Simulating Expansive and Detailed Ecosystems in Real-Time

Alexander N. Irausquin-Petit (student)[1]*, Ms. Van't Slot (teacher)

1. Dwight-Englewood School, 315 E Palisades Ave, Englewood, NJ, 07631, United States

* alex.irausquinpetit@gmail.com

Abstract

This paper proposes a solution to simulating and handling a large, detailed, single-biome ecosystem. I plan to use a multi-scale approach to structuring the simulation world and mathematical approximations for real-world mechanics in order to optimize the simulation to run in real-time on consumer hardware. My multi-scale data structure will use a compound grid system with varying grid sizes, isolating simulation systems to individual grid levels. This data structure will also be able to perform high-speed neighbor checks and range queries. My simulation will use this data structure to simulate tree growth, water flow, and weather systems that will all be linked together to create a complex, dynamic, and realistic ecosystem.

Keywords

Simulation; Data Structure; Realtime; Large Scale; Ecosystem;

1. Introduction

Simulating an ecosystem, especially on large scales, has been a very challenging task to perform interactively in real-time. There are several approaches to parts of this problem, such as simulating tree growth or simulating water flow, but either they sacrifice scale (1 tree, 3ms per iteration) in order to perform at interactive speeds with high levels of detail [Reffye et. al. 1997] or they sacrifice speed to increase scale (468,199 plants at 18,796 ms per simulation step [Makowski et. al. 2019]). This project proposes a solution to creating an ecosystem simulation that runs in real-time on large scales (10 x 10 km, with around 100,000 plants) while running in around one millisecond per step and having high levels of detail.

Although I worked to balance accuracy and speed when designing this simulation, I still had to sacrifice accuracy in some places to achieve my desired speed. I did this by using mathematical approximations of real-world physical mechanics. By reducing a complex interaction to a simple mathematical equation I was able to maintain similar results to a true-to-life simulation while also being able to significantly speed up the simulation. The water simulation is an example of this, where I simplified the physical properties of 3D water into a 2D plane. Furthermore, due to the nature of the grid data structure, the resolution of the weather and water simulations is limited to the dimensions of their level cells. I did this because of the large simulation scales, where simulating water and weather with sub-one-meter resolutions will be too computationally expensive and not very important at such scales. In reducing the complexity to increase the speed of the simulation, I had to sacrifice accuracy. For the tree

growth, I used a node-based tree, which made the tree representable in a very computer-efficient format with the effect of reducing the detail and accuracy of the grown tree. This approach maintains a lot of accuracy while also consolidating the complex interactions for every branch and leaf in a tree into only a structure of nodes. Overall, my approach to this simulation is not targeted at fields requiring extremely precise and accurate simulations but is instead targeting fields that need large-scale simulations but not very accurate results such as video games, 3D animation, and forestry.

1.1 Data Structure

The data structure contains all the data being simulated, so designing an efficient data structure is pivotal in having this simulated ecosystem run in real time. I needed the structure to be able to perform adjacency queries, positional queries, and data-type queries, be able to store different levels/resolutions of data, be able to have data read from and written to in a structured and organized manner, and, of course, have as little overhead as possible (preferably under a millisecond per tick). I initially planned on using a pre-existing solution to this problem but found none that met all of my requirements and ultimately had to settle on a custom solution. The custom solution is a hybrid sparse grid and hierarchical quadtree - a data structure that partitions a space into progressively smaller quadrants, combining the multi-level structure of hierarchical quadtrees that allows for fast positional queries and multiple resolutions of data with the benefits of the fast adjacency checking, memory efficiency, and uniformity of sparse grids. The data structure is composed of multiple *Levels*, each Level representing data of a certain resolution. For example, the highest Level would have the highest resolution, and in turn most detailed data; the lowest Level would have the lowest resolution and the least detailed data.

1.2 Tree Growth

There have been several approaches to simulating the growth of a tree or plant. Many have achieved accurate and realistic results but at the cost of being able to either simulate only one tree or a small number of trees [Hart et. al. 2003; Reffye et. al. 1997]. Makowski et. al. [2019] was able to accomplish an impressive level of detail at large scales but it was not simulated at interactive speeds. My approach attempts to find a compromise between speed and accuracy at very large scales in order to achieve real-time simulation speed. I used a node system to represent trees, where each node represents a branch of the tree. Each tree has a root node, the base of the trunk. The growth of a tree is based on its *Hydration* and *Energy* values. *Hydration* is the amount of water in the tree. *Energy* is a value computed from the sun exposure, number of branches, and the number of leaves. If the *Energy* is above a certain threshold, it can grow a new branch. If the *Energy* is a certain amount above the growth threshold, the tree

can reproduce. If the *Energy* falls below zero, then the tree dies. Each tree also has a set of parameters that control how it grows - essentially its genes. When a tree reproduces, each of its parameters has a random chance to mutate, which changes its value by a small random positive or negative amount.

1.3 Water

To create the above-ground, or surface, water I used a method called the hydrostatic pipe model (or the column-based model), which isolates water into individual columns. There have been many approaches using this method [Št'ava et. al. 2008] for interactive large-scale simulations like this. The benefit of this method is that it is very efficient for simulating very large-scale bodies of water. Its efficiency comes from the way a column consists of a single number that represents the height of the water in that column. At every step of the simulation, each column attempts to flow into its neighboring columns, creating a flow of water between cells. In every step of the simulation, first, a small amount of water evaporates (only if there is not a lot of water in the column), then the amount of water flowing into neighboring cells is calculated, and finally, the new height of the water column is calculated. Soil water saturation is then simulated by absorbing some surface water, evaporating, being absorbed by trees, diffusing to neighboring columns, and then, if saturated, releasing some water as surface water.

1.4 Weather

While I was unable to implement a weather simulation in the way I had originally planned due to timing constraints, I still plan to add a more complete weather simulation to this project in the future. Weather is a very complex system, more intricate and deeply interconnected than any other part of an ecosystem. It directly affects the flora, fauna, and terrain of an area and is also directly affected by the flora, fauna, and terrain. As such, a weather simulation is indispensable to an ecosystem simulation, and getting an accurate and detailed weather simulation is crucial. Once again, there have been several approaches to simulating weather. Some have tackled cloud dynamics [Harris et. al. 2003 and Lauer et. al. 2013] while another has covered precipitation [Stevens et. al. 2020]. Harris et. al. [2003] achieves interactive and accurate simulations by spreading out cloud dynamics computation over several rendered frames. Lauer et. al. [2013] and Stevens et. al [2020] achieve physically-accurate simulations at high levels of detail (0.156 km to 2.5 km resolution) at extremely large scales (global or continent-sized) but nowhere close to interactive speeds (36 to 48 hours to simulate 24 hours on datacenter computers). My approach is very different compared to those mentioned above, as it does not simulate the weather for every point in the simulation space, instead simulating the weather for one point and applying it across the whole simulation space. I had to significantly simplify the weather simulation to accommodate the

aforementioned time limitation. The current approach in this paper (I have future plans to update this paper with a weather simulation similar to my original plan) sacrifices a significant amount of accuracy in order to achieve interactive speeds by approximating the complex interactions that the aforementioned approaches simulated physically and by computing only one value for the weather and applying it globally throughout the simulation space.

2. Methods

My solution offers a unique approach to simulating an ecosystem in real time by separating different steps of the simulation and staggering each step of each part so as to spread out the computation load over more frames. My simulation consists of four parts: the above-ground (or surface) water simulation, the below-ground (or soil) water simulation, the tree growth simulation, and the weather simulation. While these most certainly do not encompass the full complexity of an ecosystem in the real world, they provide enough detail for an ecosystem to be believable while also being capable of handling very large simulations at an interactive speed.

