THE RELATIONSHIP BETWEEN PROCEDURAL GENERATION TECHNIQUES:
CELLULAR AUTOMATA AND NOISE

by

JESSE HIGNITE

Advisor

DANIEL PLANTE

A senior research paper submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
in the Department of Mathematics and Computer Science
in the College of Arts and Science
at Stetson University
DeLand, Florida

Summer Term

2017
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................... 3

LIST OF FIGURES ................................................................................................. 4

1. INTRODUCTION ................................................................................................. 5

2. BACKGROUND ................................................................................................... 5

3. CONCEPTS
   3.1 CELLULAR AUTOMATA ............................................................................. 6
   3.2 NOISE ........................................................................................................... 9

4. IMPLEMENTATION ............................................................................................... 10

5. RESULTS ............................................................................................................. 1

6. CONCLUSION ..................................................................................................... 1

7. FUTURE WORK .................................................................................................. 1
ABSTRACT

Procedural generation provides an automatic alternative to manually creating data. With this automatic generation of data, programs are capable of outpacing human generation of resources. A problem arises with automatic generation: unnatural patterns that make the automatic nature of the data obvious. This is particularly apparent in visual data such as textures or maps. There are a multitude of algorithms designed for various applications, such as Perlin noise generation for textures and simulation of airflow within a wind system. Procedural generation has the potential for providing more variety to visual systems than humans reasonably could, and its advancement could allow for more dynamic scientific simulations and more interesting media. Of particular interest for this project are the interactions between the various concepts in procedural generation, as this allows existing algorithms to be applied in a manner that leads to a multitude of permutations. The hope is that by applying cellular automata and Perlin noise together, the end result will be a more appealing solution than either may be on their own. Cellular automata and Perlin noise should allow us to adjust the complexity of generated caves.
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cellular Automata Initial State</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Cellular Automata Final State</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Cave Screenshot</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Wrapped Plane Meshes</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Cellular Automata Grid</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>Volumes vs. Lower Bound</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>Volumes vs. Upper Bound</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Volumes vs. Scale</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>Open Space vs. Scale</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>Lower Bound vs. Ratio</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Upper Bound vs. Ratio</td>
<td>20</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The purpose of Procedural Generation is to algorithmically create data that would otherwise need to be manually created. Applying procedural generation shifts workload from an individual to the algorithm, allowing far more content to be created than would otherwise be possible [1]. Procedural generation is often used to create various resources for video games, to the point where one could create a game that entirely uses content that has been generated algorithmically. Games that employ procedural generation often use it for creating maps, items, characters, graphics and many other things.

There are a wide variety of techniques available for procedural generation. These techniques have a tendency to leave their own mark on the results. This may lead to a sense of familiarity with particular methods that may manifest itself in recognizable patterns or a general feeling of “sameness” even when viewing results from separate instances. Procedural generation methods are always being refined and tweaked in order to maintain variety.

2. BACKGROUND

Early computers, being limited in memory, benefitted greatly from procedural generation techniques. Rogue, a dungeon-crawler game developed by Michael Toy and Glenn Wichman in 1980 is an example of the early use of procedural generation [2]. The game features procedurally generated dungeons and items. As such, no two plays of the game are the same.

Procedural generation allows an unlimited number of resources to be available without sacrificing storage. Algorithmically creating assets for a piece of software (particularly a game) makes for a much smaller filesize. For example, a 2004 game titled .kkrieger came in at 96
kilobytes [3]; in comparison, at that time many games were being released on DVDs, requiring multiple gigabytes of space.

While video games are a significant area of application for procedural generation, it has other uses as well, and it could theoretically be applied in any context that requires data creation. In films, procedural generation can be used in much the same way as in video games, creating visual content (like trees) to add to a scene. One notable example is Peter Jackson’s The Lord of the Rings, which has battles with thousands of combatants. Rather than hiring thousands of extras for the battle, the film used a piece of software known as MASSIVE to generate the scenes [4].

The aim of this research is to apply the concept of cellular automata in conjunction with Perlin noise. Ideally, this will yield results that are more interesting than that of either concept independently.

3. CONCEPTS

3.1. CELLULAR AUTOMATA

Mathematician John Horton Conway created his “Game of Life” in 1970 [5]. The Game of Life is played on a two-dimensional grid of cells. The cells are given an initial state, where each cell is either alive or dead. The cells then have a set of rules applied to them repeatedly. These rules dictate the state of the cell in the next generation. Conway created his set of rules to determine the state of a cell based on its neighbors:

1. If a living cell has two or three living neighbors, it continues to live.
2. A living cell with more than three living neighbors dies.

