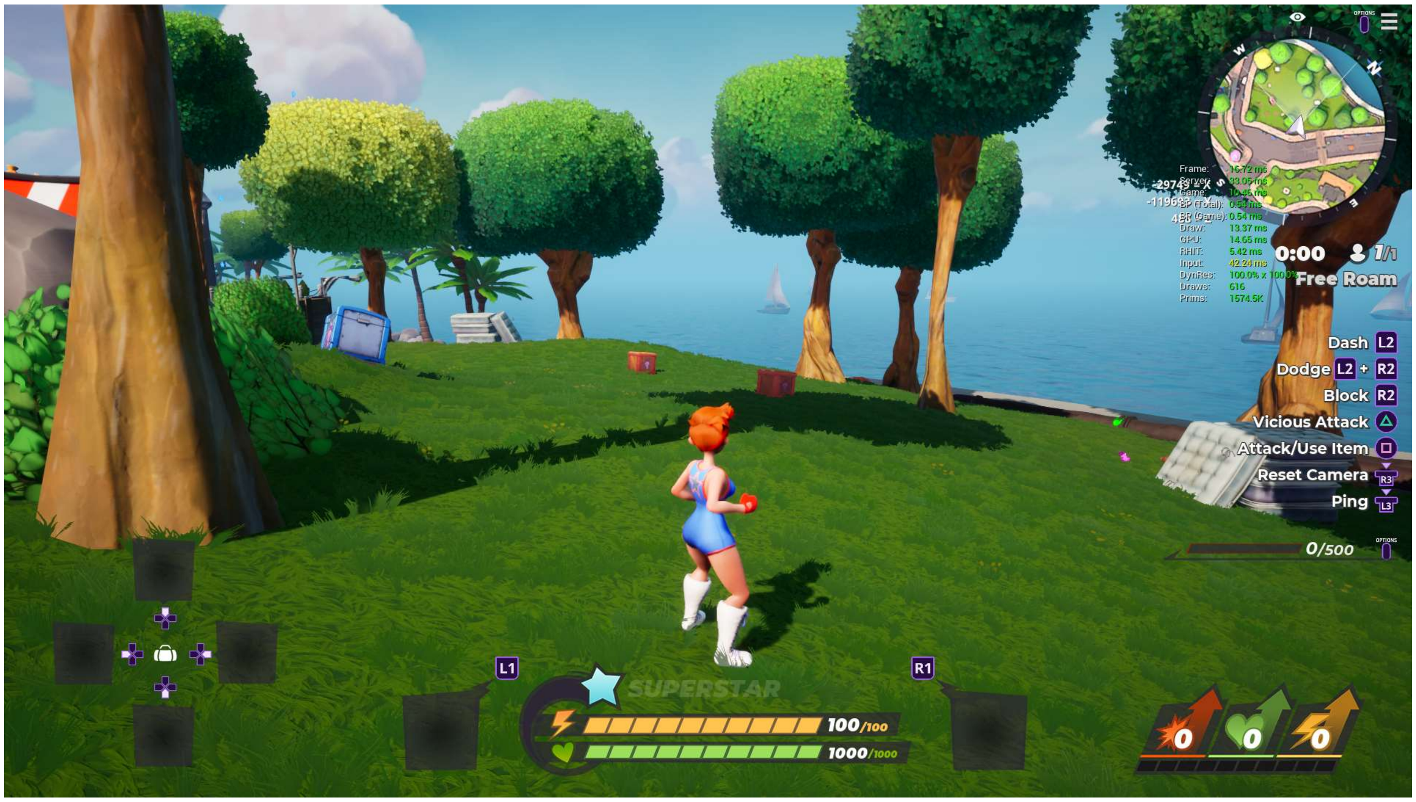
Now that I've laid all this out – just a reminder again with what these passes were before.



And back to our results. This is a total savings of ~1 ms from these passes, although obviously those benefits will vary based on the scene composition, 0.92 ms total for this shot once you subtract the classification cost

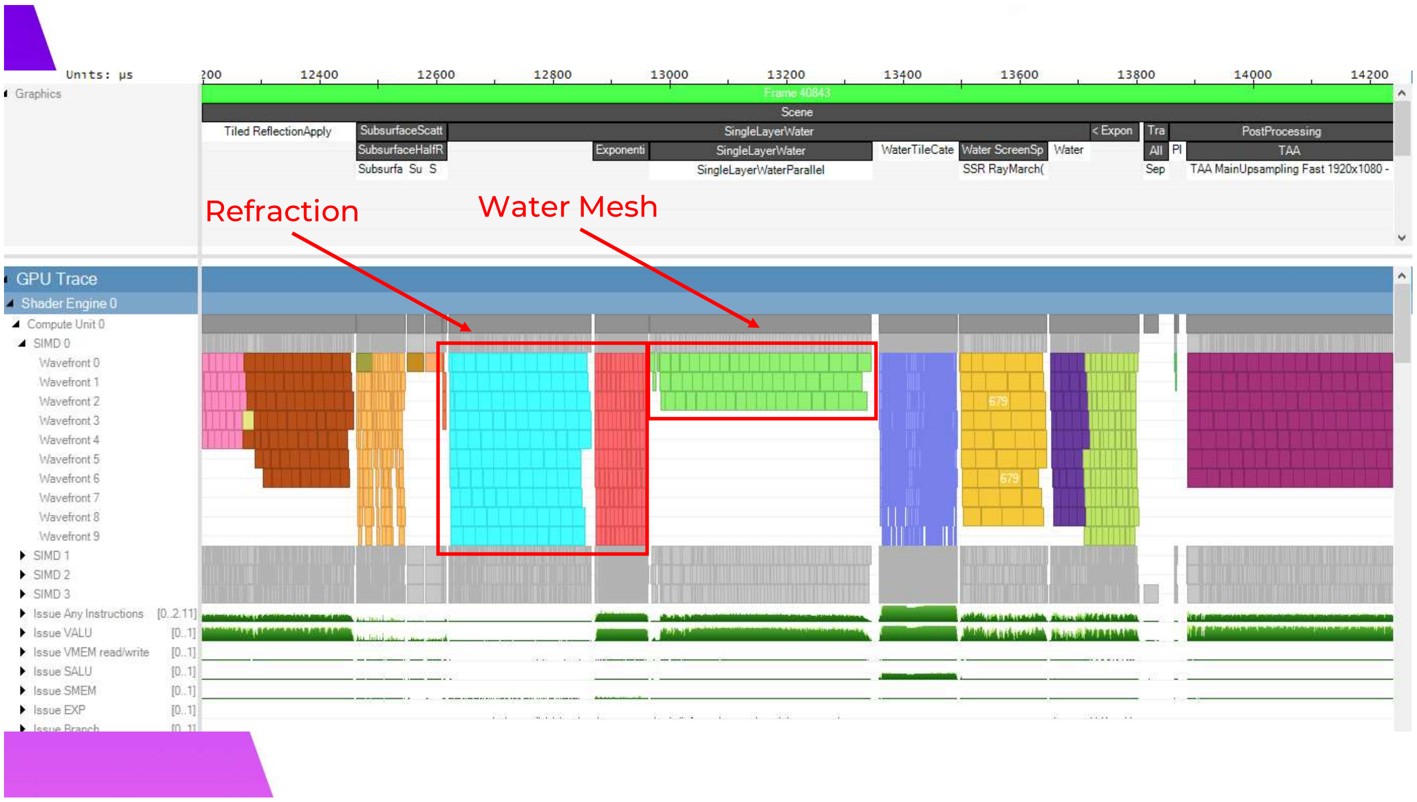
Water Stenciling

I just spent an amount of time talking up the virtues of tile classification, but sometimes it’s just not the right tool for the job. I want to take a moment to show a small example where just using the stencil buffer is a better idea.

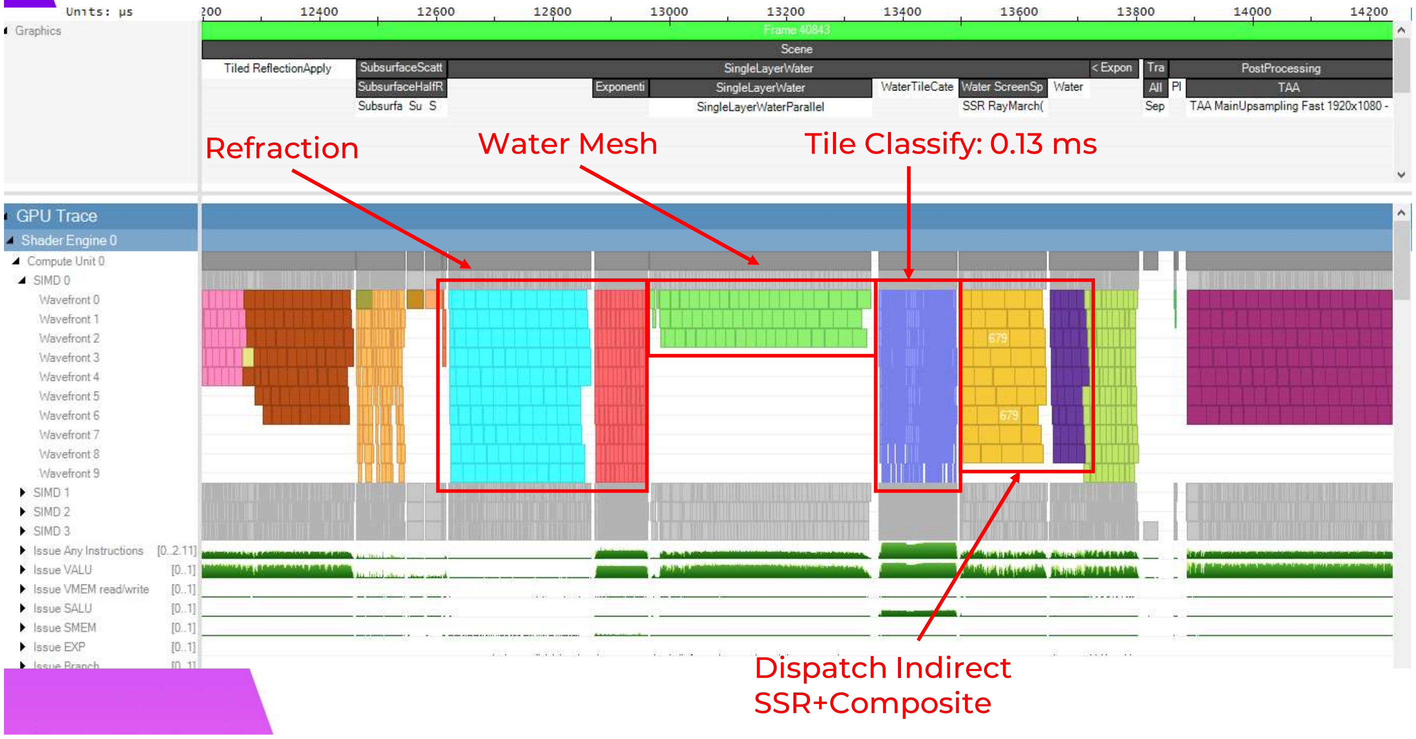


```
Frame:        16.72 ms
Server:       33.05 ms
Game:         10.46 ms
BP (Total):   0.54 ms
BP (Game):    0.54 ms
Draw:         13.37 ms
GPU:          14.65 ms
RHIT:         5.42 ms
Input:        42.24 ms
DynRes:       100.0% x 100.0%
Draws:        616
Prims:        1574.5K
```

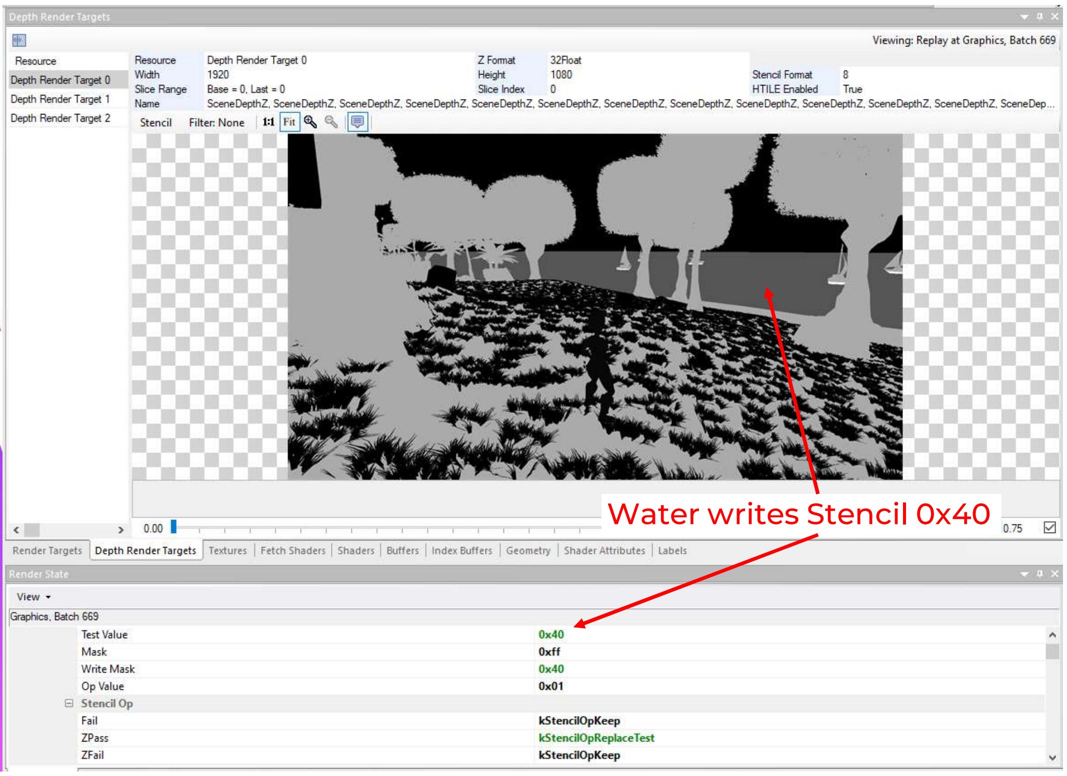
Consider this shot on PS4 at the edge of our Island. UE4’s water system is really nice out of the box, and is taking up a reasonable number of pixels in this frame. The water is taking 1.1 ms to render on PS4.



The water system first copies the scene color out to a secondary buffer and applies fog – this is for refraction. Then the water mesh renders to the full gbuffer like a regular opaque draw. Having an updated depth buffer has benefits for TAA in particular.



Then a classify is done on the gbuffer, very similar to what I did with reflections and subsurface, and just like before, Dispactch Indirect is used to run SSR and Composite only the tiles that were classified as having water in them

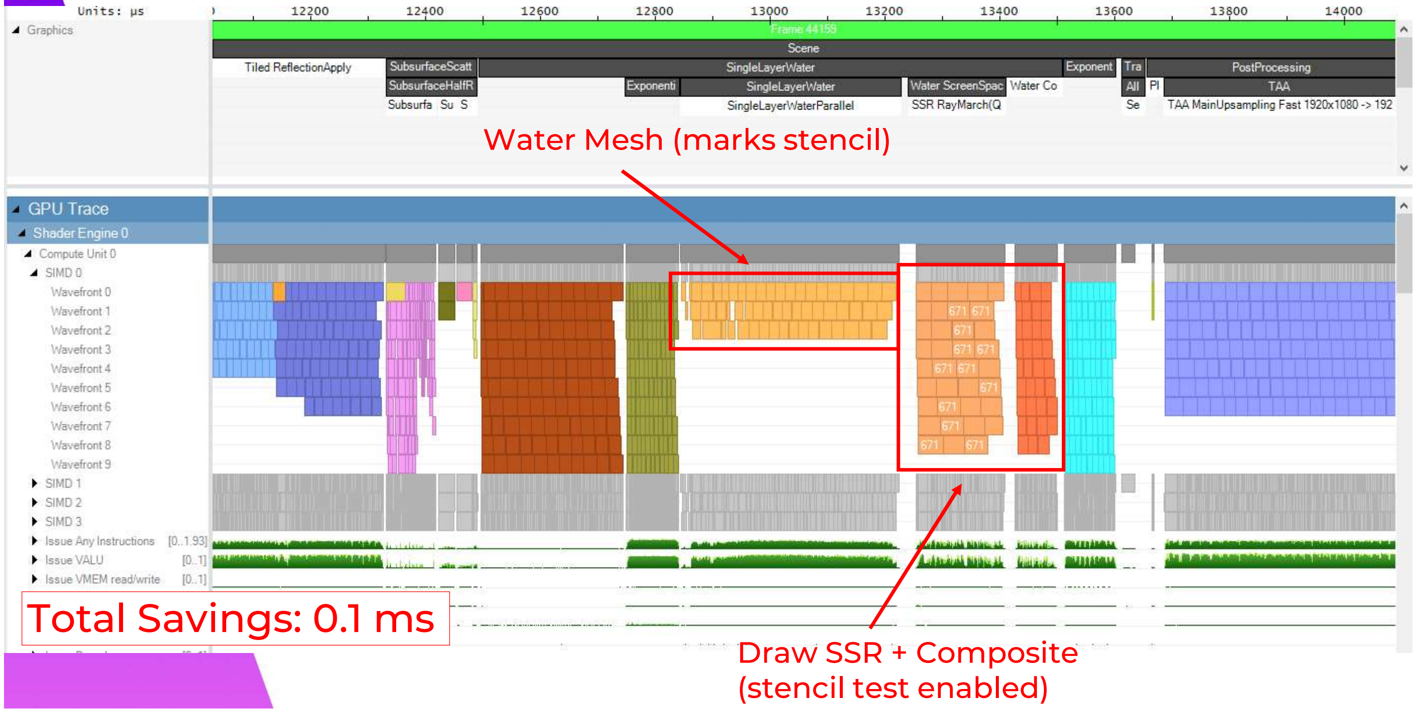


Resource	Depth Render Target 0	Z Format	32Float		
Width	1920	Height	1080	Stencil Format	8
Slice Range	Base = 0, Last = 0	Slice Index	0	HTILE Enabled	True
Name	SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDep...				

Graphics, Batch 669

Test Value	0x40
Mask	0xff
Write Mask	0x40
Op Value	0x01
Stencil Op	
Fail	kStencilOpKeep
ZPass	kStencilOpReplaceTest
ZFail	kStencilOpKeep

However, if you think about it – we're actually doing a visibility test with the water mesh right before tile classify when it is rasterizing with ztest enabled – what we can do instead is simply mark a stencil bit when the water mesh is rendering and then enable the stencil test after. You can see here that we've set the stencil to write 0x40 to the water pixels that pass the depth test.

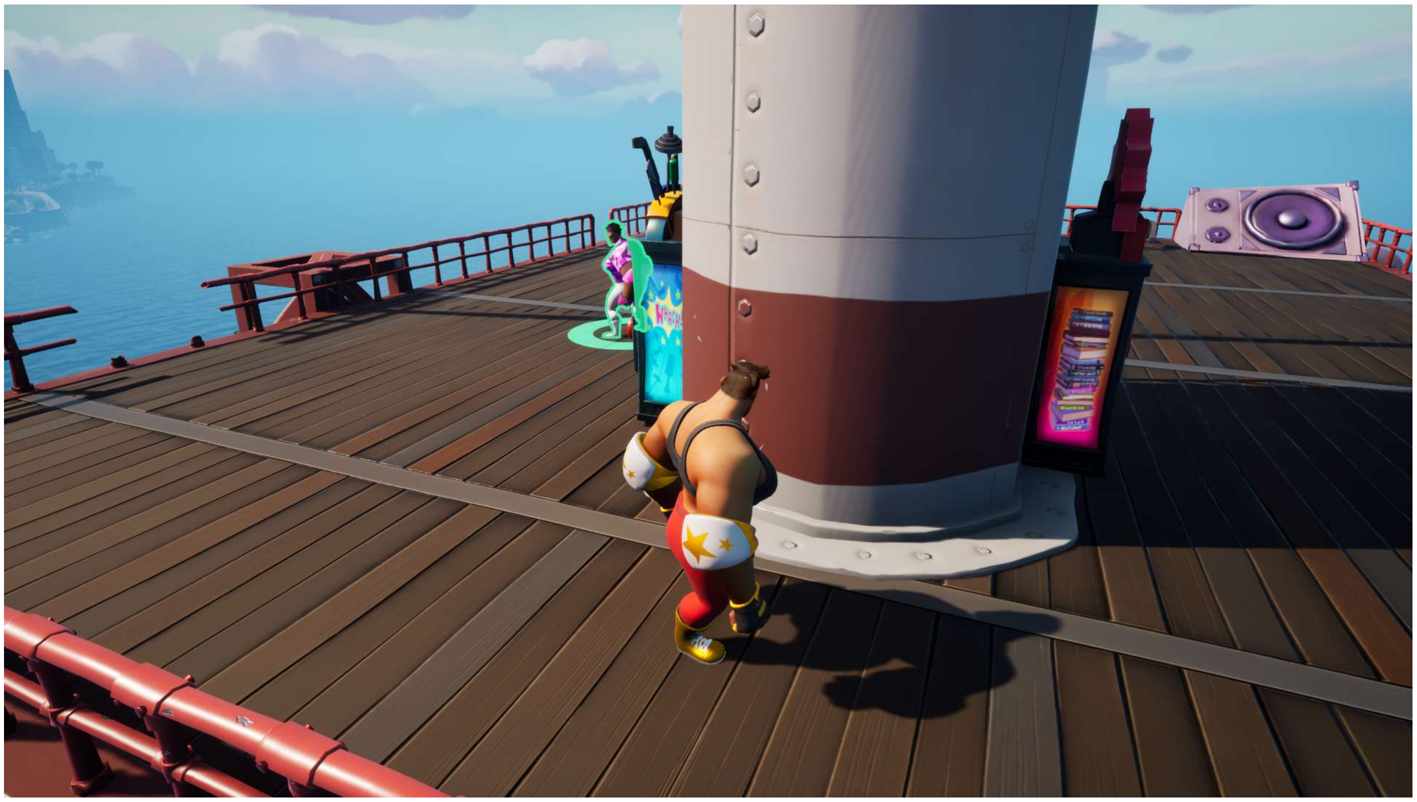


Here's what happens in razor - the tile classify is now missing between rendering the water mesh and handling SSR+Compositing. This saves 0.1 ms for very little effort, we don't get the full 0.13 ms of the tile classify back because we do end up with a barrier before the next pass now.

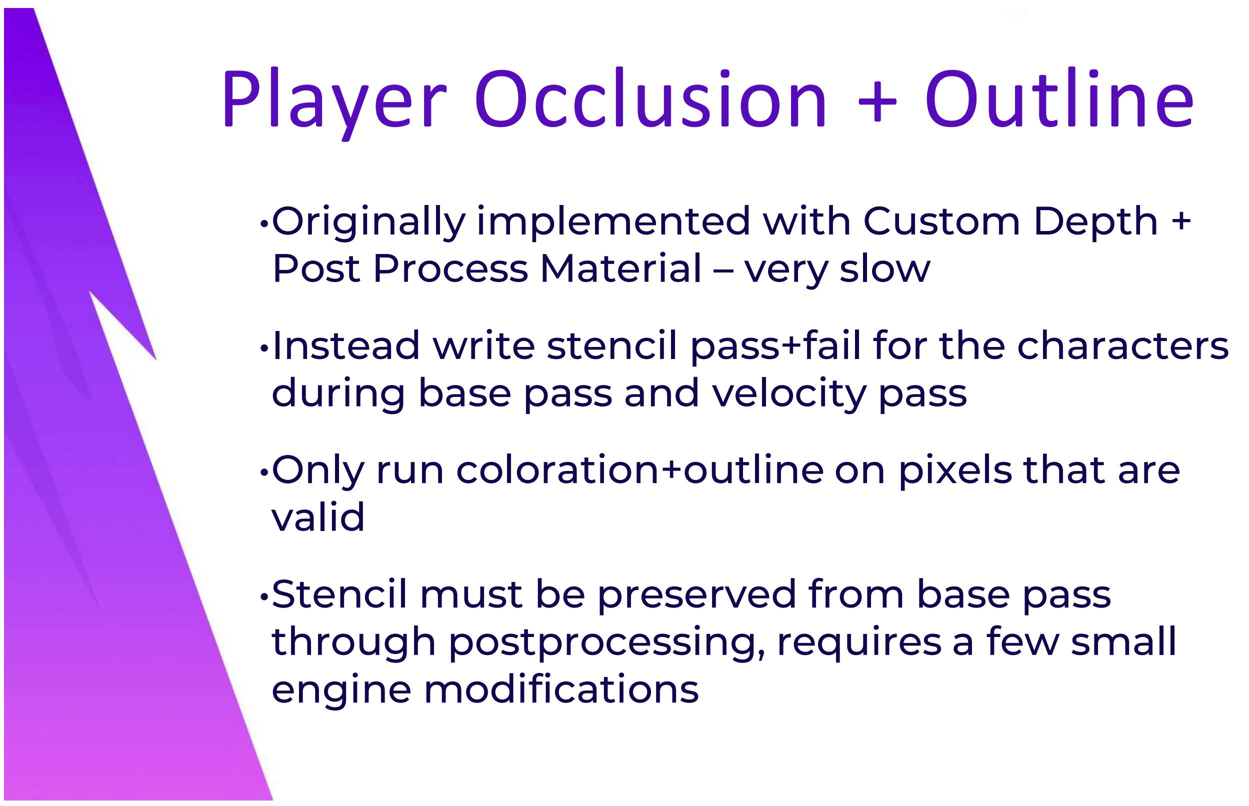
Credit to Oleksii from Dragon's Lake for doing this implementation

Player Occlusion + Outline

My enthusiasm to use all parts of the stencil buffer and efficient EarlyZ utilization extends to our development of new effects for the game as well. I collaborated with my colleague Karinne Lorig to develop an outline and occlusion tint effect for our co-op modes.



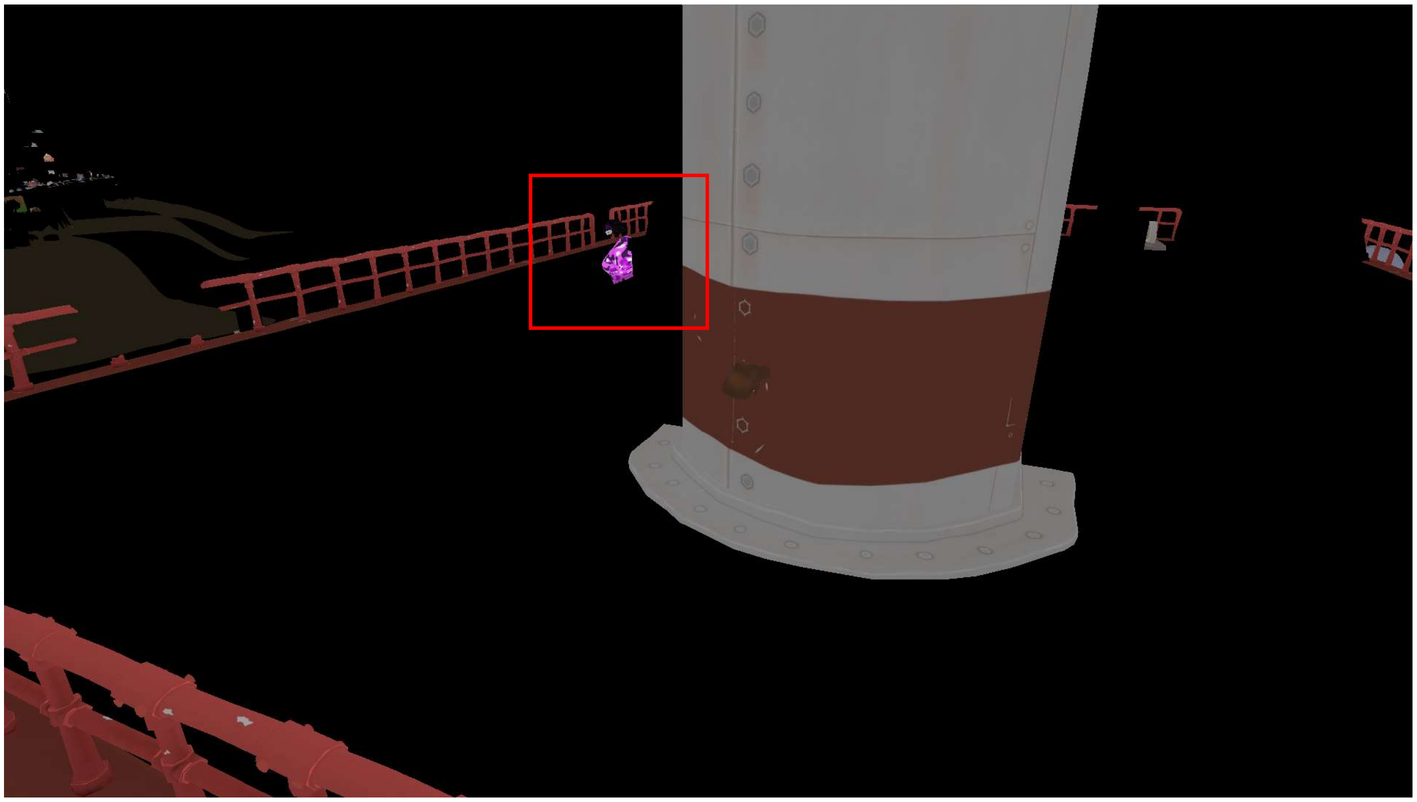
Here is a shot where you can see both parts of the effect on my teammate, which makes them easily identifiable in a brawl with another team, and a more subtle color of the occlusion effect is used on the active player to make sure their silhouette is always visible even if there is an object between the camera and the player.



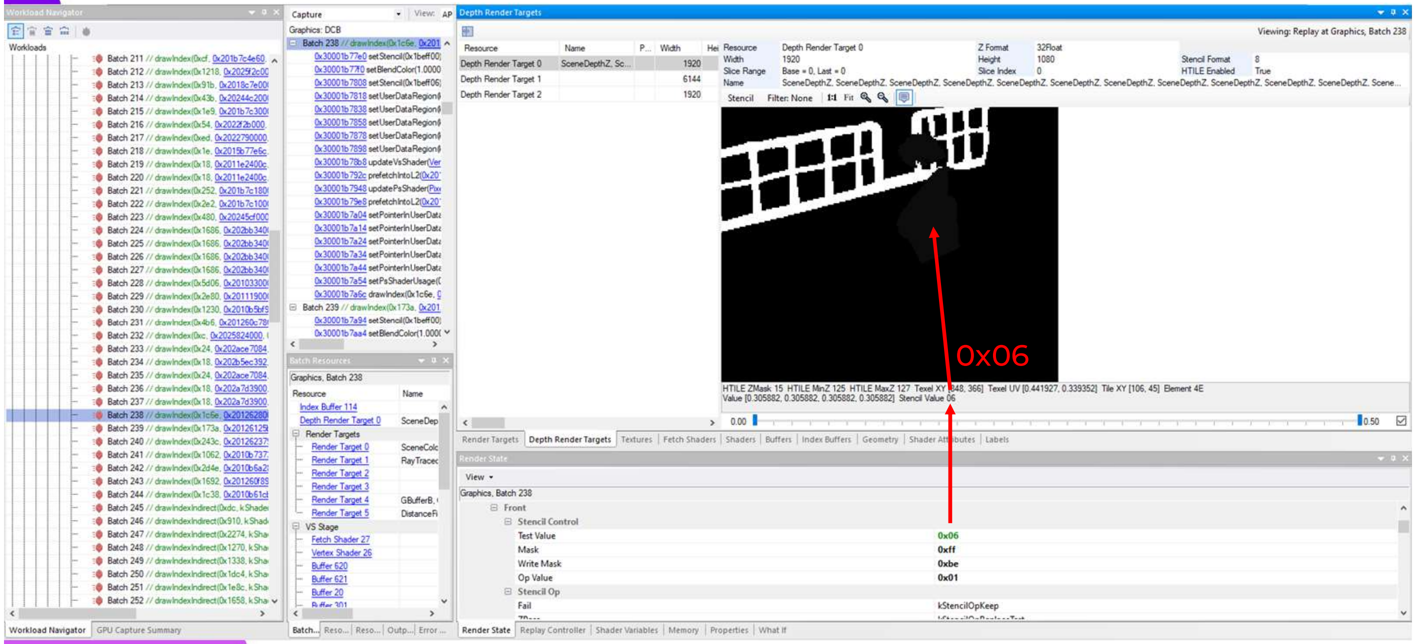
Player Occlusion + Outline

- Originally implemented with Custom Depth + Post Process Material – very slow
- Instead write stencil pass+fail for the characters during base pass and velocity pass
- Only run coloration+outline on pixels that are valid
- Stencil must be preserved from base pass through postprocessing, requires a few small engine modifications

This effect was originally prototyped by artists using UE4's Custom Depth feature and a post process material draw. This has a number of drawbacks: the extra depth buffer consumes memory and requires a full clear, and the characters must be rendered an additional time to it. Furthermore, the apply step runs as a full screen pass. We instead moved to setting stencil bits in the base pass and velocity pass, and then ran a custom shader on only the pixels needed. We did have to make few small engine mods needed to preserve the stencil through the frame, and we have reclaimed some stencil bits from other features that we were not using in the engine.



Let’s walk through the stenciling passes - paying attention to the rendering of the pink shirt – I’m showing the albedo from the base pass here to make it easy to spot.



```
Batch 211 // drawIndex(0xcf, 0x201b7c4e60,
Batch 212 // drawIndex(0x1218, 0x2025f2c00
Batch 213 // drawIndex(0x91b, 0x2018c7e00
Batch 214 // drawIndex(0x43b, 0x20244c200
Batch 215 // drawIndex(0x1e9, 0x201b7c300
Batch 216 // drawIndex(0x54, 0x2022f2b000,
Batch 217 // drawIndex(0xed, 0x2022790000,
Batch 218 // drawIndex(0x1e, 0x2015b77e6c,
Batch 219 // drawIndex(0x18, 0x2011e2400c,
Batch 220 // drawIndex(0x18, 0x2011e2400c,
Batch 221 // drawIndex(0x252, 0x201b7c180
Batch 222 // drawIndex(0x2e2, 0x201b7c100
Batch 223 // drawIndex(0x480, 0x20245cf000
Batch 224 // drawIndex(0x1686, 0x202bb340
Batch 225 // drawIndex(0x1686, 0x202bb340
Batch 226 // drawIndex(0x1686, 0x202bb340
Batch 227 // drawIndex(0x1686, 0x202bb340
Batch 228 // drawIndex(0x5d06, 0x20103300
Batch 229 // drawIndex(0x2e80, 0x20111900
Batch 230 // drawIndex(0x1230, 0x2010b5bf9
Batch 231 // drawIndex(0x4b6, 0x201260c78
Batch 232 // drawIndex(0xc, 0x2025824000,
Batch 233 // drawIndex(0x24, 0x202ace7084,
Batch 234 // drawIndex(0x18, 0x202b5ec392,
Batch 235 // drawIndex(0x24, 0x202ace7084,
Batch 236 // drawIndex(0x18, 0x202a7d3900,
Batch 237 // drawIndex(0x18, 0x202a7d3900,
Batch 238 // drawIndex(0x1c6e, 0x20126280
Batch 239 // drawIndex(0x173a, 0x20126125
Batch 240 // drawIndex(0x243c, 0x20126237
Batch 241 // drawIndex(0x1062, 0x2010b737
Batch 242 // drawIndex(0x2d4e, 0x2010b6a2
Batch 243 // drawIndex(0x1692, 0x201260f89
Batch 244 // drawIndex(0x1c38, 0x2010b61c
Batch 245 // drawIndexIndirect(0xdc, kShade
Batch 246 // drawIndexIndirect(0x910, kShad
Batch 247 // drawIndexIndirect(0x2274, kSha
Batch 248 // drawIndexIndirect(0x1270, kSha
Batch 249 // drawIndexIndirect(0x1338, kSha
Batch 250 // drawIndexIndirect(0x1dc4, kSha
Batch 251 // drawIndexIndirect(0x1e8c, kSha
Batch 252 // drawIndexIndirect(0x1658, kSha
```

```
Graphics: DCB
Batch 238 // drawIndex(0x1c6e, 0x201
    0x30001b77e0 setStencil(0x1beff00)
    0x30001b77f0 setBlendColor(1.0000
    0x30001b7808 setStencil(0x1beff06)
    0x30001b7818 setUserDataRegion(
    0x30001b7838 setUserDataRegion(
    0x30001b7858 setUserDataRegion(
    0x30001b7878 setUserDataRegion(
    0x30001b7898 setUserDataRegion(
    0x30001b78b8 updateVsShader(Ver
    0x30001b792c prefetchIntoL2(0x20
    0x30001b7948 updatePsShader(Pix
    0x30001b79e8 prefetchIntoL2(0x20
    0x30001b7a04 setPointerInUserData
    0x30001b7a14 setPointerInUserData
    0x30001b7a24 setPointerInUserData
    0x30001b7a34 setPointerInUserData
    0x30001b7a44 setPointerInUserData
    0x30001b7a54 setPsShaderUsage(
    0x30001b7a6c drawIndex(0x1c6e, 0
Batch 239 // drawIndex(0x173a, 0x201
    0x30001b7a94 setStencil(0x1beff00)
    0x30001b7aa4 setBlendColor(1.000
```

Graphics, Batch 238

Resource	Name
Index Buffer 114	
Depth Render Target 0	SceneDep
Render Targets	
Render Target 0	SceneColc
Render Target 1	RayTracec
Render Target 2	
Render Target 3	
Render Target 4	GBufferB,
Render Target 5	DistanceFi
VS Stage	
Fetch Shader 27	
Vertex Shader 26	
Buffer 620	
Buffer 621	
Buffer 20	
Buffer 301	

Viewing: Replay at Graphics, Batch 238

Resource	Name	P...	Width	Hei
Depth Render Target 0	SceneDepthZ, Sc...		1920	
Depth Render Target 1			6144	
Depth Render Target 2			1920	

Resource	Depth Render Target 0	Z Format	32Float		
Width	1920	Height	1080	Stencil Format	8
Slice Range	Base = 0, Last = 0	Slice Index	0	HTILE Enabled	True
Name	SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, Scene...				

