Procedural Hilly City Generation with Perlin Noise & L-Systems

Raul Hernandez
Massachusetts Institute of Technology
rhern@mit.edu

1. Introduction/Motivation

When scenes are created and stored in a traditional manner (i.e. storing individual scene objects on file), these files can end up being very large when the scene is large. Such an approach does allow for greater customization of a scene, but if memory-efficiency is a target goal in storing the scene, then this approach does not reach such a goal.

Scenes can instead be generated using procedural generation methods, where a given set of rules or functions dictate (either deterministically or stochastically) how scene objects appear or are placed into a scene. This greatly increases memory-efficiency as scene objects are no longer individually stored; rather, only a typically small set of rules are stored that can be applied to generate the scene. In the case of scenes that have imitations of natural structures (e.g. hills), a form of random number generation (RNG) that has an "organic" appearance must be used—this is where Perlin Noise and L-Systems come into play.

2. Background Work

A YouTuber by the name of Geoffrey Datema conceived an idea for city block generation using L-Systems that consisted of using L-Systems to subdivide city blocks into smaller city blocks.[1] In this project, I used a similar approach of generating city blocks with L-Systems, but I instead utilized the L-Systems to dictate the number of buildings that would be placed onto each city block.

3. Background Knowledge

3.1. Perlin Noise

Perlin Noise is an RNG approach designed by Ken Perlin that manifests an "organic" feel to the numbers which it generates.[2] At a high level, the 2D Perlin Noise algorithm (used in this project) generates random gradient values at integer coordinate values on the Cartesian plane, and uses values derived from those gradients to compute the noise at a particular point via interpolation with a smooth function. Below is some pseudocode for the 2D algorithm:

```
As pre-computation, generate
random gradient values

Perlin_noise(x, y) function:

Get gradients $g_i$ at $(\lfloor x\rfloor,\lfloor y\rfloor)$,
$(\lfloor x\rfloor+1,\lfloor y\rfloor)$, $(\lfloor x\rfloor,\lfloor y\rfloor+1)$, and
$(\lfloor x\rfloor+1,\lfloor y\rfloor+1)$
Compute vectors $v_i$ pointing
from coordinates above to
(x,y)
Bilinearly interpolate dot($g_i$,
$v_i$) at (x,y) using coordinates
above, and return this value
```

3.1. Perlin Noise (cont.)

Below are some examples of products made from the 2D Perlin Noise algorithm:

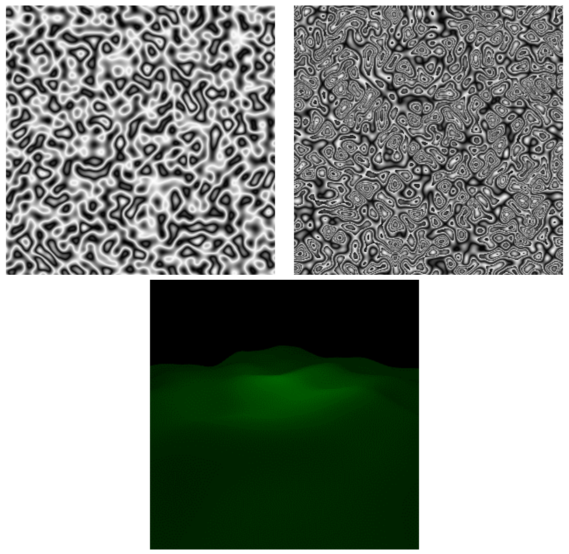


Figure 1. Different images generated from 2D Perlin Noise. The top-left figure shows a smooth transition between intensity, where values close to 0 or 1 are darker, and values close to 0.5 are lighter. The top-right image shows a similar pattern, where values close to 0 are darker and values closer to 1 are lighter. The bottom image shows an isolated hilly scene with heights determined by Perlin Noise.

