Procedurally Generated Surfaces Using L-Systems

Ryan Fitzpatrick

Abstract

The purpose of this project was to investigate whether L-Systems could be a feasible method for procedural landscape generation. Procedural environment generation is becoming a more common task in modern video games and simulations, especially when trying to render extremely large surface areas (ie. planets, solar systems, a universe.) The idea was that using the recursive nature of L-Systems, and their natural evolving level of detail, would provide a single simple method for generating unique surfaces across varying levels of detail. It was also hoped that the recursive structure of L-Systems would allow us to define an extremely large area, such as a planet, but focus landscape generation to specific nodes in the L-System tree, thus saving memory while also having any other part of the planet easily generated in only a couple of steps. The main idea for generating the surfaces was to use a form of parametric L-System that recursively divides a surface area into quadtrees. Each quadtree would have an associated height, thus allowing us to use any level of the L-System as a heightmap for the target surface. Overall the results were quite promising, and these parametric quadtree L-Systems may provide a relatively simple method for rendering large surfaces in varying levels of detail.

L-Systems

Before investigating the types of grammars used in this project, it is important to understand specifically what an L-System is. An L-System is a formal grammar, and is defined as the triple (V, ω, P), where V is the set of symbols recognized by the grammar, ω is the initial state of the system, and P is a set of production rules. All production rules in P map a single symbol in V to a new string of symbols, where all these new symbols must also be in V. ω must also be a string containing only symbols found in V. Here is a classic example:

V: {A, B}
ω: A
P: {(A → AB), (B → A)}

which produces:

(*n* = 0): A
(*n* = 1): AB
(*n* = 2): ABA
(*n* = 3): ABAAB
(*n* = 4): ABAABABA
(*n* = 5): ABAABABAABAAB
(*n* = 6): ABAABABAABAABABAABABA

(*n* = 7): ABAABABAABAABABAABABAABAABABAABAAB

Note that the system becomes more and more detailed as *n* increases. These are called generations, and are used to track to evolution of the L-System string. It is also important to note that at each generation, every symbol in the previous string is replaced by new symbols by applying the production rules found in P. Because of this, we can view the system evolution in a tree structure, like so:

```
n=0:              A
                 / \
n=1:            A   B
               / |    \
n=2:          A  B     A
             /|  |     | \
n=3:        A B  A     A  B
           /| |  |\    |\  \
n=4:      A B A A B   A B A
```

A couple key things to note here: first is the similarities between each generation along with a consistent overall structure, and second is the recursive structure of the L-System tree. Both are useful properties of L-Systems that we will exploit in this project. Another important consideration is that L-Systems can also be extended. Two common extensions used in our L-System model are the concepts of parametric and stochastic L-Systems. Parametric L-Systems are not surprisingly L-System grammars whose production rules contain parameters, like in this example:

V: {A, B}
ω: A(0, 1)
P: {(A(x, y) → A(x + 1, y)B(x, x)), (B(x, y) → A(x + y, y + 2)}

which produces:

(*n* = 0): A(0, 1)
(*n* = 1): A(1, 1)B(0, 0)
(*n* = 2): A(2, 1)B(1, 1)A(0, 2)
(*n* = 3): A(3, 1)B(2, 2)A(2, 3)A(1, 2)B(0, 0)
(*n* = 4): A(4, 1)B(3, 3)A(4, 4)A(3, 3)B(2, 2)A(2, 2)B(1, 1)A(0, 2)

Parametric L-Systems allow us to store associated data with every generation, and it allows for the data to evolve itself, and control the evolution of the system. Stochastic L-Systems allow for multiple production rules for a given symbol, along with a probability which determines which rule is applied. This allows for a wide variety of strings to be produced from the same system. Here is an example:

V: {A, B}
ω: A
P: {(A(0.5) → AB), (A(0.5) → BB), (B(0.75) → A), (B(0.25) → AA)}

which produces:

(*n* = 0): A
(*n* = 1): AB
(*n* = 2): BBAA
(*n* = 3): AAABBAB
(*n* = 4): ABABBBAABBAA
(*n* = 5): ABABBAAAAABABAABBAB

This system could have produced man different systems, usually we associate a stochastic L-System with its own seed, to maintain consistent evolutions. All of these properties and extensions are beneficial for the quadtree L-System used in our procedurally generated surface code.

