Thesis Proposal: Garden of Eden

William B. Jackson

February 5, 2014

1 (really bad) Abstract

Garden of Eden (GoE) is an exercise in procedural generation of lifelike worlds. Using parallelized computations, it randomly generates a forest scene of realistically shaped and proportioned asymmetric trees on top of a basic topographical map. This map is then rendered in 3D, with support for user navigation. The end result of this project is a sort of game: a static rendering of a natural environment, exploring how humans find visual pleasure and meaning in virtual environments. The passive interaction of the user is vital to this simulation, as it reflects how one would observe a natural environment; ideally, the simulation would evoke similar reactions to the environment GoE seeks to emulate, thus exploring the concept of a natural, the distinction between real and virtual, and one's own sense of place.

2 Related Work

A significant body of work relating to natural feature generation exists in the video game industry. Minecraft[3], for example, uses advanced terrain and feature generation techniques to procedurally create such natural features as mountain ranges, cave systems, chasms, coasts, and flora biomes, as well as artifical features such as temples, villages, and mine shafts. It achieves truely impressive scenes using simplified graphics, resulting in a stylised look rather than striving for verisimillitude. Another game worth mentioning in this context is Proteus[1], a small-scale "game of audio-visual exploration and discovery" that allows users to explore a highly-stylized generated island. There are many other games that use feature generation techniques, either as the creative force of the game world or to fill in between created features; I mention these two primarily because, through both gameplay and simple, yet awesome, natural scenery, they served as inspiration for this project.

3 Terrain

3.1 Midpoint Displacement

The Midpoint Displacement algorithm, also known as the Diamond-Square algorithm, is an easy way to generate surface height maps for terrain. The algorithm is as follows[2]:

```
Loop while quadrilateral side is at least 1
    // Square step
    For each discrete square of given quadrilateral side length
        Get midpoint of square
        Calculate random offset as a random number in a specified range times a scale
        Midpoint becomes average of square's corners plus random offset

    // Diamond step
    For each discrete diamond defined by the square step's calculated midpoints
```

```
        Get midpoint of diamond
        Calculate random offset as a random number in a specified range times a scale
        Midpoint becomes average of diamond's corners plus random offset

    Decrease scale by a constant fraction
    Half quadrilateral side
```

The algorithm is works well to generate fractal surfaces, and allows for a decent amount of control over the percieved "roughness" of the resultant surface in how one defines the scale factor. However, it is fairly slow, and there are also well-documented artifacts, notably for steeper surfaces. These factors can be mitigated somewhat with careful selection of constants, discussed below.

3.2 Feature Extraction

3.3 Gaussian Filter

4 Trees

4.1 L-Systems

An L-System, as defined in [4] describes a recursive substitution system to build increasingly complex strings. A system consists of an initial value or set of values, and a series of rules used to recursively replace sections of the initial set. The set consists of productions, or operations that, when combined with a rule, represent a certain kind of predictable growth. Productions can, for example, represent a single branching structure, or trunk growth. As such, L-Systems are an excellent way to represent the growth of trees, as they give a simple "seed" rules by which it can branch in a biologically feasable way into a much larger system. As L-Systems consist of self-replacement rules, they tend to form fractal systems, which are "self-similar" at different levels of detail. In an L-System, this self-similarity at smaller levels of detail comes from the recursion depth, which replaces productions at small levels with smaller versions of the designs seen at a larger scale.

4.1.1 Productions

A production defines a particular recursive structure. In combination with its replacement rule, it acts similarly to a replacement function; a single production can be replaced by a set of other productions, including itself, leading to interesting fractal shapes. In this project, a production does not represent any physical shape, and is not drawn graphically; instead, it's replacement rule can contain graphical information, allowing a production to represent a specific physical occurance. A production can, for example, represent a bijunction branching, or a bending of a branch due to biological influences, or the thickening of the trunk due to age.

4.1.2 Parametric L-Systems

A parametric L-System contains parametric productions, or, more simply, producitons that take parameters. Production rules change to accomidate parameter logic, matching, for example, only if a production has a "time" greater than 5. In this way, productions representing specific growth actions can have significantly different geometry while still having the same basic structure. A branch production, for example, might take an angle parameter, which would make larger branches diverge less dramaticaly than twigs.

