README: Minecraft

April 10, 2023

Weiran Huang
wh8557

Sicong Che
sc67349

Collaboration Report

For this milestone, We used a pair coding format to collaborate, so both of us were involved in every part of this lab. The workload split is 50%~50%. We were able to implement all required features and 3 optional features.

Deployment

As permitted by Professor Vouga, we hosted our Minecraft (We call it "**Min**craft-js", as it is a minimum version of Minecraft) on GitHub Pages after the deadline. It is available here. You can play with it online if you wish.

Optional Features

The technical specifics of the additional optional features we incorporated are detailed at the end of this report. In summary, we implemented three optional features, totaling 20 points. For your reference and testing purposes, we have provided a list of the keys to activate these features below:

🕯 Day-Night Cycle

1. **digit key 1**: enable a 30-second day-night cycle.
2. **digit key 2**: enable a 1-minute day-night cycle.
3. **digit key 3**: enable a 2-minute day-night cycle.
4. **digit key 0**: deactivate the day-night cycle and reset the light position.

🕯 Decorative Elements

No extra operations needed. You can see the grass and rocks all around the world map.

🔔 Time-varying Perlin Noise

By **persistently pressing the 'P' key** on the keyboard, you can enable the time-varying Perlin noise functionality. While maintaining the key in a pressed state, you will be able to observe dynamic alterations in the terrain's patterns.

> ✨ For technical details and illustrations of the effects, please refer to the "Optional Features" section at the end of the document.

Required Features

The required features include terrain synthesis, procedural textures, and FPS control.

Terrain Synthesis

1. **Value Noise Patch**: This is implemented in `noise.ts`. We incorporated three octaves using `2x2, 4x4, 8x8` patches respectively. Each number in the patch was generated using `100 * this.rng.next()` using a random seed of `<chunk_center_x>&<chunk_center_y>&<patch_size>`. As the height of each cell in Minecraft is an integer, we perform a `floor` operation after the height is generated; also, as the final height is the sum of multiple octaves, and has a chance to exceed the

range of 100, we clip the final height to limit it. After this step, we were able to get the terrain as in Figure 1. we can see that the terrains generated using value noise are very natural and smooth (inside the chunk), and we can observe the structures of landforms such as mountain peaks and basins.

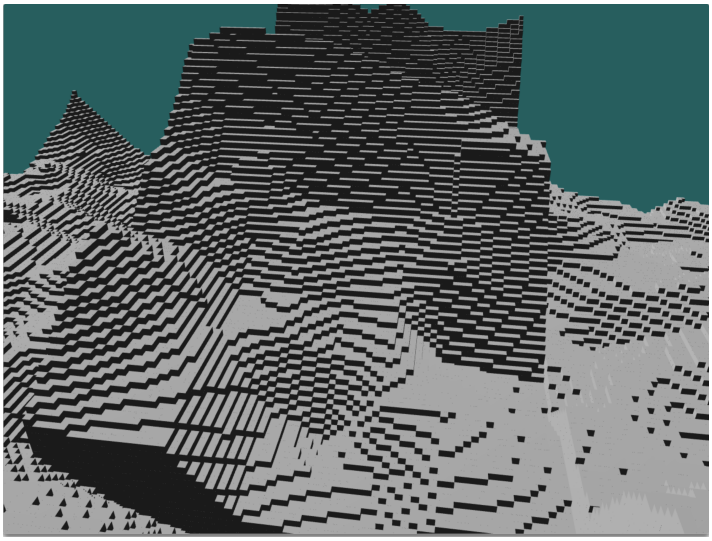


Figure 1: the tri-octave value noise terrain (seamless boundaries not implemented yet in this figure)

2. **Instanced Cube Rendering**: The implementation of instanced cube rendering is conveniently included in the starter code provided to us. For this step, our tasks are straightforward: a) stack the cubes for each cell, and b) eliminate any hidden cubes to enhance performance. We employ an algorithm which dictates that if the current height is lower than all neighboring cells, only the top cube is rendered; otherwise, we render from the lowest neighboring cell to the peak. While the implementation for this step is relatively simple, it is essential to be mindful of boundary conditions. By utilizing occlusion culling, we have successfully reduced the number of cubes to render, resulting in a significant performance improvement.

