L-system for generation of trees in Unity

Manuel Pagliuca - 975169
October 19, 2022

Abstract

Modern machines have made it possible to break through hardware limitations that stood in the way of older generations of video games. The use of *procedural content generation* has been widely used in the video game industry for the creation of *pseudo*-randomly generated assets, such as *maps*, *levels*, *characters* and other objects that make the game experience unique. The goal of this project is to implement an L-system for the generation of *pseudo*-random *trees*. Basic knowledge about *language theory* is essential to implement the system to generate and manage the language that will describe the trees.

The paper will begin by introducing L-systems, the objectives of the project, and the development tools used. This will be followed by a discussion of the technical implementation of the project. The result obtained is an application capable of producing fairly realistic results in a smooth way (considering a limited number of iterations/recursion levels), offering four different bases for tree generation (one for roots) with the possibility to make changes to the production rules in real-time.

1 Introduction

Lindenmayer's system is a parallel system of *rewrite* (in mathematics, the operation of "rewriting" consists in the substitution of an object in place of a part of another object, according to a precise formal rule) and a type of **formal grammar**. It was introduced and developed by *Aristid Lindenmayer* in 1968 [3], a Hungarian theoretical biologist and botanist who used this system to describe the behavior of plant cells and to model plant growth processes (fig. 1). Because of its features, like its *recursive* nature, L-systems are widely used for *fractal* generation (as the recursion leads to the *self-similarity*).

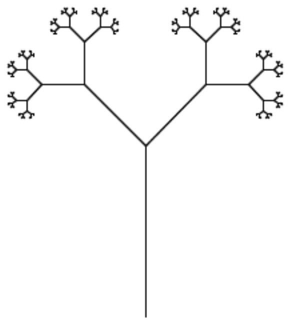


Figure 1: An example of a tree generated with an L-system using (and reusing) segments. Its fractal structure (*self-similarity*) can be easily seen there.

L-systems grammars are very similar to the *semi-Thue* grammar, they are commonly defined as a tuple:

$$G = (V, \omega, P)$$

where :

- V is a **set of symbols** (*alphabet*) containing both the elements that can be *replaced* and those which cannot be replaced (*terminals* or *constants*).
- ω is the **axiom** which is a string of one or more symbols that defines the initial state of the system.
- P is the set of **production rules** defining the way variables can be replaced with combinations of constants and other variables.

The rules of the L-system grammar are applied iteratively starting from the initial state. As many rules as possible are applied *simultaneously*, per iteration. If the production rules were to be applied only one at a time (in a sequential manner), one would quite simply generate a *language*, rather than an L-system (L-systems are strict subsets of languages).

L-system realized in this project consists of a single production rule and an axiom X.

$$X \rightarrow Production\ Rule$$

In addition to the theoretical basis to generate our L-system, we need a mechanism to utilize the generated string in geometric structures. Then we need symbols belonging to the string to act as references for procedures that draw graphics on the screen (similar to *turtle/vector graphics*).

The goal of this project is to realize a context-free and deterministic L-system for the generation of *pseudo*-random trees [8], in which the direction and the angle of branches (and roots) are not constant but randomly sampled from distribution in order to obtain a realistic effect (fig. 2).

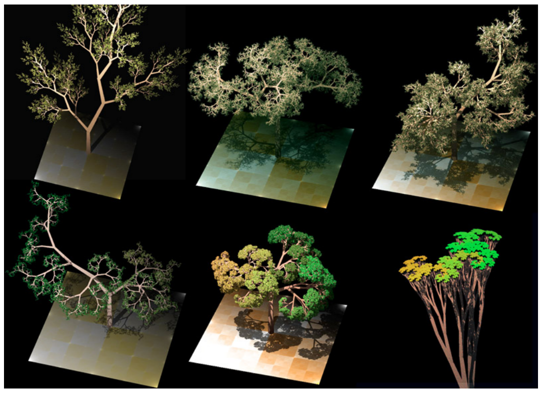


Figure 2: Realistic L-system trees.