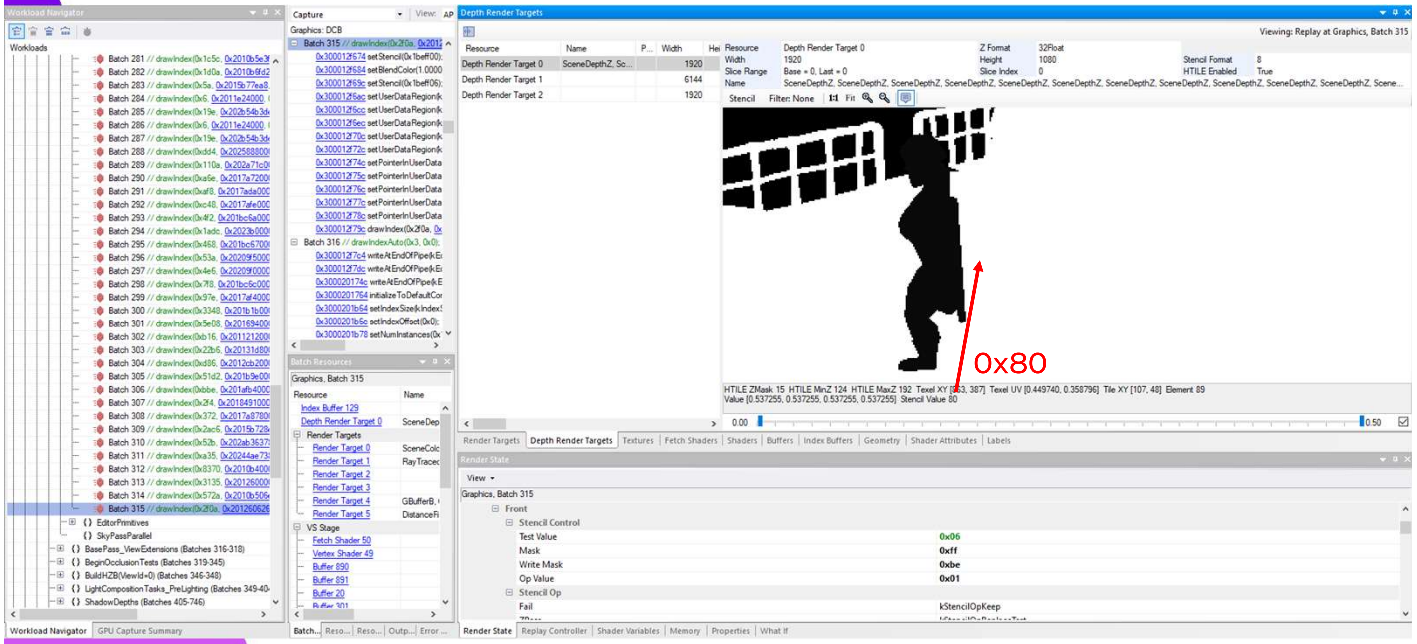
```
HTILE ZMask 15  HTILE MinZ 125  HTILE MaxZ 127  Texel XY [848, 366]  Texel UV [0.441927, 0.339352]  Tile XY [106, 45]  Element 4E
Value [0.305882, 0.305882, 0.305882, 0.305882]  Stencil Value 06
```

0.00 0.50

Graphics, Batch 238

Front	
Stencil Control	
Test Value	0x06
Mask	0xff
Write Mask	0xbe
Op Value	0x01
Stencil Op	
Fail	kStencilOpKeep

And in Razor you can see in this particular shot that the stencil bits are set and the shirt writes out 0x06 for fragments that pass the depth test


Workload Navigator
Workloads
Capture
View: AP
Depth Render Targets
Batch Resources
Render State
View
Workload Navigator | GPU Capture Summary
Batch... | Reso... | Reso... | Outp... | Error ...
Render Targets | Depth Render Targets | Textures | Fetch Shaders | Shaders | Buffers | Index Buffers | Geometry | Shader Attributes | Labels
Render State | Replay Controller | Shader Variables | Memory | Properties | What If
0x80

```
Batch 281 // drawIndex(0x1c5c, 0x2010b5e3f
Batch 282 // drawIndex(0x1d0a, 0x2010b6fd2
Batch 283 // drawIndex(0x5a, 0x2015b77ea8,
Batch 284 // drawIndex(0x6, 0x2011e24000, (
Batch 285 // drawIndex(0x19e, 0x202b54b3d
Batch 286 // drawIndex(0x6, 0x2011e24000, (
Batch 287 // drawIndex(0x19e, 0x202b54b3d
Batch 288 // drawIndex(0xdd4, 0x202588800
Batch 289 // drawIndex(0x110a, 0x202a71c0
Batch 290 // drawIndex(0xa6e, 0x2017a7200
Batch 291 // drawIndex(0xaf8, 0x2017ada000
Batch 292 // drawIndex(0xc48, 0x2017afe000
Batch 293 // drawIndex(0x4f2, 0x201bc6a000
Batch 294 // drawIndex(0x1adc, 0x2023b000
Batch 295 // drawIndex(0x468, 0x201bc6700
Batch 296 // drawIndex(0x53a, 0x20209f5000
Batch 297 // drawIndex(0x4e6, 0x20209f0000
Batch 298 // drawIndex(0x7f8, 0x201bc6c000
Batch 299 // drawIndex(0x97e, 0x2017af4000
Batch 300 // drawIndex(0x3348, 0x201b1b00
Batch 301 // drawIndex(0x5e08, 0x20169400
Batch 302 // drawIndex(0xb16, 0x20112120
Batch 303 // drawIndex(0x22b6, 0x20131d80
Batch 304 // drawIndex(0xd86, 0x2012cb200
Batch 305 // drawIndex(0x51d2, 0x201b9e00
Batch 306 // drawIndex(0xbbe, 0x201afb4000
Batch 307 // drawIndex(0x2f4, 0x2018491000
Batch 308 // drawIndex(0x372, 0x2017a8780
Batch 309 // drawIndex(0x2ac6, 0x2015b728
Batch 310 // drawIndex(0x52b, 0x202ab3637
Batch 311 // drawIndex(0xa35, 0x20244ae73
Batch 312 // drawIndex(0x8370, 0x2010b400
Batch 313 // drawIndex(0x3135, 0x2012600
Batch 314 // drawIndex(0x572a, 0x2010b506
Batch 315 // drawIndex(0x2f0a, 0x201260626
{ } EditorPrimitives
{ } SkyPassParallel
{ } BasePass_ViewExtensions (Batches 316-318)
{ } BeginOcclusionTests (Batches 319-345)
{ } BuildHZB(ViewId=0) (Batches 346-348)
{ } LightCompositionTasks_PreLighting (Batches 349-40
{ } ShadowDepths (Batches 405-746)
```

```
Graphics: DCB
Batch 315 // drawIndex(0x2f0a, 0x2012
    0x300012f674 setStencil(0x1beff00);
    0x300012f684 setBlendColor(1.0000
    0x300012f69c setStencil(0x1beff06);
    0x300012f6ac setUserDataRegion(k
    0x300012f6cc setUserDataRegion(k
    0x300012f6ec setUserDataRegion(k
    0x300012f70c setUserDataRegion(k
    0x300012f72c setUserDataRegion(k
    0x300012f74c setPointerInUserData
    0x300012f75c setPointerInUserData
    0x300012f76c setPointerInUserData
    0x300012f77c setPointerInUserData
    0x300012f78c setPointerInUserData
    0x300012f79c drawIndex(0x2f0a, 0x
Batch 316 // drawIndexAuto(0x3, 0x0);
    0x300012f7c4 writeAtEndOfPipe(kE
    0x300012f7dc writeAtEndOfPipe(kE
    0x300020174c writeAtEndOfPipe(kE
    0x3000201764 initializeToDefaultCo
    0x3000201b64 setIndexSize(kIndex
    0x3000201b6c setIndexOffset(0x0);
    0x3000201b78 setNumInstances(0x
```

Graphics, Batch 315

Resource	Name
Index Buffer 129	
Depth Render Target 0	SceneDep
Render Targets	
Render Target 0	SceneColc
Render Target 1	RayTracec
Render Target 2	
Render Target 3	
Render Target 4	GBufferB,
Render Target 5	DistanceFi
VS Stage	
Fetch Shader 50	
Vertex Shader 49	
Buffer 890	
Buffer 891	
Buffer 20	
Buffer 301	

Viewing: Replay at Graphics, Batch 315

Resource	Name	P...	Width	Hei
Depth Render Target 0	SceneDepthZ, Sc...		1920	
Depth Render Target 1			6144	
Depth Render Target 2			1920	

Resource	Depth Render Target 0	Z Format	32Float		
Width	1920	Height	1080	Stencil Format	8
Slice Range	Base = 0, Last = 0	Slice Index	0	HTILE Enabled	True
Name	SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, Scene...				

Stencil Filter: None 1:1 Fit

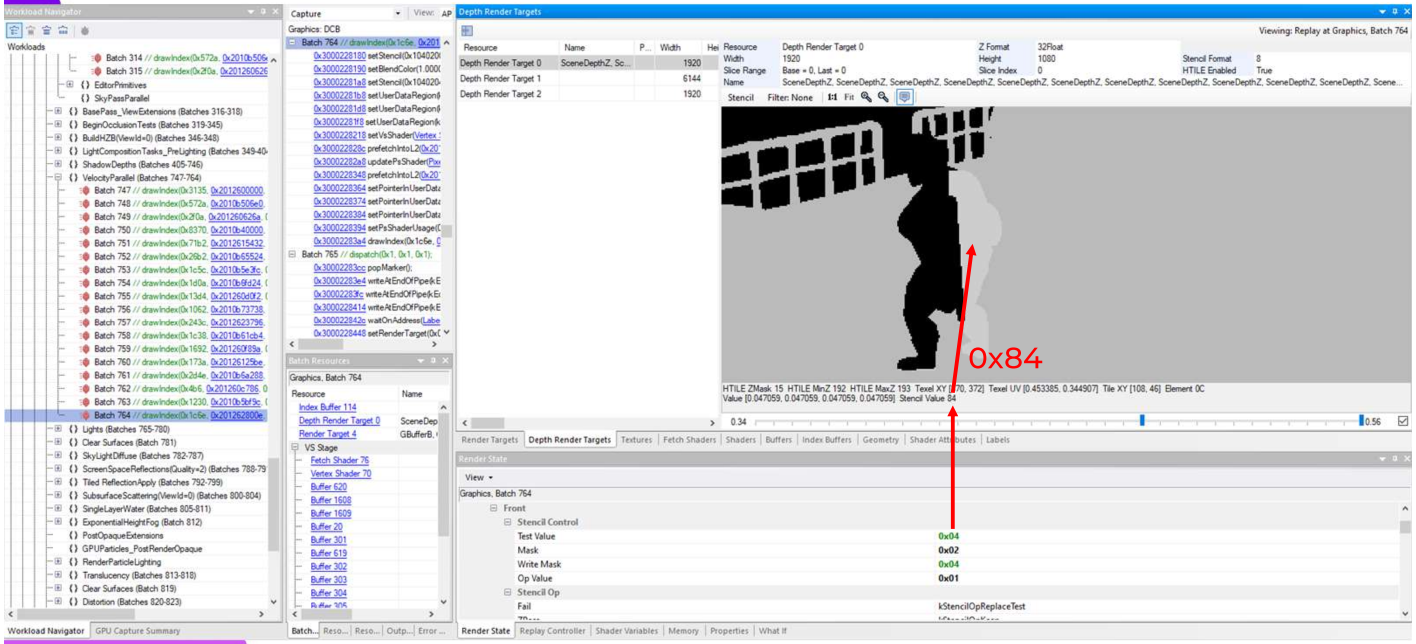
```
HTILE ZMask 15  HTILE MinZ 124  HTILE MaxZ 192  Texel XY [863, 387]  Texel UV [0.449740, 0.358796]  Tile XY [107, 48]  Element 89
Value [0.537255, 0.537255, 0.537255, 0.537255]  Stencil Value 80
```

0.00 0.50

Graphics, Batch 315

Front	
Stencil Control	
Test Value	0x06
Mask	0xff
Write Mask	0xbe
Op Value	0x01
Stencil Op	
Fail	kStencilOpKeep

At the end of the base pass, you can see that the stencil is filled out with the remaining draws. The non-character stencil is set to 0x80, you can see a nice outline of the environment from the Receives Decals bit being set.


Depth Render Targets
Viewing: Replay at Graphics, Batch 764
0x84
Render Targets | Depth Render Targets | Textures | Fetch Shaders | Shaders | Buffers | Index Buffers | Geometry | Shader Attributes | Labels
Render State | Replay Controller | Shader Variables | Memory | Properties | What If

Resource	Name	P...	Width
Depth Render Target 0	SceneDepthZ, Sc...		1920
Depth Render Target 1			6144
Depth Render Target 2			1920

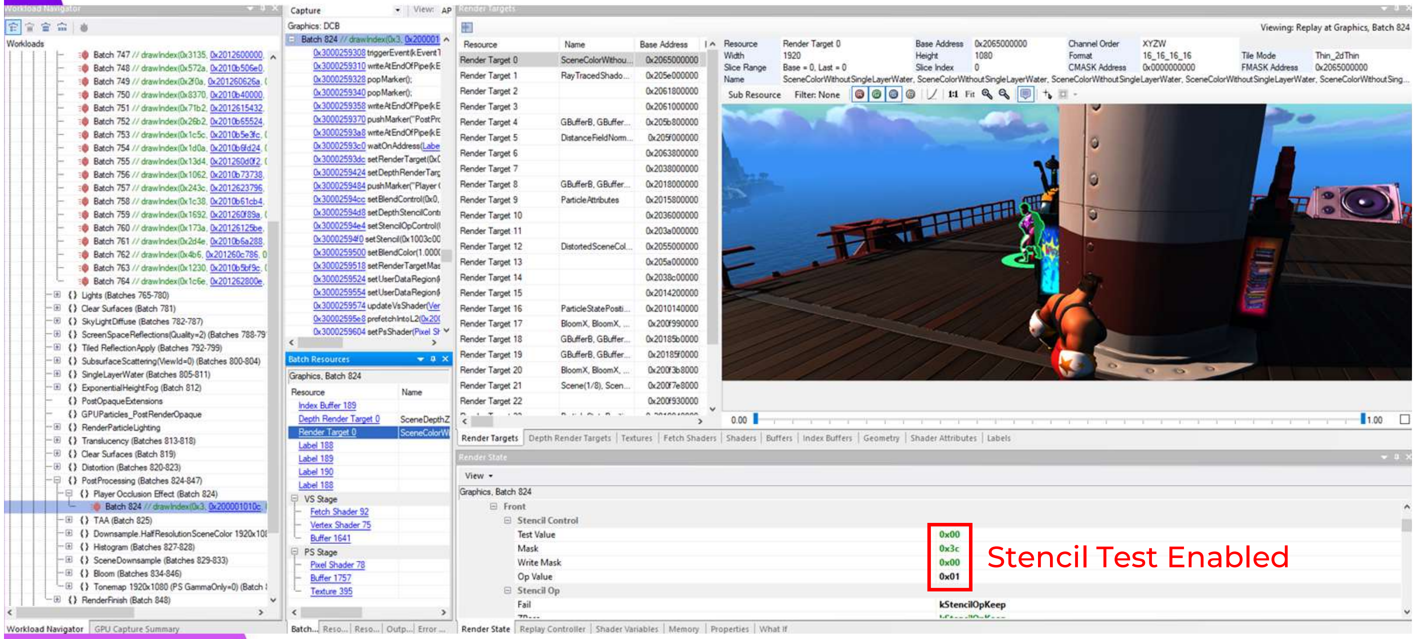
Resource	Depth Render Target 0	Z Format	32Float		
Width	1920	Height	1080	Stencil Format	8
Slice Range	Base = 0, Last = 0	Slice Index	0	HTILE Enabled	True
Name	SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, SceneDepthZ, Scene...				

```
HTILE ZMask 15  HTILE MinZ 192  HTILE MaxZ 193  Texel XY [170, 372]  Texel UV [0.453385, 0.344907]  Tile XY [108, 46]  Element 0C
Value [0.047059, 0.047059, 0.047059, 0.047059]  Stencil Value 84
```

Graphics, Batch 764

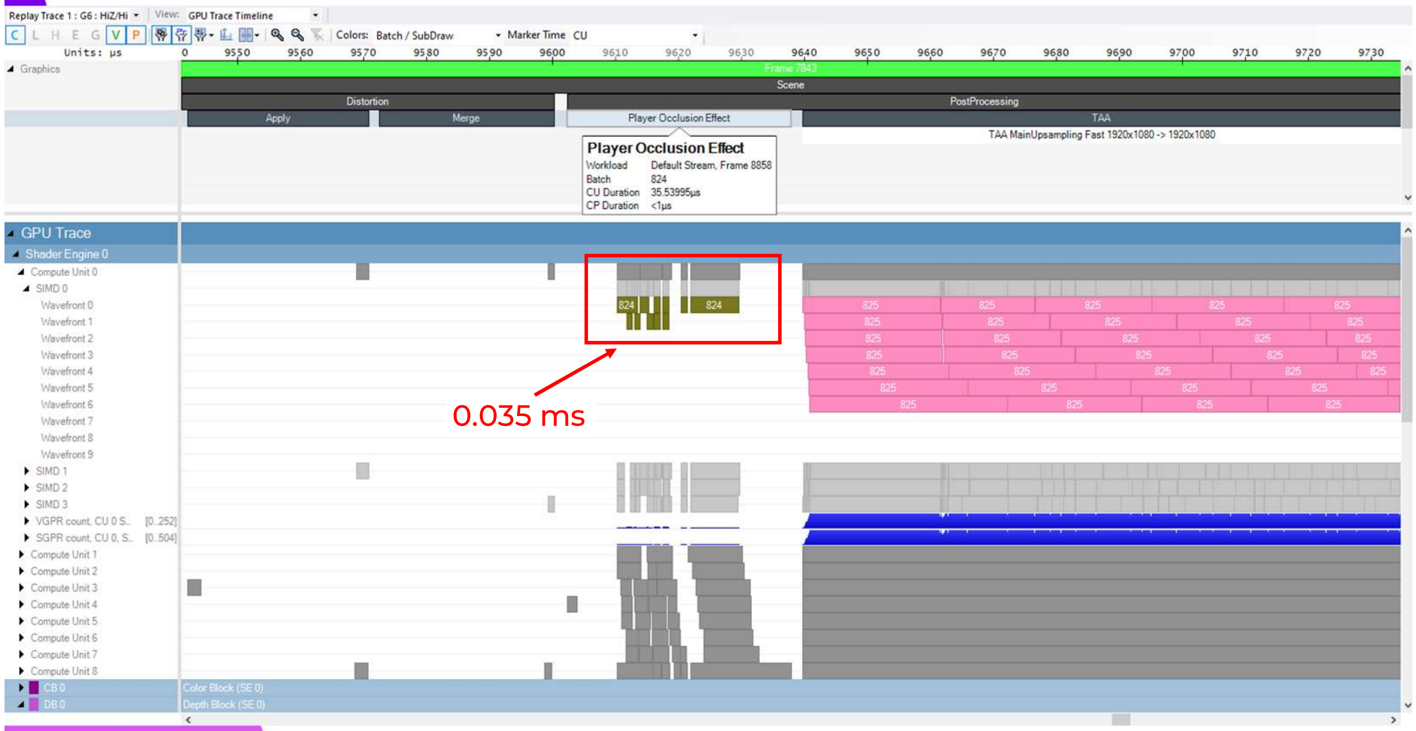
```
Front
  Stencil Control
    Test Value      0x04
    Mask            0x02
    Write Mask      0x04
    Op Value        0x01
  Stencil Op
    Fail            kStencilOpReplaceTest
```

Then, we also need to mark the fragments that fail the depth test to get the rest of the character silhouette, we handle this during velocity rendering when the characters are already going to render a second time. Stencil can be written on zfail which is what we do here. EarlyZ is handling writing to those stencil locations, but pixel shader waves are launched for the fragments that pass the ztest to write out to the velocity buffer. The occluded parts of the character now have 0x84. Note that any characters with this effect enabled must have occlusion queries disabled on them or else they will stop rendering entirely when fully occluded.


Graphics, Batch 824

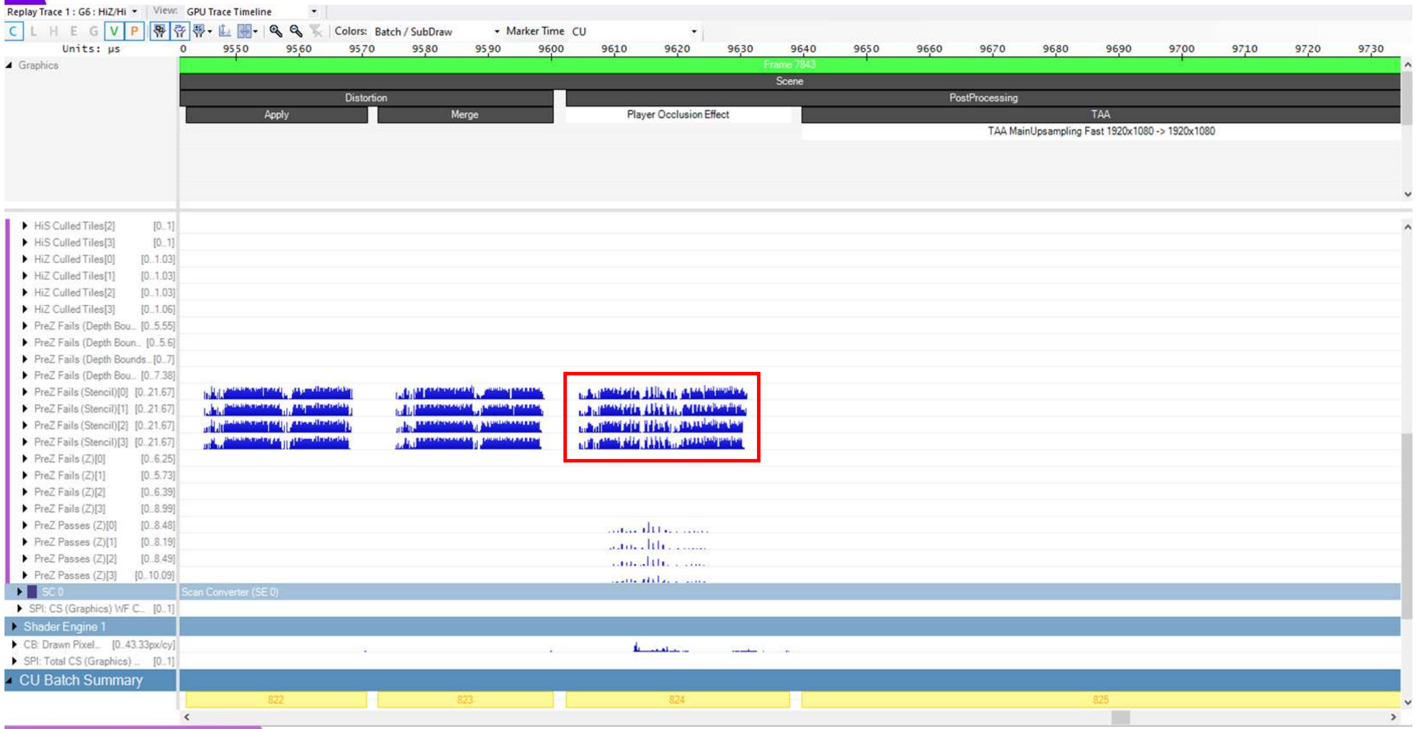
Front	
Stencil Control	
Test Value	0x00
Mask	0x3c
Write Mask	0x00
Op Value	0x01
Stencil Op	
Fail	kStencilOpKeep

Finally, the post process effect is run before TAA on the fragments that the player rendered to. The shader samples neighboring pixels to render outlines at the edge of the character, and the center sample determines the player color and if the occlusion tint should be applied. The shader has to be run before TAA or else the stencil positions will not be valid for the un-jittered camera matrix. However, this is actually ideal, because TAA provides nice anti-aliasing on the effect. Because we only render on the inside of the character silhouette, we do not fight against the TAA algorithm as the outline effectively moves with the depth values of the character.



Player Occlusion Effect
Workload Default Stream, Frame 8858
Batch 824
CU Duration 35.53995µs
CP Duration <1µs

To illustrate how efficient this is, here is the razor trace for that frame. You can see the occlusion effect only take 0.035 ms, with very few pixel shader waves actually running. I have worked on a number of projects that only use 1-2 bits of the stencil buffer, so I am very happy that we frequently use all 8 bits of the buffer in a single frame to cull workloads with EarlyZ.

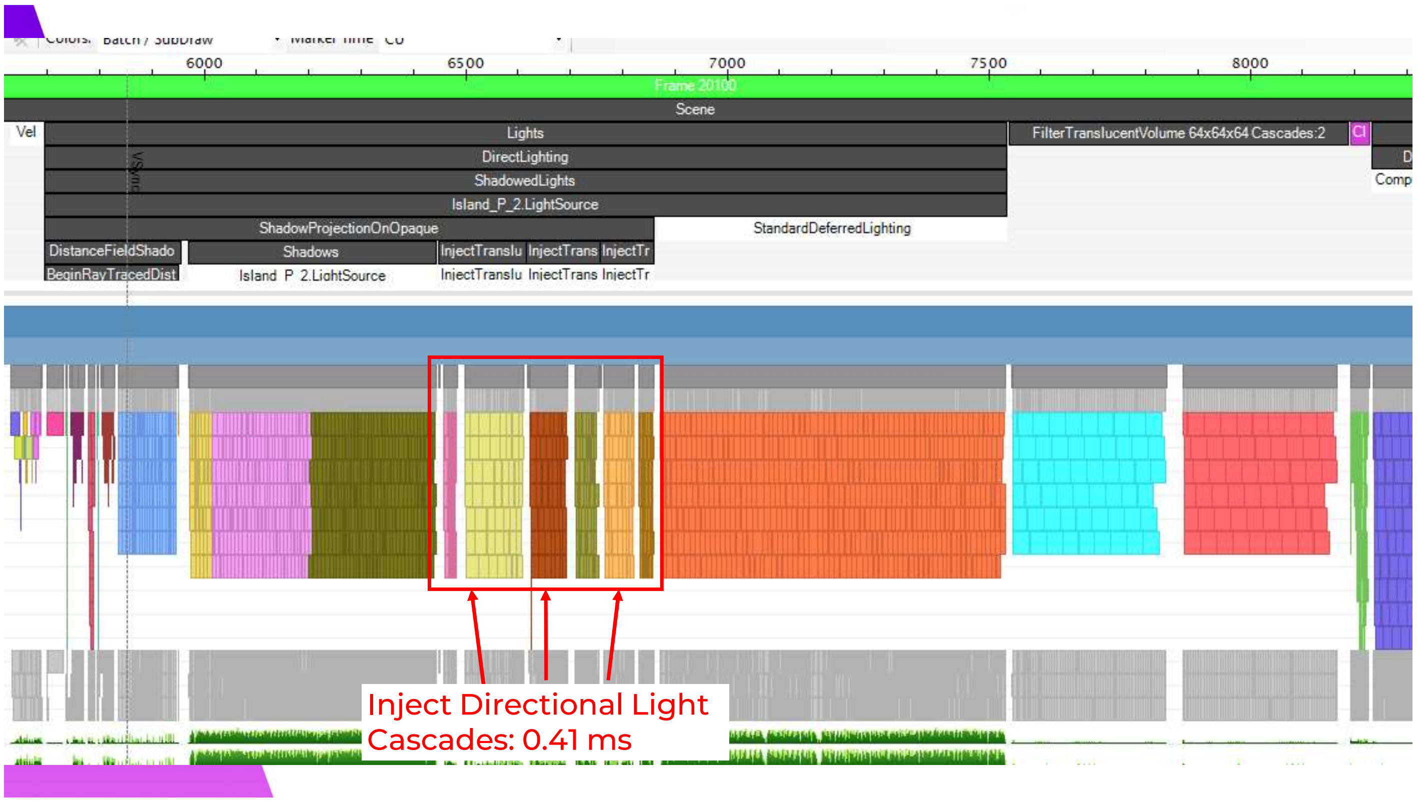

Replay Trace 1 : G6 : HiZ/Hi View: GPU Trace Timeline Colors: Batch / SubDraw Marker Time CU

Units: μs
0 9550 9560 9570 9580 9590 9600 9610 9620 9630 9640 9650 9660 9670 9680 9690 9700 9710 9720 9730
Graphics
Frame 7843
Scene
Distortion
PostProcessing
Apply
Merge
Player Occlusion Effect
TAA
TAA MainUpsampling Fast 1920x1080 -> 1920x1080
HiS Culled Tiles[2] [0..1]
HiS Culled Tiles[3] [0..1]
HiZ Culled Tiles[0] [0..1.03]
HiZ Culled Tiles[1] [0..1.03]
HiZ Culled Tiles[2] [0..1.03]
HiZ Culled Tiles[3] [0..1.06]
PreZ Fails (Depth Bou... [0..5.55]
PreZ Fails (Depth Boun... [0..5.6]
PreZ Fails (Depth Bounds... [0..7]
PreZ Fails (Depth Bou... [0..7.38]
PreZ Fails (Stencil)[0] [0..21.67]
PreZ Fails (Stencil)[1] [0..21.67]
PreZ Fails (Stencil)[2] [0..21.67]
PreZ Fails (Stencil)[3] [0..21.67]
PreZ Fails (Z)[0] [0..6.25]
PreZ Fails (Z)[1] [0..5.73]
PreZ Fails (Z)[2] [0..6.39]
PreZ Fails (Z)[3] [0..8.99]
PreZ Passes (Z)[0] [0..8.48]
PreZ Passes (Z)[1] [0..8.19]
PreZ Passes (Z)[2] [0..8.49]
PreZ Passes (Z)[3] [0..10.09]
SC 0
Scan Converter (SE 0)
SPI: CS (Graphics) WF C... [0..1]
Shader Engine 1
CB: Drawn Pixel... [0..43.33px/cy]
SPI: Total CS (Graphics) ... [0..1]
CU Batch Summary
822
823
824
825

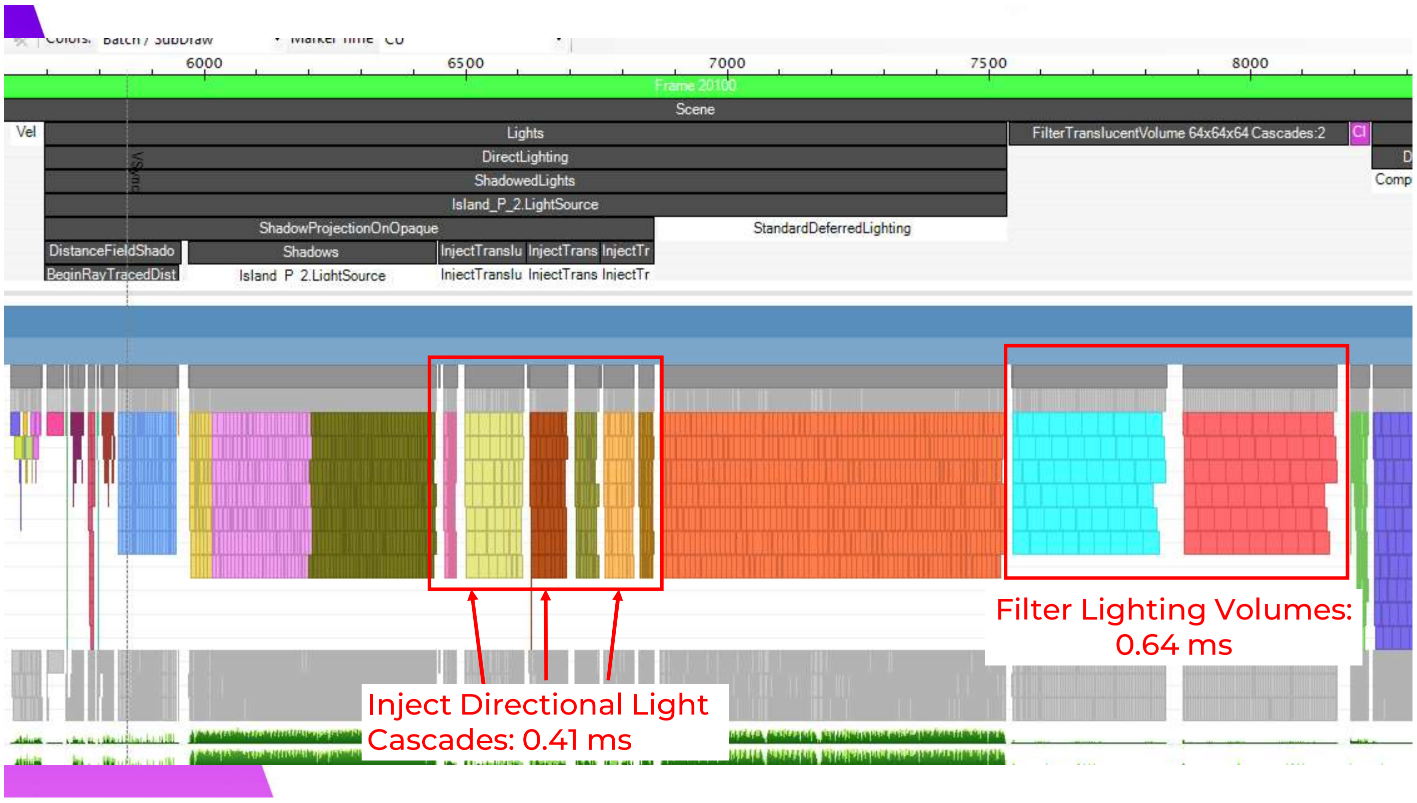
And here I have scrolled down to show that a large amount of this effect is just spent with EarlyZ rejecting the stencil bits. We have not configured HiStencil but theoretically this could be a use case that would benefit from it, but the time is so minimal that is not a priority and enabling HiStencil does have trade-offs with HiZ quality on GCN.

Translucent Lighting

In the spirit of removing unnecessary work from our frame, I wanted to change how lit translucency was handled for our game compared to what UE4 does by default.



Stock UE4 accumulates lighting in a volume texture and then translucent particles and meshes can read from it to cheaply apply shading. Each light must be injected – you can see that happening once for each shadow cascade on our sunlight. There are two cascades on the translucent lighting volume, so there are a total of 6 injections happening here. More injections have to occur if there are additional dynamic lights rendering in the bounds of the lighting volume.

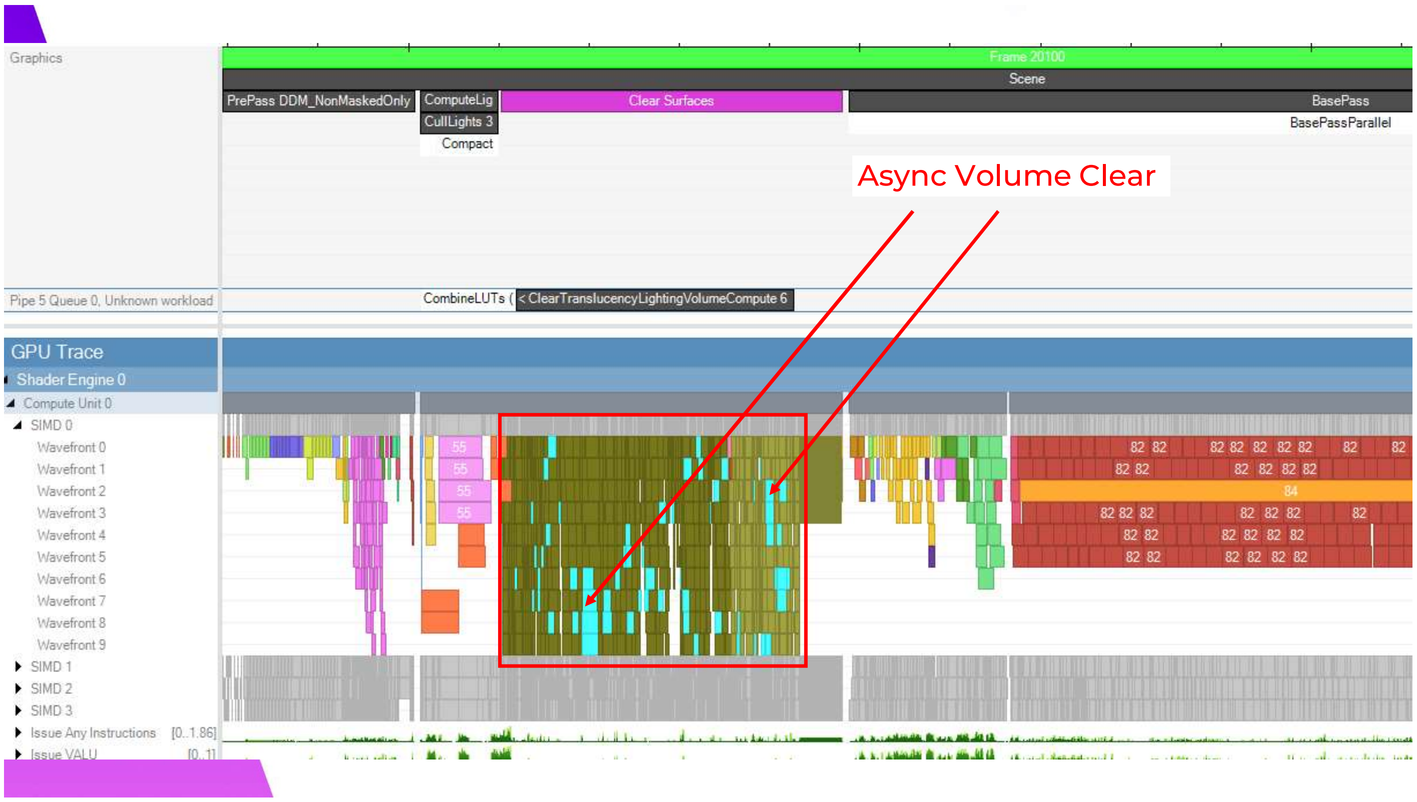


After lights are injected as part of the main lighting loop, the volume gets filtered to soften the shadows and make aliasing at the low resolution of the volume less noticeable. This takes 0.6 ms for two 64^3 lighting volume cascades.

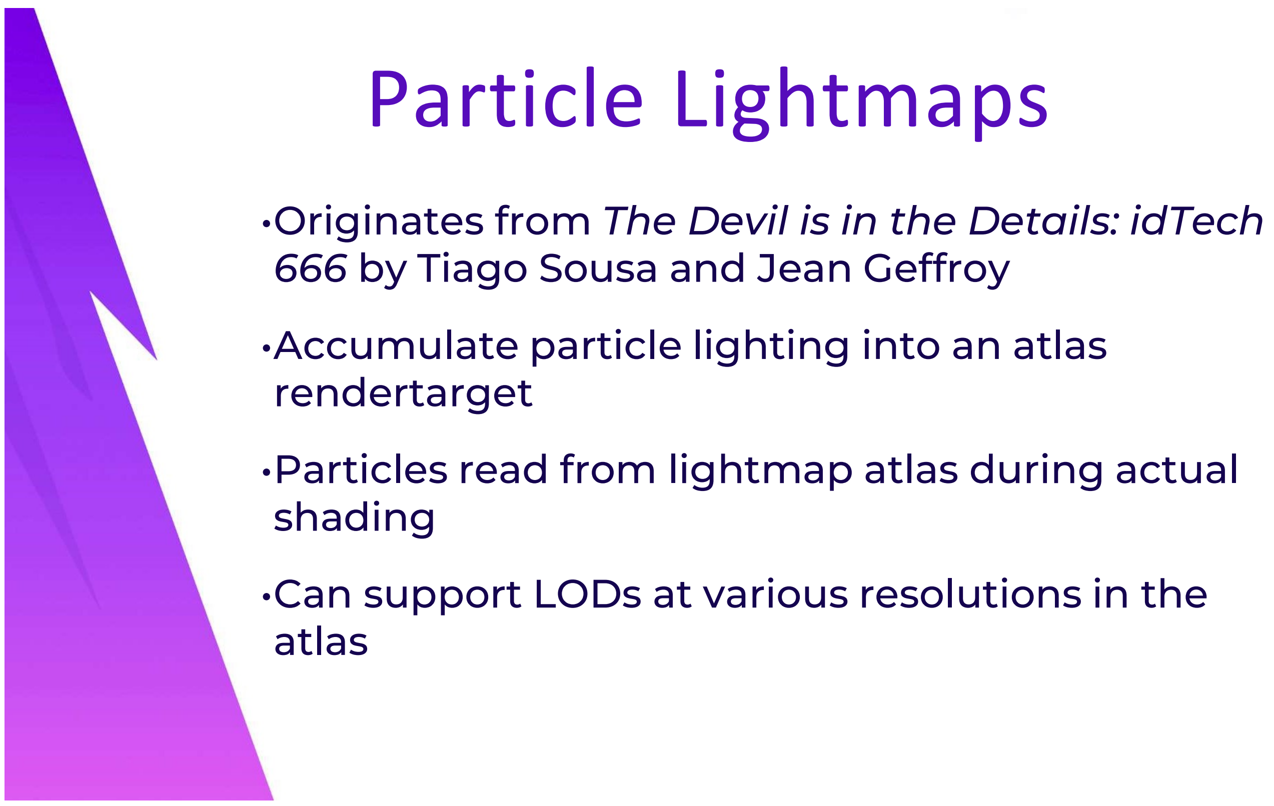
The design consequence here is that the system is designed to have a high cost on the number of lights present in a scene, but a very low cost per lit translucent pixel processed.

In total, the translucent volume is taking ~1 ms per frame to support particle lighting. The only option to disable this in the engine is to resort to only using per-pixel forward shading on any lit translucency, which is quite expensive on something like smoke particles.

This is a lot to be paying for something that frequently doesn’t contribute to the look of the final image if there aren’t actually any lit particles or translucency firing, and our game is a little goofy, we don’t have *that* much lit translucency. My goal was to eliminate all of these fixed costs from our frame.



The volume must also be cleared each frame, this happens in async compute, although you can see it's not behaving super well in this trace and it is overlapped with other DRAM heavy work in the surface clears. It can be effectively free with the right scheduling though.