3.2. L-Systems

L-Systems are a method of deterministic or stochastic letter sequence (*word*) generation, created by biologist Aristid Lindenmayer, based on a set of pre-defined *rules* that change *letters* from a pre-defined *alphabet* into other letter sequences.[3] The determinism of the generation comes from whether the pre-defined rules are deterministic.

The procedure for generating a word from an L-System is the following. First, begin with an initial L-System word called an *axiom*. Then, at every step of the L-System word generation, apply the L-System's deterministic/stochastic rules to transform each letter in the current word into the rule's dictated letter sequence.

Below is an example of a Koch Snowflake fractal that is generated from the following L-System alphabet/axiom/rules:[4]

Alphabet:

MOVE (Draw line in faced direction)
RIGHT (Turns 90° clockwise)
LEFT (Turns 90° counterclockwise)

Axiom:

MOVE

Rules (Deterministic):

MOVE → MOVE RIGHT MOVE LEFT MOVE LEFT MOVE RIGHT MOVE
RIGHT → RIGHT
LEFT → LEFT

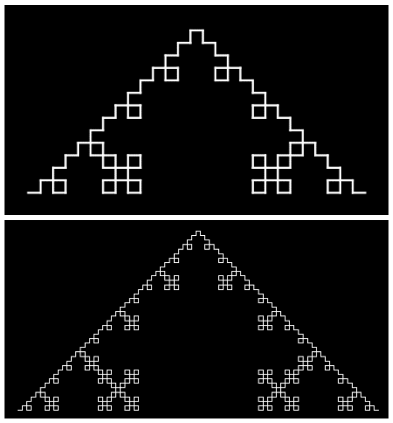


Figure 2. Different Koch Snowflake fractals that were generated by the L-System described above. The top image depicts a Koch Snowflake generated by 3 iterations of the L-System. The bottom image depicts a Koch Snowflake generated by 4 iterations of the L-System.

4. The Hilly City Generation Pipeline

The general approach to generating the complete hilly city was to modularize the modeling process on a city block basis, then render everything with ray tracing.

To generate the hilly cities promised by my project, I first began by generating the ground level of each city block. Each city block had a 16x16 grid of squares, each of which was defined by two triangles. The colors of the triangles were chosen to reflect grass/road textures. Triangles near the border of the city block were colored dark gray (as roads), and all other triangles were green (as grass). The height of the hills at every triangle vertex was determined by a 2D Perlin Noise function's value at that point. The random Perlin Noise used distinct random gradients for every distinct render.

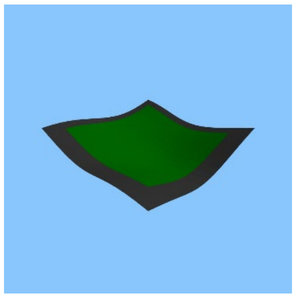


Figure 3. A sample of a city block's ground level, showing Perlin Noise-defined hill height and different color textures.

Once each city block's ground level was modeled, sections of the city block's ground were subdivided into groups of triangles placed together into a bounding volume, which were placed into a Bounding Volume Hierarchy (BVH). This is a method by which computationally cheap high-level intersections with multiple primitives' bounding volumes are checked for early in rendering, and thus potentially rejects many primitives whose expensive intersection operations do not need to be done. The hierarchy comes from a recursive approach to this which further accelerates the rendering process. Ultimately, 4x4 subgroups of hill squares were placed into a total of 16 BV subgroups per city block. The use of this optimization resulted in a 200-300% speedup in the rendering time of most city scenes.

