



Gotta Go Fast

Graphics Optimization For Tech Artists

Garrett Stevens
Technical Artist - Funomena

GAME DEVELOPERS CONFERENCE
MARCH 18-22, 2019 | #GDC19

Who is This Guy

Garrett Stevens - Technical Artist, Prototyping, etc.



Currently at Funomena, recently shipped Luna: Moondust Garden for Magic Leap

Also part of a small 3 person dev team as Polymath

Previously at Apple doing things

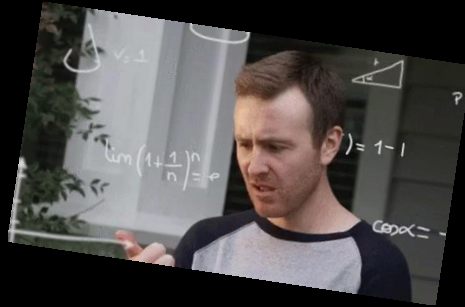
Before that on the prototyping and demos team at Meta AR

Before that spent a few years at intel Prototyping for experimental hardware

Spent a lot of time trying to get things to run faster on things that are underpowered, and overheated.

Not many people can say they've written a shader that made hardware actually explode.

Don't Panic



As a tech Artist, particularly as a new tech artist, optimization can sound pretty scary
There is a lot of information (and misinformation) out there at every level

- from outdated info
- hunches and superstition
- just plain crazy theories (Reddit, Forums)

Scope of this Talk

NOT about

- Low Level GPU optimization
- Lots of rendering math / counting every instruction
- A tutorial for one of 50 profilers

Is about

- An approachable intro to graphics optimization
- Making shaders and post effects run faster
- Becoming more aware of your various resources
- Basic Tips and Tricks

I'm not smart enough to teach anyone about deep shader optimization

- This is for entry level to intermediate Tech artists
- Who want an intro or a survey of optimization basic

Know Your Enemy

Understand the limitations of your target hardware

- PC
 - Ballpark your min spec
 - Will you have multiple quality settings?
 - Test on Nvidia, AMD, and Intel GPUs
- Console + Mobile
 - Specific hardware means specific debuggers
 - Emulation will only get you so far



PC

- If you're simultaneously releasing on multiple platforms, test on them early. Maybe have a dedicated toaster machine or a few crappy android tablets with various releases. You don't have to build to these all the time, but a sanity check might save your bacon.
- If you have dedicated time and people to port later, don't leave them a huge mess (relying heavily on unsupported features, weird unreadable over-optimization)
- Are you targeting toasters or aircraft carriers?
- What color is the gpu in your dev machine? Are you testing on red and blue ones too?

Console

- Mention Playstation, Xbox, Switch equivalents

Mobile

- Be constantly testing. Lots of things just behave differently or are unsupported on Android.



Jokingly exaggerate separation of the gpu vs cpu “ I write all my code on the gpu, I ripped out my cpu years ago” etc

Reiterate our focus on GPU profiling. Suggest working closely with an engineer familiar with cpu

Profiling.

CPU or GPU

Here we're focusing on the GPU, though in production it might not be so easy to talk about them separately.

- Try to partner with an engineer for CPU optimization
- Identify CPU waits in the GPU profiler
- Not an exact science



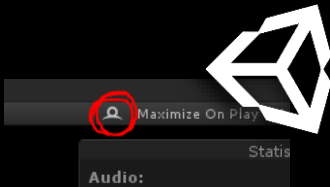
- Kind of a “you got your peanut butter in my chocolate” situation



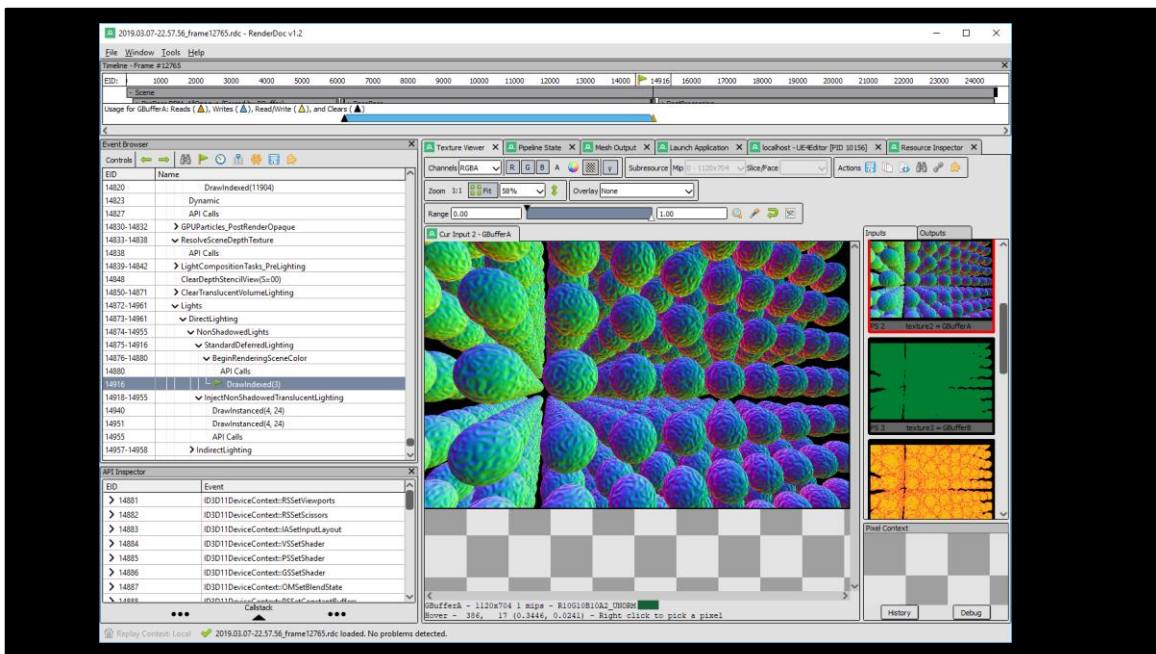
Lots of profilers and tools I wont mention
A whole host of AMD and Intel tools
Various degrees of support and usefulness
We're going to survey A range of tools from General -> Specific

