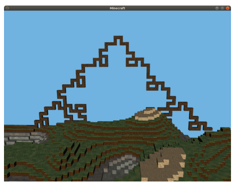
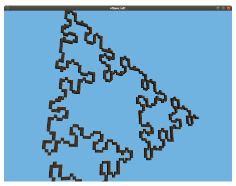
Graphics Final Project Writeup

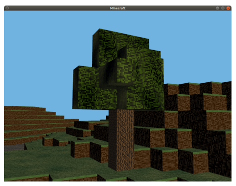
Neil Patil and Kevin Chen

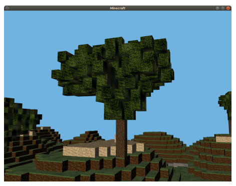
1 Project Goals and Key Algorithms

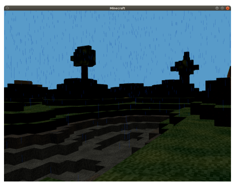
We made improvements to Minecraft, focusing on procedural generation of trees, water physics, and weather effects.

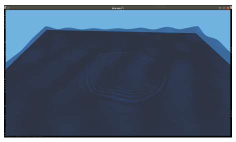

(a) L-System Koch Curve


(b) L-System Serpinski Triangle


(c) L-System Simple Tree


(d) L-System Large Tree


(e) Rain


(f) Water Physics (after droplet)

1.1 Procedural Generation of Trees

Procedural Generation of trees was accomplished using L-Systems. An L-System is essentially a form of recursive grammar which starts with a base string (called an axiom rule) and progressively expands this string by expanding rules for some number of iterations. Characters in the string can be interpreted to do various actions. For example, the L-System generating the Koch curve above is:

axiom: A rule: A $\rightarrow A + A - A - A + A$

Where A means draw/move forward, - means pitch down 90 degrees, and + means pitch up 90 degrees.

Trees are essentially just these rules applied in more complex ways and in three dimension, with special characters for wood versus leaves (F and f respectively).

In order to create more variation in trees (so that each tree looks different), we used the idea of a *stochastic* L-System. Here, a rule can be replaced with a different rule with a random probability. For example, to get variations in leaves in one of our trees, we can do B → [+++f][-f]f OR [***f][///f]f with probability 1/2

In order to draw these L-Systems, we define each symbol to correspond to an axiom, and interpret them using turtle graphics (discussed more below).

1.2 Weather Effects

We primarily implemented rain as a weather effect. In order to implement this, we created a basic particle system, where each "raindrop" was a small line segment whose position and velocity were recorded. In order to create a better-looking rain effect, each particle also had a "life" whose value was randomly initialized and progressively decreased as time went on. When the raindrop's life was depleted, it was deleted. This had the effect of "fading" rain in and out instead of hitting the ground already. We also procedurally generate our rain based on the height map of the terrain, and use culling to prevent performance from suffering too much.

Finally, we implement the idea of a "storm" - the heavier the rain is, the dimmer the lighting gets. Additionally, "lightning" periodically randomly flashes if the storm is heavy enough. Walking out of the storm, the day gets brighter and visibility better.

Press 'R' to toggle rain on/off.

1.3 Water Physics

The goal with water in this project was accomplishing physically realistic simulation in real-time as opposed to simply rendering water based on height functions as we did in the second Menger milestone. This was accomplished with a heightfield simulation approach, where the height of water at each location in a discrete grid is maintained. At each time step, each entry in the heightfield is updated to create a smooth, fluid movement.

So how is each height updated? The key observation here is that the vertical force applied at each point in the fluid is proportional to the curvature at that point. If the curvature is positive (i.e. the surrounding heights are higher), that column of water should experience an upward force, and if curvature is negative the column of water should experience a downward force. With this observation, it is simply a matter of applying Newton's second law to derive an acceleration at each point and have heights move accordingly at each time step.

Rain interacts with the water as well by contributing to the height of water at the location of contact which will subsequently apply a downward force in the next iteration. A simulated floating cube is also included, and the buoyant force is derived from Archimedes' principle.

2 Implementation Details

2.1 Procedural Generation of Trees

In order to develop L-Systems, we wrote a parser to read L-System rules from a json file (src/trees.json), reading attributes like each rule, the angle to draw at, and how often to spawn the generated file.

Then, when a tree is called, one of these systems is randomly picked. We have a recursive procedure that gradually expands the string for some number of iterations.

Finally, once the L-System string is generated, the tree is drawn by interpreting the string using turtle graphics. In the terrain generator, we implement "turtle graphics". Each character in the string corresponds to a movement of the turtle - for example, 'f' means move the turtle forward while drawing leaves, '+' means yaw the turtle right by some angle, '[' means push the turtle's position and orientation to a stack, and ']' means restore the state from the stack.

To move the turtle, we store a quaternion corresponding to the direction, a vector corresponding to position, and a stack that keeps track of state. As the trutle moves, the blocks it draws are added to the list of cubes to be rendered.

2.2 Weather Effects

In order to render rain, we use a data structure to keep track of a particle system. This holds all of the positions and states of the rain particles, as well as implements features to cull unnecessary particles.

One problem we had to solve was that rendering all of the rain all of the time was very inefficient. In order to solve this, we rendered rain as procedurally generated with the terrain, as well as used distance-based culling.

In order to implement the storm, we keep track of an "intensity" that each cube has of rain. This affects how much rain falls in the square, and is also used to darken the environment. In order to darken the environment, we change the ambient term in the shaders according to the intensity. In order to implement lightning, we randomly "flash" the ambient term by increasing it for a few milliseconds.

Rain is rendered as a set of semi-transparent OpenGL lines. The shader rain.vert and rain.frag is used to render these.

2.3 Water Physics

With the heightfield method above in mind, water physics is accomplished by maintaining the height and vertical velocity of water at each location in a discrete grid. Updates at each time step are implemented by estimating curvature at each point in the grid and updating the velocity and height of that position based on the force proportional to curvature.

As far as rendering goes, the water is represented as quads for each collection of 4 neighboring points. Additionally there is a "debug" mode that renders small cubes at each discretized point on the surface which makes the movement of water much easier to see. These small cubes are accomplished with instanced rendering.

To really show off the physics of the ocean, the implementation allows for simulating a droplet of water and the ocean interacts with rain. The ocean-rain interaction is accomplished by checking each rain particle for contact at each time step and increasing the height of water at the location of contact as described in the first section.

Lastly, the implementation has the height of the water vary with the weather; when there's no rain, it slowly decreases, and it will increase with the rain. Since the height update methodology guarantees no preservation of overall volume, the difference in total volume at each time step as well as the target change in total volume is distributed evenly amongst all columns. This accomplishes volume preservation while keeping simulation smooth.