1 Profilers

1.1 Instrumentation Timers

Instrumentation timers are used to profile the engine and game. Since these times introduce overhead, serialization of the timing data is preformed on a separate-thread to prevent blocking the main-thread during measurement.

In "Dev" build configuration, the timings are automatically captured and outputted to *Results.json*. These results can then be analysed in two methods:

1. Visualise in a timeline using the website https://ui.perfetto.dev/ (formerly chrome://tracing)
2. Visualize in a pie-chart using a python script located in *InsturmentationVisualizer/InstumentationVisualizer.py*.

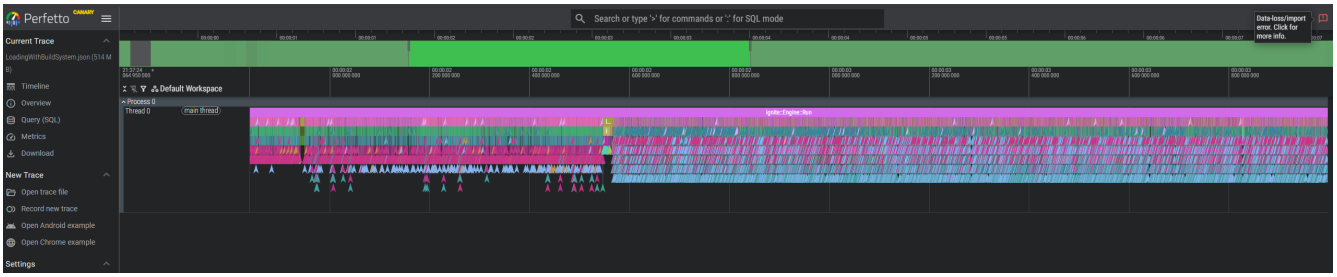

Figure 1: Screenshot showing an example of the timeline visualization.

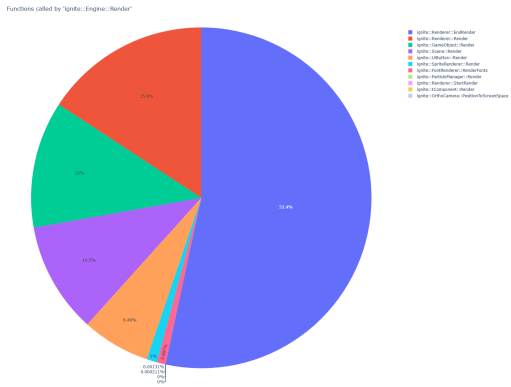

Figure 2: Screenshot showing an example of the pie-chart visualization.

1.2 Memory Profiler

The memory profiler provides runtime statistics and an overview of memory usage to the developer. It can be used to visualise memory fragmentation, identify the sources of allocations, and examine the current state of memory.