RenderDoc <https://renderdoc.org/>

- Open Source! <https://github.com/baldurk/renderdoc>
- Supports all major graphics APIs (Including Vulkan)
- Unreal & Unity Integration



First up is Renderdoc
My favorite for desktop profiling
Unity will detect your Renderdoc install!
Unreal has a native plugin that can be enabled!



- Renderdoc capture of an unreal profiling frame
- Inspecting steps of the deferred rendering pipeline
 - We'll return to this later

Nvidia Tools <https://developer.nvidia.com/tools-overview>



- Nsight Graphics
 - Many Nsight series tools (Systems, Compute, VS/Eclipse integration)
- Tegra Debugger
 - Specific debug bridge provided by platform partner
- nvProf
 - <https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>
 - Command-line profiler, CUDA support

Most time spent with the tegra debugger

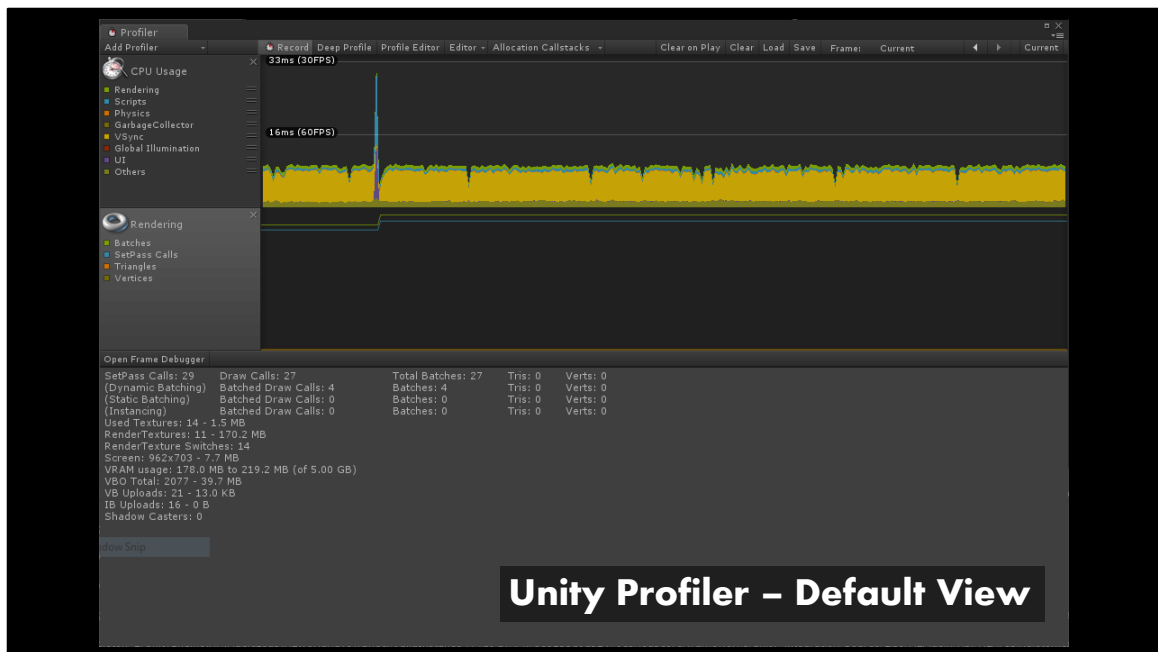
There are just as many AMD tools, as well as PIX, and various Console specific tooling used with dev kits.

Unity Optimization

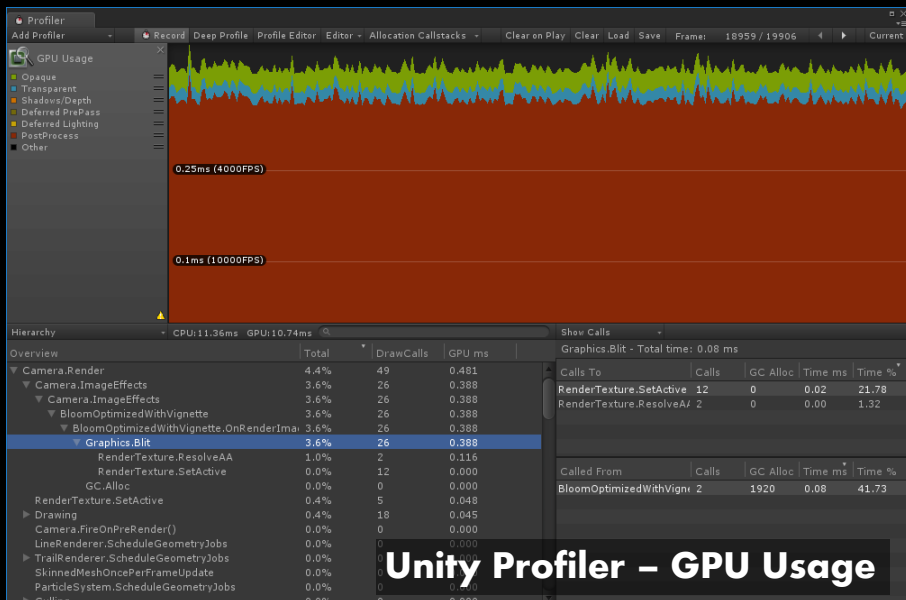
- Frame Debugger
 - <https://docs.unity3d.com/Manual/FrameDebugger.html>
- GPU Profiler
 - <https://docs.unity3d.com/Manual/ProfilerGPU.html>
- RenderDoc Integration
- #IFs and Pragmas in shaders
- New LWRP and SRP
 - Still experimental, worth looking at

- Remember all in-engine profiling may be affected by editor overhead

- Has its uses nice quick overview of what's happening
- Pretty sparse



This is what you see by default, a generalized view



Unity Profiler – GPU Usage

But you can get fairly specific. The GPU Usage tab is closed by default, open that thing up!