3. **Seamless Chunk Boundaries**: As demonstrated in Figure 1, value noises enabled us to attain smoothness within individual chunks. However, due to the independent creation of neighboring chunks, considerable inconsistencies emerged at the boundaries. To address this issue, we can incorporate the neighboring chunks' boundary rows, columns, and corners into the current patch, ensuring that the bilinear interpolation takes into account the surrounding information. Consequently, this approach achieves seamless smoothness, even at the boundaries. From a hovering position, we can see the seamlessly connected chunks as shown in Figure 2.

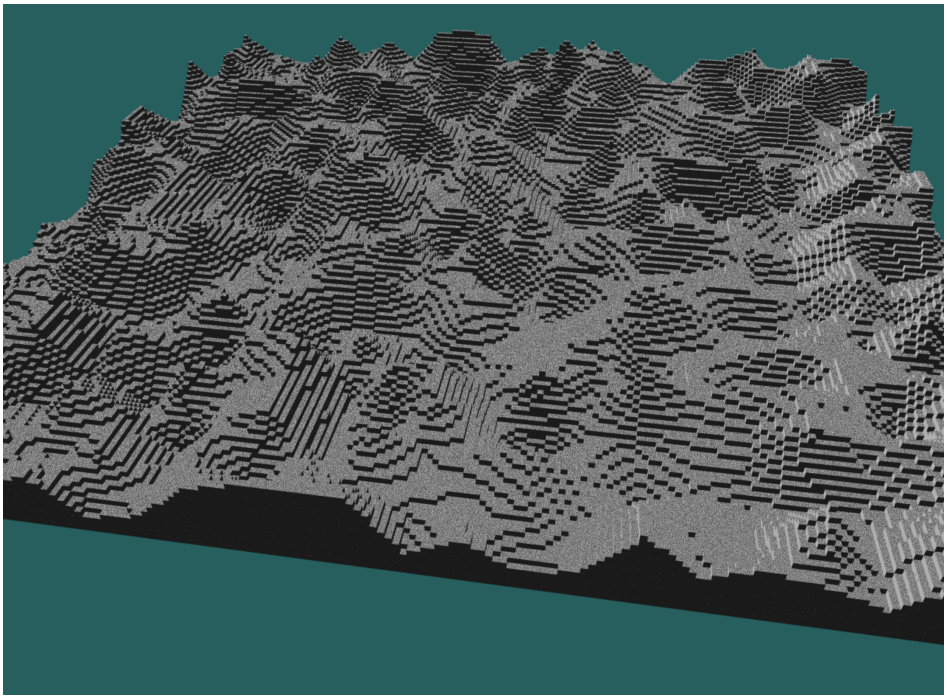


Figure 2: Seamless boundaries between chunks viewed from a hovering position.

4. **Lazy Chunk Loading**: To enable lazy chunk loading, we maintain a `map<string, Chunk>` to map the center position string (e.g. `"0,0"`) to the corresponding chunk. For each time frame, we determine the center chunk, i.e. the chunk that the player currently stands in, and update the visible chunks to it and its eight neighbors.

Procedural Texture

1. **Perlin Noise Implementation**: We implement the basic Perlin noise algorithm as we discussed in class, and we use bicubic interpolation instead of mix() of GLSL to make it smoother.

