

# Graphics Application Profiler Methodology

## Contents

---

- Preparation
- Problem Statement
- [ RenderDoc][#renderdoc]

## Preparation

---

### Capture Tips

In order to increase the reliability of current capture, you shall

- get a representative and repeatable capture
- close other applications (erase different resource occupations other applications might make)
- disable CPU scaling (prevent CPU from changing clock frequency by the time turbo)

[ [Unity - Optimize your game with the Profile Analyzer](#) ]

## ✧ Problem Statement

---

### | GPU or CPU bound

**CPU bottleneck** happens when the processor isn't fast enough to process and transfer data. The according observation we may made is that CPU is keeping busy to dealing with resources and operations, while GPU hangs waiting for data and command transferred from CPU.

Likewise, **GPU bottleneck** happens when CPU issues commands and data in time, whereas it takes time for GPU to actually consume them. Now, it is CPU waiting for GPU.

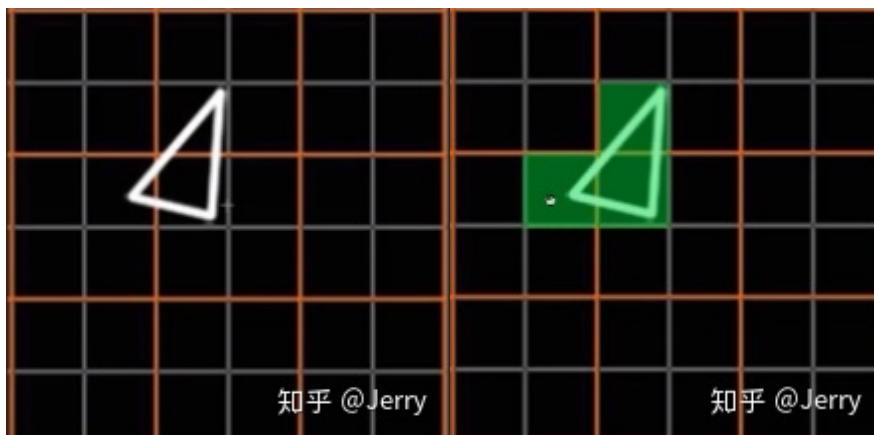
Sometimes, your profiling process might get stuck somewhere in between, which might be because it is bounded by its GPU bottleneck. Thus, first check bottleneck should ease you from suffering.

- Unity: In Unity, there is a variable called `Gfx.WaitForPresent`. If the median of the value is nonzero, then the CPU is waiting for GPU to finish its activity before it can continue.

## H5 Solution: Reducing draw calls

## H5 Problem: Overshading

Overshading happens when rendering polygons from far distance, where the size of polygons appear to be small. The resulting polygon may look like LHS, while after the rasterization, it may look like RHS.



But the hardware specifies the rasterization processes a "quad" at a time, a block of 4 pixels arranges in a 2x2 pattern. Consequently, the GPU actually processes much more pixels than we are expected. The extra pixels are merely accessed but not shaded. Now, imaging a scenario where there are another polygon of equal size being put right next to the current polygon.

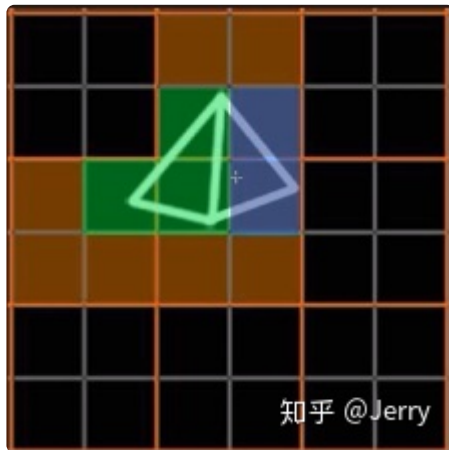


图1

Apparently, for two polygons, there are overlapping pixels (i.e., highlighted red area) meaning they are overdrawn.

