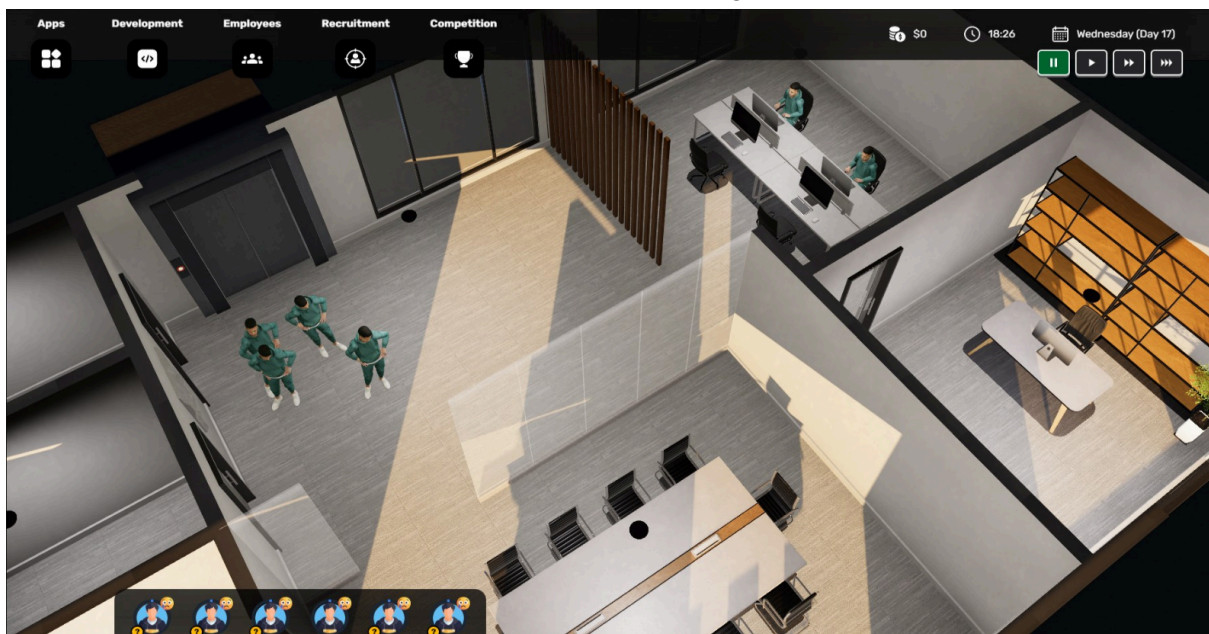


Case Study: Hovgaard Games

I was brought in as a contractor Tech Artist into Hovgaard Games to help with two things - they needed prettier lighting for their upcoming project Startup Company 2 and they needed it fast and cheap. Also, the performance in another of their games was poor and they asked me for an analysis and best practice overview.

Startup Company 2 - Improving the lighting on a dime

Upon opening up their project, I was stricken by the flat render, non-existing shadows and overall blandness of the scene as seen in the below image.



All objects except walls and floors were supposed to be dynamic as players could place them which ruled out baked GI. The client allotted me 10 hours for analysis and to take measures. After some consideration I employed the following techniques:

