Minecraft at Home

Tim Gubskiy & Andy Nguyen

Abstract

We created a simplified version of the popular game Minecraft, entirely using ThreeJS, employing many of the features that Minecraft is known for. We implemented procedural generation of terrain using Perlin noise, dynamic loading, and unloading of chunks to improve performance, mesh instancing to decrease render calls, physics for player and block interactions, and the ability to add and remove blocks of various textures. We also created a simple UI that allows the player to cycle through the materials they have available and a crosshair for placing blocks.

Features and Implementations

Project Structure

To organize our code we used an Object-oriented approach similar to that used in the example code, but with a few extra classes that added our desired functionality. Our classes consisted of GameScene, Player, and Map, all of which were called and used by each other and app.js.

The GameScene class was responsible for creating our scene, and generating new terrain while loading and unloading parts of the map. The player class controlled player movement and handled collisions, and the Map class was used to create an efficient way to access all the blocks in our world by splitting the world into 16x16 block chunks and storing chunk data in a hashmap.

This project structure made our code much more readable and made coding much easier and also provides a great scaffolding for future improvements to be made to the game.

Placing and destroying blocks

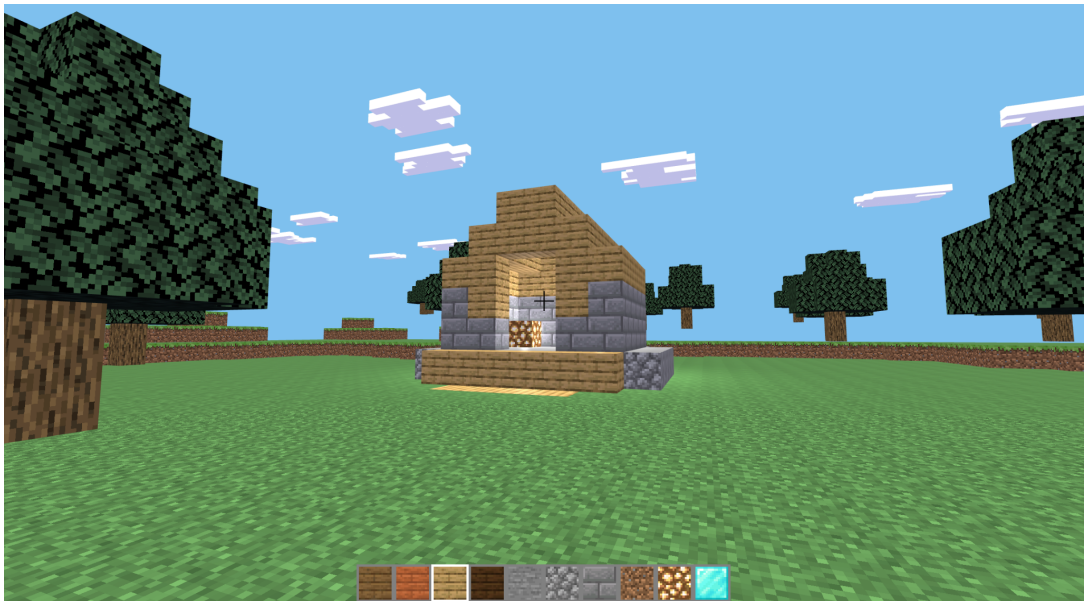
The core basis of our game is the ability to place and destroy blocks in our scene. This simple mechanic allows for an insane amount of creativity while being bound to a very small set of concrete rules.

The entirety of our scene is composed of axis-aligned, 1x1x1 cubes, that have integer positions so that none of the blocks can overlap, and all have a unique position. We use the Map class that we created to store the positions of all of the blocks in our scene, using a HashMap that splits our world into 16x16 block chunks. This is important because although our scene contains all of the meshes necessary for rendering, our physics engine and dynamic chunk-loading system require the block information to be stored elsewhere.

Placing and removing the box takes advantage of the raycasting functionality built into ThreeJS. On every animation frame, we cast a ray out of the center of the camera and detect what blocks it intersects with. ThreeJS conveniently sorts these intersections by distance automatically, so if an intersection is detected we can look at just the first intersection to

determine the position we are looking at. To calculate the position of where a block should be placed we translate the point of intersection towards the camera position by a small amount and then floor the coordinates to ensure that they are aligned with our grid. A temporary box reticle will show the player where a block will be placed when they hover their cursor over the world, and this box gets destroyed and recreated every frame at the new location.

When the player clicks their mouse, we check if the event was a left or right click, and remove or add a block respectfully. To add a block we change the texture of the reticle box, to that of the active slot on the player's hotbar, and prevent the box from being deleted on the next frame of the game. The block position is also added to the Map object using the addBlock function, allowing it to be used to for collision detection. When removing a block we do the opposite and delete the object that the ray intersected with, and remove the block at that position from the Map object.