This simulation is designed to simulate only a single biome, and for the purpose of this paper, it will be simulating a temperate deciduous forest, although the growth characteristics of trees, soil water capacity, terrain, and rainfall amounts can be changed. I built this simulation using the Unity game engine, version 2021.3.18f1, all code in this project is written in C# and ShaderLab. It runs in real-time on a laptop with an Intel® Core™ i7-1065G7 CPU at 1.30 GHz, integrated graphics, 16 GB of RAM, and Windows 11. The metrics and data shown in the results section were taken on a desktop with an Intel® Core™ i5-10400F CPU at 2.90 GHz, an RTX 2070 Super GPU, 32 GB of RAM, and Windows 11.

2.1 Data Structure

When designing the data structure, I went with a hybrid sparse grid and hierarchical quadtree, combining the multi-level structure of hierarchical quadtrees that allows for fast positional queries and multiple resolutions of data with the fast adjacency checking, memory efficiency, and uniformity of sparse grids.

The data structure is composed of multiple *Levels*, each Level representing data of a certain resolution. Levels can only be integers. For example, the highest Level has the highest resolution, and in turn most detailed data; the lowest Level has the lowest resolution and the least detailed data.The hierarchical system along with the relationships between each cell is visualized in *Figure 1: Data Structure Hierarchy*.

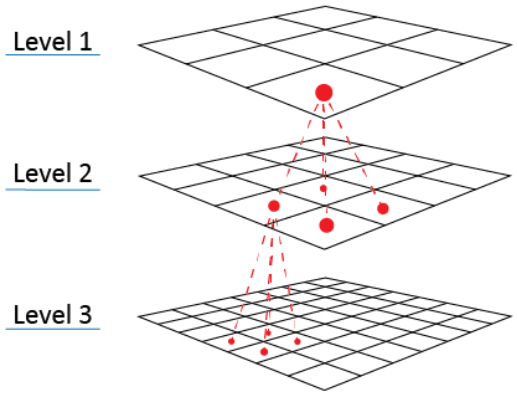

Figure 1: Data Structure Hierarchy

Each cell (example cells shown with red dots in *Figure 1*) contains a list of cells from the level directly below that are within its bounds. This is to facilitate fast positional queries. Each cell also knows its neighbors so that it can easily perform adjacency checks.

It would be a set of different-sized sparse grids, where each cell contains references to all the cells one size smaller than itself that it contains within its bounds. This allows it to perform very efficient neighbor checks while also being able to perform efficient range queries. Each cell has a list of Cell Data - a reference to an object in the 3D scene (called a game object). Each grid also holds a list of Grid Data. Grid Data (not the same as Cell Data) is not a local data type, instead being assigned one per grid and representing the entire grid. This type of data is useful in situations where the data must be executed by the GPU and must be stored as a render texture (a type of texture used by the Unity game engine) instead of cells (for example, the water simulation). A type of Grid Data can only reside on one level, for example, the soil saturation render texture can only ever exist on Level 1 and not on any other level.

2.1.1 Code Implementation

The main components of the data structure are described programmatically, with example code (this is not the actual code, just a simplified example version), is as follows:

1. A `GridCell` class that holds its position, bounds, and level (which is an integer number for its level and a list of Cell Data types it can contain), child cells, parent cell, and a dictionary of data name to the data object. The following example code provides all the data a cell holds but does not include the constructor or any methods (for brevity).

```
public class GridCell {
	public Vector2 center;
	public Bounds2D bounds;

	// Contains level integer and the cell data types that level can
contain
	private GridLevel level;

	private GridCell parentCell;
	private List<GridCell> childCells;
	// 0 = above, 1 = left, 2 = below, 3 = right
	private GridCell[] neighbors;

	private Dictionary<string, List<AbstractCellData>> data;
}
```

2. A `Grid` class that holds its size in the simulation space, it's Level Class (which is a data container that hold an integer number for its level and a list of Cell Data types it can contain), its parent grid and its child grid, a list of its cells (currently, in the source, it is a dense grid instead of a sparse grid due to time constraints), and a list of its Grid Data. Again, a simplified example is provided below.

```
public class Grid {
	public Vector2 size;

	private GridLevel gridLevel;

	public Grid parentGrid;
	public Grid childGrid;

	private List<GridCell> cells;

	public Dictionary<string, AbstractGridData> gridData;
}
```

3. An `AbstractGridData` class that acts as a template for other Grid Data classes to inherit from this. It contains the abstract `GetData`, `SetData`, and `Dispose` methods whose functionality is implemented by the inheriting class, and is used by the data structure manager to manipulate the Grid Data this holds. Note that this example (along with the actual one in the source) does not contain any actual data objects. This is because the data object can be different for each inheriting class and so they must implement it. A simple example is below.

```
public abstract class AbstractGridData {
        // Copies the data from this AbstractGridData to the given object.
        public abstract void GetData(object target);

        // Copies the data from the given object to this AbstractGridData.
        public abstract void SetData(object source);

        // Dispose this data. (Dispose of any buffers, textures, etc from
memory). This should ALWAYS be called before this object is destroyed or
discarded. (Although, this should not be commonly used because you want to keep
the same data object at runtime)
        public abstract void Dispose();
}
```

4. An `AbstractCellData` class. This is similar to the abstract Grid Data class except for the fact that it actually holds data such as the game object (an object in the 3D scene), its transform (position, rotation, scale, etc.), and a boolean for if it is static or not. An example is below, with methods and constructor excluded.

```
public abstract class AbstractCellData {
        // Whether this cell data is static. A static cell data means that its
gameObject does not move around, and so would always be in the same cell.
        private bool isStatic

        // The object in the 3D scene
        private GameObject gameObject;

        // The position, rotation, and scale of the game object
        private Transform transform;
}
```

2.1.2 Grid Levels

Each level of the data structure is pre-assigned what types of data it should contain. As is described above, the `GridLevel` object contains an integer number referring to its Level in the hierarchy and a list of Cell Data types that that Level can contain. The different Levels and the data they contain as they appear in the current state of this simulation are compiled below in *Table 1: Implemented Levels*.

Level	Resolution	Uses
1 - Small	1m	Surface and soil water level, soil type, and terrain elevation, soil
2 - Medium	10m	Objects like trees, rocks, plants, animals, etc.
3 - Large	25m	Used to speed up positional queries.

Table 1: Implemented Levels

I had originally planned for a more diverse level system (shown below in *Table 2: Planned Levels*) but as I worked on the project it became apparent to me that the Cell Data is best used only for containing physical objects in the simulation world and simulation-wide data should be a Grid Data, because of the large speed boost that running the simulation on the GPU provides.

Level	Resolution	Uses
1 - Tiny	1m	Surface water level and terrain elevation and slope
2 - Small	10m	Average terrain elevation and slope (averaged from the small level), soil type, soil water data, and tree cover
3 - Medium	20m	Weather data like temperature, humidity, and fog, entities like trees, rocks, plants, animals, people, etc.
4 - Large	1km	Weather data like wind speed, wind direction, clouds, and precipitation
5 - Super Large	10km	Spatially partitioning large sections of the simulated world, only contains child cells for streaming nearby parts of the world in and out of memory and doesn't contain any other data

Table 2: Planned Levels

2.1.3 Ticking

I needed to make sure that every system in the simulation operated in a specific order. This is so that data is read and written in such a way that it improves efficiency and provides a more stable simulation. To do this, I created a Systems Manager, which has the job of handling every system or class that needs to be ticked at regular intervals. For a class to be ticked it needs to implement the `ITickableSystem` interface by implementing a `TickPriority` variable, a `TickInterval` variable, and the `BeginTick`, `Tick`, and `EndTick` methods. The process for ticking is (for every frame):

1. Calculate which tickable object will tick now using its `TickInterval`.
2. Sort the list of tickable objects by their `TickPriority` value (-1 goes first no matter what and is reserved for the data structure manager) in ascending order so that the lowest value is ticked first.