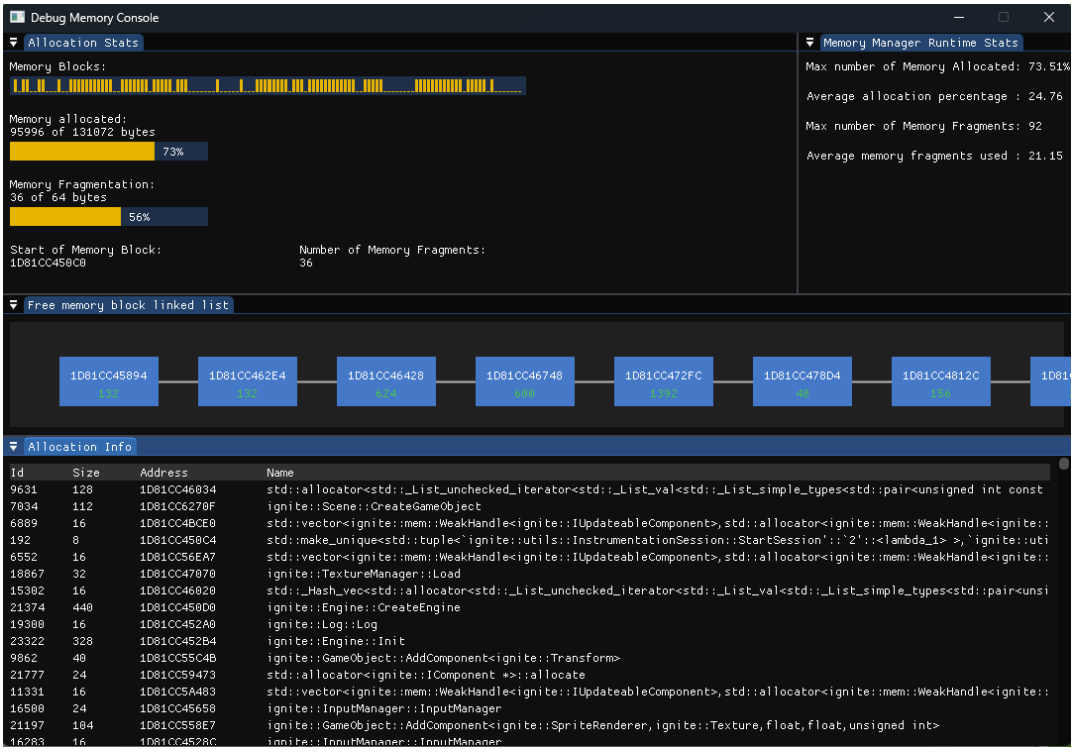

Figure 3: Screenshot showing example of the memory profiler.

2 Base profiling

2.1 Timeline Visualizer

1. Within the `Render` function, multiple calls to the same function are observed. This occurs because the render functions of all game objects are invoked, even for objects that don't override the method.

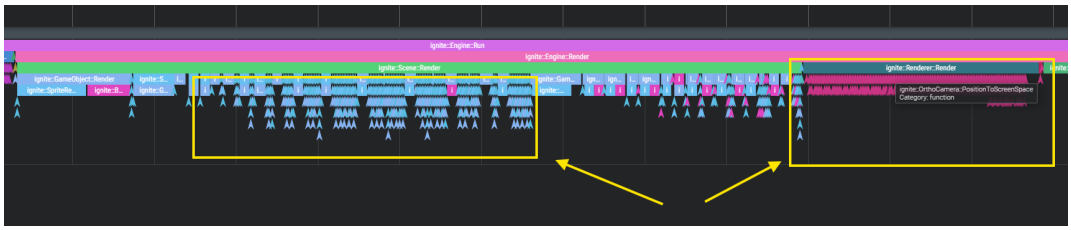

Figure 4: Timeline visualisation showing hotspots in render function.

2. Similarly, the `Update` function exhibits the same behaviour, as indicated by the hotspots. Additionally, a potential bottleneck seems to occur during font loading.

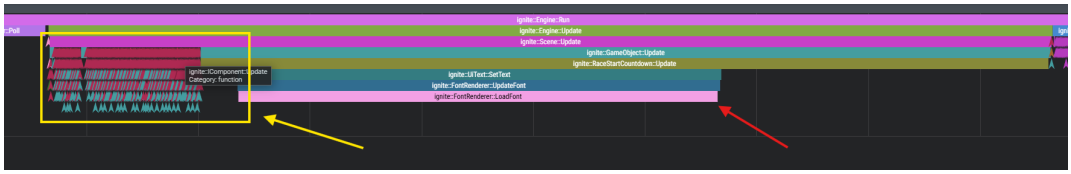

Figure 5: Timeline visualisation showing hotspots in `Update` function (yellow arrow) and loading fonts causing a delay (red arrow).