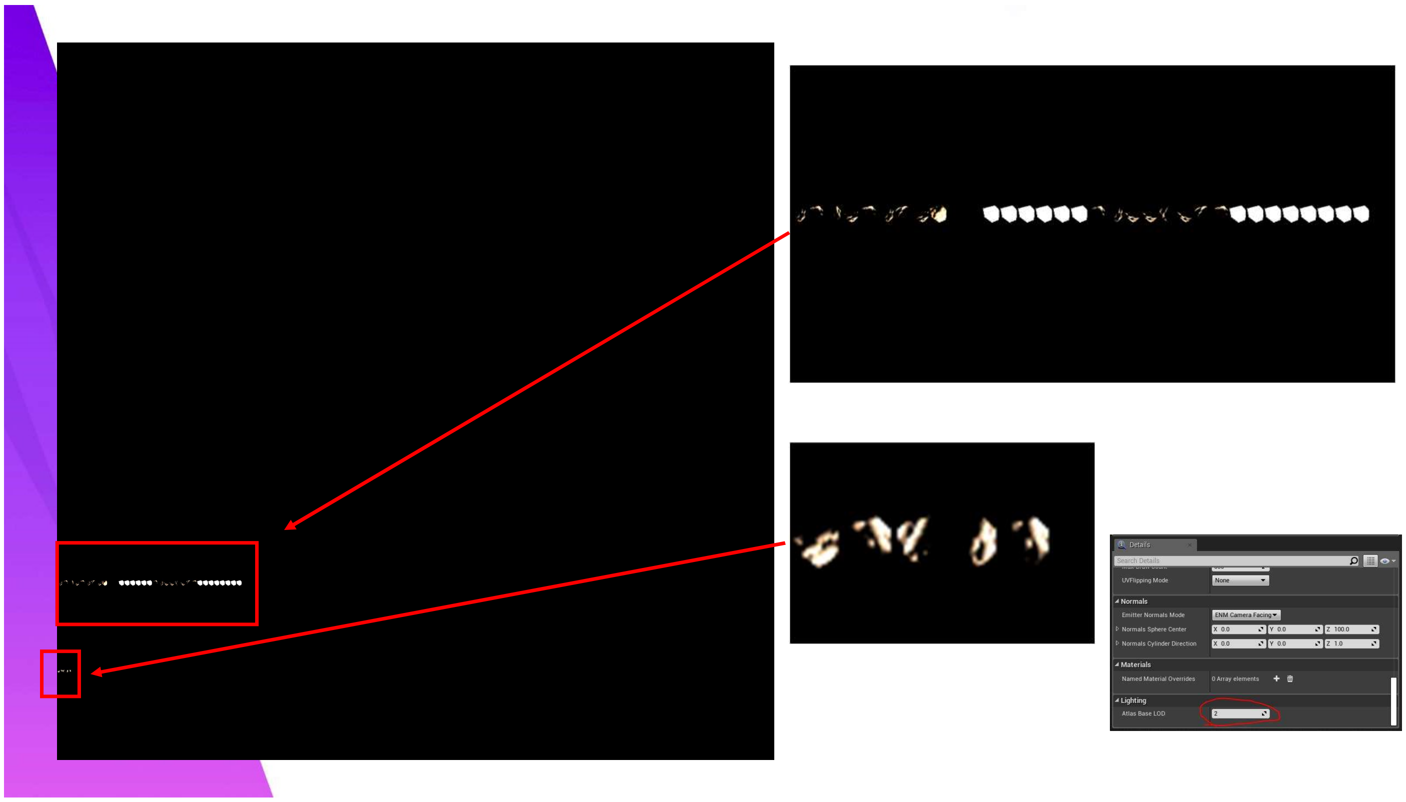
Particle Lightmaps

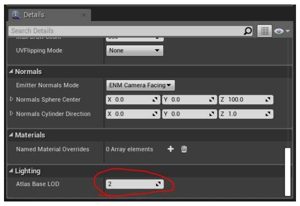
- Originates from *The Devil is in the Details: idTech 666* by Tiago Sousa and Jean Geffroy
- Accumulate particle lighting into an atlas rendertarget
- Particles read from lightmap atlas during actual shading
- Can support LODs at various resolutions in the atlas

In pursuit of this goal, I worked with my colleague Rusty Swain to implement a system similar to what was used in DOOM 2016, a particle lightmap system that particles accumulate their lighting into at less than the final resolution of the particles on screen, but still higher quality than what you would get from just shading the particle vertices.

We select an LOD resolution for an effect based on distance from camera, and Doom could even reduce the shading frequency in the time dimension or have the atlas update in async compute. We only do LODs and update the atlas every frame for simplicity, but if art direction started pushing more lit particles sprites we would definitely look into additional optimization.

http://advances.realtimerendering.com/s2016/Siggraph2016_idTech6.pdf





Details

UVFlipping Mode: None

Normals

Emitter Normals Mode	ENM Camera Facing		
Normals Sphere Center	X 0.0	Y 0.0	Z 100.0
Normals Cylinder Direction	X 0.0	Y 0.0	Z 1.0

Materials

Named Material Overrides: 0 Array elements

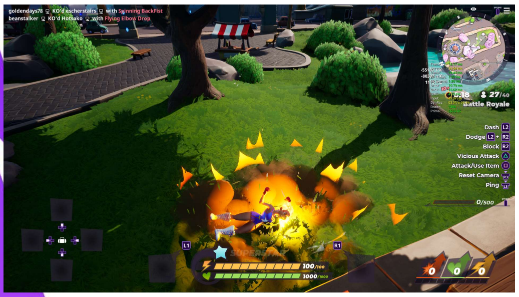
Lighting

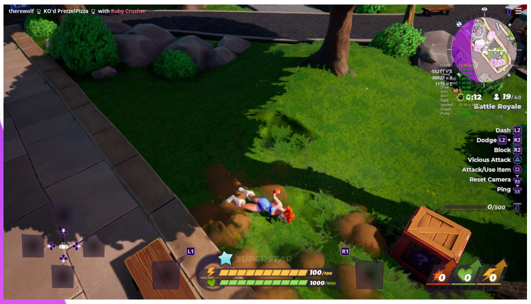
Atlas Base LOD: 2

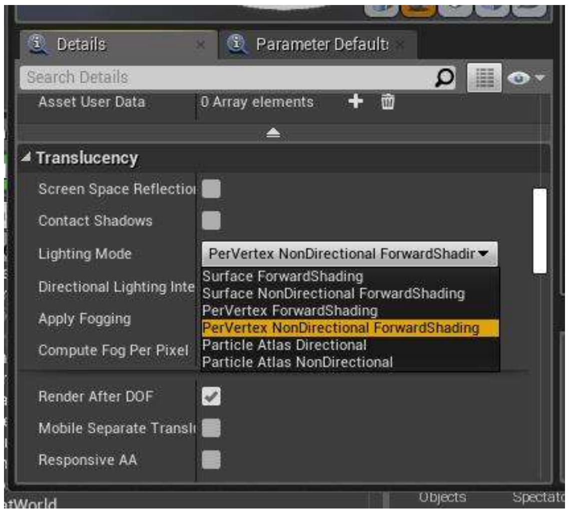
Here you can see a particle effect going into our atlas - this takes the sprite's position and normal map and calculates a lighting result at each texel. This is some smoke coming off of an impact crater of another rumbler that landed nearby me in the match. Updating this particular effect in the atlas took 20 microseconds, and gets better results than the translucent lighting volume. Having lit particles render to that atlas every frame does simplify managing allocations within it because the particles can be full repacked at each LOD every frame.

EXTRA NOTE:
Artists can tune a base LOD bias for if they want it to bet more texels per quad, so by default sprites will not take up more than 32 x 32 pixels for lighting when the emitter is close to the camera. LOD0 takes 128x128 pixels, but is almost never used by our artists. Emitters will drop an LOD level for ever 25% reduction in estimated screen size of the effect so there is dynamic LOD selection occurring for us within the atlas.

Mesh Lighting







Asset User Data	0 Array elements
Translucency	
Screen Space Reflectio	☐
Contact Shadows	☐
Lighting Mode	PerVertex NonDirectional ForwardShadir
Directional Lighting Inte	
Apply Fogging	
Compute Fog Per Pixel	
Render After DOF	✓
Mobile Separate Transl	☐
Responsive AA	☐

Surface ForwardShading
Surface NonDirectional ForwardShading
PerVertex ForwardShading
PerVertex NonDirectional ForwardShading
Particle Atlas Directional
Particle Atlas NonDirectional

We only support sprite based particles in the lighting atlas. Meshes need to go down a forward shading path – it's okay for them to receive per-pixel forward shading when needed, but we added support for vertex shading which is frequently used on mesh particles in our game. We also added support for NonDirectional or "Wrap" shading in both the per-pixel and per-vertex paths.

UE4 has a clustered lighting data structure for forward shading – the vertex shading path reads the light data at the cluster it intersects and accumulates it into a spherical harmonic. This is an old trick at this point, but this allows us to keep per-pixel normal map variation on a vertex lit object, which fits really nicely with our art style.

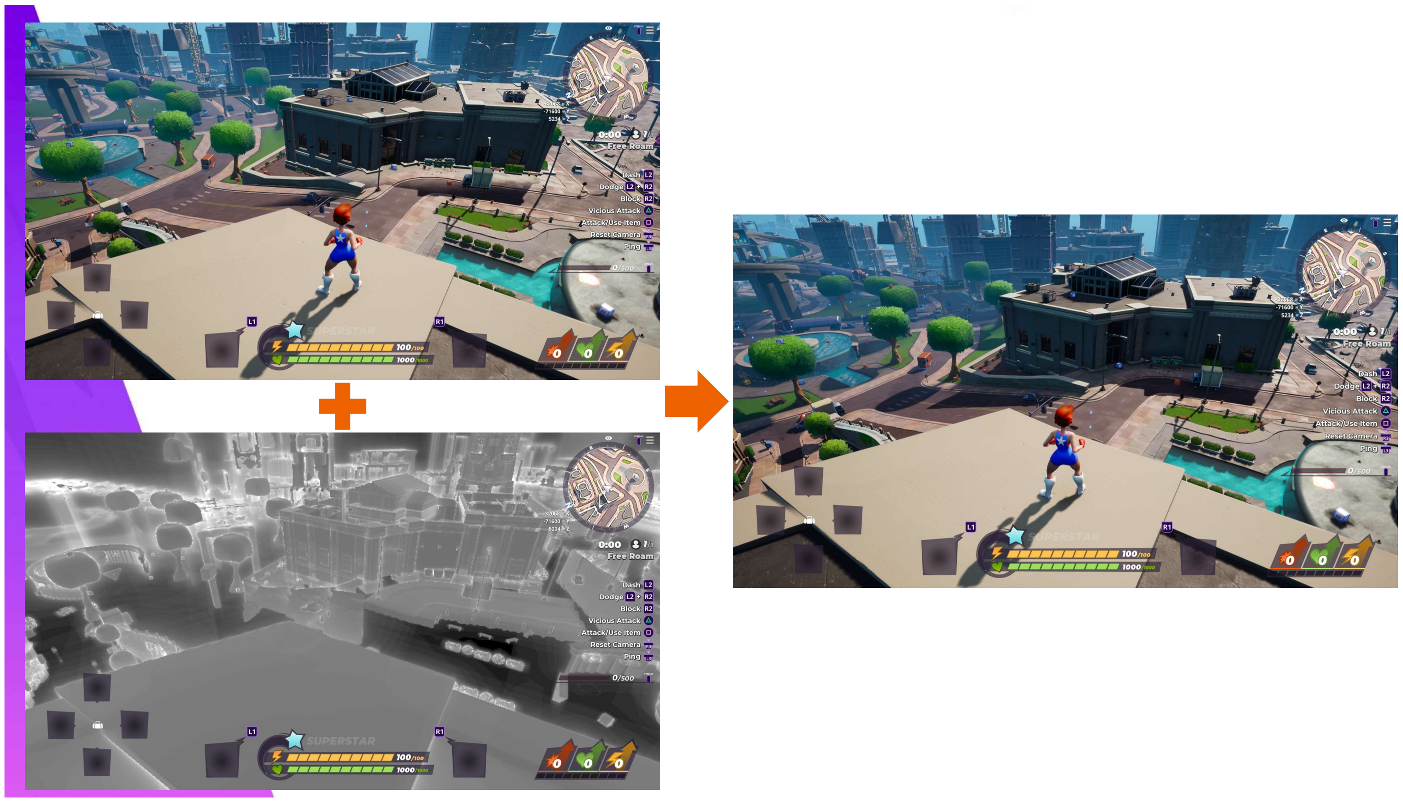
In most frames we see 0.5 ms improvement in frame time, and in many frames we see a full 1 ms better performance than using the translucent lighting volume path.

Part 2:
Reactive Optimizations

Now, while all that work was proactively done, we also did a lot to respond to performance problems we did not fully expect until content reached the limits of what we could do.

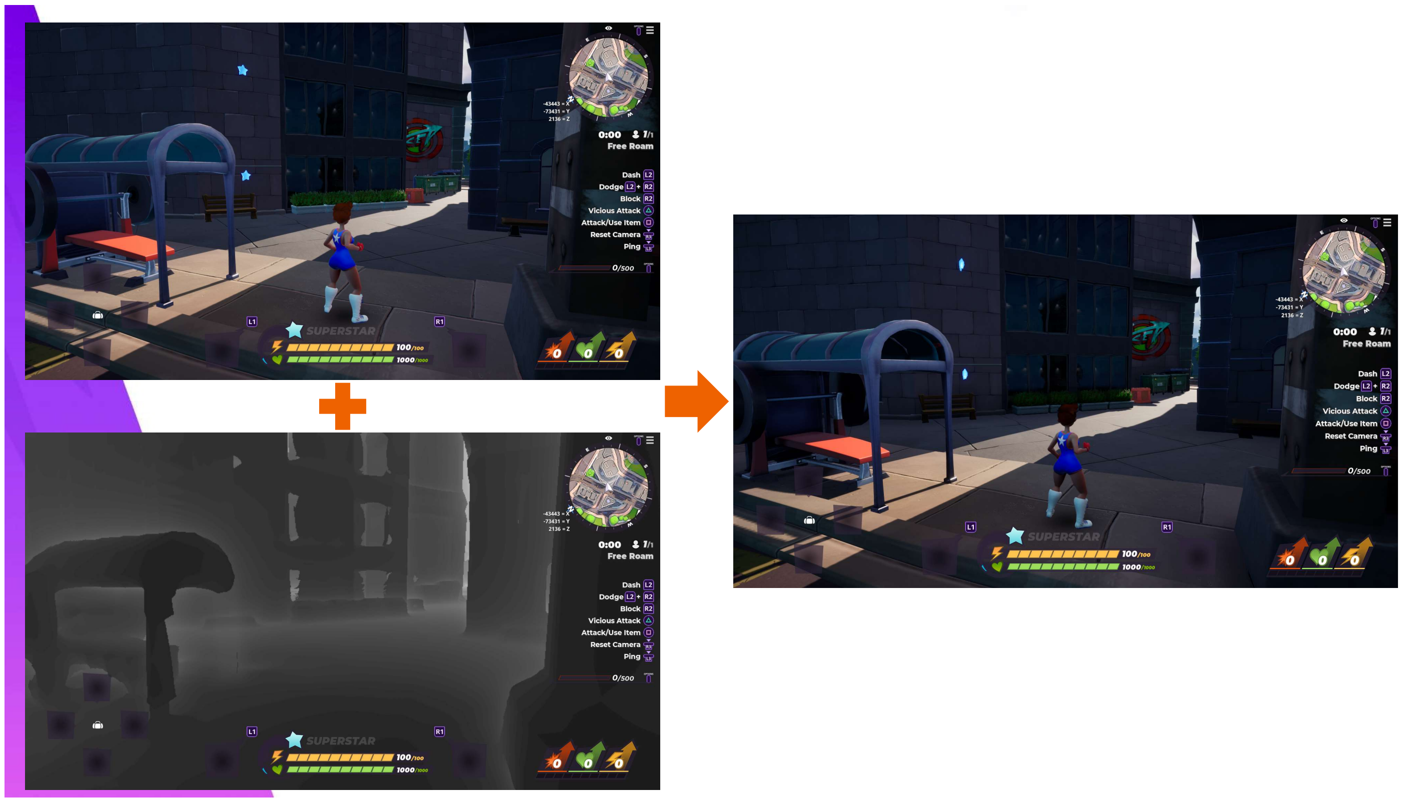
Background: Lighting

I need to set the stage a little first with some decisions we made early on that would impact our performance later, specifically with how we built our environment

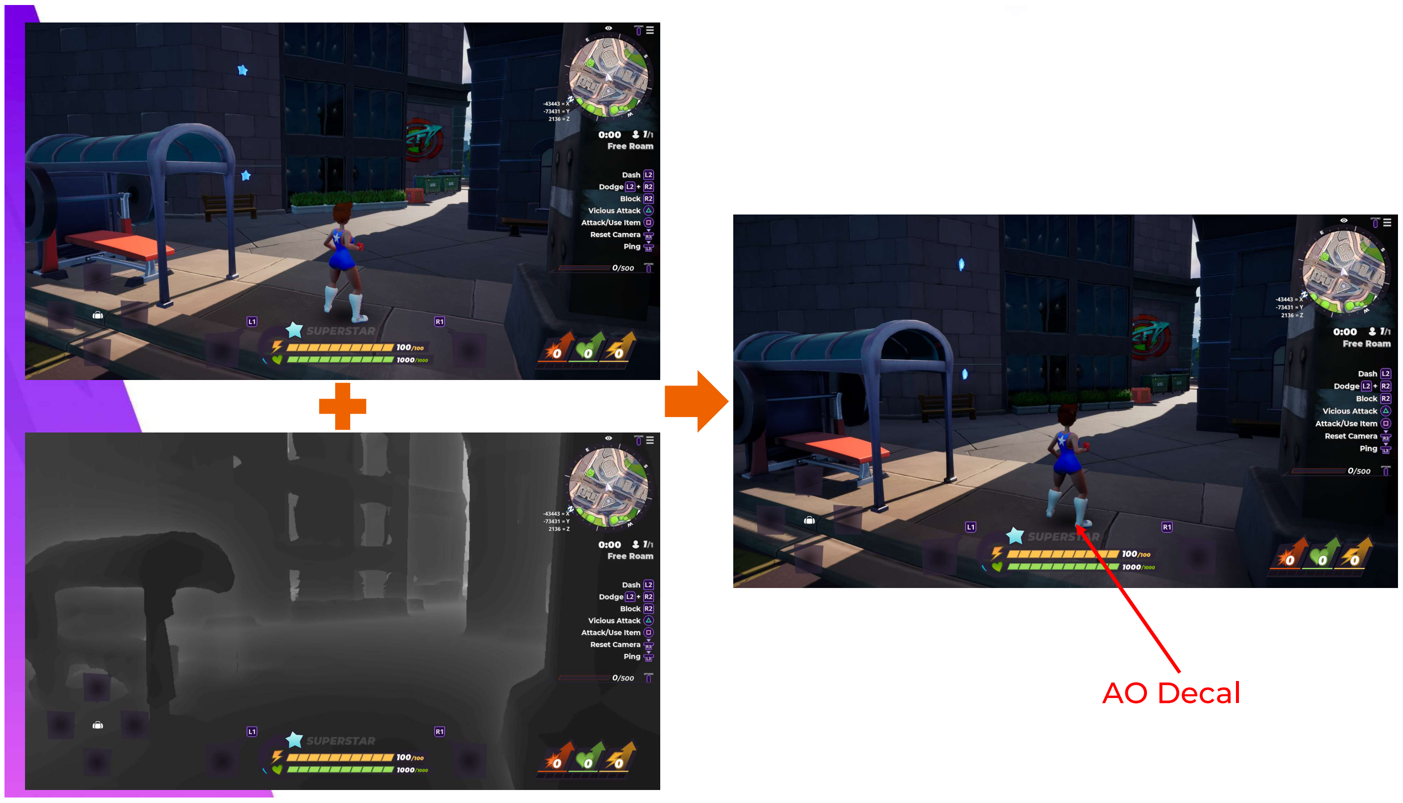


Early on we decided to lean in heavily on Unreal's distance field features. Our medium to far shadows are all traced against mesh distance fields that are calculated offline for static meshes, which allocate into an shared atlas texture accessed by compute shaders on the GPU. In the images I've included here, you can see the world without the distance field shadows in the top left, and a visualization of the per-object distance fields in the bottom left, together which make the final shot in the right with far shadows.

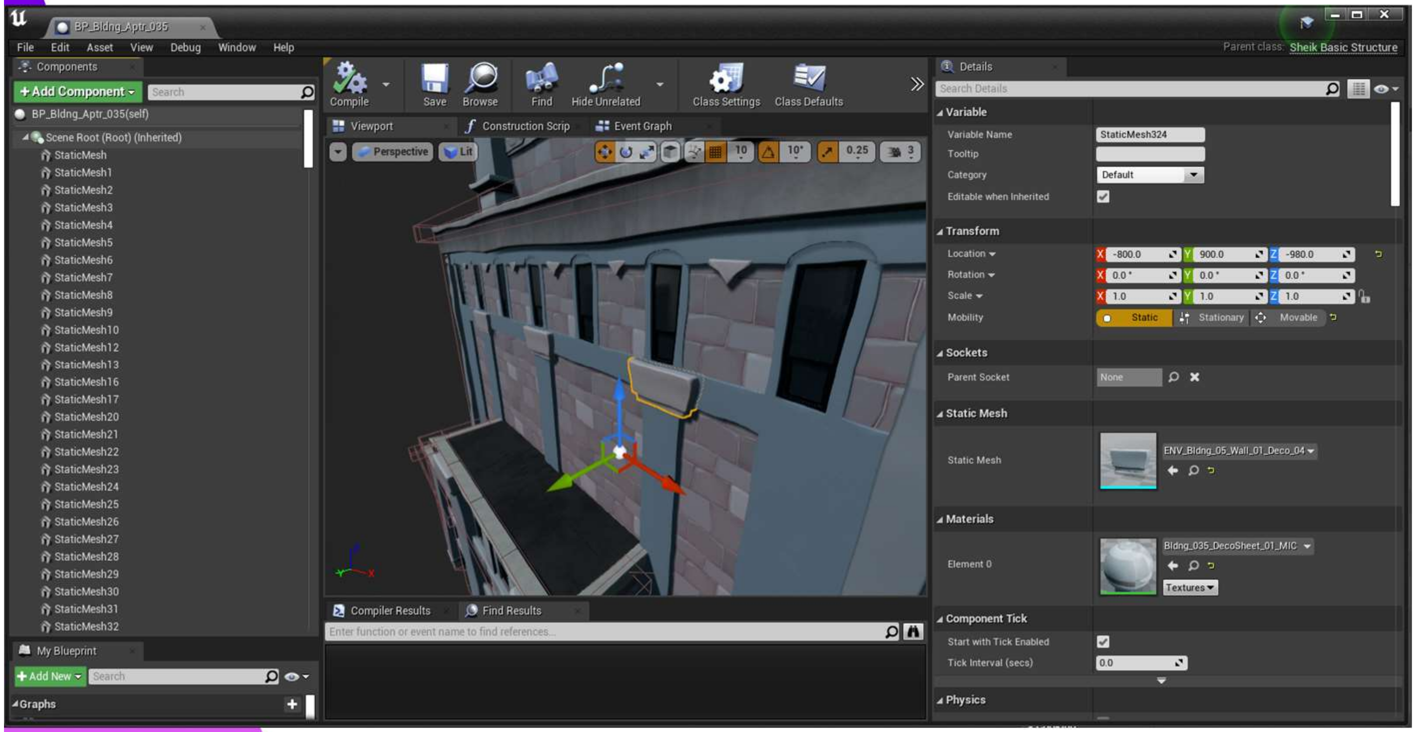
EXTRA NOTE:
The terrain supports using a heightfield representation, but we are able to get away with disabling heightfield shadows on last-gen consoles because our artists and designers almost always choose to place decorator skirt meshes near changes in elevation like cliffs. The artists like it aesthetically and the designers like it because the game's traversal system works really well with wall climbing.



We also use distance field ambient occlusion, which traces against a lower quality clip map that composites the individual distance fields into a global distance field representation. This clipmap can be extremely low quality but DFAO still successfully adds a lot of definition to our shadowed areas. The only other ambient occlusion comes from a channel in the Gbuffer that artists can write out to from material graphs. This AO is pretty important to our look because we don’t have any global illumination baked into lightmaps – just an ambient skylight that has a diffuse component (spherical harmonic at runtime) and a detailed specular environment map.



One minor detail I'll point out while we're on the subject. The ambient shadowing under the characters is just a simple decal that modifies only the ambient occlusion gbuffer. Super simple, but didn't work out of the box in Unreal and our designers really appreciate the characters always having some sort of shadow anchoring them to the ground.



Variable

Variable Name: StaticMesh324
Tooltip:
Category: Default
Editable when Inherited: ✓

Transform

Location: X -800.0 Y 900.0 Z -980.0
Rotation: X 0.0° Y 0.0° Z 0.0°
Scale: X 1.0 Y 1.0 Z 1.0
Mobility: [✓ Static] Stationary Movable

Sockets

Parent Socket: None

Static Mesh

Static Mesh: ENV_Bldng_05_Wall_01_Deco_04

Materials

Element 0: Bldng_035_DecoSheet_01_MIC

Component Tick

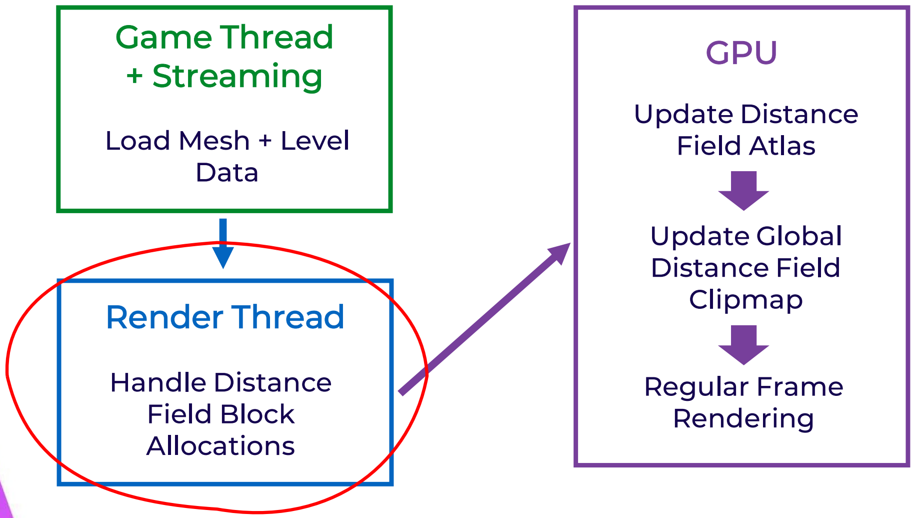
Start with Tick Enabled: ✓
Tick Interval (secs): 0.0

Physics

A consequence of the distance field data is that it is all stored in volume textures – meaning that it really likes being associated with static meshes that have minimal empty space, for example we don't want the volumes to include the empty space on the interiors of buildings.

We really leaned into Unreal's dynamic instancing system and built our assets out of many smaller mesh components from very early on in development, this minimizes texels wasted on empty space . These smaller meshes are grouped together into entities we call "basic structures" - you can see that this piece of a structure is its own mesh which is then instanced multiple times on the building, and the building has many static mesh instances making it up in the component panel on the left.

Distance Field Upload

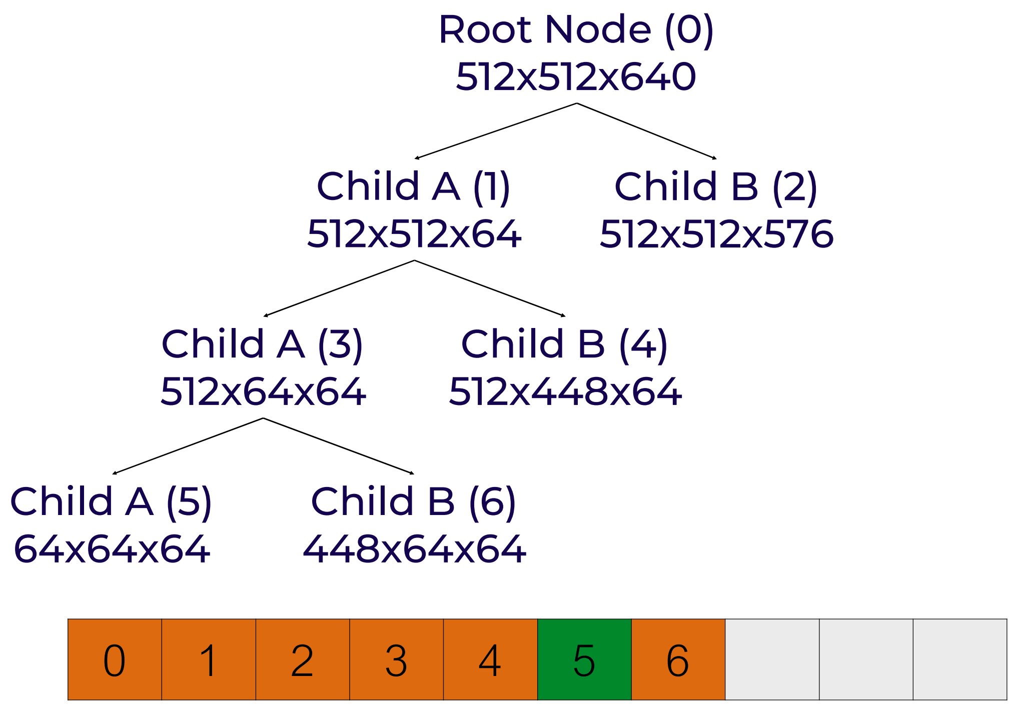

Game Thread
+ Streaming
Load Mesh + Level Data
Render Thread
Handle Distance Field Block Allocations
GPU
Update Distance Field Atlas
Update Global Distance Field Clipmap
Regular Frame Rendering

The distance field data has a number of systems it has to pass through while loading in before it is ready for rendering. This is after the main game thread runs it through the regular streaming process on the mesh and gameplay data.

The first is that the render thread manages the allocations within the distance field atlas with a block allocator that tracks where a particular mesh's data will be in the atlas and tries to minimize fragmentation within that resource.

Finally, there are multiple operations that must happen over on the GPU - it copies from the individual distance field memory into their assigned places in the atlas, and finally any regions of the global clipmap that need to be dirtied are reprocessed, including scrolling of the clipmap as the player moves through the world.

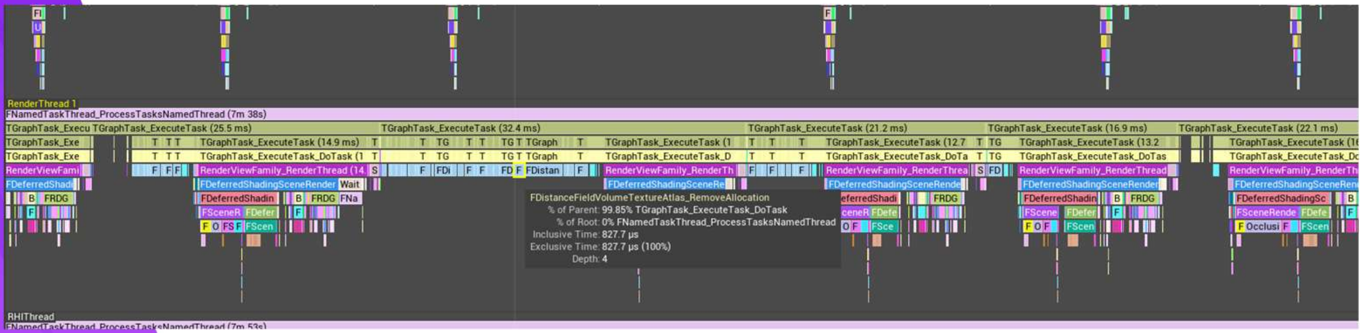
In the interest of time I'm going to cover my optimization effort on the render thread cost – but I've included details on GPU improvements in the appendix for these slides for those interested.



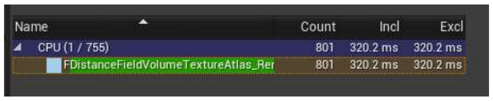
Now, recall that I mentioned that the quality and atlas memory usage encouraged the artists to use many small meshes. This pushed the render thread atlas manager to the limit. The block allocator constructs a binary tree of allocations with each node splitting the space in one of the 3 dimensions, which I've illustrated here for a single 64x64x64 allocation. The actual nodes are stored in a linear array, which I've marked the indices of. You can see that Node 5 is the only leaf node in this example.

Due to the increasingly large number of distance fields the artists were fitting into the atlas, removals and additions to the atlas became very expensive on the render thread. This time would probably be inconsequential in a more limited tree, but it became a problem for us as the distance field count in any given scene grew.

Removal Cost


RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (7m 38s)
TGraphTask_ExecuteTask (25.5 ms)
TGraphTask_ExecuteTask (32.4 ms)
TGraphTask_ExecuteTask (21.2 ms)
TGraphTask_ExecuteTask (16.9 ms)
TGraphTask_ExecuteTask (22.1 ms)
RenderViewFamily_RenderThread
FDeferredShadingSceneRender
RHIThread
FNamedTaskThread_ProcessTasksNamedThread (7m 53s)

FDistanceFieldVolumeTextureAtlas_RemoveAllocation
% of Parent: 99.85% TGraphTask_ExecuteTask_DoTask
% of Root: 0% FNamedTaskThread_ProcessTasksNamedThread
Inclusive Time: 827.7 µs
Exclusive Time: 827.7 µs (100%)
Depth: 4

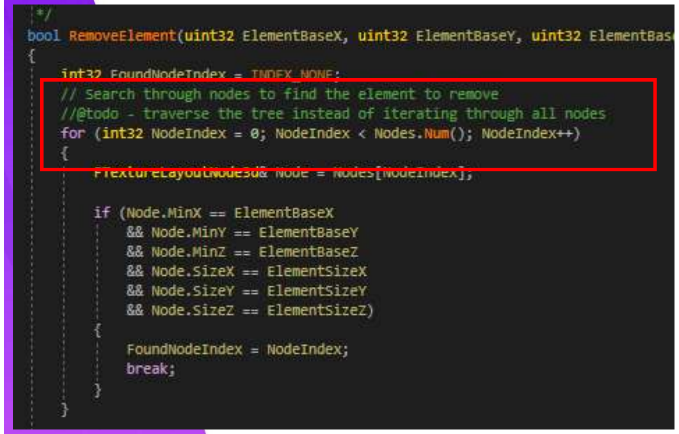


Name	Count	Incl	Excl
CPU (1 / 755)	801	320.2 ms	320.2 ms
FDistanceFieldVolumeTextureAtlas_Rer	801	320.2 ms	320.2 ms

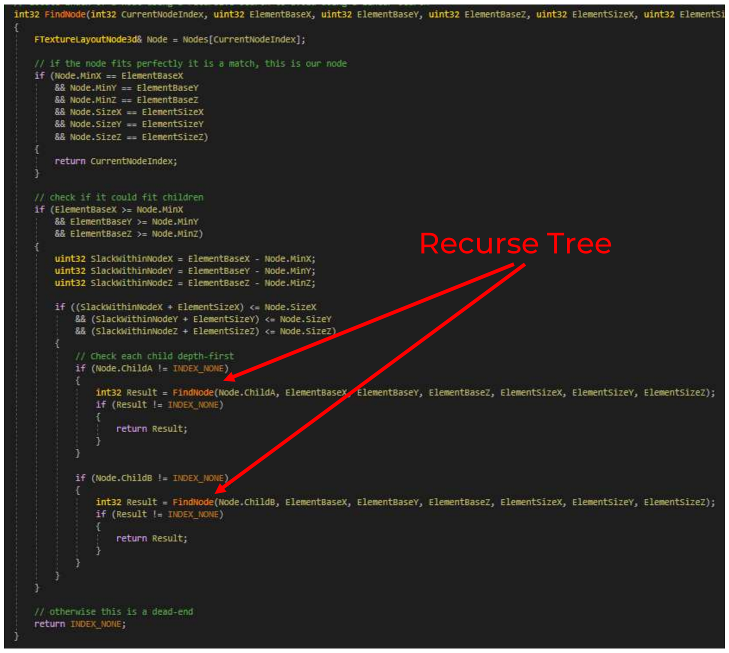
Average Incl Time: 0.40 ms

Removals were the first problem I tackled – here you can see that after a garbage collect the render thread is processing a ton of expensive removals in the atlas for multiple frames as incremental gc is processed. The cost to do a removal is variable but in one section I profiled the average cost was 0.40 ms. Even with throttling that seems expensive.

Let's investigate the code and look for improvements. This is going to be an exercise of algorithm optimization – think Big O notation



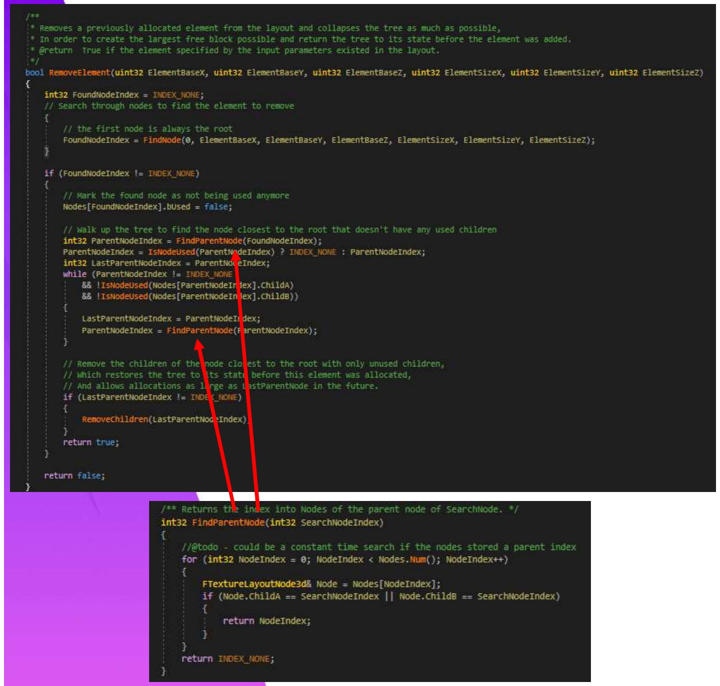
```
*/
bool RemoveElement(uint32 ElementBaseX, uint32 ElementBaseY, uint32 ElementBas
{
    int32 FoundNodeIndex = INDEX_NONE;
    // Search through nodes to find the element to remove
    //@todo - traverse the tree instead of iterating through all nodes
    for (int32 NodeIndex = 0; NodeIndex < Nodes.Num(); NodeIndex++)
    {
        FTextureLayoutNode3d& Node = Nodes[NodeIndex];

        if (Node.MinX == ElementBaseX
            && Node.MinY == ElementBaseY
            && Node.MinZ == ElementBaseZ
            && Node.SizeX == ElementSizeX
            && Node.SizeY == ElementSizeY
            && Node.SizeZ == ElementSizeZ)
        {
            FoundNodeIndex = NodeIndex;
            break;
        }
    }
```



```
int32 FindNode(int32 CurrentNodeIndex, uint32 ElementBaseX, uint32 ElementBaseY, uint32 ElementBaseZ, uint32 ElementSizeX, uint32 ElementSi
{
    FTextureLayoutNode3d& Node = Nodes[CurrentNodeIndex];

    // if the node fits perfectly it is a match, this is our node
    if (Node.MinX == ElementBaseX
        && Node.MinY == ElementBaseY
        && Node.MinZ == ElementBaseZ
        && Node.SizeX == ElementSizeX
        && Node.SizeY == ElementSizeY
        && Node.SizeZ == ElementSizeZ)
    {
        return CurrentNodeIndex;
    }

    // check if it could fit children
    if (ElementBaseX >= Node.MinX
        && ElementBaseY >= Node.MinY
        && ElementBaseZ >= Node.MinZ)
    {
        uint32 SlackWithinNodeX = ElementBaseX - Node.MinX;
        uint32 SlackWithinNodeY = ElementBaseY - Node.MinY;
        uint32 SlackWithinNodeZ = ElementBaseZ - Node.MinZ;

        if ((SlackWithinNodeX + ElementSizeX) <= Node.SizeX
            && (SlackWithinNodeY + ElementSizeY) <= Node.SizeY
            && (SlackWithinNodeZ + ElementSizeZ) <= Node.SizeZ)
        {
            // Check each child depth-first
            if (Node.ChildA != INDEX_NONE)
            {
                int32 Result = FindNode(Node.ChildA, ElementBaseX, ElementBaseY, ElementBaseZ, ElementSizeX, ElementSizeY, ElementSizeZ);
                if (Result != INDEX_NONE)
                {
                    return Result;
                }
            }

            if (Node.ChildB != INDEX_NONE)
            {
                int32 Result = FindNode(Node.ChildB, ElementBaseX, ElementBaseY, ElementBaseZ, ElementSizeX, ElementSizeY, ElementSizeZ);
                if (Result != INDEX_NONE)
                {
                    return Result;
                }
            }
        }
    }

    // otherwise this is a dead-end
    return INDEX_NONE;
}
```

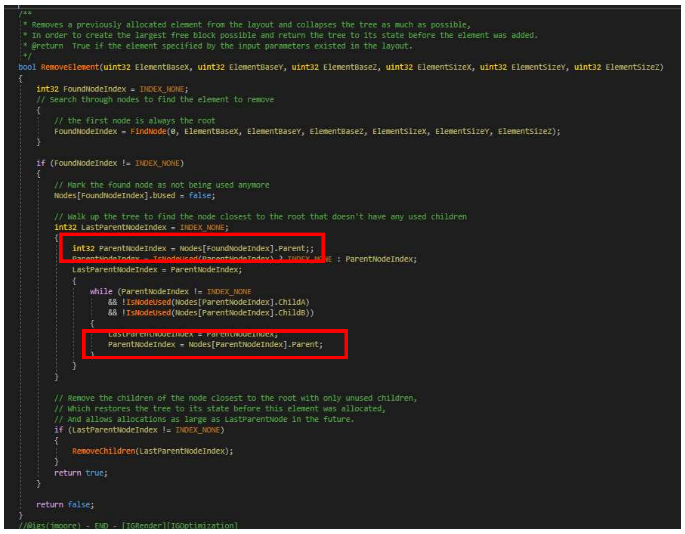
First up – there are a number of linear searches with todo’s in the code by the original authors
– it seems like a no-brainer to get those addressed.

