PS4 basic profiling with Razor CPU

Introduction

Razor for PS4(TM) is a profiling tool that is part of the PS4 SDK. It exists in two forms: **Razor CPU** for overview and CPU-related profiling, and **Razor GPU** for detailed GPU profiling. These tools are invaluable for investigating performance issues. This document covers **Razor CPU**.

On a PS4 SDK installation with default paths, Razor is located at:

- **Razor CPU**: "C:\Program Files (x86)\SCE\ORBIS\Tools\Razor\binx64\orbis-razorx64.exe"
- **Razor GPU**: "C:\Program Files (x86)\SCE\ORBIS\Tools\New Razor GPU\bin\PS4RazorGPU.exe"

While the Unity Profiler can be used to see where time is being spent and investigate performance, framerate and glitch issues, Razor provides more detailed information that helps when improving performance.

Getting started

Razor is used for two main activities: finding general framerate problems, and finding specific short-lived spikes or glitches.

For general framerate investigations, the first step is to use **Razor CPU** to establish whether your game is CPU-bound or GPU-bound. The difference is as follows:

- **CPU-bound**: Framerate is being dictated by the work done on the CPU. This could be script work in fixed, general or late update, or the CPU side costs of animation, draw calls and physics work. Note in particular that a title *can* be CPU-bound on rendering work (draw calls in particular). This is where Razor can provide more detail: just because the Unity profiler suggests a high load in rendering, it may not mean the title is actually GPU-bound.

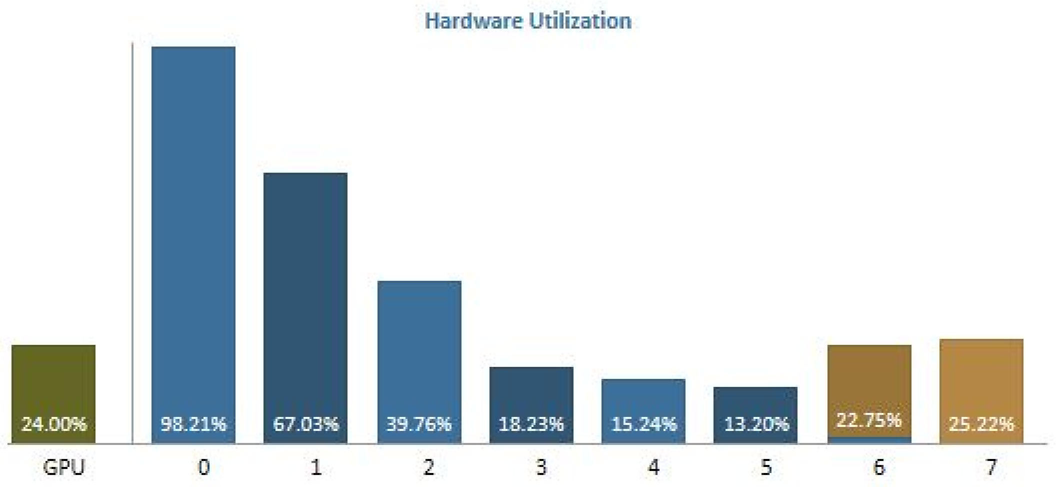
- **GPU-bound**: Framerate is being dictated by the workload on the GPU. This could be Shader cost, Image Effects, or complex geometry (very large numbers of polygons).

If your game is CPU-bound, then altering Shaders, reducing poly counts, Textures and using cheaper Image Effects has no effect on your framerate. Likewise, optimising script code won't make any difference when your game is GPU bound. It is therefore very important that you establish where to start with your optimizations. It is also sometimes useful to know where you have opportunity. For example, if your title is running at an acceptable framerate but CPU activity far exceeds GPU activity, then you might consider adding GPU work (such as Image Effects) without any impact on the frame rate.

Razor offers a number of different types of profiling captures, and the SDK documentation describes these in great detail. The most basic capture, and the best one to start with, is the **Concurrency Capture**, which shows core usage and thread activity. Razor's **Timeline** view also allows you to see the standard Unity profiling markers in this type of capture; however, you can only see these in a Development build.

