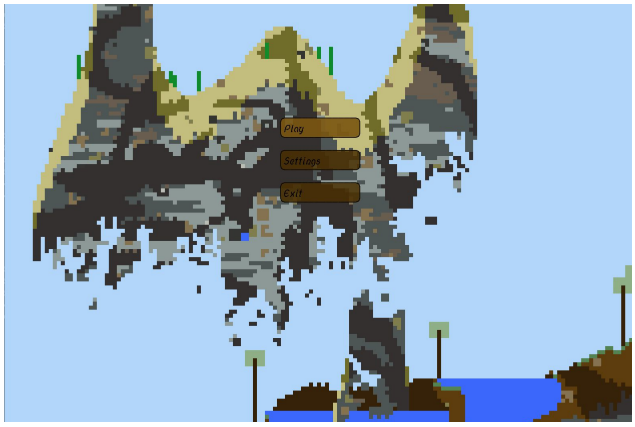
Michael Dillender's
Project Portfolio

1. Aeronef (*github.com/mikedillender/Aeronef*)

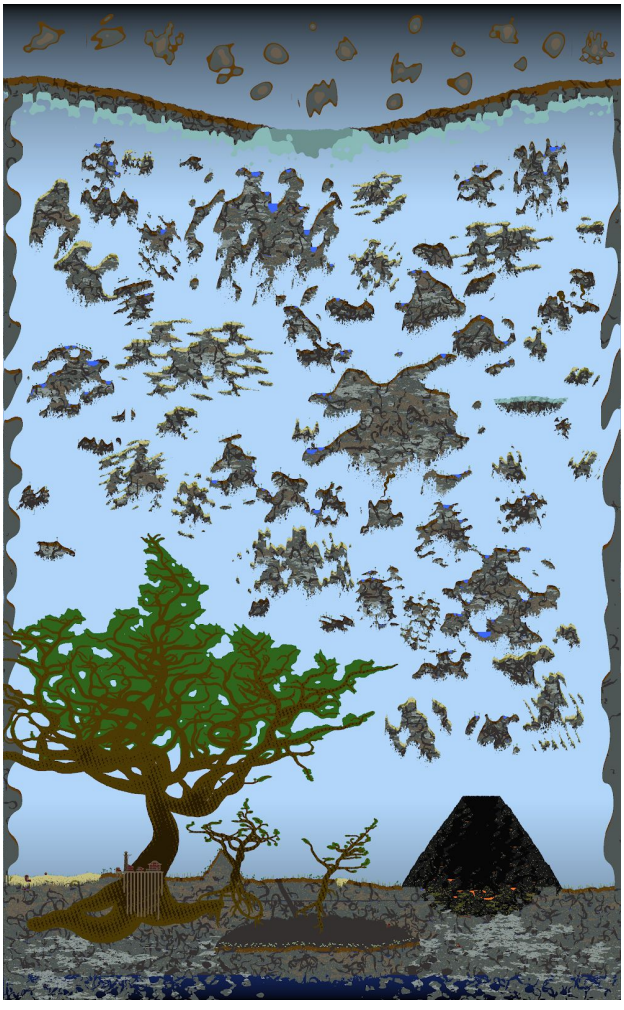
Video documentation can be found on the "Mike Dillender" YouTube Channel



This has been my most creative and long-running endeavor, currently sitting at 32,477 lines of code. As such, it would be nearly impossible for me to describe the creation of it within a reasonable word-count. Instead, I will explain some of the interesting and complex features/algorithms that are incorporated into it.

Notable Algorithms:

- Random World Generation
  - The most complex algorithm is the world generation. Every time a player starts a new game, a new world is procedurally generated, one of these maps is shown to the right. This process alone is around 8,000 lines and takes most computers around 45 seconds to complete.
  - Noise Generation (*github.com/mikedillender /MiscApplets/tree/noisegencontroller*)
    - I independently developed a process for noise-generation using an array of sinusoidal vector functions. This is demonstrated by the unique shapes of the islands in the above images of map generation.
  - Cave Generation
    - Caves are created like worms, tracking their path and removing tiles they pass through.