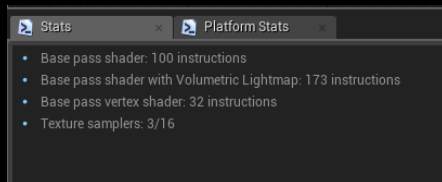
- Here we see a massive chunk of our rendering is our bloom effect. Even with a black frame, screen space effects cost the same.

Unreal Optimization

- Console Commands
 - *r.ProfileGPU, r.SetRes, r.ScreenPercentage, r.ABunchOfOtherStuff*
 - Shortcut: “Ctrl+Shift+,” to show the GPU profiler
 - While running, “stat GPU” to show Live GPU Profiler
- Material Optimization + Feature branching
- Optimization View Modes
- RenderDoc Plugin

- Remember all in-engine profilers are affected by editor overhead

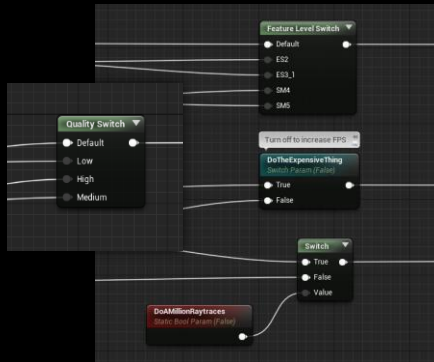
Material Stats



- Shows Instruction Count
 - Vertex Instructions
- Texture Samples

- All engine profilers are affected by editor overhead
- We'll get to platform stats in a minute

Compile Time Switches

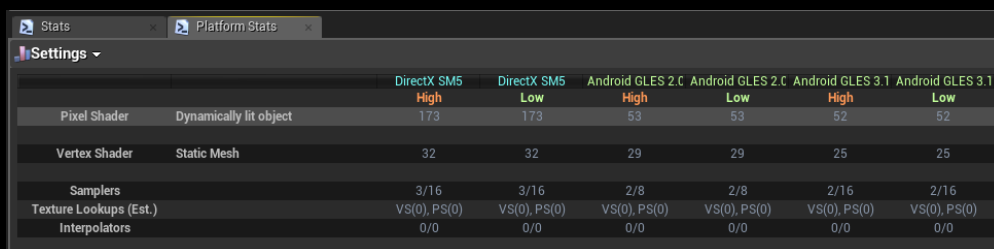


- Feature Level Switch
- Quality Switch
 - `r.MaterialQualityLevel`
- Switch Param
- Switch + Static Bool

- All engine profilers are affected by editor overhead

Platform Stats

- Need platform specific compiler
 - Go grab the Android Mali Compiler



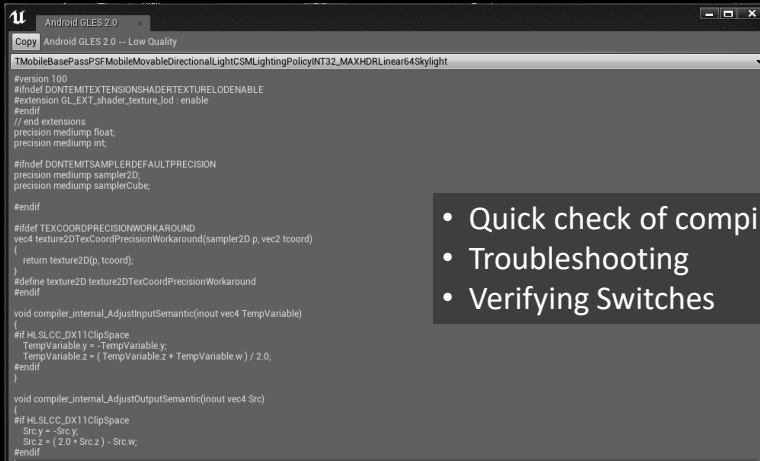
The screenshot shows a 'Platform Stats' window with a 'Settings' dropdown. The table displays performance metrics for various platforms, including DirectX SM5 and Android GLES 2.C, 2.C, and 3.1. The metrics are categorized by Pixel Shader, Vertex Shader, Samplers, Texture Lookups (Est.), and Interpolators.

		DirectX SM5 High	DirectX SM5 Low	Android GLES 2.C High	Android GLES 2.C Low	Android GLES 3.1 High	Android GLES 3.1 Low
Pixel Shader	Dynamically lit object	173	173	53	53	52	52
Vertex Shader	Static Mesh	32	32	29	29	25	25
Samplers		3/16	3/16	2/8	2/8	2/16	2/16
Texture Lookups (Est.)		VS(0), PS(0)	VS(0), PS(0)	VS(0), PS(0)	VS(0), PS(0)	VS(0), PS(0)	VS(0), PS(0)
Interpolators		0/0	0/0	0/0	0/0	0/0	0/0

- You'll need a platform specific shader copiler to get a preview here.

