Govind Joshi (gdj84)
Albin Shrestha (as89652)
https://github.com/govindjoshi12/minecraft-peridynamics

Computer Graphics Honors Final Project: Minecraft

Goals:
Our team decided to create a lightweight Minecraft replica through the use of Perlin Noise for terrain generation. The real scope of Minecraft is extremely vast, so we made sure to keep the goals of our project limited. Our main goals were to create a chunk-based terrain generator with smooth gradients; “Minecraft-Esque” features such as biomes, block placement, etc. were relegated to the “if-time” category, and unfortunately, we weren’t able to achieve those in the time allotted. Initially, we felt quite ambitious; we not only wanted to recreate a Minecraft-Esque world using WebGL, but also a maze generation system with a proper entrance and exit, with a labyrinthian atmosphere. Although we were unable to achieve our maze generation goal, we are satisfied with our current Minecraft world based on the amount of time we worked on the project. The main algorithms we worked on involved block generation, chunk generation, block-face culling, chunk replacement, Perlin noise generation, and a day-night cycle.

Key Algorithms:
The key algorithms when it comes to voxel-based game engines involve efficiency. There are hundreds and thousands of blocks being rendered to the screen at any given time, and it is necessary to cull as many vertices as possible. Advanced techniques include frustum culling, but we did not implement that. We focused on reducing the number of vertices, and thus it is possible to selectively draw cube faces. Therefore, cube faces neighboring other non-transparent faces are not rendered, including between chunk boundaries. Furthermore, it is necessary to generate new chunks around the chunk the player is currently in. In order to do this efficiently, we change the origin of the n x n chunks based on the new chunk the player is in, and only remove those chunks that are out of bounds based on this new origin, then generate the rest. The chunks array in the World Class is maintained such that the current origin chunk is at index 0, and the top-right chunk is at the final index. Perlin Noise is generated with my typescript adaptation of Steven Gustavson’s noise library. The chunk shader takes the player position as a uniform in order to create a fade effect for vertices further away from the camera. Finally, I was initially using the RenderPass class as-is to render chunks, but soon realized it doesn’t “auto-fetch” attributes, so I wrote attribute updating methods.

Component Work:

Camera: The camera controls are split into two compliments depending on which mode the player is in. If the player is in the default or survival mode then we use a locked camera where we disable the roll of the camera. Alternatively in creative mode, the camera is free to move as it was in the virtual mannequin project with a few adjustments to make it feel and run smoother.

Player movement: We tried to keep with the default controls for our player movement. WASD to move, Space to jump, Ctrl to crouch, and Shift to run. Implementing the WASD movements was fairly simple, however, where it got difficult was the jumping and crouching movements along with gravity. To get the correct player y-position we had to create a method called get getHighestSolidBlockY() which gave us the highest block at a given xz-value for the player. This allowed us to put the player one eye level above the highest solid block in that position giving the illusion of gravity.

Day/Night cycle: Implementing a Day/Night cycle ended up being trickier than we had an original thought. Having the background color slowly go from light blue to black in a smooth gradient was very difficult. We were mainly confused on how to reset the value so that it could not jump from black to light blue but rather black to dark blue to light blue. We eventually concluded using the delta time and scaling our gradient-based on that value.

Crosshair: Adding a player crosshair was the main goal of ours so it provided an fps feel to our project and it showed us where the center of the screen was at all times which became very helpful as we generated more and more blocks.

Switching modes: We have two mode switches in our code. “C” for survival/creative camera modes and “P” for pointer lock mode. Both of these modes were added as both a debugging tool as well as an added feature. Pointer lock mode feels more authentic to the Minecraft experience. And being able to effortlessly switch camera modes allows for some good exploration.

Govind Joshi (gdj84)
Albin Shrestha (as89652)
https://github.com/govindjoshi12/minecraft-peridynamics

Our Implementation:

Implementation and Bugs:

Our implementation is based on the World Class acting as a "Chunk Manager" for a list of chunks currently in the world, with each chunk being a 16x32x16 sized collection of blocks. Each class has a logical and actual representation; when the world is initially created, it calls the world's update method, which creates chunks around the player's initial position, but only the "logical" representation; Each chunk's list of blocks is populated through WorldGen.generateChunk() which procedurally generates blocks with smooth heights. After all the logical creation is done, chunk.update() creates faces for each block based on each block's neighbors; block faces touching other non-transparent faces are not generated. Chunks are continually updated around the player through updateChunksEfficiently(), which treats the player position as the center, and calculates the origin of this new chunk coordinate system. Several utility methods in World help convert between world, chunk, and block coordinate systems and help retrieve different blocks and chunks.

Memory: There are several known limitations with the current implementation of our project. I realized far too late that each block was storing far too much data; a block doesn't need to store all of its current positions, indices, etc. I removed several unnecessary items each block was stored, but each block still stores its "Texture Indices," or the offset into the texture atlas for its textures. To improve my implementation, I would create a static database of all texture atlas offsets for each type of block, and retrieve it from a block's id when necessary.

Chunk Replacement: I initially maintained a list of renderPass objects; this list was the same length as the list of chunks and each renderer was "connected" to a corresponding chunk. However, once I implemented the more efficient algorithm for chunk replacement mentioned above, the "connections" between the renderPasses and the corresponding chunks become jumbled and confusing. I tried storing a renderPass in each chunk, but this was a terrible idea because creating a new renderPass, binding textures, and setting it up is a terrible waste of time. Ultimately, I settled on one World renderPass object, and all the positions, indices, normals, and UVs are propagated up from each block to each chunk, and finally to the world. This works decently well; moving between chunks has a reduced stutter. However, I only implemented this a few minutes ago, and I noticed an issue where previously generated chunks are not removed. I don't have time to debug the issue, but I am touting it as a feature, not a bug because looking back at far away chunks with the fade effect looks cool.

Jumping and Gravity: One major bug we were unable to resolve due to a lack of time was our issues with jumping and gravity. Rather than a robust system where the player would be able to jump up to a block, we had to take a small shortcut by looking at the position of the highest solid block at the position the player is standing on and making that the ground. With more time we would be able to fix this issue, but like most things in life, lime is the one thing we are lacking.

Sound: We spent a great deal of time trying to implement sound into our program and it creates an ambiance that Minecraft greatly relies on. Albin even had a close friend of his create a soundtrack for our project however due to a few issues we were unable to add this feature. Firstly we realized that JS does not play well with local audio files so we decided to upload the mp3 to our UTCS website so we can call it on the web, however, this led to a few more issues as we need read permissions on the file as well as the audio not playing in most cases. We were able to get audio working with another sound we found online but felt that it was not authentic to the Minecraft experience/ we did not own the audio, so we decided to leave it out.

Other "Missing" Features

Many Minecraft-Esque features are missing such as biomes, water, block placement, block-breaking, etc. I hope to work on a few of these features over the summer to make this a fully featured WebGL voxel engine.