Agent Based Cities

Scott Melero
CMPM 164, Fall 2019
University of California, Santa Cruz
stmelero@ucsc.edu

Jacob Wynd
CMPM 164, Fall 2019
University of California, Santa Cruz
jwynd@ucsc.edu

ABSTRACT

In computing, procedural generation is a method of generating data algorithmically as opposed to creating it manually. The use of procedurally generated maps has origins in the Tabletop RPG genre of games which we are big fans of, and so for this project, we looked into various ways we could procedurally generate cities in Unreal Engine 4.23. We set out to generate organic looking, fantasy cities that would generate at runtime with some degree of randomness and were different every time. We created our semi-random cities by implementing a tile-based generation method, in which each tile is a level filled with predefined assets that are spawned at random on a grid. It is accompanied by a tile-based road system that is drawn on the city at runtime.

CCS CONCEPTS

• Procedural Generation • Level Instancing • Level Streaming

KEYWORDS

Procedural Generation, L-Systems, Splines, Prefabricator, Diamond Square Method, Voronoi Noise.

1 L-Systems

The most challenging aspect of this project was finding an algorithm that we could implement in a short time, with limited Unreal Engine experience. We originally wanted to create cities using L-systems. Specifically using a method described in the Senior Thesis paper, "General Methods For The Generation Of Seamless Procedural Cities", By Tobias Elinder. It was written for the Department of Computer Science and Faculty of Engineering, Lund University. An L-System is a parallel rewriting system that consists of an alphabet of symbols that can be used to make a series of production rules. They were introduced in 1968 by Astrid Lindenmayer, a Hungarian theoretical biologist, and botanist at the University of Utrecht. Lindenmayer used L-systems to model the growth of trees. This algorithm can also be updated to create organic yet sophisticated cities as Elinder described in his paper. The way this works is that agent actor in the Unreal Engine draws a road system that spans across a specified volume. Then a secondary agent actor defines areas which do not overlap with roads where buildings go, followed by an agent that populates those building volumes with random assets. Using some kind of heightmap or noise function, we wanted to create 'neighborhoods', where the appearance of certain structures was more likely than others.

In researching this system further, we realized that we lacked the time to properly implement the techniques used by Elinder in his seamless procedural city project. The demo project with the source plugins is open source and available for download on GitHub, but we were unable to make it operational on our machines. Concluding that we would not be able to implement this algorithm, we looked for other alternatives.

1.1 Tile Based Level Streaming - Overview

The method we settled on was a tile bases generation method. Although we weren't able to get chaotic, and organic cities that we were hoping form, we were able to generate a more structured city using tiles. We came across a youtube tutorial that demonstrated a method of procedurally generating random levels by spawning a series of sublevels on a grid. This technique allowed us to produce a square city, where we could adjust the size indefinitely, and provided a solid framework to expand upon so we could define neighborhoods, and have a system layout that makes sense.

The way this method works, as described above, is by having a persistent level that spawns a series of its sublevels on a grid. Each sublevel contains a 20x20 tile with a prefab on it, which contains a series of predefined assets. These levels contained pieces like big buildings, smaller buildings, tress, general static meshes for clutter, as well as road splines. Each tile was designed to fit perfectly with any other tile in the collection to make a seamless city at runtime. Since we planned to make fantasy cities, we used the 'Advanced Village Asset Pack', which contains a nice collection of fantasy-themed assets, free on the Unreal marketplace.

