The Making of BlockCraft

Date: November 10, 2020 | Editor: Ibrahim Ahmed

I was 8 years old when I first encountered the game, Minecraft. It was like no other game I had played. The expansive open-world gameplay led to endless possibilities. It was so immersive I would spend hours playing it, losing track of time. Another 8 years later, I made my own version of Minecraft, playable on the web browser. I probably spent at least 200 hours working on this game, and slowly but surely, the game evolved to the point where I'd like to showcase it. Here's my journey on how BlockCraft was made.

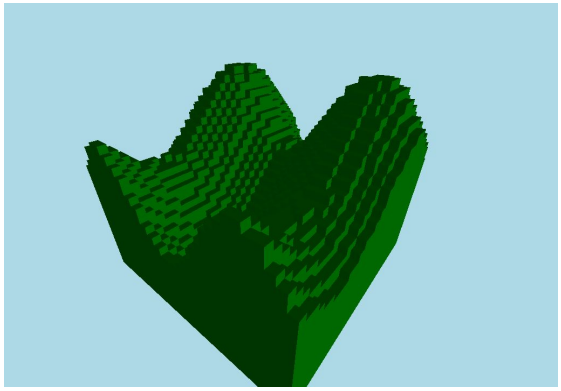
The Beginning?

It was a chilly afternoon, coming back from school on Friday, March 13th, the last day before the spring break. "It's only going to be three weeks," I thought to myself, "The COVID-19 quarantine will surely be over by then." The whole school board was entering a three-week quarantine because of the coronavirus pandemic that struck the whole world. It was not until a few days later, with COVID-19 cases continually rising, that I realized this quarantine will probably be dragged on for a much longer time. With all this free time, I might as well spend it on something I've always wanted to do for a long time, recreate Minecraft. This wasn't my first attempt at making a Minecraft clone on the web. A whole year ago, back in the first semester of my 10th grade, I first picked up on a 3D rendering library on the web called Three.js. My first attempt was very primitive and was riddled with bugs and glitches, as I was still learning the library. This time, however, I was going to make it right. No glitches, no bugs, no obstacles that will stop me.

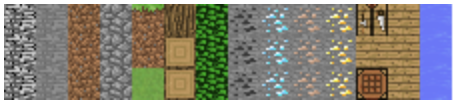
The Beginning.

First things first, I needed to render some blocks. It first seemed pretty easy as Three.js provided straightforward tools to create cube geometry and turn it into a mesh that can be rendered on the screen. Since I was making a “voxel” game, which is composed primarily of blocks, I organized the blocks using chunks, which is basically a giant block consisting of smaller blocks. However, my first attempt in implementing this was very naive, as I added each block as a separate entity. I soon ran into performance issues with this and the game became essentially unplayable. Let me try to explain. In Minecraft, chunks are made up of 16 x 16 x 256 blocks, which amounts to 65,536 per chunk. An average player uses a render radius of 10 chunks, which is about 441 chunks. In total, we have reached over 28 million blocks! I knew I had to reduce this number significantly, so I turned to the web for help.

The problem that existed was that many blocks are being rendered even if they are completely covered by other blocks. When blocks can’t be seen, blocks don’t need to be rendered! This resource proved to be extremely useful as it explained how to reduce the number of blocks rendered, or more specifically, the block faces. Basically, before rendering the chunk, each block face is checked to see if their neighbouring block is empty. If it is we can place a face there, but if not, the face is covered and thus does not need to be rendered. Immediately, I saw a massive performance increase as the number of blocks/block faces rendered was greatly reduced.



The game wouldn’t look like Minecraft if there weren’t any textures. Adding textures went fairly smoothly and I used a method many game developers use called UV mapping. In short, it’s the process of projecting a 2D image to a 3D model, which in this case, is a block. At first, I imported each texture separately, but this proved to slow while loading the game. To make lives easier, I used a texture atlas, which is a combination of all the textures that each block uses, stitched into one image. This allows the browser to quickly load the textures and easily access a specified texture. Here’s a texture atlas I used early on:

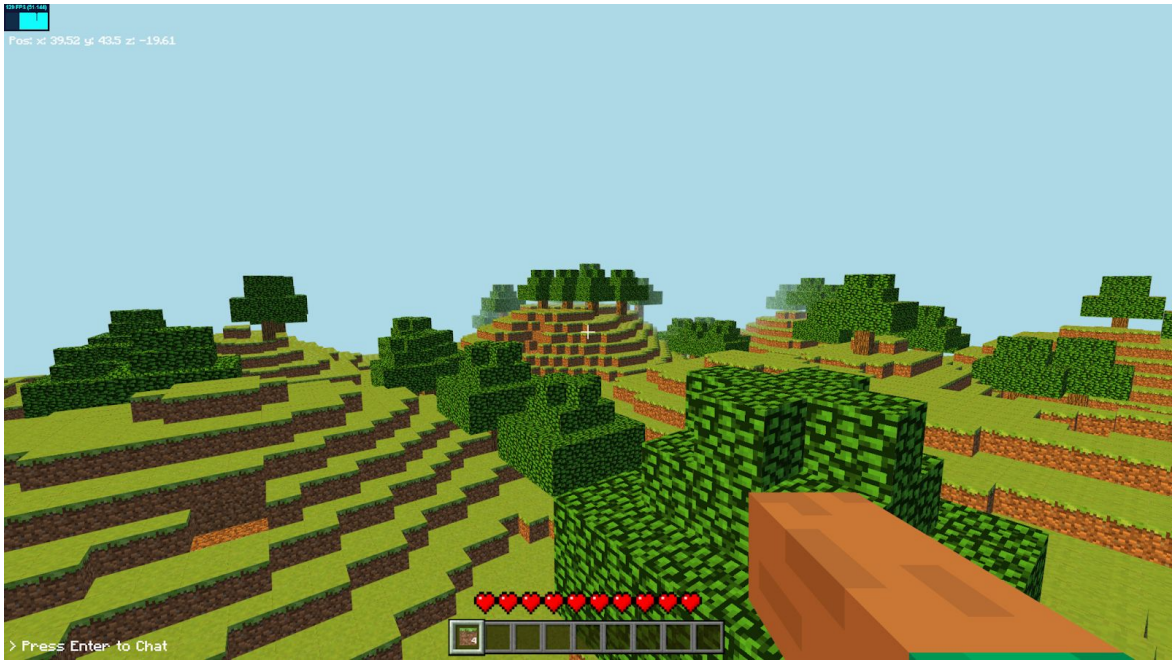


However, this later proved to be a large hassle as I had to manually update the texture atlas using GIMP each time I wanted to add a new block. I moved to a more versatile option by stitching the images together when loading the game, so that I could easily import whichever textures I wanted.

Next on the bucket list was to support multiplayer. I was fairly familiar with implementing multiplayer as I have made many multiplayer games on the browser before. The library I used is called Socket.io and uses an event-based communication method between the server and client (browser). The main issue I ran into was to model the main character, Steve. I wasn't able to import the Steve skin texture atlas as I wasn't very experienced with mapping a texture atlas to a more complex model. Consequently, I had to painstakingly redraw each part of the Steve skin using a pixel art editor and slowly build the model from head to toe. However, after days of remaking Steve, I was fairly satisfied with the result.

Here's how the game looked 7 days into the making of BlockCraft.

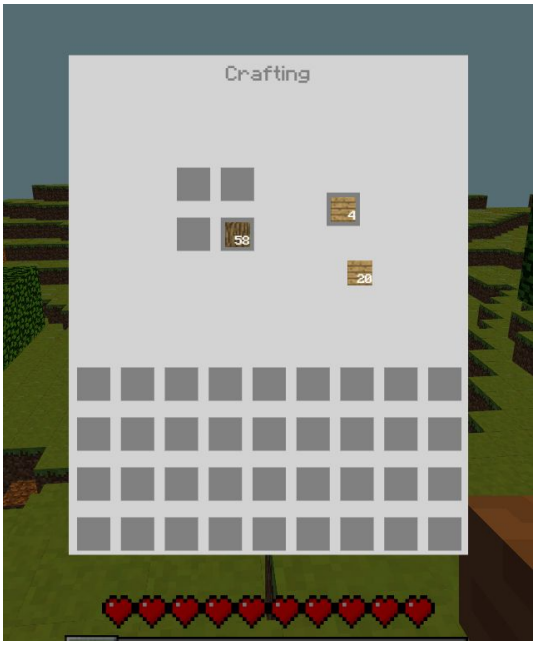
The game was starting to come alive and you could definitely see similarities to the game I wanted to recreate.



