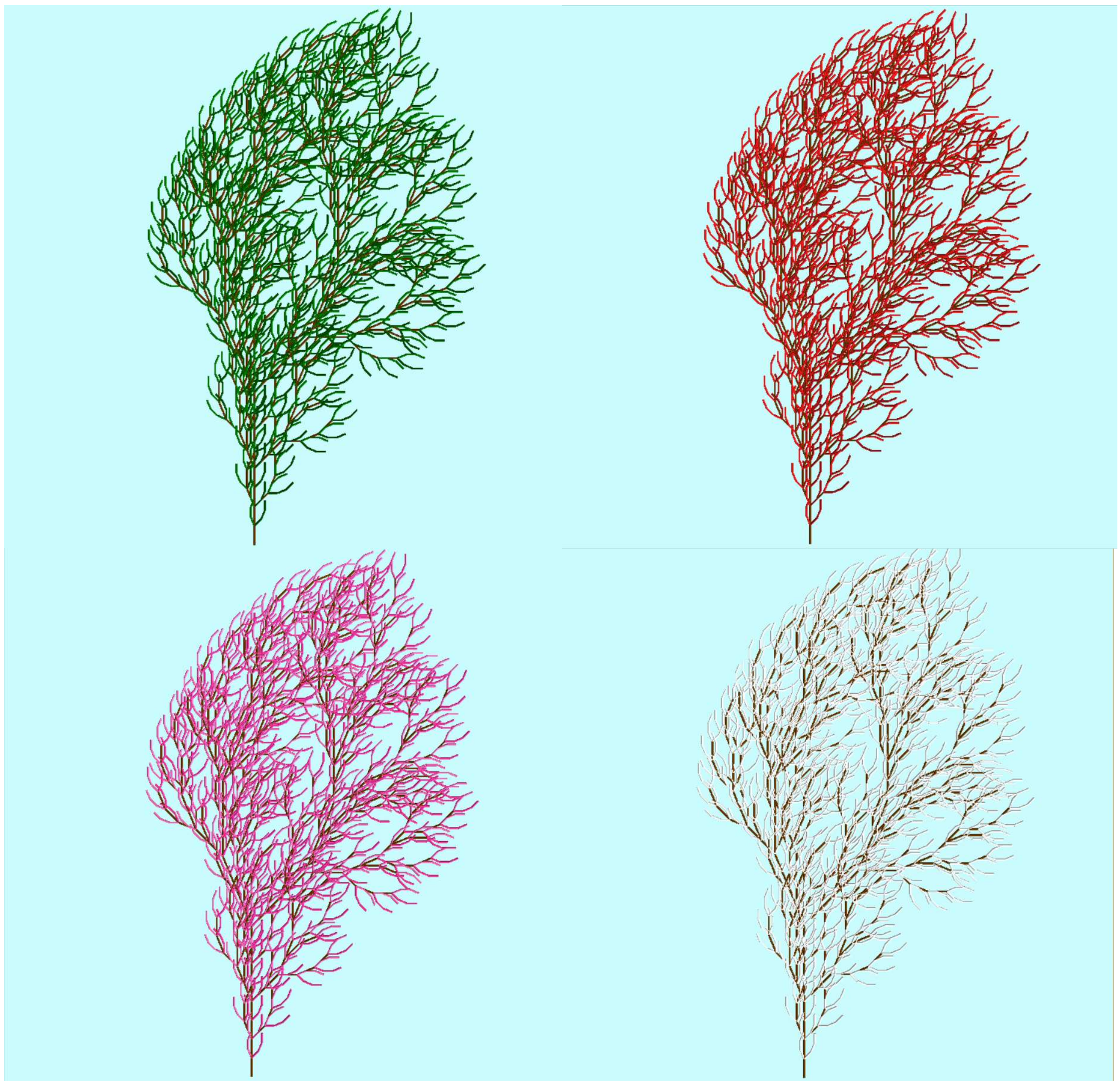
L-Systems Generator

CW1 – Mathematics and Graphics for Computer Games I

Elio de Berardinis

Submission Date: Nov. 24th 2015

No. of Pages: 10



INTRODUCTION

Lindenmayer systems (L-systems) were initially developed to organize the complex nature of plant development into a mathematical theory and to better understand this process in the field of plant biology. Subsequently, the theory was given a geometrical interpretation (Turtle Graphics) and became an effective tool to generate plants (and non-plants) graphic models [1].