3. Zooming out shows that loading fonts is a regular time-loss in the `Update` function.

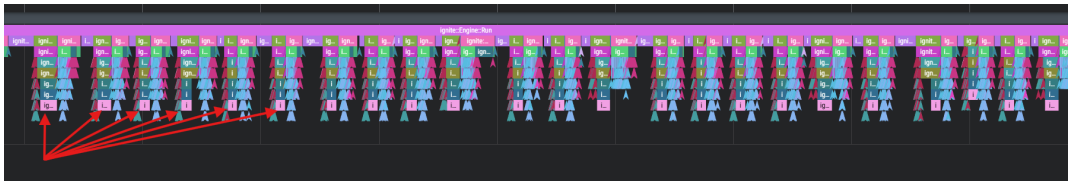

Figure 6: Timeline visualisation showing regular time-loss due to updating fonts in update function.

4. Parsing levels take a significant amount of time, introducing many allocations and fragmentations due to strings.

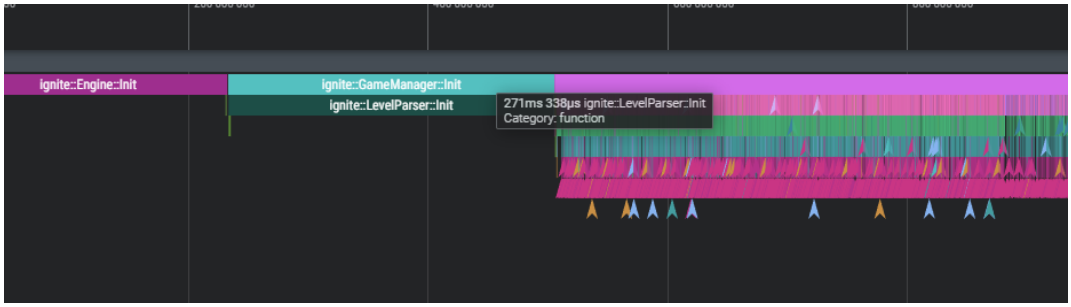

Figure 7: Timeline visualisation showing how expensive the level parsing is.

2.2 Pie-Chart Visualizer

1. `Engine::Render` takes up 56.1% of `Ignite::Update`, with an average of 268.09 μs per call.

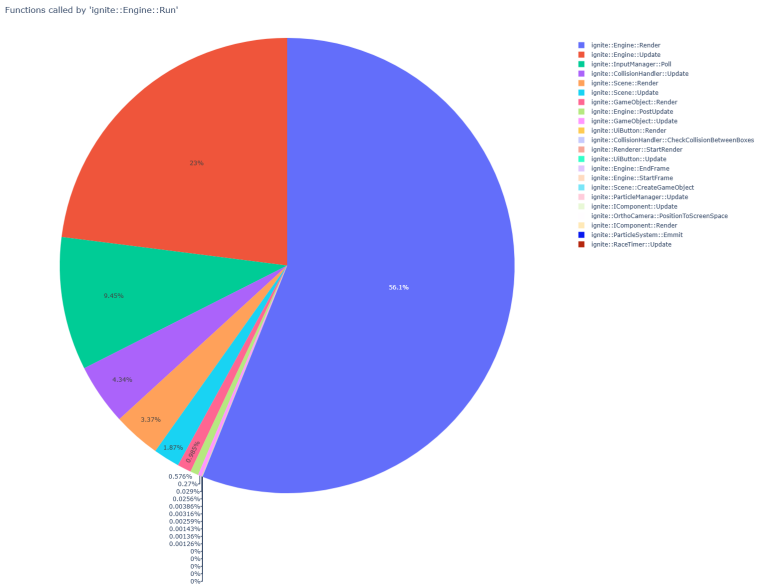

Figure 8: Plot showing time distribution of `Engine::Run`.

2. `GameObject::Update` accounts for 70.4% of `Scene::Update` time. `Engine::Update` contributes to 23% of `Engine::Run` (as seen in Figure 8), with an average execution time of 259.07 μs.