Then, the L-System *stochastic* pattern of the number of buildings per city was computed for each render. The L-System was the following:

Alphabet:

MOVE BLDG

Axiom:

BLDG MOVE

Rules:

BLDG → [empty] (20% chance)
BLDG → BLDG (40%)
BLDG → BLDG BLDG (40%)
MOVE → BLDG MOVE (40%)
MOVE → MOVE MOVE (20%)
MOVE → MOVE (40%)

This L-System setup allowed for a randomness in the number of buildings per city block. The words from the L-System were interpreted in the following manner. Every *n*-consecutive sequence of "BLDG" letters indicates that the city block will have an *n* x *n* grid of buildings (manually capped at 2x2). The "MOVE" letter acts as the cutoff for the building count for one city block to the next. City block building patterns are placed in a z-axis major manner. For example, the L-System word "BLDG BLDG MOVE BLDG MOVE" would be a city where the first city block (down the z-axis) would have a 2x2 grid of buildings, the second would have 1x1, the third would have 2x2, the fourth 1x1, and so forth. The L-Systems used for building counts used 10 steps for word generation.

Once the L-System pattern is finalized, the buildings are modeled and placed onto their respective city blocks. Buildings had a roof defined by a parallelogram, or more specifically, a rectangle; the intersection approach with this primitive is almost identical to that of a triangle, but the condition on the barycentric coordinates β and γ is $0 \leq \beta,\gamma \leq 1$ instead. Similarly, the buildings' walls were made by a custom primitive I named WindowWall. Intersection with this primitive is almost identical to that of a parallelogram, but the material varies between a wall material (which had a total of 3 distinct choices) and a reflective window material based on the local barycentric coordinates of the intersection point. The window count and height was determined by "inorganic" RNG. By using parallelograms rather than triangles for building modeling, this halved the number of primitives that were used to represent buildings, compared to a pure triangle representation, thus reducing render time.

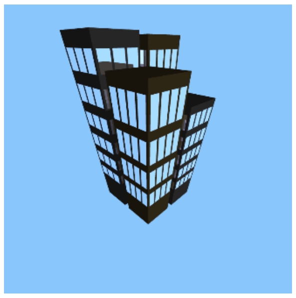


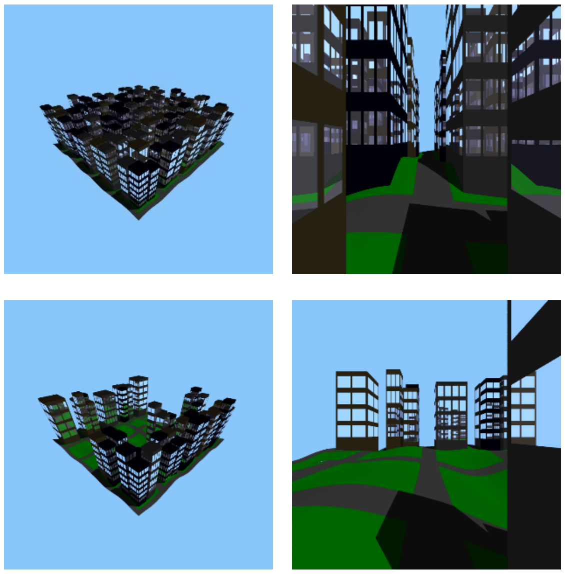
Figure 4. A sample of how a 2x2 grid of isolated buildings appears, with random wall color texturing and reflective windows enabled by ray tracing.

For faster rendering, buildings' walls and the buildings themselves were also placed (on a city block basis) into the BVH system also used by the city blocks' hills.

Finally, all these procedurally generated hills and building models were rendered using the same ray tracing functionality as in 6.4400's Assignment 4, but done in Java 1.8, which basically entailed re-implementing the OpenGL and GLOO framework in Java. This was done to increase my fundamental understanding of the frameworks, alongside leveraging my comfort with the Java language. As for details on the ray tracing, 1) a directional light was used to emulate the sun's lighting over a large metropolitan city, 2) a 3x3 average sampler was used for anti-aliasing per pixel, 3) reflections off the buildings' windows were computed with 4 ray bounces, and 4) shadows and a sky background were used for an effect of realism.