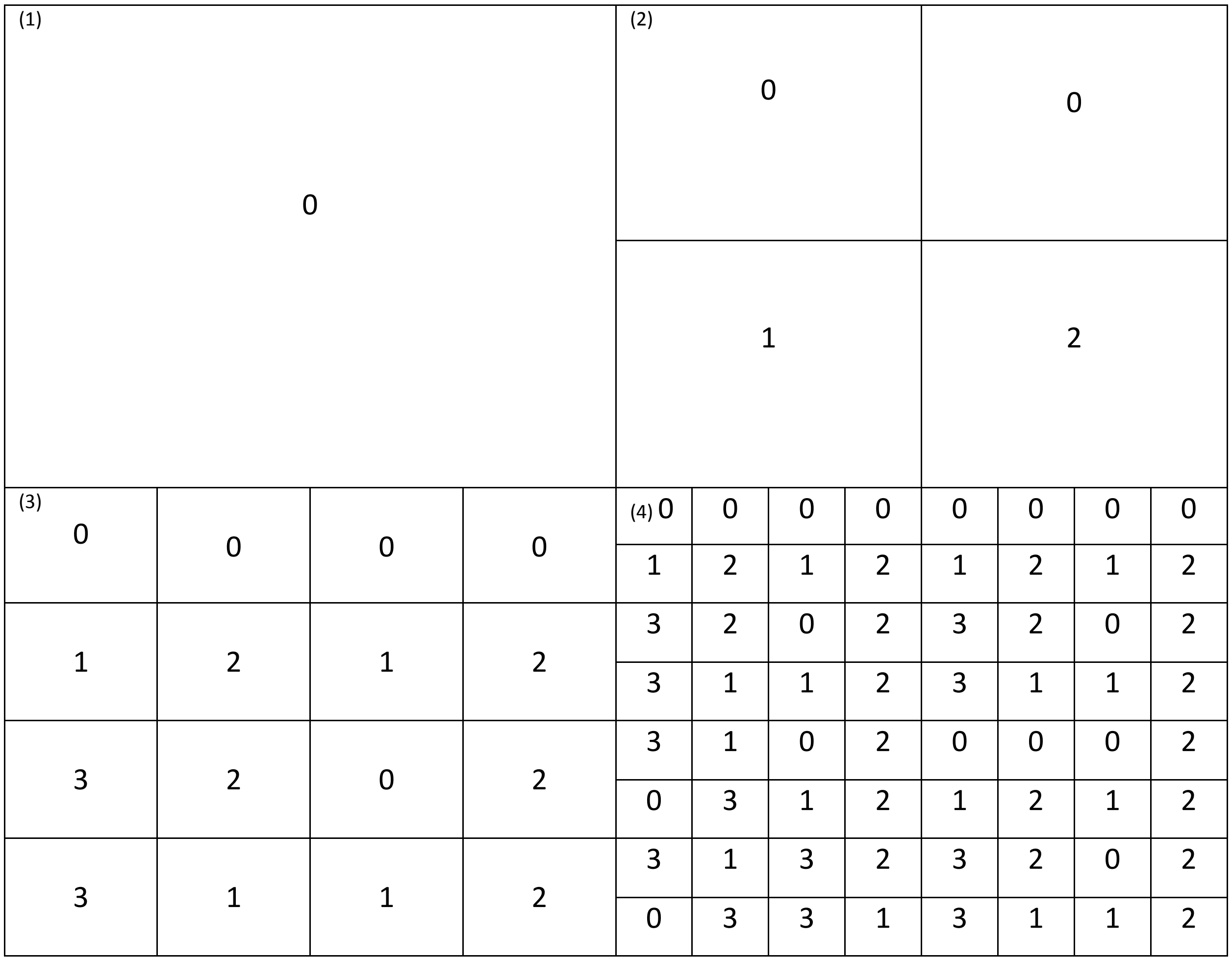
L-Systems as Quadtrees

The main idea for this project was to recursively divide a landscape into quadrants, where each quadrant may be broken down further into other quadrants, and so on. This allows us to determine exactly how much detail we need for a specific quadrant on a much larger grid. For generation, the idea was to use an L-System whose production rules all transform a symbol into 4 other symbols, thus the evolution of the system will continuously break the landscape down into more focused quadrants. Here is an example:

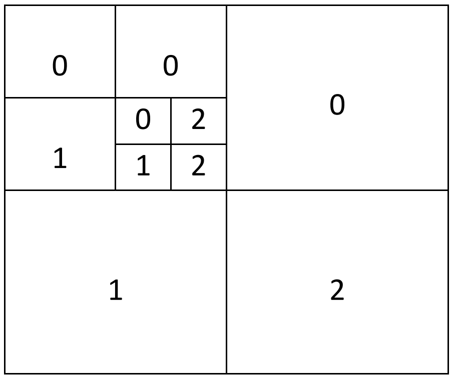
V: {0, 1, 2, 3}

ω: 0

P: (0 → 0012), (1 → 3231), (2 → 0212), (3 → 3103)



At each evolution step, each cell in the grid is broken into 4 new quadrants, allowing us to quickly divide a large region into smaller, more manageable chunks. Also, remembering the fact that L-System quadtrees are recursive, we can choose to expand certain chunks more deeply than others, allowing us to only focus on generating detailed surfaces for visible chunks. Here is an example:



We will explain the targeted expansion in later sections. So, now we have a structure for our L-System quadtree that can recursively divide a large area into many smaller quadrants, the next step is using the system actually generate the surface associated with these quadrants.