3. A living cell with less than two living neighbors dies.

4. A dead cell with exactly three living neighbors changes its state to living.

All changes to cell states happen at the same time, so changes to a cell do not influence any other changes in the same generation. Conway did all of this on a checkerboard with colored counters, but now it is a simple exercise in programming to simulate the Game of Life on a computer.

Pseudocode for Conway’s Game of Life

```c
int neighbors( int[][] board, int x, int y ){
    // return number of living neighbors of the given cell, the four orthogonal and four diagonal
}

game_of_life( int[][] board, int width, int height ){
    initialize next_generation as 2D array with same height and width
    for( i = 0; i < width; i++ )
        for( j = 0; j < height; j++ )
            int neighbors = neighbors( board, i, j )
            if( board[i][j] ){
                if( neighbors > 3 || neighbors < 2 )
                    next_generation[i][j] = 0; }
            else
                if( neighbors == 3 )
                    next_generation[i][j] = 1;
            return next_generation;
}
```
Conway’s Game of Life is one specific instance of cellular automata; the rules may certainly be tweaked. However, Conway’s rules are well-established, and using them as a starting point in the usage of cellular automata in other areas helps translate the concepts of his game to other applications. Understanding cellular automata illustrates the draw of procedural generation: a known starting point that has the potential of resulting in a vast number of unknowns, quickly leading to the variety that we desire.

While cellular automata may have many unexplored uses, in procedural generation they are known to be useful for generating 2D caves. Seeding the grid can lead to desired results. By seeding a grid with a percentage of live cells and applying Conway’s rules a handful of times, it is possible to generate caves for a 2D dungeon crawling game or a similar purpose.

In this example, Figure 1 and Figure 2 are the initial seeding and the grid after 4 iterations of the rules, respectively. In this particular cellular automaton, live cells represent solid rock while the dead cells represent clear space. The rules were altered slightly, removing the overpopulation rule. From various test runs it appears that additional iterations applying the rules tend to smooth the walls, removing odd jaggedness and isolated caverns. It should be noted that the complexity of this simple algorithm is O(n^2).

Unfortunately, cellular automata are really only useful for generating two-dimensional caves, and their use for three-dimensional caves has not really been attempted [6]. A 3D cave could perhaps use 3D cellular automata, but that changes the behavior of the algorithm to the point
where it is unlikely to be fit for the purpose. 2D cellular automata may be useful for generating cavern boundaries in three dimensions, but another algorithm would be necessary for determining the floor and/or ceiling.

3.2 NOISE

Noise is a different beast altogether from cellular automata. Noise is the pseudo-random generation of data that is often used for procedurally generating textures. There are various algorithms for generating noise, but we are interested in gradient noise. Think of basic noise as the static on a television. Gradient noise instead has relatively smooth transitions between different values. An example of this is the generation of clouds in various image editing software. In 1983 Ken Perlin created the first type of gradient noise that became known as Perlin noise [7].

Perlin noise makes use of a grid consisting of nodes, where each node has a gradient vector that has been randomly chosen. The grid may be of any number of dimensions. The noise function takes the coordinates of the location to where it is to be applied and finds the corresponding location on the grid. There are $2^n$ nodes which make the corners of whatever cell the point is in on the grid, where $n$ is the number of dimensions on the grid. A distance vector is found between each node and the point. For each node, the dot product is found between the distance and gradient vectors. Finally, the vectors resulting from each dot product are used to interpolate a value for the given point. Typically, this is a float value between -1 and 1.

Perlin noise is often used for terrain generation. Minecraft is a well-known game that makes use of Perlin noise to create its terrain [8]. Gradient noise makes for a great terrain generation method, thanks to its tendency to smooth value differences. In generating 3D terrain, though, its
only real strength is establishing a height map of the landscape. This keeps it from being entirely useful on its own for generating other terrain types, such as a cave.

4. IMPLEMENTATION

We decided to use Three.js for handling our 3D caves. Three.js is an open source JavaScript library that utilizes WebGL for rendering 3D graphics. The library has been in use for several years now, thus many resources are available. Three.js also provides methods for accessing data that is important for analysis.

