Lindenmayer Systems:
An Unreal Engine 5 L-System Generator

Dinh Huy Henry Ha
Student ID: 33871483

Mathematics for Games and VR/AR
MSc Games Programming

Github: https://github.com/HenryHa993/L-Trees
Video: https://youtu.be/3PYcd5rQyrM
Download: link

November 27, 2024

Abstract

The objective of this project was to produce a customisable L-System generator, which is intuitive and fun to use for users. This project was implemented in Unreal Engine, using C++ for the generator and blueprints for UI.

Keywords— L-Systems, Unreal Engine, C++, Blueprints

Contents

1 Features **3**
- 1.1 User Interface . . . 3
- 1.2 Presets . . . 4
- 1.3 3D L-Systems . . . 4
- 1.4 Hotkeys . . . 4

2 Classes **5**
- 2.1 LSystemGeneratorSubsystem . . . 5
- 2.2 LSystemGeneration . . . 6
- 2.3 LSystemStructs . . . 6
- 2.4 User Interface . . . 6
- 2.5 Presets . . . 7

3 Results **8**

4 Conclusion **10**

Chapter 1

Features

1.1 User Interface

The primary goal behind the user interface was ease-of-use. To achieve this, the UI had to clearly outline what parameters were required to generate an L-System as well as allow the user to edit and add them.

To achieve this, the UI is broken down into three sub-menus, as inspired by Finn Tanner's L-System Tool [1]. These sub-menus include the main settings, rule settings and function settings. The main settings enable users to adjust values such as the axiom and angle, encapsulating the core parameters of an L-System. The rules settings allow the user to map multiple variables to a string. Lastly, the function settings allow the user to map constants and variables to a specific function.

When the user enters an invalid value, these values are flagged and the field is tinted red to warn the user. This brings clarity to the user interface, clearly outlining what kinds of values are expected from the field. If these values remain when the user attempts to generate an L-System, they are simply skipped.

Some existing L-System generators offer limited extensibility to the user. For example, a finite number of rules or fixed function mappings that the user can work with [2]. To improve on this, the project allows the user to add/delete as many rules and functions as required. The system also allows for the user to map variables and constants to a range of different functions. This is useful, as grammars are often inconsistent between different L-Systems found online. One such example can be seen through the difference in the rules presented in [3] and [4].

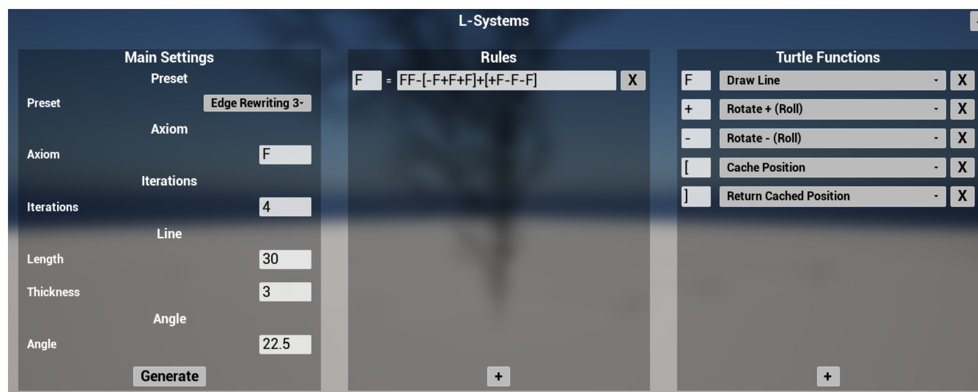


Figure 1.1: L-Systems settings menu.

1.2 Presets

Another feature this project offers is loading of presets. This can be done through either hotkeys (1-8) or through the L-Systems menu. Loading a preset using either technique will in turn populate the fields of the L-System menu and generate the system.

Populating the fields allows the user to explore these presets. The user is able to edit the values as well as add their own additional rules/functions as they wish. Manipulating existing L-Systems is a great way of getting used to the UI and introducing someone to the topic of L-Systems.

The presets include a range of L-Systems, namely node-rewriting and edge-rewriting systems. Amongst them are also 3D L-Systems; a 3D Hilbert Cube [4] and a 3D-2 bracketed tree [5].

1.3 3D L-Systems

There are multiple 3D L-Systems online which utilise additional rotation functions about the pitch and yaw [4][5]. In combination with common 2D L-System functions [3], the system can generate 3D L-Systems such as the tree seen in Figure 1.2.

Figure 1.2: 3D Bracketed Tree.

1.4 Hotkeys

Other quality of life features include the ability to increment and decrement values found within the main settings. These settings include the line length, line thickness, angle and iteration. These allow the user to see how different parameters affect the look of the tree.