Taking Inventory

Over the next couple of days, I spent my time adding the little things that Minecraft had. Here are some of the things I'm talking about: Name tags, player health + regeneration, chat messaging, blocking breaking animation, trees, caves, ore generation, flying, you name it.

Most importantly, I added an inventory system. In between my time developing this game, I looked at many other attempts to recreate Minecraft. Many of them had the gist of Minecraft, with basic terrain generation, block breaking, etc. However, I found very few projects that took it above and beyond to implement multiplayer PvP and even crafting. I wasn't going to stop there, as I only fulfilled the "Mine" of Minecraft but not the "craft" of Minecraft.



I had two options I could take to create the inventory GUI. I could simply just use plain HTML and add interactivity with jQuery, or I could use the HTML5 canvas element which supports 2D drawing and animation. I decided to take the latter since I believed it could offer more versatility than plain HTML. After a week of tinkering, I whipped up a basic looking GUI that includes a 2 x 2 crafting table as well as a manipulatable inventory. You could move items, right-click to split item counts in half, and double click to collect the same items. At last, you could craft a crafting table!

Beaconator

A month into making the game, you could say the game looks similar to Minecraft Classic, one of Minecraft's earliest versions. The key difference is that BlockCraft, my iteration of Minecraft Classic, used WebGL on the browser, as opposed to OpenGL which is much faster and more powerful. The issue with running games on the browser is that Javascript code is run synchronously or single-threaded. In simpler terms, instructions on the browser are run one at a time. This meant that while a block of code is being executed, no other code will currently be executed. Tasks that take a long time, especially chunk loading, will often create large lag spikes that make the game hard to play. It was so bad, I wasn't able to run the game smoothly even on my beefy computer. As a side note, running BlockCraft on the worst computers like Chromebooks was inevitably slow, so my priorities were not to optimize those. I resorted to desperate solutions such as reducing the chunk size and decreasing the render distance, but all of these were only band-aid solutions. I tried looking into multi-threading with Web workers but I never got it to work. Because of the performance issue, I was so put off that I decided to quit developing BlockCraft.

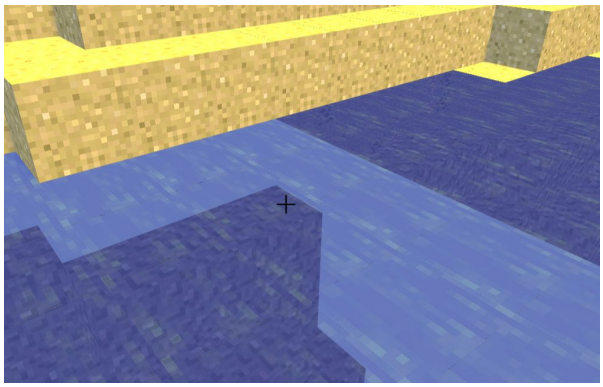
A few weeks later, I decided to take another crack at implementing multi-threading into BlockCraft. So what exactly is multi-threading? Basically, a computer contains many threads that can each run simultaneously or asynchronously. By moving tasks that take a long time such as chunk loading to another thread, we can keep the thread that is rendering the game running smoothly. Web workers is an API (application programming interface) that allows multi-threading to work on browsers. With gained experience working with web workers, I was able to off-load the chunk geometry calculations to separate workers, making the game run much more smoothly while chunks were loading. All in all, I was happy with resuscitating BlockCraft and I was able to continue to work on the game without any further performance issues.