Our design takes advantage of an object-oriented approach. There is a central terrain generator object that is created by the application. The terrain generator has functions for creating the caves and modifying their various aspects. This terrain generator handles the
individual attributes of its caves, as well as the cellular automata and noise used in their generation.

A cave is defined as an object comprised of two Three.js plane geometries. The two geometries represent the top and the bottom of the cave. As one might expect, two flat planes will leave visible empty space between them unless they are occupying virtually the same space. For the purposes of our application, we made the decision to give the planes a curvature to allow them to intersect around the edges. This forces the planes to create an enclosed space prior to any further alteration to their heightmaps by the cellular automata and noise. Ultimately, this is an aesthetic choice rather than a functional one, as implementations of this cave generation could provide some other method of closing the edges (such as planes that form walls) or might desire the open edges.

The planes are “wrapped” to the edge of a sphere. A real example of this method would seem similar to adhering a piece of paper to the surface of a ball. We use a ghost scene – a scene that is never actually rendered – to calculate the changes to the plane. A sphere is generated for the purposes of the wrapping calculations. The sphere is given a diameter that is the same as the length of our planes. A standard sphere gives an undesirable end-result as it creates a large bubble in the center, leading to a significant disparity in distance between the top and bottom of
the cave between the center and the edges. To prevent this, we skew the sphere. Three.js provides a relatively straightforward way to scale geometry on each axis. Our goal was to make the sphere look more like an M&M or a ball that has pressure applied to its top and bottom. For our purposes, scaling the z-axis of the sphere to 75% of its original size produced results where the bubble was not particularly noticeable.

For each vertex of the plane geometry, we clone its position and extrude its z-position by an amount that is greater than our sphere’s z-axis radius. We then raycast from the vertex to the position of the sphere, checking for a collision with our sphere object. In order to further reduce any potential bubbling in the center of the planes, the z-position for the collision is capped at 75% of the highest potential position. This gives each of the planes the look of a plate – that is, it is flat in the center and after a certain point it gradually curves upward. Without this constraint, the planes take on the appearance of a convex lens, displaying the bubble effect.

In Three.js, plane geometries are initialized with a number of line segments for the height and the width. Each line segment is comprised of two vertices, so if the geometry is initialized with $n$ line segments, it will have $n+1$ vertices. All planes generated in our application are square simply because it makes the process easier; there is no reason any of our methods will not apply to other rectangles.
Our cellular automata operate on a grid based on the vertices of the plane. Each cellular automaton is an object comprised of several attributes. A cellular automaton is given a seed when initialized, either provided in the constructor or generated randomly. This seed is used primarily to seed the noise generator. The cellular automaton can be one of a few types; flat, random, or noise. The flat type of cellular automaton and the random type seed the grid with a static or a random value for each point, respectively. This is essentially how a 2D cellular automata-driven cave is generated. The noise type of cellular automaton, however, is the one in which we are most interested. Given the x and y coordinate of a point on the grid, the Perlin (or Simplex) noise gives a value that we use for the height at that point. We also have the option of scaling the noise; this is akin to zooming in and out on a map. Scaling the noise up will result in a more random appearance, as there is a greater distance between each point. Scaling the noise down gives a smoother, more gradual appearance, as values change more closely together.

The process for determining the next generation for a cellular automaton closely resembles the method previously discussed, but there are differing rulesets. We are able to manipulate the upper and lower bounds of neighbors for determining whether to increase or decrease the current point. Depending on the ruleset, the evaluation of the neighboring points differs as well. The standard ruleset is the one previously described: a count of the living
neighbors, killing the current point from overpopulation or loneliness. A new ruleset involves a sum of the neighbors, increasing the current height if it is between the bounds or decreasing if it is outside them. A third ruleset takes the average of the neighbors rather than the sum. The amount by which the points are shifted is determined by our displacement value.