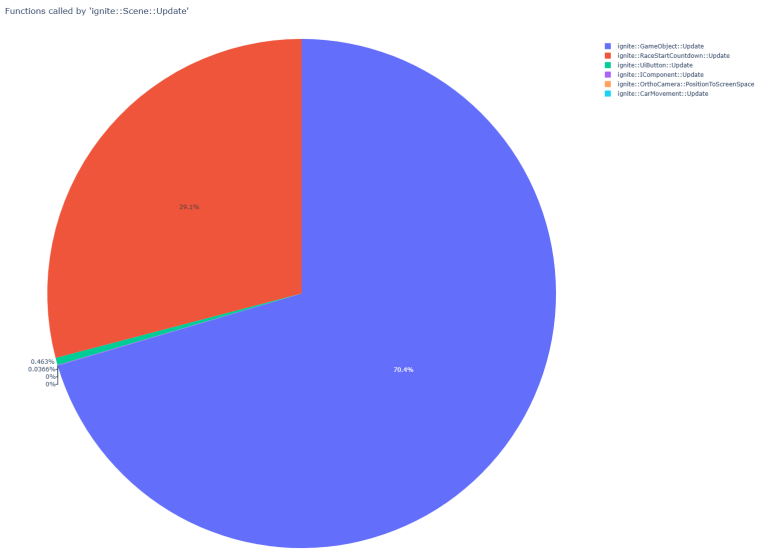

Figure 9: Plot showing time distribution of `Scene::Update`.

3. The observation from the timeline (Section 2.1.3) is supported by the fact that 98.9% of `GameObject::Update`'s execution time is to updating text.

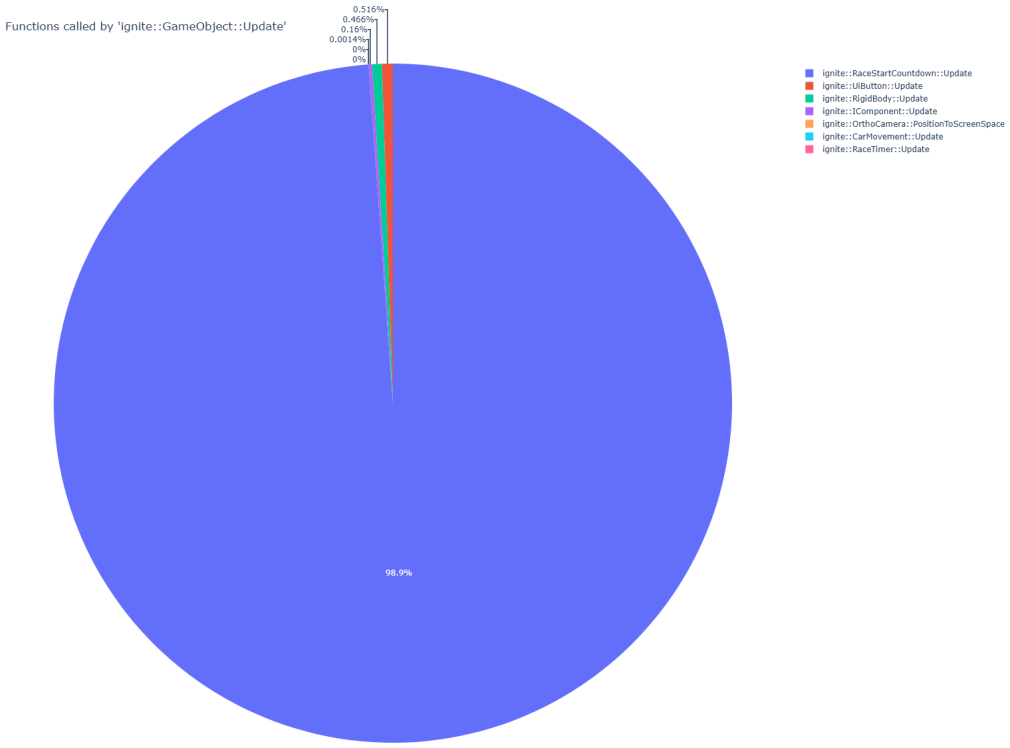

Figure 10: Plot showing the time distribution of `GameObject::Update`.