-

Code View



```
Android GLES 2.0
Copy Android GLES 2.0 -- Low Quality
TMobileBasePassPSFMobileMovableDirectionalLightCSMLightingPolicyINT32_MAXHDRLinear64SkyLight
version 100
#define DONT_EXTENSIONS_SHADER_TEXTURE_LOADABLE
#define GL_EXT_shader_texture_lod - enable
#define
// end extensions
precision mediump float;
precision mediump int;

#define DONT_EXTENSIONS_SAMPLER_DEFAULT_PRECISION
precision mediump sampler2D;
precision mediump samplerCube;

#define
#define TEXCOORD_PRECISION_WORKAROUND
vec4 texture2D(TexCoordPrecisionWorkaround(sampler2D p, vec2 tcoord))
{
    return texture2D(p, tcoord);
}
#define texture2D texture2D(TexCoordPrecisionWorkaround)
#define
void compiler_internal_AdjustInputSemantic(inout vec4 TempVariable)
{
    #if HLSLCC_DX11ClipSpace
        TempVariable.y = TempVariable.y;
        TempVariable.z = (TempVariable.z + TempVariable.w) / 2.0;
    #endif
}

void compiler_internal_AdjustOutputSemantic(inout vec4 Src)
{
    #if HLSLCC_DX11ClipSpace
        Src.y = -Src.y;
        Src.z = (2.0 - Src.z) - Src.w;
    #endif
}
```

- Quick check of compiled code
- Troubleshooting
- Verifying Switches

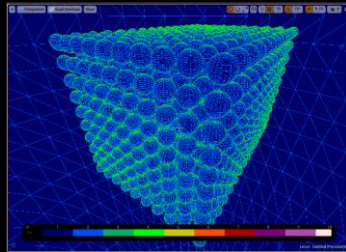
- If this is useful to you, you probably want to be rooting around in the render pipeline source code anyway
- There's some great write-up's of extending the deferred shader online that I'll link at the end

Optimization Viewmodes

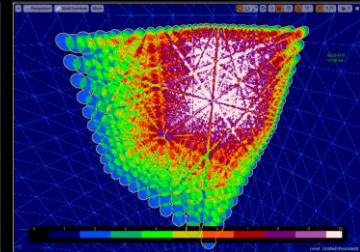
View modes



Quad Complexity



Transparent Overdraw



There are a large number of options here, including visualizing specific gbuffers, to lots of other super useful visualization modes.

- Lighting
- Texel density



When Do I Optimize Again?



Many different mindsets here, I'm just sharing my experiences

Pre-Pro Optimization

- Look for content and performance guidelines for the platform
 - Find out what you're in for
- Predicting Bottlenecks is Difficult
 - Especially on new hardware, a new pipeline, or a new project
- Simple tests can influence art direction and design constraints
 - Custom lighting or Look Dev
 - Custom pipeline or tooling
- Might be wildly off despite best intentions



What can you do to start thinking about optimization in preproduction effectively?

Platform guidelines

- what other projects are doing
- In our case it was a Tegra. Pretty decent, but not magical for VR / AR.

Lighting + lookdev

- Maybe you find out you need to use stylized lighting, or no lighting
- Maybe shadows are off limits
- Maybe you find you have room to do something unique

Pipeline + tooling

- Seeing those needs early can help set your priorities for tools
- Atlasing, batching, baking, polycount, shader complexity

Production Optimization

- Pre-emptive optimization lengthens iteration time
 - Everything takes longer
- Features/Effects will be vital for different projects at different times
 - Communicating a core gameplay idea might be worth the cost
 - That one system you spent 2 weeks optimizing might go in the trash
- Spotting Pitfalls and Practicing Good habits
 - Knowing when to Refactor and when to wait
 - Legibility and Comments > Micro Optimization
 - Periodic auditing of art resources is a good call
 - If work is done preemptively and cut later, you played yourself



Easier with experience – You'll spot your own bad habits over time

Trying to plan too far ahead can be very counterproductive
Much of the work happens, well, during the project

Deadline Optimization

- The project needs are definitely clearer
 - You know the size of the bag (90fps, must fit on Game & Watch)
 - Your to-do list + priorities are obvious
- Collect all the low hanging fruit
 - Texture size
 - Particle Overdraw
 - Lighting and Shadow Settings
 - Neuter Post Effects (4x MSAA to 2x MSAA)
 - Uncheck the MakeGameSlower boolean
- You know how much time is left
 - Makes it easier to throw things in the bin
 - You are hopefully “feature complete”



Go through Lighting and Shadow Settings
Double check everything you can
Audit everything you can

Uh-Oh

- When to raise a red flag
 - your producer or lead has a higher level view by design
 - If something needs to be bubbled up early, don't be afraid to do so
 - it's better to have a feature die earlier than later





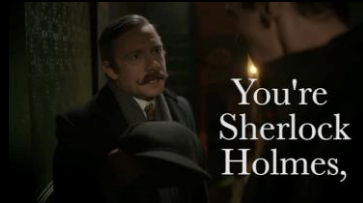
Process + Priorities



Need to add oculus flowchart for troubleshooting here, “when in doubt...” the black box method

Process + Prioritizing

- Understand the Problem
 - Identify problem areas using the tools
 - Form a hypothesis and attempt to verify
 - Repeat
- Understand the Fix
 - Remove the feature and look at the delta
 - The fix will never be faster than that
 - Is it an easy fix, or a tough fix?
 - Is it worth fixing?
 - Is it okay to just cut the feature?
 - Is it more or less important than your other heavy hitters?