Block Placing

To determine which block the player wants to place we create a hotbar UI element with 10 block textures. The hotbar is an overlay over the ThreeJS canvas, using HTML images and CSS styling to adjust the positioning as well as giving each item a frame, and the selected item an active frame with a different color. The scroll wheel and number keys can both be used to change the currently selected item, determined by the variable activeItem in the Player class.

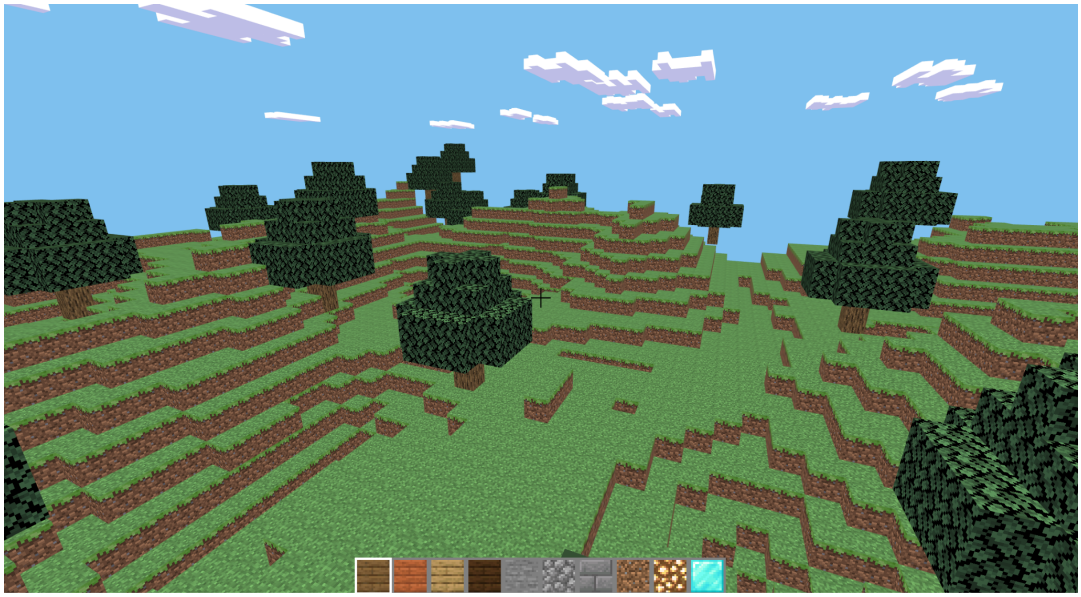
In the player class, we store an array called hotbar, which contains all of the items on the hotbar as objects that store the HTML element, the item texture, and an attribute that determines if the item is glowing or not. This final feature was added so that we could add Glowstone to our game! If this attribute is enabled when adding the block it will also add a point light to the scene that lives inside the block!

Procedural Terrain Generation

One of the staple features of the game Minecraft is the procedurally generated world, which includes many different terrain features like mountains, valleys, and caves, as well as

procedurally generated entities like trees and mobs. For our game, we wanted to tackle the terrain generation aspect, as well as randomly instantiating entities like trees!

To start we tried to make procedurally generated flat terrain, adding a 1 block layer of ground to every chunk within our render distance. This is where we ran into our first problem, we were initially using BoxGeometries in our scene for simplicity, but terrain generation vastly increased the number of blocks we had tanking our framerate as we would move outside of the starting chunk. To remedy this we swapped to BoxBufferGeometries, which did increase our frame rate back to an acceptable level.



Terrain Generation

To generate complicated and natural-looking terrain we turned to a concept we learned about during our raytracer assignment, Perlin noise. Perlin noise is something that is used widely in game design for terrain generation and is actually what is used by the real Minecraft to procedurally generate their world. For ease of implementation, we decided to use an external Perlin noise library for JS that allowed us to define the size of the grid we wanted and would return an array of heights at the coordinates in the grid. To translate this into our blocky terrain we used two Perlin heat maps, one that defined chunk biomes, and another that defined the actual terrain elevation. To extract elevation from this we multiplied the two heat maps together, using the world coordinates for one, and the chunk coordinates for the other, and a third factor which was the distance from the origin up to a certain maximum. This gave us some cool biomes, creating small local regions that are very flat and also large mountain ranges that look awesome.

While generating our terrain for every block added we also will randomly, with a probability of 0.005, generate an oak tree at that position! The tree data is stored in a JSON file that describes the position and texture of every block in the tree.