The below L-System can serve as an example, and is used as the L-System module's tester in `./test/lsys\_console.html`.

ω : B(2)A(4,4)
p_1 : A(x,y) : $y \leq 3$ → A(x*2, x+y)
p_2 : A(x,y) : $y > 3$ → B(x)A(x/y, 0)
p_3 : B(x) : $x < 1$ → C
p_4 : B(x) : $x \geq 1$ → B(x-1)

In this example, ω gives the initial productions with parametric values. The rules have changed both to check the values of the parameters, and replace accordingly, and to modify the parameters in the replacement step. In this way, the productions will have different parameters at each level of recursion, reflecting changing growth at more depth.

4.1.3 Turtle Graphics

As stated before, productions give no graphical information as to how a tree is drawn. Thus, in addition to productions, an L-System must implement a system of graphical information. In order to create branching structures, LOGO-style turtle graphics are used to draw on the HTML5 canvas. Invented as a simple graphics drawer for educational purposes, turtle graphics consist of a set of commands that direct the movement of a "turtle", or a drawing point. In 3D space, three dimensions of orientation are controlled by issuing commands with a specified radial change; drawing occurs by moving the turtle forward. The turtle commands implemented are as follows:

Command	Symbol	Argument	Description
_F	F	time	Moves turtle forward by specified time value (turtle rate is constant), drawing a line (a cylinder primitive) from its previous location to its new one.
_f	f	time	Moves turtle forward by specified time value, drawing no line between positions.
_pitch	&	radians	Rotate turtle about its own x-axis by specified amount of radians.
_yaw	+	radians	Rotate turtle about its own y-axis by specified amount of radians.
_roll	/	radians	Rotate turtle about its own z-axis by specified amount of radians.
_set	!	width	Sets the drawn line (3D cylinder) to the specified width (diameter).
_push	[	none	Pushes current turtle position and orientation into a stack of previous states.
_pop	]	none	Pops current turtle position and orientation from a stack of previous states, resetting its current state to the new one.

The above commands are included in production replacement rules, and basically define what a production draws. In this way, a production might be used not just to define where a branch or other such growth action occurs, but also to draw the branch upon its replacement with turtle graphics commands. The amount of turtle graphics commands required to draw an appropriately detailed tree is incredibly large; the exact "turtle string", or set of commands, used to draw one tree is printed to the console in `./test/tree.html`.

4.2 Biological Considerations

As with most graphical projects, the generation of trees must strike a balance between emulating actual biological methods and using efficient computational design. In this section, I will discuss some of the

biological phenomenon modeled in the creation of trees, how they were modeled, and the effect that they have on tree growth.

4.2.1 Discussion of Assumptions

Some assumptions were made to simplify and speed up the generation of tree structures:

1. All branches are straight. In reality, a tree branch is frequently curved, often noticeably. However, this project uses straight branches exclusively, as it allows for the rendering of a branch with a single, simple, cylindrical primitive. Future work could use warped line segments to represent a branch, extruded to a width specified by the turtle.

2. All trees in this project are of a similar age, growing at a similar speed. In a real forest, the actual ages of the trees tend to vary, even while the "age of the forest" grows roughly together. Furthermore, due to a multitude of growth factors, trees of similar age might grow at drastically different speeds. The end result is that in a real forest, trees might be noticeably different in size. This project will not seek to emulate that, though it will result in a fairly noticeable visual artefact. Future extensions to the `Forest` module might consider tree age in the planting and growth steps.

3. "Twigs", or branches of an arbitrarily small size, are not to be rendered. While this is possible for users of the software package, as it involves merely increasing the depth to which the L-System is run, it results in an exponentially larger amount of primitives and a noticeably laggier simulation.

4.2.2 Bijunctions

A cursory glance out the window will reveal that the vast majority of branching junctions split the root branch into two smaller branches. The reasons for this will not be discussed in this paper, however, for visual integrity, this rule will be maintained. A *branching junction* here is defined as any place, usually represented in the L-System as a single production, where a single branch divides and deviates into a set of smaller child branches. Such a junction that splits into exactly two smaller branches will be defined as a *bijunction*.