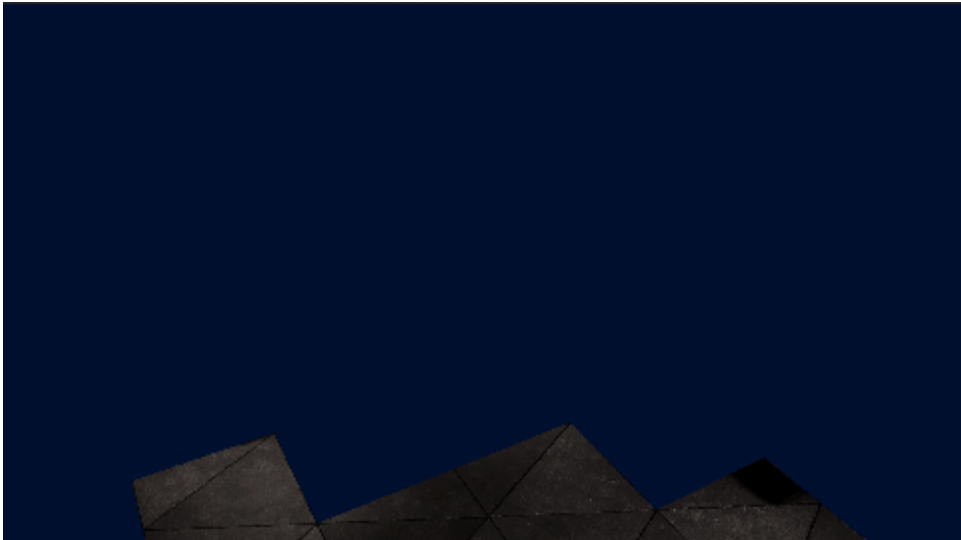
- Using APV's (Unity's Adaptive Probe Volumes), capturing the static scene.
- HTrace GI plugin
- HTrace AO plugin
- Dynamic Shadows



The shadows, real time GI and AO plugins improved the lighting tremendously, as seen in the above image, but expectedly caused a performance hit, frame times exceeding 20ms rendered with a 3070 @ 1440p.

▼ UniversalRenderPipeline.RenderSingleCameraInternal: Main Camera	99.0%	76	20.289
▼ ExecuteRenderGraph	99.0%	76	20.289
▶ HTraceSSGI SSGI Pass	32.3%	23	6.627
▶ DrawOpaqueObjects	30.0%	1	6.159
▶ HTraceAO GTAO Pass	19.5%	6	3.996
▶ HTraceSSGI GBuffer Pass	5.5%	3	1.135
▶ DrawDepthNormalPrepass	2.5%	1	0.517
▶ DrawTransparentObjects	2.4%	1	0.509
▶ RG_TAA	2.1%	1	0.439
▶ Draw Main Light Shadowmap	1.0%	3	0.219
▶ DrawMotionVectors	0.7%	1	0.153
Setup Camera Properties	0.6%	2	0.136
▶ Draw Additional Lights Shadowmap	0.5%	10	0.122
▶ RG_UberPost	0.4%	1	0.082
▶ RG_TAACopyHistory	0.3%	1	0.062
▶ BlitFinalToBackBuffer	0.2%	1	0.054
▶ DrawScreenSpaceUI	0.2%	3	0.045

The scene was not particularly poly-dense, but by analysing the frame it became obvious that there was a large amount of drawcalls going on due to non functional batching and many unique shaders.



Above: Example of non functional batching and overall opaque drawcall hell (we want the scene to be drawn in large chunks, not many discrete calls)