This one is a linear search through the node array to try to find the allocation associated with the distance field we are removing, which we can change to a recursive traversal through the tree from the root node. Since we know the position and size of the node this is pretty close to going from linear to logarithmic in search time, depending on how balanced the tree is.



```cpp
/**
 * Removes a previously allocated element from the layout and collapses the tree as much as possible,
 * In order to create the largest free block possible and return the tree to its state before the element was added.
 * @return  True if the element specified by the input parameters existed in the layout.
 */
bool RemoveElement(uint32 ElementBaseX, uint32 ElementBaseY, uint32 ElementBaseZ, uint32 ElementSizeX, uint32 ElementSizeY, uint32 ElementSizeZ)
{
    int32 FoundNodeIndex = INDEX_NONE;
    // Search through nodes to find the element to remove
    {
        // the first node is always the root
        FoundNodeIndex = FindNode(0, ElementBaseX, ElementBaseY, ElementBaseZ, ElementSizeX, ElementSizeY, ElementSizeZ);
    }

    if (FoundNodeIndex != INDEX_NONE)
    {
        // Mark the found node as not being used anymore
        Nodes[FoundNodeIndex].bUsed = false;

        // Walk up the tree to find the node closest to the root that doesn't have any used children
        int32 ParentNodeIndex = FindParentNode(FoundNodeIndex);
        ParentNodeIndex = IsNodeUsed(ParentNodeIndex) ? INDEX_NONE : ParentNodeIndex;
        int32 LastParentNodeIndex = ParentNodeIndex;
        while (ParentNodeIndex != INDEX_NONE
            && !IsNodeUsed(Nodes[ParentNodeIndex].ChildA)
            && !IsNodeUsed(Nodes[ParentNodeIndex].ChildB))
        {
            LastParentNodeIndex = ParentNodeIndex;
            ParentNodeIndex = FindParentNode(ParentNodeIndex);
        }

        // Remove the children of the node closest to the root with only unused children,
        // Which restores the tree to its state before this element was allocated,
        // And allows allocations as large as LastParentNode in the future.
        if (LastParentNodeIndex != INDEX_NONE)
        {
            RemoveChildren(LastParentNodeIndex);
        }
        return true;
    }

    return false;
}
```

```cpp
/** Returns the index into Nodes of the parent node of SearchNode. */
int32 FindParentNode(int32 SearchNodeIndex)
{
    //@todo - could be a constant time search if the nodes stored a parent index
    for (int32 NodeIndex = 0; NodeIndex < Nodes.Num(); NodeIndex++)
    {
        FTextureLayoutNode3d& Node = Nodes[NodeIndex];
        if (Node.ChildA == SearchNodeIndex || Node.ChildB == SearchNodeIndex)
        {
            return NodeIndex;
        }
    }
    return INDEX_NONE;
}
```

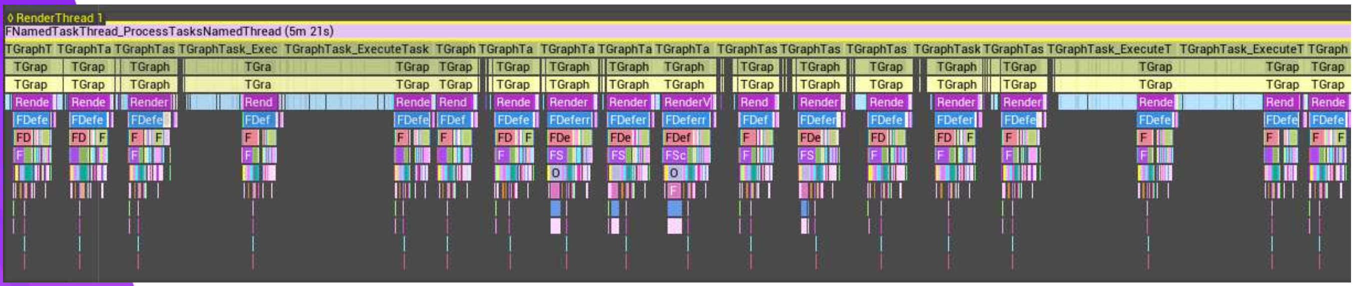


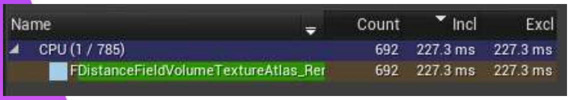
```cpp
/**
 * Removes a previously allocated element from the layout and collapses the tree as much as possible,
 * In order to create the largest free block possible and return the tree to its state before the element was added.
 * @return  True if the element specified by the input parameters existed in the layout.
 */
bool RemoveElement(uint32 ElementBaseX, uint32 ElementBaseY, uint32 ElementBaseZ, uint32 ElementSizeX, uint32 ElementSizeY, uint32 ElementSizeZ)
{
    int32 FoundNodeIndex = INDEX_NONE;
    // Search through nodes to find the element to remove
    {
        // the first node is always the root
        FoundNodeIndex = FindNode(0, ElementBaseX, ElementBaseY, ElementBaseZ, ElementSizeX, ElementSizeY, ElementSizeZ);
    }

    if (FoundNodeIndex != INDEX_NONE)
    {
        // Mark the found node as not being used anymore
        Nodes[FoundNodeIndex].bUsed = false;

        // Walk up the tree to find the node closest to the root that doesn't have any used children
        int32 LastParentNodeIndex = INDEX_NONE;
        {
            int32 ParentNodeIndex = Nodes[FoundNodeIndex].Parent;;
            ParentNodeIndex = IsNodeUsed(ParentNodeIndex) ? INDEX_NONE : ParentNodeIndex;
            LastParentNodeIndex = ParentNodeIndex;
            {
                while (ParentNodeIndex != INDEX_NONE
                    && !IsNodeUsed(Nodes[ParentNodeIndex].ChildA)
                    && !IsNodeUsed(Nodes[ParentNodeIndex].ChildB))
                {
                    LastParentNodeIndex = ParentNodeIndex;
                    ParentNodeIndex = Nodes[ParentNodeIndex].Parent;
                }
            }
        }

        // Remove the children of the node closest to the root with only unused children,
        // Which restores the tree to its state before this element was allocated,
        // And allows allocations as large as LastParentNode in the future.
        if (LastParentNodeIndex != INDEX_NONE)
        {
            RemoveChildren(LastParentNodeIndex);
        }
        return true;
    }

    return false;
}
//@igs(jmoore) - END - [IGRender][IGOptimization]
```

After the block being unloaded is identified, the RemoveElement function needs to find which parent nodes need to be unloaded with it. It traverse up from the node repeatedly calling this FindParentNode function, which upon closer inspection also has a handy todo comment in it that it is once again doing an unnecessary linear seach.

This is even simpler to resolve, we'll pay the extra 4 bytes per node to store the parent node index when blocks are inserted, to be able to turn these calls to FindParentNode into just an assignment

Removal Cost

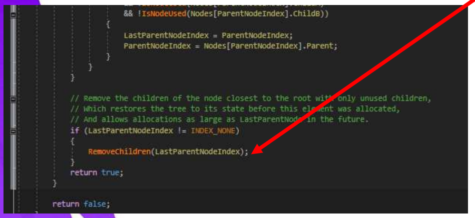




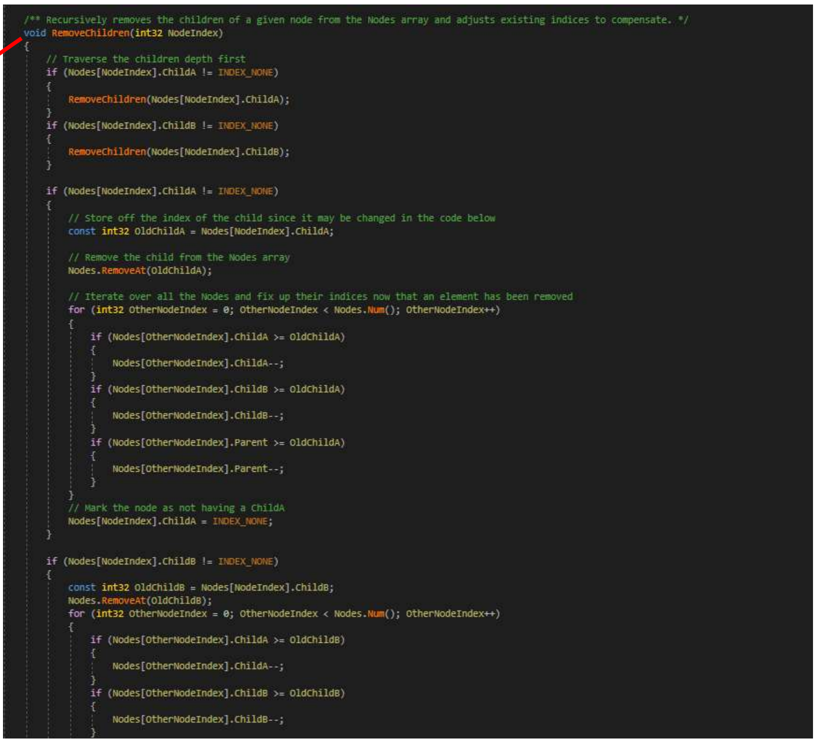
Name	Count	Incl	Excl
CPU (1 / 785)	692	227.3 ms	227.3 ms
FDistanceFieldVolumeTextureAtlas_Rer	692	227.3 ms	227.3 ms

Average Incl Time: 0.33 ms

Alright let’s check in on how we’re doing. Better, but still not great. We’re still averaging a third of a ms per removal which does not scale to dozens of removals happening in a single frame. You can see we frequently are spending more than 16 ms a frame processing large numbers of removals, which takes me to looking at more code, guided by function sample profiling.



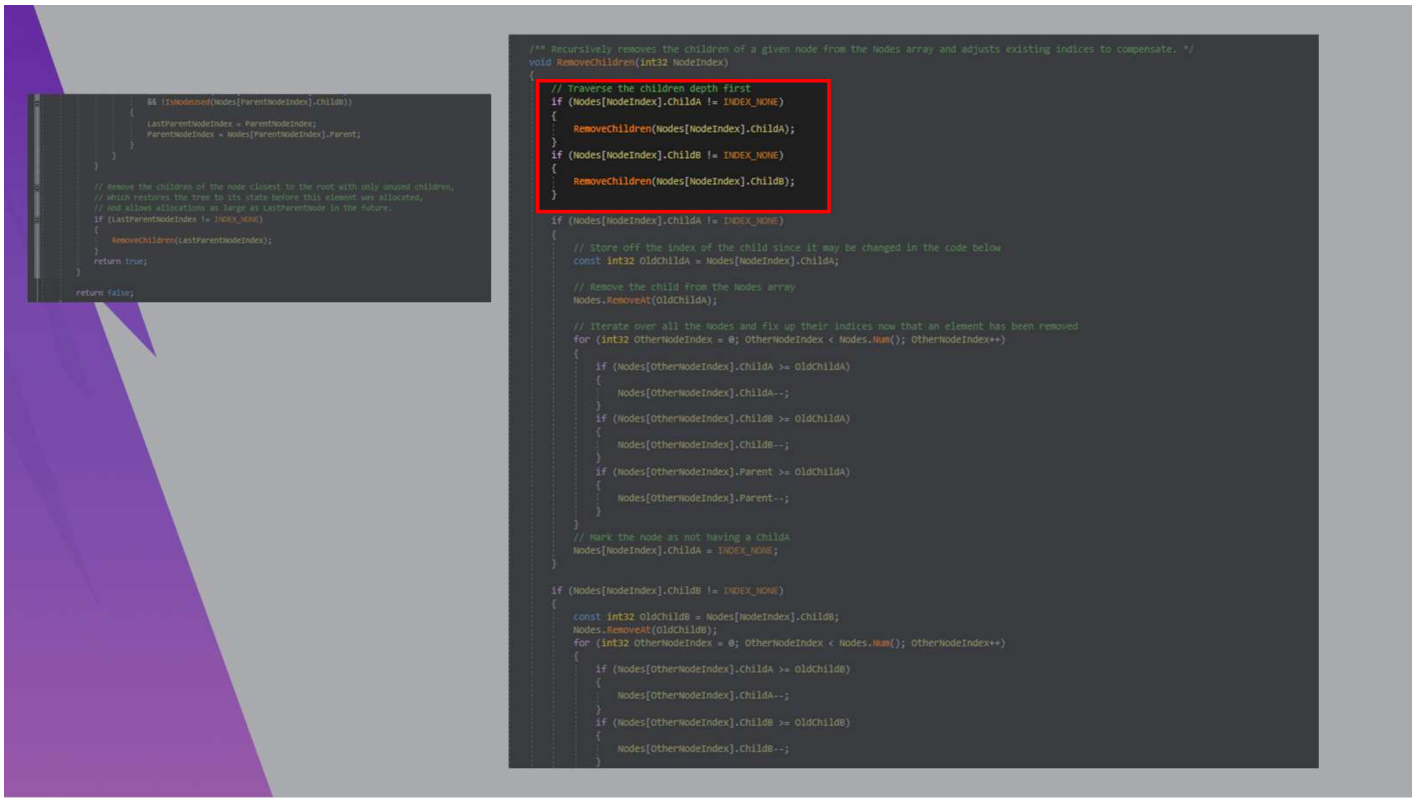
```
                && !IsNodeUsed(Nodes[ParentNodeIndex].ChildB))
            {
                LastParentNodeIndex = ParentNodeIndex;
                ParentNodeIndex = Nodes[ParentNodeIndex].Parent;
            }
        }
    }

    // Remove the children of the node closest to the root with only unused children,
    // Which restores the tree to its state before this element was allocated,
    // And allows allocations as large as LastParentNode in the future.
    if (LastParentNodeIndex != INDEX_NONE)
    {
        RemoveChildren(LastParentNodeIndex);
    }
    return true;
}

return false;
```



```
/** Recursively removes the children of a given node from the Nodes array and adjusts existing indices to compensate. */
void RemoveChildren(int32 NodeIndex)
{
    // Traverse the children depth first
    if (Nodes[NodeIndex].ChildA != INDEX_NONE)
    {
        RemoveChildren(Nodes[NodeIndex].ChildA);
    }
    if (Nodes[NodeIndex].ChildB != INDEX_NONE)
    {
        RemoveChildren(Nodes[NodeIndex].ChildB);
    }

    if (Nodes[NodeIndex].ChildA != INDEX_NONE)
    {
        // Store off the index of the child since it may be changed in the code below
        const int32 OldChildA = Nodes[NodeIndex].ChildA;

        // Remove the child from the Nodes array
        Nodes.RemoveAt(OldChildA);

        // Iterate over all the Nodes and fix up their indices now that an element has been removed
        for (int32 OtherNodeIndex = 0; OtherNodeIndex < Nodes.Num(); OtherNodeIndex++)
        {
            if (Nodes[OtherNodeIndex].ChildA >= OldChildA)
            {
                Nodes[OtherNodeIndex].ChildA--;
            }
            if (Nodes[OtherNodeIndex].ChildB >= OldChildA)
            {
                Nodes[OtherNodeIndex].ChildB--;
            }
            if (Nodes[OtherNodeIndex].Parent >= OldChildA)
            {
                Nodes[OtherNodeIndex].Parent--;
            }
        }
        // Mark the node as not having a ChildA
        Nodes[NodeIndex].ChildA = INDEX_NONE;
    }

    if (Nodes[NodeIndex].ChildB != INDEX_NONE)
    {
        const int32 OldChildB = Nodes[NodeIndex].ChildB;
        Nodes.RemoveAt(OldChildB);
        for (int32 OtherNodeIndex = 0; OtherNodeIndex < Nodes.Num(); OtherNodeIndex++)
        {
            if (Nodes[OtherNodeIndex].ChildA >= OldChildB)
            {
                Nodes[OtherNodeIndex].ChildA--;
            }
            if (Nodes[OtherNodeIndex].ChildB >= OldChildB)
            {
                Nodes[OtherNodeIndex].ChildB--;
            }
```

Following the search up the parent nodes to decide the sub-tree to remove, there's this RemoveChildren function that is called to actually remove those nodes.

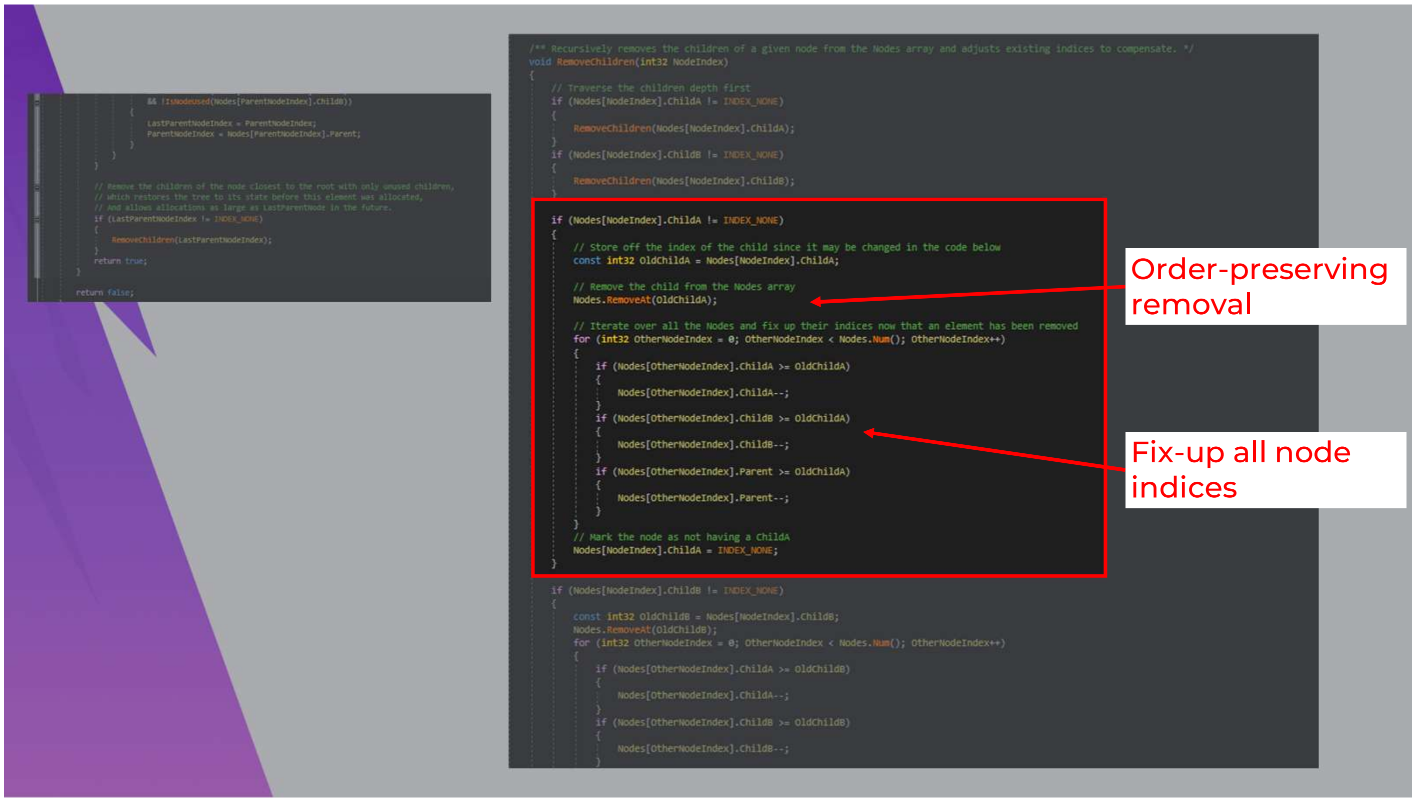
Function sampling profiling told me that this was a hot function – so let's take a look at what's happening inside of it and see if we can identify some improvements.



```
                    && !IsNodeUsed(Nodes[ParentNodeIndex].ChildB))
                {
                    LastParentNodeIndex = ParentNodeIndex;
                    ParentNodeIndex = Nodes[ParentNodeIndex].Parent;
                }
            }
        }

        // Remove the children of the node closest to the root with only unused children,
        // which restores the tree to its state before this element was allocated,
        // And allows allocations as large as LastParentNode in the future.
        if (LastParentNodeIndex != INDEX_NONE)
        {
            RemoveChildren(LastParentNodeIndex);
        }
        return true;
    }

    return false;
```

```
/** Recursively removes the children of a given node from the Nodes array and adjusts existing indices to compensate. */
void RemoveChildren(int32 NodeIndex)
{
    // Traverse the children depth first
    if (Nodes[NodeIndex].ChildA != INDEX_NONE)
    {
        RemoveChildren(Nodes[NodeIndex].ChildA);
    }
    if (Nodes[NodeIndex].ChildB != INDEX_NONE)
    {
        RemoveChildren(Nodes[NodeIndex].ChildB);
    }

    if (Nodes[NodeIndex].ChildA != INDEX_NONE)
    {
        // Store off the index of the child since it may be changed in the code below
        const int32 OldChildA = Nodes[NodeIndex].ChildA;

        // Remove the child from the Nodes array
        Nodes.RemoveAt(OldChildA);

        // Iterate over all the Nodes and fix up their indices now that an element has been removed
        for (int32 OtherNodeIndex = 0; OtherNodeIndex < Nodes.Num(); OtherNodeIndex++)
        {
            if (Nodes[OtherNodeIndex].ChildA >= OldChildA)
            {
                Nodes[OtherNodeIndex].ChildA--;
            }
            if (Nodes[OtherNodeIndex].ChildB >= OldChildA)
            {
                Nodes[OtherNodeIndex].ChildB--;
            }
            if (Nodes[OtherNodeIndex].Parent >= OldChildA)
            {
                Nodes[OtherNodeIndex].Parent--;
            }
        }
        // Mark the node as not having a ChildA
        Nodes[NodeIndex].ChildA = INDEX_NONE;
    }

    if (Nodes[NodeIndex].ChildB != INDEX_NONE)
    {
        const int32 OldChildB = Nodes[NodeIndex].ChildB;
        Nodes.RemoveAt(OldChildB);
        for (int32 OtherNodeIndex = 0; OtherNodeIndex < Nodes.Num(); OtherNodeIndex++)
        {
            if (Nodes[OtherNodeIndex].ChildA >= OldChildB)
            {
                Nodes[OtherNodeIndex].ChildA--;
            }
            if (Nodes[OtherNodeIndex].ChildB >= OldChildB)
            {
                Nodes[OtherNodeIndex].ChildB--;
            }
```

The first thing to note that this is once again a recursive function traversing down each child from the node we've selected for removal. So it makes sense that this function showed up in profiling.

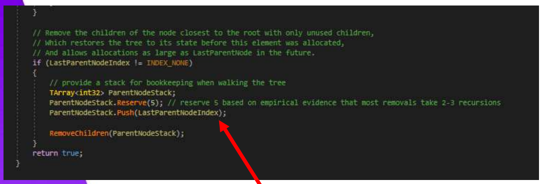


```
                && !IsNodeUsed(Nodes[ParentNodeIndex].ChildB))
            {
                LastParentNodeIndex = ParentNodeIndex;
                ParentNodeIndex = Nodes[ParentNodeIndex].Parent;
            }
        }
    }

    // Remove the children of the node closest to the root with only unused children,
    // which restores the tree to its state before this element was allocated,
    // And allows allocations as large as LastParentNode in the future.
    if (LastParentNodeIndex != INDEX_NONE)
    {
        RemoveChildren(LastParentNodeIndex);
    }
    return true;
}

return false;
```

```
/** Recursively removes the children of a given node from the Nodes array and adjusts existing indices to compensate. */
void RemoveChildren(int32 NodeIndex)
{
    // Traverse the children depth first
    if (Nodes[NodeIndex].ChildA != INDEX_NONE)
    {
        RemoveChildren(Nodes[NodeIndex].ChildA);
    }
    if (Nodes[NodeIndex].ChildB != INDEX_NONE)
    {
        RemoveChildren(Nodes[NodeIndex].ChildB);
    }

    if (Nodes[NodeIndex].ChildA != INDEX_NONE)
    {
        // Store off the index of the child since it may be changed in the code below
        const int32 OldChildA = Nodes[NodeIndex].ChildA;

        // Remove the child from the Nodes array
        Nodes.RemoveAt(OldChildA);

        // Iterate over all the Nodes and fix up their indices now that an element has been removed
        for (int32 OtherNodeIndex = 0; OtherNodeIndex < Nodes.Num(); OtherNodeIndex++)
        {
            if (Nodes[OtherNodeIndex].ChildA >= OldChildA)
            {
                Nodes[OtherNodeIndex].ChildA--;
            }
            if (Nodes[OtherNodeIndex].ChildB >= OldChildA)
            {
                Nodes[OtherNodeIndex].ChildB--;
            }
            if (Nodes[OtherNodeIndex].Parent >= OldChildA)
            {
                Nodes[OtherNodeIndex].Parent--;
            }
        }
        // Mark the node as not having a ChildA
        Nodes[NodeIndex].ChildA = INDEX_NONE;
    }

    if (Nodes[NodeIndex].ChildB != INDEX_NONE)
    {
        const int32 OldChildB = Nodes[NodeIndex].ChildB;
        Nodes.RemoveAt(OldChildB);
        for (int32 OtherNodeIndex = 0; OtherNodeIndex < Nodes.Num(); OtherNodeIndex++)
        {
            if (Nodes[OtherNodeIndex].ChildA >= OldChildB)
            {
                Nodes[OtherNodeIndex].ChildA--;
            }
            if (Nodes[OtherNodeIndex].ChildB >= OldChildB)
            {
                Nodes[OtherNodeIndex].ChildB--;
            }
```

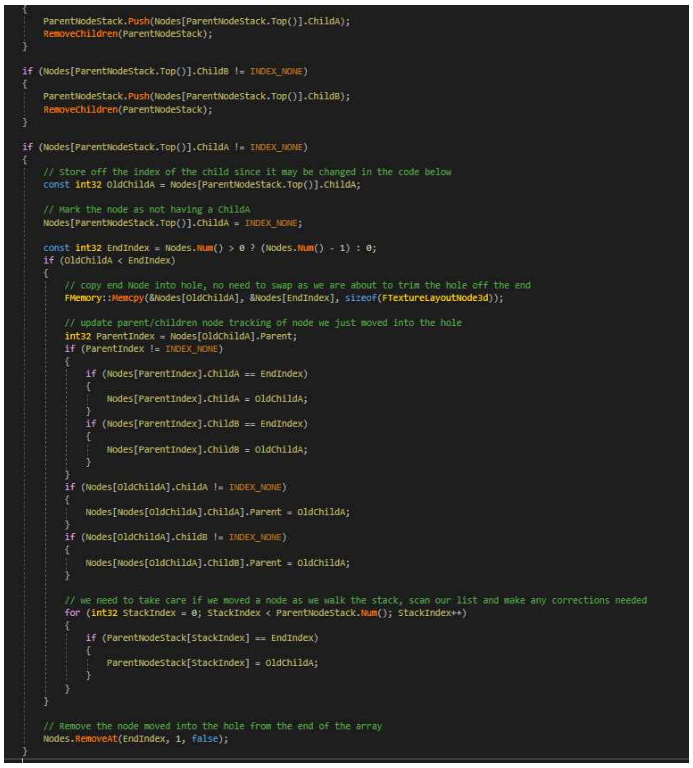
Now let's look at the actual work is happening – each child is removed from the node array. This is handled by calling RemoveAt on the array holding all the nodes – which is using Unreal's TArray container class. This is an order-preserving removal, which internally means a large memmove of all the nodes following the removal index are copied down into the hole.

Then in the following loop, each node has the indices of their children iterated and adjusted to account for the shift in the array. Now, you may be familiar that there is a faster way to remove an item from an array if you *don't* care to preserve the order, which is to simply copy the final element into the hole that was just created. I'd like to do that here to avoid iterating every node inside a sub-tree traversal by simply fixing up the child indices of the parent node – which is a constant time access due to the prior change I showed where we store that on each node now. However, this is tricky to do in practice because we are trying to modify the tree as we are traversing it.



```
    }

    // Remove the children of the node closest to the root with only unused children,
    // Which restores the tree to its state before this element was allocated,
    // And allows allocations as large as LastParentNode in the future.
    if (LastParentNodeIndex != INDEX_NONE)
    {
        // provide a stack for bookkeeping when walking the tree
        TArray<int32> ParentNodeStack;
        ParentNodeStack.Reserve(5); // reserve 5 based on empirical evidence that most removals take 2-3 recursions
        ParentNodeStack.Push(LastParentNodeIndex);

        RemoveChildren(ParentNodeStack);
    }
    return true;
}
```

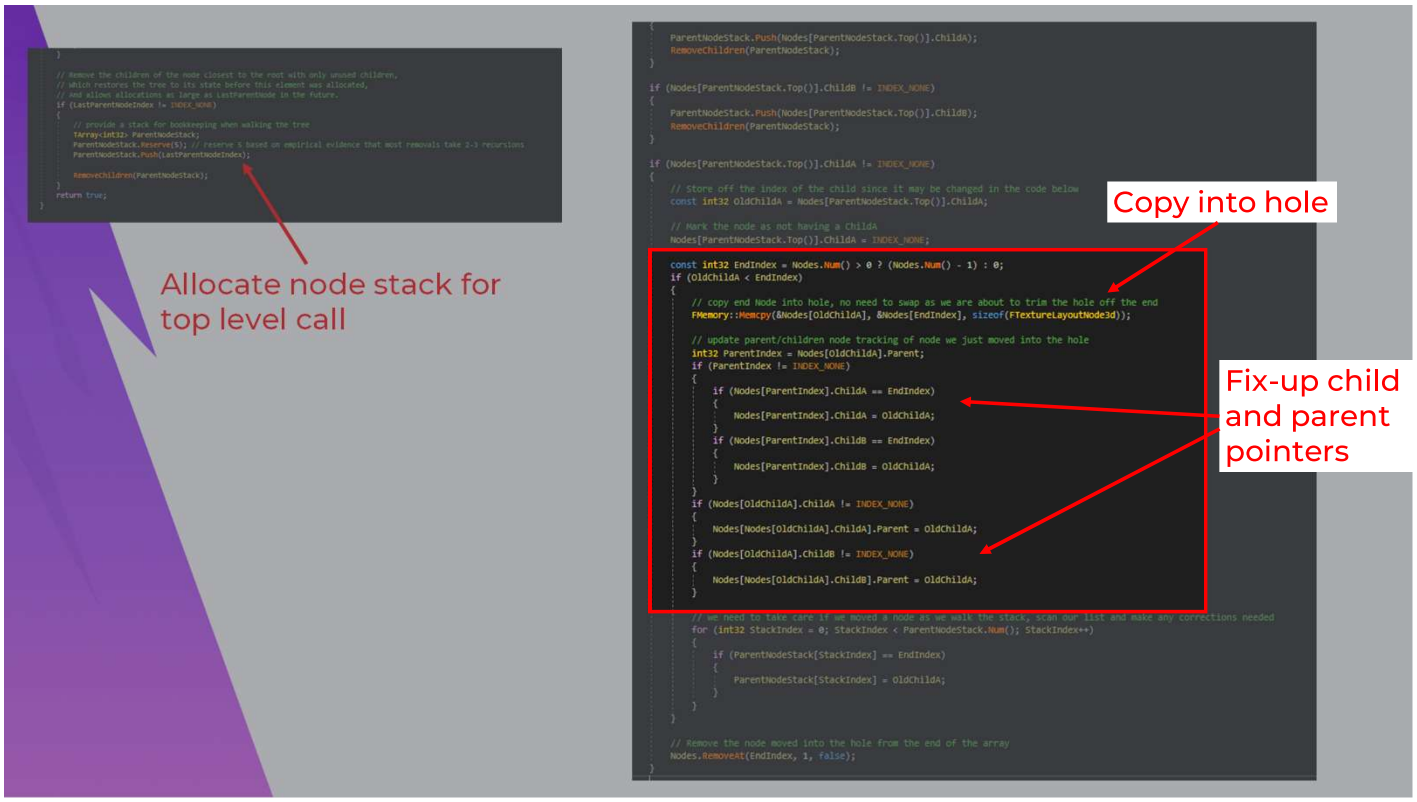
Allocate node stack for top level call



```
{
    ParentNodeStack.Push(Nodes[ParentNodeStack.Top()].ChildA);
    RemoveChildren(ParentNodeStack);
}

if (Nodes[ParentNodeStack.Top()].ChildB != INDEX_NONE)
{
    ParentNodeStack.Push(Nodes[ParentNodeStack.Top()].ChildB);
    RemoveChildren(ParentNodeStack);
}

if (Nodes[ParentNodeStack.Top()].ChildA != INDEX_NONE)
{
    // Store off the index of the child since it may be changed in the code below
    const int32 OldChildA = Nodes[ParentNodeStack.Top()].ChildA;

    // Mark the node as not having a ChildA
    Nodes[ParentNodeStack.Top()].ChildA = INDEX_NONE;

    const int32 EndIndex = Nodes.Num() > 0 ? (Nodes.Num() - 1) : 0;
    if (OldChildA < EndIndex)
    {
        // copy end Node into hole, no need to swap as we are about to trim the hole off the end
        FMemory::Memcpy(&Nodes[OldChildA], &Nodes[EndIndex], sizeof(FTextureLayoutNode3d));

        // update parent/children node tracking of node we just moved into the hole
        int32 ParentIndex = Nodes[OldChildA].Parent;
        if (ParentIndex != INDEX_NONE)
        {
            if (Nodes[ParentIndex].ChildA == EndIndex)
            {
                Nodes[ParentIndex].ChildA = OldChildA;
            }
            if (Nodes[ParentIndex].ChildB == EndIndex)
            {
                Nodes[ParentIndex].ChildB = OldChildA;
            }
        }
        if (Nodes[OldChildA].ChildA != INDEX_NONE)
        {
            Nodes[Nodes[OldChildA].ChildA].Parent = OldChildA;
        }
        if (Nodes[OldChildA].ChildB != INDEX_NONE)
        {
            Nodes[Nodes[OldChildA].ChildB].Parent = OldChildA;
        }

        // we need to take care if we moved a node as we walk the stack, scan our list and make any corrections needed
        for (int32 StackIndex = 0; StackIndex < ParentNodeStack.Num(); StackIndex++)
        {
            if (ParentNodeStack[StackIndex] == EndIndex)
            {
                ParentNodeStack[StackIndex] = OldChildA;
            }
        }
    }

    // Remove the node moved into the hole from the end of the array
    Nodes.RemoveAt(EndIndex, 1, false);
}
```

To get around this problem, I allocate and pass around a stack of parent nodes as we recurse back down the tree.

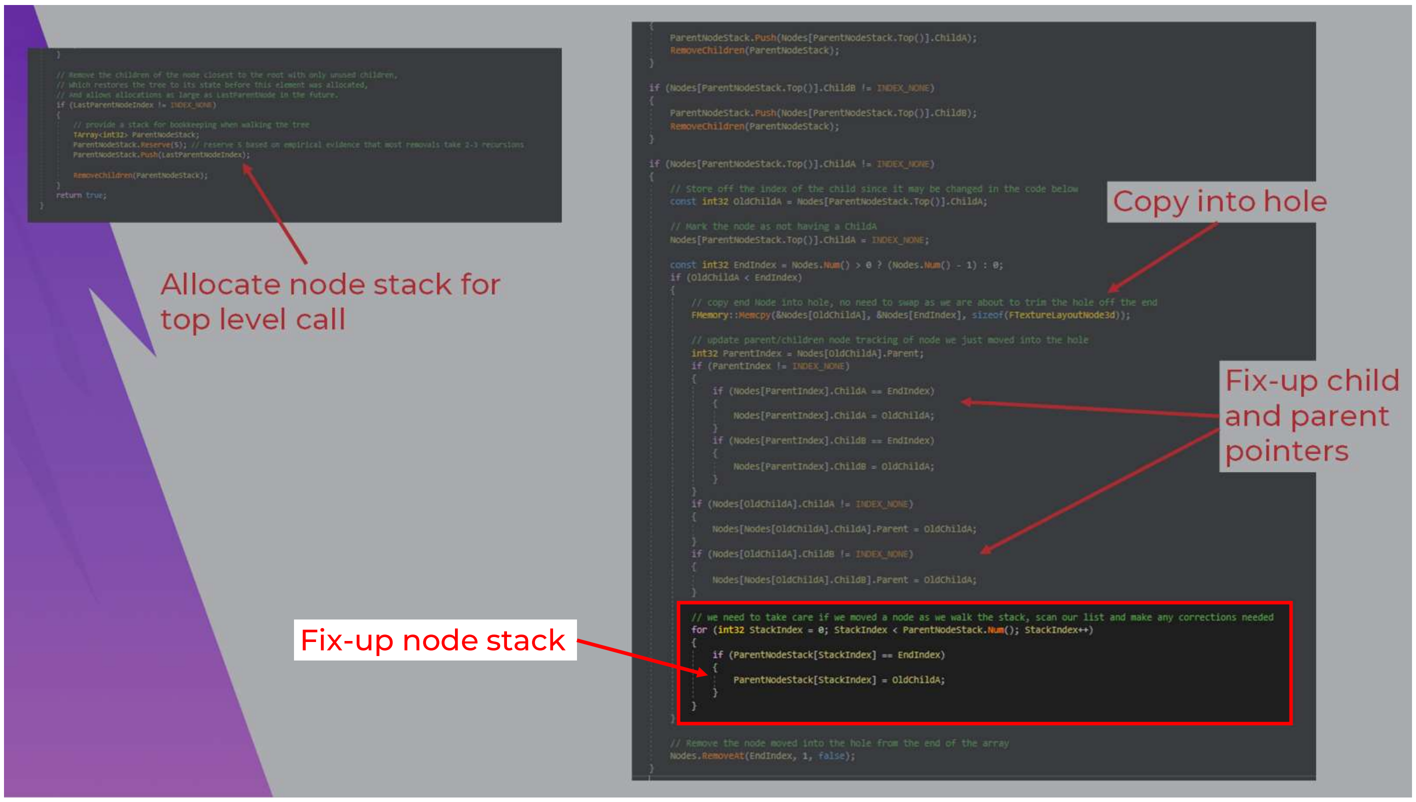
This allows us to perform fixup on the stack and avoid getting lost during our traversal as we remove nodes. This can be seen here in the updated logic inside of the RemoveChildren function.



```
    }

    // Remove the children of the node closest to the root with only unused children,
    // Which restores the tree to its state before this element was allocated,
    // And allows allocations as large as LastParentNode in the future.
    if (LastParentNodeIndex != INDEX_NONE)
    {
        // provide a stack for bookkeeping when walking the tree
        TArray<int32> ParentNodeStack;
        ParentNodeStack.Reserve(5); // reserve 5 based on empirical evidence that most removals take 2-3 recursions
        ParentNodeStack.Push(LastParentNodeIndex);

        RemoveChildren(ParentNodeStack);
    }
    return true;
}
```

```
{
    ParentNodeStack.Push(Nodes[ParentNodeStack.Top()].ChildA);
    RemoveChildren(ParentNodeStack);
}

if (Nodes[ParentNodeStack.Top()].ChildB != INDEX_NONE)
{
    ParentNodeStack.Push(Nodes[ParentNodeStack.Top()].ChildB);
    RemoveChildren(ParentNodeStack);
}

if (Nodes[ParentNodeStack.Top()].ChildA != INDEX_NONE)
{
    // Store off the index of the child since it may be changed in the code below
    const int32 OldChildA = Nodes[ParentNodeStack.Top()].ChildA;

    // Mark the node as not having a ChildA
    Nodes[ParentNodeStack.Top()].ChildA = INDEX_NONE;

    const int32 EndIndex = Nodes.Num() > 0 ? (Nodes.Num() - 1) : 0;
    if (OldChildA < EndIndex)
    {
        // copy end Node into hole, no need to swap as we are about to trim the hole off the end
        FMemory::Memcpy(&Nodes[OldChildA], &Nodes[EndIndex], sizeof(FTextureLayoutNode3d));

        // update parent/children node tracking of node we just moved into the hole
        int32 ParentIndex = Nodes[OldChildA].Parent;
        if (ParentIndex != INDEX_NONE)
        {
            if (Nodes[ParentIndex].ChildA == EndIndex)
            {
                Nodes[ParentIndex].ChildA = OldChildA;
            }
            if (Nodes[ParentIndex].ChildB == EndIndex)
            {
                Nodes[ParentIndex].ChildB = OldChildA;
            }
        }
        if (Nodes[OldChildA].ChildA != INDEX_NONE)
        {
            Nodes[Nodes[OldChildA].ChildA].Parent = OldChildA;
        }
        if (Nodes[OldChildA].ChildB != INDEX_NONE)
        {
            Nodes[Nodes[OldChildA].ChildB].Parent = OldChildA;
        }

        // we need to take care if we moved a node as we walk the stack, scan our list and make any corrections needed
        for (int32 StackIndex = 0; StackIndex < ParentNodeStack.Num(); StackIndex++)
        {
            if (ParentNodeStack[StackIndex] == EndIndex)
            {
                ParentNodeStack[StackIndex] = OldChildA;
            }
        }
    }

    // Remove the node moved into the hole from the end of the array
    Nodes.RemoveAt(EndIndex, 1, false);
}
```

Here is the operation where we take the node at the end of the array and copy it into the location where we are doing a removal. Then we fix-up the child and parent node references to account for the node we just moved.

