CS334/ECE30834: Assignment #4 - Procedural-it!
L-Systems and Basic Procedural Modeling

Out: March 14, 2025
Back/Due: April 4, 2025

Summary:
The objective of this assignment is to implement a mini procedural tree modeler using L-Systems. Your assignment is to read in a simple text file containing the rotation angle *A* (in degrees), the number of iterations *N*, the axiom *S* (i.e., starting symbol), and rule(s) *R*. Then, create a string representation of a tree T by successively applying rules to the axiom, for *N* iterations. Finally, convert the string representation into a set of 2D line segments following "turtle drawing" logic to produce the geometry of the tree.

Specifics:

1. Start with the templates from the course website. The templates support Windows and Linux environments. Compile and run the templates. Then, make your changes below.
2. **Input Parsing (30%)**
   The program will read files as in the following example:
   `tree1.txt`

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

   The first number is the rotation angle *A*, the second number is the number of iterations *N*, the third line is the axiom *S*, and all of the following lines are the set of rules *R*. This example only has one rule, but there can be more than one (but less than 10). Each rule must have a : symbol. The character before the : is the *predecessor*, and the string after the : is the *successor*. The predecessor must be a single character, but the successor can be of any length but on a single line of text. Whitespace should be ignored.

   Implement the `LSystem::parse()` function in `lsystem.cpp`, and store the relevant fields according to the comments in that function. The rules are stored in a `std::map<char, std::string>`, which is a C++ associative container that maps characters, the *predecessors*, to strings, the *successors*. Refer to the C++ standard documentation to see how to use it: https://www.cplusplus.com/reference/map/map/

3. **Rule Application (30%)**
   Starting from the axiom, rules are applied to each character in the string to produce a new string. Then on the next iteration, the rules are applied to the resulting string from the last iteration, producing another string. You will implement

applying these rules, given an input string, and returning a string with the rules applied. For each character in the input string, you must check if there is a rule that would replace it. If there is, append the replacement string (the successor) to the output string. If there is no rule to replace the current character, append the character itself to the output string. Write your implementation in the `LSystem::applyRules()` method in `lsystem.cpp`. There will not be any visual output until the next step is completed, so to verify the correct behavior, you can print the output string before returning it.

4. **Geometry Generation (30%)**

   Given a string with rules already applied, you will generate a set of 2D line segments to represent the geometry of the L-System using "turtle drawing" logic. Each character of the string should be interpreted as such:

`f,F,g,G`	Draw a line segment and advance forward (5%)
`s,S`	Advance forward without drawing a line segment (5%)
`+`	Rotate *A* degrees counterclockwise (5%)
`-`	Rotate *A* degrees clockwise (5%)
`[`	Push the current draw state (5%)
`]`	Pop the last-pushed draw state (5%)

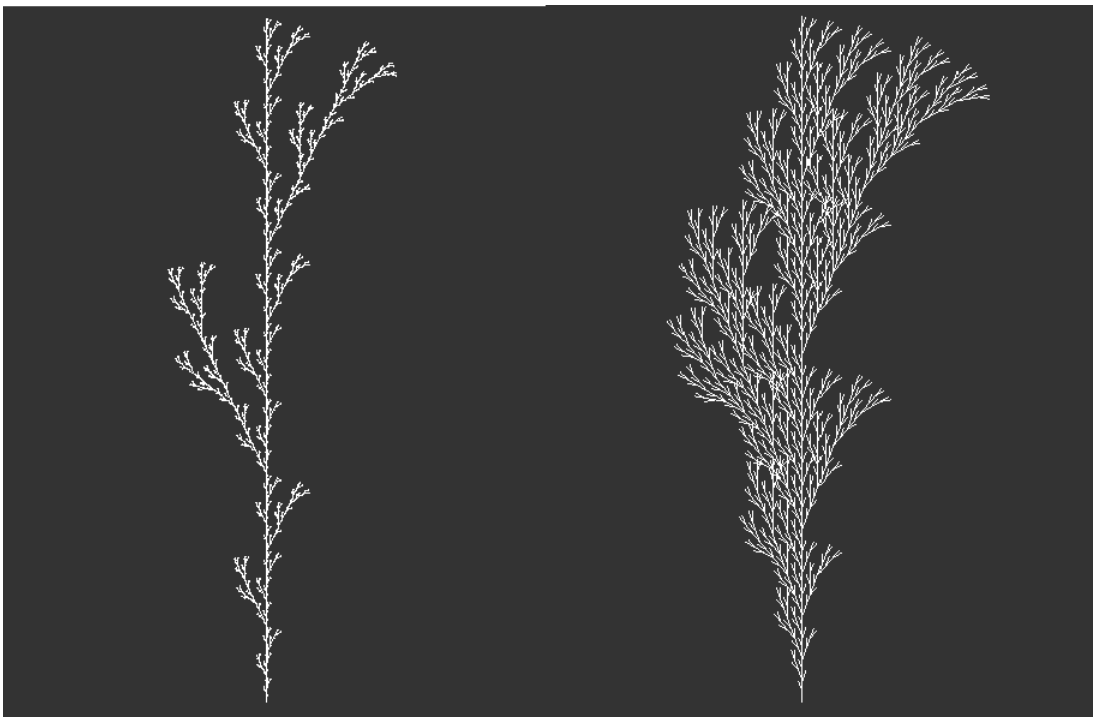
   Any character not listed above should be ignored.