Optimizations

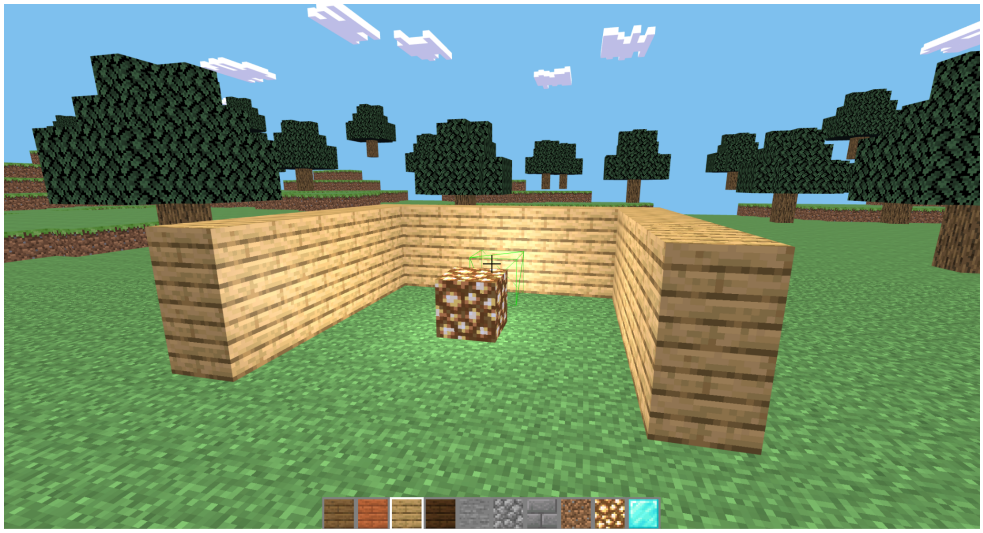
Predictably adding elevation changes once again exploded the number of meshes that we had present in our scene so we were in dire need of some sort of optimization. After some research we came across MeshInstancing, a way to reduce the number of render calls to the GPU

as long as all of the Meshes have the same textures. Conveniently the terrain, which in our game state consists of only grass blocks, is perfect for this optimization. As such we converted all of our terrain generation to utilize MeshInstancing, a huge optimization for our performance, bumping our frame rate up once again. However, some other challenges came with using MeshInstancing. For one we would have to change the way we delete blocks from our world, since we would delete entire chunks at a time if we delete the object that our ray intersects with. Our fix for this, which can probably be improved with more time, was to relocate the single instance of the mesh that our ray intersected with to a location under the map so that it cannot be seen.

Our second optimization was to dynamically load and unload the terrain as it came into view. If you play our game you can see this happening since our render distance is set to 2 chunks. As our player moves around in the world chunks that come into our render distance will be generated and added to our scene, and the MeshInstance object will be added to the Map class so that we can access it later. If the chunk has already been generated, instead of regenerating it, we will simply add the MeshInstance for that chunk back to the scene. When a chunk gets out of range, we will delete the MeshInstance from the scene, greatly improving our performance when we move outside of the starting chunk, allowing our player to explore the entirety of our infinite world.

Lighting

We wanted the scene to have some form of dynamic lighting. We added ambient lighting to the world which brightened up the world, allowing the player to see the terrain clearly and mimic the same ambient lighting from Minecraft. We then added a hemisphere light to brighten the scene even more. To allow the player to interact with the light aside from simply viewing the bright scene, we added the glowstone block from Minecraft which adds point lights to the scene at the block's position, allowing the player to light up the world while they build.



Glowstone Lighting

Collisions

We decided for the player to move around in the scene with somewhat realistic collisions rather than just floating through the scene, which requires detecting and handling collisions between the player and the terrain. To detect collisions with the terrain, we have the terrain stored in our Map object which is a spatial hashing object for our map. This allows us to hash the player's position and surrounding positions to find adjacent blocks quickly. After finding adjacent blocks we try to collide the player with blocks below the player and around the player to avoid clipping through the floor and clipping through blocks.

For floor collisions, we first check if the player's "feet" intersect with the blocks below by finding the corners of the player's hitbox and checking if any of the corners of the hitbox intersect the blocks. If any of the corners intersect with a block below, the player camera is adjusted back upwards which moves the hitbox and effectively keeps the player above the ground.

The player is modeled as a 2-block high model, meaning that we need to check collisions for both the upper and lower half of the player with the terrain. For both halves of the player, we simplify the problem into a 2D collision problem by checking adjacent blocks in the x and z axes. After finding adjacent blocks, we check if the player camera is inside a certain distance from any of the adjacent blocks and in which direction the player is primarily moving. We then simply move the character back the same distance it had moved forward that frame for that direction and check if the player is still colliding. If the player is still colliding, we move the character back in the other direction, at which the player is ensured to not intersect with the blocks.