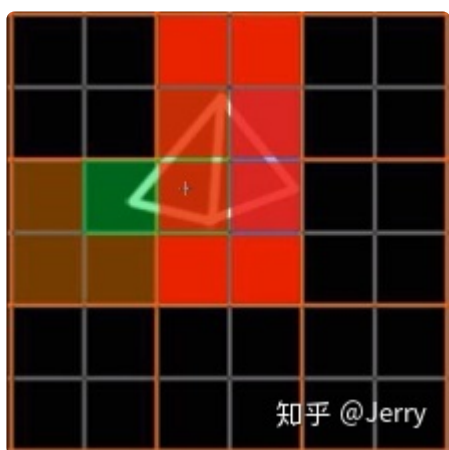
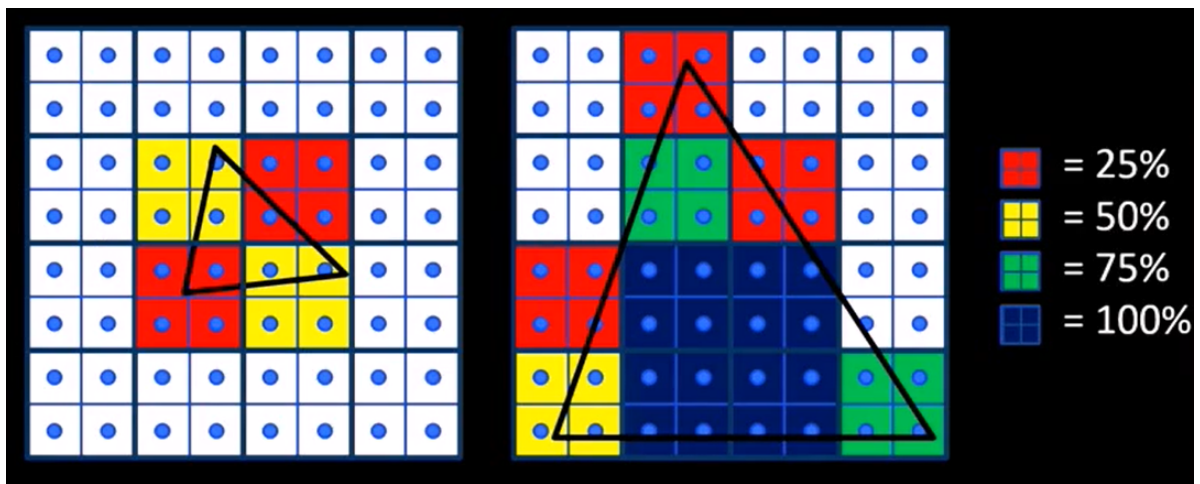
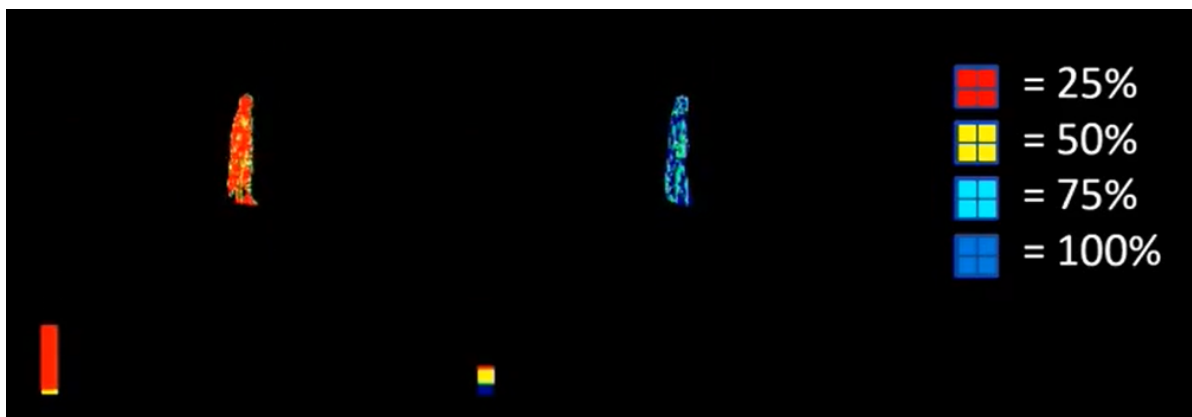
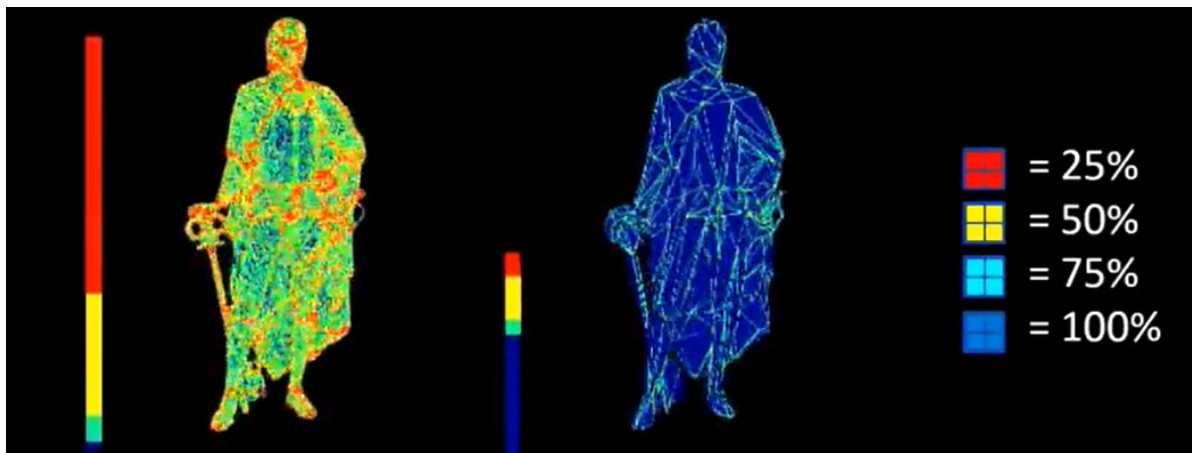


图2



## H5 Solution: lods culling



Latency



The author, Baldur Karlsson, of RenderDoc claimed, "RenderDoc is not a profiler, it is a debugger." [ [Wiki - Profiling](#) ]

## Terminology

For the future reference, there are a certain number of term needs to be clarify.

- Draw call: Draw call is requesting graphics card to render drawable objects.
- Events: Events can be interpreted as API calls in the high level. Events essentially are actions, including draws, dispatches and modifications of resources. Nevertheless, state setting and other CPU-update calls like Map are not included and are available in API Inspector window. Draw calls can be regarded as Events, whereas Events do not necessarily need to be draw calls.