1.1 The tree structure

The desired tree must be conceptually like the one depicted (fig.3). Therefore it must have *branches, leaves* and *roots*. In the project, several production rule models will be provided for branches (these will allow different geometric developments of the trees) and for roots.

Figure 3: Tree concept.

2 Development tools

To implement the project, the following tools were used :

- **Game engine**, Unity 2019.4.13f1 [10]
- **IDE** (*Integrated Development Environment*) Visual Studio 2019 [7], since the debugger was very well integrated with the Unity engine.
- **Programming language**, C# [2].
- **VCS** (*Version Control Systen*), Git [4] & GitHub [6].

In addition to these tools, online resources such as *Wikipedia* [1], slides from the course itself and the online book "*The Algorithmic Beauty of Plants*" [9].

3 Unity implementation

3.1 Project structure

The project is set in a bare scene containing only a terrain manually created using the modeling tools provided by Unity, to which a simple texture has been applied (the origin of the assets is mentioned at the end of this section).

There are four main C# scripts in the project, each script will be discussed in a separate subsection. They have been built respecting the "*Single Responsibility Principle*" (*SRP*, where the '*S*' from the *SOLID* [5] acronym). The scripts and their respective tasks are :

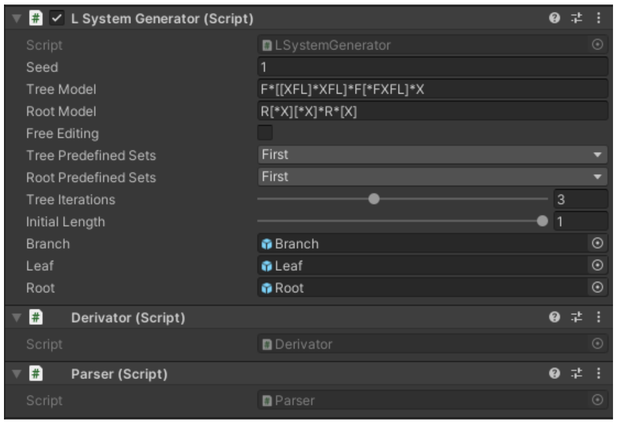
- *CameraMovement.cs*, deals with the movements of the player within the scene.
- *LSystemGenerator.cs*, deals with the graphical generation of the tree, using the derivator and the parser.
- *Derivator.cs*, takes care of deriving the initial model for the specified number of iterations.
- *Parser.cs*, is responsible for parsing the derived string.

3.2 Main objects

The objects present in the scene are :

- Main camera
- Directional light
- Terrain
- Origin

With the exception of the *Origin* object, the others are of minor relevance to the purpose of the project. The *Origin* is the point from which the tree will be generated upward at first (branches) and then downward (roots). The three main scripts for generating the tree are applied to this object (*LSystemGenerator.cs*, *Derivator.cs* and *Parser.cs*).



L System Generator (Script)

Script	LSystemGenerator
Seed	1
Tree Model	F*[[XFL]*XFL]*F[*FXFL]*X
Root Model	R[*X][*X]*R*[X]
Free Editing	
Tree Predefined Sets	First
Root Predefined Sets	First
Tree Iterations	3
Initial Length	1
Branch	Branch
Leaf	Leaf
Root	Root

Derivator (Script)

Script	Derivator

Parser (Script)

Script	Parser

Figure 4: Scripts attached to the "*Origin*" object.

The "*Main Camera*" object will have the *CameraMovement.cs* script attached to handle scene navigation.

3.3 Camera movement

This script will not be discussed in detail as it is not relevant to the scope of the project, but it needs to be explained as it provides convenient functionality for players in running the application. The movements provided by the script are :

- Classic first-person movements with the *W, A, S* and *D* keys.
- The *CTRL* key is used to move up vertically (on the positive *Z* axis).
- The spacebar key is used to move down vertically (on the negative *Z* axis).

- The *SHIFT* key, when pressed in combination with the *W, A, S* and *D* keys, provides acceleration of movement in the specified direction.