The included L-System `TernaryTree` uses a single branching junction, modeled with the production `A()`, to represent a *trijunction*, defined as a branching junction that splits into exactly three children[4]. There are benefits to using trijunctions. A trijunction is the smallest possible branching junction that fills a volumetric space, where a bijunction instead is isolated to a single plane. Thus, trijunctions rapidly result in a "fuller" looking tree, where bijunctions frequently have the effect of looking planar, or "flat". Trijunctions also allow the branch count to grow cubically, thus requiring shallower recursion to "fill out" a decent-looking tree. This allows for more flexibility in tree appearance: as the L-System used for `TernaryTree` and `RandomTree` grows the size of the tree while constructing branches, a tree with trijunctions will appear more detailed at a smaller sizes.

Thus, careful consideration must be taken to "filling out" a bijunction-exclusive tree. One approach is to use *pseudo-trijunctions*, a branching junction consisting of two stacked bijunctions. These junctions, where a single branch will split off immediately before a branch fork, are fairly common in nature, and allow for trijunction behavior without violating the bijunction rule. For further discussion of the methods used to fill out a bijunction tree, see the below section on Probabilistic L-Systems.

4.2.3 Tropism

4.2.4 Growth Density

4.2.5 Elevation and Gradation

4.3 Randomization

In nature, the growth and configuration of trees is a result of both biology and environment, and modeling tree geometry should consider their affects. This paper will not seek to exactly model the growth

details, though it will consider some of them in the design of randomization algorithms. For this paper, it is sufficient to say that some growth factors are internal to the tree's genetics, information physically encoded and manifesting in commonalities between trees of the same species, whereas some are external to the tree, resulting in commonalities between trees of different species in the same growth conditions[5]. Internal information is reasonably represented by the L-System, which specifies deterministic growth patterns for a specific tree. Taken alone, other influence, trees generated by a single L-System display little variation, to the point of being identical. I suggest that the codified rules of L-Systems be considered a heuristic for genetic patters of tree growth.

The tree's local environment also plays a large hand in growth and configuration. As described in the above section, such factors external to the tree as Tropism, Growth Density, and Elevation affect the presence of trees, the rate of growth, branching, and the general tendency to grow in a specific direction. Weather can affect the tree, too: heavy snow might bow a mighty branch, or a bolt of lightning might split a trunk. Temperature patterns can affect the growth in subtle ways, as well.

The process described below are an imitation of the above factors, seeking to replicate with selective randomness what a perfectly deterministic, physical system would otherwise be able to reproduce. To the degree that they are based on actual growth and environmental factors, they will be justified; many, however, are simply heuristic hacks that result in a realistic geometry.

4.3.1 Parameter variation

4.3.2 Probabilistic L-Systems

Thus far, L-Systems depicted have had replacement rules that match evenly to all of the productions defined. This results in static branching structure, as the geometry is deterministic and limited in configuration. However, using a `Rule`'s `condition` parameter, productions can be replaced randomly with one of a set of options. The `condition` parameter could, for example, switch the branching style as recursion depth increases, leading to a different configuration for branches than what is used for the trunk.

Using a provided random value from Javascript's built-in `Math.random()`, `condition` can also be replaced probabilistically. This can be used to represent several styles of branching. For example, while trees only ever bijunct, trijunction-like branches can occur by closely stacking bijunctions. This style of branching can thus be interspersed randomly with the usual bijunction style.

The L-System used to create the trees in Garden of Eden use this style of probabilistic branching. The branching production, 'A', can be any of the following styles:

- Normal bijunction: Branch continues along current vector, with another branch splitting off at a small acute angle.
- Fork bijunction: Branch splits into two, with both diverging at a small acute angle from the previous vector.
- Pseudo-trijunction: One branch from a fork trijunction is moved down the parent branch, resulting in a normal bijunction stacked below a fork bijunction. See figure 1.