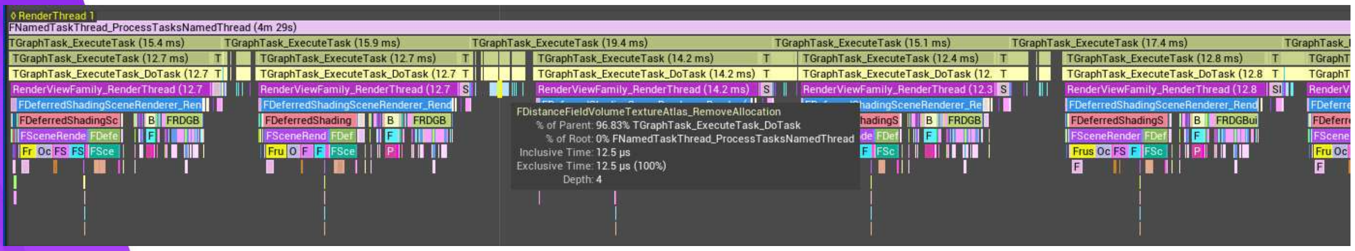


```
    }

    // Remove the children of the node closest to the root with only unused children,
    // Which restores the tree to its state before this element was allocated,
    // And allows allocations as large as LastParentNode in the future.
    if (LastParentNodeIndex != INDEX_NONE)
    {
        // provide a stack for bookkeeping when walking the tree
        TArray<int32> ParentNodeStack;
        ParentNodeStack.Reserve(5); // reserve 5 based on empirical evidence that most removals take 2-3 recursions
        ParentNodeStack.Push(LastParentNodeIndex);

        RemoveChildren(ParentNodeStack);
    }
    return true;
}
```

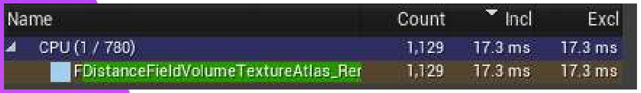
```
{
    ParentNodeStack.Push(Nodes[ParentNodeStack.Top()].ChildA);
    RemoveChildren(ParentNodeStack);
}

if (Nodes[ParentNodeStack.Top()].ChildB != INDEX_NONE)
{
    ParentNodeStack.Push(Nodes[ParentNodeStack.Top()].ChildB);
    RemoveChildren(ParentNodeStack);
}

if (Nodes[ParentNodeStack.Top()].ChildA != INDEX_NONE)
{
    // Store off the index of the child since it may be changed in the code below
    const int32 OldChildA = Nodes[ParentNodeStack.Top()].ChildA;

    // Mark the node as not having a ChildA
    Nodes[ParentNodeStack.Top()].ChildA = INDEX_NONE;

    const int32 EndIndex = Nodes.Num() > 0 ? (Nodes.Num() - 1) : 0;
    if (OldChildA < EndIndex)
    {
        // copy end Node into hole, no need to swap as we are about to trim the hole off the end
        FMemory::Memcpy(&Nodes[OldChildA], &Nodes[EndIndex], sizeof(FTextureLayoutNode3d));

        // update parent/children node tracking of node we just moved into the hole
        int32 ParentIndex = Nodes[OldChildA].Parent;
        if (ParentIndex != INDEX_NONE)
        {
            if (Nodes[ParentIndex].ChildA == EndIndex)
            {
                Nodes[ParentIndex].ChildA = OldChildA;
            }
            if (Nodes[ParentIndex].ChildB == EndIndex)
            {
                Nodes[ParentIndex].ChildB = OldChildA;
            }
        }
        if (Nodes[OldChildA].ChildA != INDEX_NONE)
        {
            Nodes[Nodes[OldChildA].ChildA].Parent = OldChildA;
        }
        if (Nodes[OldChildA].ChildB != INDEX_NONE)
        {
            Nodes[Nodes[OldChildA].ChildB].Parent = OldChildA;
        }

        // we need to take care if we moved a node as we walk the stack, scan our list and make any corrections needed
        for (int32 StackIndex = 0; StackIndex < ParentNodeStack.Num(); StackIndex++)
        {
            if (ParentNodeStack[StackIndex] == EndIndex)
            {
                ParentNodeStack[StackIndex] = OldChildA;
            }
        }
    }

    // Remove the node moved into the hole from the end of the array
    Nodes.RemoveAt(EndIndex, 1, false);
}
```

Following this I want to call attention to this code – here we need to look at our stack of nodes we're maintaining and make sure that if one of the nodes in our traversal path was the one we just moved, we need to fix-up the location in our stack so we don't get lost as we continue our traversal

Removal Cost


RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (4m 29s)
TGraphTask_ExecuteTask (15.4 ms)
TGraphTask_ExecuteTask (15.9 ms)
TGraphTask_ExecuteTask (19.4 ms)
TGraphTask_ExecuteTask (15.1 ms)
TGraphTask_ExecuteTask (17.4 ms)
TGraphTask_ExecuteTask (12.7 ms)
TGraphTask_ExecuteTask (14.2 ms)
TGraphTask_ExecuteTask (12.4 ms)
TGraphTask_ExecuteTask (12.8 ms)
TGraphTask_ExecuteTask_DoTask (14.2 ms)
RenderViewFamily_RenderThread (14.2 ms)
FDeferredShadingSceneRenderer_Rend
FRDGB

FDistanceFieldVolumeTextureAtlas_RemoveAllocation
% of Parent: 96.83% TGraphTask_ExecuteTask_DoTask
% of Root: 0% FNamedTaskThread_ProcessTasksNamedThread
Inclusive Time: 12.5 µs
Exclusive Time: 12.5 µs (100%)
Depth: 4

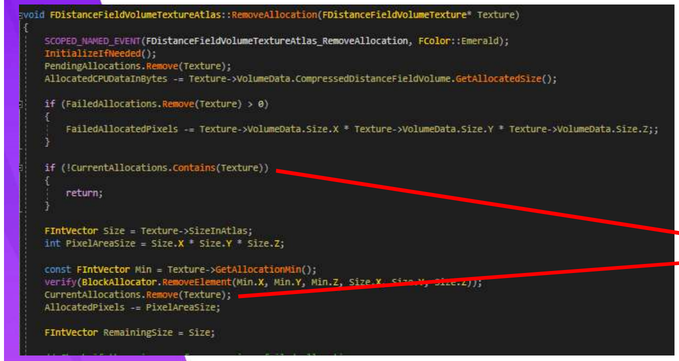


Name	Count	Incl	Excl
CPU (1 / 780)	1,129	17.3 ms	17.3 ms
FDistanceFieldVolumeTextureAtlas_Rer	1,129	17.3 ms	17.3 ms

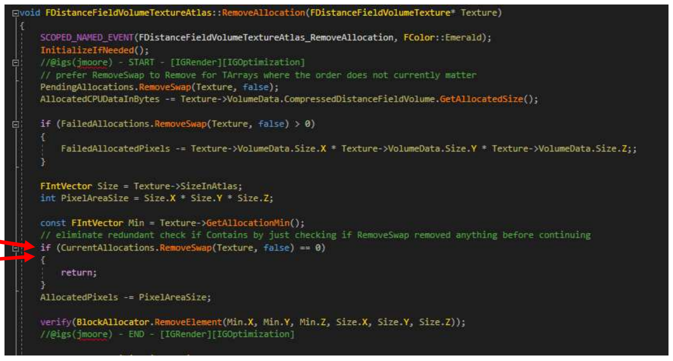
Average Incl Time: 0.02 ms

Now back in a profiling trace, this is what I was looking for –we're seeing a 20x improvement on the average removal time from where we started. I can almost call this a day at this point, you can see now that the time spent doing removals no longer dwarfs the time spent on regular frame rendering

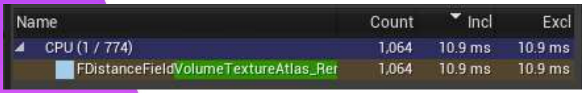
Higher Level Code



```
void FDistanceFieldVolumeTextureAtlas::RemoveAllocation(FDistanceFieldVolumeTexture* Texture)
{
    SCOPED_NAMED_EVENT(FDistanceFieldVolumeTextureAtlas_RemoveAllocation, FColor::Emerald);
    InitializeIfNeeded();
    PendingAllocations.Remove(Texture);
    AllocatedCPUDataInBytes -= Texture->VolumeData.CompressedDistanceFieldVolume.GetAllocatedSize();

    if (FailedAllocations.Remove(Texture) > 0)
    {
        FailedAllocatedPixels -= Texture->VolumeData.Size.X * Texture->VolumeData.Size.Y * Texture->VolumeData.Size.Z;;
    }

    if (!CurrentAllocations.Contains(Texture))
    {
        return;
    }

    FIntVector Size = Texture->SizeInAtlas;
    int PixelAreaSize = Size.X * Size.Y * Size.Z;

    const FIntVector Min = Texture->GetAllocationMin();
    verify(BlockAllocator.RemoveElement(Min.X, Min.Y, Min.Z, Size.X, Size.Y, Size.Z));
    CurrentAllocations.Remove(Texture);
    AllocatedPixels -= PixelAreaSize;

    FIntVector RemainingSize = Size;
```



```
void FDistanceFieldVolumeTextureAtlas::RemoveAllocation(FDistanceFieldVolumeTexture* Texture)
{
    SCOPED_NAMED_EVENT(FDistanceFieldVolumeTextureAtlas_RemoveAllocation, FColor::Emerald);
    InitializeIfNeeded();
    //@igs(jmoore) - START - [IGRender][IGOptimization]
    // prefer RemoveSwap to Remove for TArrays where the order does not currently matter
    PendingAllocations.RemoveSwap(Texture, false);
    AllocatedCPUDataInBytes -= Texture->VolumeData.CompressedDistanceFieldVolume.GetAllocatedSize();

    if (FailedAllocations.RemoveSwap(Texture, false) > 0)
    {
        FailedAllocatedPixels -= Texture->VolumeData.Size.X * Texture->VolumeData.Size.Y * Texture->VolumeData.Size.Z;;
    }

    FIntVector Size = Texture->SizeInAtlas;
    int PixelAreaSize = Size.X * Size.Y * Size.Z;

    const FIntVector Min = Texture->GetAllocationMin();
    // eliminate redundant check if Contains by just checking if RemoveSwap removed anything before continuing
    if (CurrentAllocations.RemoveSwap(Texture, false) == 0)
    {
        return;
    }
    AllocatedPixels -= PixelAreaSize;

    verify(BlockAllocator.RemoveElement(Min.X, Min.Y, Min.Z, Size.X, Size.Y, Size.Z));
    //@igs(jmoore) - END - [IGRender][IGOptimization]
```



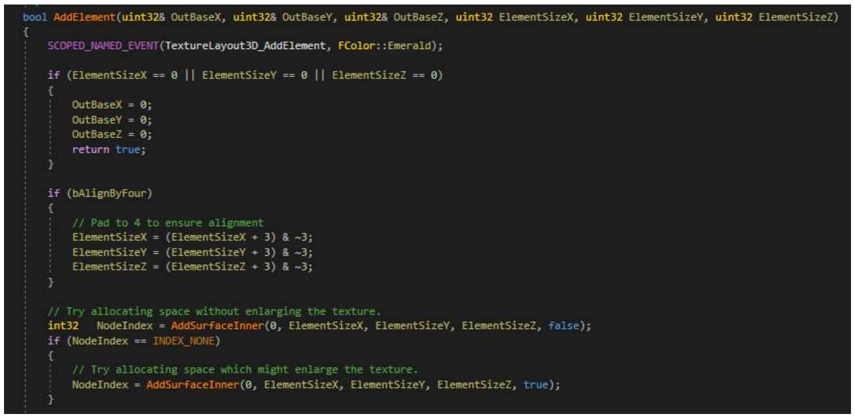
```
Name                                 Count    Incl      Excl
CPU (1 / 774)                        1,064    10.9 ms   10.9 ms
  FDistanceFieldVolumeTextureAtlas_Rer  1,064    10.9 ms   10.9 ms
```

Average Incl Time: 0.01 ms

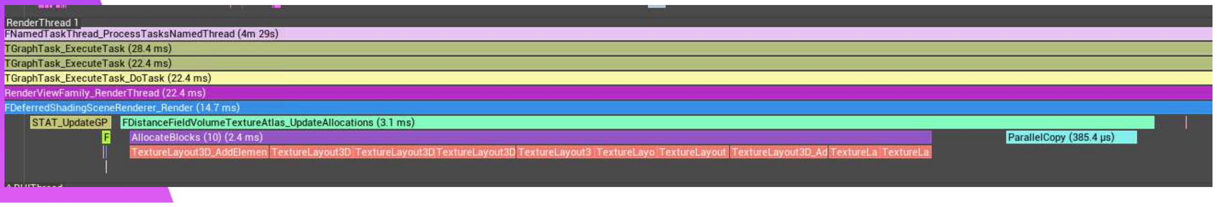
There’s one more thing improvement that I want to mention that is a bit of a pet peeve of mine. Don’t do a search and remove separately! The top level management that uses the block allocator I was optimizing has several lists that can use RemoveSwap instead of Remove because order doesn’t matter, and then the call to Contains (which will do a linear search) can be eliminated by just using the fact that RemoveSwap will return the number of elements removed.

This halves our removal time again – although this was a just a small portion of the cost before our other improvements were made. This is a 40x improvement overall. Again, I’m sure that this time is inconsequential for many games, the game we were building coupled open world level streaming with high numbers of small distance field allocations, it became very worthwhile for us to spend time optimizing this code.

Insertion Cost

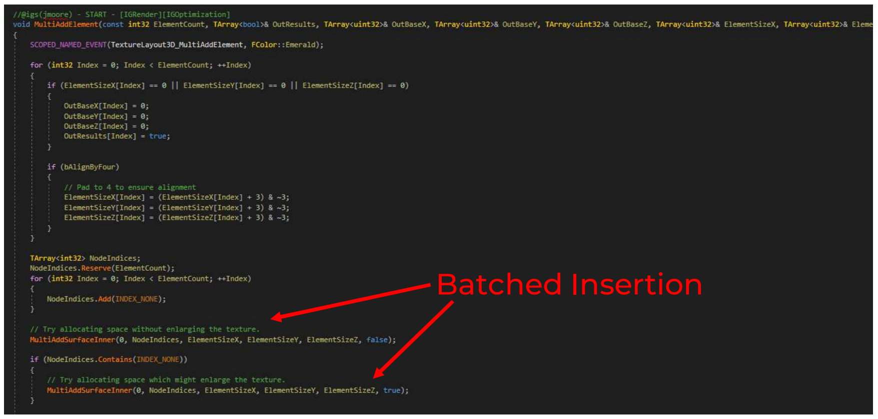


```
bool AddElement(uint32& OutBaseX, uint32& OutBaseY, uint32& OutBaseZ, uint32 ElementSizeX, uint32 ElementSizeY, uint32 ElementSizeZ)
{
    SCOPED_NAMED_EVENT(TextureLayout3D_AddElement, FColor::Emerald);

    if (ElementSizeX == 0 || ElementSizeY == 0 || ElementSizeZ == 0)
    {
        OutBaseX = 0;
        OutBaseY = 0;
        OutBaseZ = 0;
        return true;
    }

    if (bAlignByFour)
    {
        // Pad to 4 to ensure alignment
        ElementSizeX = (ElementSizeX + 3) & ~3;
        ElementSizeY = (ElementSizeY + 3) & ~3;
        ElementSizeZ = (ElementSizeZ + 3) & ~3;
    }

    // Try allocating space without enlarging the texture.
    int32   NodeIndex = AddSurfaceInner(0, ElementSizeX, ElementSizeY, ElementSizeZ, false);
    if (NodeIndex == INDEX_NONE)
    {
        // Try allocating space which might enlarge the texture.
        NodeIndex = AddSurfaceInner(0, ElementSizeX, ElementSizeY, ElementSizeZ, true);
    }
```

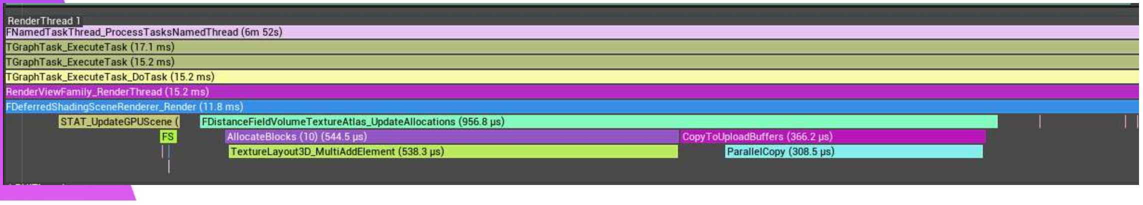

RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (4m 29s)
TGraphTask_ExecuteTask (28.4 ms)
TGraphTask_ExecuteTask (22.4 ms)
TGraphTask_ExecuteTask_DoTask (22.4 ms)
RenderViewFamily_RenderThread (22.4 ms)
FDeferredShadingSceneRenderer_Render (14.7 ms)
STAT_UpdateGP
FDistanceFieldVolumeTextureAtlas_UpdateAllocations (3.1 ms)
F
AllocateBlocks (10) (2.4 ms)
ParallelCopy (385.4 µs)
TextureLayout3D_AddElemen
TextureLayout3D
TextureLayout3D
TextureLayout3D
TextureLayout3
TextureLayo
TextureLayout
TextureLayout3D_Ad
TextureLa
TextureLa

Besides removals from the block allocator, insertions also had lots of room for improvement – these were walking the tree looking for a suitable hole to allocate the DF into, but each insertion would start over at the root node. This seemed redundant – in the trace associated with this code you can see AddElement is called 10 times, and the tree is potentially walked twice by each call – this is taking 2.5 ms in a single frame

Insertion Cost



```
//@igs(jmoore) - START - [IGRender][IGOptimization]
void MultiAddElement(const int32 ElementCount, TArray<bool>& OutResults, TArray<uint32>& OutBaseX, TArray<uint32>& OutBaseY, TArray<uint32>& OutBaseZ, TArray<uint32>& ElementSizeX, TArray<uint32>& Eleme
{
    SCOPED_NAMED_EVENT(TextureLayout3D_MultiAddElement, FColor::Emerald);

    for (int32 Index = 0; Index < ElementCount; ++Index)
    {
        if (ElementSizeX[Index] == 0 || ElementSizeY[Index] == 0 || ElementSizeZ[Index] == 0)
        {
            OutBaseX[Index] = 0;
            OutBaseY[Index] = 0;
            OutBaseZ[Index] = 0;
            OutResults[Index] = true;
        }

        if (bAlignByFour)
        {
            // Pad to 4 to ensure alignment
            ElementSizeX[Index] = (ElementSizeX[Index] + 3) & ~3;
            ElementSizeY[Index] = (ElementSizeY[Index] + 3) & ~3;
            ElementSizeZ[Index] = (ElementSizeZ[Index] + 3) & ~3;
        }
    }

    TArray<int32> NodeIndices;
    NodeIndices.Reserve(ElementCount);
    for (int32 Index = 0; Index < ElementCount; ++Index)
    {
        NodeIndices.Add(INDEX_NONE);
    }

    // Try allocating space without enlarging the texture.
    MultiAddSurfaceInner(0, NodeIndices, ElementSizeX, ElementSizeY, ElementSizeZ, false);

    if (NodeIndices.Contains(INDEX_NONE))
    {
        // Try allocating space which might enlarge the texture.
        MultiAddSurfaceInner(0, NodeIndices, ElementSizeX, ElementSizeY, ElementSizeZ, true);
    }
```



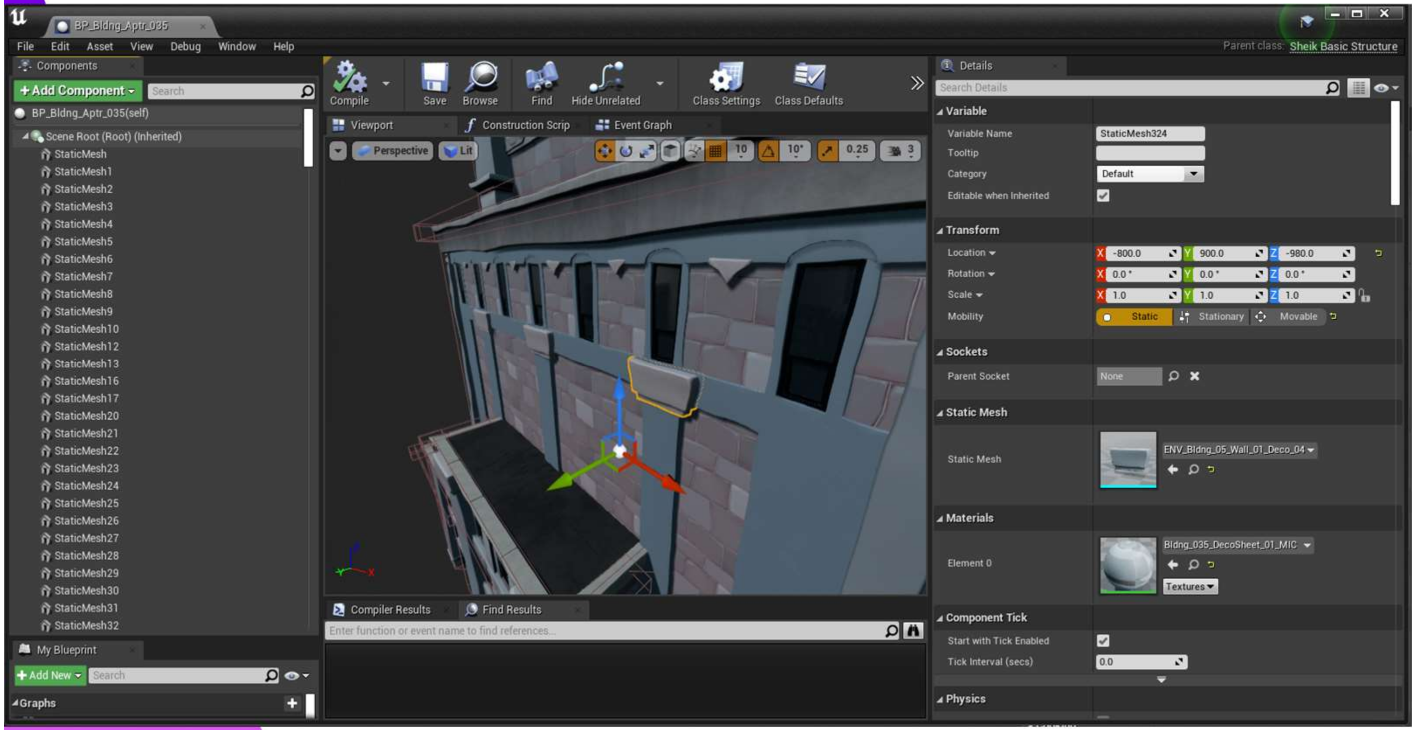
```
RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (6m 52s)
TGraphTask_ExecuteTask (17.1 ms)
TGraphTask_ExecuteTask (15.2 ms)
TGraphTask_ExecuteTask_DoTask (15.2 ms)
RenderViewFamily_RenderThread (15.2 ms)
FDeferredShadingSceneRenderer_Render (11.8 ms)
STAT_UpdateGPUScene (   FDistanceFieldVolumeTextureAtlas_UpdateAllocations (956.8 µs)
FS                      AllocateBlocks (10) (544.5 µs)                 CopyToUploadBuffers (366.2 µs)
                        TextureLayout3D_MultiAddElement (538.3 µs)     ParallelCopy (308.5 µs)
```

I got much better performance passing around a reference to the full list of desired allocations and checking each node against that list. This means more work being done when a node is visited, but less times traversing parts of the tree redundantly.

Here you can see that for a frame with 10 insertions, it is 2 ms faster than before.

Reactive Optimization: Too Many Primitives

I’ve talked extensively now about the pain points caused by our large numbers of distance fields, but let’s look at another render thread challenge we encountered – also related to our decisions with how we built the environment


BP_Bldng_Aptr_035
File Edit Asset View Debug Window Help
Parent class: Sheik Basic Structure
Components
+ Add Component
Compile Save Browse Find Hide Unrelated Class Settings Class Defaults
Viewport Construction Scrip Event Graph
Perspective Lit
Details
Compiler Results Find Results
Enter function or event name to find references...
My Blueprint
+ Add New
Graphs

BP_Bldng_Aptr_035(self)
Scene Root (Root) (Inherited)
StaticMesh
StaticMesh1
StaticMesh2
StaticMesh3
StaticMesh4
StaticMesh5
StaticMesh6
StaticMesh7
StaticMesh8
StaticMesh9
StaticMesh10
StaticMesh12
StaticMesh13
StaticMesh16
StaticMesh17
StaticMesh20
StaticMesh21
StaticMesh22
StaticMesh23
StaticMesh24
StaticMesh25
StaticMesh26
StaticMesh27
StaticMesh28
StaticMesh29
StaticMesh30
StaticMesh31
StaticMesh32

Variable
Variable Name: StaticMesh324
Tooltip:
Category: Default
Editable when Inherited: ✓

Transform
Location: X -800.0 Y 900.0 Z -980.0
Rotation: X 0.0° Y 0.0° Z 0.0°
Scale: X 1.0 Y 1.0 Z 1.0
Mobility: Static Stationary Movable

Sockets
Parent Socket: None

Static Mesh
Static Mesh: ENV_Bldng_05_Wall_01_Deco_04

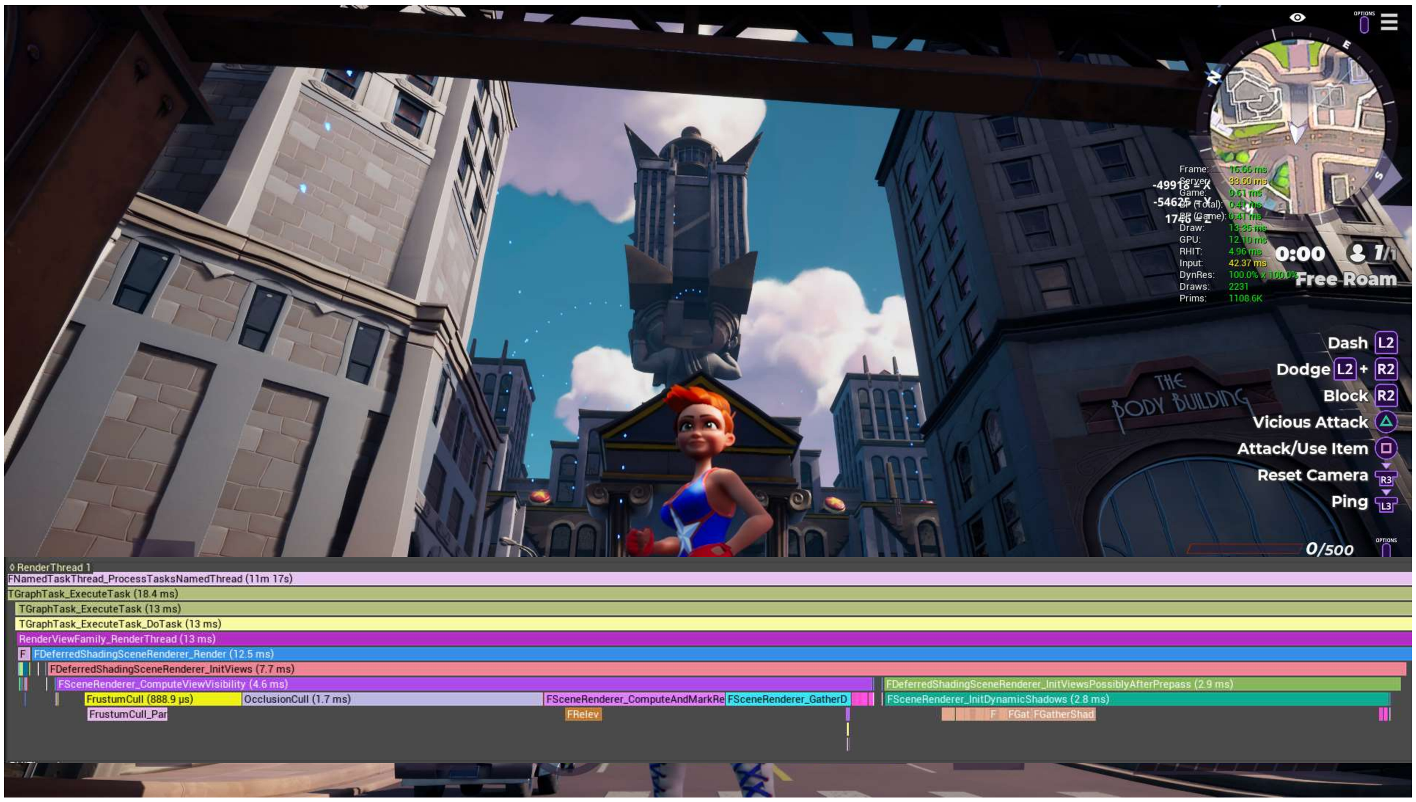
Materials
Element 0: Bldng_035_DecoSheet_01_MIC
Textures

Component Tick
Start with Tick Enabled: ✓
Tick Interval (secs): 0.0

Physics

As a reminder – our assets are constructed out of many smaller mesh components kit bashed together.

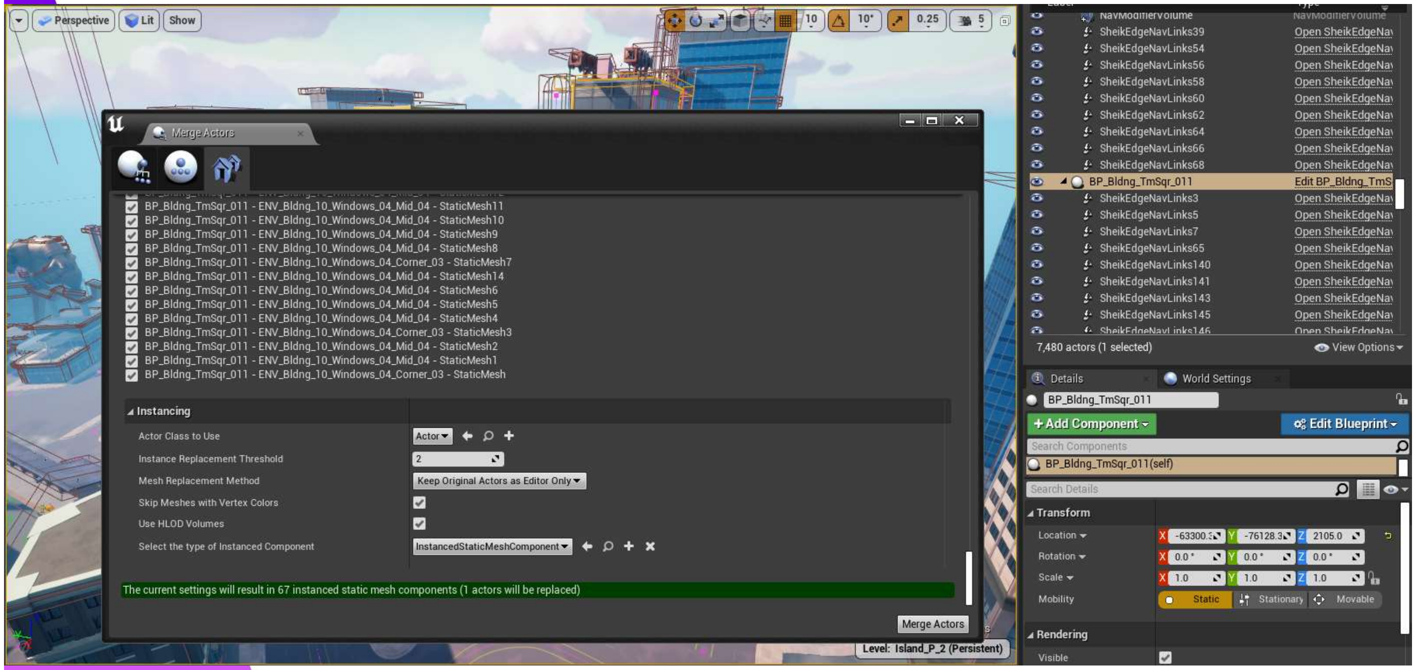
The scope of the project had expanded after our vertical slice milestone and our art team pushed for even more polish and detail than was in our original plan. We started hitting the limits of dynamic instancing and became largely bottlenecked on what is known as “Init Views” in Unreal Engine – this is where frustum and occlusion culling takes place.



```
Frame:        16.66 ms
Server:       33.60 ms
Game:         9.61 ms
BP (Total):   0.41 ms
BP (Game):    0.41 ms
Draw:         13.85 ms
GPU:          12.10 ms
RHIT:         4.96 ms
Input:        42.37 ms
DynRes:       100.0% x 100.0%
Draws:        2231
Prims:        1108.6K
```

```
◊ RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (11m 17s)
TGraphTask_ExecuteTask (18.4 ms)
 TGraphTask_ExecuteTask (13 ms)
 TGraphTask_ExecuteTask_DoTask (13 ms)
 RenderViewFamily_RenderThread (13 ms)
 F FDeferredShadingSceneRenderer_Render (12.5 ms)
    FDeferredShadingSceneRenderer_InitViews (7.7 ms)
     FSceneRenderer_ComputeViewVisibility (4.6 ms)                 FDeferredShadingSceneRenderer_InitViewsPossiblyAfterPrepass (2.9 ms)
        FrustumCull (888.9 µs)  OcclusionCull (1.7 ms)  FSceneRenderer_ComputeAndMarkRe FSceneRenderer_GatherD   FSceneRenderer_InitDynamicShadows (2.8 ms)
         FrustumCull_Par                                 FRelev                                    F FGat FGatherShad
```

Here is an example of a challenging shot in front of some buildings that are made up of many primitives that are processed individually before instancing. You can see here that InitViews is taking up a hefty 7.7 ms, which is approaching half of our total frame time.



```
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh11
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh10
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh9
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh8
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Corner_03 - StaticMesh7
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh14
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh6
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh5
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh4
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Corner_03 - StaticMesh3
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh2
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Mid_04 - StaticMesh1
☑ BP_Bldng_TmSqr_011 - ENV_Bldng_10_Windows_04_Corner_03 - StaticMesh
```

Instancing

Actor Class to Use	Actor
Instance Replacement Threshold	2
Mesh Replacement Method	Keep Original Actors as Editor Only
Skip Meshes with Vertex Colors	✓
Use HLOD Volumes	✓
Select the type of Instanced Component	InstancedStaticMeshComponent

The current settings will result in 67 instanced static mesh components (1 actors will be replaced)

Merge Actors

Level: Island_P_2 (Persistent)

NavModifierVolume	NavModifierVolume
SheikEdgeNavLinks39	Open SheikEdgeNav
SheikEdgeNavLinks54	Open SheikEdgeNav
SheikEdgeNavLinks56	Open SheikEdgeNav
SheikEdgeNavLinks58	Open SheikEdgeNav
SheikEdgeNavLinks60	Open SheikEdgeNav
SheikEdgeNavLinks62	Open SheikEdgeNav
SheikEdgeNavLinks64	Open SheikEdgeNav
SheikEdgeNavLinks66	Open SheikEdgeNav
SheikEdgeNavLinks68	Open SheikEdgeNav
BP_Bldng_TmSqr_011	Edit BP_Bldng_TmS
SheikEdgeNavLinks3	Open SheikEdgeNav
SheikEdgeNavLinks5	Open SheikEdgeNav
SheikEdgeNavLinks7	Open SheikEdgeNav
SheikEdgeNavLinks65	Open SheikEdgeNav
SheikEdgeNavLinks140	Open SheikEdgeNav
SheikEdgeNavLinks141	Open SheikEdgeNav
SheikEdgeNavLinks143	Open SheikEdgeNav
SheikEdgeNavLinks145	Open SheikEdgeNav
SheikEdgeNavLinks146	Open SheikEdgeNav

7,480 actors (1 selected) View Options

Details World Settings

BP_Bldng_TmSqr_011

+ Add Component Edit Blueprint

Search Components

BP_Bldng_TmSqr_011(self)

Search Details

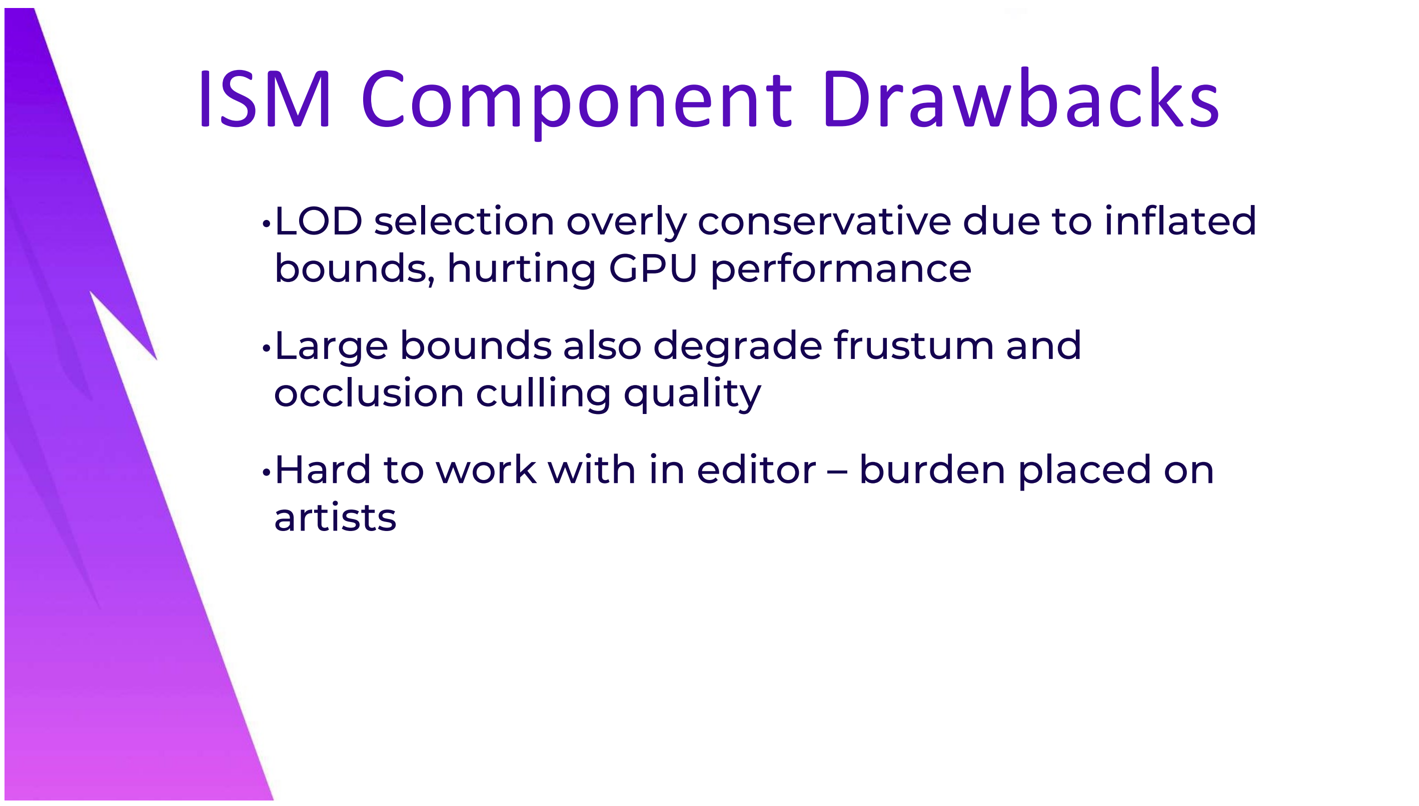
Transform

Location	X -63300.5	Y -76128.3	Z 2105.0
Rotation	X 0.0°	Y 0.0°	Z 0.0°
Scale	X 1.0	Y 1.0	Z 1.0
Mobility	Static	Stationary	Movable

Rendering

Visible ✓

Now – Unreal 4 has an older system for managing costs of large numbers of primitives, which is to merge compatible components into a single “instanced static mesh component.” These manually instance their meshes, ignoring the dynamic instancing system, and go through the renderer as a single primitive.