3.4 Derivator

This class is *used* by the L-system generator itself (*LSystemGenerator.cs*), which will take care of calling the main "*Derive()*" (listing 1) method providing the desired number of iterations as input. There is no control over this number by the *Derivator.cs* class.

The method consists of two nested for loops, the first one iterates for a number of times equal to the variable "*iterations*", and the second one iterates on each character of the string provided as input.

Initially the variable "*derivedString*" corresponds to the axiom symbol, this is set through the "*SetAxiomAndRules()*" setter of the derivator class, which takes care of passing the axiom and the production rules.

Listing 1: *Derive()* method from Derivator.cs

```
public string Derive(int iterations)
{
    StringBuilder buffer = new
       StringBuilder();

    for (int i = 0; i < iterations; i++)
    {
        foreach (char c in derivedString)
            buffer
            .Append(rules.ContainsKey(c)?
             rules[c] : c.ToString());

        derivedString =
           buffer.ToString();
        buffer = new StringBuilder();
    }

    return derivedString;
}
```

3.4.1 Time complexity

For each iteration of the outer loop, we will replace the axiom X with the respective production rule, proceeding in this way we will obtain an exponential expansion of the initial string. Considering the following production rule :

$$X \rightarrow F[-X][+X]$$

During the computation of the method (listing 1), the production rules, represented by the variable "*rules*" (dictionary/hash-table type) will contain the previously exposed association. Inside the variable "*derivedString*" there will be the axiom X. Then, for a number of times equal to "*iterations*" whenever it will encounter the axiom, it will substitute with the rule. The complexity of this algorithm will depend on the number of iterations of the external cycle times the number of axioms per iteration (which in turn depends on the selected model).

$$O(\#outerIterations \cdot \#axiomsPerIterations)$$

We can see that the number of axioms (the *base*) as well as the instruction in the nested for loop will increase exponentially. This is because at each iteration the number of axioms will be the next power of the number of axioms at the previous iteration (check the table 1).

Although in this algorithm the number of iterations is fixed, the logic is the same as in the recursive algorithm for computing the *Fibonacci* sequence with the difference, only that it is written with two for loops, where the number of recursive calls (*branches*) is equal to the number of axioms in the production rule (in the example we are considering there are two $F[-X][+X]$).

Table 1: Example of a few iterations considering the previous production rule $X \rightarrow F[-X][+X]$

Iteration	Derived String
0	X
1	F[-X][+X]
2	F[-F[-X][+X]][+F[-X][+X]]
3	F[-F[-F[-X][+X]][+F[-X][+X]]][+F[-F[-X][+X]][+F[-X][+X]]]
4	...

For each outer iteration, the number of inner iterations will grow exponentially with respect to the base, in this case, the base is 2 (there are two X in the production rule), so the number of instructions would be equal to :

$$2^0 + 2^1 + 2^2 + ... + 2^n$$

where n is the number of selected iterations (which is the depth of the call tree), then it would be enough to simply apply the *sum of powers of 2* to know the number of instructions executed in the loop, which will be $2^n - 1$. The overall complexity of the two nested for loops can be described as :

$$O\left(2^n - 1\right) = O\left(2^n\right)$$

However, two things should be considered :

- the number of iterations of the inner loop depends on, but is **not** equal to, the number of axioms in the string. The inner for loop iterates over all characters in the string in addition to the axioms (if we considered only the number of axioms we are making underestimate of time complexity).
- the input n (the number of outer iterations) in this case is not a number that can grow much since it is limited internally by a range of iterations between [1,6] (fig. 10).

Thus it is an iterative algorithm with a time complexity that can vary from the best case of about $\approx O(1)$ (*linear complexity*), to the worst case of about $\approx O(2^6)$ (*sixfold complexity*), both *underestimate*.

We can describe the complexity more generally using the number of initial axioms present within the production rule and the number of outer iterations (i.e. n).

$$O\left(\#initialAxioms^{\#outerIterations}\right)$$

3.5 Parser

The main purpose of the parser (defined in the *Parser.cs* script) is to take the derived string as input and read the symbols within it. Based on the detected symbols it will delegate the operations to be performed by the generator (*LSystem-Generator.cs*).