Movement:	L-System	Iteration	Angle	Line Length	Line Thickness
WASD : Move	TAB : L-System Menu	O : --	K : --	N : --	V : --
Mouse : Look	1 - 8 : Presets	P : ++	L : ++	M : ++	B : ++

Figure 1.3: Hotkeys available to the user.

Chapter 2

Classes

2.1 LSystemGeneratorSubsystem

Found: *Source/FlecsTest/LSystemGeneratorSubsystem.cpp*

This class stores the rules, functions and default parameters used to generate and draw an L-System. Rules and functions are implemented using `TMap<TCHAR, FString>` and `TMap<TCHAR, FunctionType>` respectively. Maps provide efficient lookups, making it a better alternative to traversing an array or list. They map characters to specific strings and function types, making this generator suitable for context-free L-Systems. To add to these maps, `AddRule(...)` and `AddFunction(...)` are used. These filter out invalid inputs by checking string lengths, ensuring invalid rules are skipped. This is primarily called by the UI upon clicking the generate button. This is made possible using the `UFUNCTION(BlueprintCallable)` macro.

Once parameters are set, the subsystem can generate a string based on the rules stored. This is done through the `Generate()` and `GenerateWithIterations(int Iterations)` methods. The former represents a single iteration on the current string (`CurrentString`), using the `Rules` map. When called, it converts `CurrentString` to a character array and traverses it. If a character exists as a key in `Rules`, it's mapped string value is added to a new string. Otherwise, the character itself is added. Once the array is traversed through, the new string is stored as the new `CurrentString`. The latter of the generate functions runs `Generate()` a number of times, depending on the input number (`Iterations`). This allows a specific number of iterations to take place from a single function call, instead of multiple `Generate()` calls. Both functions are implemented under the `UFUNCTION(BlueprintCallable)` macro, allowing UI and other blueprints to call them directly.

This class is also responsible for drawing the tree. The `Draw(AActor& Actor)` method is intended to take an actor reference. This allows the line drawings to take place from the actors origin, allowing specific placement in the scene. Once called, the character array generated from the `CurrentString` is traversed. If a character is found in the functions map (`TurtleFunctions`), the mapped `FunctionType` enum is put through a switch statement and the corresponding drawing function is run.

The first of the drawing functions allows for rotations about the three principal axis; the roll, pitch and yaw. If the user intends to draw in 2D, it is advised to use the yaw functions. To reduce confusion, the function drop-down menu groups the functions required for 2D at the top, and puts more 3D oriented rotation functions at the bottom [Fig. 2.1].

Other drawing functions include transform caching. Implementing this required a LIFO (Last-In First-Out) data structure, which was done using a `TArray`. This was intuitive as it has both `push()` and `pop()` functions required. When a character mapped to the caching function is found, it will simply push the actor's transform onto the stack. When a character mapped to the 'return to cache' function is found, the actor's transform is set to the popped transform.

Lastly, the line drawing function translates the actor's transform component forwards by a distance depending on the line length. It then draws a line between the previous position and the new position.

This class inherits from the `UGameInstanceSubsystem`, which functionally makes the class a singleton. This is appropriate for the project, as global access allows for the UI elements to store parameters and call methods directly from a global generator.

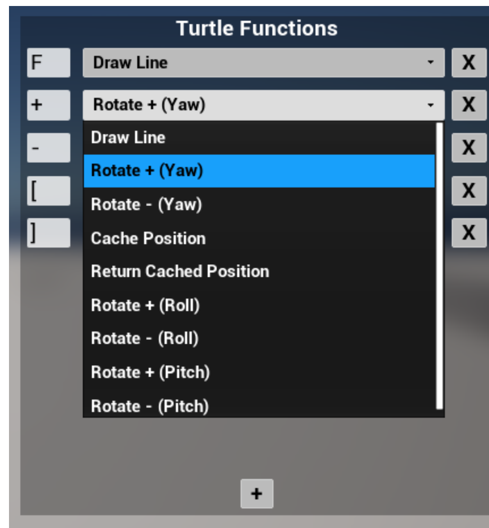


Figure 2.1: Function drop-down menu.

2.2 LSystemGeneration

Found: *Source/FlecsTest/LSystemGeneration.cpp*

This class is responsible for drawing L-Systems to a specific location. It has a blueprint callable function called `Draw()`. This calls the `LSystemGeneratorSubsystem`'s `Draw(...)` method, inserting a self reference as input. This allows the system to be drawn to the location where the actor is placed in the scene. Each time `LSystemGeneratorSubsystem` generates an L-System, it broadcasts its `FOnGenerate` delegate. This triggers each `LSystemGeneration`'s `Draw()` call. By default, all objects inheriting from the `LSystemGeneration` class will add their `Draw()` function to the delegate on `BeginPlay()`. The `BP_TestGeneration` blueprint inherits from this C++ class and can be found in the centre of the scene.