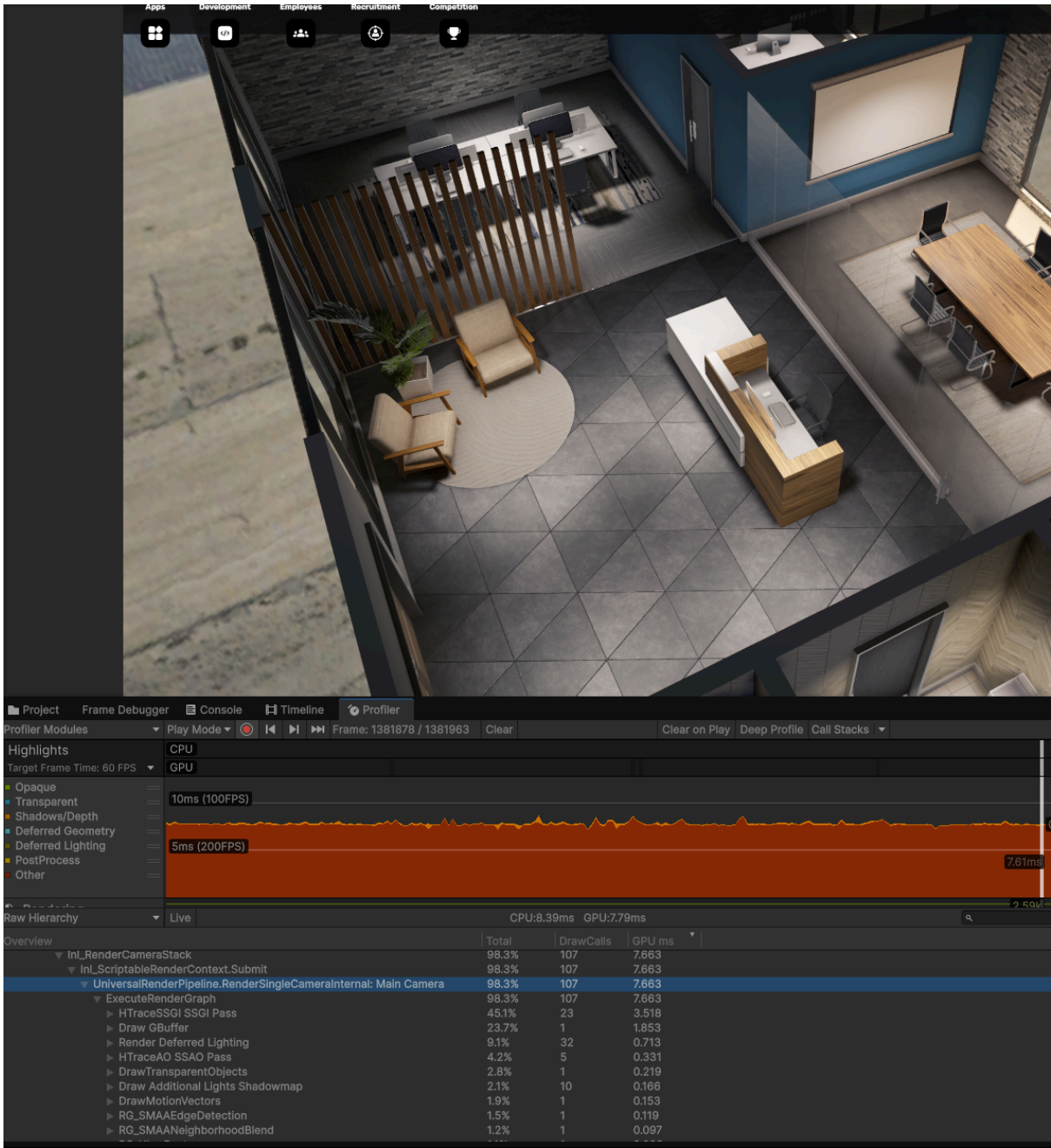
Measures taken here were:

- Merging several lit shaders into one by using texture atlases
- Switch render pipeline from Forward to Deferred+

Below: This resulted in the scene being rendered in large chunks due to getting batching to work by merging shaders.



After this it was smooth sailing, frame times went down to around 7.5 ms, resulting in a happy client :)



Below I have pasted a performance investigation document that was requested by the client due to the poor framerate.

Big Ambitions performance investigation

Below follows a short performance investigation mostly focused on GPU times. The game was rendered on a Ryzen 5950x, RTX 3070, 64GB ram system at 1440p.

Shadowcasting lights

At any given moment, the game renders at least a dozen shadow casting lights. The performance cost of shadow casting lights affect many rendering steps as well as CPU times, but most of it ends up in **RenderDeferredLightingCompute**.