Heightmaps from L-Systems

For the process of actually generating a surface, we treat our previously defined L-System quadtree structure as a heightmap. In order to accomplish this, we must extend our previous simple model, to a new parametric model. In our updated model, each quadrant will be associated with a height, so after several evolutions of our system, the resulting grid will be a heightmap modelling our surface. So, we can change the above definition to a new parametric one, where each symbol has 2 parameters: the current height, and a decay parameter. The current height parameter represents the height value of the given quadrant, this is what's used in the height map, and the decay parameter is used to control the change in height as the system evolves. By choosing a decay between 0 and 1, the relative change converges to 0 as the system evolves, this smooths our map and allows us to maintain a more consistent looking structure across evolutions. In our new system, we also define functions for calculating the new heights after an evolution, these usually consist of adding a delta to the previous height, using the decay as a smoothing factor. Here is an example of a formal definition of our new system:

V: {0, 1, 2, 3}

ω: 0(1.0, -0.0321, 1.0)

P: 0(x, y) → 0(x+(-0.0552*y),y*0.6532) 0(x+(0.2376*y),y*0.9985) 1(x+(0.1236*y),y*0.8832) 2(x+(0.5499*y),y*0.7412),

1(x, y, z) → 3(x+(-0.0452*y),y*0.8562) 2(x+(-0.4446*y),y*0.8685) 3(x+(0.1437*y),y*0.8665) 1(x+(0.0032*y),y*0.9974),

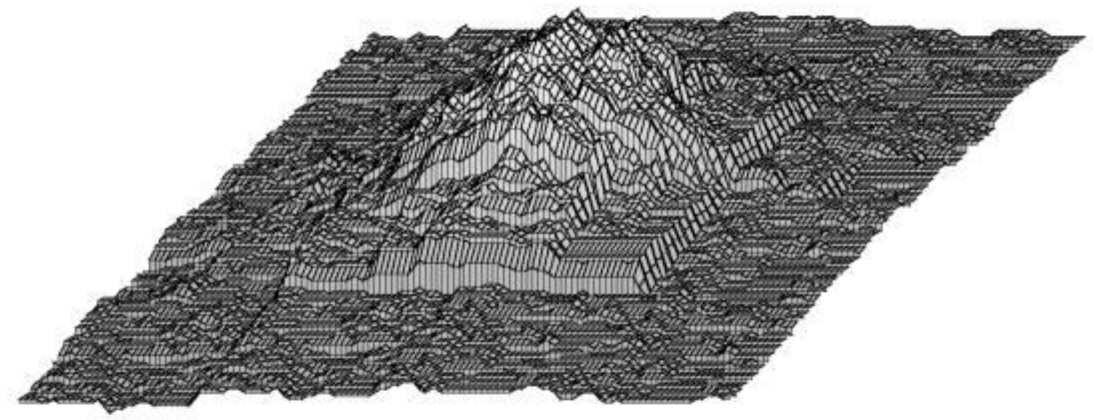
2(x, y, z) → 0(x+(0.2252*y),y*0.7532) 2(x+(-0.4476*y),y*0.9775) 1(x+(0.1445*y),y*0.6624) 2(x+(0.5321*y),y*0.7002),

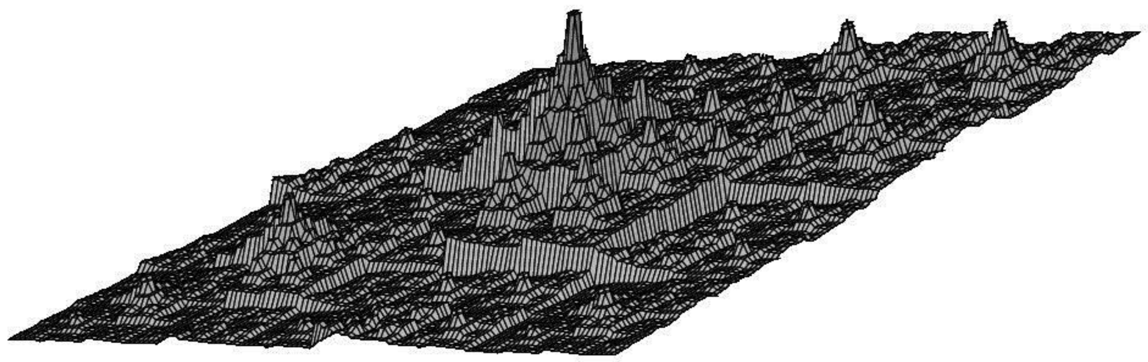
3(x, y, z) → 3(x+(-0.0023*y),y*0.6412) 1(x+(0.6783*y),y*0.9700) 0(x+(-0.8834*y),y*0.4235) 3(x+(-0.5210*y),y*0.7774)

We can also add randomness to our model by making it stochastic. There a couple ways of doing this, basically we can use a combination of having the production rules expand to random strings, with random deltas, and random decay parameters. Then if we associate a system with a seed, we can use a single L-System to generate an extremely large set of unique surfaces.

