Optimizations and Performance Improvements

# Overview

During the development period I had noticed a steady decrease in performance with the addition of additional features. To counter this I begin profiling my code base. I aimed to implement two types of profiling techniques, which should work on both the Caanoo device and on the PC development environment. The techniques where timer based profiling using a high frequency timer and sampling based profiling. My code itself has two main pipelines. Both the render pipeline and the logic pipeline could contain potentially large bottlenecks causing loss in performance.

It is also worth noting that prior to any profiling attempts, frame rate on the PC varied between 100 – 150 frames per seconds (FPS) and 8 – 12 FPS on the Caanoo device.

# High Frequency Timer Profiling

Figure : Render and Update Time

The high frequency timer is used by timing how long a given piece of code takes to execute. I begin by adding two timers; one that queried the time it took to update the game play logic and the other that queried how long it took to render the scene to the screen. The results where then dumped into an automatically generated Html line graph [Figure 1].

As can be seen from the produced line graph the bottle neck currently lied within the rendering of the scene and not the updating of the logic.

Although there where optimizations to be made within the updating of logic, since it would still take the same amount of time to render the scene there was (at this point) little gain from optimizing the logic, thus I began with graphical optimizations.

# Optimizing Entity Rendering

Using more profiling timers I was able to pinpoint the bottle neck of the rendering, which was located in the entity rendering. Entities have two potential code paths, one for animated entities (which also supports entity rotation) and non-animated entities. The rendering for non-animated entities was as simple as possible, however calculating if an entity was animated was done within the render method, so this was moved to a pre-calculated variable (removing an additional addition every render).

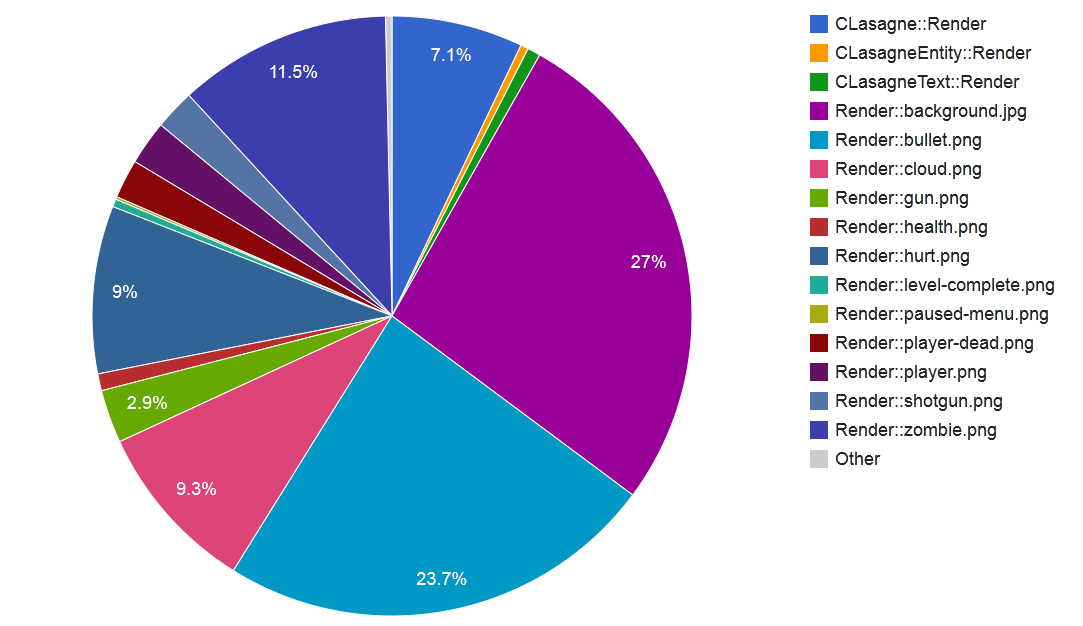
However the calculations involved for the animated entities could be optimized to a larger degree. To animate and rotated the entity I had to offset into the animated sprite sheet to create a new SDL\_Surface just for the current frame of the animation, then take the new frame surface and rotate it by a given amount, finally the rotated frame surface can be blitted to the screen. This technique was being performed every render for every entity as well as requiring the creation and deletion of two SDL surfaces.