3. Call the `BeginTick` method in every tickable object, in the order of the sorted list.
4. Call the `Tick` method in every tickable object, in the order of the sorted list.
5. Call the `EndTick` method in every tickable object, in the order of the sorted list.

This maintains a proper order of execution and the `BeginTick` and `EndTick` methods allow for the implementers to all be fully initialized for the current tick *before* the current tick happens. Moreover, I made it so the manager ticked different systems at different speeds as a way of spreading out computation over multiple frames, which are a single rendered image of the simulation being displayed on the screen (most current displays refresh their image 60 times per second).

In order to make these simulations run in real time I employed a state-of-the-art technique, called "simplification", in order to boil down complex physical and chemical interactions into simpler mathematical equations and in order to represent complex natural structures like trees in a simpler format in the simulation.

2.1.4 Data Reading & Writing

The data structure is designed specifically so that its data cannot be directly accessed through a class reference and can only be accessed through the use of an interface, thus decoupling the main data structure from any class that needs to use it and preventing unwanted interference through the process of a tick. For a class to read or write data from the data structure it needs to implement either the `IReadDataStructure` or the `IWriteDataStructure` interfaces. The implementing class indicates what data it wants to read/write and has a respective method in order to receive/send data. In `BeginTick`, the data structure sends requested data to every reading class. In `EndTick`, the data structure receives and stores the data returned by every writing class. The data structure manager is ticked before any other system to make sure that every system that reads/writes data is processed before they themselves are ticked.

2.2 Water Simulation

One of the most important parameters that will affect the growth of plants is water. I plan to simulate two types of water, physical water, and soil water. Physical water is above-surface water, like lake and river water, flood water, and ocean water, and won't affect the growth of trees in my simulation.

Soil water is the water inside of the soil, and is a parameter used by trees to determine its growth rate. Accurately simulating water flow through soils is a very challenging field that is far too complex to simulate on such a large scale. In order to perform water flow simulations I plan to use an approximation of this complex system. Referring back to the grid structure, each Level-1 cell (1m x 1m) will contain the local terrain data. Of that data, the water simulation will use the elevation, soil type, and physical and soil water data.

There are two approaches commonly used when simulating water. The first method is called particle-based simulation (PBS) which simulates individual water particles to create highly accurate and realistic water simulations [Müller et. al. 2003]. Unfortunately, these simulations are very computationally expensive to perform on large scales. The second approach is called a shallow-water simulation (SHS) which simulates only the surface of the water body. This method is significantly faster than PBS and is effective at simulating water on large scales, albeit not as accurate due to how it simplifies the three-dimensional nature of water into two dimensions. I used a version of this simulation called the hydrostatic pipe model (or the column-based model). The benefit of this method is that it is very efficient for simulating very large bodies of water. Its efficiency comes from the way a column consists of a single number that represents the height of the water in that column. This makes it a grid-based simulation, where each grid cell is a column of water, and so it is inherently parallelizable which allows it to be run on the GPU (graphics processing unit), offering a significant speed boost compared to simulating it on the CPU (central processing unit).

2.2.1 Surface Water

My implementation of this simulation uses an image where each pixel is a column of water and its color represents the height of the water in it. This effectively represents the grid structure of the simulation's corresponding Level. At every step of the simulation, each column attempts to flow into its neighboring downhill columns, creating a water flow between cells. In every step of the simulation, first, a small amount of water evaporates (only if there is not a lot of water in the column), then the amount of water flowing into neighboring cells is calculated, and finally, the new height of the water column is calculated.

The first part of the surface water simulation is water flow. Water flows from high-elevation cells to low-elevation cells, using the height values of the terrain. In order to run this simulation on the GPU I used the built-in Compute Shader system in the Unity game engine. Compute shaders are scripts that are run on the GPU to perform large, repetitive tasks. Any data that needs to be inputted or outputted from the shader is formatted as a texture that is passed back and forth between the shader and the main code.

The second part of the surface water simulation is evaporation. Evaporation works so that more water evaporates the less water is contained in the cell. Evaporation operates according to the following equation:

$$Water\ Height\ After\ Evaporation = h - ((-2.4389 \cdot (h - 0.689655)^3) \cdot E) \quad (1)$$

In which h is the current water height in that cell and E is the Surface Evaporation Constant.

2.2.2 Soil Water

Soil water is simulated through a four-step process. The first is absorption, where a small amount of surface water is absorbed into dry soil. The second is diffusion, where the soil water diffuses downhill. The third step is release, where saturated soil releases a small amount of its water into the surface. The fourth step is evaporation, where, similarly to surface water, a small amount of water is lost only if there is no surface water above the soil.

To simulate the absorption of surface water I use the soil type and soil saturation of the cell that the water is flowing into. I used the article from the Noble Research Institute on Soil and Water Relationships [Ball Unknown Date] for information on how soil particle density affects its absorptivity. It tells us that the denser a soil is, the less water it can absorb and contain. My implementation of this makes it so that there is a range of soil density from 0 to 1 (the pixel color value range), which affects the amount of water a cell can hold. This data is static, meaning it does not change throughout the simulation and is held as an image in order to be sent to the GPU. This density value is used to compute the amount of water absorbed, which is subtracted from the surface water level and added to the soil saturation. The formula for this is as follows:

$$Water\ Absorbed = (d - s) \cdot I \quad (2)$$

In which d is the local soil density, s is the soil saturation, and I is the Absorption Constant.

To simulate the evaporation of soil water, I lower the saturation so that it approaches 0 ($\lim_{t \to \infty} c = 0$ where t is time and c is soil saturation). I use the following equation and an evaporation constant to calculate the new soil saturation value after evaporation occurs:

$$Saturation\ After\ Evaporation = c \cdot ((1 - E) \cdot (1 - d)) \quad (3)$$

In which c is the current soil saturation, E is the soil evaporation constant, and d is the local soil density.

To simulate soil water use, where tree roots absorb soil water, the code samples a soil use texture created by the tree growth manager (more details on this in the next subsection). The values in this texture are also on a scale of 0 to 255. Each pixel in this texture is a sum of all the water consumed by each plant contained inside that cell over the last 10 ticks (these are not system ticks, but based on the number of times the water simulation manager is ticked). Finally, I will add up both the amount of water lost to evaporation and the amount absorbed by plants to get the total amount of water a cell will lose. The amount of water absorbed is calculated using the following equation:

$$Water\ Used = \frac{u}{s^2} \cdot (1 - d) \cdot U \tag{4}$$

In which u is the consumption of water in the tree-Level cell containing this cell, s is the sampler scaler - the number of soil cells there are in one cell on the tree Level, d is the local soil density, and U is the soil use constant.

Then, to simulate diffusion I set the saturation of a cell to the average amount of water in the neighboring cells, but only if that neighbor is above the current cell. This biases the diffusion in a downhill direction, creating a more accurate model of soil water diffusion.

Finally, for the last step I simulate water release. This is the process in which a saturated soil cell can release a small amount of water back into the surface, but only if no surface water exists above it. I calculate the water released using the following equation and then add that value to the surface water level and subtract it from the soil saturation value:

$$Water\ Release = c \cdot (1 - \frac{s}{T}) \cdot R \tag{5}$$

In which c is the current soil saturation, s is the current surface water level, T is the soil release surface water threshold (a constant), and R is the soil release constant.

All of these formulas work together to determine the final saturation value of a soil cell. This mathematical approach minimizes inter-cell interactions and decreases the complexity of the simulation, in turn drastically reducing the computation time. Another significant speedup is using a Compute Shader, as mentioned above, that allows this simulation to run entirely on the GPU.