L-systems are based on the concept of rewriting: complex objects are developed by iteratively replacing parts of a simple initial object using a set of “rules”. Turtle Graphics interpretation is given to each object and the structure is then drawn and rendered. In this context a simple L-system is then mathematically defined by 3 elements: The alphabet (V) containing all the possible words generated by the system, an initial statement or “axiom” (w) and the set of rewriting rule rules (P) for specific symbols in the alphabet called variables. In practice, the alphabet contains variables subjected to replacing rules and constants that are commonly given a turtle graphics interpretation of modifying certain parameters (e.g. angles, lengths, widths) or “push” (saving) and “pop” (retrieving) of a position/orientation in the structure. An example for a basic L-system and the turtle interpretation for each symbol is shown below:

Alphabet (V)

<u>Variables</u>: F (draw forward)

<u>Constants</u>:

+: turn left 26 degrees

-: turn right 26 degrees

[: push position and angle

]: pop position and angle

Axiom (w)

F

Rule (P)

F = F [+ F] F [- F] F

This example shows one of the simplest L-system which is 2D deterministic and context free known as DOL. In particular this is an edge-rewriting system creating a plant-like structure [1]. Other plant DOL systems are known as node-rewriting and include an extra variable serving as placeholder (not interpreted as “draw forward” by turtle) and having its own rewriting rule.

The outcome of a DOL system is always expected and fixed, from here the attribute deterministic. The rule chosen for each variable is also not affected by its “neighbours” (preceding or subsequent symbol), from here the attribute context-free.

The first step to make these systems more interesting is to add a stochastic feature where more than 1 rule exists for each variable and this is chosen randomly with a certain probability. This way each iteration will have a different outcome based on the randomly chosen rule for each symbol at that iteration. A modification of the rules for the previous example is shown below:

F = (0.6) F [+ F] F [- F] F

F = (0.4) F [- F] FF

Where the number in parenthesis represents the probability at which each rule is chosen.

Finally, a more sophisticated L-System is obtained by taking into account its context (or variables' neighbours) and choosing a (possibly random) rule only if certain conditions are met. These conditions can be dictated by the specific features of a certain plant/shape that one wants to obtain.

The field and tuning of L-system is a vast one and includes tweaking textures, creating complex 3D structures, sophisticated context-sensitive and stochastic models, most of which is beyond the scope of this coursework whose objectives are outlined in the next section.

OBJECTIVES

The basic objective of this coursework was to develop a solution to produce 8, 2D L-Systems whose details were to be read from a file. Of this 8 systems, 6 were simple plant-like DOL (3 edge-rewriting and 3 node-rewriting) detailed in Lindenmayer's "The Algorithmic beauty of plants" and the remaining 2 were to be chosen among provided or external sources. Also, the iteration increment/decrement as well as other parameters (e.g. angle, thickness etc.) of each system had to be assigned to specific hotkeys to be easily changed by the user. On top of these basic requirements, this work had the objective to add a relevant colour code for the plant structures and creating the extra examples as stochastic. Finally, each model would be given a "3D feel" by rendering each segment as a 3D cylinder rather than a simple line. Details on how these objectives were developed and achieved are presented in the next section.

IMPLEMENTATION and DEVELOPMENT

The project was created and developed with Andy Thomason's OCTET Framework in C++ using Microsoft Visual Studio 2013. Briefly, a new OCTET example was created through the Python script "make_example" within OCTET's scripts as detailed in the Framework README file [2]. The basic new example created by the script contains a "main.cpp" file and a header file named "LSystems.h" (the name given by the user to the new OCTET project). The automatically generated main file contains the information to run the example whose details are then contained in the #-included LSystems.h. The header file at this point can generate a simple 3D shape in space.

In order to separate the main functions of the L-System program, an extra header file named "tree_elio.h" was created and #-included in LSystems.h. The new header (tree_elio.h) would deal with reading the files containing the different L-systems details, store the read data and handle the process of re-writing, while the first header (LSystems.h) would deal with the graphical interpretation of drawing, rendering and handling the user inputs while constantly updating the scene at every frame. Details on the content of these header files as well as their interaction is described below:

tree_elio.h

This header deals with reading the data from a text file, store the information and rewrite the model at each iteration. All the content of the file is created in a class within OCTET namespace called "tree". For the purpose of storing the data, dynamic arrays (type "`dynarray<char>`" in OCTET) were created to separately save information about the variables, the constants, the axiom and the type of rule used at each iteration (for stochastic models). To store the rewriting rules, a hash map (type "`hash_map<char, dynarray<char>>`" in OCTET) was created to easily assign each character (or key as called in the project) to its specific rule.

The function "`void read_text_file(int example_number)`" reads the file of a specific L-system whose identification is given by the function argument "example_number" and saves it into a new array (`dynarray<uint8_t> file_content`). This array is then given as argument to the function "`void read_data(dynarray<uint8_t> file_content)`" that sorts the data in the correct previously detailed dynamic array or hash_map (variables, constants, axiom, rules). The sorting is easily done by the function as all text files are written in a specific way as shown in the example below:

X , F;	(First line: variables saved in the variables dynarray)
+ , - , [,];	(Second line: the constants saved in the constants dynarray)
X;	(Third line: the axiom saved in the axiom dynarray)
X - > F [+ X] F [- X] + X;	(From the 4th line onwards: the rules saved in the hash map)
F - > F F;	

Once the first function reads the file and stores the read characters in the array the second function separates each type of data, sore it into a different array/hash_map and moves to the next type.