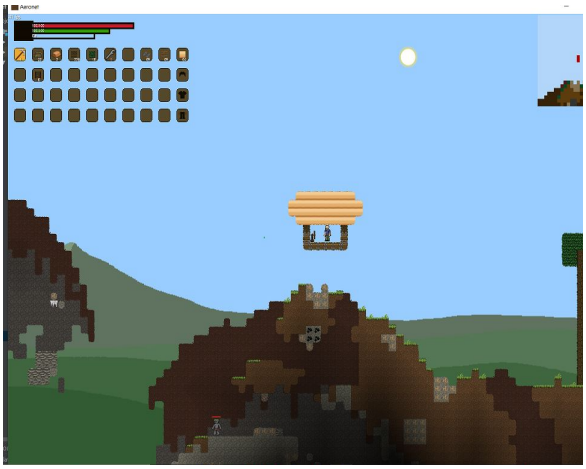


        - This algorithm was adapted to create the world tree in the bottom left of the map.
    - Island Generation
        - Uses noise-generation to create a unique shape, then runs a recursive decay algorithm on the bottom to make it seem as if the islands have eroded and fallen off. After this, caves are generated all throughout the island.
        - After erosion and cave-gen, grass and trees (or sand and cactus) are placed on top of each island, making them look more natural and pleasant.
        - Finally, each island is placed into the most empty region of the map.
        - Each map consists of around 50 islands of varying sizes.

    - Pre-Built Structures
        - Structures can be drawn in *Photoshop* and saved directly as a .png, I have written a script that allows the game to convert this .png data into a structure, and can be placed into the world. This is used for the village at the base of the world-tree.
- Enemy Pathing
    - The enemies use an adaptive pathing system to follow the player
    - They can not only navigate through mazes (finding the fastest possible route), but jump over pitfalls if they are in the way.
- Lighting
    - The lighting system is very efficient, and unlike many similar games, is extremely smooth.
    - The system allows for updates to the lighting every frame, making entity lights possible.
    - Of course, blocks and tiles block light.
    - Certain tiles in the open-air act as light-sources along with torches, but their brightness changes based on the time (i.e. it is dark during the night).

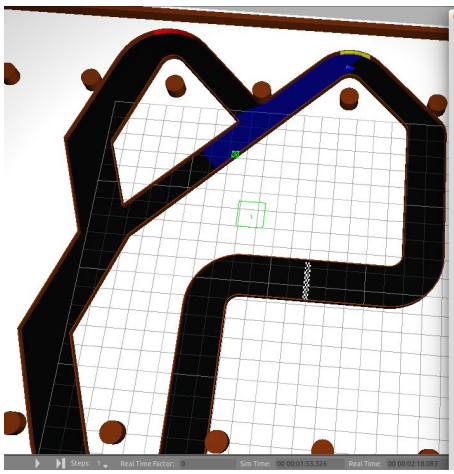
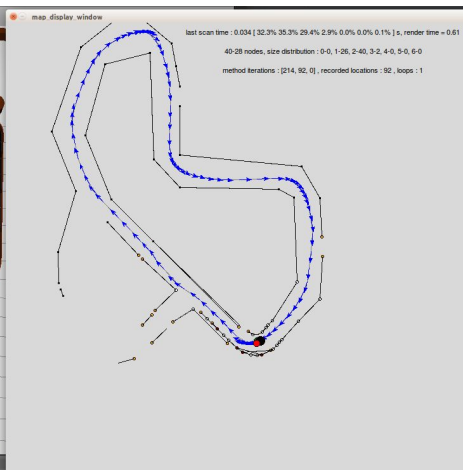
- Fluid Dynamics
    - Water flows from tile to tile to find its level.
    - Obviously, the fluids cannot flow through blocks.
- Animations
    - All entities are animated to some degree
    - All sprites can have up to two degrees of rotation, which is rather difficult to achieve.