2.3 Tree Growth

I used a node-based approach to simulate the growth of a tree. This structure enables the tree to be easily represented in computer memory and also allows for efficient handling and storing of trees [Marian42 et. al. 2021]. Each node in the tree represents a single branch and holds a reference to its parent branch and a list of the branches growing from it. The growth of each tree is determined by its *Parameters*, or genes, that specify various characteristics of its growth (branch length, trunk thickness, leaf size, height, branch density, etc.). A tree also has an age value and a hydration value, both of which are used in its growth process.

To reduce complexity and increase performance I used a single-biome approach instead of determining biomes dynamically based on precipitation and temperature like in Makowski et. al. [2019]. My biome of choice is a temperate seasonal forest, specifically those found on the northeastern coast of the United States. Keeping track of tens of thousands of trees and plants is handled entirely by the data structure, ensuring that the only thing the simulation manager has to deal with is the simulation itself. This decoupling of code systems improves readability which has the effect of not only speeding up development but also speeding up the simulation itself by making sure code is reused efficiently and by ensuring that different systems don't interfere with each other.

2.3.1 Tree Management

All trees in the simulation are ticked, managed, and interfaced with the data structure by a *Tree Manager* class. This class uses the interfaces mentioned in *Section 4.1: Data Structure* to be ticked by the Systems Manager and communicate with the Data Structure. For every tick, the Tree Manager chooses 5 random trees to be ticked. At the end of every tick, all new trees are sent back to the data structure to be stored and all just-dead trees are sent to the data structure to be removed. All trees are stored on Level 2 cells in the Data Structure.

For every tick, the following actions performed for each tree being ticked:

1. The tree's water use and water absorption are calculated, and the resulting new hydration value is calculated. The Energy of each node is also calculated.
2. If the tree's hydration is above its *Growth Threshold* then it will attempt to grow new branches.
3. If the tree's hydration is above its *Reproduction Threshold* and its age is also above its *Reproductive Age Threshold* then it will attempt to reproduce.
4. If the tree's hydration is below its *Pruning Threshold* then it will prune some of its leaves or branches in an attempt to reduce its consumption of water.

5. If the tree's hydration is at or below zero then it dies and is added to a list of *nulled trees* that will be sent to the data structure for destruction at the of the tick.

2.3.2 Hydration

The Hydration value of a tree is calculated at the end of a tree's tick. It first calculates the water used during that tick and then subtracts it from its current hydration. It then calculates the water absorbed by the tree's "roots." The calculation for the water use is as follows:

$$Water\ Use = \sum_{node}^{C} ((S \cdot e \cdot r^2 \cdot l^2) + (p \cdot L \cdot Q \cdot R)) \quad (6)$$

In which C is the number of nodes in the tree, $node$ is the current node in the loop, S is the Branch Size Use Constant, e is the energy of the current node, r is the radius of the current node, l is the length of the current node, p is a binary value representing whether or not the current node is a leaf (0 if false, 1 if true), L is the Leaf Use Constant, Q is the tree's genetic Number of Leaves Per Branch, and R is the tree's genetic Leaf Size.

The calculation for water absorption is as follows:

$$Water\ Absorption = A \cdot r + l \quad (7)$$

In which A is the Water Absorption Constant, r is the root node radius, and l is the root node length. The calculated Water Absorption value is then subtracted from the calculated Water Use value to get the Hydration delta. That delta is then subtracted from the tree's Hydration value to get the new Hydration value.

2.3.3 Growth

When a tree attempts to grow new branches, it will choose the top x branches with the highest *Energy* value. The *Energy* of a branch is a calculation of the sunlight that reaches the branch and the depth of that branch in the tree. The equation for the energy of a branch is as follows:

$$Energy = 1 - 0.0001 \cdot d - e^r \quad (8)$$

In which d is the depth of the node in the tree, and r is the sunlight exposure (calculated by getting the distance to the nearest object directly above that branch).

A branch grows with respect to heliotropism (when a plant follows the movement of the sun) and intersection avoidance. This means that when a new branch is created, it will grow in the direction of the sky but also away from nearby obstacles. Whenever a branch attempts to grow a new branch off of itself, it first calculates how long that new branch would be and then determines where that new branch would grow.

The equation for branch length is as follows:

$$New\ Length\ = (map\{L, T \Rightarrow 0, 1\} : h\ ) \cdot B^{d} \quad (9)$$

In which L is the tree's genetic Branch Length, T is the tree's genetic Trunk Length, h is that node's *Trunkiness* value - a number representing how close the node is to the trunk, B is the tree's genetic Branch Length Falloff, and d is the node's depth in the tree. The map function refers to remapping the value v from the range of min_1, max_1 to the range of min_2, max_2 with the notation $map\{min_1, max_1 \Rightarrow min_2, max_2\} : v$.

A *Growth Check* is when a node looks around its endpoint for a valid position where a new branch can grow to. This process first takes that branch's current direction and randomizes it a small amount using the tree's genetic Angle Randomness value. It then checks the distance to the nearest object in that direction. If no object is found, the new branch is said to have an infinite Growth Range and proceeds to the next step in the Growth Check process. If it does encounter an object then it keeps track of that direction and the resulting Growth Range and tries again by performing another Growth Check. Each growth attempt has a budget of 20 Growth Checks before the direction that gives the lowest distance gets used if no perfect direction is found.

2.4 Weather Simulation

Because of time constraints, I was unable to create a fully-localized weather simulation. My original plan was to simulate weather data like temperature, humidity, cloud cover, precipitation, and wind speed and have their data contained on different grid levels (see *Table 2: Planned Levels*). Unfortunately, I was not able to create such a sophisticated simulation but I have plans to do so in the future and will add that to this project when I do so.

With the time that I had, I created a simple weather simulation. It consists of only one weather point that is applied across the entire simulation space. The weather simulation system has its own time-of-day system which uses a daily temperature curve to calculate the temperature throughout the day.

The temperature curve is also offset by the current day of the year (basic season simulation). A base value for temperature, humidity, cloud cover, wind speed, and precipitation likeliness is calculated at the beginning of each new day using the previous day's values. Throughout the day, the simulation uses those base values along with the current temperature to compute the current value for each of those parameters. These values are written to the Data Structure through Grid Data instead of Cell Data, as originally planned because it is only one value for the entire simulation space.

3. Results & Discussion

To collect relevant data on the operation of the simulation, I measured key values from each separate part of the simulation. All measurements were taken on a desktop with an Intel® Core™ i5-10400F CPU at 2.90 GHz, an RTX 2070 Super GPU, 32 GB of RAM, and Windows 11. The world size was 1000 m x 1000 m, with 100,000 trees.

3.1 Timing Metrics Graphs

I ran the simulation in a standalone application for one simulation year and compiled the timing metrics in *Figure 2: Split Tick Time (ms)*, shown below.

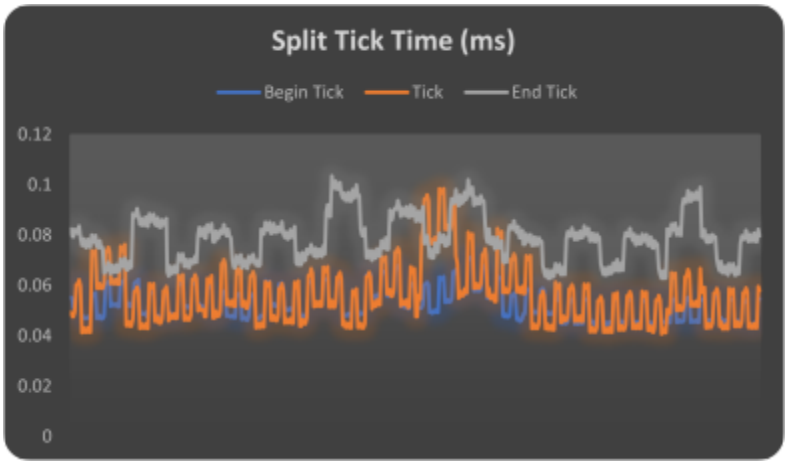

Figure 2: Split Tick Time (ms)

The total tick time is shown below in *Figure 3: Total Tick Time (ms)*.