2.3 LSystemStructs

Found: *Source/FlecsTest/LSystemStructs.h*

This class is consists of the structs and enums used in `LSystemGeneratorSubsystem`. In initial planning, it was expected that the project may need additional structs for the customisation of the tree, however these were not required. Instead, it stores the `FunctionType` enum, which specifies the type of function assigned to a character. It is used in the `TurtleFunctions` map and in Unreal blueprints for the presets. See Figure 2.2.

2.4 User Interface

Found: *Content/L-Trees/UI*

User interface is broken up into multiple blueprint classes. All changes to L-System parameters should be reflected in the menu, making the UI the centrepiece of the project. Any adjustments made through hotkeys or presets goes through the UI in order to adjust the appropriate fields to reflect the current generation.

This is done by storing the parameters separately in the UI itself. Once the user presses the generate button any values, rules and functions stored in `WBP_LSystemInterface` are passed to the `LSystemGeneratorSubsystem`. There, the values will be checked and stored if valid. The generation will then take place.

```
4   UENUM(BlueprintType)
5   enum FunctionType
6   {
7       Line,
8       RotatePositiveRoll,
9       RotateNegativeRoll,
10      Cache,
11      ReturnCache,
12      RotatePositiveYaw,
13      RotateNegativeYaw,
14      RotatePositivePitch,
15      RotateNegativePitch
16  };
```

Figure 2.2: `FunctionType` enum.

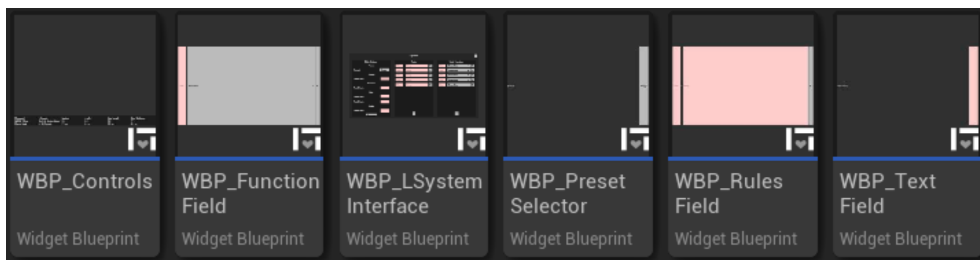


Figure 2.3: User interface components.

2.5 Presets

Found: *Content/L-Trees/Core/Presets*

As UI is done using blueprints, it was appropriate to implement preset classes at this level also. This is because preset values should be reflected in the UI as well as the drawing. `BP_PresetLSystemBase` is the base class for all presets, and encapsulates all the parameters required to generate and draw a system.

Axiom	
Angle	
Line Length	
Iterations	
Rule Inputs	0 Array element
Rule Outputs	0 Array element
Function Inputs	0 Array element
Function Outputs	0 Array element

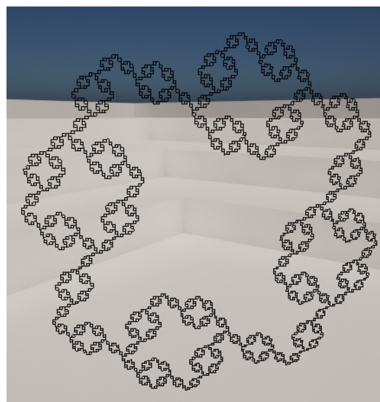
Figure 2.4: Preset base parameters.

Chapter 3

Results

The result is an application which can generate and draw a both 2D and 3D L-Systems entirely from the UI. This gives the user the ability to fully form their own L-Systems, as well as edit pre-existing systems included in the project. The UI is intuitive, allowing the user to add and delete any amount of rules and functions. In addition, the application prevents invalid inputs on both a UI and systems level. Lastly, the user can edit any main settings through use of hotkeys.

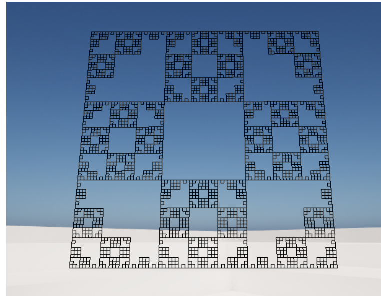
To evaluate the generator itself, I will put a side-by-side comparing variations of the Koch Curve L-Systems found from *The Algorithmic Beauty of Plants* [6] to the drawings produced by my app.