- Airships
    - Players can not only create their homes from materials they gather but can turn those homes into airships.
    - Players can control the amount of air in the balloons, controlling their upward lift.
    - If not powered by propellers, the ship is controlled by the wind (which can be determined based on the movement of the clouds in the background)



- Steam Power
    - This is highly integrated into the aforementioned airships. Taking heavy inspiration from the steampunk genre and gilded age technology, the player can create a steam boiler and use it to power an engine.
    - Steam engines can rotate gears, which in turn, can rotate propellers.
        - These propellers, when placed on airships and used with valves, can turn the player-built ships into thermal airships, allowing for full control of its motion.
- Efficient Tile Rendering
    - World files are 2,520x4,050 tiles (for reference, the player is 1x2 tiles), but it only renders the tiles that are on screen, and visible.
    - If the tile is too dark, it will not render, making it much more efficient.
- Automatic World Saving / Loading
    - Whenever the player changes the environment, the change is seamlessly saved to the hard-drive, so no matter when the user exits the game, their data is saved.
    - It also independently saves the player's location and inventory data, and minimap data.
    - Along with player change saving, the biggest save file is the full map.
        - Most world files are 2,520x4,050 tiles, but large worlds are 7,560x12,150 tiles.
        - Each tile consists of five layers: block, wall, complex block, pipe, and liquid.
        - Uncompressed, these files are around half a gigabyte, but with my compression algorithm, they are saved at a mere 17 megabytes.

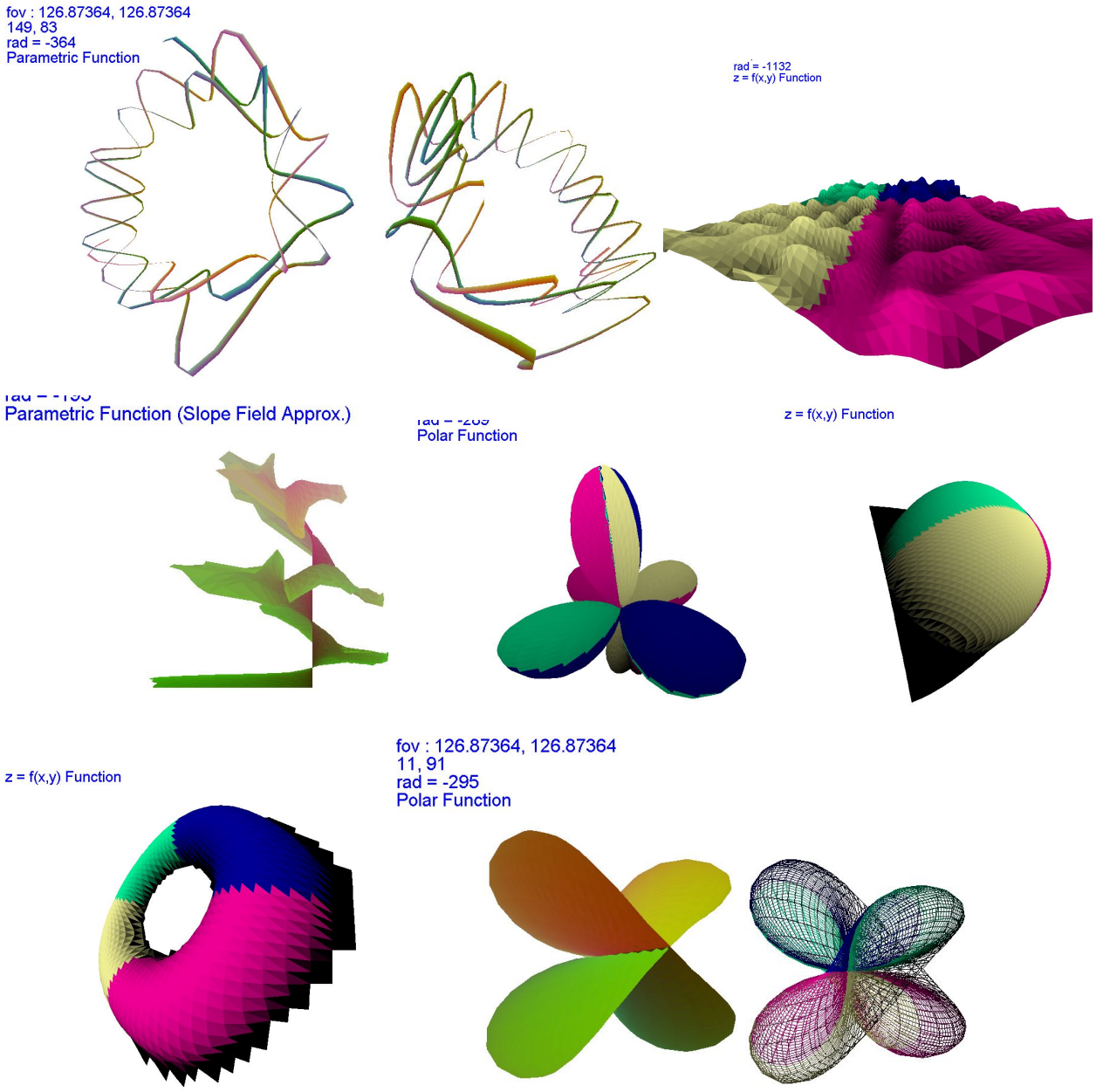
2. Autonomous Vehicles (*https://github.com/BHS-AV/AV_Algorithm*)