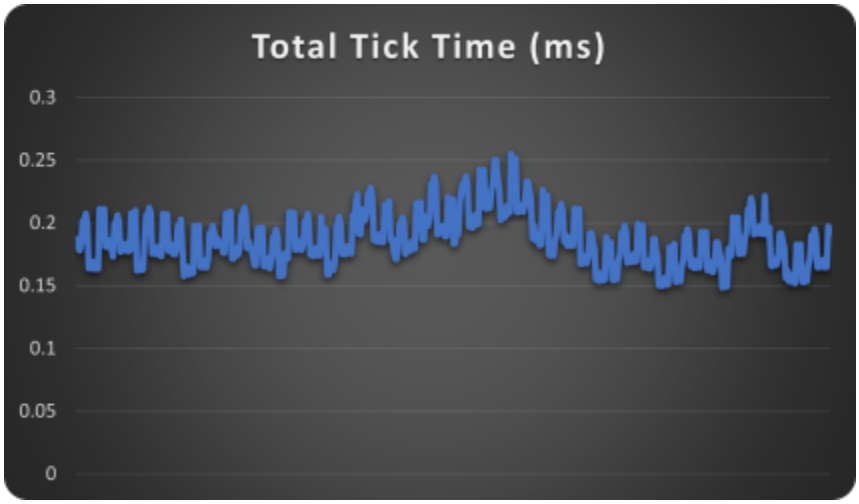

Figure 3: Total Tick Time (ms)

The timing metrics for Grid Data being read and written to the Data Structure are compiled in *Figure 4: Grid Read/Write Times (ms)*.

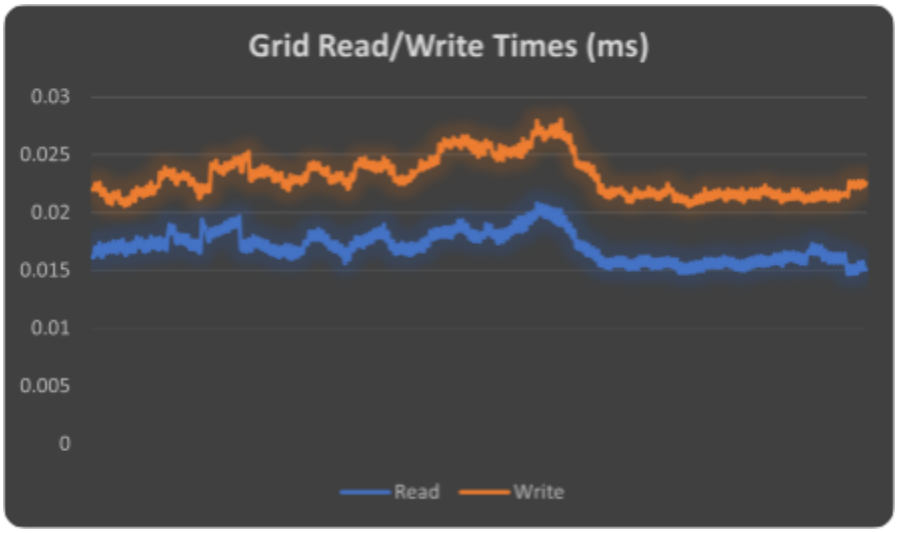

Figure 4: Grid Read/Write Times (ms)

The timing metrics for Cell Data being read and written to the Data Structure are compiled in *Figure 5: Cell Read/Write Times (ms)*.

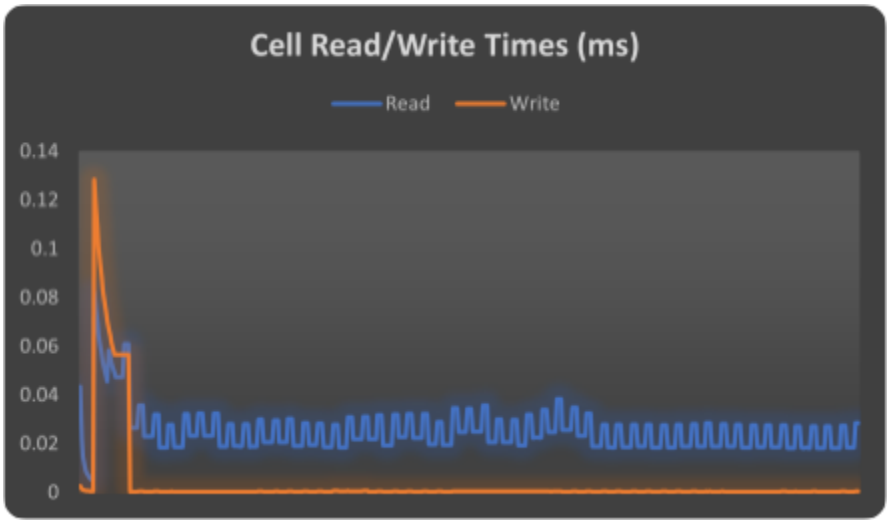

Figure 5: Cell Read/Write Times (ms)

These results are far above my expectations and I am very satisfied with the very high speeds that this simulation performs. My original goal was an overall tick time of around one second, but the result ended up being under a millisecond. This is significantly faster than expected.

3.2 Weather Graphs

I also compiled the data from the weather simulation into several graphs, shown below.

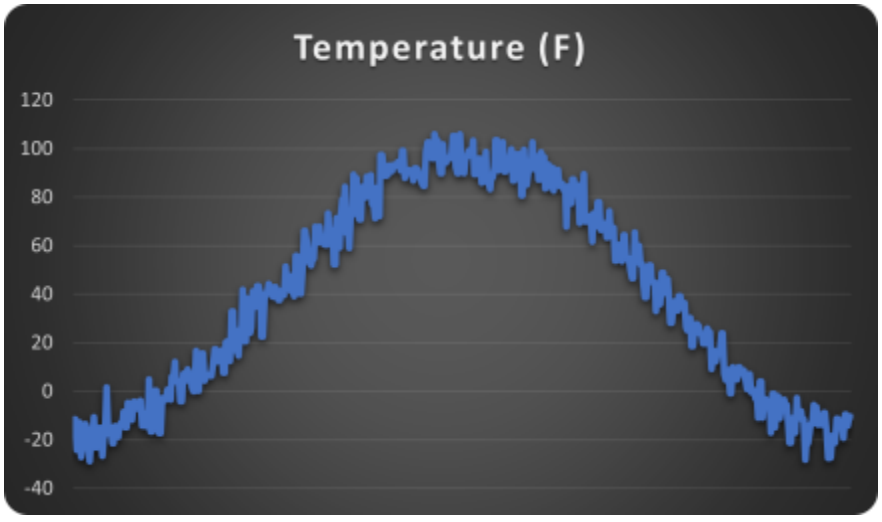

Figure 6: Temperature (F)