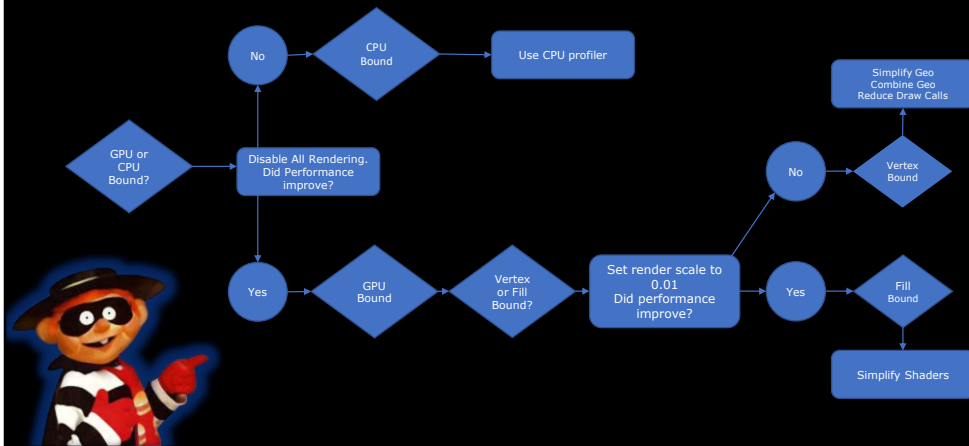


Effects as UI

- Effects communicate core gameplay concepts
 - Success vs Failure
 - Valid vs Invalid (actions, ranges, locations)
 - Inventory, counts, and amounts
 - Cooldowns and timing
- Modifying these effects can change player perception
 - This should be an ongoing conversation
 - Can be dangerous late in production
 - Artists, Animators, Designers, and Producers all bring insight to the table

Oculus Optimization Tutorial

<https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-performance-tutorial/>

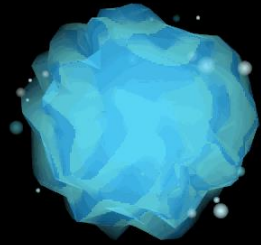


If this looks very familiar to you, good. It's a great chart.

This graph is recreated from an oculus presentation – It describes a process

Utilization: Vertex or Pixel shader?

- Where are your biggest hits?
 - Find out if any of them are dumb mistakes
 - Look for patterns
- Fluctuations over the app lifetime
 - Actors or particles spawn, different levels
 - Are you looking at a particularly expensive effect?
- Can you shuffle work around to even the load?
 - Moving instructions from pixel to vertex shader
 - Computing things offline instead of runtime
 - Baking textures instead of using procedurals
 - Baking lighting, using vertex color instead of ID masks



Spoiler Alert: It's almost always the pixel shader

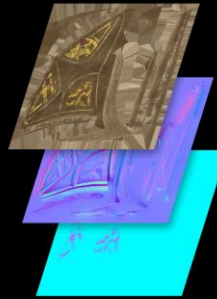
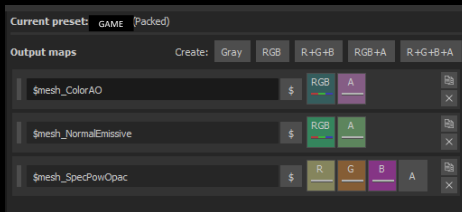
Tips + Tricks

- Shader Branching, Loops, Expensive math
- Overdraw
 - Know your blend modes and passes
- Drawcalls
 - Instancing, batching
- Texture Samples
 - Atlasing, ID Maps
 - Channel Packing and Precision Crimes
- Utilization
 - It's probably the pixel shader
- Hardware Weirdness
 - Tiled Rendering paths on mobile, Android shenanigans



Tips + Tricks

- Channel Packing
 - Substance has a great interface for this
 - Be aware of various compression profiles



- In certain compression schemes various channels are discarded or compressed more harshly
- Sometimes green is given more range, use it for something like roughness
- Be careful with normal maps
- Sub channel packing is possible, works best with masks
- Trading unique samplers for dragging everyone along for the ride, usually worth it

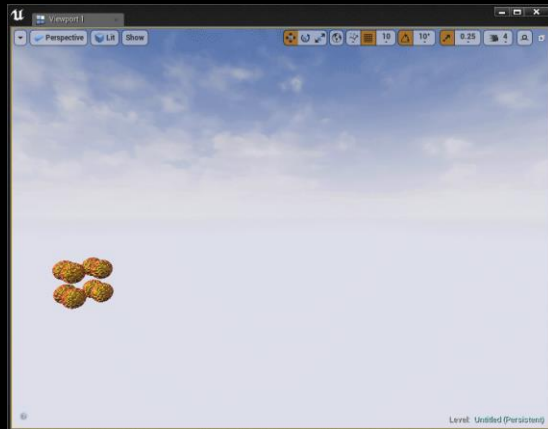
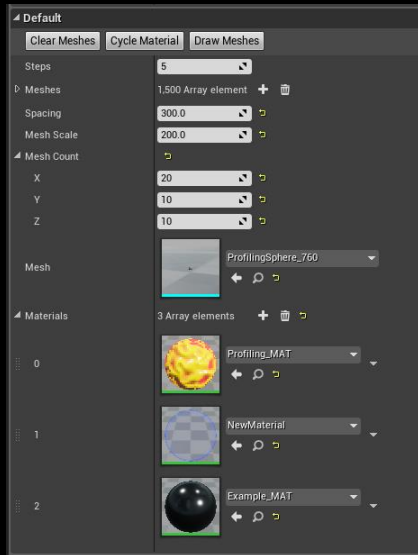
Tips + Tricks