To resolved this expensive rendering technique, I started by changing the creation of the frame surface from being created every render too only being created if the rotation and or current animation frame has changed [Figure 6]. Meaning in the best case scenario the creation will only be done 20 times a second. Furthermore optimizations to edge cases such as the rotation of the entity being zero where made.

# Sampling Profiling

Profiling via sampling is the collection of the current state of an application via sampling said application. To do this I implemented a separate thread within my program, from which I would sample my main application. I have previous experience of implementing a sampling profiler in Windows via the Win32 API. So I quickly implemented this by getting the current execution address of the application then resolving that into the current method the address represents. Using this gave little to know help, once again showing that the ‘hotspot’ of the code resided within entity rendering.

I decided to change my sampler due to two reasons. Firstly my implementation will only work on windows, secondly the current sampling results gave no benefits over my previous profiling attempts. Instead of using the current processing address, I would instead store useful information within a string and simply query that string at a given sampling rate. The string stores the current method within my render as well as the name of the entity that is currently being rendered, allowing me to see which entities are having the biggest impact on performance. This also allows my sampling profiler to work across the two platforms. The sampled data collected was then output to an Html pie chart at runtime [Figure 2].

These sampling results instantly show that something is wrong. A large proportion of the render time was being spent on the rendering of bullets. However the bullet sprites are small non animated entities, indicating an error in my code.

My entities store a visibility and depth state, which is checked before the blitting of the entity. However, I was using a single std::vector for all my entities, which meant they all had to be checked for all levels of depths and all visibility states (this being an n2 algorithm).

To fix this, I changed from using a single list, too two lists of arrays of depth, 10 arrays of vectors for the visible entities and 10 arrays of vectors for the hidden entities. With each element of the array represents a depth level. This increase cache hits as well as only accessing elements within a vector (which on the Caanoo is extraordinarily slow) that are required.

Figure 2: Render Sampling Results

The sampling also showed that the rendering of overlays which are shown little, had a big impact on render time. Code-wise I believed the render algorithm was robust enough to not have this kind of overhead, leading me to believe there was a problem with the image asset itself. Due to this belief I went and converted all my images to the same format as the screen. This also improved the render time of all non-transparent textures, but had a marginal improvement on transparent textures (however I did not want to sacrifice alpha blending for colour keying).

# Frame Rate Consistency

Whilst playing through the game I noticed that over-time the frame rate would increase, implying some kind of leak within the code. Firstly I checked for actual memory leaks, of which there where none. Followed by checking for resource leaks from reading/writing to files, again nothing was leaking.

To help with debugging and profiling I decided to implement more on-screen real-time debug output [Figure 3]. The leak was caused by dead enemies, whom although they were dead, where still being rendered invisibly. Removing the dead entities from the renderer on death fixed this frame rate issue.

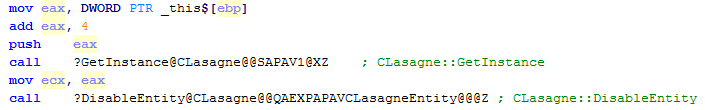
Figure 3: Real-Time Debug Performance Graphs