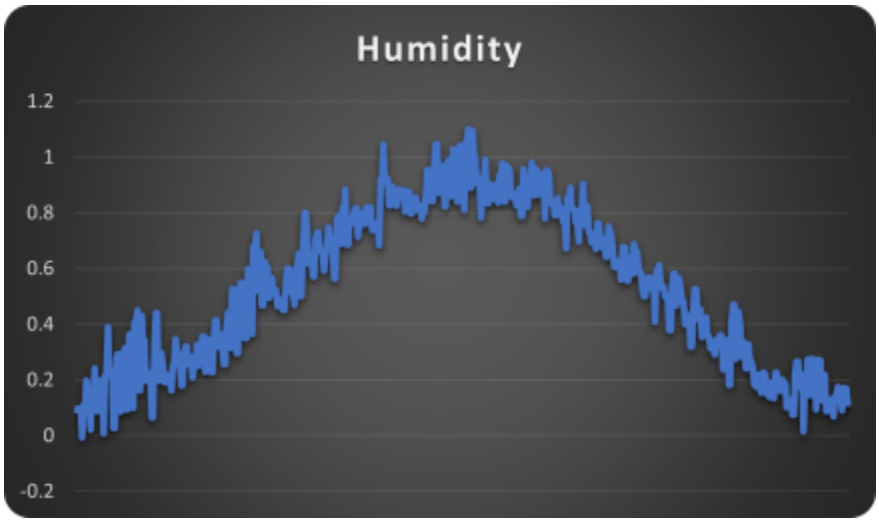

Figure 7: Humidity (relative)

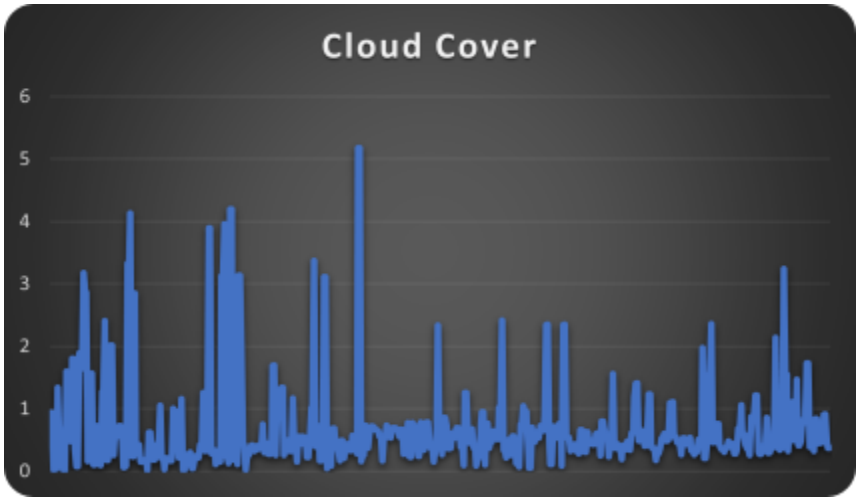

Figure 8: Cloud Cover

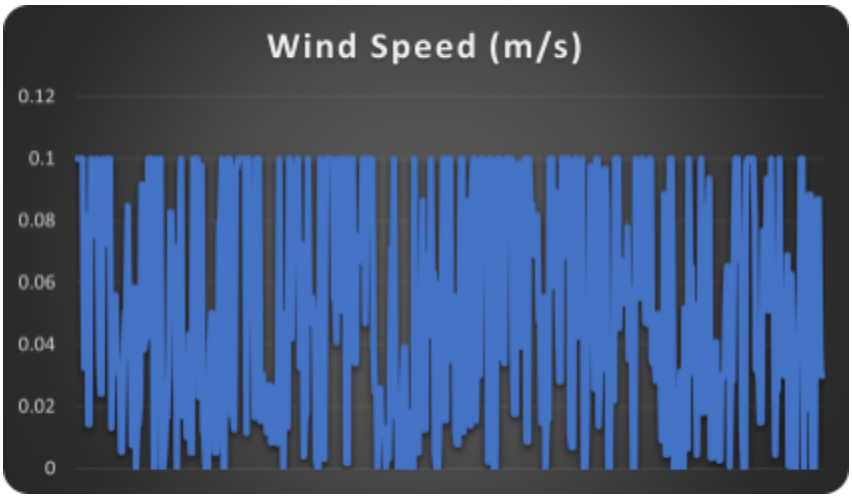

Figure 9: Wind Speed (m/s)

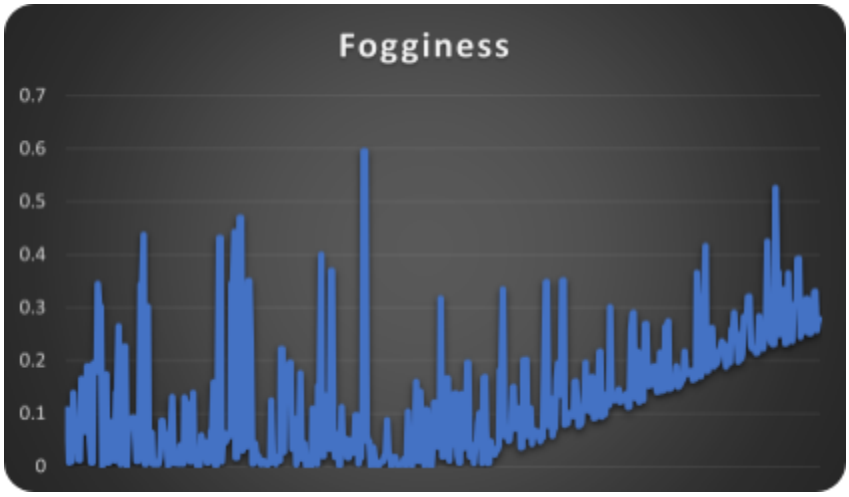

Figure 10: Fogginess (relative)

When these graphs are compared to those provided by climatestations.com, one can see a similarity in the results produced by my weather simulation. Even though my weather simulation was much more simplified than my original plan, it still managed to produce accurate enough results that, with some extra fine-tuning, could become even more accurate. The wind speed, cloud cover, and fogginess are very inaccurate while temperature and humidity are both quite accurate.

3.3 Tree Growth Images

Below are some images of trees grown in the simulation. These trees are fully grown and represent the species of tree being simulated here.

Figure 11: Single Tree

Figure 12: Example of heliotropism. The tree is boxed in and grows out of a hole in the side to reach the light.

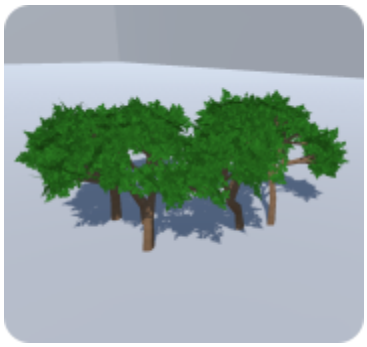

Figure 13: Small Cluster of Trees

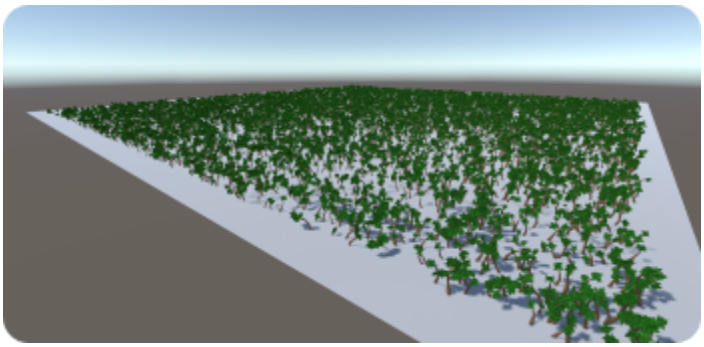

Figure 14: Large Forest of 10,000 Juvenile Trees

3.4 Water Simulation Images

I did not have the time to create a 3D visual representation of the water simulation so the only way I have to visualize it is through its texture. The height map used for the water simulation is shown below.