Next, the function "`void evolve()`" is called at each iteration to expand the axiom according to the rule (deterministic examples). Briefly, the axiom array is read and each time a variable is encountered and recognized by the function "`bool is_char_in_array(char ch, dynarray<char> arr)`" the rule matching that particular character is stored instead of the single character in the newly created "new_axiom" dynarray. If a constant is encountered it gets saved unchanged in "new_axiom". At the end of the process the content of "new_axiom" is copied to the initial "axiom" dynarray that gets resized. Now "axiom" dynarray contains the information about the grown model.

Two different stochastic models were implemented based on the increased amount on randomness introduced. To iterate the first type of stochastic system the function "`void evolve_stoc()`" was developed. This function works exactly like the previously seen "`void evolve()`" for deterministic systems with the addition that the rule to choose from is randomly chosen among 3 available in the relevant example's text file. This is achieved by "seeding" the random function "rand()" with the current time through the function "`srand(static_cast<unsigned int>(time(NULL)))`". Next the random function generates a random number that selects the rule to use through a simple "switch case" construct. The character identifying the rule to use is then stored in one of the initially defined dynamic array (`dynarray<char> rule_type_stoc`) to be used in the devolution method (see later in the description). The amount of randomness in this case is limited as 1 different rule is chosen at each iteration for all the variables. In other words, if the axiom at some stage is "FF + [F] - F" all the F will follow the same rule at that iteration.

The second type of stochastic method is handled by the function "`void evolve_stoc_type_2()`" and increases the variability by choosing a new rule each time a variable is encountered rather than 1 different global rule for each variable in the axiom. In this case the rule type used each time is not stored.

So far we described how the axiom is changed when the system grows. When the system is stepped back or "devolved" things are handled a bit differently. For the deterministic models the function "`void evolve()`" is simply called 1 time less than the current iteration (e.g. if we are at iteration 4 it will be called 3 times) as for deterministic models the outcome will always be the same. For the first stochastic model if we were to call the function "`void evolve_stoc()`" in the same way we would end up with a different model. For this reason the function "`void devolve_stoc(unsigned int iteration)`" was developed. This is a modified version of "`void evolve_stoc()`" where the rule to use at each iteration is retrieved in the dynamic array "`rule_type_stoc`" created during the evolution method. For the second stochastic system a devolution method was partially created but not optimized, hence it was not included at this stage (for more on this please check the details in the "Conclusion and Credits" section of this report).

The last 3 functions present in "tree_elio.h", "`void reset_stoc()`", "`void decrese_stoc_array()`" "`dynarray<char> get_axiom()`" have housekeeping and resetting purposes. The first one resets the "`rule_type_stoc`" array, the second one resizes it and the third one returns the new axiom created in one of the evolution methods.

LSystems.h

This header deals with drawing and rendering the L-systems and geometrically interpreting the information read and stored by the methods shown in tree_elio.h. Similarly to tree_elio.h, all the content of this file is included in the class "lsystems" defined in the namespace OCTET.

In this class, another class called "`class node`" was created where 2 constructors and simple functions and variables were declared to set and retrieve the information about position and angle of each segment used to draw the L-system.

Next, a series of important parameters were declared and/or initialized such as an object of the class "tree" to access the functions present in tree_elio.h, the different colour materials as well as many different constants (geometrical parameters of the segments) and system states (current iteration, maximum iteration, current example, camera displacement etc.).

In OCTET things are handled by 2 main functions "`void app_init()`" called at the beginning to initialize everything and "`void draw_world(int x, int y, int w, int h)`" that updates the scene at every frame. App_init calls a number of different functions and methods at the beginning, specifically:

- "`t.read_text_file(current_example)`" Reads the first L-system file and operates on it as explained before.

- "`set_MAX_iteration()`" A simple housekeeping and optimization function that sets the maximum number of iteration for a specific L-system in order to minimize chances of crashing.

- "`is_stoc()`" Checks if the current system is stochastic or deterministic.

- "`print_instruction()`" Prints hotkeys and other instructions (current example, iteration) for the user to the console.

- Initializes the camera position and the different colours according to RGB values.

- "`create_geometry()`" in order to interpret and draw the axiom read from the first file (in this case is the axiom itself or "iteration 0".

The function "`create_geometry()`" gives turtle graphics interpretation to the read files by assigning the correct meaning to the different variables and constants (e.g. F draw forwards; +, - increase/decrease the angle; [,] push and pop position and angle). In addition to this it changes the colours of the branches according to extra variables for colour code inserted directly in the text files of the L-system. When the command for "draw forward" is encountered the function "`vec3 draw_segment`" is called. Here the position of the future segments are calculated, placed in the correct node in space and given a mesh according to the correct colour.