Table 2: Table of symbols and the operations that are performed on them.

Symbol	Operation
X	Do nothing (the axiom is not useful for the parser).
F	Generate a branch object.
R	Generate a root object.
L	Generate a leaf object.
$*$	Rotation by a random angle of one of the following verses: backward, left, right, and forward (also randomly chosen).
[	Push the transforms data. Saves the information about the position, length, and width of branches and roots.
]	Pop the transforms data. Retrieve the information about the position, length, and width of branches and roots.

As with the *Derivator.cs* class, the *Parser.cs* class is used by the generator as an independent logical unit (always according to the *SRP* [5]). It is used within the generator code by passing a reference to the generator itself and the derived string (both through setters).

3.6 L-System Generator

The *LSystemGenerator.cs* is the main class that deals with the management of the system's states (e.g., position of branches and roots), and its graphical representation.

The script itself is assigned (together with *Derivator.cs* and *Parser.cs*) to the *Origin* game object. This object will be moved around in the 3D world in order to generate the graphical components of the tree (*branches*, *roots* and *leaves*), all the transform operations will occur on this object. The initial position of the *Origin* object in the 3D space is set as $\langle 0, 0, 10\rangle$.

The rotation of a random angle of the Origin object in one of 4 directions (also random) is done based on a **seed** made explicit in the inspector (fig. 10). Although a random choice of direction and angle of branches (or roots) occurs, this L- system **is not stochastic,** since such a system would have to place a *probability distribution* on the choice of the rule to be used (at each iteration). Instead, in this implementation, the pseudo-randomness is contained within the operation corresponding to the symbol '$*$'.

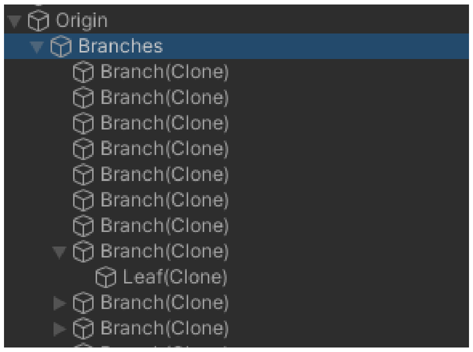
N.B.: While using the "free edit" mode, if a symbol not covered by the grammar is inserted, an *exception* will be thrown.

3.6.1 Data structures

There are several data structures for system management and implementation; below is a list with a brief description of their purposes.

- A **dictionary** that encapsulates the basic patterns (production rules).
- Different **variables** which are the main parameters of the tree generation. Since they are defined as serialized fields, they allow the user to manage the tree generation directly from the inspector panel (the user can change the variable values while the application is running).
- A **struct** (called *BufferedData*) used to perform *buffering* of the variables describing the current tree generation. This structure makes it possible to check whether the user has made changes to the current variables (via the inspector panel) by comparing the current values of the variables with those that were buffered at the end of the previous generation (this method provides a way to know that a new tree needs to be generated).
- A **stack** to handle the different states branches and roots can assume during generation. Each branch (or root) has its own transform, lengths, and thicknesses that vary as the tree grows.
- Two **dictionaries** containing the rules for the tree and root.
- A **Derivator**.
- A **Parser**.
- Two **lists** for collecting branch and root objects, respectively.

- Several general-purpose *variables* and *constants* needed for implementation are not shown because they are irrelevant to the design illustration (e.g., two game objects enclosing all branch and root objects as children, tree growth limits, ...).
  - It is worth highlighting the **two** GameObject variables in particular, as they will be the parents of all objects belonging to the branch and root lists (respectively), behaving like "**containers**" (fig. 5).



```
▼ Origin
  ▼ Branches
      Branch(Clone)
      Branch(Clone)
      Branch(Clone)
      Branch(Clone)
      Branch(Clone)
      Branch(Clone)
      Branch(Clone)
    ▼ Branch(Clone)
        Leaf(Clone)
    ▶ Branch(Clone)
    ▶ Branch(Clone)
```

Figure 5: The node "container" father of all branches, and the terminal branches in turn are the fathers of the leaves.

