Isabel Chien
6.837 Fall 2015
12/10/15

Modeling Plants with L-Systems

Motivation

L-systems are a straightforward and effective way to model recursive structures, namely plants. As such, they are beneficial in scientific research, especially with regards to biology and botany. Not only can they be used to simply model plants, they can also be used to model plant growth, as well as recognizing plants. Plants modeled from L-systems, including stochastic L-systems, can be used to classify plantlike structures, which would have clear benefits to biological and botanical research[1]. There are not only scientific applications to modeling plants, however—realistically rendered plants can be used in video games, animated movies, or any form of graphics that require a realistic landscape.

L-systems can be applied to more than plants, however. At their simplest, they can model fractals as well as other recursive structures. Indeed, in my research for this project, I came across a paper that used L-systems to generate electronic music[2]. There are many possible such extensions to the basic L-system structure, which is one of its many conveniences.

Background

L-systems, or Lindenmayer systems, were developed in 1968 by Aristid Lindenmayer. Lindenmayer was a theoretical biologist and botanist, who resided at Utrecht University in Hungary. He used L-systems to model plant growth, especially that of algae, and extended these systems to be able to model complex plants and plant development. With Przemyslaw Prusinkiewicz, Lindenmayer authored *The Algorithmic Beauty of Plants*, which describes the principles of L-systems, as well as the extension to fractals, then 2D plants, and eventually 3D plants. Stochastic and randomized systems are detailed as well[3]. I also read a paper by Prusinkiewicz that described L-systems and provided additional examples that I was able to implement[4]

---

[1] Samal, Ashok K.; Peterson, Brian; and Holliday, David J., "Recognizing Plants Using Stochastic L-Systems" (1994). CSE Conference and Workshop Papers. Paper 37.
[2] http://modularbrains.net/support/SteliosManousakis-Musical_L-systems.pdf
[3] Lindenmayer, Aristid; Prusinkiewicz, Przemyslaw. “The Algorithmic Beauty of Plants.” (1968).
[4] P. Prusinkiewicz: Graphical applications of L–systems. Proceedings of Graphics Interface '86 / Vision Interface '86, pp. 247–253.

Approach

To eventually reach the point where I would be able to generate realistic, randomized, 3D plants, I followed the progression covered in the Lindenmayer's book. First, I implemented the rewriting system, got it to function with fractals, then added the necessary elements for 2D plants and added stochastic elements. Finally, I extended this process to generate 3D plants.

I used Javascript to create the bulk of my logic, with a class LSystem(axiom, theta) that could solve for the final string to be used for drawing and also draw as using a provided rule. Drawing occurred via the library Three.js, which allowed me to generate lines for the 2D objects and cylinders for the 3D objects. I also created classes Tree() and Fractal() that contain examples of fractals and trees that can be rendered.

At its core, an L-system is a rewriting system and a kind of formal grammar. It consists of a collection of pre-determined symbols that can be used to make up strings, production rules that elucidate the function of each symbol, and an initial axiom, which is a string that defines how the grammar should begin. To complete the rewriting process, each character is examined and replaced with its rule, if there exists a rule for that character. Any number of iterations can occurs; typically, that is pre-specified by the user. Below is an example of a system as well as the results of two iterations.

Example: Koch Island
variables: F
axiom: F+F+F+F
rules: F -> F+F-F-FF+F+F-F
angle: 90 degrees

Iteration 1: F+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F

Iteration 2: F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F-F+F-F-FF+F+F-FF+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F-F+F-F-FF+F+F-FF+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F-F+F-F-FF+F+F-FF+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F-F+F-F-FF+F+F-FF+F-F-FF+F+F-F+F+F-F-FF+F+F-F+F+F-F-FF+F+F-F-F+F-F-FF+F+F-F

Once the final string is completed, it can be used to generate the target shape. Characters typically follow rules; in this example, + is to turn right, and – is to turn right by the specified angle, 90 degrees. F is to move forward by some specified length (in my implementation, this is a variable parameter).

Once I was satisfied with the results of my fractals—I also implemented the option to create a rainbow fractal, which is interesting as it shows the progression of the line drawing—I moved on to include the necessary characters for 2D plant generation. Only two new definitions for characters are needed to generate satisfactory 2D plants, the left bracket '[' and right bracket ']'. The left bracket indicates to push the current state, which is the current position and direction in the rendering process, and the right bracket serves to pop the current state, which essentially transports the current drawing position to a past position.

Extending this to create stochastic trees, which add an element of randomization to the tree grammars and therefore aide in generating realistic plants, was a matter of modifying the code that solved for the final system. Instead of having a character point to a string as a rule, stochastic objects involve having a character point to an array of strings as a rule. The code that solves for the final system simply recognizes if the rule is an array and retrieves a string from a certain index of that array with uniform probability.

Example: Stochastic Trees
variables: F
axiom: F
rules: F -> [F[+F]F[-F]F, F[+F]F, F[-F]F]
angle: 22.5

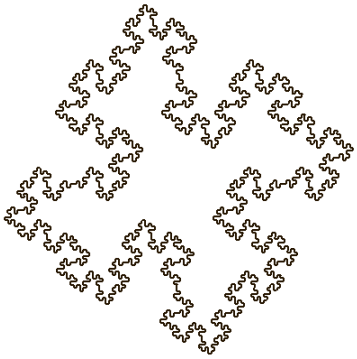
Finally, I moved to rendering 3D plants. The first step was a manipulation of the drawing technique. Whereas for the 2D objects I had been using lines in Three.js, I created a method that rendered cylinders instead of lines. To make those cylinders more realistic, I made them tapered, and also decreased the radius of the cylinders with branching, so leaves and branches appear thinner than the branches they came from. Additionally, I randomized the x and y angles of any branching in order to create a tree that extended branches from any direction around the trunk. I also determined which elements were leaves by determining the depth of each branch and coloring them based upon that

depth—green for those that were above a certain parameter. The colors were also randomized to add realism to the leaves of the plants.

