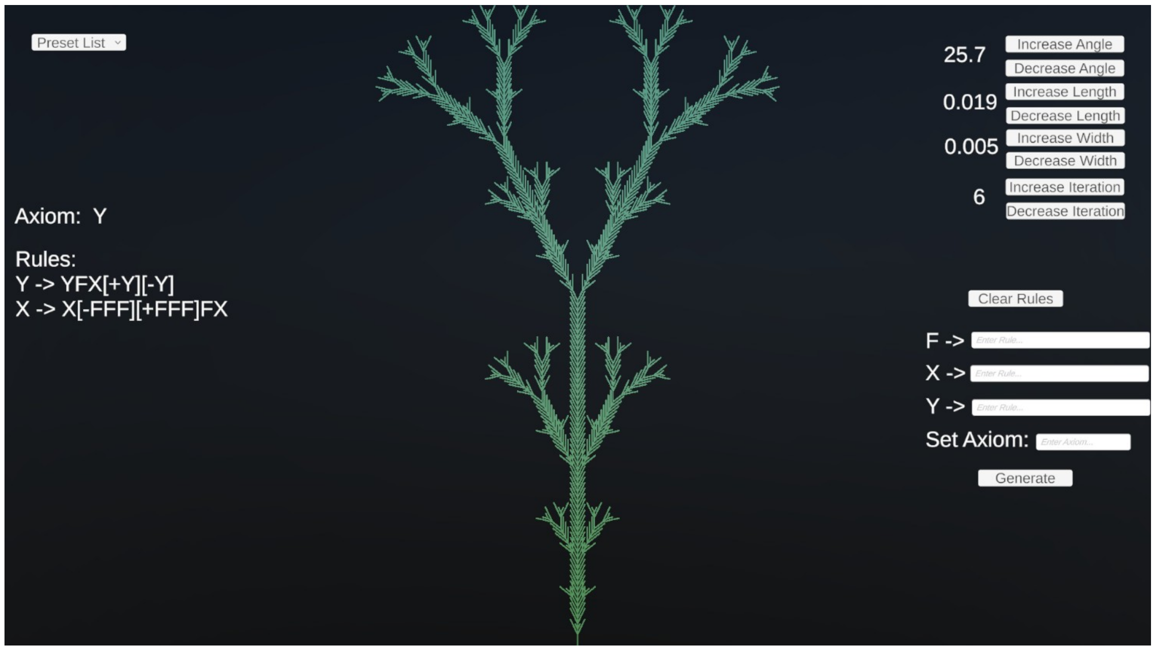
Lindenmayer-System Coursework

Henry Albert

Mathematics for Games and V/AR

November 28, 2025



Axiom: Y

Rules:
Y -> YFX[+Y][-Y]
X -> X[-FFF][+FFF]FX

25.7
0.019
0.005
6

Video: https://youtu.be/edyPuEwXF8Q

Code: https://github.com/HenryAlbert55/L-Systems-Coursework

Abstract

This project explores the implementation of a Lindenmayer System (L-system) engine within Unity using C#, with the goal of generating visually coherent, plant-like structures through formal grammar rewriting. L-systems provide a compact rule-based framework capable of producing complex forms from simple symbolic expressions, making them well-suited for procedural modelling. The engine developed for this project expands user-defined axioms and production rules into increasingly detailed strings, which are then interpreted in real time to construct structures from line-rendered segments and instanced prefabs. A stack-based approach is employed to manage branching behaviour, allowing the system to create intricate multi-level forms that resemble biological growth patterns.

The system was tested on a range of grammars, from simple fractals to more elaborate botanical models, and consistently produced stable and coherent structures. Performance remained strong at moderate iteration depths, with predictable declines as the exponential nature of L-systems increased object counts. The resulting models are fully interactive Unity objects capable of participating in lighting, physics, and environmental effects, demonstrating the engine's value for real-time applications. Through this implementation, the work highlights the effectiveness of L-systems as a tool for procedural content generation and showcases the visualization of formal grammars in an interactive environment.