3.6.2 Awake()

The first function that will be loaded will be the *Awake()* function, it will just allocate the necessary data structures like the derivator, parser, and "container" nodes. After that the *Start()* function will be called, which will sequentially :

- buffer the data of the actual generated tree (null data if the tree has not yet been generated).
- call the method for generating the upper branches.
- reset the position to the origin.
- call the method for generating the roots.
- resets the position again to the origin.
- assigns the game objects for branches and roots to the container nodes.

3.6.3 Update()

Inside the *Update()* method two important methods are implemented, they are the *tree regeneration* (when a change is recognized) and the *free real-time editing* of the tree production rule.

The tree regeneration is done by checking for differences between the buffered data in the generated tree with the values of the current inspector variables. If any differences are detected, the following operations will take place :

- Destruction of the old tree.
- Cleaning data structures containing branch and root nodes.
- Reset of *Origin* position to $\langle 0, 0, 10 \rangle$.
- Call of the *Start()* method (generation of a new complete tree, with branches and roots).

The second feature is implemented using a boolean serialized field as a flag (fig. 6), when this is enabled it will be possible to modify the production rule via keyboard, otherwise any change will be reset to the previously selected default model.

Free Editing ✓

Figure 6: The boolean flag (from the inspector panel from 10) that enables the editing of the tree rule (Inspector view of the *Origin* object).

3.6.4 Branches and roots generation

The generation of the tree is done by calling two procedures (*GenerateBranches()* and *GenerateRoots()*) for the generation of all branches and all roots. These two procedures are very similar, the first will be described in a more detailed way, while for the second, only the major differences will be highlighted.

The generation procedures call the derivative by setting the production rules (rules of branches are different from those of roots). Once set, this is used to derive the string, after the set number of iterations (modifiable by the inspector). The obtained string will be provided as input to the parser, which will start the parsing process for drawing the graphics. The generation methods end here, the difference with the method that generates roots is that the number of iterations is limited to a maximum of four (*MAX_ROOTS_ITERATIONS*).

More in detail, one of the most important functions offered by *LSystemGenerator.cs* class (but invoked by the parser) is the one to generate a single branch (called *GenerateBranch()*, for the root it is called *GenerateRoot()*). The branch itself (and also the root object) is a *LineRenderer* (fig. 7) object, two 3D coordinates are required to set the start and end points of the branch. It is also possible to set the thickness at specific points on the object.

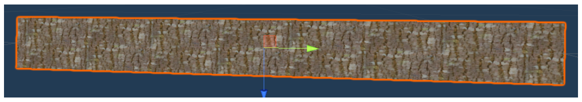


Figure 7: *LineRenderer* object that represents a branch.

When this method is called, what happens is that the *Origin* object is translated upwards by an amount equal to the length of the branch (temporarily saving the previous position in a variable). Then a branch object is instantiated, and the two positions of the *LineRenderer* are set with the temporary saved (start point) and translated position (endpoint).

The thickness is managed with a method called *SetSegmentWidth()* (which is also used within root generation). At the end of the procedure, the newly generated branch object is added to the list of branches.

3.7 Push&Pop

The *Push()* and *Pop()* operations allow you to save and retrieve the states of the branches (and roots) by means of a LIFO (*Last In First Out*) paradigm. A segment **state** (since the procedure and the stack is the same for both branches and roots) consists of its transform, length, and thickness (or *width*, since the LineRenderer is actually a 2D). These features are enclosed in a special struct (called *State*), and the objects instantiated by this struct will be pushed or popped into the appropriate stack.

During a *Push()* operation the state data is saved using the following methodology :

- The **transform** is the position and rotation of the segment.
- The **length** of the segment is *decreased* by a constant then saved. This ensures that when the next segment will be generated it will *peek* from the stack a shorter length.
- The **width** (or *thickness*) of the segment will tend to decrease naturally as the tree grows, so the starting point of the segment will be equal to the endpoint of the previous segment (*state*) and the endpoint will be equal to the starting point decreased by a constant.
  Decreasing the thickness is done in a predefined manner, by adding the decremented thickness to the inside of the stack and reusing the value at the time of segment generation (via *Peek()*).

