Mapinator

by

Gustavo Reyes, Alejandro Matos, Antonio Lugo, Jamel Peralta

Introduction:

Mapinator is a simple and easy to use map creation language. Utilizing the Unity engine and python, it produces a simple procedurally generated map with customizable parameters. Once the language is used to create a desired map, it generates C# scripts that can be run and rendered using the Unity game engine. The main components of this language are the parser, lexer and C# scripts. The main function of this language is the creation of complex C# scripts from simple text inputs.

The language prioritizes functionality and ease of use, therefore the syntax required to use it is relatively simple. The main goal of this language is to make map creation and game development more accessible to creators, allowing people to code complex map algorithms with a few simple commands. The functions and properties that the language can create and edit means game-makers are free to focus on more important tasks, letting the language do most of the difficult scripting work.

Language Tutorial:

Setting up Mapinator:

1. Install Python 3.
2. Install JetBrain's Pycharm.
3. Install Unity.
4. Download PL_Project.zip from the GitHub page.
5. Extract all the files.

Running Mapinator:

1. Open the folder PL_Project in JetBrain's Pycharm IDE.
2. Search for the text file called "MapinatorCode.txt" and write your code there.
3. When the code is ready, run the MainClass.py.
4. Search for the "output" folder and import all the scripts to Unity.

Making your first map using Mapinator:

1. First, we need to create an instance of the map. This is the map that we will modify with the other commands. We need to type:

   GenerateMap myMap

2. Now we need to set the size of the map. For that, we will do:

   SetMapSize 500 500

3. We create an id for the map using the command:

   SetSeed myMap 1

4. Next, we will modify the distance between higher peaks. To do this, use:

   SetLacunarity myMap 4

5. We need to control the level of details shown by the map. Use the command:

   SetDetails myMap 4

6. To control the general height of the map, we write the command:

   SetHeightMult myMap 10

7. We need to modify the value to represent the map in a flat plane. Type in:

   SetNoiseScale myMap 20

8. Now we want to control how elevations are distributed or clumped together.

   SetNoiseDensity myMap 8

9. To control how much high values affect the map, we can use:

   SetPersistence myMap 1

10. To finalize the code, simply use:

    GenerateCode myMap

Language Reference Manual :

Commands:

1. GenerateMap map_name: Creates a map with the specified name.
2. SetMapSize map_name number number: Creates a map with the specified size.
3. SetSeed map_name number: Creates a random id used to generate the noise map.
4. SetLacunarity map_name number: It is a value to controls distance between peaks.
5. SetDetails map_name number: It is a value to controls the level of details to show the map.
6. SetHeightMult map_name number: It is a value to controls the general height of the map.
7. SetNoiseScale map_name number: It is the scale to represent the map in a flat plane.
8. SetNoiseDensity map_name number: It is a value to control how elevations are distributed or clumped together.
9. SetPersistence map_name number: It is a value to control how higher values affect the map.
10. GenerateCode map_name: Command need to finalize the code.

Language information:

For a program to work steps must be taken in a certain order:

1. Make sure you're writing your code in "MapinatorCode.txt" inside the "PL_Project" folder inside JetBrain's Pycharm IDE .
2. Make sure you put values for these variables and run the MainClass.py.
    a. size
    b. id
    c. lacunarity
    d. details
    e. heightMult

f. noiseScale

g. noiseDensity

h. persistence

3. Import all the scripts from the “output” folder to Unity and run.

The only user action needed besides data input is importing files into Unity, cutting hours of development time for anyone trying to create maps in Unity.

Language Development:

Translator Architecture:

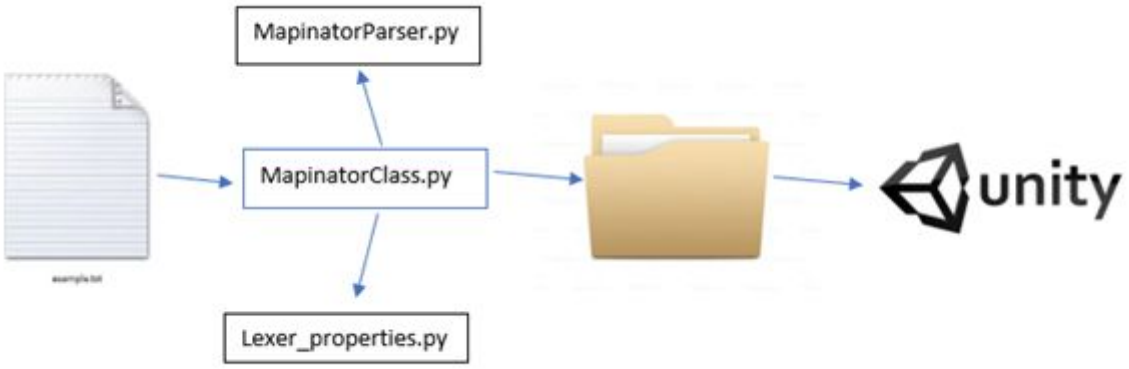

Image: A possible result from the code of Mapinator.