ISM Component Drawbacks

- LOD selection overly conservative due to inflated bounds, hurting GPU performance
- Large bounds also degrade frustum and occlusion culling quality
- Hard to work with in editor – burden placed on artists

However, stock Instanced Static Mesh components have a number of drawbacks compared to using individual mesh components that dynamically instance. The first two are that GPU performance degrades – the larger bounds on the merged primitive results in an overly conservative LOD selection and culling result. It also places a lot of burden on the artists to ask them to do this manually, which is orthogonal to the performance characteristics but still a concern.

GPU-driven rendering

- GPU driven rendering of meshes with MultiDrawIndirect can allow us to handle culling + LOD selection on the GPU
- Seb Aaltonen first presented many of these ideas in the talk *GPU-Driven Rendering Pipelines*
- But we don’t have the resources to rewrite UE4!

The Cadillac solution is to instead go down the path of full GPU-driven rendering. Issue just a small number of commands on the CPU and handle LOD selection and culling on the GPU instead of dealing with expanded bounds on the primitives on the CPU. Seb Aaltonen pioneered a lot of these ideas in his part of the talk *GPU-Driven Rendering Pipelines* from Siggraph 2015. But our little didn’t have the resources to do that ambitious of a re-write of the UE4 renderer.

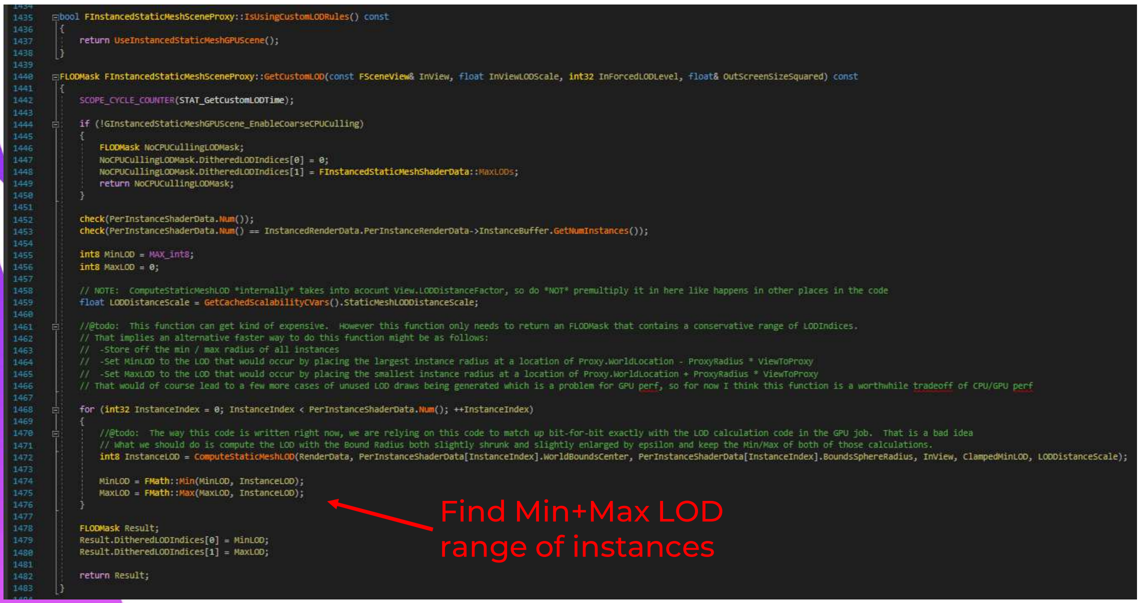
http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pptx

GPU-driven ISM Component

- Go halfway – modify ISM component to render with a special ISM GPU Scene
- Convert compatible components on Basic Structure actors into ISM components at cook time
- Render each ISM with DrawIndirect
- Keep ISM’s grouped into the Basic Structures (do not merge buildings) so coarse CPU LOD selection and culling is still effective

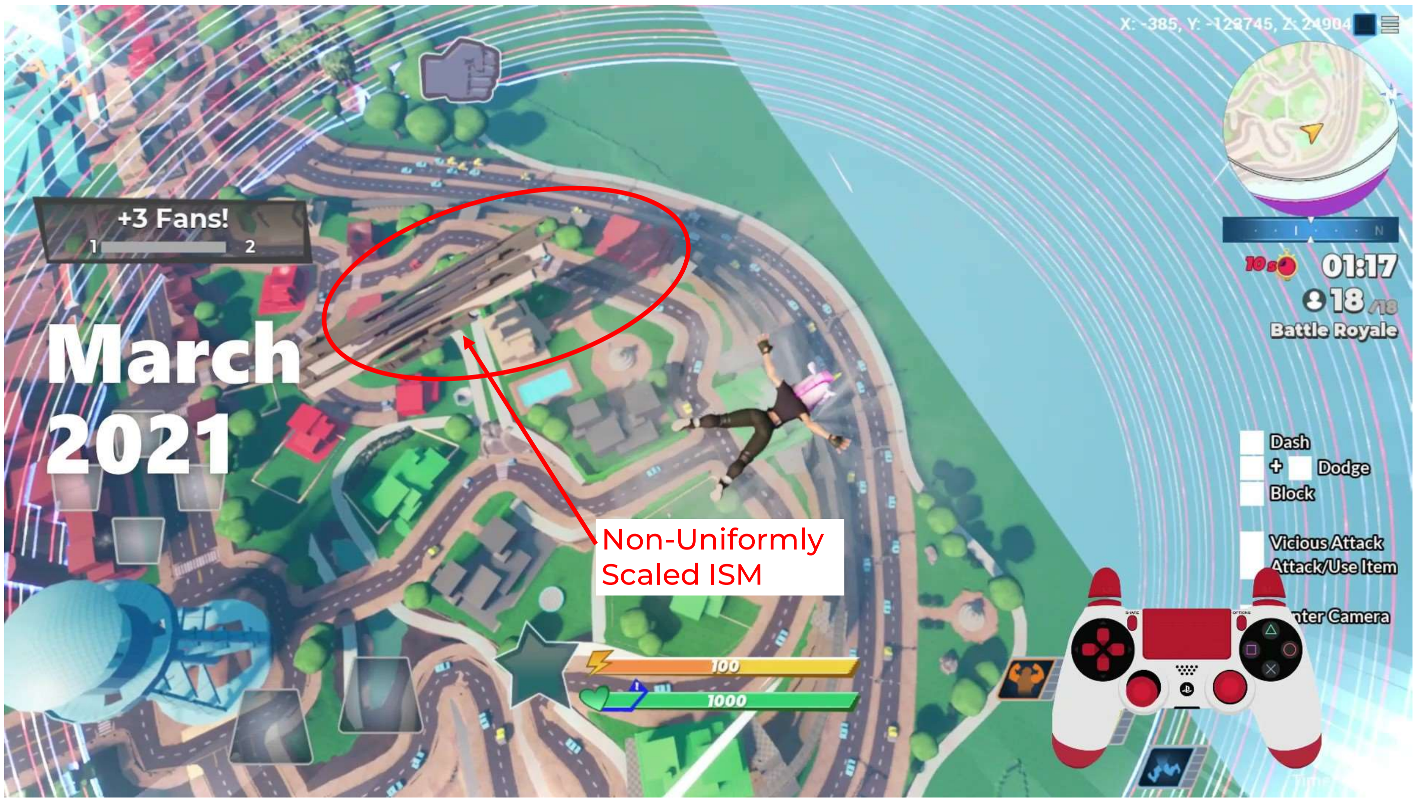
My colleague Nate Mefford figured out a really great compromise that fit our project, to go halfway and modify ISM component to render with a special ISM GPU Scene. We convert compatible components on Basic Structure actors into ISM components at cook time, which keeps the burden off of the artists. We keep ISM’s grouped into their owning Basic Structures, so components that could have dynamically instanced between two neighboring buildings within view do not do so. This is because buildings offer a nice unit of coarse CPU-side LOD selection and culling which helps keep our GPU overhead low.

CPU LOD Selection



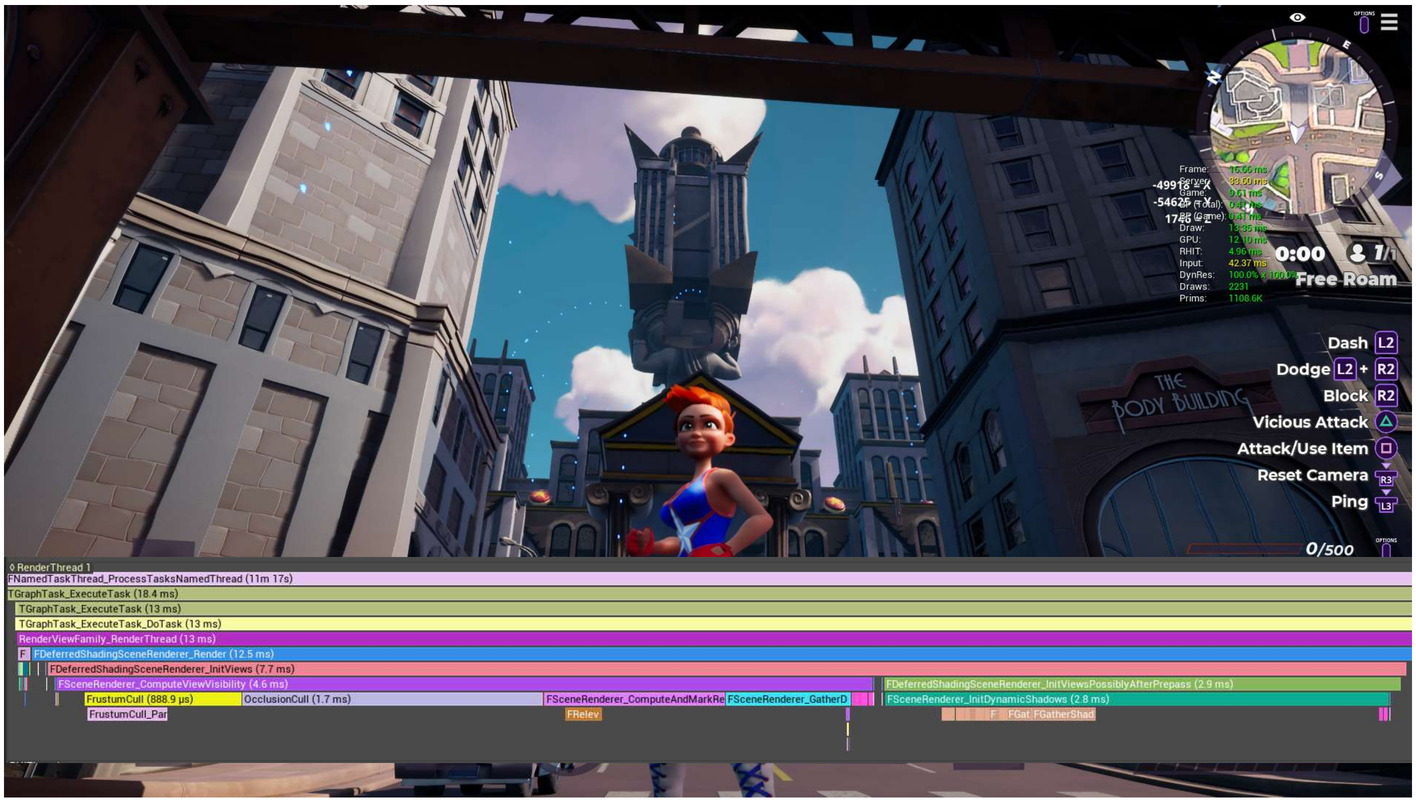
```
bool FInstancedStaticMeshSceneProxy::IsUsingCustomLODRules() const
{
    return UseInstancedStaticMeshGPUScene();
}

FLODMask FInstancedStaticMeshSceneProxy::GetCustomLOD(const FSceneView& InView, float InViewLODScale, int32 InForcedLODLevel, float& OutScreenSizeSquared) const
{
    SCOPE_CYCLE_COUNTER(STAT_GetCustomLODTime);

    if (!GInstancedStaticMeshGPUScene_EnableCoarseCPUCulling)
    {
        FLODMask NoCPUCullingLODMask;
        NoCPUCullingLODMask.DitheredLODIndices[0] = 0;
        NoCPUCullingLODMask.DitheredLODIndices[1] = FInstancedStaticMeshShaderData::MaxLODs;
        return NoCPUCullingLODMask;
    }

    check(PerInstanceShaderData.Num());
    check(PerInstanceShaderData.Num() == InstancedRenderData.PerInstanceRenderData->InstanceBuffer.GetNumInstances());

    int8 MinLOD = MAX_int8;
    int8 MaxLOD = 0;

    // NOTE:  ComputeStaticMeshLOD *internally* takes into acocunt View.LODDistanceFactor, so do *NOT* premultiply it in here like happens in other places in the code
    float LODDistanceScale = GetCachedScalabilityCVars().StaticMeshLODDistanceScale;

    //@todo:  This function can get kind of expensive.  However this function only needs to return an FLODMask that contains a conservative range of LODIndices.
    // That implies an alternative faster way to do this function might be as follows:
    //  -Store off the min / max radius of all instances
    //  -Set MinLOD to the LOD that would occur by placing the largest instance radius at a location of Proxy.WorldLocation - ProxyRadius * ViewToProxy
    //  -Set MaxLOD to the LOD that would occur by placing the smallest instance radius at a location of Proxy.WorldLocation + ProxyRadius * ViewToProxy
    // That would of course lead to a few more cases of unused LOD draws being generated which is a problem for GPU perf, so for now I think this function is a worthwhile tradeoff of CPU/GPU perf

    for (int32 InstanceIndex = 0; InstanceIndex < PerInstanceShaderData.Num(); ++InstanceIndex)
    {
        //@todo:  The way this code is written right now, we are relying on this code to match up bit-for-bit exactly with the LOD calculation code in the GPU job.  That is a bad idea
        // What we should do is compute the LOD with the Bound Radius both slightly shrunk and slightly enlarged by epsilon and keep the Min/Max of both of those calculations.
        int8 InstanceLOD = ComputeStaticMeshLOD(RenderData, PerInstanceShaderData[InstanceIndex].WorldBoundsCenter, PerInstanceShaderData[InstanceIndex].BoundsSphereRadius, InView, ClampedMinLOD, LODDistanceScale);

        MinLOD = FMath::Min(MinLOD, InstanceLOD);
        MaxLOD = FMath::Max(MaxLOD, InstanceLOD);
    }

    FLODMask Result;
    Result.DitheredLODIndices[0] = MinLOD;
    Result.DitheredLODIndices[1] = MaxLOD;

    return Result;
}
```

When I say that coarse CPU LOD selection is important, I mean that we can’t just send a draw per LOD to the command processor as that will result in too many empty batches. This code is running per-instance LOD selection on the CPU. This is a CPU vs GPU balancing act and is one clear area where the full GPU driven rendering pipeline would allow bigger gains. Right now we have to decide between being overly conservative and sending extra batches, or spending more CPU time on LOD selection eating into our savings on the render thread.



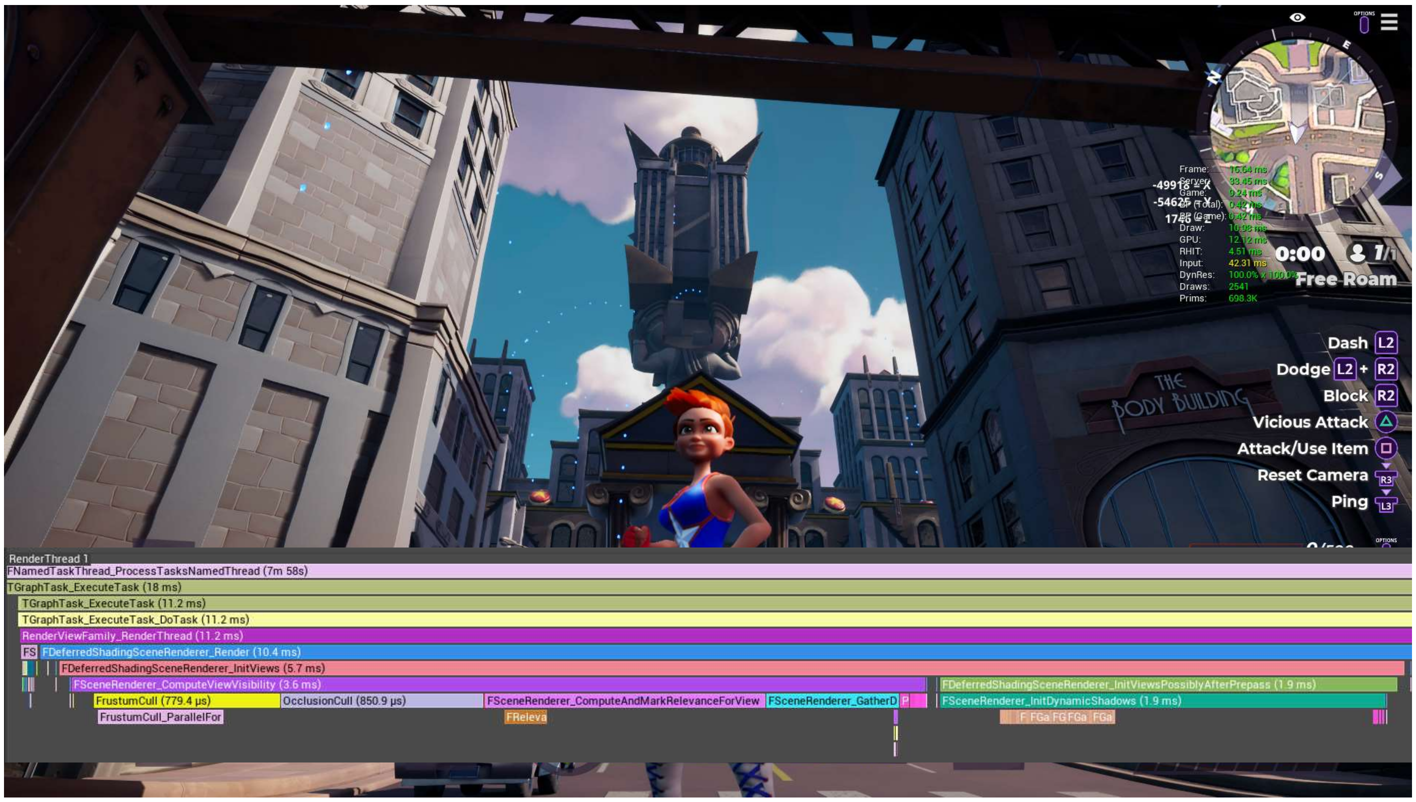
I should also mention there were lots of fiddly bits with getting negative and non-uniform scale to work correctly in the conversion to Instanced Static Meshes. I’m a believer in supporting both in a renderer, but this work really challenged my faith in that and we hit a lot of under-exercised code paths getting everything to render correctly when converted from individual static meshes to ISM’s. You can see a shot from a playtest where a storefront has stretched out into the sky above a neighborhood due to the static meshes going into the ISM conversion having non-uniform scale.

I’ve included Nate’s Unreal-specific workarounds in the appendix of these slides for those interested.



```
◊ RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (11m 17s)
TGraphTask_ExecuteTask (18.4 ms)
 TGraphTask_ExecuteTask (13 ms)
 TGraphTask_ExecuteTask_DoTask (13 ms)
 RenderViewFamily_RenderThread (13 ms)
 F FDeferredShadingSceneRenderer_Render (12.5 ms)
   FDeferredShadingSceneRenderer_InitViews (7.7 ms)
    FSceneRenderer_ComputeViewVisibility (4.6 ms)                                   FDeferredShadingSceneRenderer_InitViewsPossiblyAfterPrepass (2.9 ms)
      FrustumCull (888.9 µs) OcclusionCull (1.7 ms) FSceneRenderer_ComputeAndMarkRe FSceneRenderer_GatherD   FSceneRenderer_InitDynamicShadows (2.8 ms)
       FrustumCull_Par                               FRelev                                    F FGat FGatherShad
```

Now before I get into some more implementation details, here are the results for that challenging shot, starting with what we had before – InitViews taking 7.7 ms

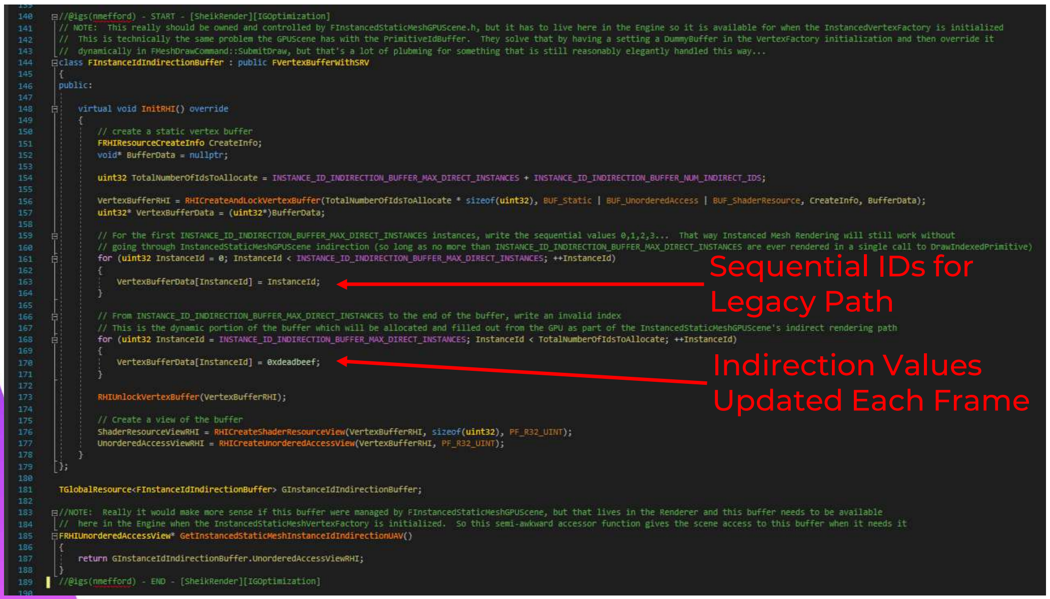


```
Frame:        16.64 ms
Server:       33.45 ms
Game:         9.24 ms
BP (Total):   0.42 ms
BP (Game):    0.42 ms
Draw:         10.98 ms
GPU:          12.12 ms
RHIT:         4.51 ms
Input:        42.31 ms
DynRes:       100.0% x 100.0%
Draws:        2541
Prims:        698.3K
```

```
RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (7m 58s)
TGraphTask_ExecuteTask (18 ms)
  TGraphTask_ExecuteTask (11.2 ms)
  TGraphTask_ExecuteTask_DoTask (11.2 ms)
  RenderViewFamily_RenderThread (11.2 ms)
  FS FDeferredShadingSceneRenderer_Render (10.4 ms)
       FDeferredShadingSceneRenderer_InitViews (5.7 ms)
        FSceneRenderer_ComputeViewVisibility (3.6 ms)                                    FDeferredShadingSceneRenderer_InitViewsPossiblyAfterPrepass (1.9 ms)
           FrustumCull (779.4 µs) OcclusionCull (850.9 µs) FSceneRenderer_ComputeAndMarkRelevanceForView FSceneRenderer_GatherD P   FSceneRenderer_InitDynamicShadows (1.9 ms)
            FrustumCull_ParallelFor                FReleva                                        F FGa FG FGa FGa
```

And here it is with ISM components on the Basic Structures. This reduces render thread time by 2.37 ms. It varies from scene to scene but it is generally in the 2-3 ms range in savings on a base PS4. And this particular shot sees GPU time virtually flat, although I see it sometimes take 0.1-0.25 ms longer GPU time, which I think is very acceptable given the CPU benefits.

Instance ID Indirection

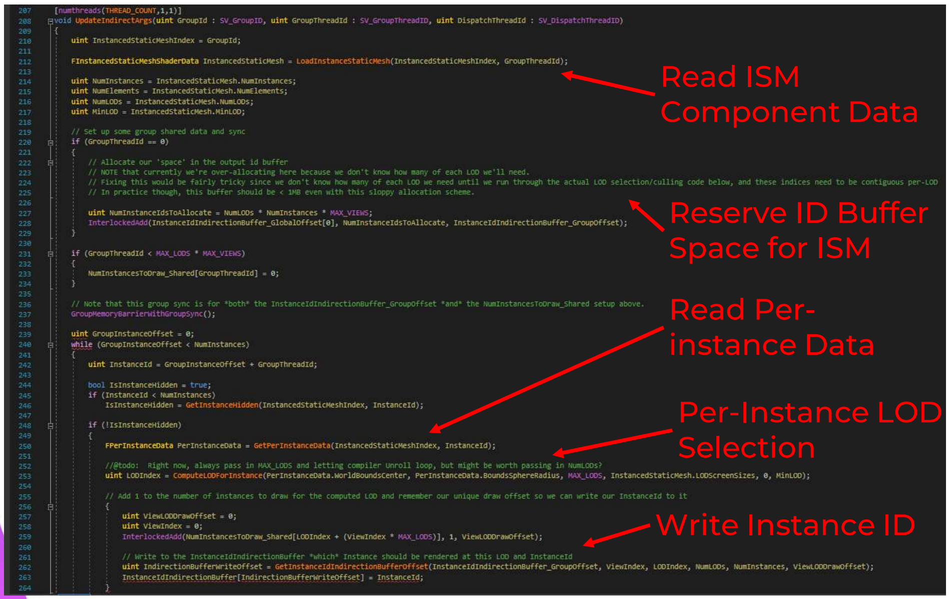


```
//@igs(nmefford) - START - [SheikRender][IGOptimization]
// NOTE:  This really should be owned and controlled by FInstancedStaticMeshGPUScene.h, but it has to live here in the Engine so it is available for when the InstancedVertexFactory is initialized
//  This is technically the same problem the GPUScene has with the PrimitiveIdBuffer.  They solve that by having a setting a DummyBuffer in the VertexFactory initialization and then override it
//  dynamically in FMeshDrawCommand::SubmitDraw, but that's a lot of plubming for something that is still reasonably elegantly handled this way...
class FInstanceIdIndirectionBuffer : public FVertexBufferWithSRV
{
public:

    virtual void InitRHI() override
    {
        // create a static vertex buffer
        FRHIResourceCreateInfo CreateInfo;
        void* BufferData = nullptr;

        uint32 TotalNumberOfIdsToAllocate = INSTANCE_ID_INDIRECTION_BUFFER_MAX_DIRECT_INSTANCES + INSTANCE_ID_INDIRECTION_BUFFER_NUM_INDIRECT_IDS;

        VertexBufferRHI = RHICreateAndLockVertexBuffer(TotalNumberOfIdsToAllocate * sizeof(uint32), BUF_Static | BUF_UnorderedAccess | BUF_ShaderResource, CreateInfo, BufferData);
        uint32* VertexBufferData = (uint32*)BufferData;

        // For the first INSTANCE_ID_INDIRECTION_BUFFER_MAX_DIRECT_INSTANCES instances, write the sequential values 0,1,2,3...  That way Instanced Mesh Rendering will still work without
        // going through InstancedStaticMeshGPUScene indirection (so long as no more than INSTANCE_ID_INDIRECTION_BUFFER_MAX_DIRECT_INSTANCES are ever rendered in a single call to DrawIndexedPrimitive)
        for (uint32 InstanceId = 0; InstanceId < INSTANCE_ID_INDIRECTION_BUFFER_MAX_DIRECT_INSTANCES; ++InstanceId)
        {
            VertexBufferData[InstanceId] = InstanceId;
        }

        // From INSTANCE_ID_INDIRECTION_BUFFER_MAX_DIRECT_INSTANCES to the end of the buffer, write an invalid index
        // This is the dynamic portion of the buffer which will be allocated and filled out from the GPU as part of the InstancedStaticMeshGPUScene's indirect rendering path
        for (uint32 InstanceId = INSTANCE_ID_INDIRECTION_BUFFER_MAX_DIRECT_INSTANCES; InstanceId < TotalNumberOfIdsToAllocate; ++InstanceId)
        {
            VertexBufferData[InstanceId] = 0xdeadbeef;
        }

        RHIUnlockVertexBuffer(VertexBufferRHI);

        // Create a view of the buffer
        ShaderResourceViewRHI = RHICreateShaderResourceView(VertexBufferRHI, sizeof(uint32), PF_R32_UINT);
        UnorderedAccessViewRHI = RHICreateUnorderedAccessView(VertexBufferRHI, PF_R32_UINT);
    }
};

TGlobalResource<FInstanceIdIndirectionBuffer> GInstanceIdIndirectionBuffer;

//NOTE:  Really it would make more sense if this buffer were managed by FInstancedStaticMeshGPUScene, but that lives in the Renderer and this buffer needs to be available
//  here in the Engine when the InstancedStaticMeshVertexFactory is initialized.  So this semi-awkward accessor function gives the scene access to this buffer when it needs it
FRHIUnorderedAccessView* GetInstancedStaticMeshInstanceIdIndirectionUAV()
{
    return GInstanceIdIndirectionBuffer.UnorderedAccessViewRHI;
}
//@igs(nmefford) - END - [SheikRender][IGOptimization]
```

Sequential IDs for Legacy Path
Indirection Values Updated Each Frame

The ISM component proxy code has a key modification in that we bind an instance indirection buffer. Regular old ISM's in UE4 will just access 0 to N instance ID's sequentially. To continue supporting this, our indirection buffer sets up a number of ID's at the front, this code path is still important to us for our grass rendering in particular. But any ISM's using the GPU scene will instead read IDs from beyond that point into the portion of the buffer that is dynamically updated by our scene update compute shader that runs each frame.

Update Args Shader

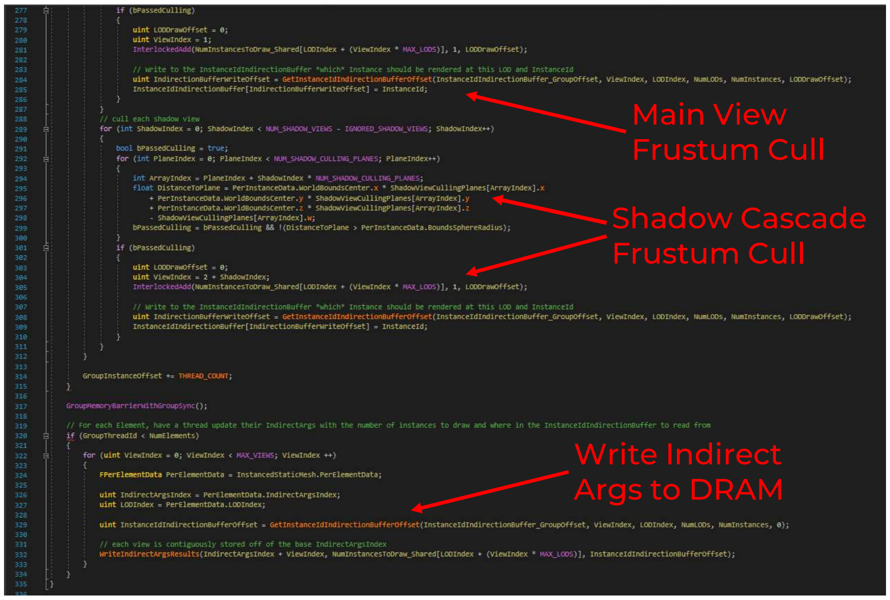


```
[numthreads(THREAD_COUNT,1,1)]
void UpdateIndirectArgs(uint GroupId : SV_GroupID, uint GroupThreadId : SV_GroupThreadID, uint DispatchThreadId : SV_DispatchThreadID)
{
    uint InstancedStaticMeshIndex = GroupId;

    FInstancedStaticMeshShaderData InstancedStaticMesh = LoadInstanceStaticMesh(InstancedStaticMeshIndex, GroupThreadId);

    uint NumInstances = InstancedStaticMesh.NumInstances;
    uint NumElements = InstancedStaticMesh.NumElements;
    uint NumLODs = InstancedStaticMesh.NumLODs;
    uint MinLOD = InstancedStaticMesh.MinLOD;

    // Set up some group shared data and sync
    if (GroupThreadId == 0)
    {
        // Allocate our 'space' in the output id buffer
        // NOTE that currently we're over-allocating here because we don't know how many of each LOD we'll need.
        // Fixing this would be fairly tricky since we don't know how many of each LOD we need until we run through the actual LOD selection/culling code below, and these indices need to be contiguous per-LOD
        // In practice though, this buffer should be < 1MB even with this sloppy allocation scheme.

        uint NumInstanceIdsToAllocate = NumLODs * NumInstances * MAX_VIEWS;
        InterlockedAdd(InstanceIdIndirectionBuffer_GlobalOffset[0], NumInstanceIdsToAllocate, InstanceIdIndirectionBuffer_GroupOffset);
    }

    if (GroupThreadId < MAX_LODS * MAX_VIEWS)
    {
        NumInstancesToDraw_Shared[GroupThreadId] = 0;
    }

    // Note that this group sync is for *both* the InstanceIdIndirectionBuffer_GroupOffset *and* the NumInstancesToDraw_Shared setup above.
    GroupMemoryBarrierWithGroupSync();

    uint GroupInstanceOffset = 0;
    while (GroupInstanceOffset < NumInstances)
    {
        uint InstanceId = GroupInstanceOffset + GroupThreadId;

        bool IsInstanceHidden = true;
        if (InstanceId < NumInstances)
            IsInstanceHidden = GetInstanceHidden(InstancedStaticMeshIndex, InstanceId);

        if (!IsInstanceHidden)
        {
            FPerInstanceData PerInstanceData = GetPerInstanceData(InstancedStaticMeshIndex, InstanceId);

            //@todo:  Right now, always pass in MAX_LODS and letting compiler Unroll loop, but might be worth passing in NumLODs?
            uint LODIndex = ComputeLODForInstance(PerInstanceData.WorldBoundsCenter, PerInstanceData.BoundsSphereRadius, MAX_LODS, InstancedStaticMesh.LODScreenSizes, 0, MinLOD);

            // Add 1 to the number of instances to draw for the computed LOD and remember our unique draw offset so we can write our InstanceId to it
            {
                uint ViewLODDrawOffset = 0;
                uint ViewIndex = 0;
                InterlockedAdd(NumInstancesToDraw_Shared[LODIndex + (ViewIndex * MAX_LODS)], 1, ViewLODDrawOffset);

                // Write to the InstanceIdIndirectionBuffer *which* Instance should be rendered at this LOD and InstanceId
                uint IndirectionBufferWriteOffset = GetInstanceIdIndirectionBufferOffset(InstanceIdIndirectionBuffer_GroupOffset, ViewIndex, LODIndex, NumLODs, NumInstances, ViewLODDrawOffset);
                InstanceIdIndirectionBuffer[IndirectionBufferWriteOffset] = InstanceId;
            }
```

Let’s take a look at that shader. Each ISM component gets a single group (the size of 1 wavefront) that allocates space into the ID buffer. This is done atomically as multiple groups are in flight at once. The some LDS is cleared to track the IndirectArgs and a barrier is inserted.

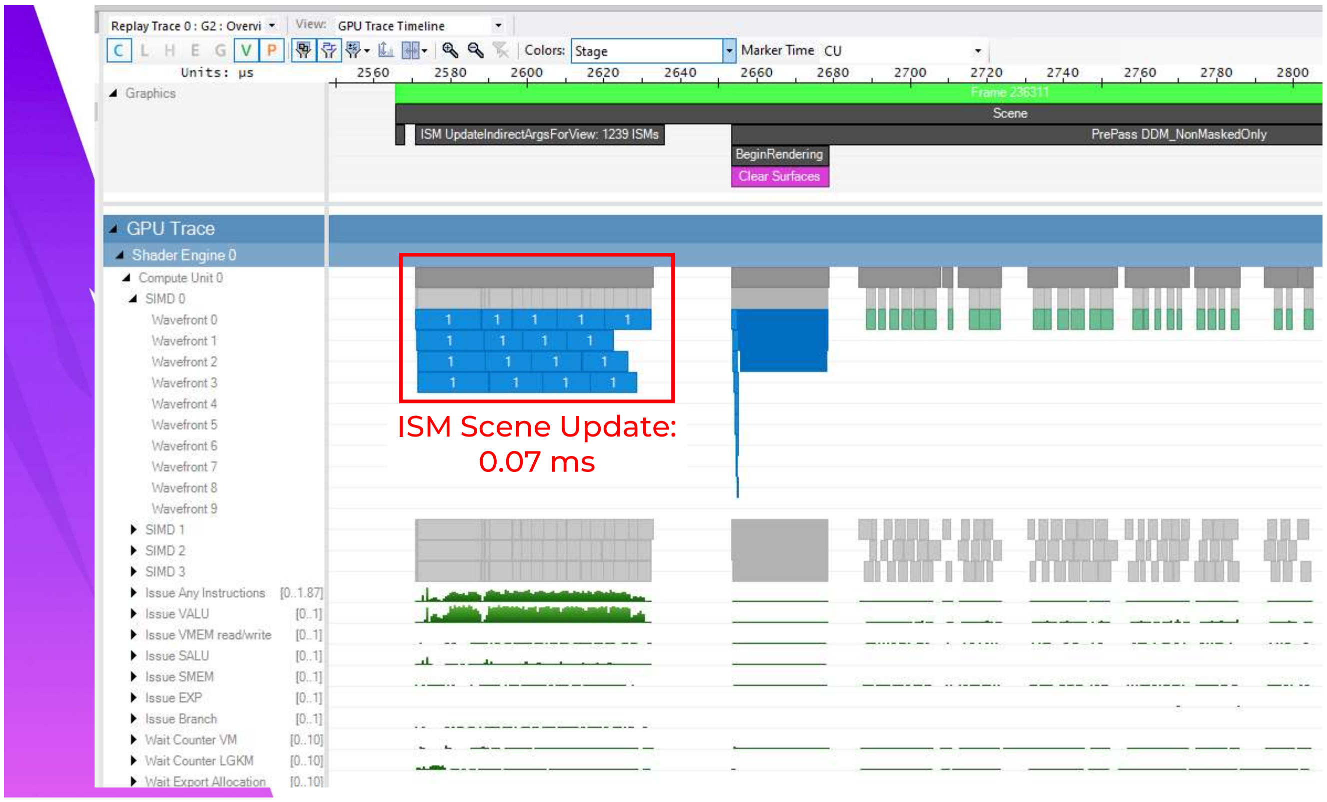
After this point we loop over the instances and perform LOD selection. The appropriate indirect arg counter in LDS is atomically incremented, and the instance ID is written to the indirection buffer. You might notice that this has similarities to the tile locations buffer from the tile classification code.