Objectives

L-systems offer a compact yet powerful mathematical framework for defining complex structures through iterative rewriting. Originally created as a model for plant development, L-systems have since become an important tool in computer graphics and procedural content generation. Their ability to create intricate shapes from simple rules makes them ideal for environments where emergent complexity is desired (Perdigão).

The goal of this project was to create a fully functional L-system engine inside Unity, written in C#, capable of generating structures that behave like natural objects. Opting for classic line-drawing methods, the project focused on translating symbols into Unity operations. Each character in the expanded L-system string corresponds to a transformation and the creation of a new line within the scene.

The primary aim was to construct a rewriting system capable of expanding an axiom into longer and more complex sequences with each iteration. This rewriting mechanism needed to be deterministic, stable, and capable of supporting arbitrary rule sets. It also needed to be simple for users to configure through a GUI, allowing parameters such as angles, segment lengths, width, and iteration depth to be adjusted without altering the code; these options are seen in Fig1.

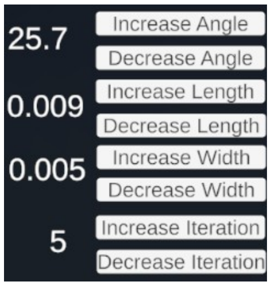


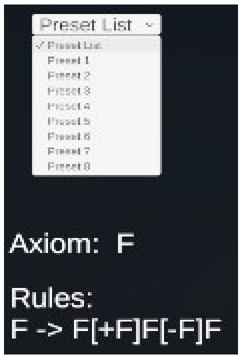
Figure 1: Parameter UI

Another major objective was the development of a rendering method that would translate L-system symbols into actions that manipulate GameObjects. Drawing lines, the renderer would instantiate a prefab and apply rotations plus positional offsets to simulate branching and structure growth. The resulting output would form a coherent tree-like arrangement where each segment is a Unity object possessing its own transform.

A further goal was ensuring the system's modularity. Grammar expansion, rendering, and state management needed to exist as separate components to maintain clarity and scalability. This separation would allow future additions, such as 3D rendering, texturing, leaf shapes, context sensitivity, and stochastic systems, to be integrated without disrupting the core structure (Lindenmayer and Prusinkiewicz).

Implemented Features

The system begins with a rewriting engine written in C#, designed to accept an initial axiom and a set of production rules. The user can see the current axiom and rules, Fig2, and altar them as they see fit, Fig3. The engine iteratively processes the axiom, replacing characters according to the rules and constructing longer sequences with each pass. These sequences may become extremely large due to exponential growth.



Axiom: F

Rules:
F -> F[+F]F[-F]F

Figure 2: Preset UI

The generator produces a final expanded string after processing the specified number of iterations, which is then passed to the rendering component.

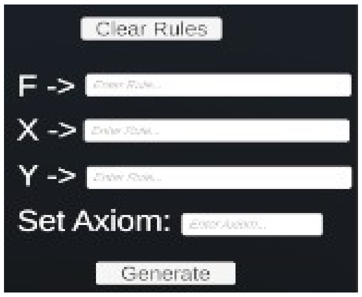


Figure 3: Axiom/Rules UI

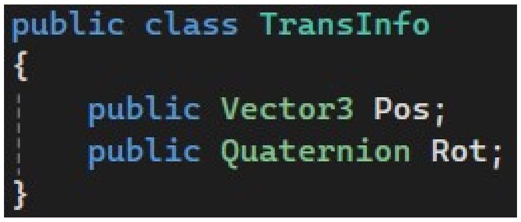
Rendering takes place inside a Unity scene using the Transform component. Each character in the expanded L-system string is interpreted as a command. Characters that represent structural growth cause the system to instantiate prefabs of short branch segments, at the current position and orientation. After placement, the system advances the transform so that subsequent segments extend naturally from the previous ones. Rotational symbols alter the transform's orientation, enabling the construction of curves and corners.

Branching is a critical aspect of natural plant modelling. To handle this, the system uses a stack to store transform states. When a symbol representing the start of a branch is encountered, the current position and rotation of the structure are saved. Later, when the corresponding end-branch symbol is reached, the stored state is restored, allowing the renderer to continue building from a previous point. This ability to temporarily diverge from the main structure and return later is essential for generating realistic trees and other natural forms.