Interfaces between the modules:

The user's code from the "MapinatorCode.txt" is handled by MapinatorClass.py where the parser and lexer comb through it and look for the parameter altering commands. Once obtained, the parser sends the parameter values to the GenerateMap class which writes the necessary C# scripts into the "output" folder. These scripts are then imported into Unity for compilation and map generation.

Software development environment:

Mapinator was developed using JetBrain's Pycharm and Visual Studio Code. The lexer tool used was PLY, which is an implementation of lex and yacc. The parser tool was created from scratch using native methods from the string class in Python to analyze the code given. Once the string was analyzed, the parser object class would link those parameters to the intermediate code called GenerateMap. GenerateMap is an object class that basically stores in its instance variables all the data needed to generate the code in C#. As soon as the code hit GenerateCode map_name, the parser would link this to GenerateMap.generatecode() which will create the C# source code and automatically open the finder where this file is located. Unity was used to run and compile the resulting C# scripts. The C# scripts themselves are an implementation of the Perlin Noise algorithm modified to be represented as a mesh generator to simulate an environment.

Test Methodology used during development:

Testing was done in parts, the three main components of the language were the parser, lexer and C# scripts. The testing for each component was mostly separate through early development. The lexer testing was focused on handling tokens and symbols required for the language. The parser testing was mainly related to generating proper C# scripts, modifying them, and handling them as inputs and outputs. These tests were accomplished by using test scripts and texts which simulated edge cases and possible user input commands. The last component that was tested independently was the C# map generator scripts. The testing for these was mostly focused on properly generating the meshes from the perlin noise

map and being able to change its attributes effectively. Once all the individual components had been properly tested, they were tested while connected together. The language was tested by inputting different text files with different commands, once it outputted the appropriate C# scripts these were directly tested in Unity to see if they behaved appropriately.

Translator Test Program:

```
GenerateMap mymap
SetMapSize mymap 50 50
SetSeed mymap 0
SetLacunarity mymap 4
SetDetails mymap 4
SetHeightMult mymap 10
SetNoiseScale mymap 20
SetNoiseDensity mymap 8
SetPersistance mymap 1
GenerateCode mymap
```

This code will create a folder with all the scripts needed by Unity to create the map.

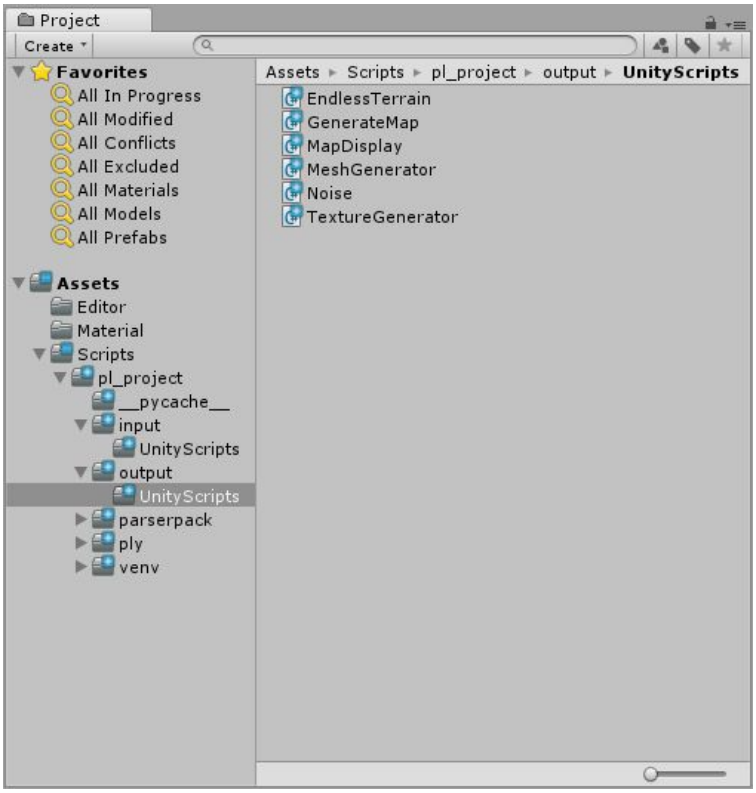

Image: Shows the folder with all the scripts.