Update Args Shader



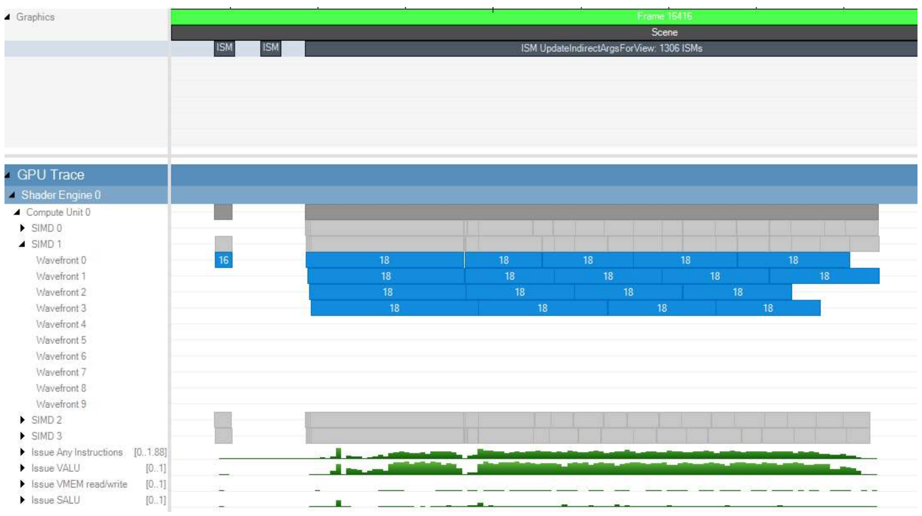
```
                if (bPassedCulling)
                {
                    uint LODDrawOffset = 0;
                    uint ViewIndex = 1;
                    InterlockedAdd(NumInstancesToDraw_Shared[LODIndex + (ViewIndex * MAX_LODS)], 1, LODDrawOffset);

                    // Write to the InstanceIdIndirectionBuffer *which* Instance should be rendered at this LOD and InstanceId
                    uint IndirectionBufferWriteOffset = GetInstanceIdIndirectionBufferOffset(InstanceIdIndirectionBuffer_GroupOffset, ViewIndex, LODIndex, NumLODs, NumInstances, LODDrawOffset);
                    InstanceIdIndirectionBuffer[IndirectionBufferWriteOffset] = InstanceId;
                }
            }
            // cull each shadow view
            for (int ShadowIndex = 0; ShadowIndex < NUM_SHADOW_VIEWS - IGNORED_SHADOW_VIEWS; ShadowIndex++)
            {
                bool bPassedCulling = true;
                for (int PlaneIndex = 0; PlaneIndex < NUM_SHADOW_CULLING_PLANES; PlaneIndex++)
                {
                    int ArrayIndex = PlaneIndex + ShadowIndex * NUM_SHADOW_CULLING_PLANES;
                    float DistanceToPlane = PerInstanceData.WorldBoundsCenter.x * ShadowViewCullingPlanes[ArrayIndex].x
                        + PerInstanceData.WorldBoundsCenter.y * ShadowViewCullingPlanes[ArrayIndex].y
                        + PerInstanceData.WorldBoundsCenter.z * ShadowViewCullingPlanes[ArrayIndex].z
                        - ShadowViewCullingPlanes[ArrayIndex].w;
                    bPassedCulling = bPassedCulling && !(DistanceToPlane > PerInstanceData.BoundsSphereRadius);
                }
                if (bPassedCulling)
                {
                    uint LODDrawOffset = 0;
                    uint ViewIndex = 2 + ShadowIndex;
                    InterlockedAdd(NumInstancesToDraw_Shared[LODIndex + (ViewIndex * MAX_LODS)], 1, LODDrawOffset);

                    // Write to the InstanceIdIndirectionBuffer *which* Instance should be rendered at this LOD and InstanceId
                    uint IndirectionBufferWriteOffset = GetInstanceIdIndirectionBufferOffset(InstanceIdIndirectionBuffer_GroupOffset, ViewIndex, LODIndex, NumLODs, NumInstances, LODDrawOffset);
                    InstanceIdIndirectionBuffer[IndirectionBufferWriteOffset] = InstanceId;
                }
            }
        }

        GroupInstanceOffset += THREAD_COUNT;
    }

    GroupMemoryBarrierWithGroupSync();

    // For each Element, have a thread update their IndirectArgs with the number of instances to draw and where in the InstanceIdIndirectionBuffer to read from
    if (GroupThreadId < NumElements)
    {
        for (uint ViewIndex = 0; ViewIndex < MAX_VIEWS; ViewIndex ++)
        {
            FPerElementData PerElementData = InstancedStaticMesh.PerElementData;

            uint IndirectArgsIndex = PerElementData.IndirectArgsIndex;
            uint LODIndex = PerElementData.LODIndex;

            uint InstanceIdIndirectionBufferOffset = GetInstanceIdIndirectionBufferOffset(InstanceIdIndirectionBuffer_GroupOffset, ViewIndex, LODIndex, NumLODs, NumInstances, 0);

            // each view is contiguously stored off of the base IndirectArgsIndex
            WriteIndirectArgsResults(IndirectArgsIndex + ViewIndex, NumInstancesToDraw_Shared[LODIndex + (ViewIndex * MAX_LODS)], InstanceIdIndirectionBufferOffset);
        }
    }
}
```

While the first arg holds all the instances for a given ISM, we also perform per-view culling for our main + shadow cascade views to further reduce wasted verts when possible. Finally we have one more barrier and write the indirect args from LDS into main memory.



The ISM scene update shader is generally pretty fast to run, in this frame it only took 0.07 ms to run. We also have to handle scattered updates of the mesh description data when a new ISM component is added, but that is not a per-frame operation, and is also very fast, just 0.01 ms in traces I took recently.

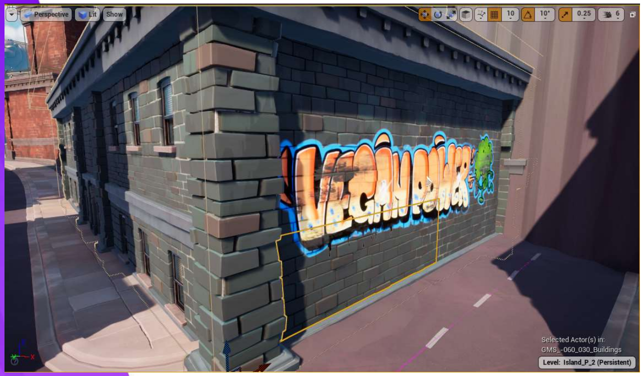
While I'm very with the results for this system, we have ideas for further improvements – dynamic merging of loose StaticMeshActors could help improve prop rendering, and the system as it is currently only does occlusion culling on the bounds of the full structure, so some form of per-instance occlusion culling could be beneficial.

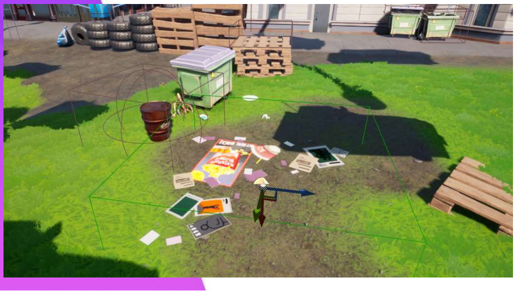


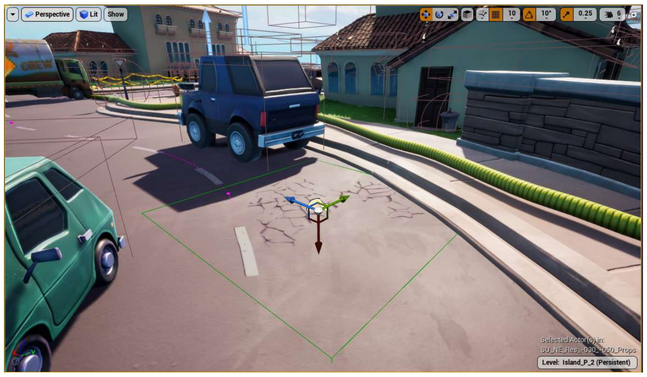
Here you can see one such trace where two small updates to the mesh descriptions are patched before the scene update shader

Reactive Optimization: Decals

Finally, one last problem I wanted to touch on was unanticipated challenges with decals late in our development cycle. My colleague Karinne and I made a number of improvements in service of getting systems to behave as we expected them to.

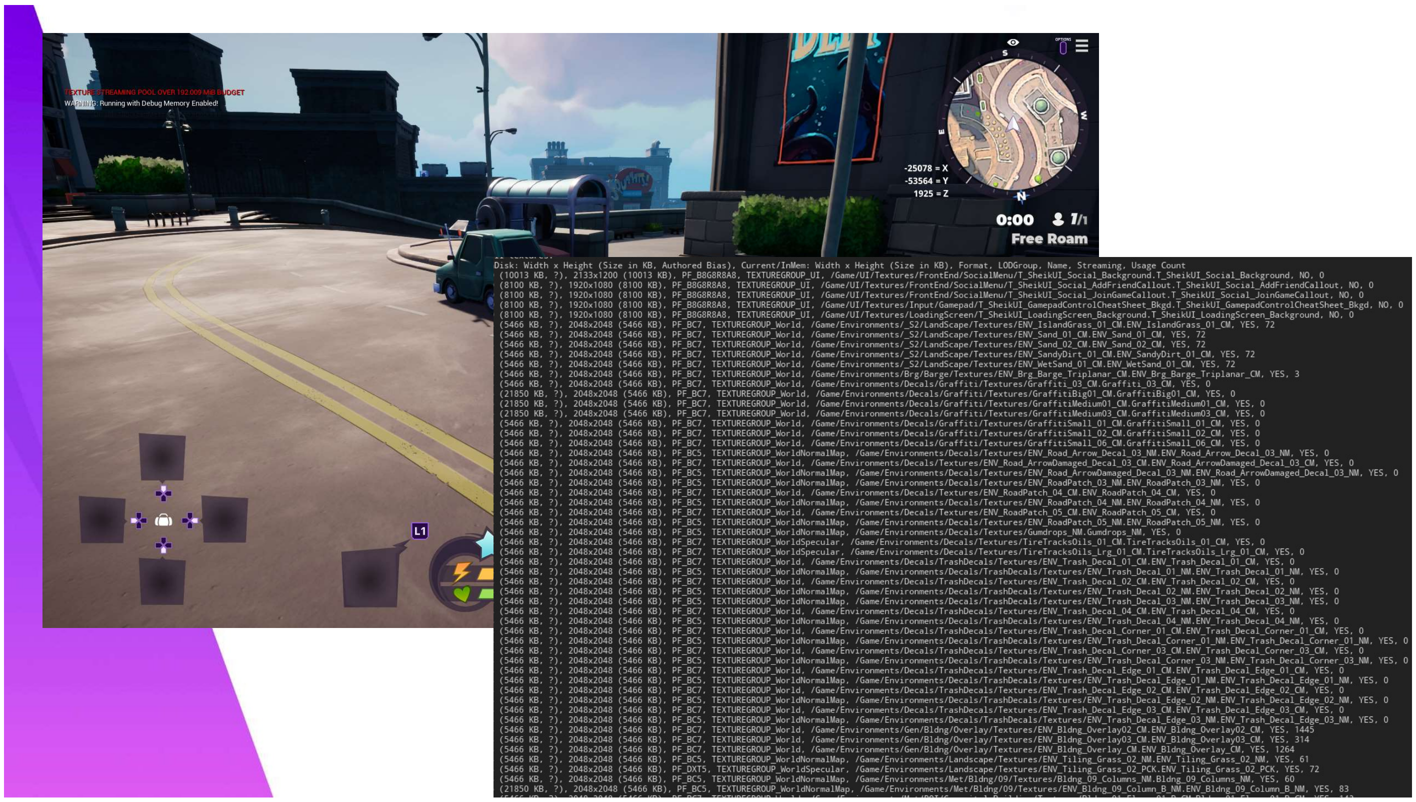






The artists and I had a bit of contention in that they (rightly) felt the environment was too stiff in our original greenlight and vertical slice builds. They wanted more variety and character, which leaning heavily on instancing was not great at supporting.

I pushed for them to try to use decals more and more, and they finally did. You can see that the graffiti on the left side of this slide is overlapping multiple instanced wall meshes that are underneath it. And the cracks on the street in the lower right are breaking up the repetition in a road mesh that is generated from a spline.

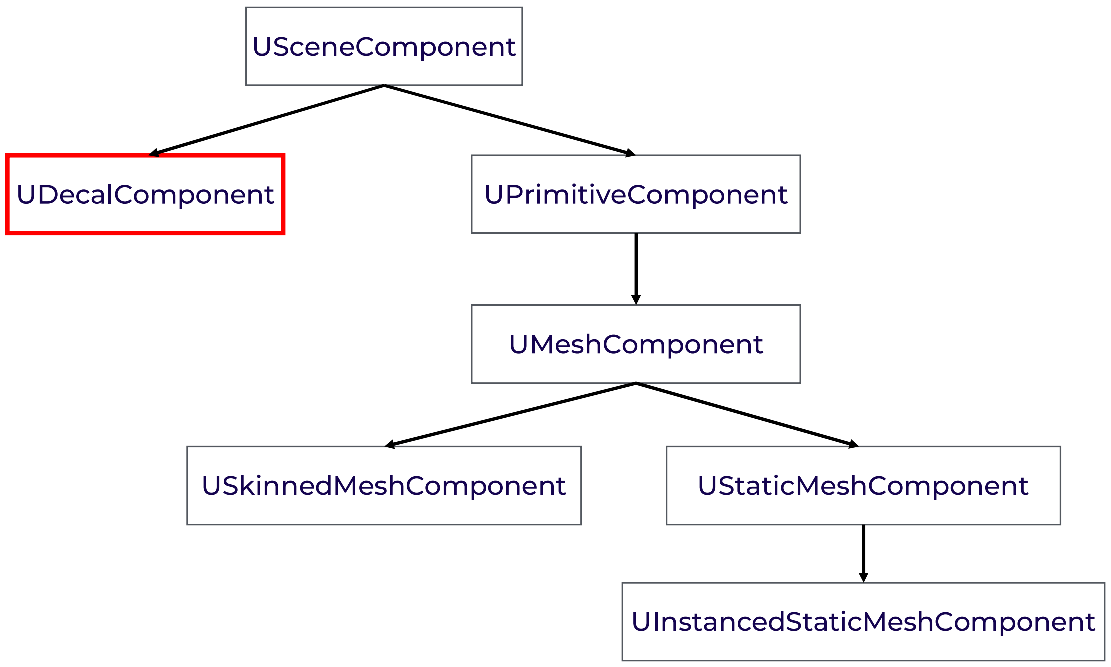


```
TEXTURE STREAMING POOL OVER 192.009 MiB BUDGET
WARNING: Running with Debug Memory Enabled!
```

```
Disk: Width x Height (Size in KB, Authored Bias), Current/InMem: Width x Height (Size in KB), Format, LODGroup, Name, Streaming, Usage Count
 (10013 KB, ?), 2133x1200 (10013 KB), PF_B8G8R8A8, TEXTUREGROUP_UI, /Game/UI/Textures/FrontEnd/SocialMenu/T_SheikUI_Social_Background.T_SheikUI_Social_Background, NO, 0
 (8100 KB, ?), 1920x1080 (8100 KB), PF_B8G8R8A8, TEXTUREGROUP_UI, /Game/UI/Textures/FrontEnd/SocialMenu/T_SheikUI_Social_AddFriendCallout.T_SheikUI_Social_AddFriendCallout, NO, 0
 (8100 KB, ?), 1920x1080 (8100 KB), PF_B8G8R8A8, TEXTUREGROUP_UI, /Game/UI/Textures/FrontEnd/SocialMenu/T_SheikUI_Social_JoinGameCallout.T_SheikUI_Social_JoinGameCallout, NO, 0
 (8100 KB, ?), 1920x1080 (8100 KB), PF_B8G8R8A8, TEXTUREGROUP_UI, /Game/UI/Textures/Input/Gamepad/T_SheikUI_GamepadControlCheatSheet_Bkgd.T_SheikUI_GamepadControlCheatSheet_Bkgd, NO, 0
 (8100 KB, ?), 1920x1080 (8100 KB), PF_B8G8R8A8, TEXTUREGROUP_UI, /Game/UI/Textures/LoadingScreen/T_SheikUI_LoadingScreen_Background.T_SheikUI_LoadingScreen_Background, NO, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/_S2/LandScape/Textures/ENV_IslandGrass_01_CM.ENV_IslandGrass_01_CM, YES, 72
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/_S2/LandScape/Textures/ENV_Sand_01_CM.ENV_Sand_01_CM, YES, 72
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/_S2/LandScape/Textures/ENV_Sand_02_CM.ENV_Sand_02_CM, YES, 72
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/_S2/LandScape/Textures/ENV_SandyDirt_01_CM.ENV_SandyDirt_01_CM, YES, 72
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/_S2/LandScape/Textures/ENV_WetSand_01_CM.ENV_WetSand_01_CM, YES, 72
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Brg/Barge/Textures/ENV_Brg_Barge_Triplanar_CM.ENV_Brg_Barge_Triplanar_CM, YES, 3
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/Graffiti_03_CM.Graffiti_03_CM, YES, 0
 (21850 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/GraffitiBig01_CM.GraffitiBig01_CM, YES, 0
 (21850 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/GraffitiMedium01_CM.GraffitiMedium01_CM, YES, 0
 (21850 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/GraffitiMedium03_CM.GraffitiMedium03_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/GraffitiSmall_01_CM.GraffitiSmall_01_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/GraffitiSmall_02_CM.GraffitiSmall_02_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Graffiti/Textures/GraffitiSmall_06_CM.GraffitiSmall_06_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/Textures/ENV_Road_Arrow_Decal_03_NM.ENV_Road_Arrow_Decal_03_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Textures/ENV_Road_ArrowDamaged_Decal_03_CM.ENV_Road_ArrowDamaged_Decal_03_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/Textures/ENV_Road_ArrowDamaged_Decal_03_NM.ENV_Road_ArrowDamaged_Decal_03_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/Textures/ENV_RoadPatch_03_NM.ENV_RoadPatch_03_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Textures/ENV_RoadPatch_04_CM.ENV_RoadPatch_04_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/Textures/ENV_RoadPatch_04_NM.ENV_RoadPatch_04_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/Textures/ENV_RoadPatch_05_CM.ENV_RoadPatch_05_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/Textures/ENV_RoadPatch_05_NM.ENV_RoadPatch_05_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/Textures/Gumdrops_NM.Gumdrops_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_WorldSpecular, /Game/Environments/Decals/Textures/TireTracksOils_01_CM.TireTracksOils_01_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_WorldSpecular, /Game/Environments/Decals/Textures/TireTracksOils_Lrg_01_CM.TireTracksOils_Lrg_01_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_01_CM.ENV_Trash_Decal_01_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_01_NM.ENV_Trash_Decal_01_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_02_CM.ENV_Trash_Decal_02_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_02_NM.ENV_Trash_Decal_02_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_03_NM.ENV_Trash_Decal_03_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_04_CM.ENV_Trash_Decal_04_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_04_NM.ENV_Trash_Decal_04_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Corner_01_CM.ENV_Trash_Decal_Corner_01_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Corner_01_NM.ENV_Trash_Decal_Corner_01_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Corner_03_CM.ENV_Trash_Decal_Corner_03_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Corner_03_NM.ENV_Trash_Decal_Corner_03_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Edge_01_CM.ENV_Trash_Decal_Edge_01_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Edge_01_NM.ENV_Trash_Decal_Edge_01_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Edge_02_CM.ENV_Trash_Decal_Edge_02_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Edge_02_NM.ENV_Trash_Decal_Edge_02_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Edge_03_CM.ENV_Trash_Decal_Edge_03_CM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Decals/TrashDecals/Textures/ENV_Trash_Decal_Edge_03_NM.ENV_Trash_Decal_Edge_03_NM, YES, 0
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Gen/Bldng/Overlay/Textures/ENV_Bldng_Overlay02_CM.ENV_Bldng_Overlay02_CM, YES, 1445
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Gen/Bldng/Overlay/Textures/ENV_Bldng_Overlay03_CM.ENV_Bldng_Overlay03_CM, YES, 314
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC7, TEXTUREGROUP_World, /Game/Environments/Gen/Bldng/Overlay/Textures/ENV_Bldng_Overlay_CM.ENV_Bldng_Overlay_CM, YES, 1264
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Landscape/Textures/ENV_Tiling_Grass_02_NM.ENV_Tiling_Grass_02_NM, YES, 61
 (5466 KB, ?), 2048x2048 (5466 KB), PF_DXT5, TEXTUREGROUP_WorldSpecular, /Game/Environments/Landscape/Textures/ENV_Tiling_Grass_02_PCK.ENV_Tiling_Grass_02_PCK, YES, 72
 (5466 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Met/Bldng/09/Textures/Bldng_09_Columns_NM.Bldng_09_Columns_NM, YES, 60
 (21850 KB, ?), 2048x2048 (5466 KB), PF_BC5, TEXTUREGROUP_WorldNormalMap, /Game/Environments/Met/Bldng/09/Textures/ENV_Bldng_09_Column_B_NM.ENV_Bldng_09_Column_B_NM, YES, 83
```

So then once this happened, the first crisis was with environment decals sucking up space in the texture pool. Unreal has a texture streaming system to handle streaming the hi-res mips separately from the regular mesh+material streaming. It turns out that decal components always request the highest mip possible for textures. Which resulted in tons of detailed graffiti trash, and road markings gobbling up texture pool space. In a pinch, we could ask the artists to slash and burn the resolution these are allowed to use – but part of the benefits of the texture streamer is that the same resource can be used at different scales in the world and adjust it's resolution in the pool based on the particular shot.

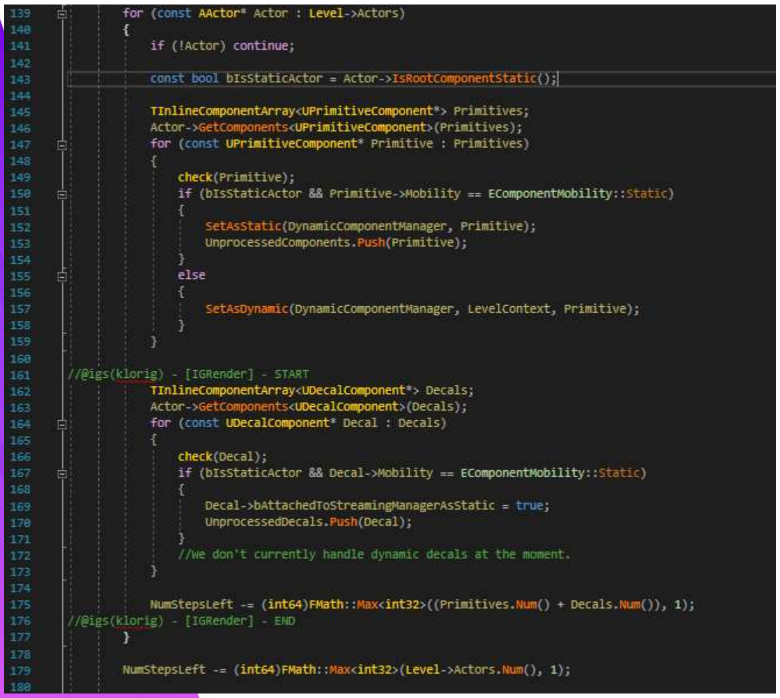
So what I did was I put Karinne on the case of figuring out why this was happening - my intuition was that this was going to just be a case of needing to write some logic to estimate the size of the decal on screen and send the requested mip count to the texture streaming system. She tracked it down to a much more fundamental decision in the engine tied to inheritance in C++.



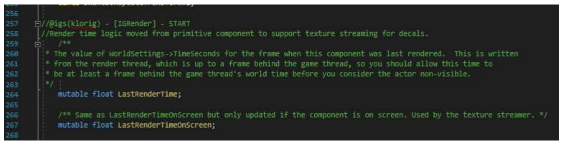
I’m illustrating a partial chain of inheritance from scene component here. A scene component in unreal is just something that has a transform matrix that is placed in a level, while a primitive is a scene component that goes through the renderer.

The problem is that at some point in the history of the engine a decision was made that Decals should not be Primitives and handled sepeartely. And likely separately from that, the Texture streaming system was engineered to operate on Primitives.

While my gut reaction was frustration that Decals were not inheriting from UPrimitiveComponent and that that was an oversight on the engine team’s part – there *are* good reasons to not make Decals into Primitives. I already covered the efforts we went through to keep our primitive count lower with Instanced Static Mesh components. For example – decals never need to cast shadows so we don’t need to waste time considering them for shadow casting like static meshes and skeletal meshes.



```
    for (const AActor* Actor : Level->Actors)
    {
        if (!Actor) continue;

        const bool bIsStaticActor = Actor->IsRootComponentStatic();

        TInlineComponentArray<UPrimitiveComponent*> Primitives;
        Actor->GetComponents<UPrimitiveComponent>(Primitives);
        for (const UPrimitiveComponent* Primitive : Primitives)
        {
            check(Primitive);
            if (bIsStaticActor && Primitive->Mobility == EComponentMobility::Static)
            {
                SetAsStatic(DynamicComponentManager, Primitive);
                UnprocessedComponents.Push(Primitive);
            }
            else
            {
                SetAsDynamic(DynamicComponentManager, LevelContext, Primitive);
            }
        }

//@igs(klorig) - [IGRender] - START
        TInlineComponentArray<UDecalComponent*> Decals;
        Actor->GetComponents<UDecalComponent>(Decals);
        for (const UDecalComponent* Decal : Decals)
        {
            check(Decal);
            if (bIsStaticActor && Decal->Mobility == EComponentMobility::Static)
            {
                Decal->bAttachedToStreamingManagerAsStatic = true;
                UnprocessedDecals.Push(Decal);
            }
            //We don't currently handle dynamic decals at the moment.
        }

        NumStepsLeft -= (int64)FMath::Max<int32>((Primitives.Num() + Decals.Num()), 1);
//@igs(klorig) - [IGRender] - END
    }

    NumStepsLeft -= (int64)FMath::Max<int32>(Level->Actors.Num(), 1);
```

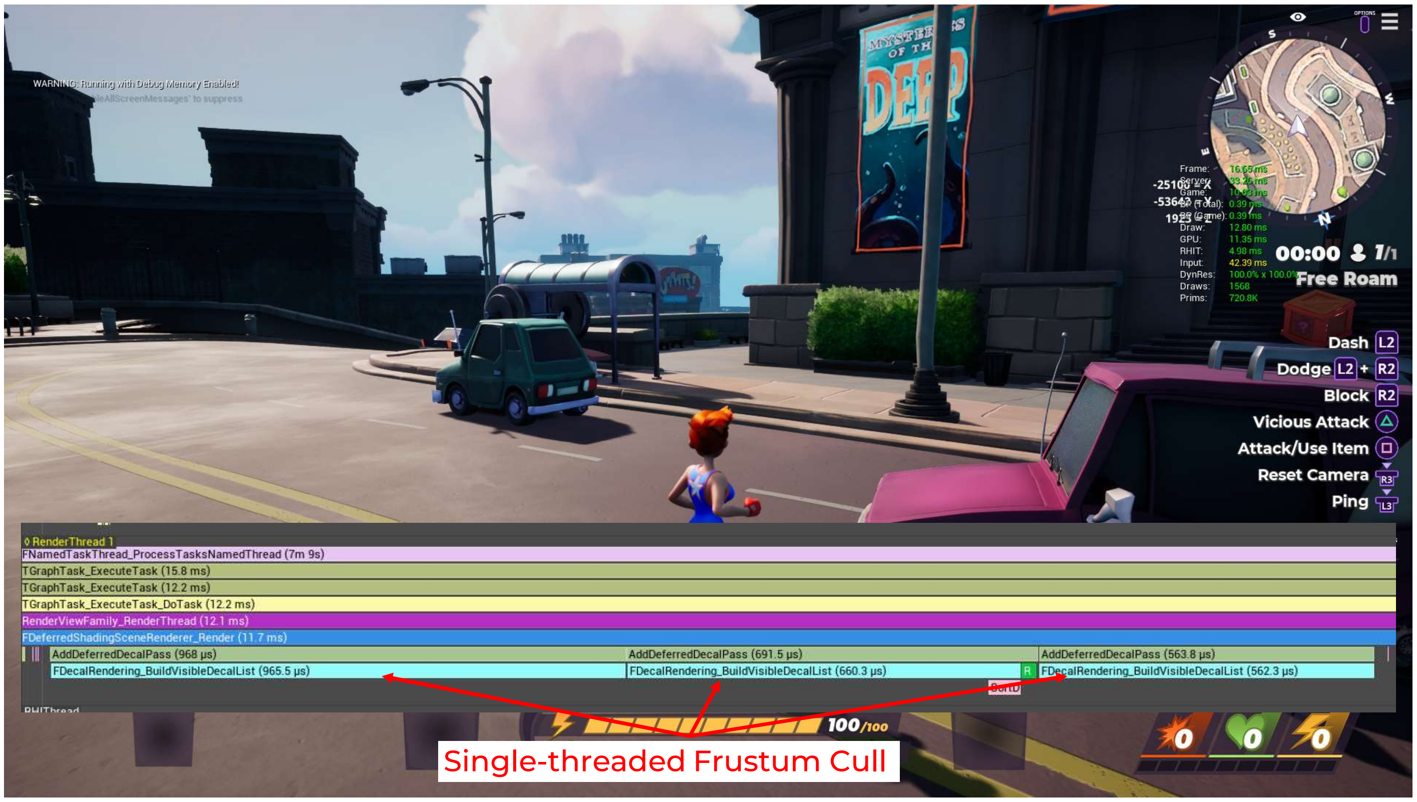


```
//@igs(klorig) - [IGRender] - START
//Render time logic moved from primitive component to support texture streaming for decals.
    /**
 * The value of WorldSettings->TimeSeconds for the frame when this component was last rendered.  This is written
 * from the render thread, which is up to a frame behind the game thread, so you should allow this time to
 * be at least a frame behind the game thread's world time before you consider the actor non-visible.
 */
    mutable float LastRenderTime;

    /** Same as LastRenderTimeOnScreen but only updated if the component is on screen. Used by the texture streamer. */
    mutable float LastRenderTimeOnScreen;
```

The fix we went with is to have the streaming manager maintain a list of UDecalComponents separately from the primitives, and promoted a small amount of data up to scene component from primitive component. This allowed all of those 2kx2k graffiti pieces to drop down as low as 64x64 when they are offscreen or far away.

I wanted to call out this particular code change because I do not think this is what I would do if I was the maintainer of the engine. For example, we could avoid making Decal Components inherit from Primitive Component by creating an interface class that each implements. The reason *not* to do this is because doing so would increase our merge burden – as Epic will be fully unaware of our change unless we discuss a pull request.

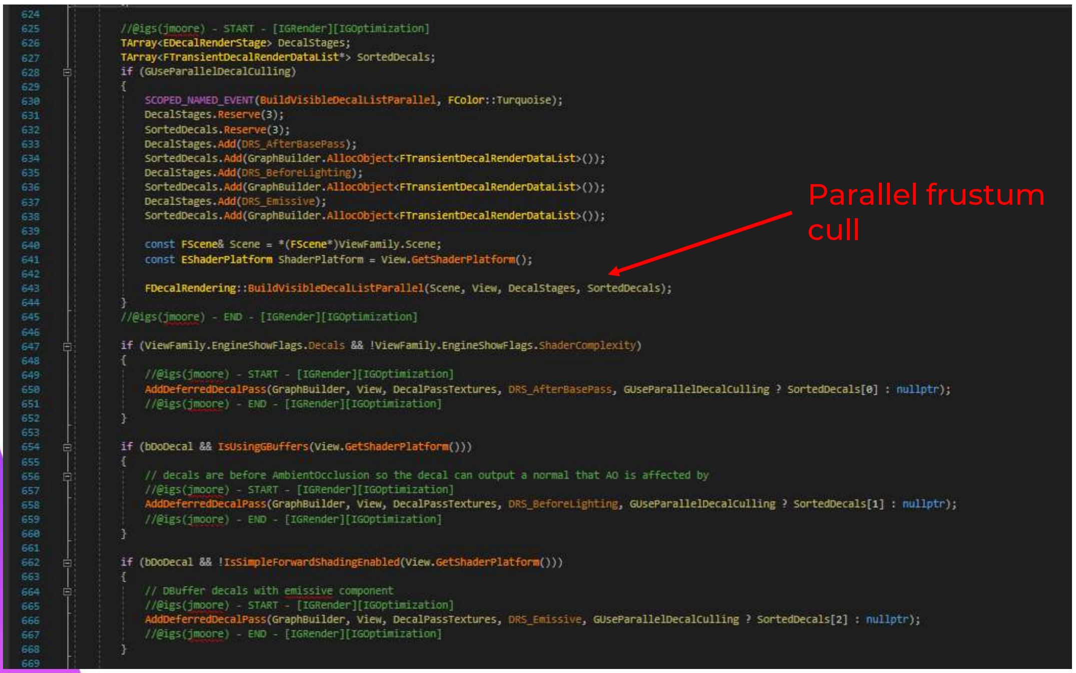
I have always found myself on game teams looking for the simplest to maintain code without sacrificing performance, and often waiting for a request up to an engine team is not feasible in a lot of contexts when working on a deadline.



```
◊ RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (7m 9s)
TGraphTask_ExecuteTask (15.8 ms)
TGraphTask_ExecuteTask (12.2 ms)
TGraphTask_ExecuteTask_DoTask (12.2 ms)
RenderViewFamily_RenderThread (12.1 ms)
FDeferredShadingSceneRenderer_Render (11.7 ms)
AddDeferredDecalPass (968 µs)                          AddDeferredDecalPass (691.5 µs)                      AddDeferredDecalPass (563.8 µs)
FDecalRendering_BuildVisibleDecalList (965.5 µs)       FDecalRendering_BuildVisibleDecalList (660.3 µs)     FDecalRendering_BuildVisibleDecalList (562.3 µs)
RHIThread
```

Now, If I had been thinking through all the consequences, I would have realized that everything I just said about Decals not being Primitives in the renderer meant that they also means that they likely don't get nearly as much attention regarding frustum culling performance on the render thread – and a little while later I started noticing that decal rendering time on the render thread was suddenly quite high as artists added more and more decals to our game. 2.2 ms in the location I showed with many decals in the surrounding area.

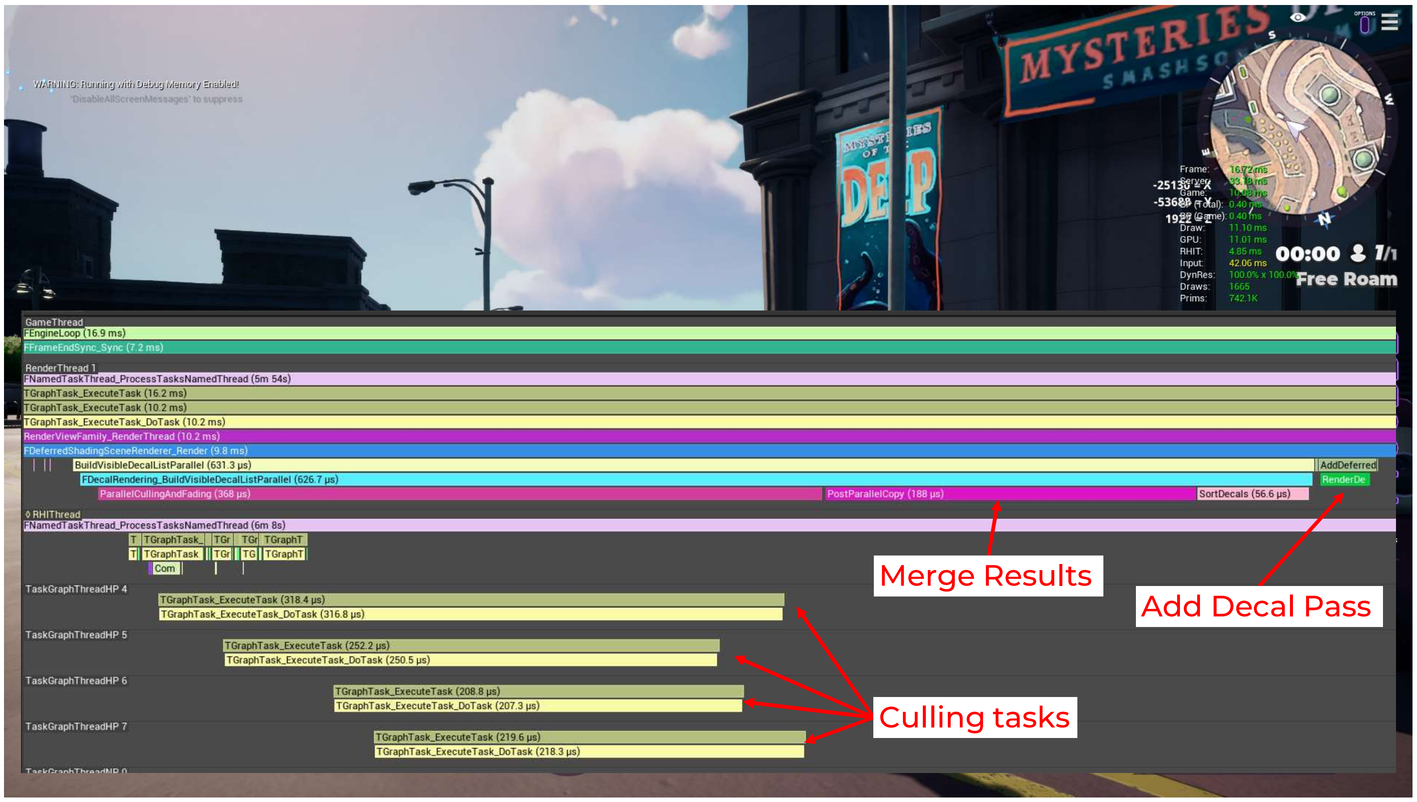
Each call to AddDeferredDecalPass is handling a different decal rendering stage is processing all of the decals in the scene separately and redoing frustum culling and distance fading – and it's happening serially on the main thread as well. Clearly, our artists were leaning more heavily into deferred decals than other games using the engine, which to be fair, is exactly what I had asked them to do.