As mentioned before, the other main function in this header is "draw_world" that updates the scene at every frame. On top of checking other housekeeping functions ("`set_MAX_iteration()`" and "`is_stoc()`") it calls the function "`void handle_input()`" that takes in the user input where the user can modify the parameters of the models.

Handle_Input() contains all the methods to iterate/de-iterate the system, move the camera, zoom in and out, increase/decrease the angle, increase/decrease the thickness of the branches, change colour of the "leaves" based on seasons and switch between different L-systems examples as well as rotating the model. For some of these actions, camera adjustments are made to present the model in the best way possible to the user (e.g. automatically change thickness of the branches or camera position) and print relevant information to the console (e.g. current iteration or example).

Briefly, the evolution method calls the function evolve(), evolve_stoc() or evolve_stoc_type_2() according to the current example type and redraws the model with the function "`draw_again()`" that simply re-sets the scene and calls create_geometry() (detailed before). The devolution method works as described in tree_elio.h by either calling the evolve() function up to the number of iteration before the current one (deterministic) or the devolve_stoc() for the first type of stochastic system. No devolution method is present for the second type of stochastic system that is simply reset when the hotkey is pressed.

All the other parameters are changed with a same method: once the parameter has been increased/decreased the whole model is redrawn in the exact same position and orientation with the help of the function "`void re_draw()`" giving the illusion everything is happening on the same model.

Camera movements are obtained implementing the rotate() function in OCTET.

When the user switches to a new example, the function "`void reset_variables()`" resets all the state variables and constants to their initial value.

The Hotkeys and their use in the program are detailed below (mouse left/right click are used to change example):

Parameters

- Spacebar/Backspace: Iterate/De-iterate the model
- Esc/Tab: Increase/Decrease the angle
- f2/f1: Increase/decrease the thickness of branches
- f3: Change colour of the leaves

Camera

- Shift/Ctrl: Zoom in / Zoom out
- Direction arrows: move camera around
- Delete: Rotate the model

Images showing the different models obtainable in this project as well as the different effects of the parameters increase/decrease/change are shown in the following pages.