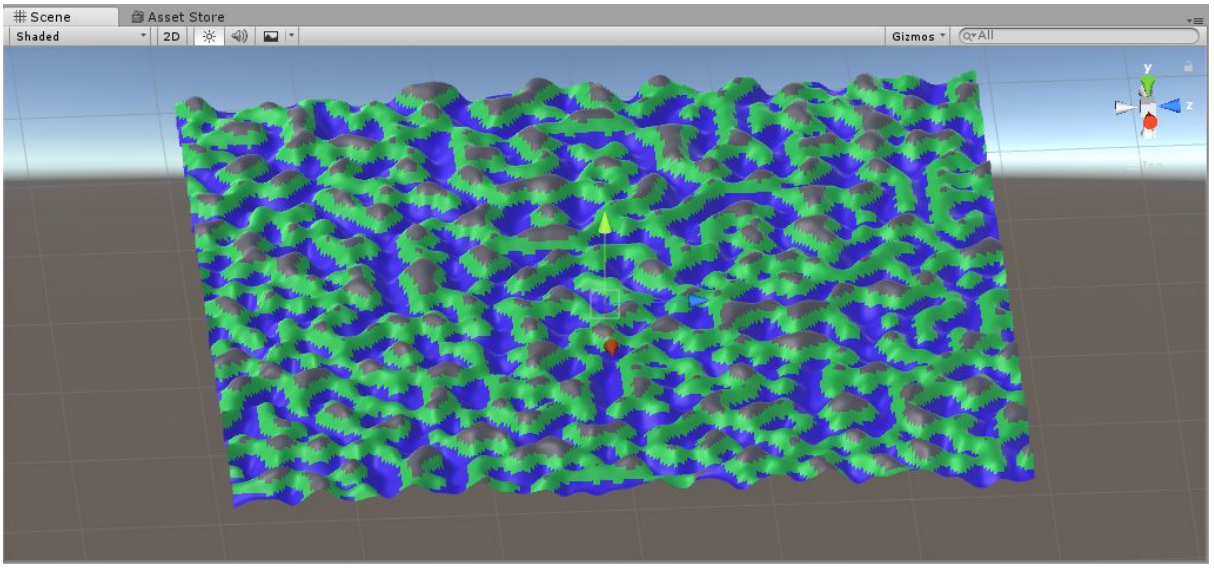

Image: Shows the map created by the previous code and scripts.

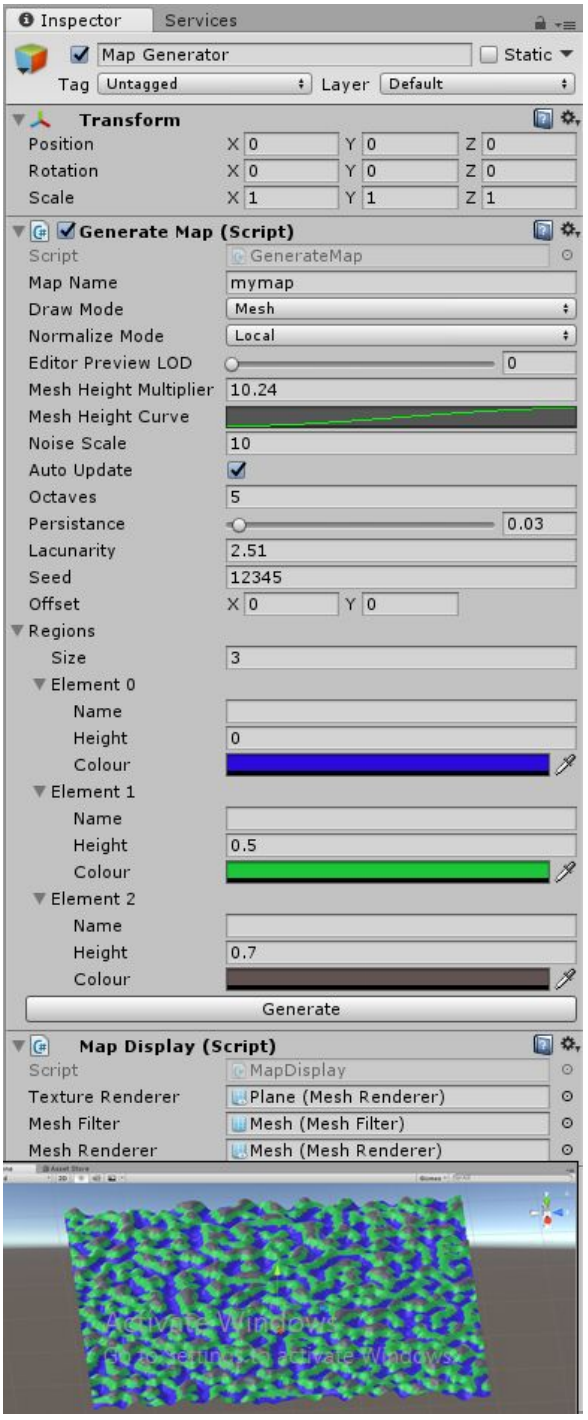

Image: Shows the properties of the map.

Conclusion:

The Mapinator programming language makes it easy to create a complex map system for games. Allowing for quick prototyping and deployment of game environments. The language eliminates the need for complex programming and scripting, allowing novice users to quickly get started making games in unity. The advantages of this language are its flexibility and ease of use, and in the future its potential to streamline even more complex game making tasks. Certain other features such as biome creation and editing are also possible future additions. The language itself is a prototype demonstrating the possibilities of a script generation system for Unity. The biggest hurdle when developing games is the initial programming complexity, which turns off newcomers and more art oriented game makers. With tools like Mapinator at their disposal, these creators can rapidly begin prototyping game systems with ease.