The *Pop()* procedure pops the top state in the stack, and the *Origin* transform is reset when the operation is performed, while the length and width are just lost (since they are not needed anymore). Using this method of recording states, the tree will be built evenly by thinning and shortening (in a conical manner) the branches or roots (fig. 8).

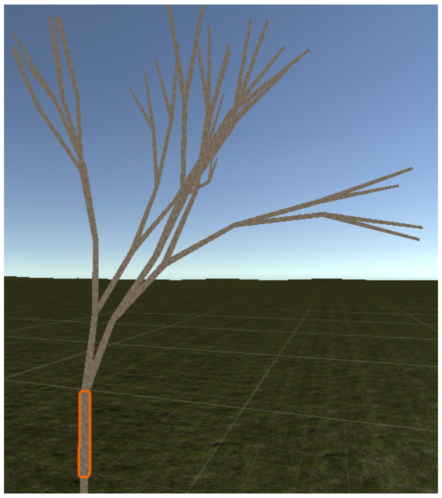


Figure 8: At the end of the tree, the branch thickness tends to decrease gradually and tends to sharpen (by a factor that depends on the number of iterations and the constants set).

3.7.1 Roots

The generation of roots is practically identical to that of branches, the only differences are that they do not have leaves, they develop in the opposite direction and the *production rule* (of the predefined model) is implemented differently.

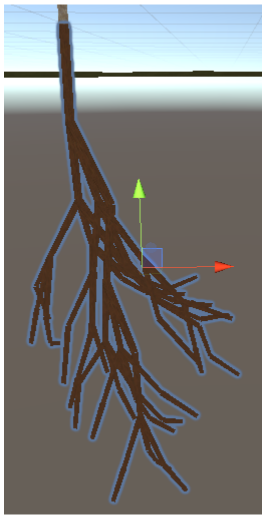
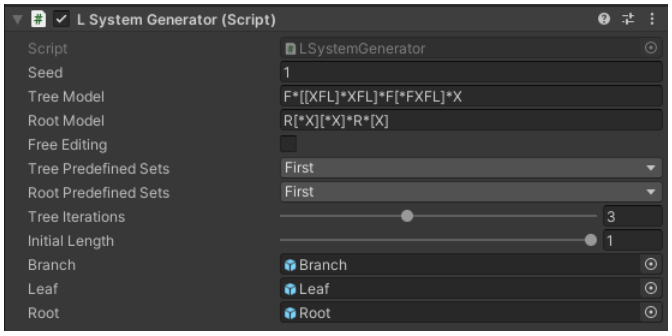


Figure 9: Roots seen from the scene window with the "*Roots*" parent node selected.

Furthermore, in the branch generation, there is a fixed base of two consecutive straight branches (inserted at the head through two *F* symbols), this is because usually trees grow a trunk before they start branching, while roots do not.

3.7.2 Inspector



L System Generator (Script)

Script	LSystemGenerator
Seed	1
Tree Model	F*[[XFL]*XFL]*F[*FXFL]*X
Root Model	R[*X][*X]*R*[X]
Free Editing	☐
Tree Predefined Sets	First
Root Predefined Sets	First
Tree Iterations	3
Initial Length	1
Branch	Branch
Leaf	Leaf
Root	Root

Figure 10: Inspector panel of *LSystemGenerator.cs*

Figure 10 shows the inspector panel for the *LSystemGenerator.cs* script, which is attached to the "*Origin*" object. In this panel, it is possible to :

- Tweak the random seed.
- Display (and tweak) the branch production rule.
- Display (and tweak) the root production rule.
- Enable/disable the free editing of the branches and roots production rule.
- Select a predefined production rule for branches.
- Select a predefined production rule for roots.
- Set the number of iterations to apply on the derivator (the range of iterations is $[1, 6]$).
- Set the initial length of branches and roots.
- Select the *GameObjects* to be attached for generation.