# Assembly Optimization

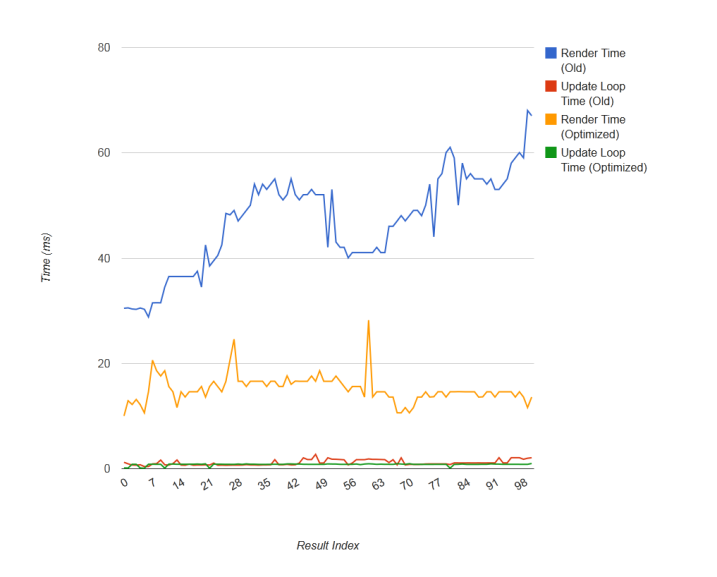
Whilst reading through the assembly code generated by my project, I noticed a large section of assembly whilst accessing methods within my engines singleton class. Some simple methods where causing twenty lines or so of assembly.

This was caused by my static instance of the singleton class being created within the get method of the singleton. This meant the compiler could not know what value the static instance was until the method was called, meaning the get method could not be inline, resulting in any method within the class being unable to inline.

Simply changing the instance declaration to be a member of the class meant the amount of assembly generated was quartered. As I access the render engine singleton frequently this has a positive effect on the performance.

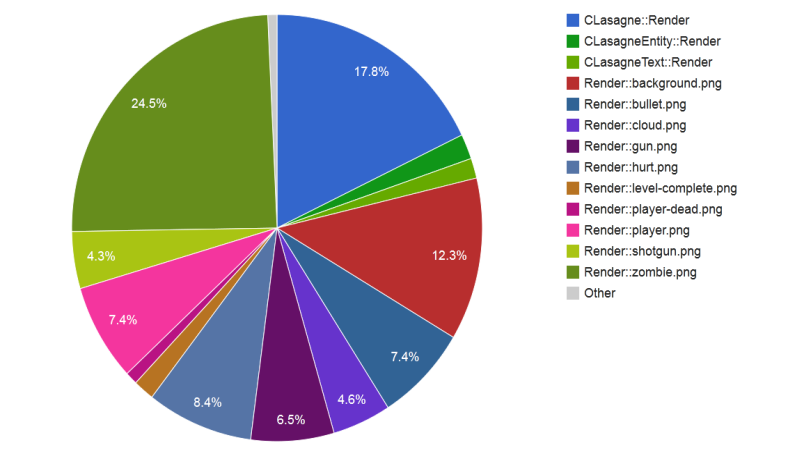


# Conclusion

With the above profiling and optimizations a steady frame rate of 800 – 1200 FPS was attainable on the PC and 40 – 60 FPS on the Caanoo. I was surprised at the amount of performance gained by simply not doing a few things, however because the few extra calls were made for multiple instances of said object, on lower end hardware such as the Caanoo the overhead was easily noticeable. The optimizations that I made also show that although the standard template libraries are useful, when it comes to writing a high performance application such as a game, they simply do not have the required performance and on hind sight wish I had written my own list which could have been optimized for my game.

Comparing the original performance graphs [Figure 1] with graphs generated after implementing the discussed optimizations can be seen in Figure 4.

Figure 4: Comparing the non-optimized and optimized results

As can be seen the optimized render is much more stable and is more than twice the speed for the length of execution. Along with this the update loop is also much more stable in the optimized build.

It can also be seen that the distribution of render time has moved away the rendering of the bullets [Figure 5]. Rendering now spends more time in the actual render loop (which is where vector element access is performed).

Figure 5: Optimized distribution of render times

# Notable Code Differences

Figure 6: This change caches rotated animated sprite surfaces rather than calculating the frame every draw