3 Optimisations

3.1 Update font only when text is changed

Updating text each frame proved to be costly. Although some overhead is inevitable, the engine's high frame rate (several thousand frames per second) makes most font updates redundant.

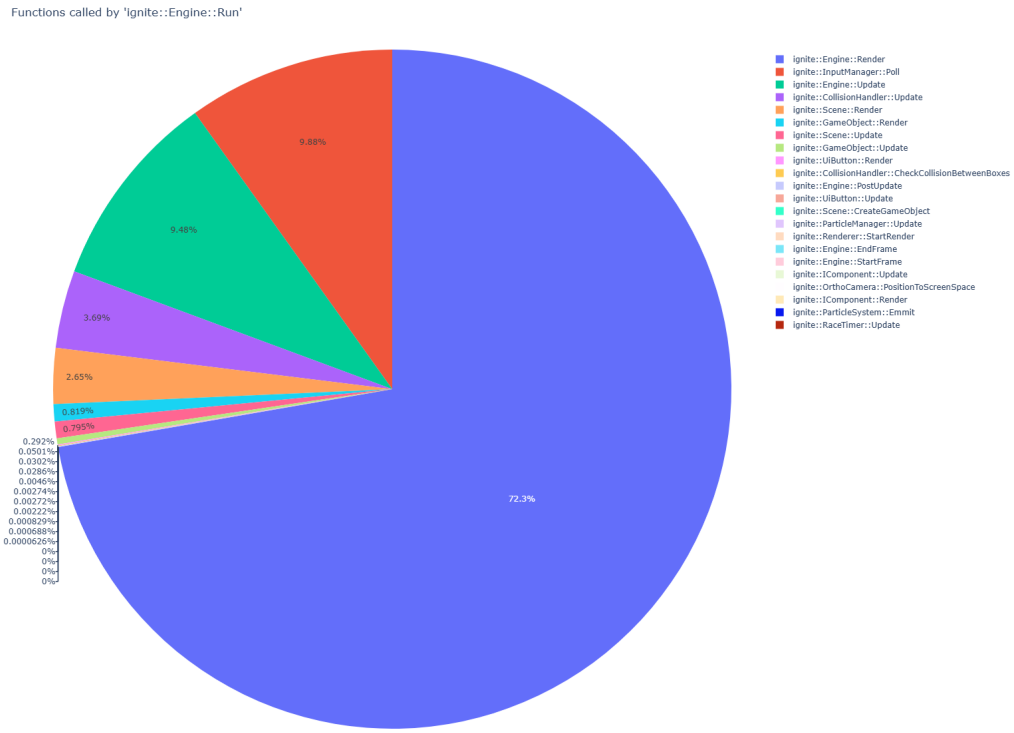

Figure 11: Plot showing time distribution in `Engine::Run`.

Update method only now takes 9.5% of `Engine::Run` with an average execution time of 95.04 μs — 2.73 times quicker [Fixes Section 2.1.2 and 2.1.3].

3.2 `IRenderable` and `IUpdateable` to prevent redundant calls

Calling `Render` and `Update` on objects without these methods caused unnecessary overhead. With the introduction of `IRenderable`, `GameObject::Render` dropped from 30.95 μs to 0.58 μs per call, becoming 53.36 times faster.