4 External resources

From the package manager (Unity asset store) :

- Bark Textures (PBR) - Volume One
- Terrain Textures Pack Free
- Hand Painted Seamless Wood Texture Vol - 6

As an external resource it was possible to find a two-dimensional image of the leaves (*Leaf.png*), the origin of the author is unknown.

5 Possible improvements

A list of possible improvements to the current project and possible expansions :

- Tree growth conditioned by the external environment. For example, what would happen if boulders were present under the roots or if an occlusion was present above the tree itself?
  - Handle occlusions.
  - Tree reactive to the seasons and weather.
  - Different development of the roots in the presence or absence of water and nutrients underground.
- Add new production rules for branches.
- Add new production rules for roots (create predefined sets).

6 Conclusion

L-systems are systems that have enabled the creation of procedural content, they have had a major impact on realism in graphical applications such as video games. This great influence is probably due to the great diversity and realism that they allow because at each execution of the application you can get a different result from a previous state. This characteristic allows the creation of geometric structures that seem more organic, this approach is highly preferable with respect to manually constructing assets, which are often limited in scope and variety due to time, budget, and memory constraints.

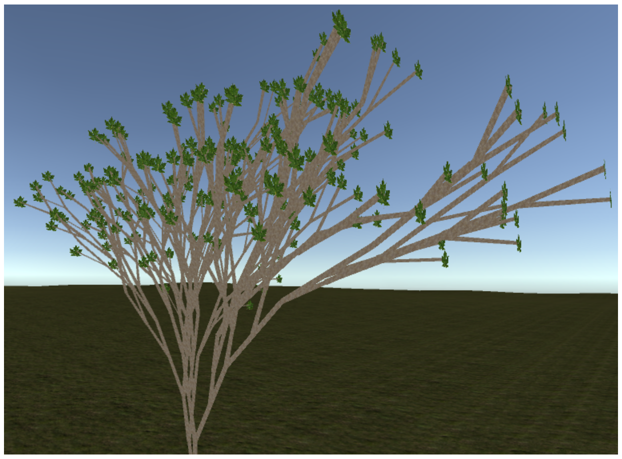


Figure 11: One possible end result.

The generation of trees is just one of the many applications of L-systems, other possible applications in the context of open-world video games include the generation of railway networks, cities, worlds, and galaxies.

A possible drawback in the use of L-systems could be given by the computational costs, as the time efficiency ends up being highly dependent on the size of the system. For example, the pseudo-random procedural generation of an entire railway network of a country might take a considerable amount of time on a moderately powerful consumer machine. Considering the developed application, it can be seen that changing a parameter initiates the complete regeneration of the tree, which approximately performs a number of operations equal to the number of symbols in the derived string (not counting axioms). However, this problem can be circumvented when it comes to generating large worlds or systems since they usually have to be generated only once, so introducing a loading phase at the application startup solves the problem without impacting the game experience.

References

[1] Community. L-system, https://en.wikipedia.org/wiki/L-system.

[2] Anders Hejlsberg. C# programming language documentation, https://docs.microsoft.com/it-it/dotnet/csharp/.

[3] Aristid Lindenmayer. *LaTeX: Mathematical Models for Cellular Interactions in Development. I and II.* Journal of Theoretical Biology, Department of Biology, Queens College, 23 August 1967. Available at https://doi.org/10.1016/0022-5193(68)90079-9.

[4] Junio Hamano Linus Torvalds. Git, https://git-scm.com/.

[5] R.C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Robert C. Martin Series. Pearson Education, 2017.

[6] Microsoft. Github, https://github.com/.

[7] Microsoft. Visual studio, https://visualstudio.microsoft.com/it/.

[8] Manuel Pagliuca. L-system project repository , https://github.com/manuelpagliuca/l-system.

[9] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. The algorithmic beauty of plants. *Endeavour*, 1997.

[10] Unity. Unity, https://unity.com/.