All settings including angles, segment length, width, the axiom, rules, and number of iterations can be modified directly from a GUI and keyboard hotkeys. This design allows the user to experiment with different parameters without recompiling the code.

Data Classes and Structures

The project relies on both internally defined structures and built in c# structures to manage the rewriting and rendering processes. The rule set is stored in a dictionary that allows quick retrieval of successor sequences for each character, ensuring that grammar expansion remains efficient even at higher iteration counts. The rewriting component must traverse the growing L-system string multiple times, so performance considerations shape how rules and strings are represented.



```
public class TransInfo
{
    public Vector3 Pos;
    public Quaternion Rot;
}
```

Figure 4: Transform Information Class

The renderer depends upon the Unity's Transform system, which provides a natural representation of spatial relationships. A class containing the transform information is used in a stack to save this data, as seen in Fig4. The stack is used to store branching states and is a crucial data structure. Each entry in the stack contains the current position and orientation of the ongoing structure. When rendering reaches a point where a branch should begin, this state is pushed to the stack. Later, when the branch ends, the renderer pops the stored state and resumes building from that position. Every instantiated branch segment begins at the end of the previous one or the last transformation popped from the stack. This behaviour enables the construction of highly detailed, multi-branched structures without losing track of the main trunk or branch hierarchy.

Results

The system was tested with a variety of L-systems ranging from simple fractals to complex plant grammars. In all cases, the generator successfully expanded the grammar and produced coherent structural output in Unity. Classical fractals such as the dragon curve were correctly formed, demonstrating that rotational transformations were applied with precision. Plant-like grammars produced branching trees that were visually comparable to examples from the reference website provided in the lecture slides (OnlineTools). Below are the comparisons in Fig5 and Fig6, left are mine and right are the references.