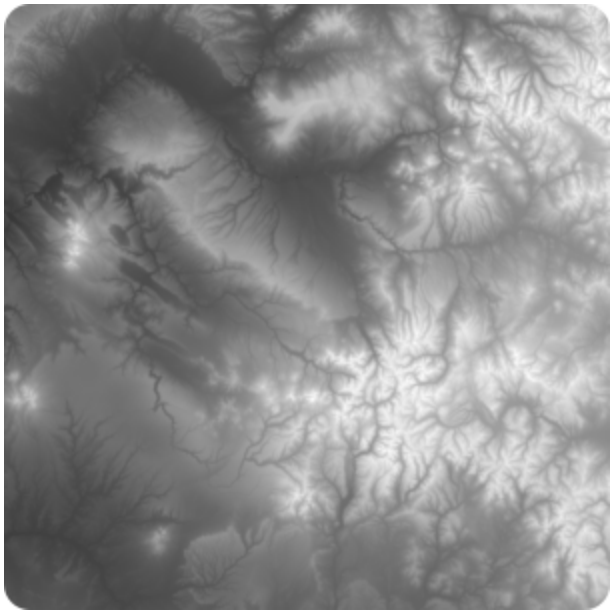

Figure 15: Heightmap

For the purposes of this demonstration, I placed a continuous water source at the center of the simulation space to ensure a steady flow of water. In an actual simulation, the water would be present in the form of rivers and lakes being continuously fed by a small source of water representative of a natural spring.

3.3.1 Surface Water

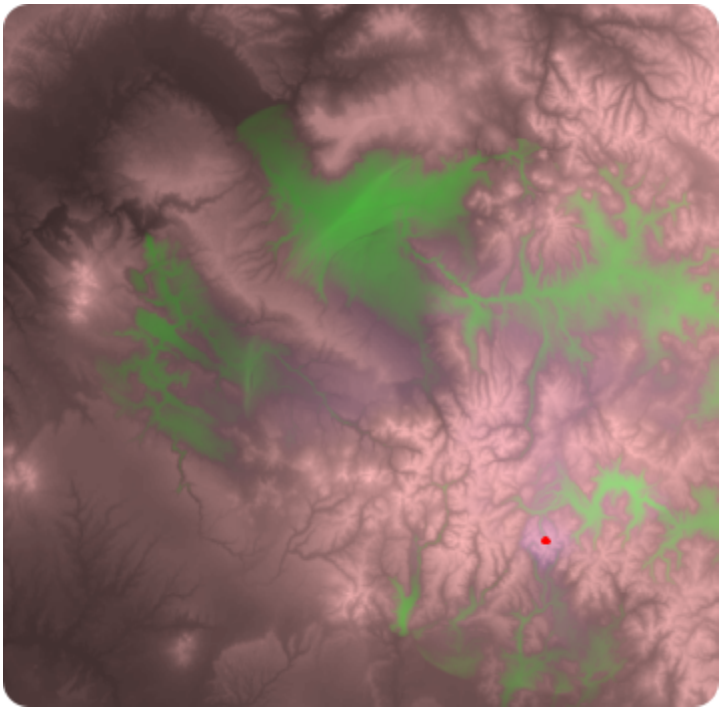

Figure 16: Water Height Composite. The pink shading is the terrain height. The green shading represents surface water. The blue shading represents soil water. The red dot represents the water source.

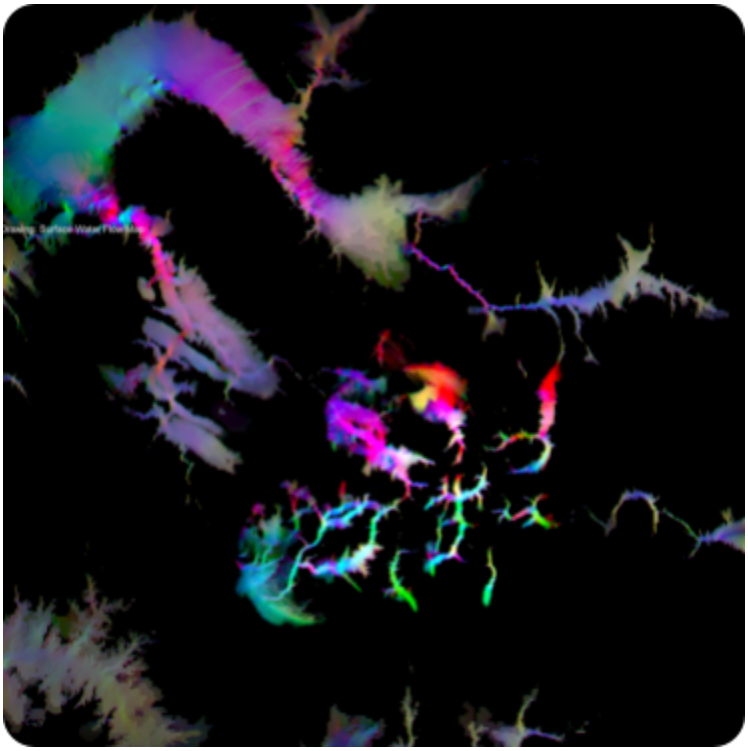

Figure 17: Water Flow (red = flow up, green = flow down, blue = flow left, alpha = flow right)

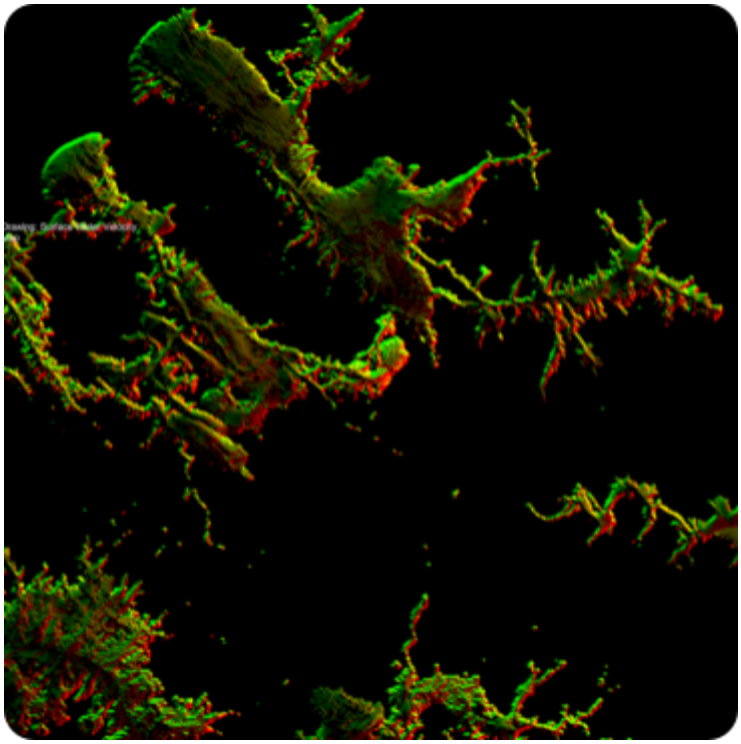

Figure 18: Water Velocity (red = x velocity, green = y velocity)

