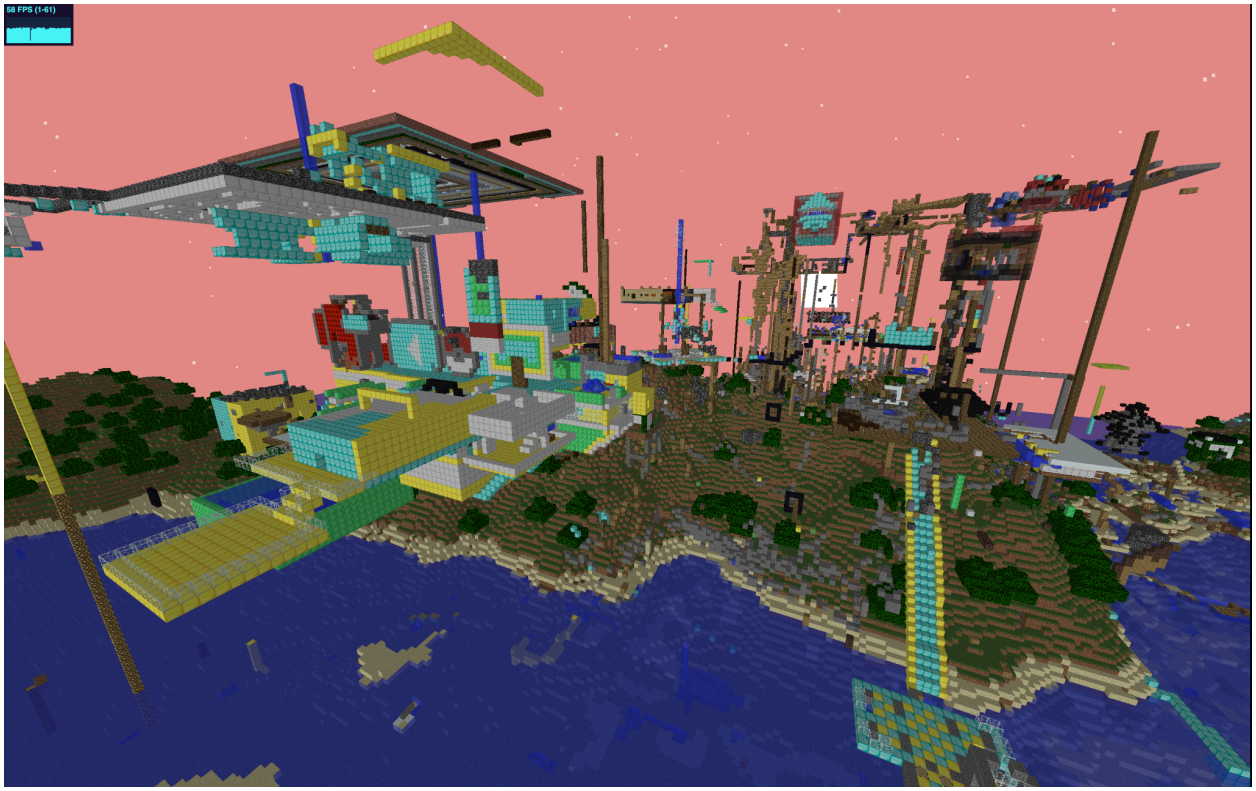
Optimizing BlockCraft for the Web Browser



Date: Jan 13, 2021 | Editor: Ibrahim Ahmed

Where was I?

It's been over a year since my last blog post on BlockCraft, as the year 2021 was probably my most important and busiest year of my life so far, with uni applications and my first term in university. Now that I finally had some time on my hands over the winter break, I decided to delve deeper into the development of BlockCraft. In case you haven't read my previous blog, BlockCraft is a Minecraft clone that's designed to run on the web browser. I was able to implement most of the basic features from Minecraft, including procedural terrain generation, multiplayer combat, and a simple inventory system. I was also able to tackle a couple performance issues, specifically with chunk generation.

However, the truth is, the performance issues weren't completely fixed. Playing the game still caused tiny but noticeable lag spikes whenever new chunks were loaded, even for a beefy computer I ran the game on. This was definitely off putting as each time the player camera would stutter and skip a few frames. This was the most annoying bug in the game and I had to find a way to mitigate this issue. Looking back at my code, I already delegated most of the cpu-intensive work of the chunk loading and rendering to a couple web workers, so the problem must be residing somewhere else in the code. Thus, I pulled out the performance tab in Chrome's DevTools to try to find the issue.

The Issue

After profiling for a few seconds, the results weren't exactly crystal clear as to what was causing the lag spikes. Most of the frame rendering time was well within the frame rate of my monitor (144hz). That's when I found clusters of empty frames that didn't seem to do anything at all. Zooming in to the empty frames showed nothing happening, so why did the browser sporadically choose to not render any frames? Fortunately, Chrome DevTools has multiple options on what to profile, so this time, I decided to profile the memory usage over time. To my surprise, the results finally showed the culprit: garbage collection. The memory graph showed the exact moments when garbage was collected, the excess memory produced from the web page that is no longer used/needed. These events just so happened to line up with the empty frame times, and now the next problem arose, what was causing all the garbage to be produced? Knowing that the frame rate dropped when new chunks were being loaded, I concluded that the garbage must be from all the data required to render the chunks.