To grab a Razor **Concurrency Capture**, run a Development build of your game on the devkit and launch **Razor CPU**. Go to **Capture** > **CPU Trace With Options** and select **Concurrency Capture,** with all other options left at default. Press **Capture** to start. Press **Stop** after you have sampled what you're interested in. Razor will then process the data and show you the information.

The initial **Summary** page shows a Hardware Utilization graphic. This is the best way to quickly establish whether your title is CPU- or GPU-bound. Here is an example:

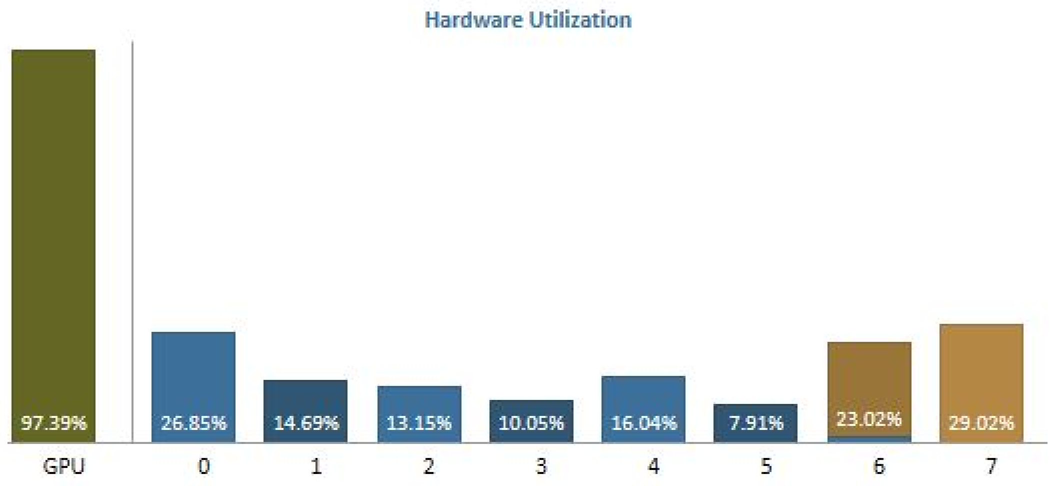


The leftmost column is the GPU workload - here it is at 24% of the total frame time averaged over the highlighted area of capture. The bars to the right of the vertical line 0-7 are the CPU cores; blue shading indicates user work, and orange indicates OS work. In this case, the title is CPU-bound on **Core 0**, which is the Unity Main Thread. Doing a lot more GPU work would not affect the frame rate. Conversely, to improve the frame rate, it is necessary to focus on the CPU workload on the main thread (Core 0) - script code, for example, runs on this core.

Unity’s CPU core usage looks like this:

- **Core 0** (Main Thread): scripts, scheduling jobs
- **Core 1** (Render Thread): takes Unity draw commands and issues native GPU instructions
- **Cores 2-6**: all other threads, including jobs, FMOD and mono threads. Note that core 6 is shared with the OS, hence the blue and orange shaded areas in the picture above.