1.2 Tile Based Level Streaming - Level design

This technique is quite simple, and doesn't require a lot in terms of blueprinting/coding. We opted to use Unreals blueprint system as there were more resources available for learning blueprinting vs C++ scripting in Unreal Engine 4. To start, you will need to create a new level. This newly created Level will act as the persistent level that will contain static assets that will always be loaded. In this level we need to add a few things. Here we put in a landscape with a flat middle plane for our city to go on top of. We added a Sky light, as well as a directional light to illuminate our scene. Make sure to make both of these light movable. Within the main persistent level, we need to create a few sublevels that will act as our tiles that we are streaming in. Create a new sublevel, name it, and make sure to double click that new sublevel in the levels panel to make it the current level. We started by placing a standard cube in the scene. Change the location (x, y, z) to (0, 0, 0) so that the cube is centered. We set the x and y scale to 20 to create a 20x20 plane that is centered in the level. This will serve as the base for our tile so that we can place assets within its dimensions, and connect level seamlessly. Place the assets that you want for that level within the cube. We initially had the idea of creating a city that was sectioned, and contained neighborhoods/districts. Each sections contain different election of levels that had similar buildings. For this project, we created a level that had big houses, one with smaller house, one had trees and other plants to represent a park, one with food stalls and barrels to represent a market. A wealthy neighborhood, a poorer neighborhood, a park, and a market, were the 4 staple sections we decided to add to our city. However, with this model, you can change the assets to be anything you want, to create more varied cities.

With a few city tiles made, it is time to start building the roads to place in between the buildings. We used the same technique as described above to build our road tiles. The roads are made up of a series of cobblestone meshes from the 'Advanced Village' asset pack we used. Instead of having to place and manipulate a ton of individual meshes, we used a spline system for each road. What the spline allowed us to do was instead of having a bunch of cobblestone mesh actors in the scene, we would have one spline actor for each road, that generated a new cobblestone mesh in between every two spline points. The splines are much easier to manipulate, making them perfect for building connecting road. They can also be used to create pipes and rivers.

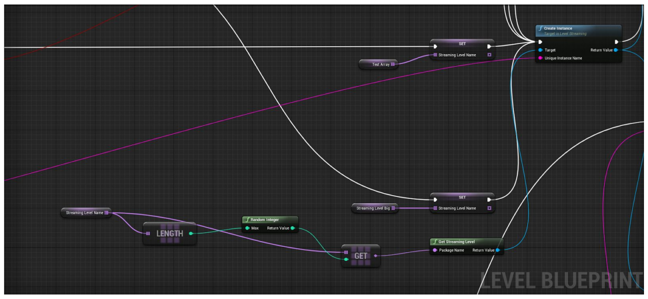
1.3 Tile Based Level Streaming - The Blueprints

In this section, we are going to describe the basic algorithmic approach that we took when generating our procedural city. Included are snippets of the core blueprints and the pseudo code for recreating this technique, which is also relatively simple to expand on.

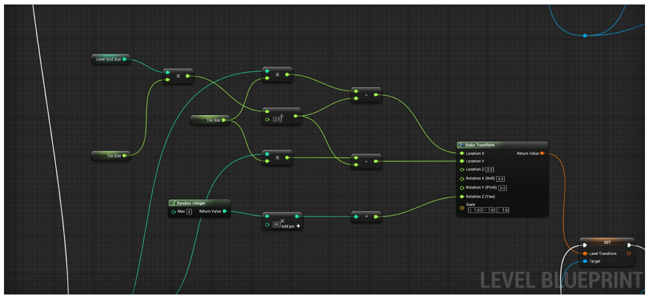
1.3.1 The City Generation

With the levels made, we can now spawn these levels on the map to form on large, coherent city. To do this, we used the Unreals blueprints system as opposed to C++. We would have preferred to build this project all in code, but found that there were many more resources online for building projects with blueprints as opposed to all C++. We made the blueprint in the persistent level blueprint because as soon as we hit play, our collection of levels will be streamed onto the persistent level. The algorithm is very straightforward. Essentially what we are going to do is for each point on the grid, we are going to store the sublevels we made into a name array, and spawn a random one. Them we will go in and place a random road tile on each of the newly spawned levels.

To start, we create a couple of for loops, one for the rows and one for the columns of the grid. These loops should iterate from 0 to the size of the overall level grid, which we will define later. For each iteration of this loop, we want to create an instance of the target streaming level. With this set, we can find a level to stream from. Create a new name array, and set its default values to the names of the sublevels created. We want to grab a random index from the Steaming Level Names array, and we do this by getting a random integer from 0 to the length of the array. This method creates a 100% random map, but the appearance of sublevels can be influenced by adding additional instances of the name to the array. For our project, we could create a city that had more parks and trees by adding two park indexes for every one building index. This blueprint is shown in the figure below.