Crunch Time Fitness has 17 shadowcasting lights



17 shadow casting lights in this instance produce a **4ms** cost on the GPU:

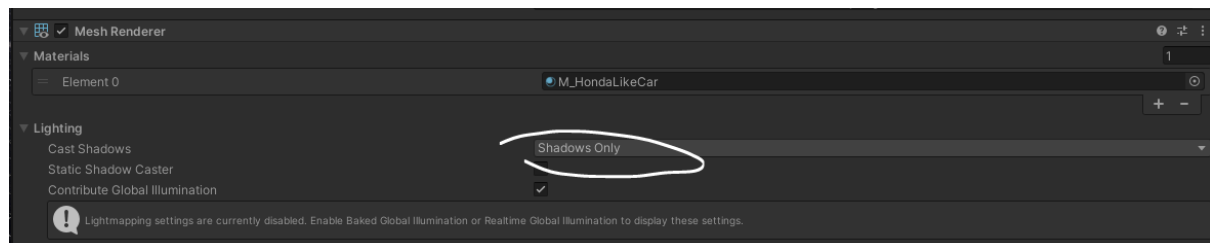
Overview	Total	DrawCalls	GPU ms
▼ PlayerLoop	89.6%	652	9.296
▼ RenderPipelineManager.DoRenderLoop_Internal() [Invoke]	89.6%	652	9.296
▼ InL_HDRenderPipelineAllRenderRequest	89.6%	652	9.296
▼ HDRenderPipeline::Render Main Camera	89.6%	652	9.296
▼ ExecuteRenderGraph	88.9%	650	9.224
▼ Deferred Lighting	39.0%	31	4.054
▶ RenderDeferredLightingCompute	39.0%	29	4.053
▶ RenderGraphClear	0.0%	2	0.001
▶ CustomPostProcessAfterOpaqueAndSky	10.9%	13	1.139
▶ RenderShadowMaps	8.0%	35	0.831
▶ PostProcessing	4.2%	22	0.439
▶ BuildLightList	3.9%	8	0.410
▶ GBuffer	3.8%	128	0.395

Reducing shadow casting lights to 1 in the gym gives around **1.4ms** cost, more than a 50% decrease in GPU times as well as reduced CPU times.

Overview	Total	DrawCalls	GPU ms
▼ PlayerLoop	85.7%	620	5.480
▼ RenderPipelineManager.DoRenderLoop_Internal() [Invoke]	85.7%	620	5.480
▼ Inl_HDRenderPipelineAllRenderRequest	85.7%	620	5.480
▼ HDRenderPipeline::Render Main Camera	85.7%	620	5.480
▼ ExecuteRenderGraph	84.7%	618	5.416
▼ Deferred Lighting	21.5%	31	1.378
▶ RenderDeferredLightingCompute	21.5%	29	1.377
▶ RenderGraphClear	0.0%	2	0.001
▶ CustomPostProcessAfterOpaqueAndSky	16.7%	13	1.068
▶ BuildLightList	10.2%	8	0.658
▶ PostProcessing	6.6%	22	0.425
▶ GBuffer	5.6%	128	0.364

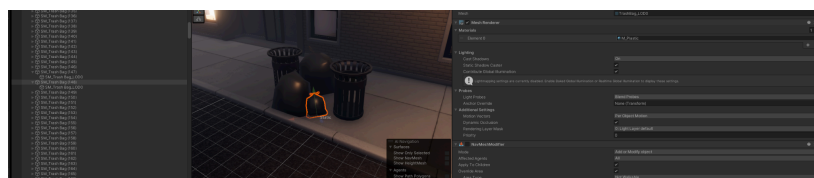
Consider cleaning up and reducing shadow casting lights where they matter the least, then move up to the ones that hurt to get rid of.. :) For indoor scenes, consider reducing to 1-2 shadow casting lights. If tolerable, remove shadowcasting on the MoonLight-directional light.

Consider using shadow proxies, i.e super low res silhouette meshes that are rendered with a fully transparent alpha-clip material using "Shadows Only" on the mesh renderers, then disable shadows for regular assets.