Generating Specific Surfaces

As it currently stands, our updated L-System is capable of generating a wide variety of surfaces, however it may be useful to note which system and seeds generate which structures. Thankfully there already exists dynamic programming algorithms that compare heightmaps. We can generate a perfect heightmap for a specific structure or landmass, and then run our L-System generator using a variety of different rules and determine which system best represents the given structure. Here are a couple of images of rendered L-Systems that were determined to closely match a hill structure:

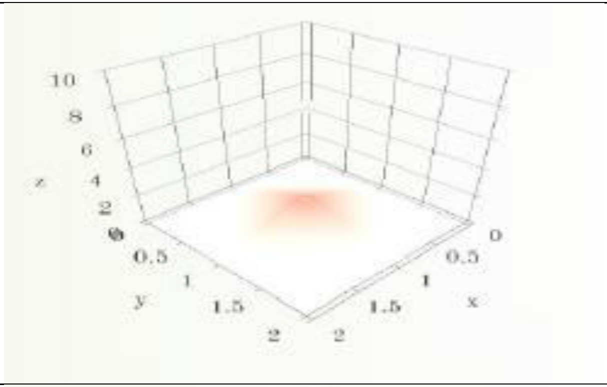


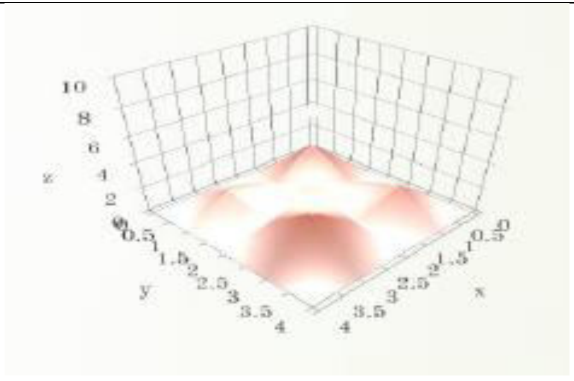


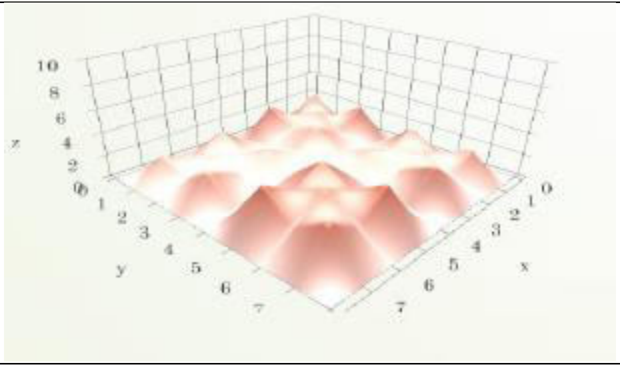
While these aren't perfect matches, they both in different ways do resemble a hill. Determining which structures produce which features gives us the ability to generate specific looking terrain, instead of highly randomized maps. What's more, because of the recursive structure of the L-System, we can use different feature specific systems at varying levels of scope. To be more precise, for a world sized map, we can first look into which L-System produces a heightmap similar to earths, then, once we've broken our planet down into smaller quadrants, we can focus on using L-Systems that generate height maps similar to specific continents, and specific regions, and cities, and so on. This also is proof that the L-System method can be used to generate a large variety of features and landmasses.

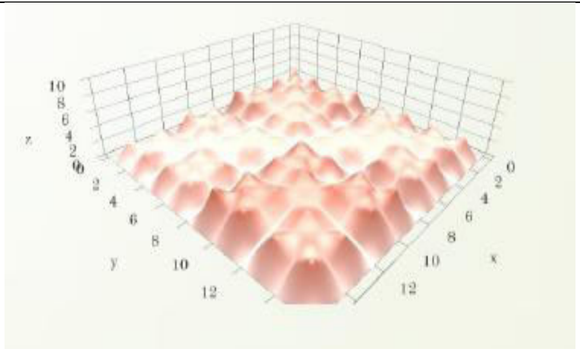
LOD using L-Systems

One of the benefits we've continuously mentioned about the L-Systems is their recursive structure, along with their ability to effortlessly expand and contract. This allows us to easily define multiple levels of detail using the same structure, and it's also very efficient to change from one LOD to another. Basically, we divide our target landmass into varying numbers of quadrants, generating the heights for each quadrant, areas with higher LOD will have more quadrants, while areas of lower LOD will only be expanded only a couple times, if at all, using way less memory. Here is an example of the same L-System being interpolated at varying levels of detail:





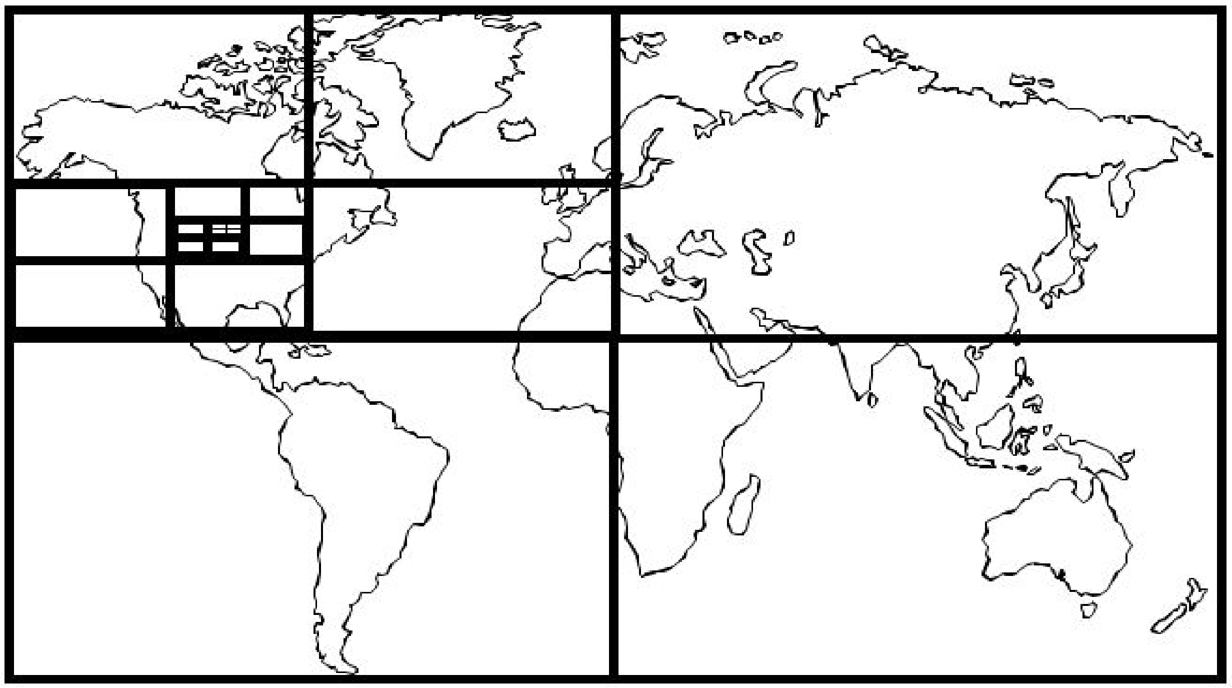




The above example shows a system at generations 1, 2, 3, and 4, so it only requires one expansions or contraction to get from one LOD to the next in this example. Since expanding/contracting our system requires only basic operations, this means we can generate and switch between multiple levels of detail very efficiently, which is ideal for programs that require rendering a large area with multiple changing LODs. The fact that L-Systems can provide an easily maintained and built-in method for portraying varying levels of detail is one of the major benefits of the L-System method, and becomes incredibly useful when looking at handling extremely large surfaces.

Rendering Large Landscapes Using L-Systems

Now that we know the L-System method is capable of generating a wide variety of interesting and unique heightmaps, and can efficiently produce varying levels of detail for a surface, we can discuss its use in rendering incredibly large surfaces in video games. We begin by looking at an earth-sized world, obviously we cannot generate a surface for a map this large, and even if we could, doing so would be pointless since we can only see an extremely small fraction of the world at any time, so we need to be smart about how we store and generate the environment. Using our L-System method, we focus on expanding the branches of our quadtree that contain our character and the surrounding area. This allows us to save a large amount of memory, while also providing us with a quick way of generating any specific chunk on our map by contracting and expanding our system. Here is a visualization of how the map may be broken down:



Here we have focused the expansion of our system to somewhere in the northern continental United States. By focusing our expansion we save a lot of memory by not generating extra nodes in area that we currently don’t care about. To be more clear, Australia is too far away from the USA to require any detail, so we do not expand that quadrant. Think of this initial step as a search, given our main characters X and Y coordinates, we recursively divide our landmass into quadrants (by expanding our L-System) until we have the specific quadrant that contains only our main characters field of view and nothing extra. Once we have found the target quadrant or chunk, we can focus on generating the heightmap for this region. Note that for all the other chunks on the map no further action is required, this saves us a lot of what would be otherwise wasted memory. Now depending on the goals of the project or game, the field of view may be quite large, let's say ours is 64km^2, once again the area that is far away and on the edge of this region requires less detail then the surface right in front of our character, so we can once again focus the expansion of our system depending on where we want the detail. In this case we will split our region into 512m x 512m quadrants (by expanding our system), and then we generate the heights for each cell. Cells further away on the border of the field of view chunk will only be expanded once or twice, providing only a few heights since detail here doesn't matter, while cells closer to our character in the center will be expanded more, since more details are required. Here is an example of a 64km^2 chunk divided into 512m^2 cells, the numbers in each of the cells are the number of heights generated for the specific region, and the blue cell is the one the player is located in:

16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	256	256	256	256	256	256	256	256	256	256	256	256	16	16
16	16	256	256	256	256	256	256	256	256	256	256	256	256	16	16
16	16	256	256	4096	4096	4096	4096	4096	4096	4096	4096	256	256	16	16
16	16	256	256	4096	4096	4096	4096	4096	4096	4096	4096	256	256	16	16
16	16	256	256	4096	4096	65536	65536	65536	65536	4096	4096	256	256	16	16
16	16	256	256	4096	4096	65536	65536	65536	65536	4096	4096	256	256	16	16
16	16	256	256	4096	4096	65536	65536	65536	65536	4096	4096	256	256	16	16
16	16	256	256	4096	4096	65536	65536	65536	65536	4096	4096	256	256	16	16
16	16	256	256	4096	4096	4096	4096	4096	4096	4096	4096	256	256	16	16
16	16	256	256	4096	4096	4096	4096	4096	4096	4096	4096	256	256	16	16
16	16	256	256	256	256	256	256	256	256	256	256	256	256	16	16
16	16	256	256	256	256	256	256	256	256	256	256	256	256	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16