## Mesh Viewer

Mesh viewer can be seen as a debug tool for modelling. The blue dot (1) in Preview is highlighted to illustrate current selected vertex data in VS Input (2). Additionally, RenderDoc highlight the primitive in red (3).

VS Input						VS Output					
VTX	IDX	in_Position			in_TexCoord	VTX	IDX	gl_PerVertex var.gl_Position			
75	89009	189.70868	216.44958	-45.94015	0.68408	21	89003	-176.71118	-285.48682	624.12256	627.75024
76	89007	201.97746	213.37148	-30.63121	0.68408	22	89002	-156.79442	-309.81287	637.2406	640.88147
77	88996	183.20207	206.01884	-22.03451	0.8291	23	89001	-176.89761	-313.48291	632.67798	636.31396
78	89009	189.70868	216.44958	-45.94015	0.68408	24	89004	-156.60347	-279.61304	627.95459	631.58594
79	88996	183.20207	206.01884	-22.03451	0.8291	25	89002	-156.79442	-309.81287	637.2406	640.88147
80	88998	172.4256	209.54945	-37.92979	0.8291	26	89003	-176.71118	-285.48682	624.12256	627.75024
81	89011	176.71375	214.29634	-57.99015	0.68408	27	89005	-176.51341	-256.86902	613.18628	616.80292

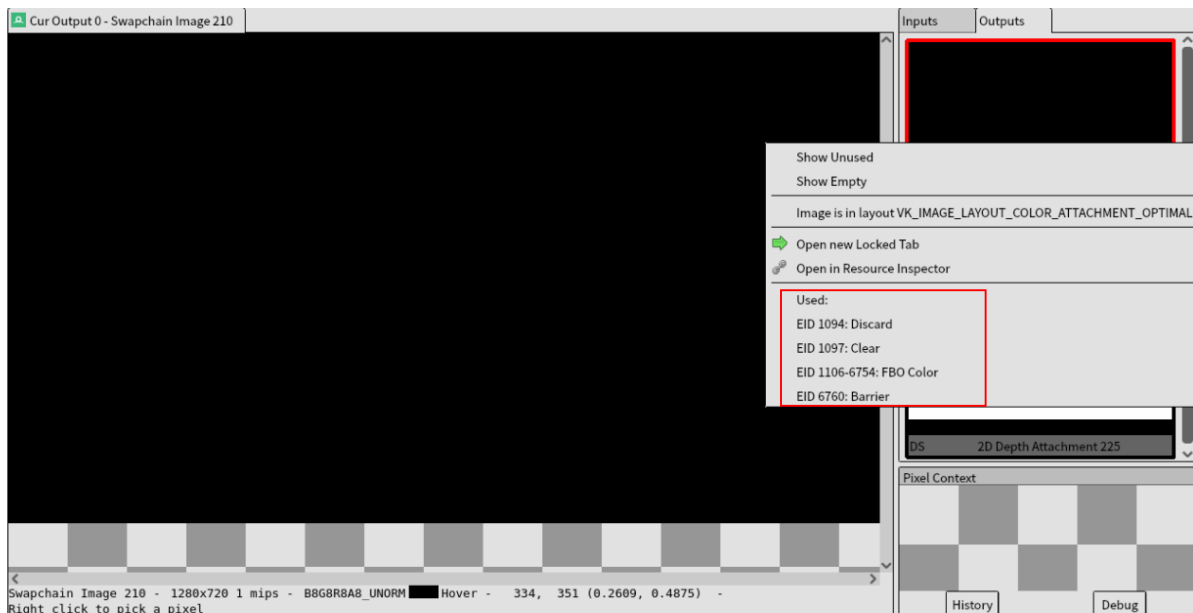
Preview					
VS In	VS Out	GS/DS Out			
Controls	Arcball	Show	This draw	Solid Shading	None
				Wireframe	Highlight Vertices

Similarly, Mesh Viewer also helps to illustrate the final output `gl_Position` after whatever transformation is made within VS Output tab and VS Out in Preview Panel.

## Texture Viewer

By right clicking on the Swapchain Image from Texture Viewer window Outputs tab, you will be able to see how it is used across events.





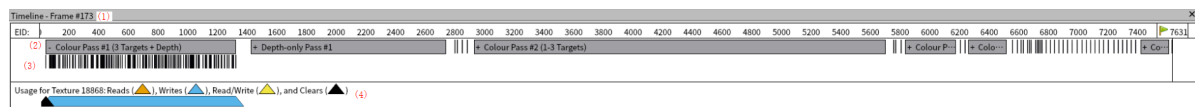
Brief explanation:

- ~~EID 1094: Discard~~ means that in Event ID 1094, the draw call has done nothing to the Swapchain Image on the Framebuffer.
- *EID 1097: Clear* means that the draw call clears the attachment for the Framebuffer.
- *EID 1106-6754: FBO Color* means that there are 5608 many draw calls drawing the scene.
- *EID 6760: Barrier* means that the application is issuing a barrier indicating the Framebuffer is prepared to be rendered.