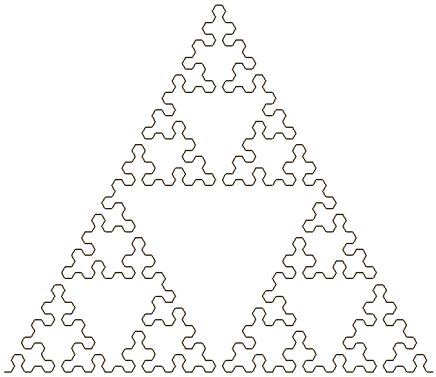
Results

The result was a simple web interface, powered by Three.js, where a user can toggle to see any of the above described shapes. This interface is viewable at http://chieni-lsystems.herokuapp.com/, and screenshots of such generated images are below.

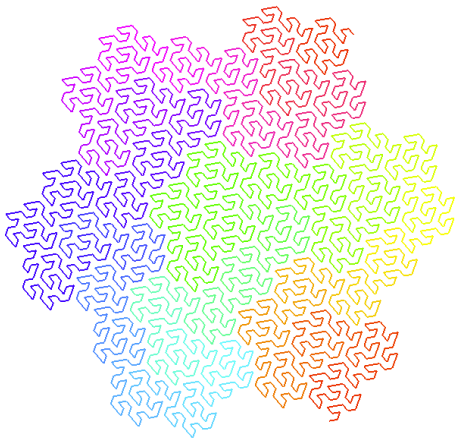
Fractals:



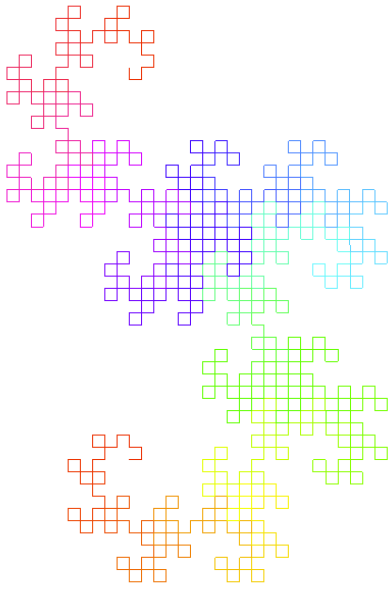
Koch Island, iterations = 3



Sierpinski Triangle, iterations = 6



Hexagonal Gopser, iterations = 4



Dragon, iterations = 10

2D Plants:

2D Stochastic Plants:

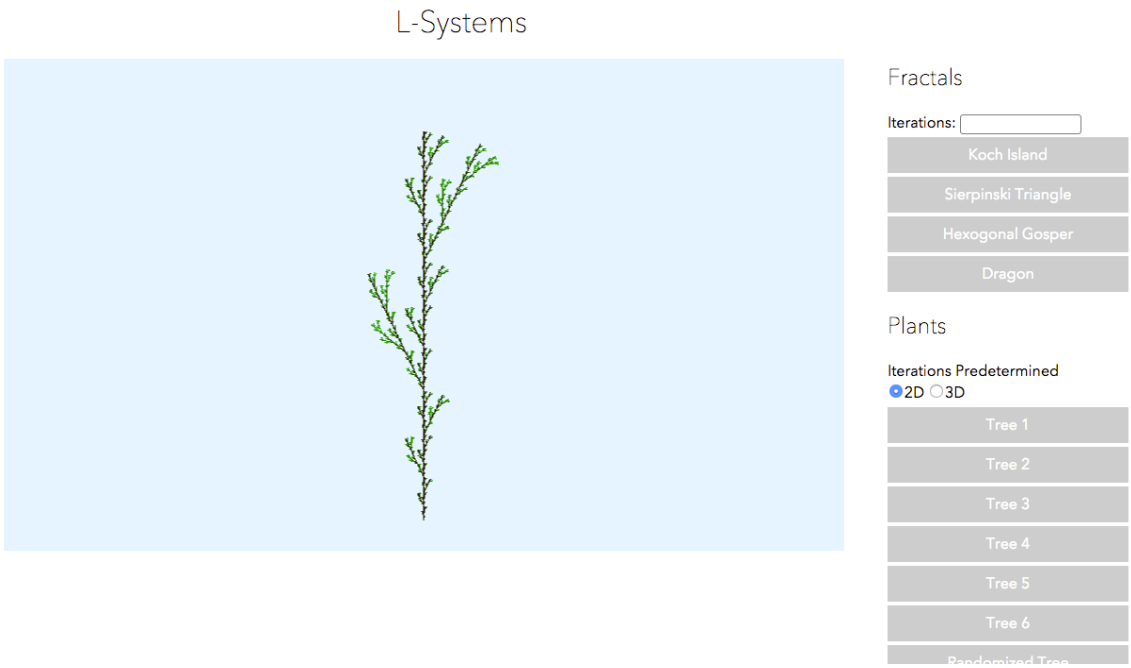
Same grammar used, generated three different times

3D Plants:

Forest of Stochastically generated trees, front view

Same forest as above, angled view

User Interface:



Conclusions

In conclusion, I was able to generate fractals, 2D deterministic plants, 2D randomized plants, as well as 3D randomized plants from L-systems. Not only that, but I included randomized and depth-based colorings of the leaves in order to add to the realism of the plants, and rainbow affects to illustrate the progression of the fractals. I saw a whole different side to graphics in this project, and gained an appreciation for the beautiful simplicity of L-systems and how they can be extended to great effect.