Build a “Benchmark” for your hardware

- Add increasingly more meshes over time
- Make a bunch of unique materials that won't get instanced
- Make a spaghetti procedural mesh
- Render 10 transparent full screen quads
- Add some giant particles for flavor
- Make sure to cast lots of dynamic shadows
- Dump FPS and any relevant data to a log file

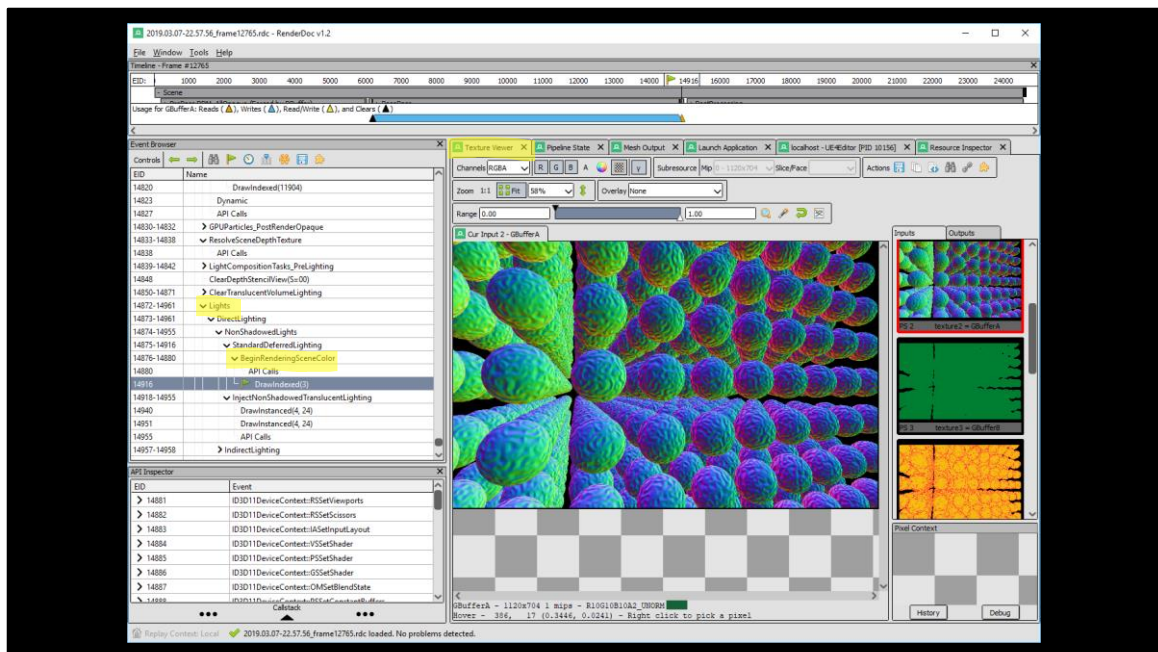
Build a stress test scene that spits out metadata and framerate. Have it be manipulated via key press or controller, if in headset. For bonus points log everything. Do this early, and kill some assumptions and bad intuition with COLD - HARD - DATA.

Example Cheeseпоof Benchmark Tool



I made this silly little guy for looking at overdraw and general shader complexity
Useful at the beginning of a project to benchmark new hardware

- Super simple, just adds a 3D array of mesh components, can cycle through an array of materials
- Logs total Number of tris shown, FPS, The current material
- Could use particles instead, could add post process quads



Heres that same render doc image from before using our test benchmarking tool

- We've drilled down into the lighting section, where we can find the gbuffers unreal is using to light the scene.
- Some of These gbuffers are also exposed in engine through the viewmodes, but not all, and not with this detail.

Example Problem

- What's going on here?
 - We're using Unity
 - We're on an AR Device
 - Our Terrain Mesh Is taking most of the frame
 - It's drawing twice
 - Its not due to stereo rendering
 - It's not a multipass shader
 - There's no duplicate components

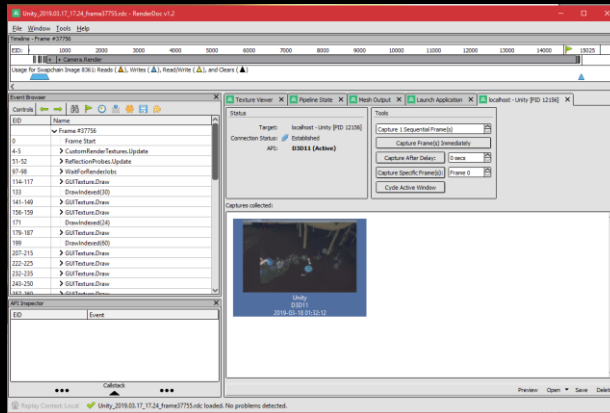
Heres a an example problem, with some clues. I'll show one at a time and give you a second to think about it.

We're on an Ar Device, We know a major hit is the scanned terrain mesh. However, this isn't a CPU problem.

The Terrain Come in as chunks... This is a clue

Even though the Terrain is Opaque, it's thowing up in the transparent Queue... This is a clue.