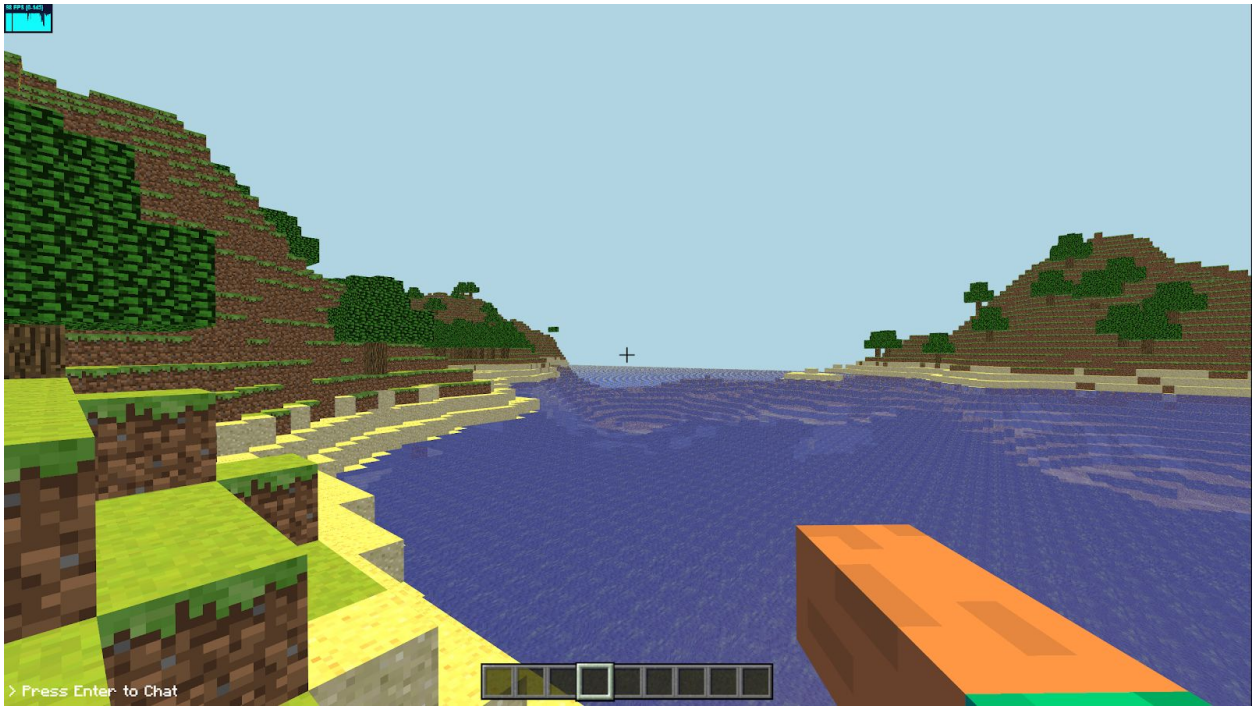
We Need to Go Deeper

No, I have not added a Nether and I'm sorry if the heading is misleading. I'm talking about adding water to the game, which acted like a whole nother dimension for me. The reason being is that water, unlike all the other blocks I have added to the game so far, is translucent. So far all the code I have written so far was to support opaque textures, but I have not considered transparent textures until now. As I said earlier in this post, block faces are checked if neighbouring blocks exist. However, with water, neighbouring blocks should also be rendered, since you can see through water. This seemed trivial to implement since I just had to check if faces were beside water, and if so, show them. However, this was not the case as I realized there were giant walls of water separating the chunks.

Since the chunks facing the outside haven't loaded neighbouring chunks yet, those faces are still rendered, creating the unintended "wall of water" effect. To mitigate this, for transparent blocks such as water, glass, and ice, faces would only be rendered if it is adjacent to an "air" block.

Though I solved the major issue, many bugs still exist with transparent blocks. At certain camera angles, water blocks create an X-ray effect are you are able to see through the world. Unfortunately, these issues are due to the capabilities of the rendering engine itself, having to deal with render order between opaque and transparent textures. However, I'm am still very satisfied with the result of adding water.





A screenshot showcasing the addition of water blending in with the beach and its surroundings

Time to Strike!

The next thing I decided to implement was item models to incorporate into PvP combat. In Minecraft, items such as swords and diamonds are extruded from a 2D image to create the illusion that the item is 3D as shown in the picture to the right. However, Three.js doesn't support direct extrusion of a pixelated 2D image to a 3D mesh, and required images to be in an SVG format. I tried for hours on end to make this work but to no avail. I ended up using sprites instead and it created a similar effect.

I wanted to make the PvP (player vs player) combat as realistic as I could. I personally enjoyed Minecraft 1.8 PvP where there is no set attack cooldown rather than the combat *with* a cooldown from Minecraft 1.9+. I decided to take the former approach when adding PvP to BlockCraft and added extra features such as sword blocking (to reduce knockback and damage) and shooting with a bow.

The End?

It's been more than half a year since I started working on BlockCraft. From where it started to where it has evolved to this day, my journey had moments of satisfaction but also riddled with problems and obstacles. Overall, this was quite the learning experience as it was probably the hardest and longest project I have worked on to date. Though there are still plans to improve the game and fix the never-ending list of bugs, I just want to say a big thank you to everyone who has play-tested and supported me through this journey. "It's all about the journey, not the destination" - Ralph Waldo Emerson