3.3.2 Soil Water

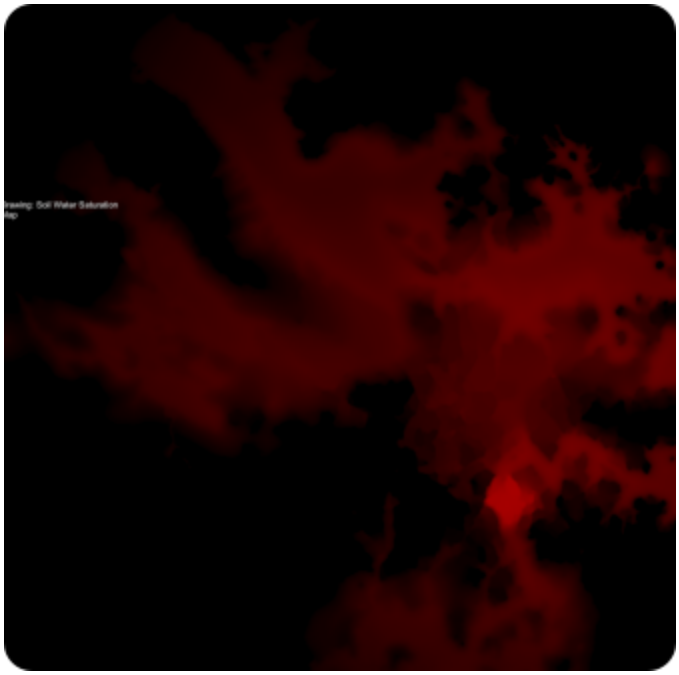

Figure 19: Soil Saturation

3.5 Discussion

This simulation runs significantly faster than I expected it would. This is in most part to the ultimately smaller size and quantity of trees but even when extrapolating these numbers to larger situations, this simulation could potentially run faster than my expectations even at my planned scale. My goal was to create a realistic simulation of a large-scale ecosystem that runs in real-time. I do want to continue working on this in the future as I have more plans to expand this project without the time constraints of this paper. I created a simulation that worked exceptionally well on a reasonably large scale of 1000 m x 1000 m. In other tests, I witnessed it working at 10 km x 10 km and even 50 km x 50 km. In each of those tests, the limitation was not the tick time but actually the initialization time, because of my grid implementation at the time. Because I was using a dense grid structure, that meant that every grid would have to be fully populated upon starting the simulation. This initialization step alone took around 30sec for the 50 km x 50 km simulation, but after that, it ran acceptably well with 3-4ms tick times. While my original goal was around 25 km x 25 km, I still don't feel that I accomplished that part of the goal because of the large initialization time. A next step to resolve this issue would be to use a sparse grid structure that allows for scaling up to such large simulation worlds.

When considering accuracy, I also believe that I accomplished my goal. The trees, while not very realistic-looking, do grow in a physically accurate and true-to-nature fashion. Furthermore, the water simulation provides an unprecedented amount of detail even at very large scales. The weather simulation, while being a makeshift placeholder until I find more time in the future to implement a proper one, is very stable and quite accurate when comparing temperature and humidity.

4. Conclusion

This project accomplished its goal of creating a large-scale, accurate ecosystem simulation that runs in real time. The entire simulation runs in under a millisecond, far exceeding my original goal for speed. The simulation is also physically accurate in some parts when simulating the weather and tree growth, meeting my goal of accuracy. Finally, it runs well above my original expectations for simulation size, albeit with a significant slowdown on initialization for very large scales, making scale the only part of this project that I feel did not meet my goals.

5. Acknowledgments

I would like to thank my teacher, Ms. Van't Slot, for her support and assistance throughout the process of creating this project and writing this paper. I also would like to thank my parents for helping keep me motivated throughout a long-term project like this one.

6. References

1. Makowski, M., Hädrich, T., Scheffczyk, J., Michels, D. L., Pirk, S., & Pałubicki, W. (2019, July). *Synthetic Silviculture: Multi-scale Modeling of Plant Ecosystems*. Synthetic silviculture: Multi-scale modeling of Plant Ecosystems. Retrieved November 14, 2022, from https://storage.googleapis.com/pirk.io/projects/synthetic_silviculture/index.html
2. Hart, J. C., Baker, B., & Michaelraj, J. (2003, February 5). *Structural simulation of tree growth and response*. The Visual Computer. Retrieved November 15, 2022, from https://sirius.cs.put.poznan.pl/~inf71284/pdf/structgrowth.pdf
3. de Reffye, P., Fourcaud, T., Houllier, F., Blaise, F., & Barthelemy, D. (1997, March 31). *A functional model of tree growth and Tree Architecture - Helsinki*. A Functional Model of Tree Growth and Tree Architecture. Retrieved November 15, 2022, from https://helda.helsinki.fi/bitstream/handle/1975/8529/silva_1997_31_3_(6)_reffye.p.pdf?sequence=3
4. Brusee A., Xavier M. (2020, January 14) *Fast K-Nearest Neighbour Library for Unity DOT*S. Github. Retrieved 23 January 2023, from https://github.com/ArthurBrussee/KNN
5. Volčini, V., Crystal H. (2021, September 23) *Fast KDTree for Unity, with thread-safe querying*. Github. Retrieved 23 January 2023, from https://github.com/viliwonka/KDTree
6. Various Authors. (2015). *Quadtree*. Wikipedia, the free encyclopedia. Retrieved 23 January 2023, from https://en.wikipedia.org/wiki/Quadtree.
7. Kebede, A. B. (2019, December 13). *Influence of Soil Type in stream Flow and Runoff Modeled for the Upper Didessa Catchment Southwest Ethiopia using SWAT model.* Research Gate. Retrieved November 21, 2022, from https://www.researchgate.net/publication/338032125_INFLUENCE_OF_SOIL_TYPE_IN_STREAM_FLOW_AND_RUNOFF_MODELED_FOR_THE_UPPER_DIDESSA_CATCHMENT_SOUTHWEST_ETHIOPIA_USING_SWAT_MODEL
8. Ball, J. (Unknown Date). *Soil and Water Relationships*. Noble Research Institute. Retrieved November 22, 2022, from https://www.noble.org/regenerative-agriculture/soil/soil-and-water-relationships/

9. Harris, M. J., Baxter, W. V., Scheuermann, T., & Lastra, A. (2003). *Simulation of Cloud Dynamics on Graphics Hardware*. Tu Wein. Retrieved November 28, 2022, from https://users.cg.tuwien.ac.at/bruckner/ss2004/seminar/A3b/Harris2003%20-%20Simulation%20of%20Cloud%20Dynamics%20on%20Graphics%20Hardware.pdf
10. Lauer, A., & Hamilton, K. (2013). *Simulating clouds with global climate models: A comparison of CMIP5 results with CMIP3 and satellite data*. *Journal of Climate*, *26*(11), 3823–3845. https://doi.org/10.1175/jcli-d-12-00451.1
11. Marian Kleineberg. (2023). *Wave Function Collapse*. GitHub. https://github.com/marian42/wavefunctioncollapse
12. Müller, M., Charypar , D., & Gross, M. (2003). *Particle-Based Fluid Simulation for Interactive Applications*. Eurographics. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1739fd145ef1d327ab301cacc017af2a87f33086
13. Šťava, O., Beneš, B., Kˇrivánek, J., & Brisbin, M. (2008). *Interactive terrain modeling using hydraulic erosion*. Eurographics. http://cgg.mff.cuni.cz/~jaroslav/papers/2008-sca-erosim/2008-sca-erosiom-fin.pdf
14. Stevens, B., C. Acquistapace, A. Hansen, et. al., 2020: The added value of large-eddy and storm-resolving models for simulating clouds and precipitation. J. Meteor. Soc. Japan, 98, 395–435, doi:10.2151/jmsj.2020-021.

7. Author Bio

Alexander Irausquin-Petit - 11th-grade student at the Dwight-Englewood School. I chose this project because of my passion for the environment. I am currently making a video game about climate change and climate disaster and found this subject area interesting in order to further my knowledge of environmental science and ecosystems. This project is also meaningful to me because I have been wanting to make something like this for a long time but never had the opportunity to do so in a proper research environment until now.

8. Project Source

Project source code can be found on this project's GitHub page: https://github.com/AIP21/Ecosystem-Sim-AIRS-Project