

Assignment #4 – Procedural-it!

L-Systems and Basic Procedural Modeling

CS334/ECE30834

Out: October 21, 2026

Due: November 6, 2026

Summary

The object of the assignment is to implement a mini procedural tree modeler using **L-systems**. The program reads a text file containing a rotation angle A in degrees, an iteration setting N , an axiom S (the starting string), and a set of production rules R . It repeatedly rewrites the string and converts the result into 2D line segments using **turtle drawing**: a virtual pen that has a position and a heading.

The supplied viewer uses **GLFW**. The main programming tasks are in `src/lssystem.cpp`; the optional stochastic-rule extension also requires changes to the rule storage in `src/lssystem.hpp`. Each nested bracketed branch uses a step length 0.75 times that of its parent, as specified in Part 4.

Part	Implementation	Weight
2	Input parsing (TODO1)	30%
3	Rule application (TODO2)	30%
4	Geometry generation, including branch scaling (TODO3)	30%
5	Random angle jitter (TODO4)	10%
5	Stochastic rules (TODO5, optional)	Bonus 15%

1. Getting Started

Download and extract the template from the course website. On Windows, open `base_glfw.sln` in **Visual Studio**, select an `x64` configuration, and build the project. On Linux, use the supplied Makefile and the build instructions in `README.txt`. Window creation, OpenGL setup, and the viewer controls are provided; do not replace them as part of the assignment.

The default input is `models/tree1.txt`. The title bar identifies the selected file and displayed iteration. The common keyboard controls are:

Control	Action
Up / Down	Select the previous / next input file.
Left / Right	Display the previous / next iteration; Right can generate another iteration.
Space or R	Reload the current input file.
Escape	Exit the program.
H or F1	Print the viewer help.

The Windows viewer also has a right-click menu. All platforms can use the keyboard controls. Reloading a text input does *not* recompile C++; rebuild after modifying a C++ source file.

2. Input Parsing (30%)

Implement `LSystem::parse()` in `lssystem.cpp` (TODO1). Read the input fields into the temporary variables identified in the source comments, then leave the supplied state-update and iteration-management code in place.

For example, `models/tree1.txt` contains:

```
25.7
6
f
f : f[+f]f[-f]f
```

Input	Meaning
First line	Rotation angle A in degrees; Part 5 extends this line with optional jitter.
Second line	Iteration setting N , interpreted by the supplied viewer as noted below.
Third line	Axiom S : the starting string.
Remaining lines	Production rules, one rule per line.

A rule has a single-character **predecessor**, a colon `:`, and a **successor** string. The successor may contain multiple characters but must remain on one input line. There may be more than one rule, but fewer than ten. Whitespace should be ignored. The provided preprocessing helpers remove comments and empty lines and trim the input lines.

For the deterministic required portion, store the rules in a `std::map<char, std::string>`: each predecessor maps to one successor. The stochastic-rule bonus in Part 5 allows several alternatives for the same predecessor and therefore needs a different storage arrangement.

Iteration convention in the supplied viewer. The axiom is displayed as iteration 0. For positive N , the viewer initially stores N states *including* the axiom. For $N = 6$, the viewer initially stores iterations 0 through 5: five rewriting steps after the axiom. The supplied framework manages this count; do not add an extra rewriting loop in your parser. The Right key can request the next iteration.

3. Rule Application (30%)

Implement `LSystem::applyRules()` in `lssystem.cpp` (TODO2). Given one input string, return the string produced by **one complete rewriting step**.

Read the input from left to right. For each character, look for its production rule. If a rule exists, append the entire successor to the output string. Otherwise, append the character itself. All replacements in this step are based on the *input* string; characters newly appended to the output are not rewritten until the next iteration.

In particular, a character with no production rule must survive the rewriting step. It may still be a drawing command, such as a bracket or a rotation symbol. Whether a character is ignored during geometry generation is a separate decision made in Part 4.

There may be no visible output until geometry generation is implemented. To check this part independently, print short intermediate strings before returning them and compare successive iterations.

4. Geometry Generation (30%)

Implement `LSystem::createGeometry()` in `lsystem.cpp` (TODO3). Interpret the expanded string from left to right and return a vector of 2D vertices. Every consecutive pair of vertices defines a separate line segment.

Symbol	Turtle action	Weight
<code>f</code> , <code>F</code> , <code>g</code> , <code>G</code>	Draw a line and advance by the current step length.	5%
<code>s</code> , <code>S</code>	Advance by the current step length without drawing.	5%
<code>+</code>	Rotate counterclockwise by A degrees.	5%
<code>-</code>	Rotate clockwise by A degrees.	5%
<code>[</code>	Save the complete state, then scale the new branch's step by 0.75.	5%
<code>]</code>	Restore the last saved state, including its step length.	5%

Any character not listed above is ignored *by the drawing interpreter*.