- Navigation
    - The navigation is admittedly rather simple, but it can explore a pathway/building completely independently, without crashing. It also manages speed depending on the situation (i.e. slower when turning or when there is a wall ahead).
- Mapping (https://github.com/BHS-AV/AV_Algorithm/blob/master/Scripts/Map.py)
    - This is the portion I have been most passionate about, as evidenced by the file's line count (2280 lines of code).
    - For the first couple of months, I used the lidar to take in data and find a "line-of-best-fit" between data-points, thus creating various walls. However, it was inefficient, and when a driving algorithm is inefficient, crashes are rather frequent. Eventually, I devised a new system, instead of making lines, I would treat the individual data-points as nodes. Once the data was taken in, the nodes were strung together, and unnecessary nodes were removed. This system is able to drive around a track, taking in well over eight thousand data-points, and create an accurate map of a mere forty nodes.
- Map Generation (*github.com/mikedillender/Gazebo-Map-Gen*)
    - After nearly a year of working on the project, I decided that for better software testing, I would need more virtual maps to demo in— but there were none online. So, I decided to make a rather basic script to randomly generate a map for the car to test in.

3. 3D Function Visualizer *(https://github.com/mikedillender/3D-Sim)*

Note: the GitHub link is the same as the Particle Simulator (#4) because this project was built from the particle simulator.



As stated in the note above, this project originated as a 3D particle simulator (#4), but soon progressed into an experiment with 3D functions. I realized that if I created a 2D-float-array, I could use the value of each index as the z-position, and the two array indexes as the x and y. Hence, by inputting a simple z=f(x,y) function, it could create a 3D rendering of it by drawing lines to adjacent indexes.

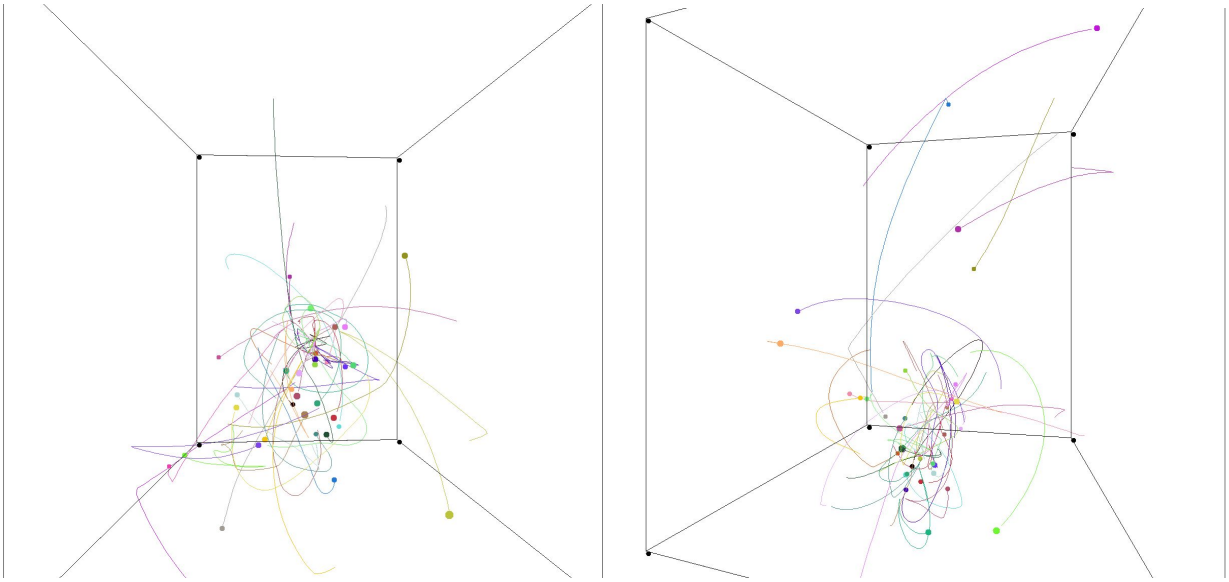
The first problem I had with this was how to handle 3D-rotations of perspective, and since I was unable to find much helpful information on how to use quaternions, I struggled with making it