## Timeline Bar

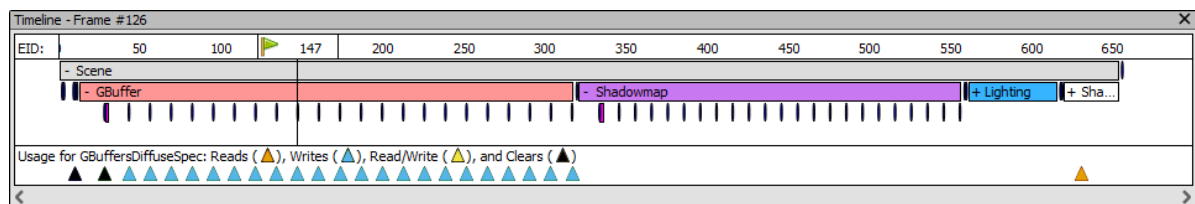
Timeline Bar can be seen as another representation of Event Browser. It displays the timeline (in EID as unit) of every event in a structural way vertically. The UI of Timeline Bar is demonstrated below (ver. Unstable Development Build v1.14, UI may vary across

different versions).



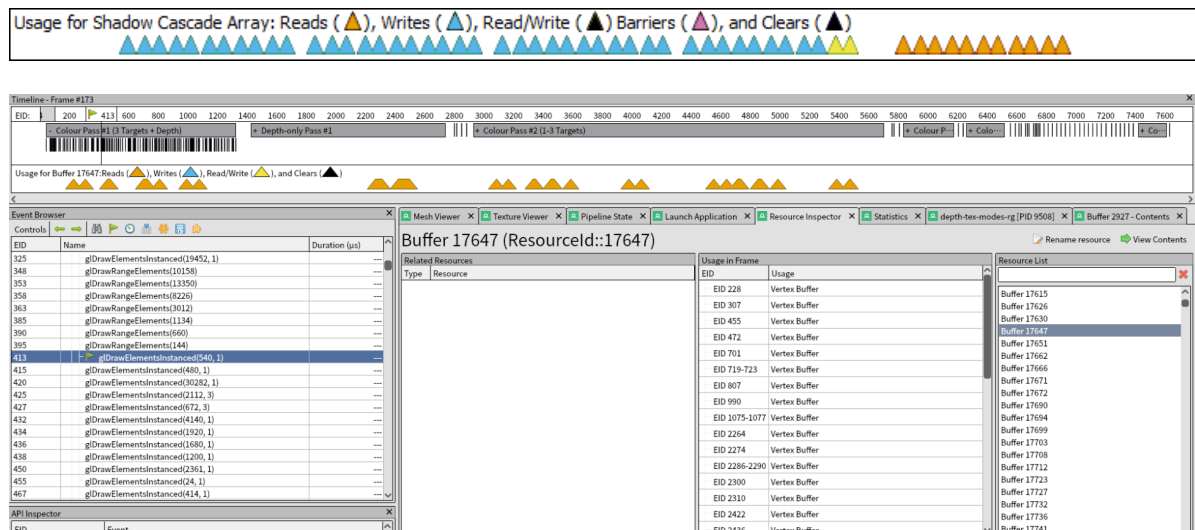
- 1 Frame Number (No.173 frame)
- 2 How long does it take during a pass (First Color Pass)
- 3 Events represented by vertical bars (|)
- 4 A specific resource (E.g., Texture/Buffer) and its usage (E.g., Read/Write/Clear) during the whole frame

An example of a Scene being separated to multiple passes.



Not only can you click a single event in Timeline Bar, but also you are able to inspect the usage of a specific resource from Resource Inspector Window. In this case, Buffer 17647 has only been read during the entire frame shown below.

Another example of usage of some sort of resources.



## Timeline Bubble

If you watch Timeline Bar window closely, you may notice the bubbles like, .

In terms of implementation of OpenGL, it is basically a large state machine. Every time of drawing requires a certain amount of state settings (VAO/VBO/Shader/Program and etc) and maybe driver guesses. A bubble is thereby the result of those state setting functions (be aware of the state settings are not recorded in RenderDoc) imported by application side. They cost both on GPU-side (updating register) and driver-side (**validating and translating** API calls to states, where guesses come).

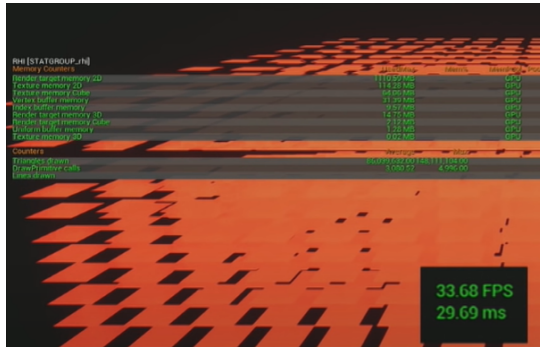
## Reducing draw calls

Draw calls have a larger impact on performance than polycount.

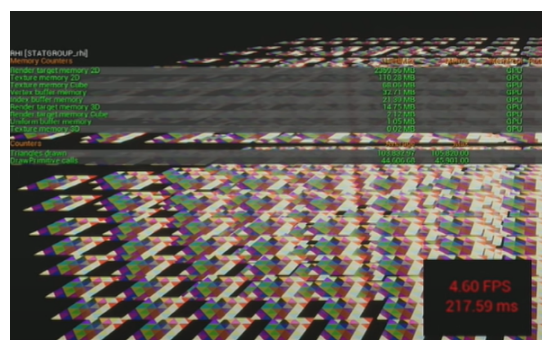
Why are draw calls have such a significant impact on performance?

...