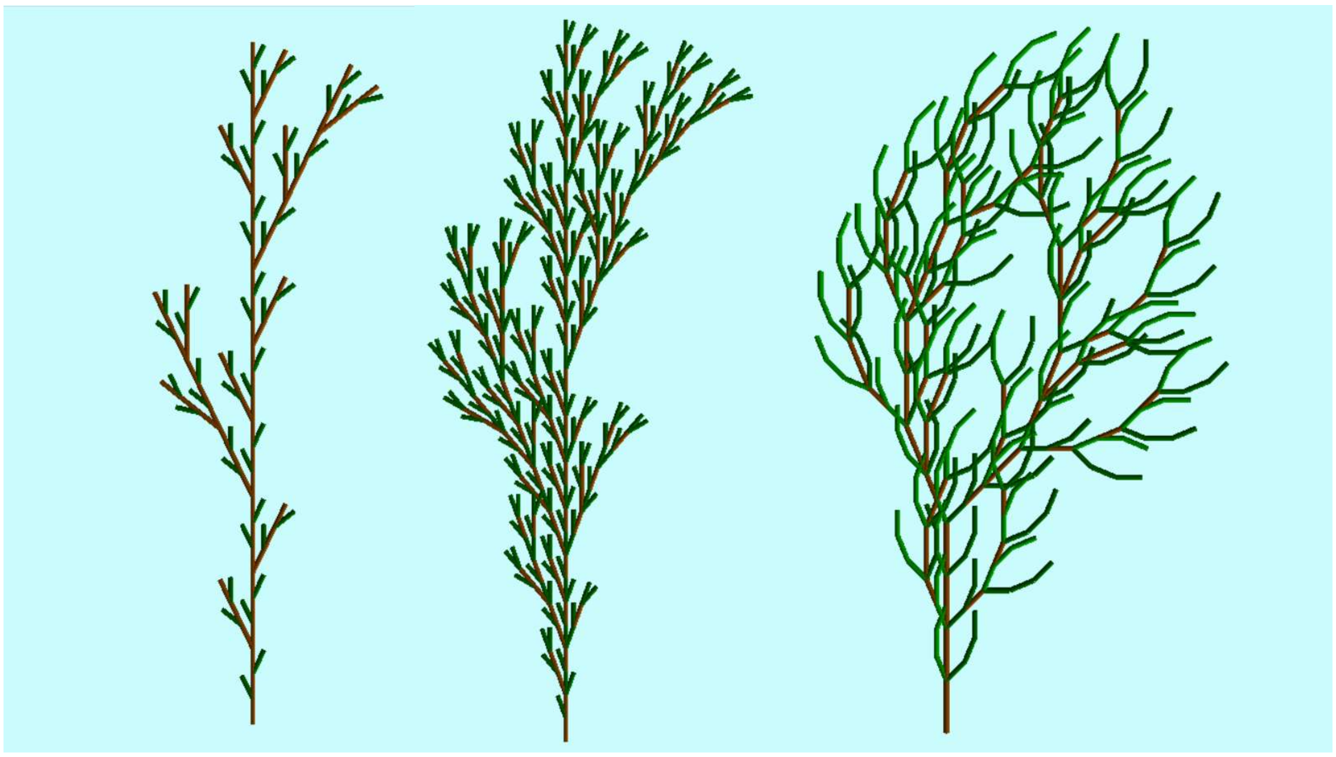


Figure 1 – Examples 1-3: Deterministic edge-rewriting systems

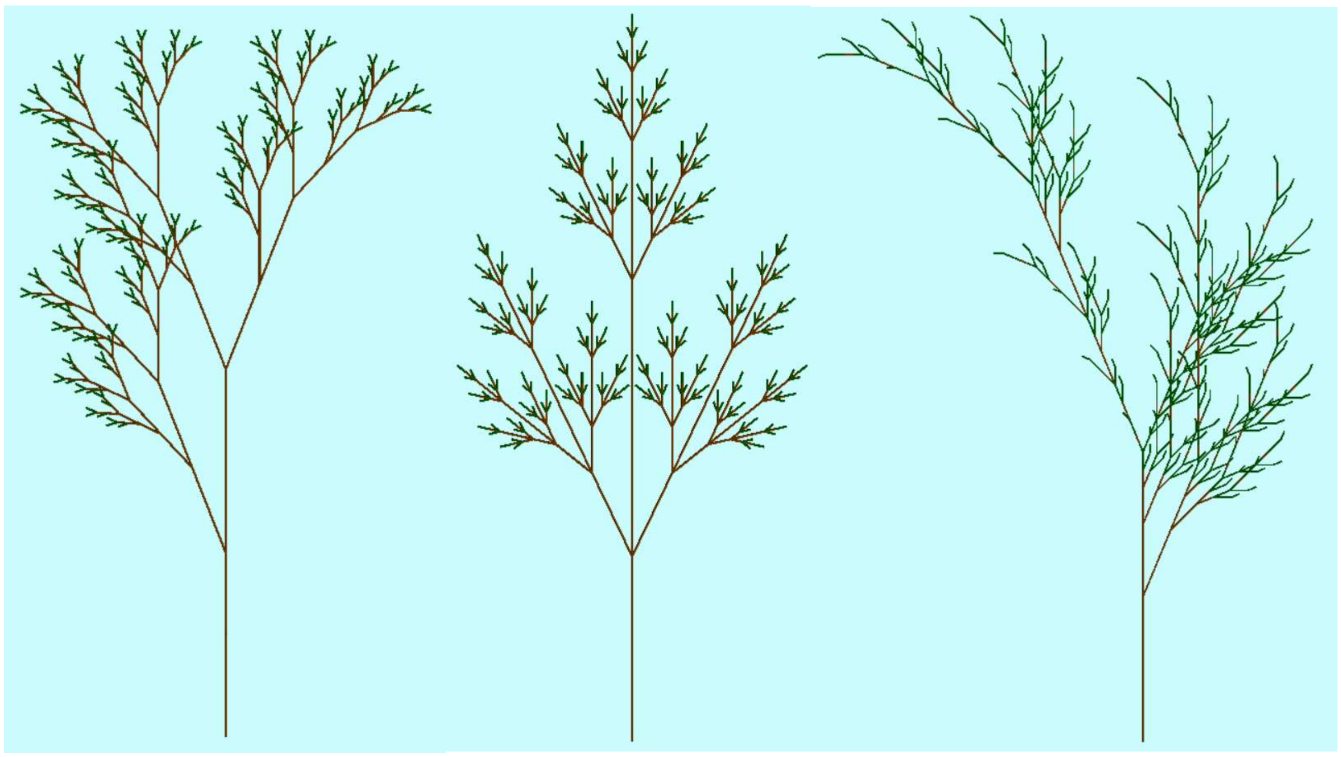