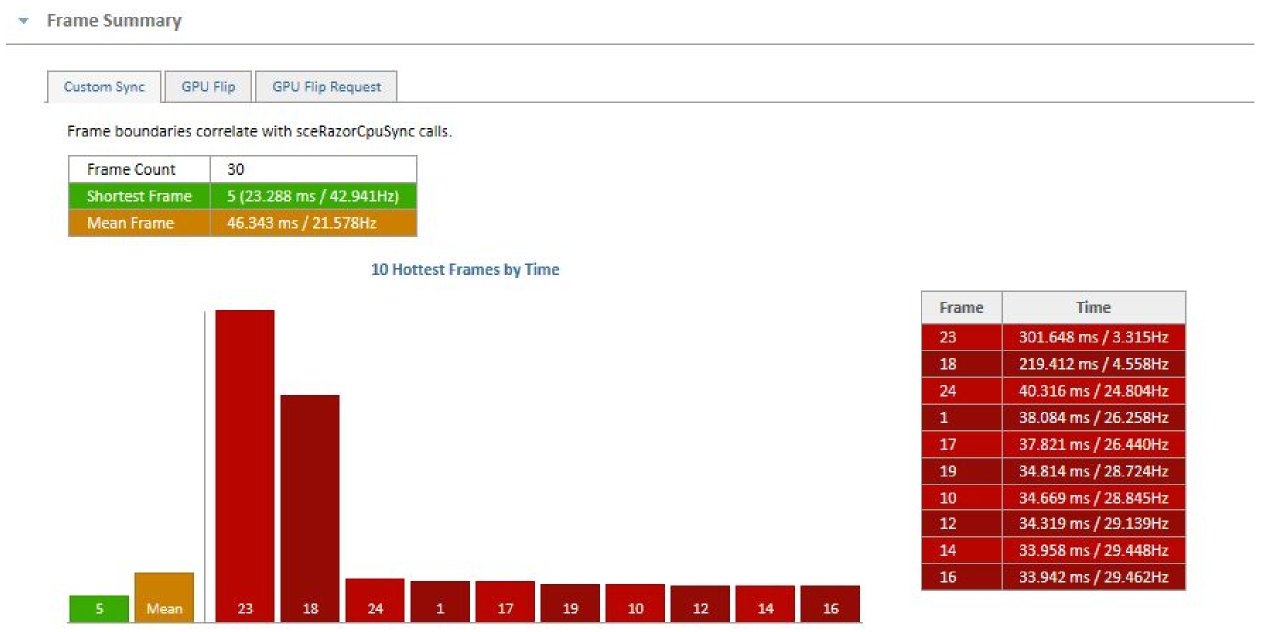
Here's a capture showing a GPU-bound situation:



The frame rate is being governed by GPU activity. Optimizing script code, for example, has no effect on frame rate. **Razor GPU** can then be used to get more information. To reiterate, once you know whether your frame rate is down to CPU or GPU workload, you can make decisions about where to spend time optimizing.

Glitches & spikes

A **Concurrency Capture** can also be used for finding the cause of short term spikes or glitches. Perform a capture as above and look at the **Frame Summary** section, at the summary page. Here is an example:


Frame Summary

Frame boundaries correlate with sceRazorCpuSync calls.

Frame Count	30
Shortest Frame	5 (23.288 ms / 42.941Hz)
Mean Frame	46.343 ms / 21.578Hz

10 Hottest Frames by Time

Frame	Time
23	301.648 ms / 3.315Hz
18	219.412 ms / 4.558Hz
24	40.316 ms / 24.804Hz
1	38.084 ms / 26.258Hz
17	37.821 ms / 26.440Hz
19	34.814 ms / 28.724Hz
10	34.669 ms / 28.845Hz
12	34.319 ms / 29.139Hz
14	33.958 ms / 29.448Hz
16	33.942 ms / 29.462Hz