Figure 3: Image generated using Perlin noise

2. **Perlin Noise Textures**: We have implemented four types of textures: **(1) Perlin Texture**: We achieved this texture by combining 5 octaves of Perlin noise. **(2) Marble Texture & Stripe Texture**: These two textures were generated by combining 5 octaves of Perlin noise and the sin() function: $I(u,v) = \sin(3u + 3v + \alpha n(\beta u + \beta v)) * 0.5 + 0.8$. For generating the marble texture, we used a larger α value of 5.0, while for generating the Stripe texture, we used a smaller α value of 1.2. **(3) Concentric Circle Texture**: We implement this texture by combining 5 octaves of Perlin noise with the sin() function: $I(u,v) = \sin\left(\sqrt{(5u-2.5)^2 + (5v-2.5)^2} + \alpha n(\beta u + \beta v)\right) * 0.5 + 0.8$. These four textures (without color) are shown in the following image:

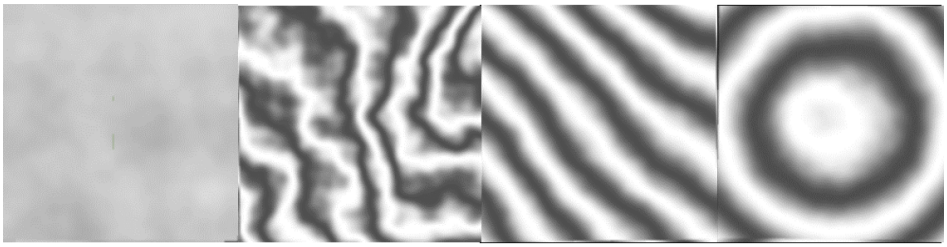
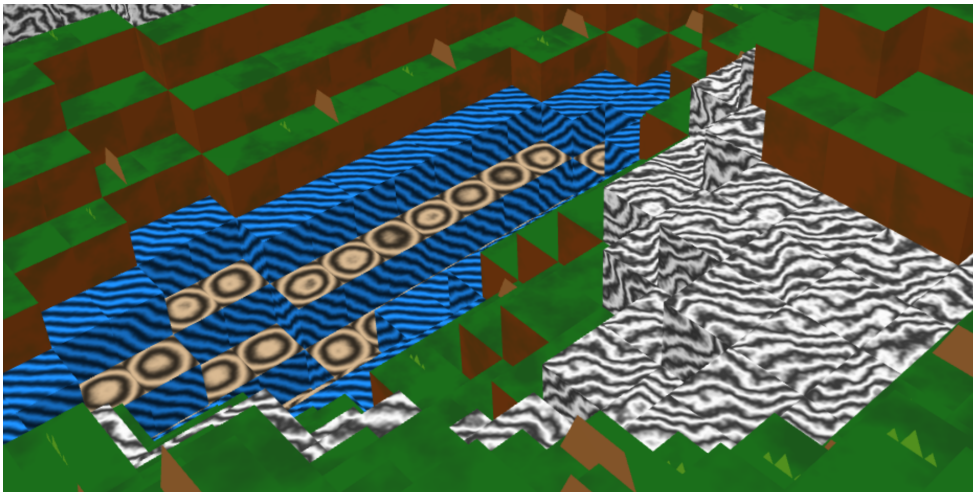
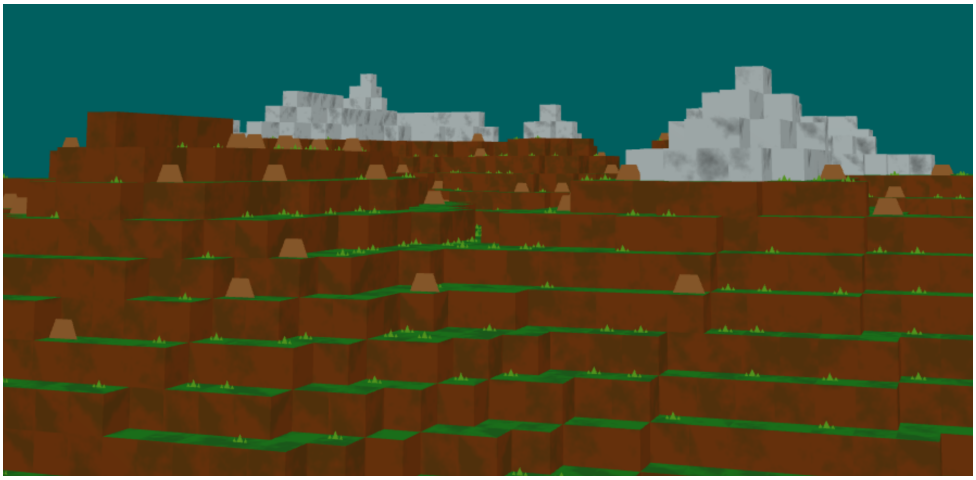
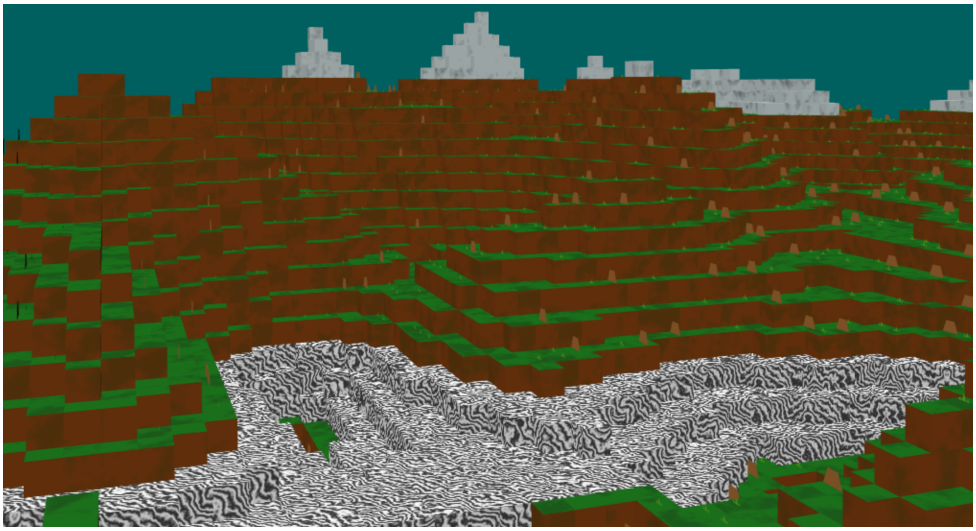