So, we now have an L-System that models our map, which is roughly size of earth, and using our targeted, or lazy expansion of the tree, we've focused on only a 64km^2 region, and defined heights only where required. The memory usage is still rather high for this specific example, considering the world size and our large and rather detailed viewable range, the L-System tree will end up using approximately 100MB of ram. However, this number really shows the efficiency of the L-System method when you consider that a fully expanded tree for an earth size map at the highest detail level we've used would take up approximately 64 petabytes of memory. It's possible to reduce the memory use by defining smaller viewing distances and changing how the grid is defined and used. A key fact to remember here is that although the vast majority of the map hasn't been generated (or expended) yet, our L-System still has the ability to generate it in only a few expansions. We can use this fact to generate new chunks as our main character journeys through the map. As the character travels away from the blue cell, we can expand our system to produce new heights and details as we get closer to boundary cells, and contract our system when cells become too far away to see. This gives us a method for rendering the entire map, while still maintaining a small memory footprint throughout the journey. The only caveat here is that modified chunks will have to be saved somewhere else separate from the L-System tree, otherwise the changes would be forgotten once the player travels far enough away from the chunk.

Future Considerations and Ideas

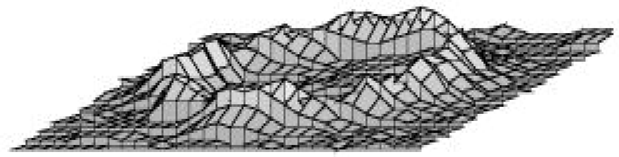
One major idea that would dramatically benefit the L-System surface generator is the use of binary triangle trees. Currently, all renderings have been done using triangle strips, however using binary triangle trees could dramatically cut down on the triangle required to render the surface, and provide an easier way of connecting chunks and managing LOD switching.

The L-System method presented here also only produces surfaces, and besides texturing, provides no distinct way of generating water, flora, or other features. However, L-Systems may also be well suited for generating these as well, especially flora. In the future, it will be worth researching how L-Systems can be used to generate unique flora such as trees, or shrubs, or flowers, as well as whether or not the generation methods for these entities work well with the surface generator, or require their own separate data structures.

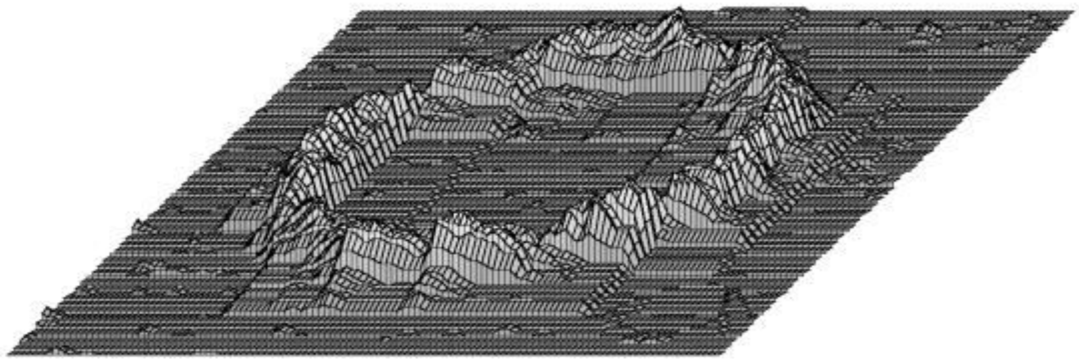
We have shown that the L-System surface generator can provide a very feasible method for storing and rendering entire planets as viewable chunks, however how does this scale to larger systems such as solar systems, or galaxies, or a universe. Currently, this system is only designed to render surfaces, not model planetary or solar bodies and their interactions. However, it may be worth-while to investigate how the L-System method may be used to render an entire universe. A system designed to hold an entire universe may require too much memory, even using the lazy expansion technique, however, using stochastic systems and 64-bit seed number may be a possibility, allowing us to produce an extremely large number of unique planets. How the galaxies are structured will have to be determined separately however.

Examples

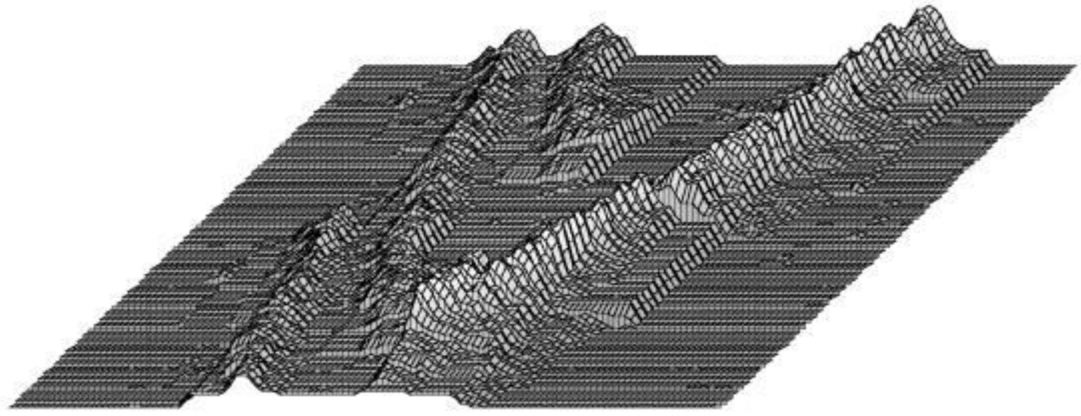
Here are some wireframe and textured renderings of several different systems. Note, for the textured renderings, the green areas are low, flat grassy regions, while the grayish areas are higher mountain regions.