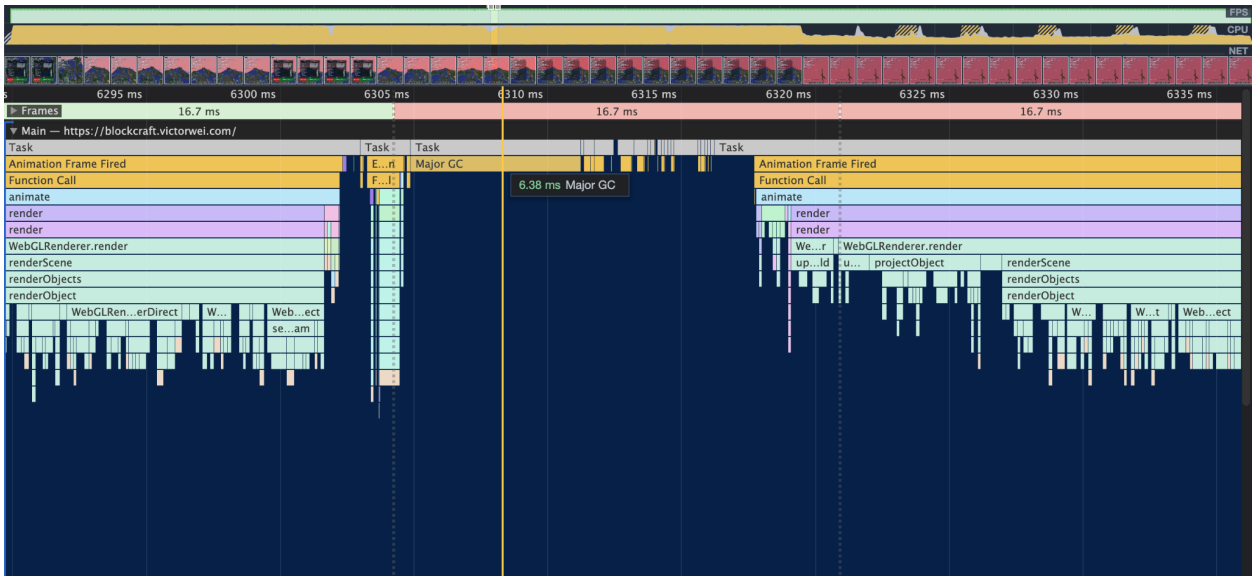


Figure 1 (above): Performance tab showing the garbage collector taking 6.38ms (a long time)

Indeed, looking at the function that creates the mesh for each chunk, it appears that several new arrays were being allocated to store the information about the positions, face indices, and normals. This means that after the chunk is rasterized to the gpu, the data is no longer needed and will then be garbage collected. Luckily, in my previous blog post I learned about a technique that is extremely useful when dealing with transferring large amounts of data that are called ArrayBuffers. Last time, a form of ArrayBuffers called SharedArrayBuffers were used to share voxel data between web workers so the information didn't need to be duplicated. This time, ArrayBuffers will be used to transfer data so that once the chunk data is produced, it can be directly taken by the GPU instead of being copied, saving both memory and cpu time.

The Solution

```
let positionBuffer = new Float32Array(new ArrayBuffer(positions.length * 4));
let normalBuffer = new Float32Array(new ArrayBuffer(normals.length * 4));
let uvBuffer = new Float32Array(new ArrayBuffer(uvs.length * 4));
let indexBuffer = new Uint16Array(new ArrayBuffer(indices.length * 2));
```

Figure 2 (above): The new method for storing geometry data for a chunk using ArrayBuffers

Implementing the fix was fairly simple, as it just took replacing the regular Javascript arrays with ArrayBuffers. Afterwards, the chunk rendering was greatly improved and the lag spikes virtually disappeared. ArrayBuffers certainly saved the day and made the game a lot more enjoyable to play. Okay, you might be wondering now if ArrayBuffers are super fast and awesome, why can't we use ArrayBuffers all the time, since it saves us memory and time? Unfortunately, the answer to that question is that ArrayBuffers can only deal with numbers, which means other data types such as strings cannot be stored. Moreover, the size of the array is fixed and cannot be changed after it is created. This limits its use cases which is why for most scenarios, regular JS arrays are perfectly fine and won't cause any major issues.

That about wraps it up for my second blog post on my developmental journey of BlockCraft. I'm planning on writing more posts related to specific topics of the game, such as texturing blocks and items, load-balancing on AWS, and crafting recipes, but considering university can be super busy at times, I'll try to post one every month. Have a great rest of your day and I'll see you in the next one!

Figure 3 (above): Here's a top down view of what people have made so far in one of the servers!