We had a difficult time implementing collision detection for the player, stemming from some lack of control from ThreeJS and some issues that were not apparent. Our first approach included raycasting from the player and intersecting multiple rays with the scene. The issue we encountered with this approach was the sheer number of rays we would need to cast from the player to ensure the player collided with the scene correctly. We often found the player colliding with planes correctly but would be able to clip through corners. Adding more rays became challenging to keep track of and messy given our implementation of the Map object. Our second approach included finding the blocks adjacent to all of the sides of the player and detecting the intersection of the player hitbox with the adjacent blocks, but here we ran into issues of the camera position and player hitbox not moving in sync, thus creating jittering effects with the player and the occasional clipping through blocks.

Player movement

To interact with the world, we built a Player class that handles controlling the player and basic first-person controls. To be able to see the scene and move around in the scene, we had a camera object built into the player class which the scene would use during rendering. To look around, we bound the camera to ThreeJS' PointerLockControls class which allows us to move the camera and look around in the scene using the mouse pointer. To move the camera around, the user uses the WASD keys along with SPACE for jumping and SHIFT for faster movement.

To ensure the player could get around in the scene with blocks, adding a jumping mechanic was crucial, which led the way to ensuring a smooth jumping experience for the player. This included ensuring the player could only jump when on the ground with some degree

of tolerance. The player class included coyote time, which allows the player a brief period after leaving an edge to register a jump, making the controls less reliant on when a frame was processed vs when the physics logic processed. We also added a short tolerance for registering a jump input just before hitting the ground, being lenient in the fact that the player does not need to be frame-perfect to register a jump right before hitting the ground.

To ensure the player was moving smoothly throughout the scene without choppy movement, the player uses momentum based on inputs. This makes the player move more smoothly throughout the scene and avoids the abrupt stop when a key is unpressed.

```
// movement constants
#MAX_SPEED = 0.7;
#SHIFT_MOD = 2;
#COYOTE_TIME = 150;
#FORGIVENESS_TIME = 300;
#MOMENTUM_DECAY = 0.5;

// movement variables
#jumpHeight = 0.8;
#jumps = 1;
#momentum = new THREE.Vector3();
#lastGrounded = 0;
#lastAttemptedJump = 0;
#grounded = false;
```

All related constants and variables for player movement

Future Improvements

Inventory

An improvement to the game's current building mechanic would be to add an inventory system for the player to keep track of more items than what is available on the current player hotbar. This would allow the player to build in the world with more freedom as they would be able to access more textures to build with. This would also allow us to develop a survival mode for the player, allowing them to store and use items they collect in the world.

Shadows

While we have a form of lighting in the game, we struggled with properly enforcing shadows as seen in the original game. As of now, the player can build overarching structures but is unable to create dark rooms or shadows from these structures. To implement this, we would need a better understanding of ThreeJS' lighting system to enforce shadow casting for all blocks in the world and be able to continuously update the shadows as the world is modified.

Mobs

A fun future feature to implement would be to add some creatures to the game for the player to interact with such animals or harmful entities, such as zombies and skeletons from the original game. To do so, we would need to extend the physics collisions for the player to other entities as well, which would require a more robust physics system for the game.

Stairs and Slabs

Implementing stairs and slabs in the game would greatly improve the current building mechanics in the game as it would give the player access to more interesting geometries. This would require an improvement to the physics system as well as stairs and slabs are not whole blocks, and thus how we store them in our Map object as well as how we handle collisions with them would need to be modified from our collision system.

Conclusion

Building this game was an amazing experience for us and we got to learn a ton about ThreeJS, including raycasting, physics, terrain generation, and mesh instantiation. We're quite proud of the final result and think the game runs pretty smoothly on most hardware and the player movement and experience feels very natural and similar to the actual game.

Acknowledgments

Halfway through working on this project, we stumbled upon a YouTube series tutorial showing how to build Minecraft in ThreeJS. We did not view any of these tutorials as when we started the project we wanted to challenge ourselves to learn ThreeJS and create a fleshed-out game on our own. In viewing our code it should be obvious that there was no reference to this online tutorial.

Works Cited

We used the ThreeJS website for documentation of functions and objects from the ThreeJS library:
https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene
Instance Meshing:
https://codeburst.io/infinite-scene-with-threejs-and-instancedmesh-adc74b8efcf4
https://threejs.org/examples/?q=instanc#webgl_instancing_dynamic

Camera Controls:
https://threejs.org/examples/?q=pointerlo#misc_controls_pointerlock
https://sbcode.net/threejs/pointerlock-controls/

Perlin Noise:
https://joeiddon.github.io/projects/javascript/perlin.html
https://www.npmjs.com/package/perlin-noise

Minecraft Textures:
https://texture-packs.com/resourcepack/default-pack/