## 86m Triangles in 3000 draw calls



## 103k Polygons in 44k drawcalls



## Modelling

Use rather fewer larger models small ones than many more bigger models also has its consequences. There are trade-offs between different complexity of models:

- Worse for occlusion
- Worse for lightmapping
- Worse for collision calculation
- Worse for memory

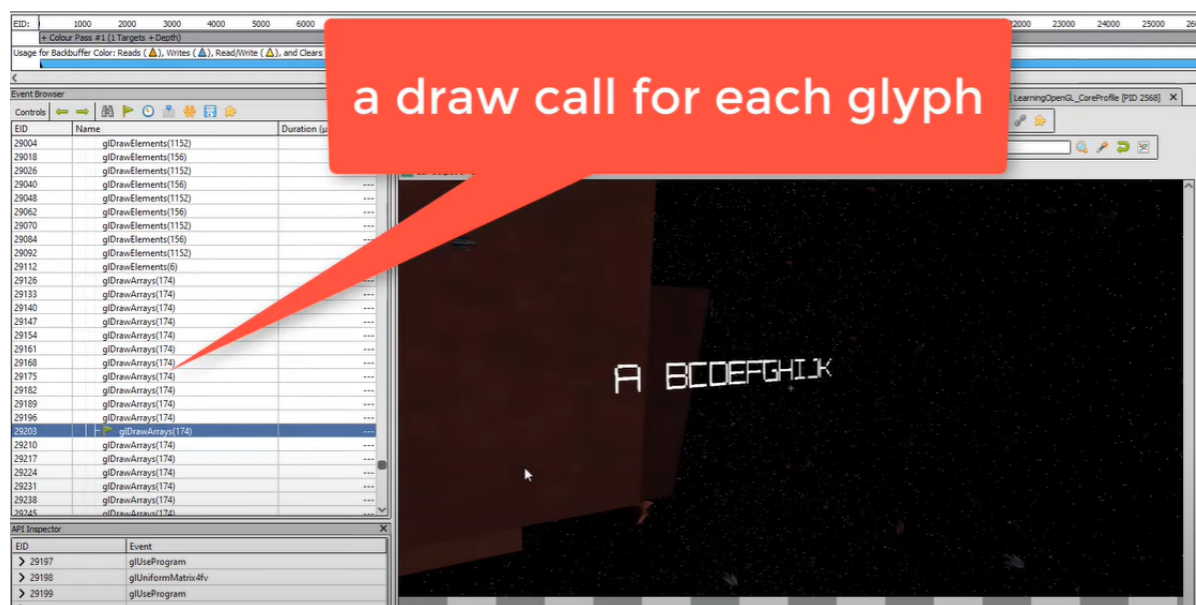
## LOD

## Group draw calls

By browsing events/draw calls step-by-step, you might come across the scenario where a series of consecutive draw calls are drawing the same meshes. In some game engines, even though you put a group of objects altogether, which have the exact same meshes, textures, materials and everything, they are seen as independent objects.

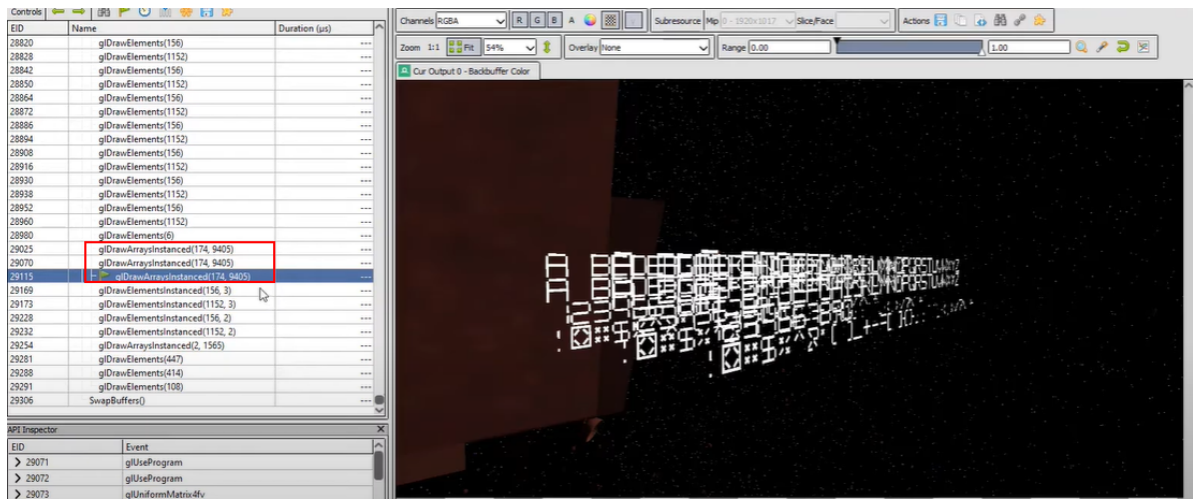
Therefore, in order to reduce the number of draw calls of rendering the same meshes, you may want to use draw instance.