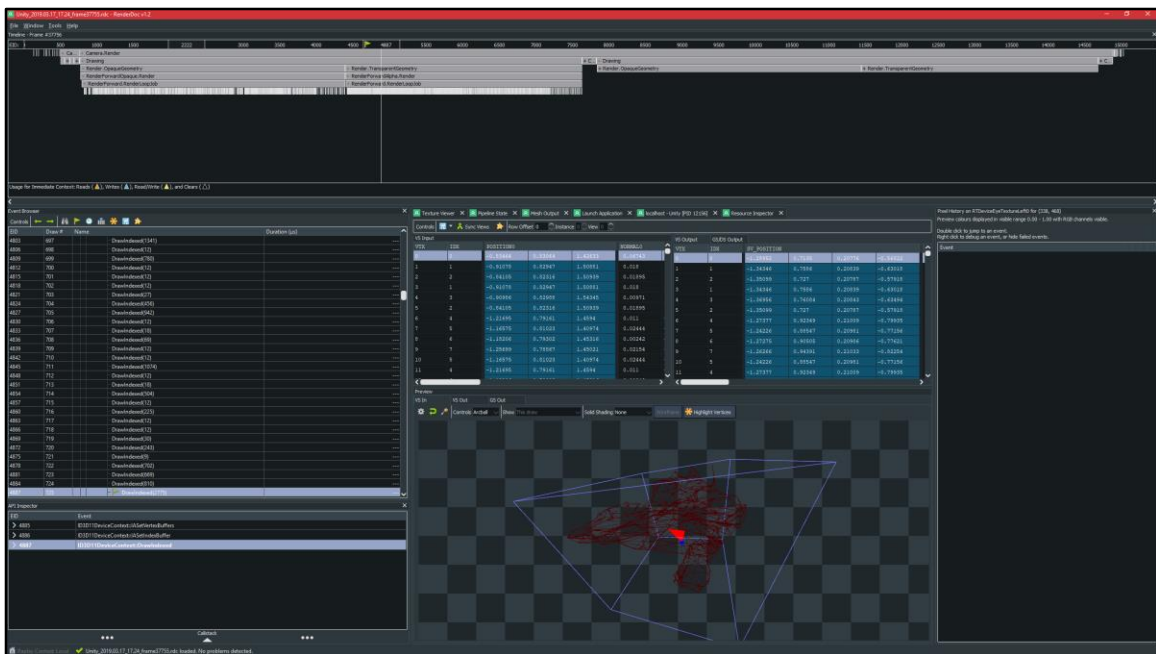
Example Problem



- Frame captured from Unity Editor

This is the default view in renderdoc after capturing a frame from the unity editor

We can see a list of any frames we've captured, with timestamps.
The Timeline view is currently collapsed

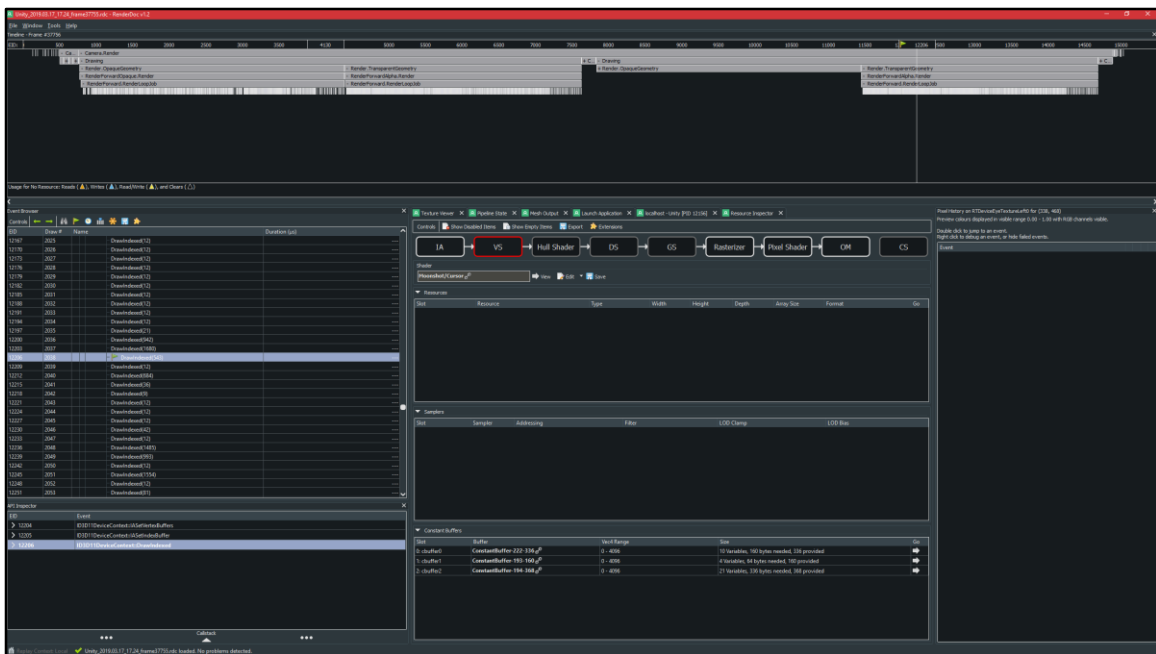


Step 1: Turn on dark mode, it makes you smarter and more powerful

Here we've expanded the timeline view/

We can ignore half because we know we're using multipass stereo, so half of the frame is a duplicate.

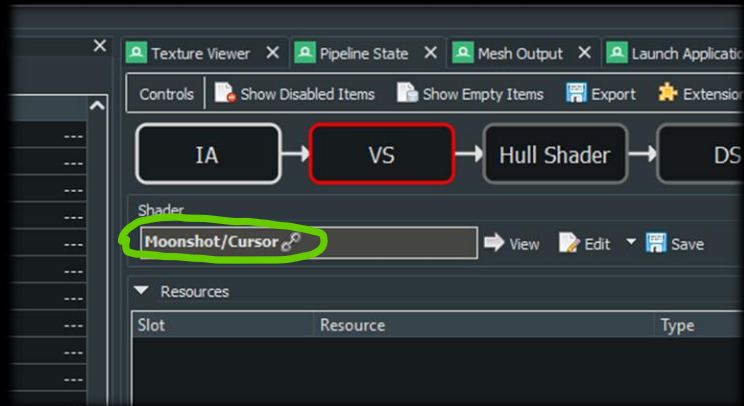
We can also see That about half the frame is used to draw our opaque geo and half for our transparent geo. But we know our terrain meshes should be drawing in the transparent queue at all, so what's the deal?