Figure 4: Four different textures. From left to right, they are Perlin Texture combining 5 octaves, Marble Texture, Stripe Texture, and Concentric Circle Texture.

3. **Textured Blocks**: We implement this feature by assigning a type to each cube and informing the GPU of each block's type, and then rendering all cubes at once. We end up with five types of textured blocks. **(1) Grass Block**: The Grass Block uses Perlin Texture, with green on top and brown on the sides. **(2) Snow Block**: When the terrain height is high, Snow Block appears. Snow Block uses Perlin Texture with white color. **(3) Creek Block**: Creek Block appears more often in low-lying areas, and uses Stripe Texture with a blue color. **(4) Cobble Block**: The Cobble Block represents the pebbles in the creek and usually appears together with Creek Block. Its texture uses Concentric Circle Texture. **(5) Marble Block**: Marble Block represents marble and uses Marble Texture with a black and white color scheme. The picture below shows these five types of blocks.







FPS

1. **Collision Detection**: To put it simply, this requirement asks us to find the minimum height for a given position; the player's position cannot be lower than the minimum height, and the player should not be allowed to move (without jumping to appropriate height) to a position where the minimum height is higher than the current position. The tricky part is handling the boundary because its neighbors could be in another chunk.
2. **Gravity**: To implement gravity, we add a `vertical_velocity` member to the `MinecraftAnimation`. Positive means the player is going upwards. In each frame, we check if the player is supported by the ground (position is higher than minimum height, or has an upward velocity), if yes, deduce a `GRAVITY * delta_time` from the current velocity. We obtain this `delta_time` by keeping a `lastTimeStamp` member in the `MinecraftAnimation` class, in every `draw()` function call, we get the current time using `Date.now()`, then the `delta_time` will be the interval between current and the `lastTimeStamp`. We can then update the player's position using basic middle-school physics knowledge.
3. **Jumping**: This is probably the simplest: check if the player is in the air, if not, give him a vertical velocity of `10`.