Line segments and movement

Start at the origin, facing the **positive y-direction**. This is a 2D position and heading, not a mesh to be loaded from a file. For a drawing command, output the current position and the new endpoint, then advance the current position. For a non-drawing move, update the position without outputting vertices.

For example, `ff` produces four vertices: the endpoint of the first segment and the start of the second segment have the same position. Choose a positive initial step length ℓ_0 ; the examples use $\ell_0 = 1$. The viewer automatically scales the completed geometry to fit the window.

Bracketed branches and step-length scaling

The complete drawing state consists of **position, heading, and current step length**. On `[`, save the parent state *before* shortening the step. The branch begins at the same position and heading, but its step length is multiplied by 0.75. On `]`, pop and restore the most recently saved state. Restoring a state does not draw a connecting line.

For bracket nesting depth d , the step length is

$$\ell_d = \ell_0 (0.75)^d.$$

Both drawing moves (`f`/`F`/`g`/`G`) and non-drawing moves (`s`/`S`) use this step. The factor changes lengths, not rotation angles. Do not shorten the parent trunk permanently, and do not apply an additional shortening merely because the rewriting iteration increased.

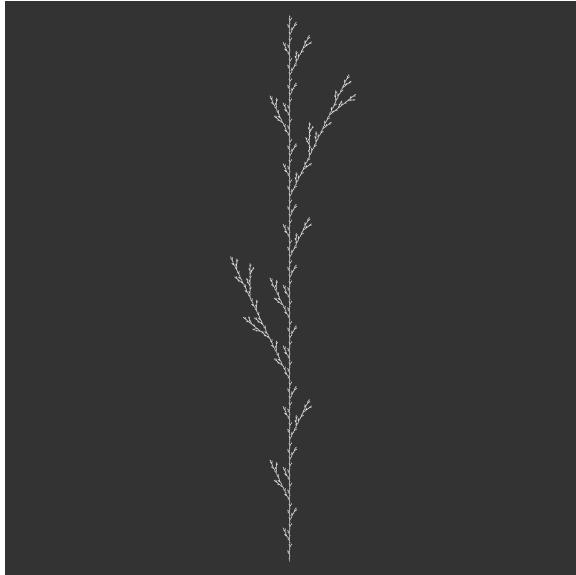
For $\ell_0 = 1$, the string `F[+F]F` gives:

Command	Location/state used	Step length
First <code>F</code>	Main trunk	1.00
<code>F</code> inside <code>[+F]</code>	Turned branch, after saving the parent state	0.75
Final <code>F</code>	Restored parent state	1.00

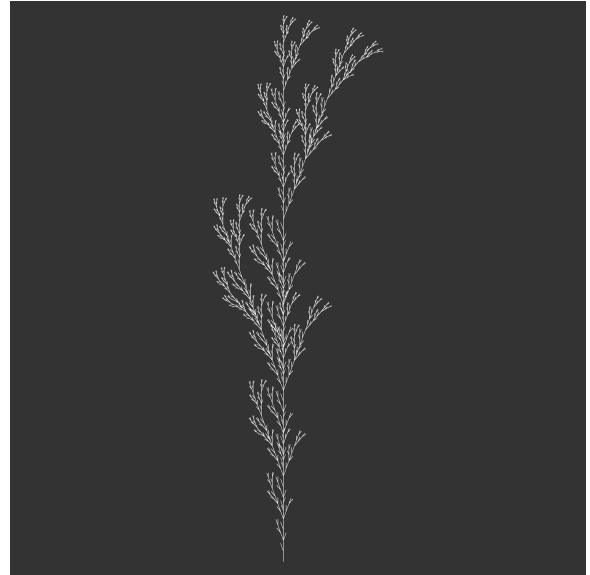
You may store the state explicitly or use transformation matrices on a stack. The representation is your choice; the state restored by `]` must include the branch scale. This requirement is part of the existing 30% for geometry generation, not a separate bonus.