5. Results

All in all, two different camera perspectives of the procedurally generated cities were rendered at a 700x700 pixel resolution. The first camera perspective was somewhat of an angled bird's eye view of the city. This perspective took about 3 min. to render at 700x700 resolution. The second perspective was that of someone walking through the city; rendering took about 7 min. at 700x700. Results can be found below:



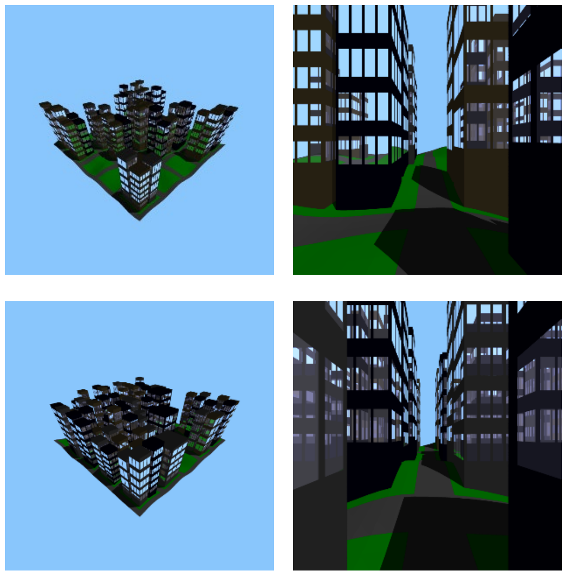


Figure 5. Renders of four distinctly procedurally generated cities, generated/rendered by the method described for this project. The left column shows the angled bird's eye view camera perspective, whereas the right column shows the same city's walking camera perspective.

While there are many possible L-System words that could have generated each pattern of building placement (due to the manual cutoff at 2x2 buildings per city block), the minimal L-System word for each city can be inferred. For example, since every city block in the topmost city depicted in the graphics has a 2x2 grid of buildings, its final L-System word may have been "BLDG BLDG MOVE," though it may have equivalently been "BLDG BLDG BLDG BLDG MOVE," for example.

6. Conclusion and Next Steps

In this project, a method for procedural generation of hilly cities with buildings was designed, implemented, and executed. The approach revolved around the two procedural generation tools of 2D Perlin Noise and L-Systems. These tools of randomness came into play when determining the height of the city's hills (via Perlin Noise) and the number of buildings to place onto each city block (via L-Systems). Buildings were modeled using typical graphics primitives, in addition to a custom WindowWall primitive that varied the material on the building walls to allow for reflective windows. Rendering was done using ray tracing to render the effect of reflective windows. To speed up rendering, the use of a Bounding Volume Hierarchy was employed.

This method of hilly city generation can be extended by increasing the realism of the cities through various facets. For example, cars driving through the roads and people walking on the grass and/or crossing the road can be added to the city scene to add a realistic sense of the city being inhabited. Additionally, clouds can be added using a very similar RNG approach as 2D Perlin Noise (such as 3D Perlin Noise). Lastly, realistic textures can be added to the buildings and ground (e.g. a rough texture for the road asphalt, grass blades for the grassy sections of the hills, etc.).

7. References

[1]Datema, Geoffrey (2021). *Other L-system Applications | 3D Graphics Overview.* https://youtu.be/BkE8t-mg3Tk?si=VcgrpC87rEScapPX.

[2]Flafla2 (2014). *Understanding Perlin Noise.* https://adrianb.io/2014/08/09/perlinnoise.html.

[3]Carey, B. (2019). *Procedural Forest Generation with L-System Instancing (Doctoral dissertation, Master's thesis, 2019*. URL: https://nccastaff.bournemouth.ac.uk/jmacey/MastersProject/MSc19/02/MastersReport.pdf).

[4]Datema, Geoffrey (2021). *Turtle Graphics in L-Systems | 3D Graphics Overview.* https://www.youtube.com/watch?v=LSSkaygNlYI.