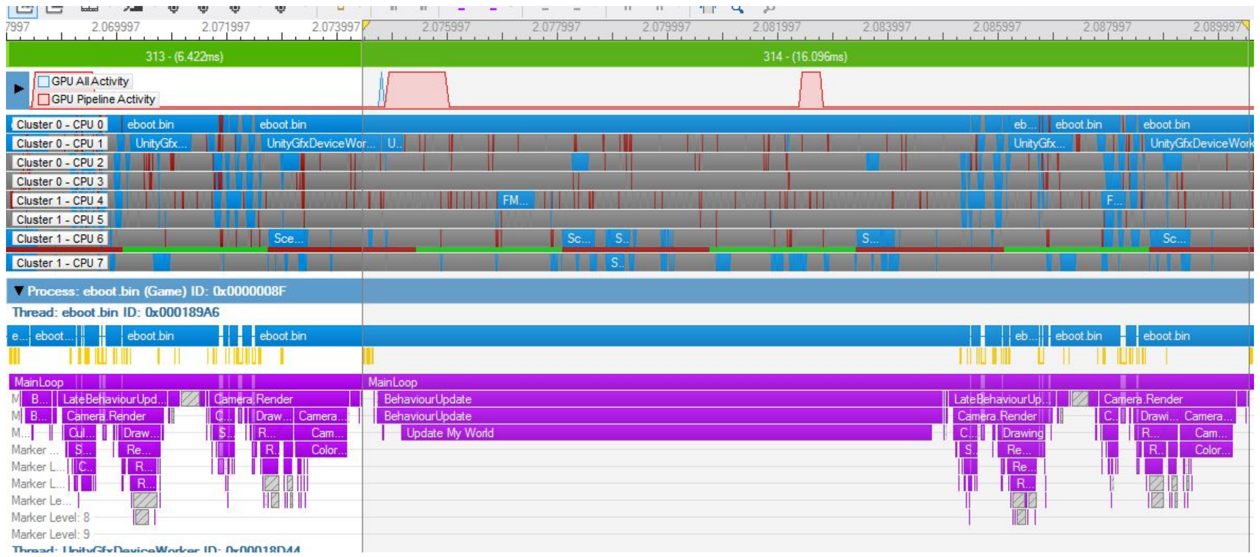
Here you can see that the capture caught 30 frames and while most were in the 30 something ms range, two frames were much slower taking over 200ms. These were frames 23 and 18. On the table on the right, click on the frame you're interested in to jump to the Timeline view and show what was going on during that frame.

Timeline view

You can switch Razor into a **Timeline** view where all captured frames are available and the Unity profiler markers can be seen (in a development build). Be sure to try this out when learning to use Razor to investigate performance. It is selectable in the **Current View** drop down box.

For spikes, use the **Timeline** to see where exceptional time is spent and investigate issues. There should be some stand-out remarkable difference between slow frames and the regular ones. Use that information to improve the spikes.

Here is an example of a spike in Main Thread in the **Timeline** view, in a Development build:



The view is zoomed to show data for two frames: frame #313 at left, and frame #314 at right. At the top of the view, over the green background, you can see that frame #313 had a duration of 6.422ms, which is an average frame for this example project. However frame #314 is a spike frame with a duration of just over 16ms.

Below it, there is an area showing usage of each CPU core over time, represented with blue bars. You can see CPU 0 is almost always fully utilized, suggesting that something running on Unity's main thread produced the spike.

Further down, you can see the profiler markers, shown as purple bars. It’s often the case that you can find the cause of a spike by comparing the length of those bars, one frame against the other. In this example, the bar lengths are very similar, except for the one labeled "Update My World", which at the same time makes its parent markers, "BehaviourUpdate" and "MainLoop", much longer. This allows you to conclude that code inside the "Update My World" marker is causing the spike. This marker is in fact a user marker added in script code (using Profiler.BeginSample) inside a Update call in a script, which is why it’s inside Unity’s "BehaviourUpdate" marker. Using User Markers is a good way to investigate where time is being spent in your scripts.

Summary

- Use **Razor CPU** to get an overview and establish if your game is CPU or GPU-bound
- Use **Razor CPU** to investigate glitches and spikes
- Razor is best used on a development build where Unity profile markers will be visible in the timeline view.

- A non-development build is slightly faster so for the most accurate general framerate assessment, use a non-development build for final testing. Note that even here, there is a slight overhead in using Razor.

- If you create your own threads, particularly native threads, **Razor CPU** allows you to investigate possible conflicts with Unity runtime activity including seeing affinity, priority and core assignment for all threads in the timeline view.

- Profile often throughout development so you keep on top of performance.

To get the most from Razor it is essential to use the SDK documentation. While a quick overview can be obtained from the information above, **Razor CPU** and **Razor GPU** are very powerful tools with a lot of options. The more time spent learning those options, the better informed you will be about your game's performance. Find more information in the Razor CPU User's Guide and the Razor GPU User's Guide.