Figure 2 – Examples 4-6: Deterministic Node-rewriting systems

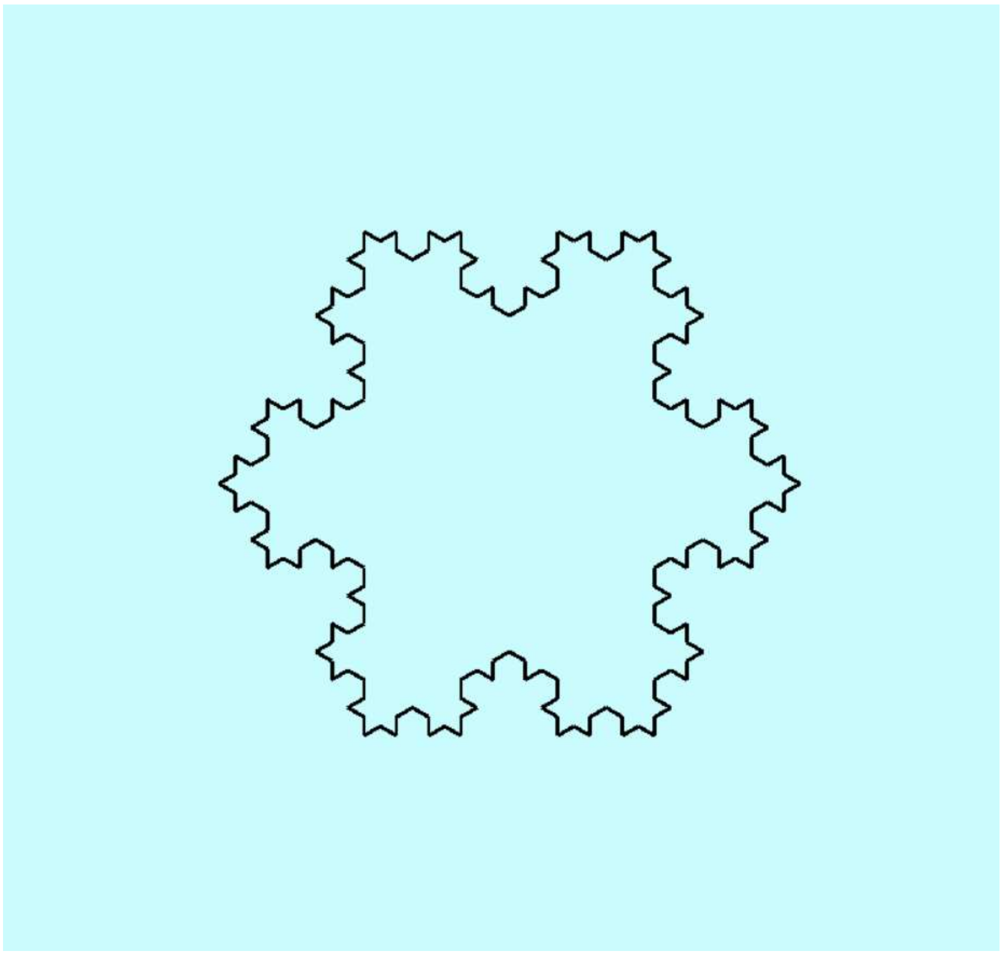


Figure 3 – Example 7: Kotch Construction "Snowflake curve"