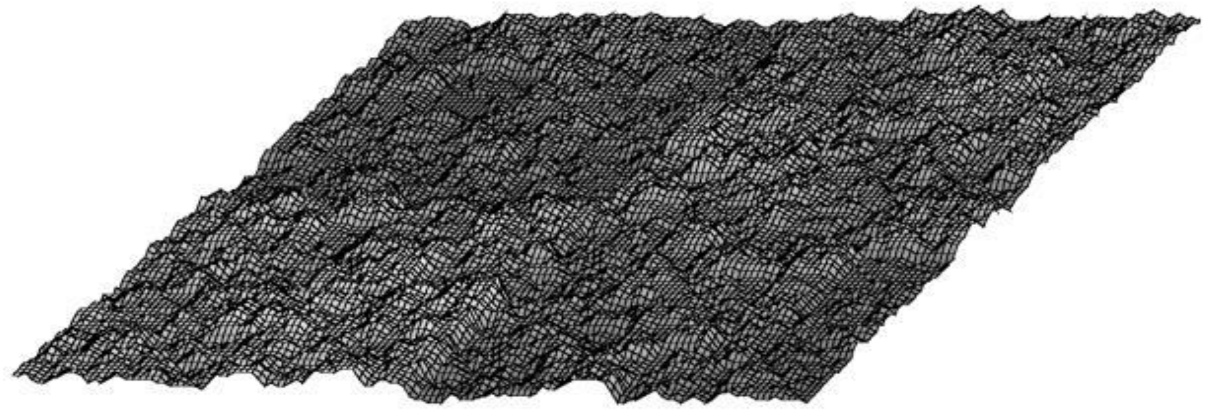
A Low LOD wireframe of a crater produced by an L-System



