L-Systems
Mathematics and Graphics for Computer Games
MSc Computer Games and Entertainment

by Adam Joyce
18 November 2015

Coursework 1: L-Systems

This document is designed to give the reader a description of my first Maths and Graphics L-systems coursework on the MSc Computer Games and Entertainment course at Goldsmiths. It also acts as a guide to my code. Please note that it does cover the code in its entirety but highlights the key classes and functions that perform specific objectives.

Structure Overview

There are a number of different files included in my submission. The key files are:

- l_system_generator.h
- l_systems.h
- data1.csv – data8.csv

The l_system_generator.h file contains the code that I use to read in the L-system parameters and generate the axiom for the current stage of the L-system's iteration.

The l_systems.h file contains the code that is responsible for parsing, drawing, and displaying the L-system at each iteration. It also handles input from the user, altering the L-system accordingly.

The data1-data8 CSV files are used to store the initial parameters for each of the eight different L-systems.

Project Objectives

My four main project objectives were as follows:

- Read in configuration files containing the L-system parameters
- Display the resulting L-systems allowing for iterative steps to be displayed using hotkeys
- Hotkey each of the eight configuration files in order to easily load and display each L-system
- Hotkey the increment and decrement of three parameters that effect the structure of each L-system model

Reading the Configuration Files

Each CSV configuration file is made up of five different L-system parameter categories:

- Variables
- Axiom
- Rules
- Angle
- Iterations

The 'l_system_generator' class, in the 'l_system_generator.h' file, reads in the CSV configuration files. The class uses its 'read_data()' function to parse the CSV files and store each of the parameters by the categories mentioned above. During the parsing process, 'read_data()' skips a number of rows in the CSV files that are used as easy to read identity markers. I included these rows to make life easier when editing values within the files.

Displaying the L-System

Once the CSV file has been read and we have obtained the initial axiom, the ‘parse_axiom()’ function in my ‘l_systems’ class is called. ‘parse_axiom()’ loops through each character in the axiom interpreting its meaning by using a switch statement. Each case in the switch statement corresponds to a constant in the L-system, with the default clause skipping over any unevaluated progression values (values meant only for expression expansion).

For the position and angle of each line in the l-system to vary, I use a stack which stores both the position and angle variables of the line (or ‘node’). The stack is a simple ‘dynarray’ containing instances of my data class ‘tree_node’.

As mentioned, an instance of ‘tree_node’ contains only two variables - a position of type vec3, and an angle of type float. These variables are meant to signify the position and angle of the node at which the next line is to be drawn.

This ‘tree_node’ stack provides capability for the branching that is present in the majority of my L-system models. It does this by creating, and pushing, a new instance of ‘tree_node’ onto the stack whenever a ‘[‘ character is encountered in the axiom. We can then continue parsing and drawing along this new branch without worrying that the branching position or angle will be lost. Once the parser reaches the end of the branch, signified with a ‘]’ character, it pops the last ‘tree_node’ instance from the stack and sets the current position and angle equal to the previous instance of ‘tree_node’.

The remaining constants in the ‘parse_axiom()’ function are ‘+’ and ‘-‘, which alter the angle of the node (and thus line), and ‘F’, which determines the colour of the line and calls ‘draw_line()’ to actually draw it.

Iterating

My ‘parse_axiom()’ function only evaluates and draws a single ‘string’ of characters. For the iteration, I took the approach of generating the next step of the system (the new axiom), refreshing the display, and then parsing this new axiom.

The expansion of the existing axiom is handled by the ‘next_iteration()’ function in the ‘l_system_generator’ class. This function traverses the old axiom, detects any variables that have corresponding rules, and replaces those variables with their rules.

I wanted my program to not only be able to iterate forward through the steps in the L-system, but also support reverse iteration. As such, during each iteration I add the previous iteration’s axiom ‘string’ to the end of an array. It is then a simple case of setting the current axiom to equal the last string in the list. This is done using ‘previous_iteration()’ in the ‘l_system_generator’ class.

The ‘iterate()’ function in the ‘l_systems’ class is responsible for detecting ‘SPACE’ and ‘BACKSPACE’ key presses before the program iterates in the appropriate direction and redraws the model.

Hotkey Configuration Files

In this project I have tied the hotkeys ‘F1’ through ‘F8’ to load and display each of the different configuration files and thus l-system models.

Similarly to the ‘iterate()’ function, ‘load_csv_file()’ is used to detect one of the eight function key presses.

After a corresponding key press it is a relatively simple process to build the new model and display it. Firstly, I call a reset function on my ‘l_system_generator’ instance which empties all the parameter storage arrays enabling a new file to read in data successfully. Then I set the camera position and line width which are dependent on which l-system model is being loaded. Lastly I update the scene and set my dynamic overlay text appropriately. This takes place in the ‘switch_tree()’ function.

Altering Parameters

My final objective was to allow the user to increment and decrement three variables that are used in the construction of each l-system model. The three variables I used are:

- Angle variation
- Line length
- Line width

This turned out to be fairly trivial given the functions I had already put in place to construct the models. I did however run into one small hiccup.

I began by associating each of the variables with the remaining function, insert, and delete keys to detect when to increment or decrement. The mapping was as follows:

- Angle variation +/- = F9/F10
- Line length +/- = F11/F12
- Line width +/- = INS/DEL

After each press I check that the new incremented value is within the maximum and minimum bounds before changing the actual value and redrawing the scene.

This did result in a peculiar problem that still has me puzzled. Two of the higher numbered function keys would only perform their operation when pressed twice in succession. It was the two keys that corresponded to the decrement operations of the angle and line length variables - F10 and F12.

I was unable to locate any bugs in my code so I can only assume that the issue lies either within the octet framework itself or some interaction between those key presses and Visual Studio.

After altering the hotkeys to the (less intuitive) mapping listed below, the problem vanished.

- Angle variation +/- = TAB/CTRL
- Line length +/- = F9/ESC
- Line width +/- = INS/DEL

Since I have included functionally to grow the line length, I have also included camera controls with the following mapping:

- Move Up/Down = UP/DOWN ARROW
- Zoom In/Out = RIGHT/LEFT ARROW

Here is a link to my **Youtube video** demonstration of the project:

https://www.youtube.com/watch?v=zTMsck3Rn_0&feature=youtu.be

References

http://www.kevs3d.co.uk/dev/lsystems/ (Eighth L-System example)

https://en.wikipedia.org/wiki/L-system