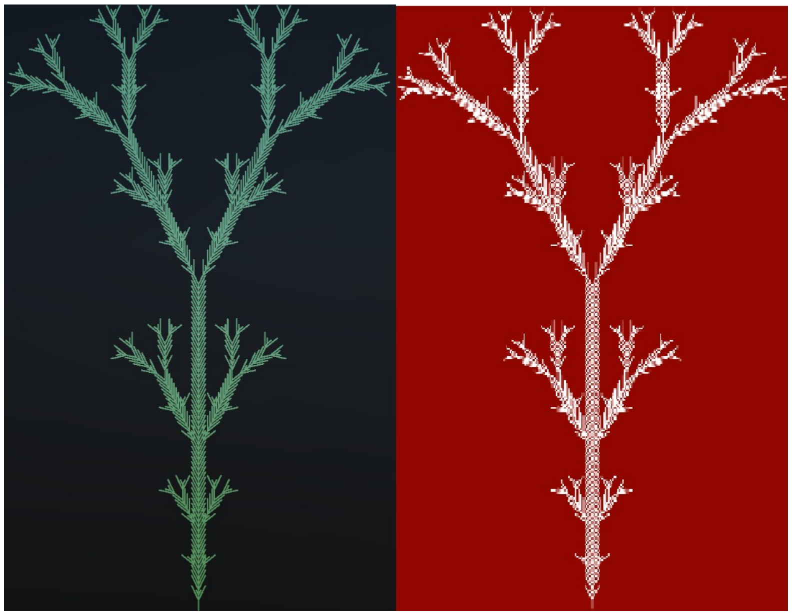


Figure 5: Bush L-System Mine Vs Reference

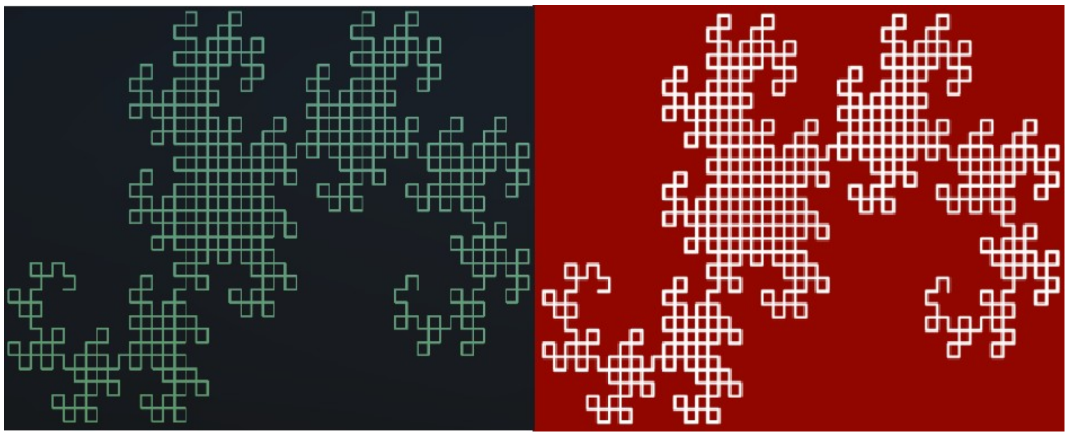


Figure 6: Dragon Curve L-System Mine Vs Reference

These two, plus the six in the project specifications, are available as preset options for the user as seen in Fig2; they were found on *L-System User Notes* (Bourke).

Performance testing revealed that the system handles moderate iteration depths well. For small numbers of iterations, rendering is nearly instantaneous. As the size of the expanded string increases, performance declines predictably due to the large number of instantiated objects. This behaviour is consistent with the known exponential nature of L-systems. For very deep grammars, additional optimization techniques such as mesh combining or object batching may be desirable (Unity Technologies), but these were beyond the immediate scope of this project.

One of the major advantages of using Unity is that the resulting structures are fully interactive. They can cast shadows, receive lighting, participate in physics simulations, or respond to environmental effects such as wind zones. The generated objects are not static images but fully functional scene elements which can be manipulated, animated, or even integrated into a larger game or simulation.

Conclusion

This project demonstrates the effectiveness of a platform for procedural modelling based on L-systems. By implementing a deterministic rewriting system capable of expanding compact symbolic rules into long structural sequences, the project successfully reproduced a variety of plant-like forms and fractal patterns. The rendering component, grounded in Unity's Transform architecture, proved effective at translating the expanded grammar into visible structures, allowing each symbol to drive concrete spatial operations. The use of a stack for managing branching states enabled the generation of highly articulated, multi-level forms that align closely with the classic behaviours described in L-system theory.

At the same time, the project highlighted the inherent computational challenges of L-systems. Because structure size grows exponentially with each iteration, performance naturally decreases as more objects are created. Although this behaviour is expected, it suggests several clear directions for improvement, including the introduction of mesh combining, object batching, or more advanced instancing techniques.

Although prototype-level, the implementation achieves all its objectives and provides a robust foundation for future expansion. Potential extensions include stochastic rules, context-sensitive grammars, more advanced rotation systems, 3D rendering, texturing, and leaf shapes. The success of this project highlights the use of L-systems not only as a mathematical abstraction but also as a practical tool for real-time content generation within modern game engines.

Bibliography

Online Tools (2025) *L-system Generator*. Retrieved from onlinetools.com:

https://onlinetools.com/math/l-system-generator

Bourke, Paul (1991, July) *L-System User Notes*. Retrieved from paulbourke.net:

http://paulbourke.net/fractals/lsys/

Perdigão, Gonçalo (2024, Aug) *The Power of Lindenmayer Systems: A Journey from Code to Natural Patterns*. Retrieved from buildingcreativemachines.substack.com:

https://buildingcreativemachines.substack.com/p/the-power-of-lindenmayer-syste
ms

Unity Technologies (2025) *Optimizing draw calls* (Unity Manual). Retrieved from docs.unity3d.com:

https://docs.unity3d.com/2022.3/Documentation/Manual/optimizing-draw-calls.ht
ml

Lindenmayer, Arsitid and Prusinkiewicz, Przemyslaw (2004) *The Algorithmic Beauty of Plants*. Retrieved from algorithmicbotany.org:

https://algorithmicbotany.org/papers/abop/abop.lowquality.pdf