A more detailed wireframe rendering of the above L-System

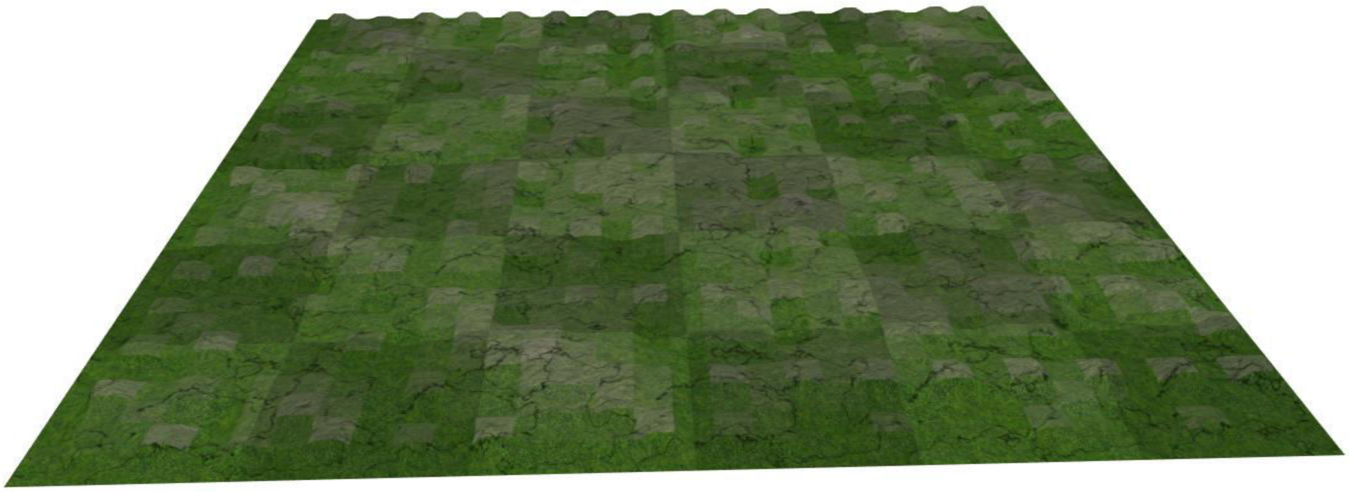


A system the produces a ridge-like environment

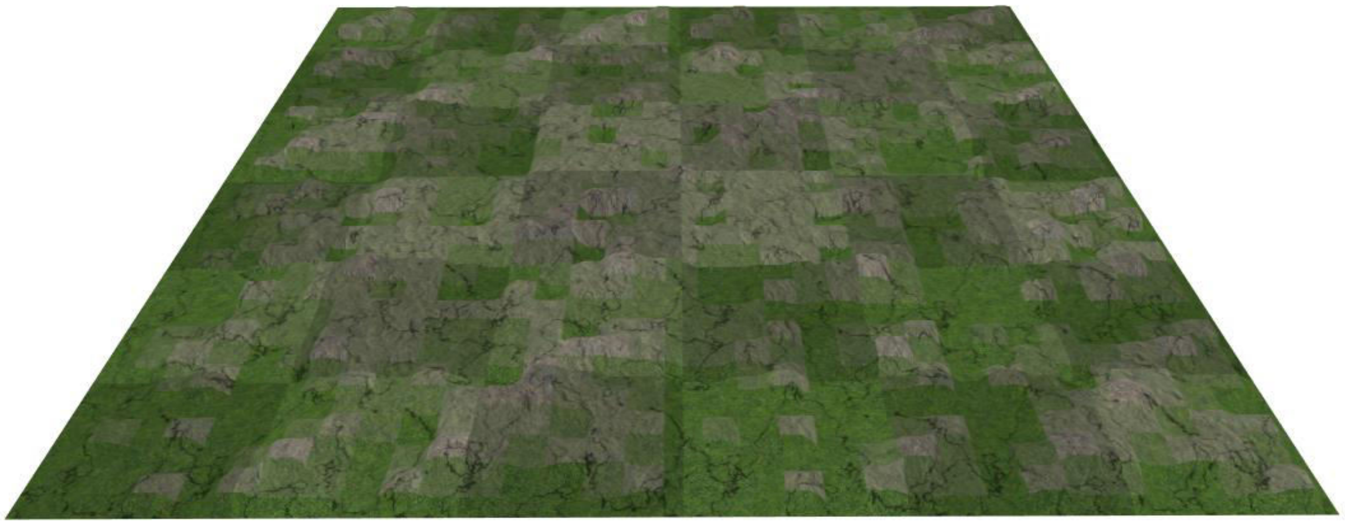


A rather flat but rough wireframe

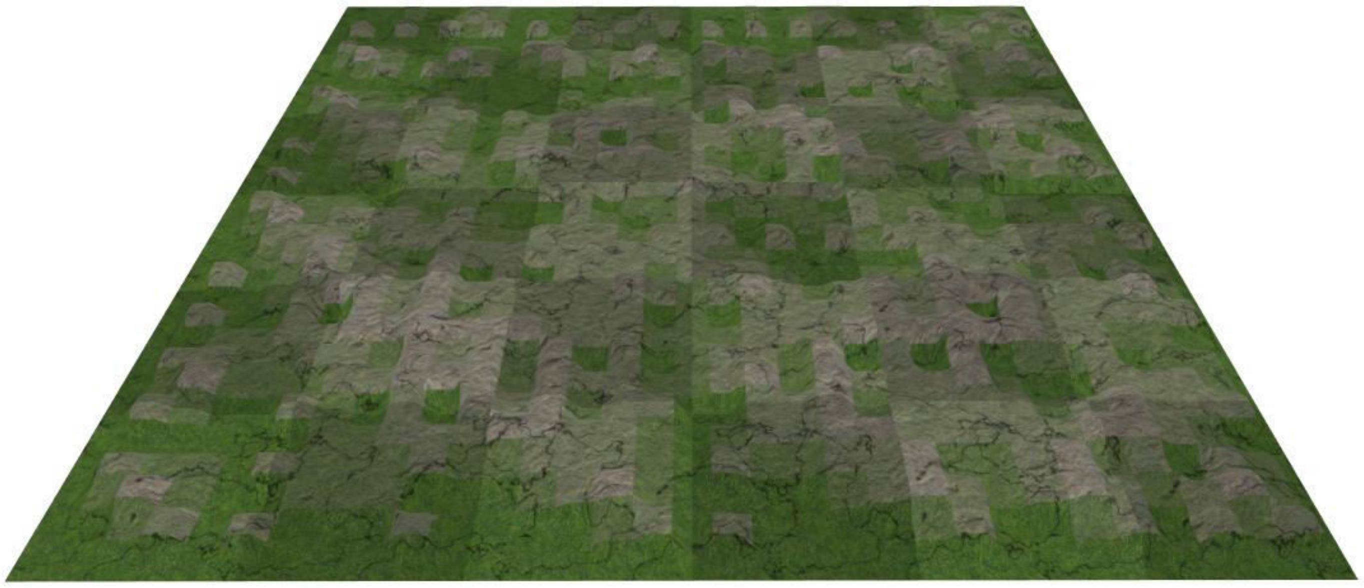
A textured rendering of a flat L-System



Another flat system, albeit with more small hills then the previous.



A region with much more hills then previous examples, this is a low detailed rendering of the last wireframe.



A mountain range terrain with several high peaks and low-lying valleys

Notes

Compared to Perlin noise generation, the L-System method is much easier to implement, requiring only a simple data structure and an expansion function operate correctly. However, the L-System method requires more memory and may be slower when rendering large areas. The L-System surface generator really shines in the context of its adaptive LOD. The L-System is capable of generating only the required heights using the lazy expansion technique, allowing for much larger visible regions to be stored in a much less memory. Overall, the L-System method seems to be a very feasible method for generating surfaces, especially for handling large areas, at the very least, it provides and effective method for generating heightmaps that may later be used in landscape generation.

ALL CODE FOR THIS PROJECT CAN BE FOUND AT: https://github.com/RyanFitzpatrick/LSystem