Without Instancing,



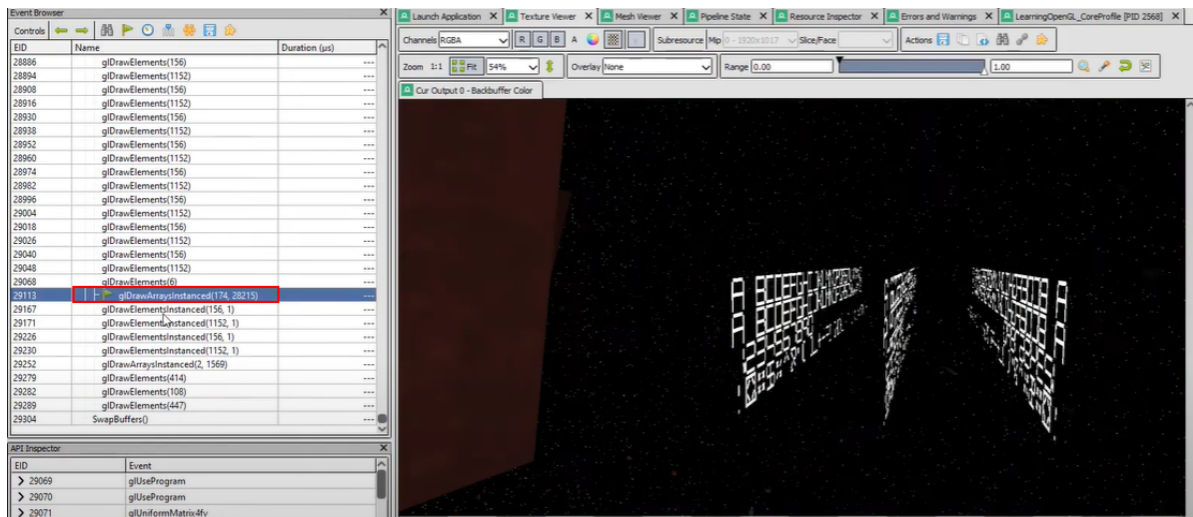
Each draw call `glDrawArrays(174)` is drawing a single glyph.

With Instancing,



As a result of applying instancing, all glyphs are drawn once and for all, other than that the application still draw the full set of glyphs three times in total.

Potentially, with batching (current live functionality in Unity)



Finally, the number of draw calls is reduced to 1. And the result of performance test with each level of optimization is demonstrated in table below.

	Without Optimization	With Instancing	With Batching
Frame Rate (fps)	~10	~32	~44

## Edit Shader

In the **Pipeline Sate Window**, the UI for all sorts of shader stages are demonstrated as follows,



- VTX: Vertex Input, Vertex Buffer Data State
- VS: Vertex Shader
- TCS: Tessellation Control Shader
- TES: Tessellation Evaluate Shader
- GS: Geometry Shader
- RS: Rasterizer
- FS\*: Fragment Shader (Pixel Shaders in Windows Platform)
- FB: Framebuffer
- CS: Compute Shader (missing arrow in the front - separate pipeline)

**i** \* Pipeline Shader Stage may vary due to the platform.

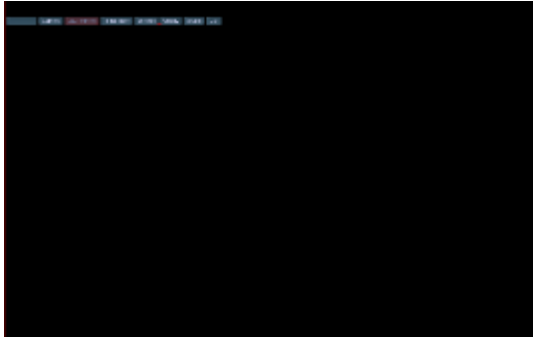
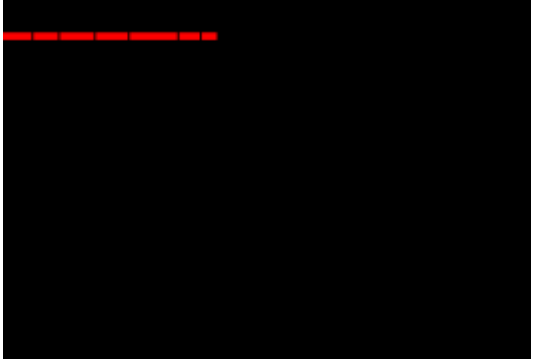
Take a fragment shader as example. The shader ID we are about to modify is 2911.



And the Edit button will promote us to the built-in shader editor. According to the main.glsl file, we may infer that `s_frag_data_0` is sampling from the input texture. Then, let us carry out a experiment on it and see how modified `s_frag_data_0` has affect on final results.

```
main.glsl
1 #version 150
2 uniform sampler2D s_texture_0;
3 in vec4 s_texcoord;
4 in vec4 s_color;
5 out vec4 s_frag_data_0;
6 out vec4 s_frag_data_1;
7 void main() {
8     s_frag_data_0 = texture(s_texture_0,s_texcoord.xy) * s_color;
9     // Edited Contents
10    // s_frag_data_0 = vec4(1.0,0.0,0.0,1.0);
11    s_frag_data_1 = s_frag_data_0;
12 }
13
```

Clicking the Refresh button would simultaneously compiles the new shader with compilation tools (glsl/hlsl/SPIR-V).

Before	After
	

As to the glsl format shader, it is easy to edit and re-compile. On the contrary, to edit the Vulkan shader is lot harder, after all SPIR-V is a intermediate language. RenderDoc accordingly provide with three solutions, decompiling SPIR-V into GLSL, HLSL or SPIR-V Asm.