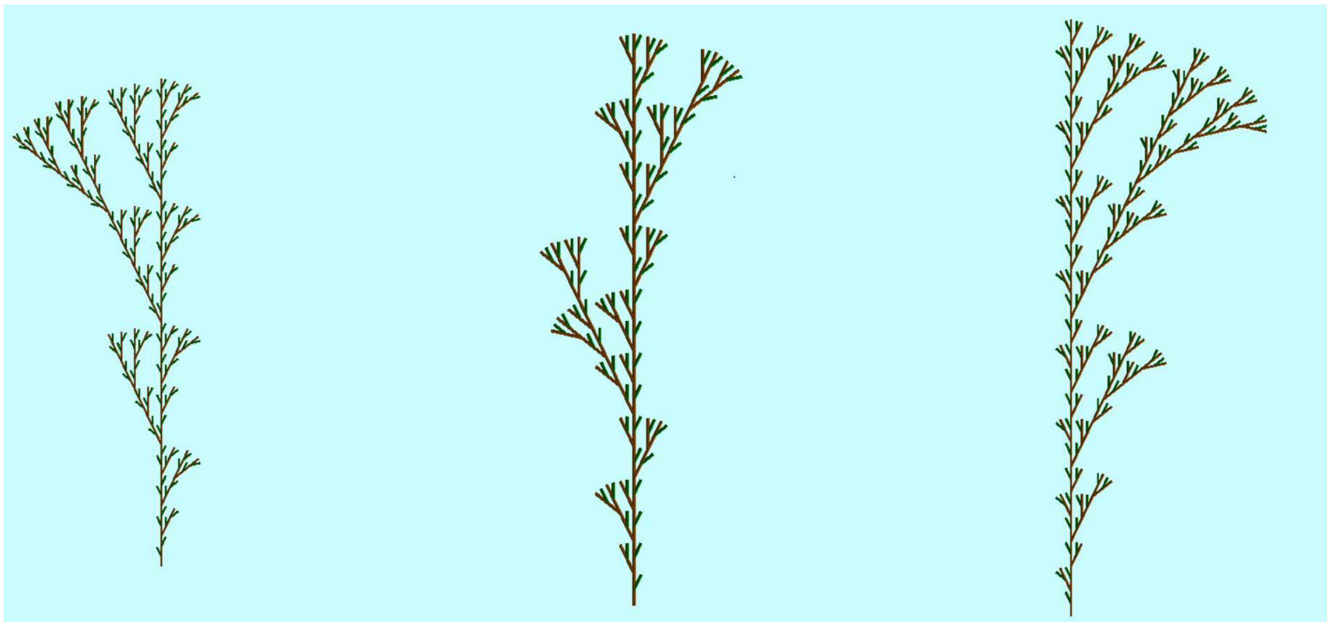


Figure 4 – Example 8: 3 different outcomes of a stochastic version of Example 1 (Stochastic Type 1)

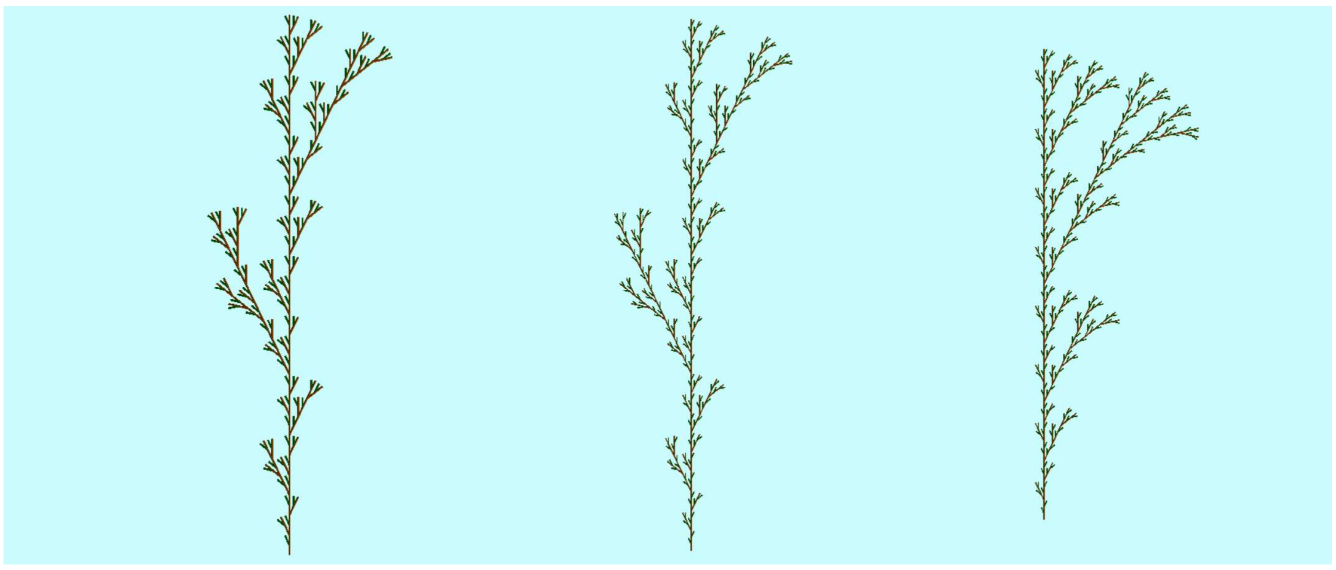


Figure 5 - Example 9: 3 different outcomes of a stochastic version of Example 1 (Stochastic Type 2)