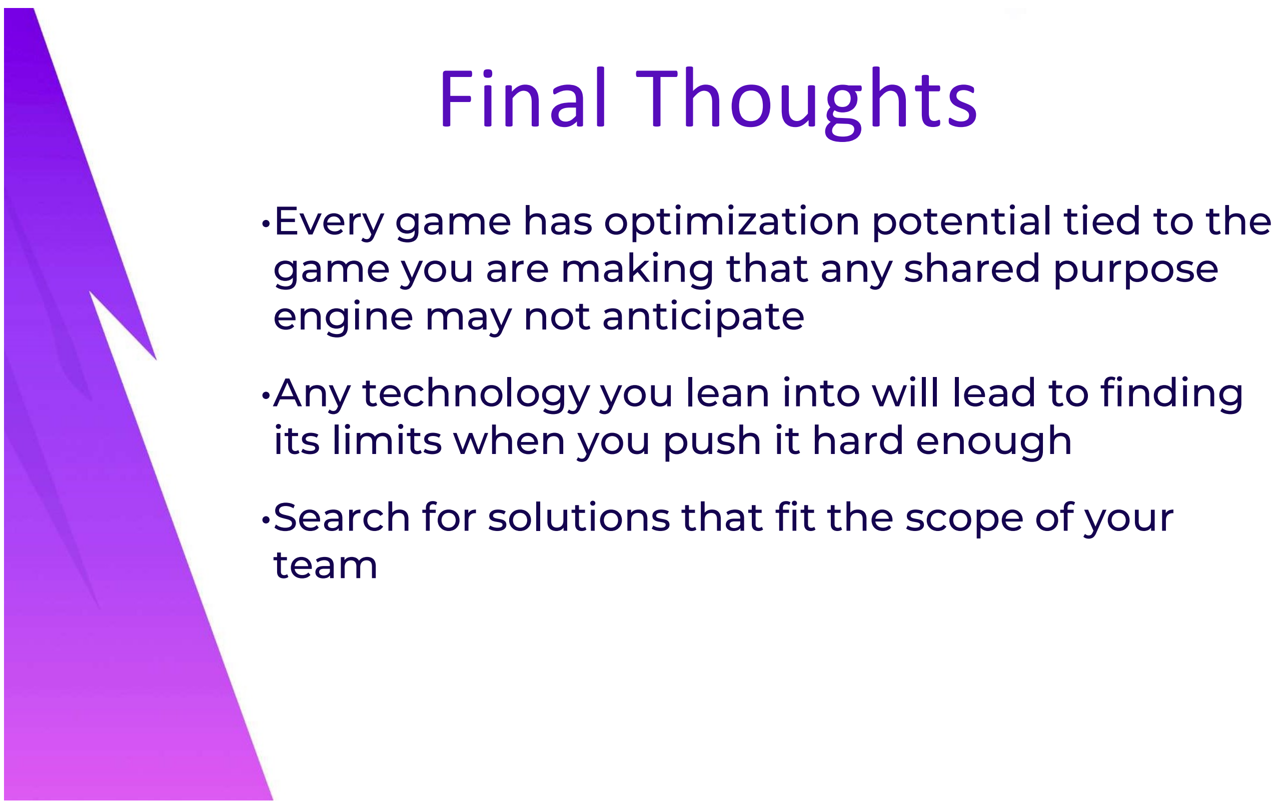
```
//@igs(jmoore) - START - [IGRender][IGOptimization]
TArray<EDecalRenderStage> DecalStages;
TArray<FTransientDecalRenderDataList*> SortedDecals;
if (GUseParallelDecalCulling)
{
    SCOPED_NAMED_EVENT(BuildVisibleDecalListParallel, FColor::Turquoise);
    DecalStages.Reserve(3);
    SortedDecals.Reserve(3);
    DecalStages.Add(DRS_AfterBasePass);
    SortedDecals.Add(GraphBuilder.AllocObject<FTransientDecalRenderDataList>());
    DecalStages.Add(DRS_BeforeLighting);
    SortedDecals.Add(GraphBuilder.AllocObject<FTransientDecalRenderDataList>());
    DecalStages.Add(DRS_Emissive);
    SortedDecals.Add(GraphBuilder.AllocObject<FTransientDecalRenderDataList>());

    const FScene& Scene = *(FScene*)ViewFamily.Scene;
    const EShaderPlatform ShaderPlatform = View.GetShaderPlatform();

    FDecalRendering::BuildVisibleDecalListParallel(Scene, View, DecalStages, SortedDecals);
}
//@igs(jmoore) - END - [IGRender][IGOptimization]

if (ViewFamily.EngineShowFlags.Decals && !ViewFamily.EngineShowFlags.ShaderComplexity)
{
    //@igs(jmoore) - START - [IGRender][IGOptimization]
    AddDeferredDecalPass(GraphBuilder, View, DecalPassTextures, DRS_AfterBasePass, GUseParallelDecalCulling ? SortedDecals[0] : nullptr);
    //@igs(jmoore) - END - [IGRender][IGOptimization]
}

if (bDoDecal && IsUsingGBuffers(View.GetShaderPlatform()))
{
    // decals are before AmbientOcclusion so the decal can output a normal that AO is affected by
    //@igs(jmoore) - START - [IGRender][IGOptimization]
    AddDeferredDecalPass(GraphBuilder, View, DecalPassTextures, DRS_BeforeLighting, GUseParallelDecalCulling ? SortedDecals[1] : nullptr);
    //@igs(jmoore) - END - [IGRender][IGOptimization]
}

if (bDoDecal && !IsSimpleForwardShadingEnabled(View.GetShaderPlatform()))
{
    // DBuffer decals with emissive component
    //@igs(jmoore) - START - [IGRender][IGOptimization]
    AddDeferredDecalPass(GraphBuilder, View, DecalPassTextures, DRS_Emissive, GUseParallelDecalCulling ? SortedDecals[2] : nullptr);
    //@igs(jmoore) - END - [IGRender][IGOptimization]
}
```

So the solution here is pretty simple – I build the three decal lists with a single frustum cull and distance fade on the task graph with a parallel For loop (very similar to primitive frustum culling in InitViews) and then I handle copying each decal that passes culling into the relevant list afterwards.

The resulting list is passed down into the different decal passes and the original per-pass processing is skipped.


WARNING: Running with Debug Memory Enabled!
'DisableAllScreenMessages' to suppress
GameThread
FEngineLoop (16.9 ms)
FFrameEndSync_Sync (7.2 ms)
RenderThread 1
FNamedTaskThread_ProcessTasksNamedThread (5m 54s)
TGraphTask_ExecuteTask (16.2 ms)
TGraphTask_ExecuteTask (10.2 ms)
TGraphTask_ExecuteTask_DoTask (10.2 ms)
RenderViewFamily_RenderThread (10.2 ms)
FDeferredShadingSceneRenderer_Render (9.8 ms)
BuildVisibleDecalListParallel (631.3 µs)
AddDeferred
FDecalRendering_BuildVisibleDecalListParallel (626.7 µs)
RenderDe
ParallelCullingAndFading (368 µs)
PostParallelCopy (188 µs)
SortDecals (56.6 µs)
RHIThread
FNamedTaskThread_ProcessTasksNamedThread (6m 8s)
TaskGraphThreadHP 4
TGraphTask_ExecuteTask (318.4 µs)
TGraphTask_ExecuteTask_DoTask (316.8 µs)
TaskGraphThreadHP 5
TGraphTask_ExecuteTask (252.2 µs)
TGraphTask_ExecuteTask_DoTask (250.5 µs)
TaskGraphThreadHP 6
TGraphTask_ExecuteTask (208.8 µs)
TGraphTask_ExecuteTask_DoTask (207.3 µs)
TaskGraphThreadHP 7
TGraphTask_ExecuteTask (219.6 µs)
TGraphTask_ExecuteTask_DoTask (218.3 µs)
Merge Results
Add Decal Pass
Culling tasks
00:00
Free Roam

In the area I showed this reduced processing down to 0.67 ms – a 1.53 ms improvement in render thread time.



Final Thoughts

- Every game has optimization potential tied to the game you are making that any shared purpose engine may not anticipate
- Any technology you lean into will lead to finding its limits when you push it hard enough
- Search for solutions that fit the scope of your team

Alright so just a few thoughts to end on. I just want to reiterate that I believe you should always be thinking about how the engineers that you are downstream from are not necessarily going to anticipate the content you are making.

And related to that, any time you push into new territory and scope as a team – you are likely going to hit the limits of systems. A lot of what I showed improvements for today worked really well for a long time in development, but eventually became untenable when the content reached a critical mass.

And related to that – always search for the solutions that fit the scope of your team. Part of why I spend so much time figuring out engineering solution is because we don't have an army of artists to try to rework content causing performance problems. And I very much appreciate when they are able to help us solve problems on the content side – we're all working together to try to make a great game in the end.

References

- Deferred Lighting in *Uncharted 4*, Ramy El Garawany, Siggraph 2016
- GPU-Driven Rendering Pipelines, Ulrich Haar, Sebastian Aaltonen, Siggraph 2015
- Separable Subsurface Scattering and Photorealistic Eyes Rendering, Jorge Jimenez, Siggraph 2016
- The Devil is in the Details: idTech666, Tiago Sousa, Jean Geffroy, Siggraph 2016
- Dynamic Occlusion With Signed Distance Fields, Daniel Wright, Siggraph 2015

Many Thanks

- Andreas Frederickson – the advisor for this talk
- Everyone that contributed to Rumbleverse rendering: especially Karinne Lorig, Nate Mefford, Rusty Swain, David Laskey, and our partners at Dragon’s Lake
- The entire Rumbleverse development team – especially our wonderful artists and the team leadership
- My little family: Kelsey, Spaceman, and Nova

Q&A

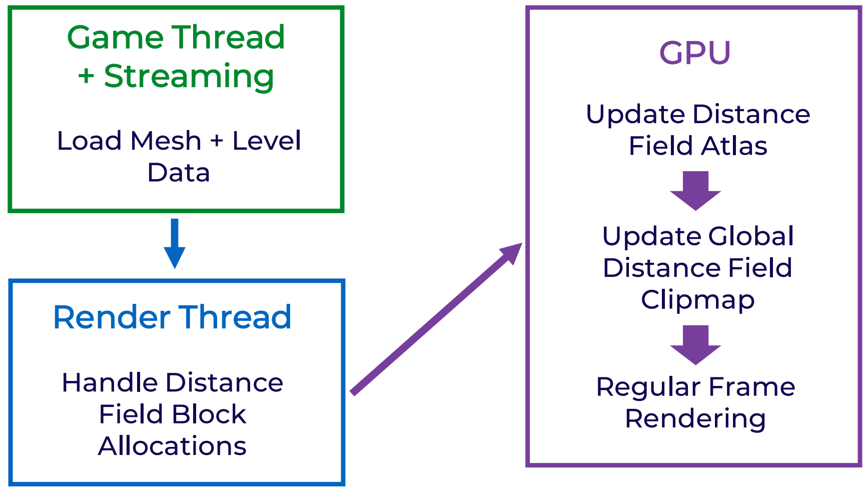
E-mail: Jon@IronGalaxyStudios.com
Twitter: @JonManatee

RUMBLEVERSE

EPIC GAMES IRON GALAXY

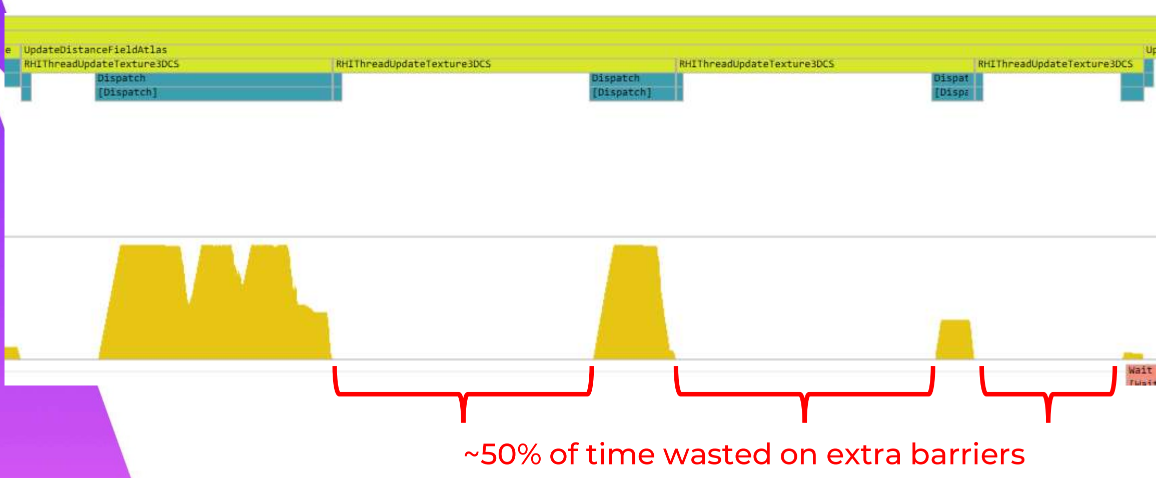
Appendix 1:
Distance Field
GPU Optimizations

Distance Field Upload


Game Thread
+ Streaming
Load Mesh + Level Data
Render Thread
Handle Distance Field Block Allocations
GPU
Update Distance Field Atlas
Update Global Distance Field Clipmap
Regular Frame Rendering

Let’s go over and consider the work happening on the GPU. As a reminder we need to copy the distance field texels into the atlas texture at the locations decided by the allocator, and then we need to update dirty regions of the global distance field clipmaps

Distance Field Atlas Barriers

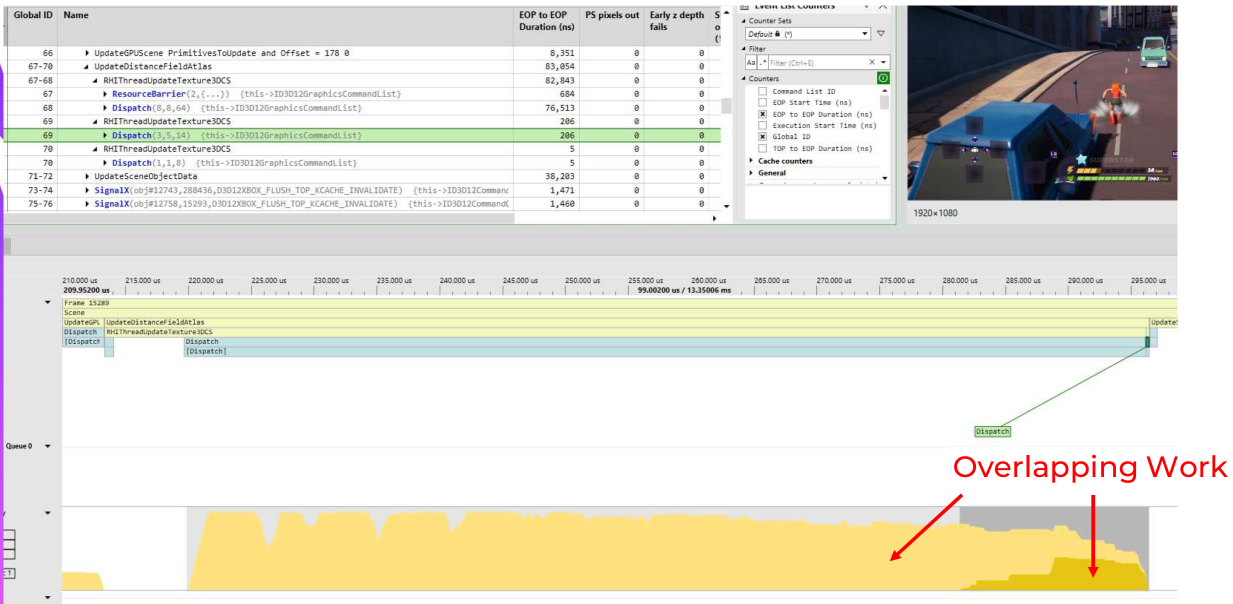

UpdateDistanceFieldAtlas
RHIThreadUpdateTexture3DCS
Dispatch
[Dispatch]
Wait
~50% of time wasted on extra barriers

The first up is uploads into the distance field atlas CS. The consoles are using a compute shader path to update these – and it is inserting a barrier between each copy. Again, this is especially bad because the artists are motivated to keep the distance field memory on each object as small as possible so they can pack more into the atlas. This means that the ratio between barrier waits and actual work on the CUs gets worse as the artists do a better and better job.

These barriers would be correct to ensure deterministic results if the regions being updated within the volume overlapped, but given that there is no need to do that as this is a texture atlas where everything gets its own spot in the buffer, it is safe for these wavefronts to be executing at the same time. None of these are going to be writing to the same places in memory.

There are four updates happening in this frame in this capture I'm showing here, but there could be dozens in some frames.

Distance Field Atlas Barriers

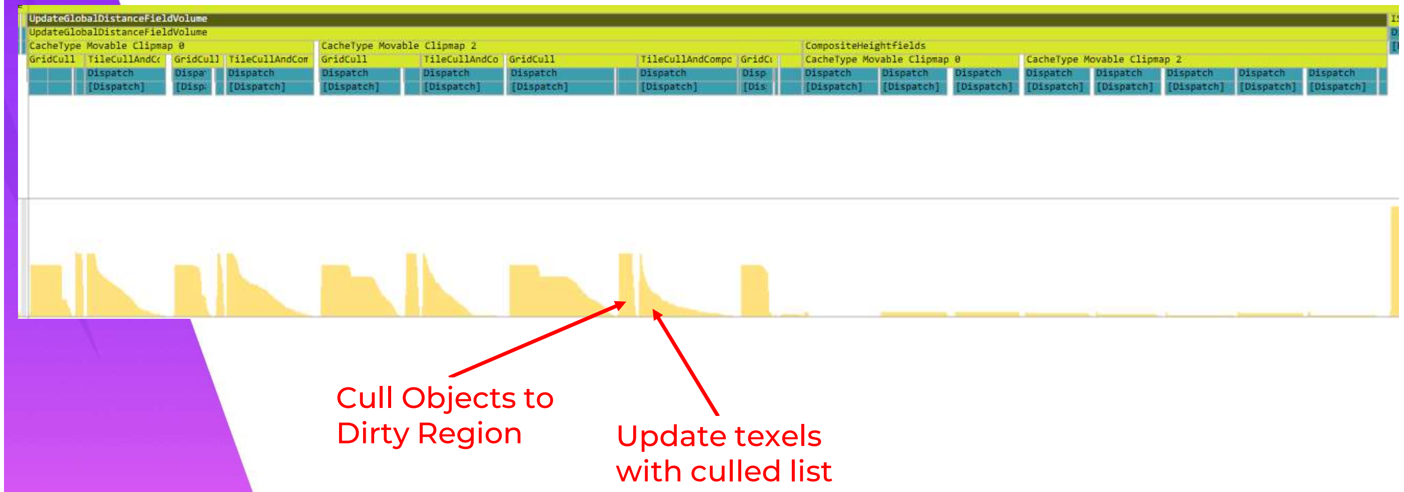


Global ID	Name	EOP to EOP Duration (ns)	PS pixels out	Early z depth fails
66	▸ UpdateGPUScene PrimitivesToUpdate and Offset = 178 0	8,351	0	0
67-70	◢ UpdateDistanceFieldAtlas	83,054	0	0
67-68	◢ RHIThreadUpdateTexture3DCS	82,843	0	0
67	▸ ResourceBarrier(2,{...}) {this->ID3D12GraphicsCommandList}	684	0	0
68	▸ Dispatch(8,8,64) {this->ID3D12GraphicsCommandList}	76,513	0	0
69	◢ RHIThreadUpdateTexture3DCS	206	0	0
69	▸ Dispatch(3,5,14) {this->ID3D12GraphicsCommandList}	206	0	0
70	◢ RHIThreadUpdateTexture3DCS	5	0	0
70	▸ Dispatch(1,1,8) {this->ID3D12GraphicsCommandList}	5	0	0
71-72	▸ UpdateSceneObjectData	38,203	0	0
73-74	▸ SignalX(obj#12743,288436,D3D12XBOX_FLUSH_TOP_KCACHE_INVALIDATE) {this->ID3D12Commanc	1,471	0	0
75-76	▸ SignalX(obj#12758,15293,D3D12XBOX_FLUSH_TOP_KCACHE_INVALIDATE) {this->ID3D12Command(	1,460	0	0

Artem from our co-dev partner Dragon’s Lake got these barriers eliminated by implementing an interface in the RHI layer that hints that there will be multiple safe updates to the same resource and only places a barrier at the beginning + end, and then we put in place a per-frame limit of 128^3 texels worth of updates per-frame to keep things from getting unmanageable from just the total amount of bandwidth needed. This keeps our time spent updating the atlas to under 0.1 ms on any given frame.

In this updated capture you can see that three distance fields are being updated and two of the barriers have been eliminated, keeping the GPU fed during these updates.

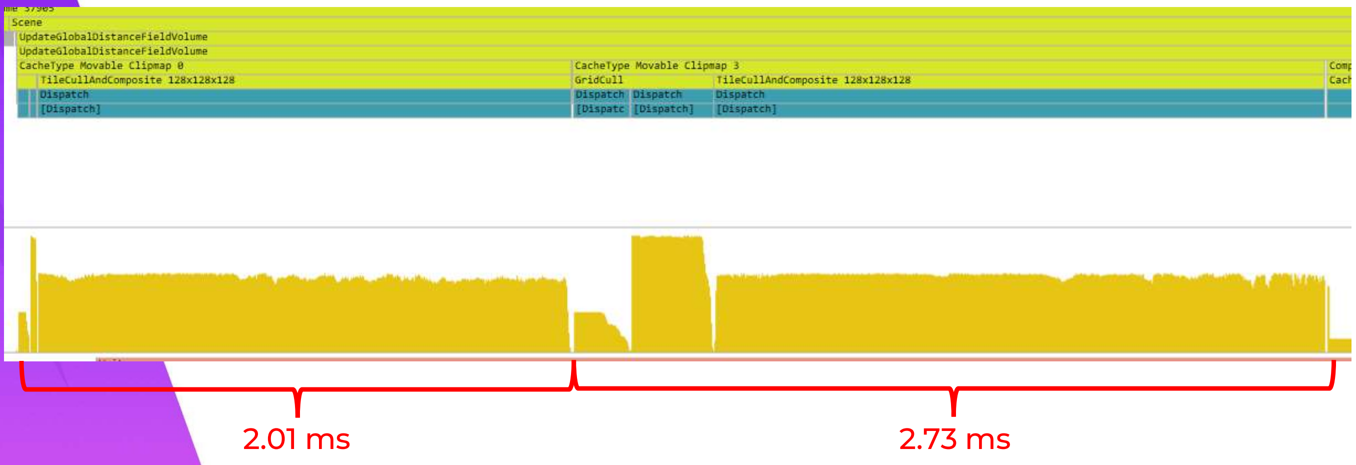
Global DF Update


UpdateGlobalDistanceFieldVolume
UpdateGlobalDistanceFieldVolume
CacheType Movable Clipmap 0
CacheType Movable Clipmap 2
CompositeHeightfields
CacheType Movable Clipmap 0
CacheType Movable Clipmap 2
GridCull
TileCullAndCo
Dispatch
[Dispatch]
Cull Objects to Dirty Region
Update texels with culled list

Analogous to the per-object distance field atlas, the global distance field clip maps are on staggered updates so that the closest clipmap updates more frequently than the largest. Heightfields for the terrain are composited after the regular DF's are updated. Getting gains here was less straightforward – the gridcull step is figuring out a reduced number of distance fields that intersect sub-regions of the area being updated. That memory for holding the culling results is re-used in each GridCull, eliminating the barriers would require allocating more memory.

That said, throttling the updates seemed like a really effective idea here, especially since this structure is only used by DFAO and not direct shadowing. We limit the number of partial update regions to 5 each frame. Regions get merged if they can form a volume of contiguous space (for example if one region fully contains another), and the largest regions are processed each frame from the pending list. I added a panic threshold to just update then entire clipmap space if there was ever 1000 pending regions piling up expecting to be processed.

Global DF Update - Full



Scene
UpdateGlobalDistanceFieldVolume
UpdateGlobalDistanceFieldVolume
CacheType Movable Clipmap 0
TileCullAndComposite 128x128x128
Dispatch
[Dispatch]
CacheType Movable Clipmap 3
GridCull
Dispatch
[Dispatc
Dispatch
[Dispatch]
TileCullAndComposite 128x128x128
Dispatch
[Dispatch]
2.01 ms
2.73 ms

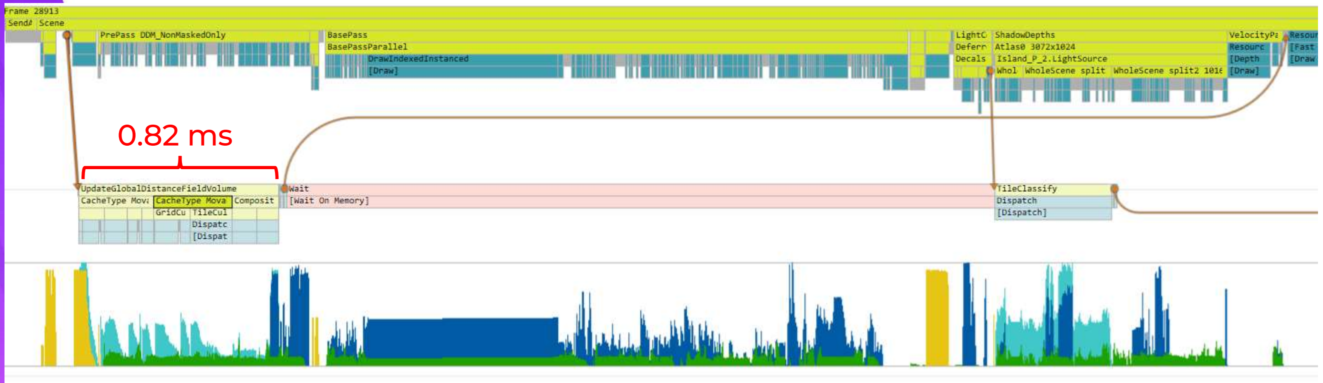
Surprisingly, we were hitting full clipmap updates, which can be quite slow – but not from my 1000 pending failsafe triggering.

It was happening from single large objects with distance fields that would invalidate the entire clip map at once. Throttling but update region count would never save us here and I was prepared to start implementing logic to slice up these regions into smaller sub sections over multiple frames, but I decided to look at what these objects were first.

It was coming from the fact that the artists liked to enable a distance field on LOD1 of many buildings levels so that the first LOD still casts a crude shadow, making the transition from the full LOD0 shadows less noticeable but still dropping down to 0 distance field memory at LOD2. These DFs that were the size of an entire buildings could easily trigger a full update of one of the clipmaps when running through the city. I'm showing that here – each of these clip maps takes >2 ms to update and we have 3 of them. We really want to avoid that case.

However- we really don't care about those LOD1 distance fields, and a really simple hack was put in place to eliminate that from happening. Which is that we simply don't process distance fields for the LOD1 meshes on the global clipmap. We know that the LOD0 meshes for that level are going to do a good job dirtying and updating the areas where buildings are placed anyways both on add and removal. The artists are generating the LOD1 distance field for the direct shadow tracing which is only traced against the distance field atlas and not the global clip map.

Global DF Update - Async

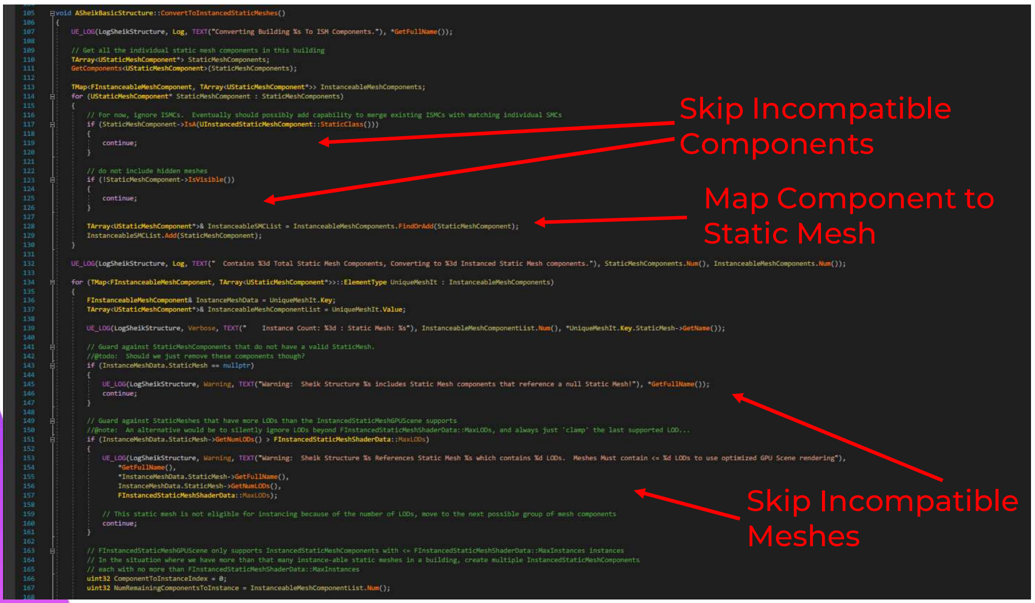


And for good measure we moved this work to async compute – we don't overlap postprocessing with the next frame with async compute like some games do. So our early frame work and depth prepass could use some async compute and this is a perfect candidate. A normal frame is going to have to scroll the clip maps as the player moves and that takes ~0.5 ms on Xbox One, which stretches to 0.8 ms when run async and stays nicely overlapped with the vertex shading work in our prepass. When additional work is needed this can spill past the prepass, but overlapping with the base pass is generally not overly problematic for us and we still see an overall reduction in spikes.

The results of the distance field updates are not needed until after our velocity rendering completes, but we do have additional async work kicking off to overlap with shadow maps. Throttling of the global clipmap updates generally prevents that from ever running that deep into the frame.

Appendix 2: UE4 Cook Conversion Static Mesh -> Instanced Static Mesh

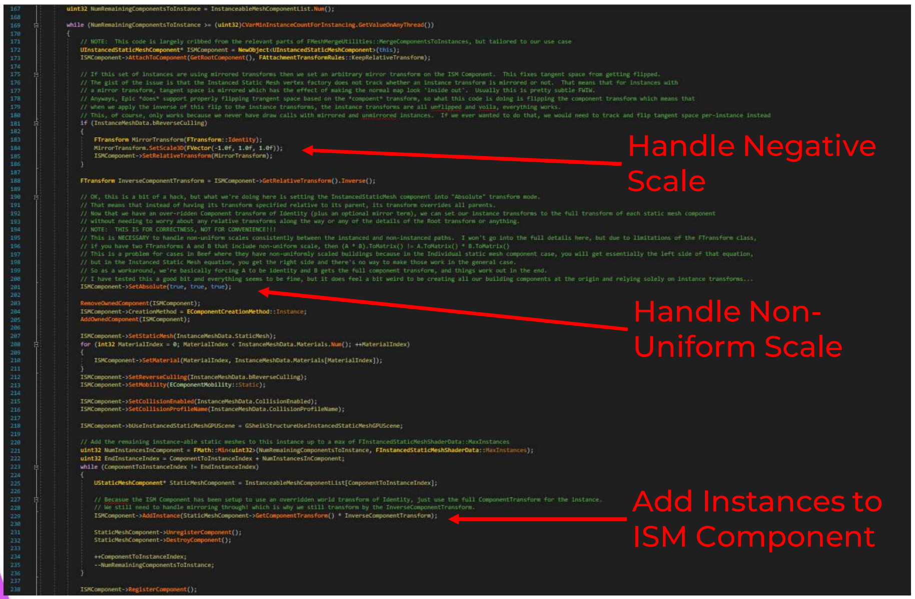
Cook-time Conversion



```
void ASheikBasicStructure::ConvertToInstancedStaticMeshes()
{
    UE_LOG(LogSheikStructure, Log, TEXT("Converting Building %s To ISM Components."), *GetFullName());

    // Get all the individual static mesh components in this building
    TArray<UStaticMeshComponent*> StaticMeshComponents;
    GetComponents<UStaticMeshComponent>(StaticMeshComponents);

    TMap<FInstanceableMeshComponent, TArray<UStaticMeshComponent*>> InstanceableMeshComponents;
    for (UStaticMeshComponent* StaticMeshComponent : StaticMeshComponents)
    {
        // For now, ignore ISMCs.  Eventually should possibly add capability to merge existing ISMCs with matching individual SMCs
        if (StaticMeshComponent->IsA(UInstancedStaticMeshComponent::StaticClass()))
        {
            continue;
        }

        // do not include hidden meshes
        if (!StaticMeshComponent->IsVisible())
        {
            continue;
        }

        TArray<UStaticMeshComponent*>& InstanceableSMCList = InstanceableMeshComponents.FindOrAdd(StaticMeshComponent);
        InstanceableSMCList.Add(StaticMeshComponent);
    }

    UE_LOG(LogSheikStructure, Log, TEXT("  Contains %3d Total Static Mesh Components, Converting to %3d Instanced Static Mesh components."), StaticMeshComponents.Num(), InstanceableMeshComponents.Num());

    for (TMap<FInstanceableMeshComponent, TArray<UStaticMeshComponent*>>::ElementType UniqueMeshIt : InstanceableMeshComponents)
    {
        FInstanceableMeshComponent& InstanceMeshData = UniqueMeshIt.Key;
        TArray<UStaticMeshComponent*>& InstanceableMeshComponentList = UniqueMeshIt.Value;

        UE_LOG(LogSheikStructure, Verbose, TEXT("    Instance Count: %3d : Static Mesh: %s"), InstanceableMeshComponentList.Num(), *UniqueMeshIt.Key.StaticMesh->GetName());

        // Guard against StaticMeshComponents that do not have a valid StaticMesh.
        //@todo:  Should we just remove these components though?
        if (InstanceMeshData.StaticMesh == nullptr)
        {
            UE_LOG(LogSheikStructure, Warning, TEXT("Warning:  Sheik Structure %s includes Static Mesh components that reference a null Static Mesh!"), *GetFullName());
            continue;
        }

        // Guard against StaticMeshes that have more LODs than the InstancedStaticMeshGPUScene supports
        //@note:  An alternative would be to silently ignore LODs beyond FInstancedStaticMeshShaderData::MaxLODs, and always just 'clamp' the last supported LOD...
        if (InstanceMeshData.StaticMesh->GetNumLODs() > FInstancedStaticMeshShaderData::MaxLODs)
        {
            UE_LOG(LogSheikStructure, Warning, TEXT("Warning:  Sheik Structure %s References Static Mesh %s which contains %d LODs.  Meshes Must contain <= %d LODs to use optimized GPU Scene rendering"),
                *GetFullName(),
                *InstanceMeshData.StaticMesh->GetFullName(),
                InstanceMeshData.StaticMesh->GetNumLODs(),
                FInstancedStaticMeshShaderData::MaxLODs);

            // This static mesh is not eligible for instancing because of the number of LODs, move to the next possible group of mesh components
            continue;
        }

        // FInstancedStaticMeshGPUScene only supports InstancedStaticMeshComponents with <= FInstancedStaticMeshShaderData::MaxInstances instances
        // In the situation where we have more than that many instance-able static meshes in a building, create multiple InstancedStaticMeshComponents
        // each with no more than FInstancedStaticMeshShaderData::MaxInstances
        uint32 ComponentToInstanceIndex = 0;
        uint32 NumRemainingComponentsToInstance = InstanceableMeshComponentList.Num();
```

Here Is the code in the basic structure class that converts the individual mesh components to ISM components. First we iterate all of the components and build a unique mapping to source meshes. We must filter out some incompatible meshes at this time as well. Then we iterate each unique mesh.

Cook-time Conversion



```cpp
uint32 NumRemainingComponentsToInstance = InstanceableMeshComponentList.Num();

while (NumRemainingComponentsToInstance >= (uint32)CVarMinInstanceCountForInstancing.GetValueOnAnyThread())
{
    // NOTE:  This code is largely cribbed from the relevant parts of FMeshMergeUtilities::MergeComponentsToInstances, but tailored to our use case
    UInstancedStaticMeshComponent* ISMComponent = NewObject<UInstancedStaticMeshComponent>(this);
    ISMComponent->AttachToComponent(GetRootComponent(), FAttachmentTransformRules::KeepRelativeTransform);

    // If this set of instances are using mirrored transforms then we set an arbitrary mirror transform on the ISM Component.  This fixes tangent space from getting flipped.
    // The gist of the issue is that the Instanced Static Mesh vertex factory does not track whether an instance transform is mirrored or not.  That means that for instances with
    // a mirror transform, tangent space is mirrored which has the effect of making the normal map look 'inside out'.  Usually this is pretty subtle FWIW.
    // Anyways, Epic *does* support properly flipping trangent space based on the *compoent* transform, so what this code is doing is flipping the component transform which means that
    // when we apply the inverse of this flip to the instance transforms, the instance transforms are all unflipped and voila, everything works.
    // This, of course, only works because we never have draw calls with mirrored and unmirrored instances.  If we ever wanted to do that, we would need to track and flip tangent space per-instance instead
    if (InstanceMeshData.bReverseCulling)
    {
        FTransform MirrorTransform(FTransform::Identity);
        MirrorTransform.SetScale3D(FVector(-1.0f, 1.0f, 1.0f));
        ISMComponent->SetRelativeTransform(MirrorTransform);
    }

    FTransform InverseComponentTransform = ISMComponent->GetRelativeTransform().Inverse();

    // OK, this is a bit of a hack, but what we're doing here is setting the InstancedStaticMesh component into "Absolute" transform mode.
    // That means that instead of having its transform specified relative to its parent, its transform overrides all parents.
    // Now that we have an over-ridden Component transform of Identity (plus an optional mirror term), we can set our instance transforms to the full transform of each static mesh component
    // without needing to worry about any relative transforms along the way or any of the details of the Root transform or anything.
    // NOTE:  THIS IS FOR CORRECTNESS, NOT FOR CONVENIENCE!!!
    // This is NECESSARY to handle non-uniform scales consistently between the instanced and non-instanced paths.  I won't go into the full details here, but due to limitations of the FTransform class,
    // if you have two FTransforms A and B that include non-uniform scale, then (A * B).ToMatrix() != A.ToMatrix() * B.ToMatrix()
    // This is a problem for cases in Beef where they have non-uniformly scaled buildings because in the Individual static mesh component case, you will get essentially the left side of that equation,
    // but in the Instanced Static Mesh equation, you get the right side and there's no way to make those work in the general case.
    // So as a workaround, we're basically forcing A to be identity and B gets the full component transform, and things work out in the end.
    // I have tested this a good bit and everything seems to be fine, but it does feel a bit weird to be creating all our building components at the origin and relying solely on instance transforms...
    ISMComponent->SetAbsolute(true, true, true);

    RemoveOwnedComponent(ISMComponent);
    ISMComponent->CreationMethod = EComponentCreationMethod::Instance;
    AddOwnedComponent(ISMComponent);

    ISMComponent->SetStaticMesh(InstanceMeshData.StaticMesh);
    for (int32 MaterialIndex = 0; MaterialIndex < InstanceMeshData.Materials.Num(); ++MaterialIndex)
    {
        ISMComponent->SetMaterial(MaterialIndex, InstanceMeshData.Materials[MaterialIndex]);
    }
    ISMComponent->SetReverseCulling(InstanceMeshData.bReverseCulling);
    ISMComponent->SetMobility(EComponentMobility::Static);

    ISMComponent->SetCollisionEnabled(InstanceMeshData.CollisionEnabled);
    ISMComponent->SetCollisionProfileName(InstanceMeshData.CollisionProfileName);

    ISMComponent->bUseInstancedStaticMeshGPUScene = GSheikStructureUseInstancedStaticMeshGPUScene;

    // Add the remaining instance-able static meshes to this instance up to a max of FInstancedStaticMeshShaderData::MaxInstances
    uint32 NumInstancesInComponent = FMath::Min<uint32>(NumRemainingComponentsToInstance, FInstancedStaticMeshShaderData::MaxInstances);
    uint32 EndInstanceIndex = ComponentToInstanceIndex + NumInstancesInComponent;
    while (ComponentToInstanceIndex != EndInstanceIndex)
    {
        UStaticMeshComponent* StaticMeshComponent = InstanceableMeshComponentList[ComponentToInstanceIndex];

        // Becasue the ISM Component has been setup to use an overridden world transform of Identity, just use the full ComponentTransform for the instance.
        // We still need to handle mirroring through! which is why we still transform by the InverseComponentTransform.
        ISMComponent->AddInstance(StaticMeshComponent->GetComponentTransform() * InverseComponentTransform);

        StaticMeshComponent->UnregisterComponent();
        StaticMeshComponent->DestroyComponent();

        ++ComponentToInstanceIndex;
        --NumRemainingComponentsToInstance;
    }

    ISMComponent->RegisterComponent();
```

Then we iterate over the instances for each mesh and build ISMs. Note that like many things in computer graphics, handling negative scale and non-uniform scale was a particular sore spot for getting everything to render correctly.

Then the instances are added to the new ISM component and the old component is destroyed, and finally the new component is registered after all the new instances are added. We do this in the class's Serialize function but some of my more recent work has made me believe it would be more correct to do this in PreSave.