   Line Segments. OpenGL expects a list of line segments, each with a start point and an end point. In other words, two connected line segments, as in the string `ff`, will output 4 vertices, with the 2nd vertex (i.e. end point of first line segment) having the same position as the 3rd vertex (i.e. start point of second line segment). You may choose any length for your line segments – the output will be automatically scaled to the window size.

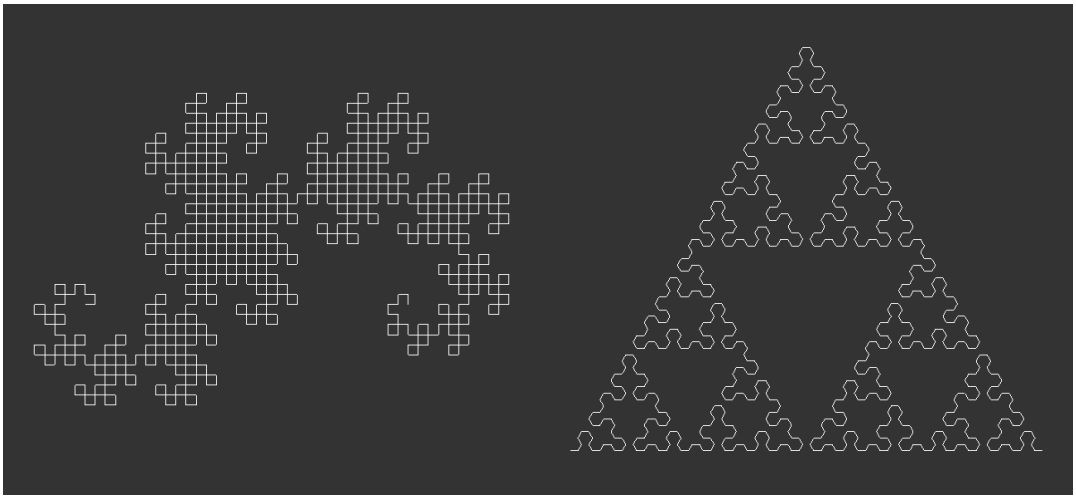
   Movement. When a line segment is drawn, the starting location of the line segment should be at the "current position". Then, the current position is advanced to the end of the segment that was just drawn. Thus, the next segment to be drawn should start where the last segment ended. Additionally, the current rotation determines the direction of the line segment. The initial rotation should be in the +A direction.

   Bracketed contexts. When a `[` is encountered, the current draw state should be pushed onto a local stack. The draw state consists of the position and orientation (rotation) of the next segment to be drawn. Later, when a `]` is encountered, the last draw state that was saved should be removed from the top of the stack, replacing the current draw state. The implementation details are up to you. (Hint: use matrix transformations).

   Write your implementation in the `LSystem::createGeometry()` method in `lsystem.cpp`. Refer to *The Algorithmic Beauty of Plants*, linked on the course website, for more information on the above (most of chapter 1). Some example outputs below:



`tree1.txt` `tree2.txt`



`dragon.txt` `sierpinski.txt`

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

   Random Angle Jitter (10%)

   Currently, every + and - command rotates the direction by the exact same angle A. As an enhancement, introduce small random variation around A, so each turn is slightly different. This can create more “natural” or “organic” tree shapes without major design changes.

   Specifically, the angleJitter J is an optional number enclosed in <> at the end of the angle value, for example:

   `tree1_rand_angle.txt`

```
25.7<6.0>
6
f
f : f[+f]f[-f]f
```

If no jitter value is provided (For example, `tree1.txt`), J is set to 0.0.

Whenever you rotate, randomly offset A by a value in [-J, +J].

Stochastic Rule (Bonus 15%)

In the above implementation, there is only one rule for any given predecessor. For extra credit, extend the parser and rule application to allow for multiple rules with the same predecessor. The extended syntax is below:

`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
```

In the rule definition, the predecessor is followed by a weight. The rule used to replace a predecessor with multiple rules is chosen randomly with a probability equal to the weight of the rule divided by the sum of all the weights of rules with that predecessor. The weights do not necessarily sum to 1. Your implementation should be backwards compatible with the non-extended syntax – that is, the weight of a rule is optional. Assume a weight of 1 if there is no weight specified. Note that you will need to make changes in several other places in order to make this work. Specifically, you will need to change how rules are stored in the class, since a `std::map` only allows a single value per key.

6. The functions/methods requiring your implementation will be marked TODO; however, depending on your particular implementation there might be other places for you to change code. You are expected to add/modify the code as necessary to ensure the application runs smoothly.
7. **Files**: Several example files are included in the template directory. You can test your solution against these example files. We have reserved others for our testing as well.

Turn-in:

To give in the assignment, please use Brightspace. Give in a zip file with your complete project (project files, source code, and precompiled executable). The assignment is due BEFORE class on the due date. It is your responsibility to make sure the assignment is delivered/dated before it is due. If you wish to receive confirmation of receipt, please ask by email in advance.

Don’t wait until the last moment to hand in the assignment!

For grading, the program will be compiled on Linux and run from the terminal (with Visual Studio as a fallback – please try to avoid platform-specific code (e.g., don't #include <windows.h>)), run without command line arguments, and the code will be inspected. If the program does not compile, zero points will be given. If you have more questions, please ask on Piazza!

Good luck!