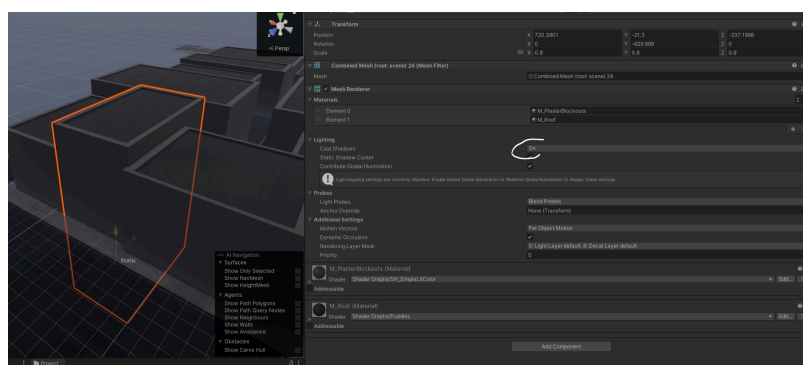


Additional small shadow-tips:

Consider removing shadows on small meshes, up to this trashbag size. AO should produce a sufficiently big dark halo for it to look believable.



Consider removing shadows on background buildings and merging the Material into 1.



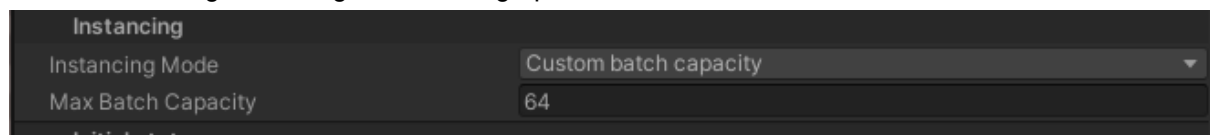
VFX Graph

The city contains 270 instances of a VisualEffect component “VFX_Steam” for ground/manhole cover smoke.

▼ PlayerLoop	35.6%	0.2%	1	1.5 KB	7.15	0.04
▶ Update.ScriptRunBehaviourUpdate	6.5%	0.0%	1	1.4 KB	1.32	0.00
▶ PreLateUpdate.ScriptRunBehaviourLateUpdate	4.4%	0.0%	1	70 B	0.88	0.00
▶ PreLateUpdate.DirectorUpdateAnimationEnd	3.3%	0.0%	1	0 B	0.66	0.00
▶ PreLateUpdate.AIUpdatePostScript	2.8%	0.0%	1	0 B	0.56	0.00
▶ PreLateUpdate.DirectorUpdateAnimationBegin	2.5%	0.0%	1	0 B	0.51	0.00
▶ FixedUpdate.PhysicsFixedUpdate	2.4%	0.0%	1	0 B	0.50	0.00
▶ PostLateUpdate.VFXUpdate	1.9%	0.0%	1	0 B	0.39	0.00
▶ PreUpdate.AIUpdate	1.6%	0.0%	1	0 B	0.33	0.00
▶ PostLateUpdate.UpdateAllRenderers	1.5%	0.0%	1	0 B	0.30	0.00
▶ FixedUpdate.ScriptRunBehaviourFixedUpdate	1.4%	0.0%	1	72 B	0.28	0.00
▶ PostLateUpdate.PlayerUpdateCanvases	1.3%	0.0%	1	0 B	0.27	0.00
EndUpdate.UpdateTextureStreamingMessages	1.1%	0.0%	1	0 B	0.22	0.00

Despite them not rendering particles as the camera is outside their rendering bounds, there is still a 0.4 ms CPU overhead when dealing with these instances.

Consider enabling instancing on the VFX graph asset on disk for a “free” reduction in cost.



Ideally VFX Graph instances should either be manually culled/deactivated when outside of the frustum. There are also techniques that allow 1 instance to deal with all particles, by distributing the spawn positions across all locations into 1 graph through position textures or buffers. There’s still plenty of headroom on the CPU currently, but it’s worth keeping in mind.