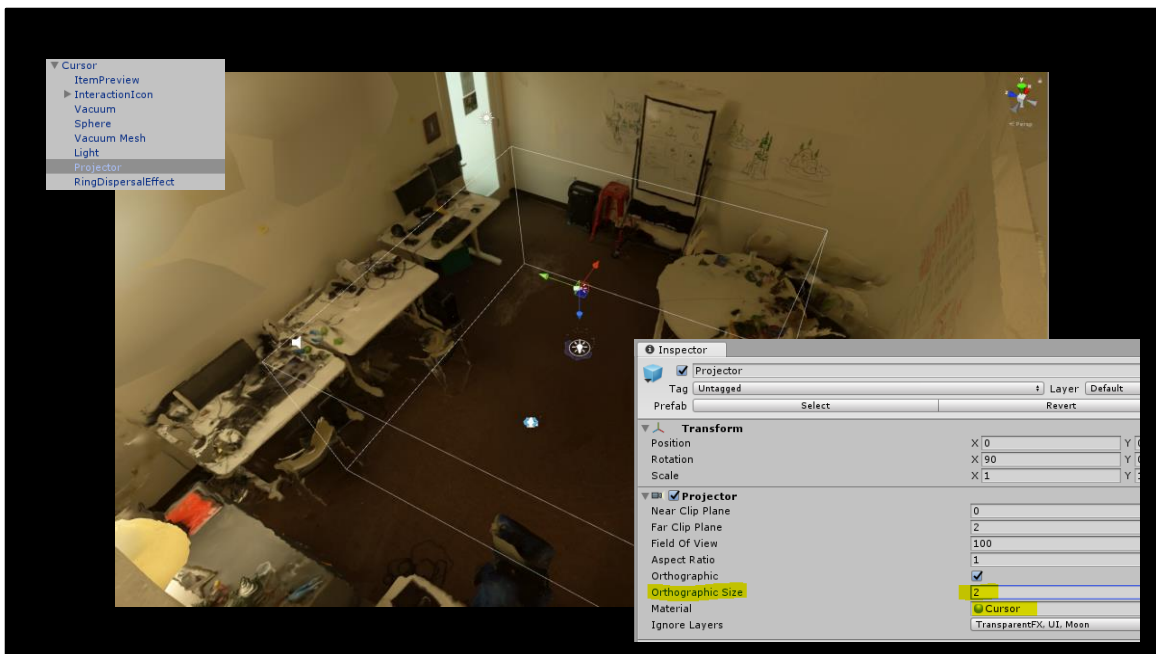
With the call to draw a piece of the transparent terrain selected, lets check out what shaders its using in the Pipeline State tab.

Hmmm.

HMMMMMMMM



AHA! Why is our terrain mesh rendering with the cursor material?
Well, our cursor is projected onto the terrain but... Oh. Oh no.



SO here's the culprit.

That box is not a bounding box.

That's the culling volume unity's Projector Component uses to determine...

Which meshes to DUPLICATE, and render the projector material on.

It's currently set to 2... meters.

Early in development the magic leap meshing api brought in meshes in tiny chunks

Later they updated to large chunks... We increased the culling size to make sure the cursor always drew on enough chunks.

Example Problem

- What happened?
 - It wasn't what we suspected (intuition failed)
 - We thought it was the for loop used in our terrain metaballs (it wasn't)
 - We used the Unity Projector Component for the game cursor
 - We enlarged the culling volume after the terrain chunks became less granular
 - This was a vital feature and while we had solutions, we had no time to implement
 - <http://blog.wolfire.com/2009/06/how-to-project-decals>
 - We decided to optimize as much as we could and eat the cost
 - The effect was vital feedback for gameplay

Sometimes, the correct answer is to let it be.

In this case, we found it so late in production, we had to tighten knobs everywhere else to make room for a vital feature.

We hit our performance targets by optimizing in other areas.

Talking Optimization With Your Team

- Gaining Perspective
 - Put your work in context by communicating across disciplines
- Beware of the Implications
 - Changing things may have consequences



Especially later in production

Talking Optimization With Your Team

- Artists might see a “Black Box”
 - May have partial knowledge or superstitions
 - Good Intentions but poor implementations
 - Many dubious ways to skin a cat
 - Empowering them can have a big impact
 - Junior artists might be all over the place
 - Senior artists might be more conservative than necessary



Talking Optimization With Your Team

- Engineers have different priorities
 - They may not be as focused on the gameplay or aesthetic implications
 - You may need to advocate for an expensive feature
- Trust but Verify
 - Don't take a passing statement from your engineer as gospel
 - Follow up with them and do your own benchmarking


*If we disable all rendering
performance increases dramatically!*

More Stuff

- LearnOpenGL.com
 - “What the heck does any of this mean?!”
- Shader and C# tutorials
 - Alan Zucconi
 - Catlike Coding
 - Unity is a great tool for learning about shaders
 - **Scriptable Render Pipeline** is pretty exciting
- Unreal Deferred Pipeline Breakdown
 - [Matt Hoffman - UE4 Rendering](#)

Further reading and resources

Questions, Comments, grievances

- garrettorious@gmail.com
- @creatosaurus 



Your shaders after this talk