We have several methods to aid in analyzing and tweaking the caves. The most notable method is one that uses flood fill to determine regions within the cave. Flood fill is the same procedure used for image editing programs that offer a fill tool. The algorithm takes a starting node, a target color, and a replacement color. However, rather than using colors, we use the height of the point. Thus, we have a target height and a resulting region. Flood fill is useful for finding contiguous sections on our grids. We define there to be two regions, the areas on the grid where the cellular automaton shift the heights up or down. This will appear similar to the 2D caves if viewed in a two-dimensional representation. Subregions are the contiguous points within the same region.
The volume of the cave is an important measurement. A cave with more volume can be assumed to generally be more open and easier to traverse, while less volume implies that the opposite is true. There are multiple approaches to calculating the volume of our caves, but we settled on a method described by Cha Zhang and Tsuhan Chen in their paper on feature extraction [8]. The plane meshes in Three.js are comprised of triangles. Each triangle is a face of
the mesh. Using the vertices of each triangle, $A$, $B$, and $C$, along with the origin, we form tetrahedrons. The volume of this tetrahedron can be found by the dot product of $A$ with the cross product of $B$ and $C$. The volume of the area between one plane of the cave and the origin is the sum of every tetrahedron respective to its faces. It is important that we calculate the volume for each plane individually and add them together, as this ensures that the volumes account for negative space; that is, if a plane crosses the origin, its volume in that space will be negative. As our wrapped planes intersect in places, the space at these intersections will not be inside the cave.

It is useful to view the cave in a 2D format. This is simple, as we just represent heights under 0 as the floor of the 2D cave and heights above 0 as walls. This representation does not account for very slight shifts that may influence the cave in practice, but it does allow us to find potential paths and determine if it is likely possible to get from one area of the cave to another. This is an application of the subregions found via flood fill. Less contiguous regions represent larger sections of open space for a given ratio of open space to closed space. Using this representation of each half of the cave, we combine them into one grid that represents open space of the entire cave.

5. RESULTS

To come to our results, we generated a series of caves and calculated our measurements. Generating these caves is relatively resource-intensive, taking roughly half of a second to generate one on our test machine. As such, it took some time to generate thousands of caves. Each data point on a graph represents a minimum of 10 caves generated and their results averaged.
One of the more interesting results is the change in volume as the lower bound for the cellular automata increases. Using the ruleset for cellular automata that utilizes the count of living neighbors, there is a noticeable decrease in the volume of the cave as the lower bound increases. Conversely, with the sum ruleset there is an increase in the volume of the cave as the lower bound increases. The ruleset for the average of neighbors does not have a significant pattern, save for the jump in volume from a lower bound of 0 and 1. We believe that the average ruleset likely corresponds more with values between 0 and 1, and values outside of this range are inconsequential.

![Figure 6](image)

The upper bound for the cellular automata yields somewhat similar results. Most notably, however, the sum ruleset joins the average ruleset in not generally corresponding with the change in the bound. The living ruleset does decrease volume as the upper bound increases.
The scale of the Perlin noise does not seem to effect much change in the volume of the cave. Values consistently fall within the same general range regardless of scale.
Unlike volume, the number of contiguous areas of open space increases substantially with the scale of the Perlin noise. We can attribute this to the more chaotic nature of the noise as it is scaled up. This is visually apparent with more ups-and-downs in the terrain.

The ratio of open space to non-open space generally stays within a consistent range. There is definitively no correlation between this ratio and the scale of the noise. There is little correlation between the ratio and the bounds of the cellular automata, but there is future potential that they could be used to influence the ratio.

**Figure 9**

Open Space Regions vs. Scale
6. CONCLUSION

Given our results, there is a clear indication that cellular automata and Perlin noise can provide the grounds for generating caves with desired parameters. Already we can see ways to influence the volume and the smoothness of the terrain within the caves. Using the information from our results, it should be relatively simple to implement something like an “enclosure factor” that would scale multiple variables in order to achieve the desired complexity of a generated cave.

7. FUTURE WORK

There are a number of improvements and additions to be made to our application. Pathing is a useful concept that may exert more control over the caves. We implemented basic pathing in order to give a little more control over the inside of the cave for aesthetic purposes. If these caves were to be used in a video game, pathing would be incredibly useful in order to guarantee a way through a cave for the player.

Giving the Perlin noise more influence over the cave beyond the initial seeding would be an interesting approach. A more intricate ruleset for the cellular automata would likely yield even
more interesting results, perhaps allowing for more fine-tuning of the cave structure. It is possible that a ruleset that considers both the top and the bottom as a whole rather than individually could give more dynamic caves. It should also be plausible that scaling the heights of the vertices within the cave could yield interesting results, as it could allow for a flatter floor with a vaulted roof, for example.
REFERENCES


[Accessed: 22 - Sep – 2016]

[Accessed: 23 – Sep – 2016]


[Accessed: 30 – Sep – 2016]