4.3.3 Gaussian Density Management

A commonly measured property of forests are the density of tree growth. As larger trees tend to block sunlight from trees growing beneath them, in general, the older a forest is, the further spaced its trees. Furthermore, as there is a "sweet spot" of growth away from an older tree where light isn't blocked and roots aren't in conflict over nutrients in the soil, trees, especially of a similar age, tend to be spaced fairly evenly. To model this, a algorithm that roughly and indeterministically manages density is preferred. The algorithm implemented uses a proximity detection algorithm[6] with the central limit theorem to filter out trees that are packed too densely. The algorithm is as follows:

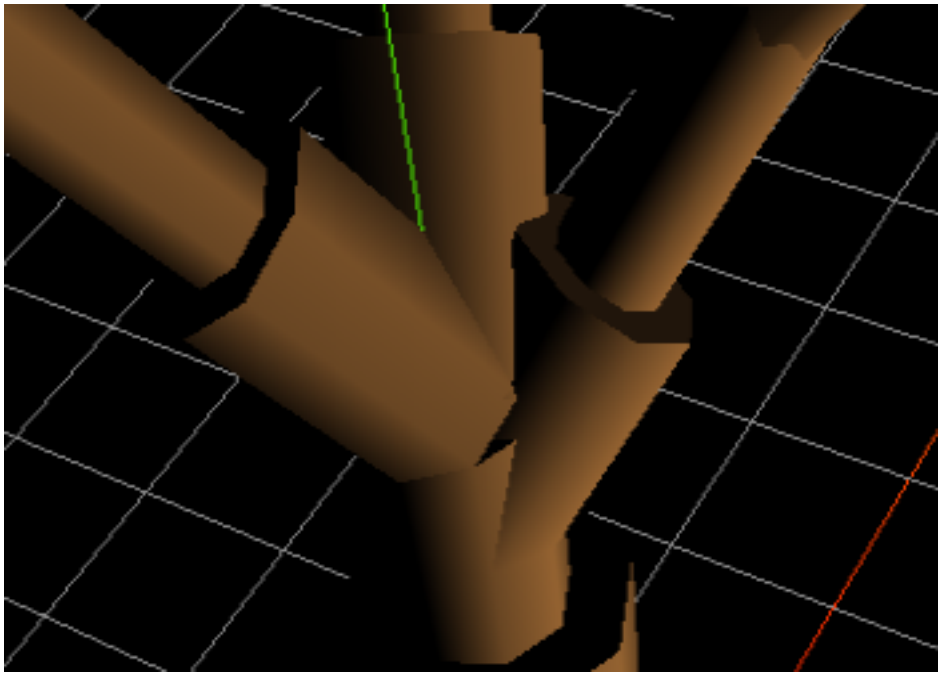

Figure 1: The result of a pseudo-trijunction production

```
Gaussian-Density-Management(density)
    // Spam step
    Plant at a random location a number of trees that, given the size of the terrain,
        results in double the desired density

    // Proximity detection step
    Create a grid where each square is half the distance between trees, calculated
        from the density
    Seed grid with first tree by mapping the tree's location to the grid and adding
        it to that spot
    For every subsequent tree t_dest,
        Map tree location to grid
        If there are any trees mapped to that location or any of its compass neighbors,
            For each neighboring tree t_src,
                Calculate the Euclidean distance between t_src and t_dest
                Construct a pair of t_src and t_dest, including the calculated
                    distance
                Add pair to a list of pairs

    // Filter step
    For each pair in the pair list
        MAP DISTANCE TO NORMAL CURVE TO GET PROBABILITY??
        FILTER USING THAT PROBABILITY??
```

4.3.4 Gradation Detection

4.3.5 Local and Global Tropism

5 Implementation

5.1 Top Level

5.2 Terrain Implementation

5.3 terrain_gen.js

5.4 Tree Implementation

The creation of a tree begins by creating a rule set for the L-System. A few of these are defined in `./js/app/lsys_rules.js`, which can be used as an example for users wishing to create their own L-Systems. This rule set is used to create an L-System object, which will be used to generate the tree's geometry. A turtle must also be constructed, and must be given a set of three.js objects in order to draw to the HTML5 canvas. Then, by calling the L-System object's `build()` method, the L-System will recursively construct a tree, storing it in its own `property` public variable. This is then passed into the turtle object's `run()` method, which will draw the constructed L-System as a single hierarchical object onto the HTML5 canvas.

5.4.1 turtle_graphics.js

The turtle graphics module consists of a turtle object, `Turtle`, a set of turtle commands to control drawing, and a `run` method, which, given a list of turtle commands, executes them in order from the current position.