HBAO

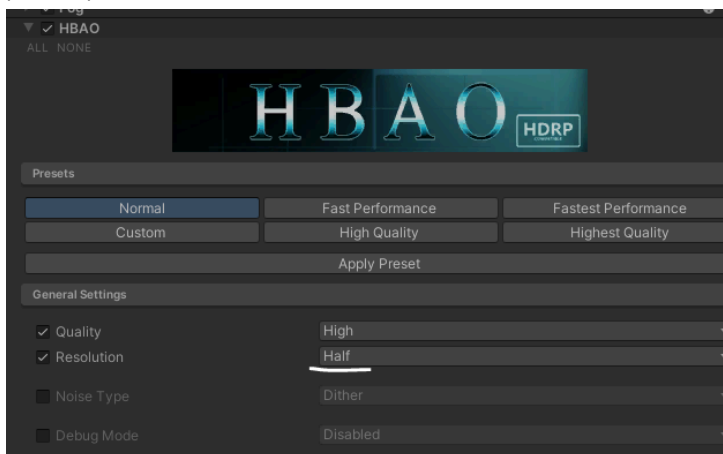
Currently the one of the most expensive post processing effects in the city scene is the HBAO giving around **1.2ms** cost. The profile was captured outdoors at night.

▼ PlayerLoop	93.5%	813	9.841
▼ RenderPipelineManager.DoRenderLoop_Internal() [Invoke]	93.5%	813	9.841
▼ InL_HDRenderPipelineAllRenderRequest	93.5%	813	9.841
▼ HDRenderPipeline::Render Main Camera	93.5%	813	9.841
▼ ExecuteRenderGraph	92.4%	811	9.723
▶ Deferred Lighting	21.5%	31	2.269
▼ CustomPostProcessAfterOpaqueAndSky	16.3%	13	1.723
▼ HBAO	15.4%	11	1.627
▶ HBAO - AO	11.7%	4	1.231
▶ HBAO - Blur	2.5%	4	0.096
▶ HBAO - Composite	1.2%	2	0.128
RenderTexture.SetActive	0.0%	0	0.000
▶ Blit Camera Texture	0.9%	2	0.096
▶ PostProcessing	11.8%	25	1.243

Reducing HBAO resolution to half cuts this cost significantly to **0.24ms**. To my eyes, there’s no discernible visual difference in doing this. The quality settings for AO do indeed tweak the settings of the HBAO post process, however the resolution seems untouched by the quality settings.

▼ PlayerLoop	82.5%	877	5.660
▼ RenderPipelineManager.DoRenderLoop_Internal() [Invoke]	82.5%	877	5.660
▼ InL_HDRenderPipelineAllRenderRequest	82.5%	877	5.660
▼ HDRenderPipeline:Render Main Camera	82.5%	877	5.660
▼ ExecuteRenderGraph	81.5%	875	5.591
▶ Deferred Lighting	21.1%	31	1.453
▶ PostProcessing	17.8%	25	1.226
▶ GBuffer	6.9%	121	0.476
▼ CustomPostProcessAfterOpaqueAndSky	5.7%	13	0.396
▼ HBAO	5.0%	11	0.343
▶ HBAO - AO	3.5%	4	0.240
▶ HBAO - Composite	0.8%	2	0.053
▶ HBAO - Blur	0.6%	4	0.045
RenderTexture.SetActive	0.0%	0	0.000
▶ Blit Camera Texture	0.7%	2	0.053
▶ BuildLightList	5.1%	8	0.351
▶ Custom Pass	3.3%	2	0.227
▶ UpdateEnvironment	3.2%	110	0.225
▶ ColorPyramid	2.3%	56	0.162
▶ RenderShadowMaps	2.3%	236	0.158
▶ DeferredDepthPrepass	2.2%	120	0.152

(Above) HBAO resolution reduced to half .



Draw calls and batching

The biggest performance benefit is to reduce draw calls. Buildings in the city are currently the biggest source of draw calls. Merging materials and shaders involves a lot of manual work and possibly retexturing and new UVs. Some aspects could probably be automated but would require some research time by yours truly.

DLSS

A fast way to gain back some GPU performance is through DLSS. It renders the game at a lower resolution and then upscales. DLSS will not reduce draw calls or help particularly much with shadow costs - it will however reduce costs of pixel-expensive shaders. I did not encounter a huge benefit with it, but lower end systems may benefit more.

<https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/deep-learning-super-sampling-in-hdrp.html>