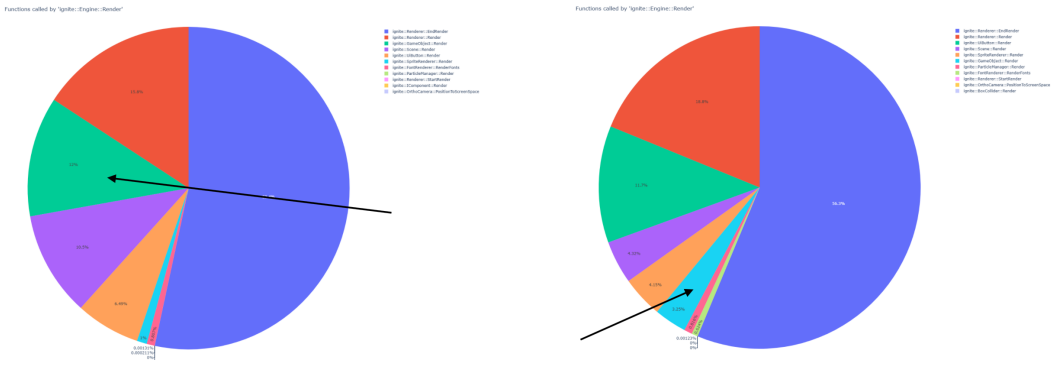

Figure 12: Plots showing difference of `GameObject::Render` before (left) vs. after (right).

`GameObject::Update` also noticed a significant reduction, decreasing from 70.4% of `Scene::Update`, to 12.5% overall [Fixes Section 2.1.1, 2.1.2 and 2.2.1].

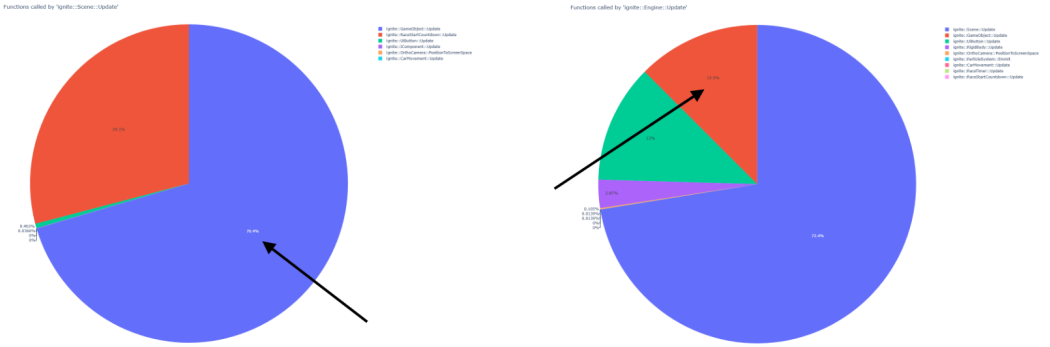

Figure 13: Plots showing difference of `GameObject::Update` before (left) vs. after (right).

3.3 Constant Memory Allocation for Render Commands

Using the memory profiler, it was seen that memory was being allocated frequently which was caused by vector allocations happening each frame (this can be seen with the gifs *RenderAllocationsBefore.gif* and *RenderAllocationsAfter.gif* in submission folder).

3.4 Level loading — Build system to prevent runtime allocations

Changing the levels to be static in release (using built level creation generated from python script) reduces `LevelParser::Init` from initial 271 ms to approximately 2 ms — over 135 times faster [Fixes Section 2.1.4].

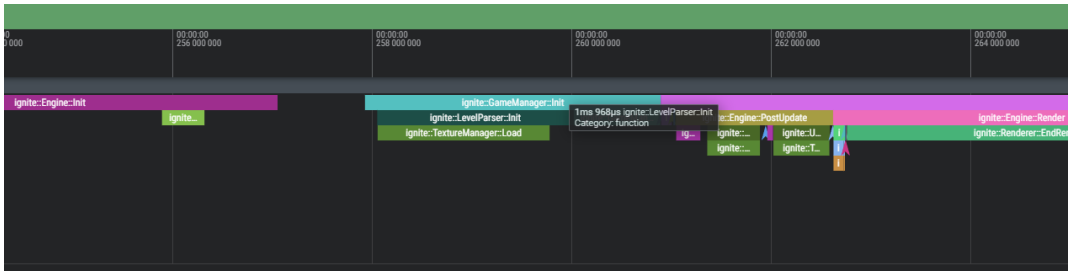

Figure 14: Timeline visualisation showing improved loading time by changing to static build comparing to runtime loading and parsing.

Word count: 422

END