- `Turtle(scene, material, radius)`: Turtle object constructor. Creates the turtle drawer, initializing it at the world origin oriented in the $+z$ direction. Turtle properties are as follows:
  1. `rate`: Defaults to 1. Rate at which turtle travels when moving forward, multiplied by the forward command's `time` parameter to get the total distance traveled.
  2. `width`: "Width" of the drawn line. Defaults to 0.25, can be passed in as an argument. Actual value is the radius of the cylinder primitive created by the turtle.
  3. `scene`: Three.js scene. Must be passed in. Graphics drawn in `run()` are added to this as a single, hierarchical object.
  4. `material`: Three.js material. Must be passed in. Defines the material of the drawn cylinder lines.
- Turtle commands: The turtle commands (`_F()`, `_f()`, `_pitch()`, `_yaw()`, `_roll()`, `_push()`, `_pop()`, and `_set()`) are all methods that specify commands to some turtle object. To use, they are attached to a `Turtle.Action` object along with the value of their argument, and then passed as a list as the argument to `run()`. Each one specifies a action relative to the turtle drawer's current position and orientation. Details of the actions are given in table 1. In general, the rotation actions work by multiplying a specific rotation matrix to the turtle object's internal orientation matrix. The use of a rotation matrix allows for relative transforms, as opposed to Euler coordinates, which require specific ordering of rotations which can lead to errors. The forward movement controls drawing; it does this by creating a cylinder of the current width and given distance (the actual given is a time parameter, which is multiplied by the turtle's `rate` to get the cylinder's length) and moving it such that it begins at the turtle's starting position and is oriented in the same direction of the turtle. Then, the turtle's position is incremented by the given distance. `_push()` and `_pop()` save the turtle's state in a basic object and store or retrieve it from an internal stack.

- `run()`: The run commands executes a given turtle string. Looping through the string in order, it run's the function's built-in `call()` function, specifying `this` as the calling object and `Turtle.Action.args` as the arguments. This binds the current turtle object to the action, allowing the turtle command to affect the turtle object's internal variables.

5.4.2 lsys.js

`lsys.js` gives the object used to describe and create an L-System of arbitrary productions and rule set. Relevant object, including `LSystem.Production` and `LSystem.RuleSet`, are defined, and the engine to run recursion to a specific depth is included in a method of the base `LSystem` object. The L-System requires both a rule set and an initial value to run, both of which are described below.

- `LSystem`: Object containing the L-System and it's rule set. The only argument is `rule_table`, the rule set for the L-System.
- `LSystem.Production`: Object containing production information. Productions, with the added information of their rule sets, represent a specific type of growth, but are not actually drawn by the turtle graphics wrapper. Production arguments are as follows:
  1. `id`: ID of the production, similar to function name. For example, in the production A(1,4), the id is "A".
  2. `args`: List of argument values. This is only assigned for initial values; the `inject` function is used to dynamically assign these values, as the resultant value is usually some function of the previous value, defined in the rule set.
  3. `inject_args`: Function used to dynamically assign arguments from within the rule set. Arguments after the replacement step are usually a function of their previous values; the width of branches, for example, might decrease by a constant scale at each level of branching. The function must take `args`, the previous production's arguments, and `consts`, a dictionary of constants defined in the rule set. It must then assign `this.args` to a list of the argument values.
- `LSystem.RuleSet` and `LSystem.Rule`: `RuleSet` and `Rule` are used to define the logic of the L-System. `RuleSet` describes the entire rule system, whereas Rule defines a single rule for a specific production and condition. The arguments for the `RuleSet` constructor are as follows:
  1. `consts`: Dictionary of defined constants, such as width reduction ratios, used in calculating the argument values at replacement.
  2. `initial`: Initial production list values for the system.
  3. `rules`: List of rule objects for this system. See `lsys_rule.js` and its below explanation for more details on `LSystem.RuleSet` construction.

  The arguments for the `Rule` constructor are as follows:
  1. `id`: ID corresponding to the production this rule affects.
  2. `condition`: Condition function to the parametric term. Function takes as argument the production object and returns a boolean. Allows the rule to match on the value of the production's parameter.
  3. `output`: List of production and turtle action objects to replace the matched production with upon recursion.

  More details about the construction of a `RuleSet` can be seen in the documentation for `lsys_rule.js`.