work. However, I eventually just settled on the admittedly inefficient method of rotationally-translating each point around the origin every render.

Eventually, while doing schoolwork, I realized I could alter my original method to graph polar functions. By making the replacing the float-array with a vector-array, I could instead have the array indexes represent the x-y orientation from the origin and store the x-y positions in the vector. This worked surprisingly well, and after I made an algorithm to panel the graphs and render them in order of distance from the observer, I started to realize the potential this program had.

Finally, I upgraded it to render parametric equations by using the x-index as a fourth variable, time. First, I made basic parametric functions with {x=f(t), y=g(t), z=h(t)}, but later added Euler approximation compatibility, using the y-index as a different initial position, with the input parameter being the derivative functions of each position variable.

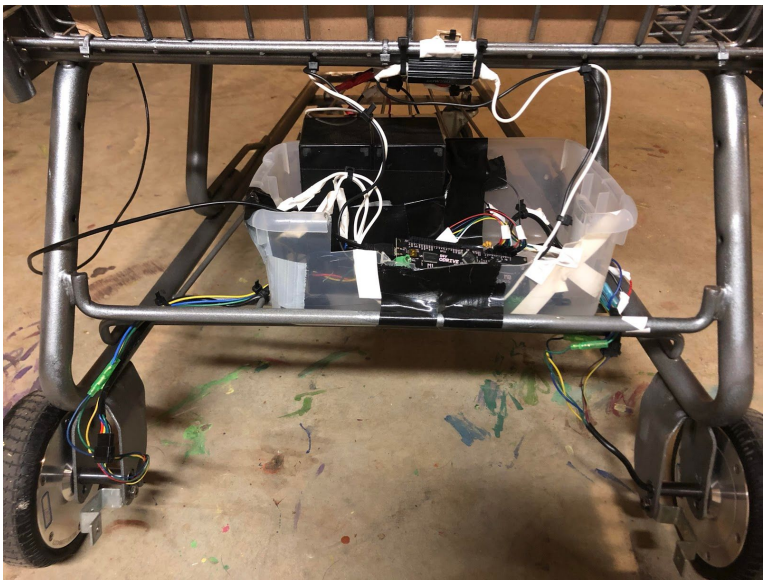
4. 3D Particle Simulator *(https://github.com/mikedillender/3D-Sim)*



I had made several particle simulators prior to this one, but since they were all 2D, I was always annoyed with my inability to model electromagnetism. Since electromagnetism is an inherently 3D force, as demonstrated by the right-hand rule, I knew I had to make a 3D-environment. So, I made this using Java's Applet graphics library.

It can model gravity, electromagnetism, pressure, and collisions between particles and trace their paths.

5. Electric Shopping Cart

I made this a couple of months ago, and it is pretty much my only large-scale project that involves much hardware. The original idea was born from a desire to take my team's work in Autonomous Vehicles to the next level, but that never really worked out with my teacher. So it became a passion project until I finished it after a couple of weeks. I found the cart abandoned in the woods near my house, and using some electronics from my basement, managed to power it with electric motors. Eventually, I managed to make it controllable via a python script on my laptop.