4. Geometry Generation: Example Outputs

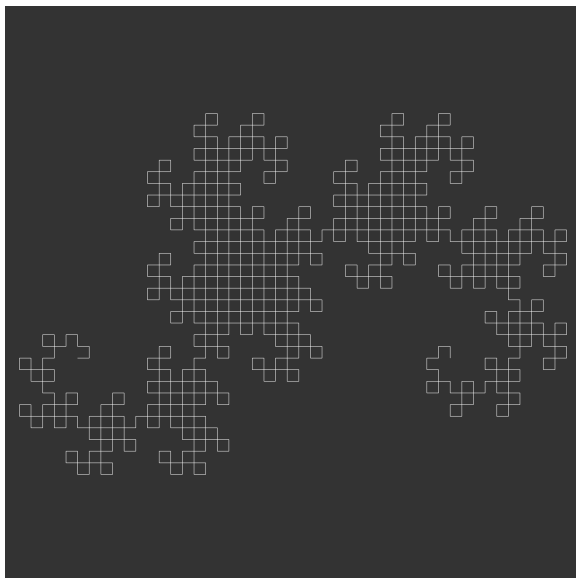
The following line-geometry examples use the supplied inputs, no angle jitter, and branch scale 0.75. Each view is fitted to its window. Match the **displayed iteration index**; small rasterization differences do not change the L-system structure.



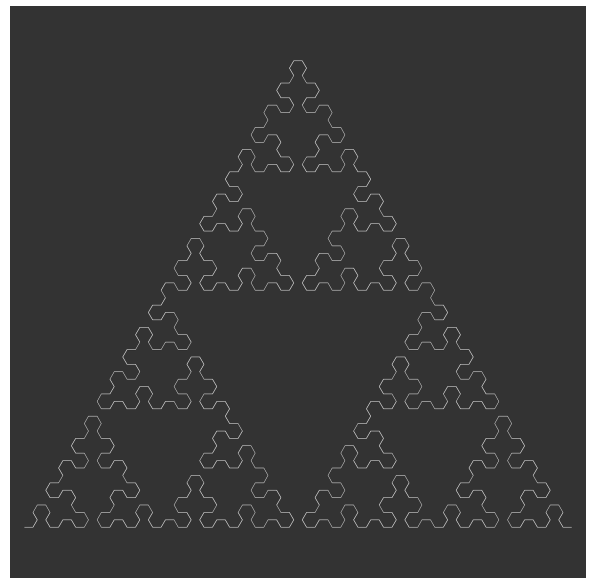
(a) `tree1.txt`
Displayed iteration 5.



(b) `tree2.txt`
Displayed iteration 5.



(c) `dragon.txt`
Displayed iteration 10.



(d) `sierpinski.txt`
Displayed iteration 6.

Figure 1: Deterministic geometry examples with branch scaling.

The two tree inputs are useful for checking saved branch states and relative branch lengths. The displayed dragon and Sierpinski inputs contain no bracketed branches, so their geometry does not receive the branch-length scaling.

For more background on the drawing interpretation, see Chapter 1 of *The Algorithmic Beauty of Plants*, linked on the course website.

5. Stochastic Generation (10% + Bonus 15%)

Random angle jitter (10%; TODO4)

Add random variation to individual turns. Read the optional jitter magnitude J in `parse()` and apply it in `createGeometry()`. The supplied `tree1_rand_angle.txt` uses:

```
25.7<10.0>
6
f
f : f[+f]f[-f]f
```

If no jitter value is present, use $J = 0$. On each turn, sample an offset uniformly in $[-J, +J]$ and add it to A . Apply the resulting turn counterclockwise for $+$ or clockwise for $-$. Jitter affects angles, not lengths; the branch-scaling rule still applies.

Stochastic rules (Bonus 15%; TODO5)

Extend parsing and rule application to support several weighted successors for one predecessor. For example, `tree_rand.txt`:

```
25.7
6
f
f 0.33 : f[+f]f[-f]f
f 0.33 : f[+f]f
f 0.34 : f[-f]f
```

For each rewritten character, choose an alternative with probability equal to its weight divided by the total weight for that predecessor:

$$\Pr(R_i \mid c) = \frac{w_i}{\sum_{j: \text{pred}(R_j)=c} w_j}.$$

Weights need not sum to 1. Maintain compatibility with deterministic inputs: the weight is optional and defaults to 1 when omitted. Since `std::map` stores only one value per key, update the rule representation in `lsystem.hpp` as well as parsing and rule selection. Reloading a stochastic input may produce a different shape; no particular random realization is prescribed.

6–7. Implementation Notes and Test Files

Functions are marked TODO; modify other code as needed. Test with the supplied inputs; additional inputs may be used for grading. Preserve the input syntax and existing example files. Branch scaling needs no extra input file.

Turn-in

Submit your complete project as a ZIP on **Brightspace**: project/build files, source code, shaders, model inputs, and a precompiled executable.

For grading, the program will be compiled on Linux and run from the terminal, with Visual Studio as a fallback. It will be run without command-line arguments and the code will be inspected. Avoid unnecessary platform-specific code in your implementation; the supplied Windows menu adapter is part of the viewer framework. If the program does not compile, zero points will be given. Please ask questions on Piazza. Start early and good luck!