- `build()`: Launcher function for the recursive L-System construction. Calls internal recursion function on each element in the initial system; thus, it requires that the `system` variable is set to a list of initial values. The internal recursion runs for a single object in the system list, matching it to the rule set and replacing it with the appropriate output string. It then recursively calls itself on each element of the output array, to a specified `MAX_DEPTH` value. Optional argument `debug` is a boolean that toggles printing of the constructed system after construction.

- `checkRule()`: Function that checks a given element of the system against the system's rule set, returning the appropriate output. Does a linear search through the list of rules, halting when a match occurs. Matches require that both the rule id and the production id match, as well as the rule's `condition` function returning a `true`. The rule's output is then cloned to avoid problems with Javascript's singleton objects affecting subsequent production values. The arguments are then injected using the production's provided `inject_args` function, which allows dynamic evaluation of the arguments based on the previous production. The output is then returned. If no match is found, the initial production is returned unchanged.

- `printSystem()`: Used for debugging. Pretty prints the current value of the L-System's `system` variable, in form `id(args...)`. Used when the `debug` flag is turned on in `build()`.

5.4.3 lsys_rule.js

`lsys_rule.js` provides a convenient package of L-System rules used in *The Algorithmic Beauty of Plants*. The rules found in it are used to create the trees in the final Garden of Eden project, but can also serve as examples to users wishing to generate their own L-Systems. The structure for a rule set is given in the above description of `LSystem.RuleSet`.

- `HondaTree`: Creates an object that inherits from `LSystem.RuleSet`. Description of a Honda L-System comes from *SOURCE WEE*. The tree is of a constant height, but branches with increasing detail at higher levels of recursion. Defined constants allow for a decent amount of variation on what is otherwise a visually highly symmetrical tree.

- `TernaryTree`: Creates an object that inherits from `LSystem.RuleSet`. Description of a Ternary L-System comes from *SOURCE WEE*. By using L-System rules that match to turtle commands, this tree actually grows both larger and more detailed at higher levels of recursion, which more closely resembles the growth of a tree. There is only one branching rule, which erroneously uses a trijunction at each branching node. The generated tree is visually quite complex, but has some symmetrical artefacts visible from specific angles. Varying constants allows for a large amount of variation on this tree structure.

6 Runtime

6.1 LList Algorithm

6.1.1 Profile

6.1.2 Asymptotic Behavior

6.2 Tree Search Algorithm

6.2.1 Profile

6.2.2 Asymptotic Behavior

6.3 Three.js Considerations

References

[1] James Arvo, David Kirk, and Apollo Computer. Modeling plants with environment-sensitive automata. In *In Proceedings of Ausgraph88*, pages 27–33, 1988.

[2] Jules Bloomenthal. Modeling the mighty maple. *SIGGRAPH Comput. Graph.*, 19(3):305–311, July 1985.

[3] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. *SIGGRAPH Comput. Graph.*, 22(4):151–158, June 1988.

[4] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, June 1982.

[5] Houssam Hnaidi, Eric Gurin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. *Computer Graphics Forum*, 29(7):2179–2186, 2010.

[6] Ed Key and David Kanaga. http://www.visitproteus.com/.

[7] Paul Martz ⟨martz@frii.com⟩. Generating random fractal terrain. *Game Programmer*, 1996, 1997.

[8] B. Mandelbrot and J. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM Review*, 10(4):422–437, 1968.

[9] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23(3):41–50, July 1989.

[10] Teong Joo Ong, Ryan Saunders, John Keyser, and John J. Leggett. Terrain generation using genetic algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1463–1470, New York, NY, USA, 2005. ACM.

[11] Markus Persson and Notch Software AB. https://www.minecraft.net/.

[12] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 2nd edition, 1996.

[13] Michael G. Ryan and Barbara J. Yoder. Hydraulic limits to tree height and tree growth. *BioScience*, 47(4):pp. 235–242, 1997.

[14] Mark Sandland <Kramii>. Re: Find all points within a certain distance of each other? http://programmers.stackexchange.com/a/129909, May 2014.

[15] Naokazu Yokoya, Kazuhiko Yamamoto, and Noboru Funakubo. Fractal-based analysis and interpolation of 3d natural surface shapes and their application to terrain modeling. *Computer Vision, Graphics, and Image Processing*, 46(3):284 – 302, 1989.