Optional Features

Day-Night Cycle: In our implementation, the change in the position of the light source is similar to the rise and fall of the sun. The color of the light will be slightly reddish during dawn and dusk, and the world's ambient color will be brighter during the day and darker at night. We use different keys to control the speed of the day-night cycle: Pressing **digit key 1** will automatically complete a 24-hour cycle within 30 seconds, pressing the **digit key 2** will automatically complete a cycle in 1 minute, and pressing the **digit key 3** will automatically complete a cycle in 2 minutes. The day-night cycle will continue automatically unless you press **dight key 0** to stop it, and pressing 0 will also make the light position reset.

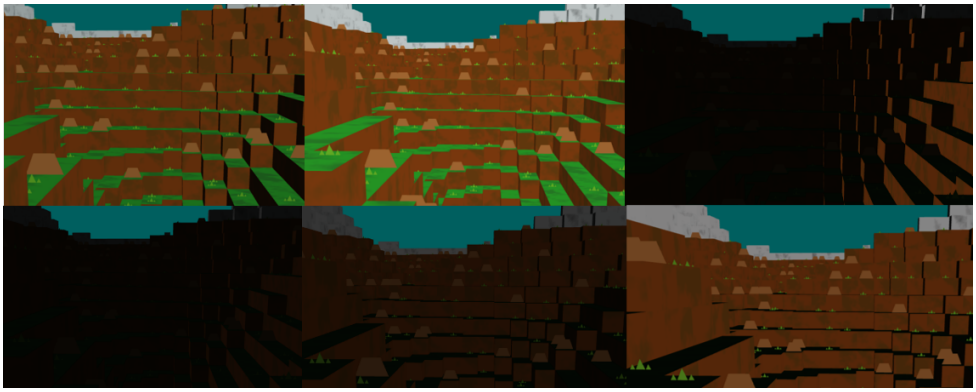


Figure 8: The day-night cycle

Decorative Elements: We have added two decorative elements in the world: grass and big rocks. Overall, we implemented decorative elements by using a similar approach to implementing cubes. We created a class for each decorative element and used instanced rendering to draw them. In terms of implementation details, we made sure that decorative elements can be seen from any direction, meaning they won't disappear no matter how the observer's viewing direction changes. Additionally, we added randomness to the placement of decorative elements so that they don't all appear in the same positions of blocks. We used random numbers to adjust the positions of decorative elements to make the world feel more realistic. We also paid attention to how decorative elements interact with other features, ensuring that they behave correctly during the day-night cycle.

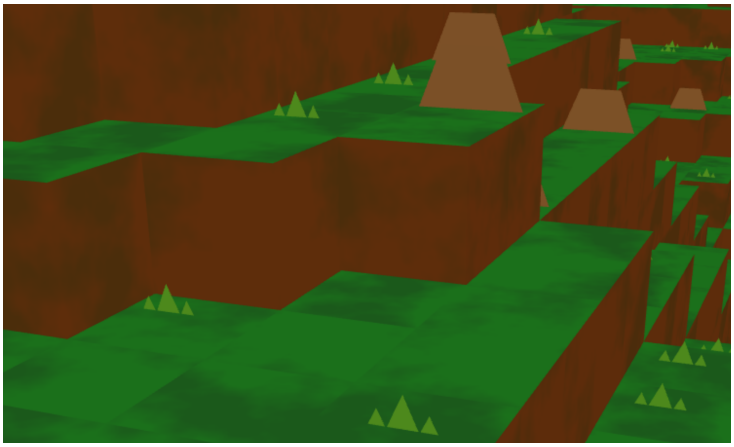


Figure 9: Decorative elements: grass and big rocks.

Time-varying Perlin Noise: We pass different seeds at different times to achieve time-varying Perlin noise, giving blocks animating textures. To test this feature, **keep pressing key P** (P needs to be held down continuously). While the P key is pressed, the world will have animating textures, presenting dynamic changes that make the world appear to be flowing! To stop this feature, release the key P.