- Decompiled GLSL requires (i) glslangValidator (ii) ES shader version 310 or higher







GLSL ES version	OpenGL ES version	WebGL version	Based on GLSL version	Date	Shader Preprocessor
1.00.17 <sup>[14]</sup>	2.0	1.0	1.20	12 May 2009	#version 100
3.00.6 <sup>[15]</sup>	3.0	2.0	3.30	29 January 2016	#version 300 es
3.10.5 <sup>[16]</sup>	3.1		GLSL ES 3.00	29 January 2016	#version 310 es
3.20.6 <sup>[17]</sup>	3.2		GLSL ES 3.10	10 July 2019	#version 320 es

- Decompiled SPIR-V Asm requires spirv-as tools
- Decompiled HLSL requires either glslangValidator or Direct3D compiler - dxc

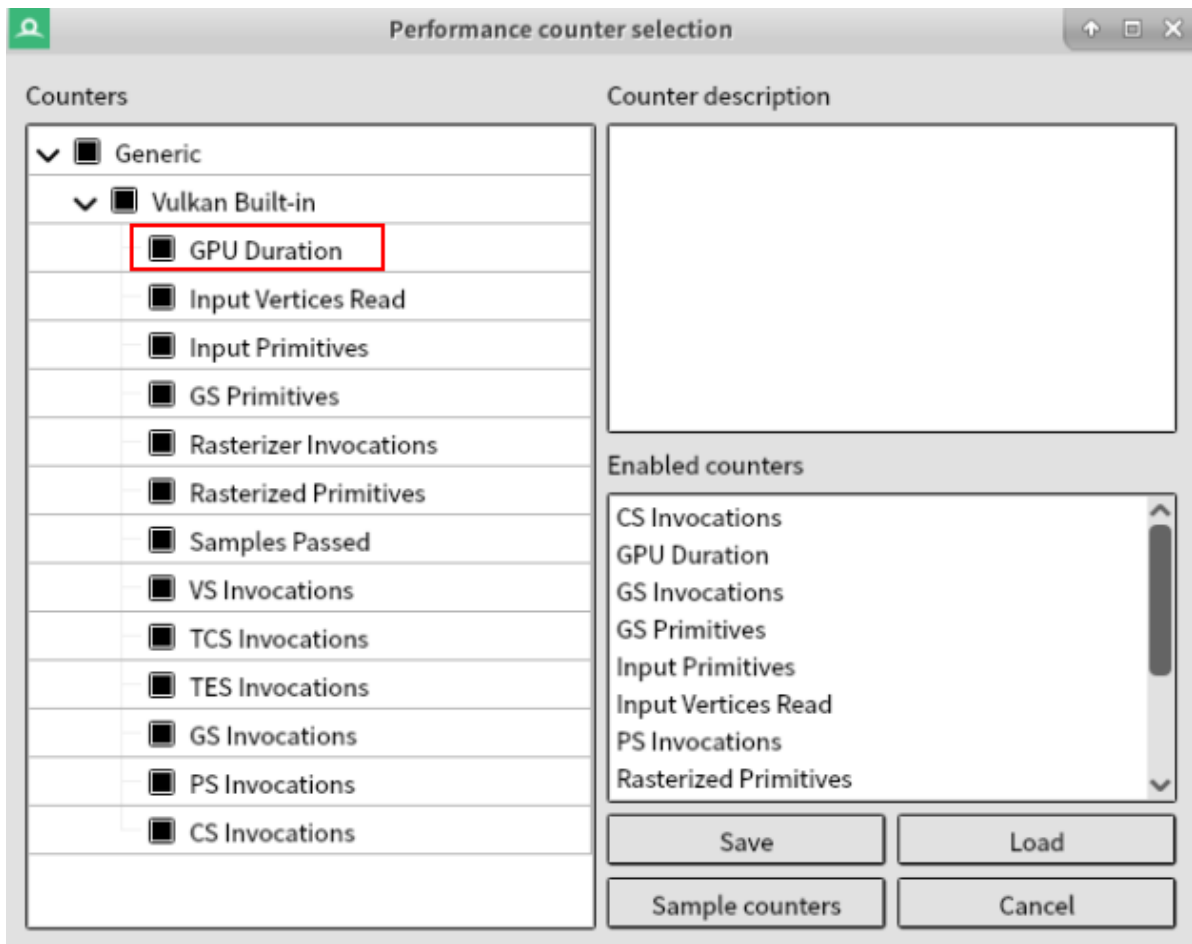
## GPU Timing

There are two ways of getting GPU duration in RenderDoc:

- 1 To time the GPU **duration** of each event, click the timer button  from Event Browser window. And you will get results in microseconds as illustrated below.

EID	Name	Duration (μs)
1494	vkCmdDrawIndexed(66, 1)	129.76
1500	vkCmdDrawIndexed(24, 1)	134.56
1506	vkCmdDrawIndexed(66, 1)	116.00
1512	vkCmdDrawIndexed(24, 1)	115.84
1518	vkCmdDrawIndexed(162, 1)	114.56
1524	vkCmdDrawIndexed(48, 1)	143.52
1530	vkCmdDrawIndexed(108, 1)	157.92
1536	vkCmdDrawIndexed(336, 1)	161.60
1542	 vkCmdDrawIndexed(5424, 1)	269.44

- 2 Capture GPU Duration counters from Performance Counter Viewer window.



However, it is important to learn that not to trust the absolute metrics. That is saying, do not believe No.1542 drawcall takes exactly  $269.44 \mu s$ . Yet, you should **believe the relative metrics**, which is, because of the same sample and query procedure, the ratio of GPU duration between No.1536 and No.1542 should maintain relative steady.