Now that we have a level to stream for that iteration of the loop, we need to set its transforms and position it. We also need to define the size of each tile, as well as the overall grid size. The grid size is arbitrary integer and can be whatever you want. The tile size will be a float that is equal to 100x the width of your levels. Having the tile size be 100x the width will ensure that your tiles are placed together seamlessly. For this example, we are going to have a tile size of 2000 and a level grid size of 4.



To find the X and Y values of our level tiles, we will calculate as follows.

Y = ((Grid Size * Tile Size)/2) - (Tile Size * Current Index for inner for loop)

X = ((Tile Size * Current Index of the outer for loop)/2) - (Tile Size * Current Index for inner for loop)

These X and Y values are then plugged into the Location X and Location Y ports of our Make transform node. Lastly, to make even more unique cities we are going to rotate each level tile by one to four 90 degree increments.

1.3.2 Road Generation

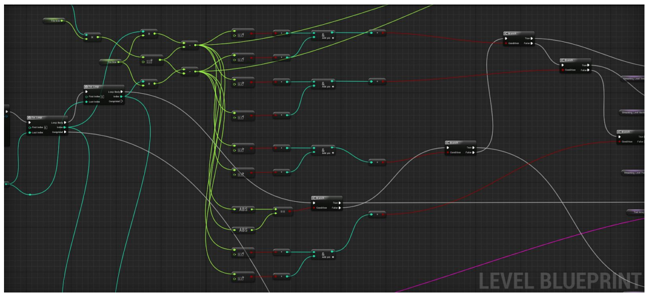
With the city built, we need roads to sit in between the buildings. The method for generating the road system is the same as generating the city tiles. The blueprint described above holds for building the road blueprint, but we need a new array to pull tiles from. Create a new name array and set the default values to be the names of the road tiles. For each point on the grid, we will spawn a random road tile chosen from this array.

1.4 Attempted Variations

After the completion of the initial randomized tile based method, several attempts to create more natural variations between different parts of the generated city were attempted. Detailed below, they were met with varying levels of success.

1.4.1 Quadrant Based Neighborhoods

With the blueprint method described above, we can create seamless, but 100% random cities with no sort of layout or structure to them. We wanted to initially create neighborhoods into our cities using the alternative methods we looked into. Tile-based city generation didn't make sectioning off the city very easy, but expanding on the base blueprint isn't very difficult. One way of generating neighborhoods was to section the city off by quadrants. All 4 quadrants would pull from a different array, each containing unique levels. To do this, we check the values of our X and Y coordinates before we get the name of our level to stream. The values of the X and Y coordinates determine which name array we feed to our level instance creator.

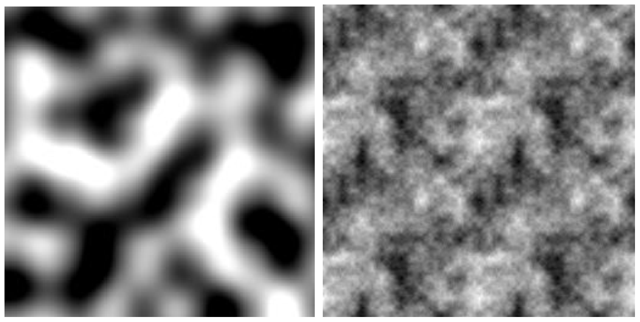


Here we have a series of if statements that check the value of X and Y for that iteration of the for loop. The way we have it set up is if X and Y are both greater than zero, we pull from the array of the big house. If X and Y are both less than zero, we pull from the array of the small house. The other 2 quadrants will contain a mix of all of the levels that we created. There is still a degree of randomness involved, but we can influence which tile goes where by assigning these rules to specific points, that way our city won't have a smaller home next to a bunch of big estates.