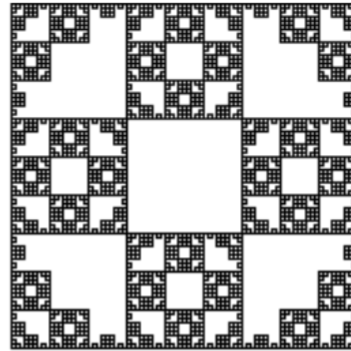
(a) Project generation

(b) *The Algorithmic Beauty of Plants*

Figure 3.1: Koch Curve variation 1: FF-F-F-F-F-F+F

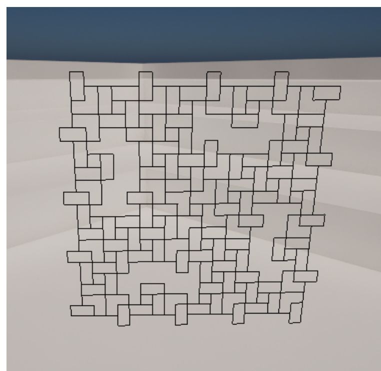


(a) label 1

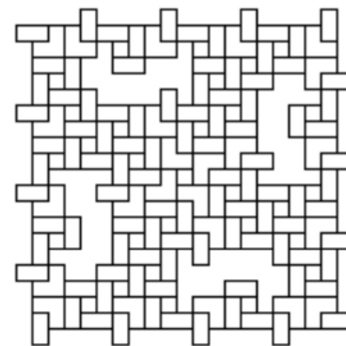


(b) Project generation

Figure 3.2: Koch Curve variation 2: FF-F-F-F-F-FF

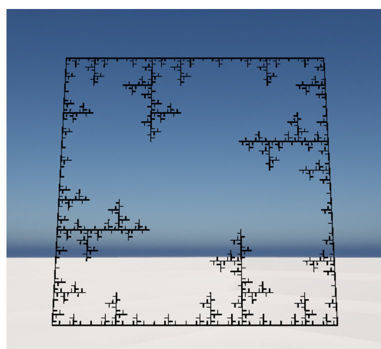


(a) Project generation

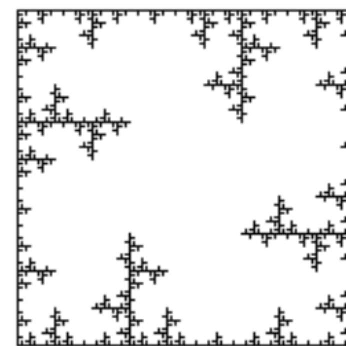


(b) *The Algorithmic Beauty of Plants*

Figure 3.3: Koch Curve variation 3: FF-F+F-F-FF



(a) Project generation



(b) *The Algorithmic Beauty of Plants*

Figure 3.4: Koch Curve variation 4: FF-F–F-F

Chapter 4

Conclusion

Despite achieving the goals set out from the beginning of development, there are improvements the project would still have benefit from. For example, the addition of stochastic L-Systems would have given the user another way to explore included L-Systems. Moreover, this would have enhanced the look of 3D L-Systems, giving them a more variation between generations.

In addition, it was initially planned to line customisation and leaves to trees. The latter would have been achieved by identifying and saving positions located before a return-to-cache function.

Overall, this was an enjoyable project which enhanced my knowledge of both tools development and L-Systems. Originally, I did want to develop this using an ECS System, however that was out of the scope of the project. As this project is built on-top of an ECS template, this is something I may do in the future.

Bibliography

[1] F. H. Tanner, “My L System Tool.” YouTube, Feb. 2022. [Online]. Available: https://www.youtube.com/watch?v=JUjdamZvicE.

[2] Online Tools, “L-System Generator - Online Tools.” Online Tools. [Online]. Available: https://onlinetools.com/math/l-system-generator.

[3] P. Bourke, “L-System Fractals.” Paul Bourke, July 1991. [Online]. Available: https://paulbourke.net/fractals/lsys/.

[4] arussell, “3D L-System with C and WPF.” CodeProject, Dec. 2014. [Online]. Available: https://www.codeproject.com/Articles/855693/3D-L-System-with-Csharp-and-WPF.

[5] H.-W. Chen, “L-System Plant Geometry Generator.” Cornell University, Jan. 1995. [Online]. Available: https://people.ece.cornell.edu/land/OldStudentProjects/cs490-94to95/hwchen/.

[6] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. New York, NY, USA: Springer-Verlag, 1990. Chapter 1: Graphical Modeling Using L-Systems.