Example of why absolute metrics should not be trusted.

i	1542	vkCmdDrawIndexed(5424, 1)	269.44
	1542	vkCmdDrawIndexed(5424, 1)	263.52
	1542	vkCmdDrawIndexed(5424, 1)	120.16
	1542	vkCmdDrawIndexed(5424, 1)	120.32

# Sample GPU Counters

- GPU Duration (?in confusion?) might not the same as Duration listed in Event Browser

2257	vkCmdDrawIndexed(6, 1)	109.28	2247	4.64	30	10	0
2262	vkCmdDrawIndexed(375, 1)	115.84	2252	4.64	222	74	0
2267	vkCmdDrawIndexed(30, 1)	115.68	2257	4.64	6	2	0
2272	vkCmdDrawIndexed(45, 1)	120.80	2262	4.48	375	125	0
2277	vkCmdDrawIndexed(72, 1)	127.84	2267	4.32	30	10	0
2282	vkCmdDrawIndexed(654, 1)	131.20					

- Vertex Shaders related variables
  - Input Vertices Read: Number of vertices read by input assembler
  - (the number of) Input Primitives
  - VS Invocations

The kinds of Input Primitives are multiple, including points, lines, line strips, triangle strips, triangle fans and etc. Yet, normally, the application uses triangles to composite meshes. And thus, we would have,

$$\text{Input Vertices Read} = 3 \times \text{Input Primitives},$$
where the constant 3 refers to the number of sides of primitives (i.e., 1 as in points, 2 as in lines, 3 as in triangles). Now, given a figure of a set of draw calls, we can easily determine the topology the primitives with such two variables.

Input Vertices Read	Input Primitives
162	54
48	16
108	36
336	112
5424	1808
426	142
108	36
84	28
1950	650
174	58
1074	358
72	24
174	58
984	328
174	58

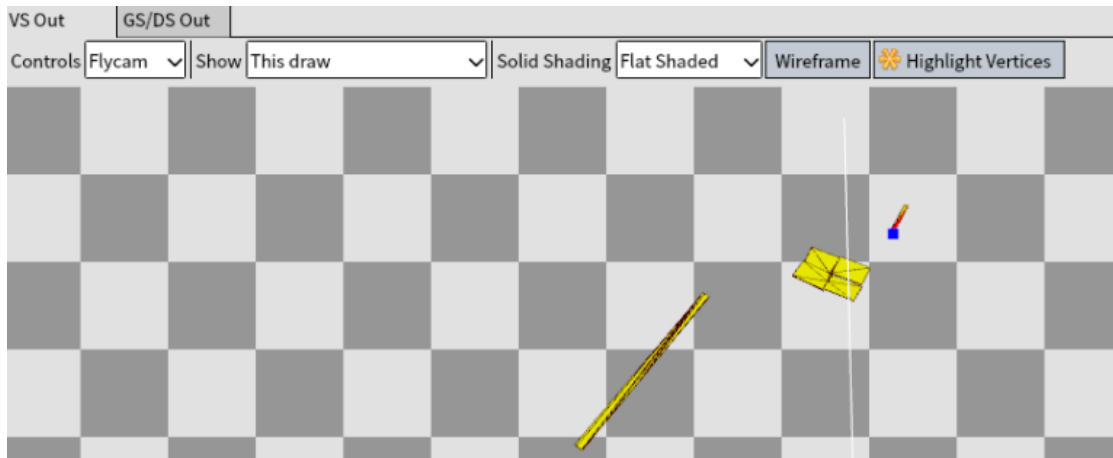
In this case, there is a constant ratio between the first column (Input Vertices Read) and the second column (Input Primitives), which is 3. Therefore, we know the meshes must be composited by triangles.

- Rasterizer related variables
  - Rasterizer Invocations (RI): Number of primitives that were sent to the rasterizer.
  - Rasterizer Primitives (RP): Number of primitives that were rendered.

It is common to realize RI (the number of primitives that were sent to the rasterizer) should be equal to RP (the number of primitives that were rendered). Well, it is not necessarily true, when you are taking clipping into account. Take an event as example.

EID	GPU Duration (μs)	Input Vertices Read	Input Primitives	GS Primitives	Rasterizer Invocations	Rasterized Primitives
2292	4.16	180	60	0	60	58

In this scenario, Rasterizer Invocations seems to have a difference in 2 units with Rasterized Primitives. Now, what does the projection view looks like?



Apparently, there are two triangles are located outside the frustum, so that they are not being rendered at all.



[Official full explanation of all GPU counters mentioned above](#)

## Reference

---

- 1 [Vulkan Session @ GDC 2016 Part II](#)
- 2 [Profiling and Optimization in UE4 | Unreal Indie Dev Days 2019 | Unreal Engine](#)
- 3 [Github RenderDoc/Wiki/Optimization](#)
- 4 [RenderDoc Document](#)
- 5 [UNREAL ART OPTIMIZATION](#)

- 6 [How do I make my game faster ? Introduction to GPU profiling : UE4](#)
- 7 [知乎-GPU分析工具RenderDoc使用](#)
- 8 [Respawning, C++ Profiling, RenderDoc, and Instancing vs Batching](#)
- 9 [GDC2014-Speed Up Your Game Using Simplygon \(Presented by Simplygon\)](#)