1.4.2 Noise Based Neighborhoods

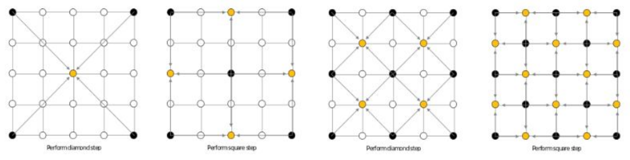
Our first attempt to escape the constraints of tiled level generation was to use noise to create neighborhoods. Perlin noise is used to create smoothly varying randomness, rather than being truly random. In using it, we could get smooth variation between neighborhoods without being confined by the grids of the quadrant based neighborhoods. Unfortunately, the two-dimensional perlin noise required for this specific application did not come natively in the Unreal blueprint library, and time constraints prevented us from completing experimentation with implementing our own blueprint based version of perlin noise.

We also briefly attempted to create a version that used the more simple value noise, which seeds random points and then linearly interpolates between them for a more gridlike division than perlin noise. This ran into many of the same problems as perlin noise when attempting to implement it in Unreal Blueprints.


Perlin Noise (left) and Value Noise

1.4.3 Diamond Square Based Neighborhoods

Our next attempt at creating neighborhoods was to use the Diamond Square algorithm to create a heightmap. Different neighborhoods would then be designated to occupy different elevations of the heightmap. The benefit of Diamond Square over the noise algorithms is that it does not need to produce an infinite sample space. Instead, it creates a single height map of a limited size. The values of the four corners are selected at random, then the center point is calculated as the average of the four corners plus or minus a small amount. This is known as the diamond step. Next, the four points orthogonal to the center at the distance of the corners are calculated as the average of its orthogonal neighbors at that distance, again plus or minus a small amount. This is known as the square step. At the end of the square step, the region has been subdivided into four on which the diamond step can be performed. The process is repeated recursively until the desired size is reached.



While we eventually had an algorithm for a diamond square that compiled and ran, it did not produce expected results. Unfortunately, Unreal Blueprints proved too difficult to debug in the time we had remaining to complete this project.

1.5 Future Improvements

We consider this project to be a work in progress. This section will detail a few of the improvements we intend to make to the project going forward.

1.5.1 Completed Neighborhoods

We believe that many of our ideas on creating more sensible neighborhoods in our maps have the potential to succeed. Our first goal will be to implement one of them and begin producing more logically ordered towns.

1.5.2 Escaping Grids

Several methods that we did not have time to fully explore could exist for hiding the gridlike nature of the tile-based generation. If for example, we have the ability to query neighboring tiles, or to more finely control where tiles might appear, we can start making tiles of varied sizes.

In addition, if we can look at neighboring tiles we can begin creating a system in which roads go through the center of tiles rather than tracing the edges. In such a system, the road could be drawn to curve more naturally leading to a more varied city.

1.5.3 Smart Road Splines

As of right now, our roads are there own tile and are spawned at random on top of the building tiles. There are designed in a way so that they, for the most part, all connected. As we improve this project, we want to implement an agent that draws roads automatically between the buildings on the map, as one continuous spline with junctions.

2 The Plugins and Assets

As far as our deliverables, we didn't use many marketplaces and/or community resources. For our tiles, we used the UE4 Prefabricator Plugin by Code Respawn, available for free on the Unreal Engine Marketplace. This plugin allowed us to store groups of meshes into one reusable asset, which became the core component of the levels we designed. The asset pack that we used was the Advanced Village Pack, by Advanced Asset Packs, also available for free on the Unreal Engine Marketplace. This contained a nice collection of fantasy-themed assets that we thought would be perfect for generating D&D style cities.

ACKNOWLEDGMENTS

Special thanks to our Professor, Angus Forbes, and our T.A Montana Fowler.

REFERENCES

[1] Tobias Elinder. General Methods for the Generation of Seamless Procedural Cities, 2017. Department of Computer Science Faculty of Engineering, Lund University.
[2] "[UE4] Procedural City with Full Interiors - Workflow Demo 1." *youtube.com*. Coqui Games, 17 Aug. 2018.
[3] "UE4 - Procedural Level Generation" https://www.youtube.com/watch?v=VmRggTwhiew&t=148s, Pub Games, 30 June. 2016.
[4] O'Brien, Nick. "Diamond-Square Algorithm Explanation and C Implementation." *Medium*, Medium, 14 Aug. 2018, https://medium.com/@nickobrien/diamond-square-algorithm-explanation-and-c-implementation-5efa891e486f.