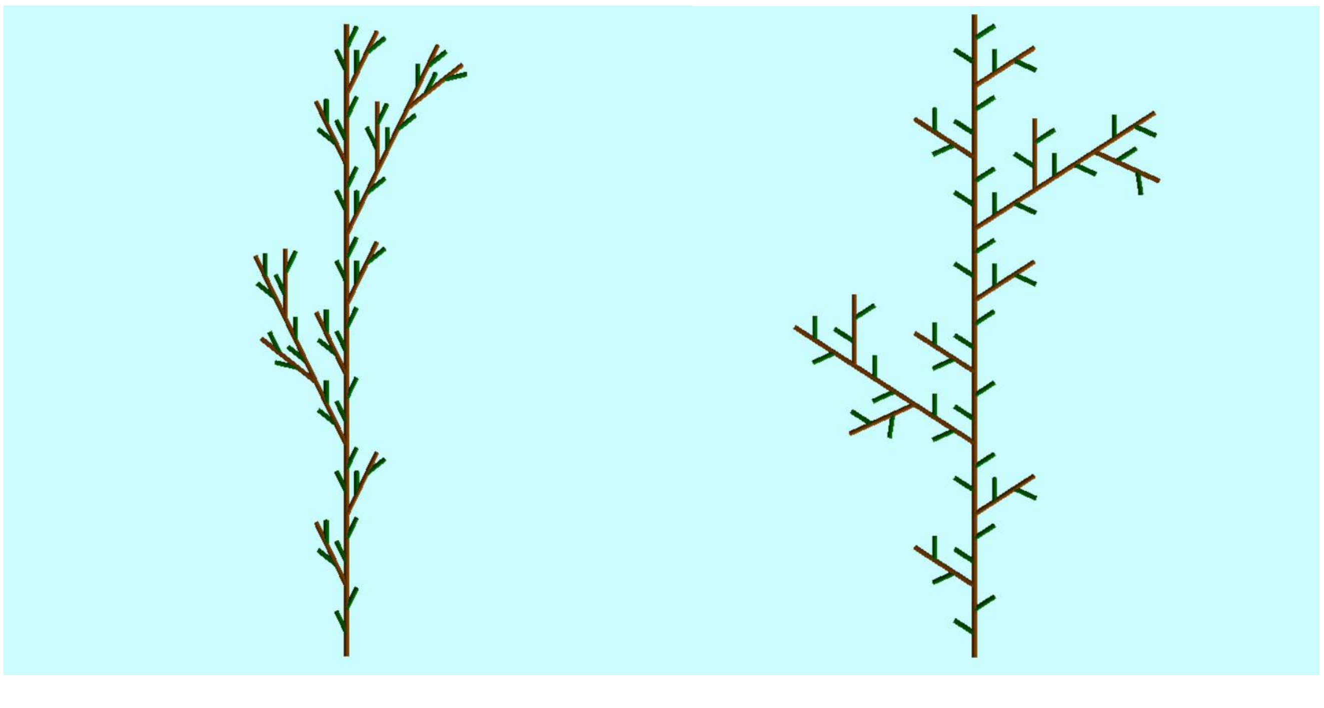


Figure 6 – Effect of Angle increase on Example 1

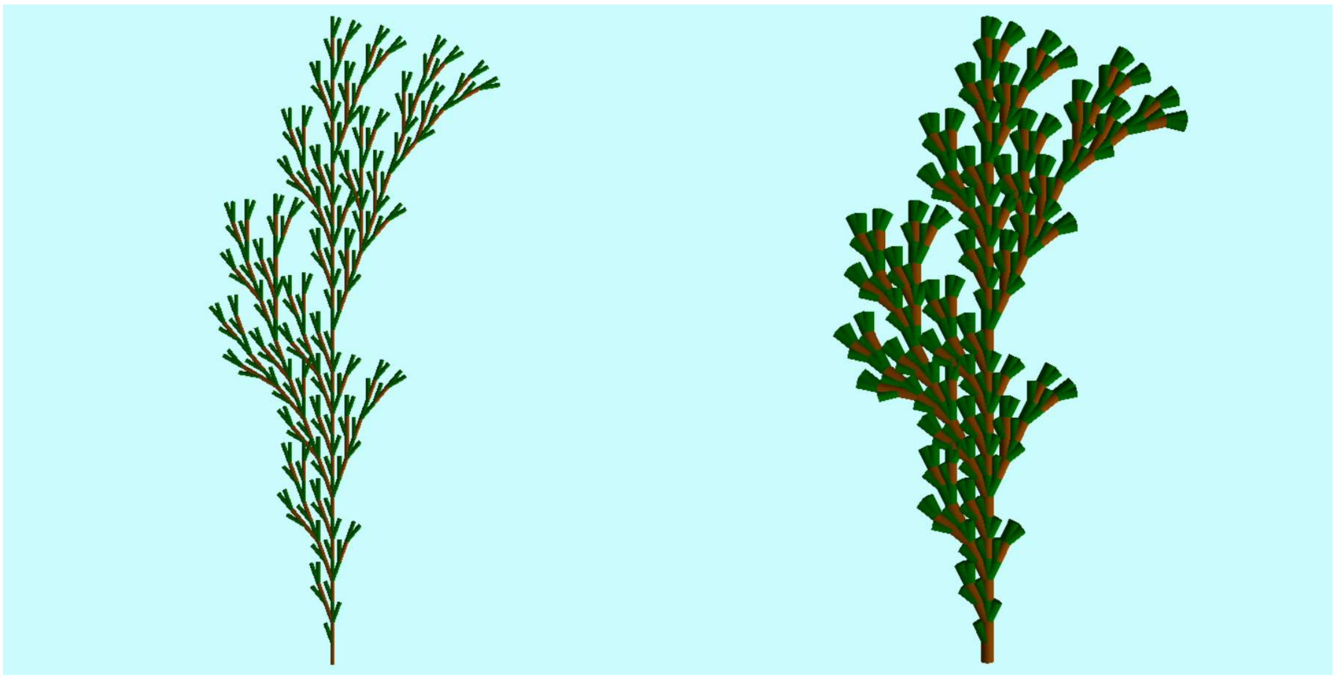


Figure 7 – Effect of thickness increase on Example 2

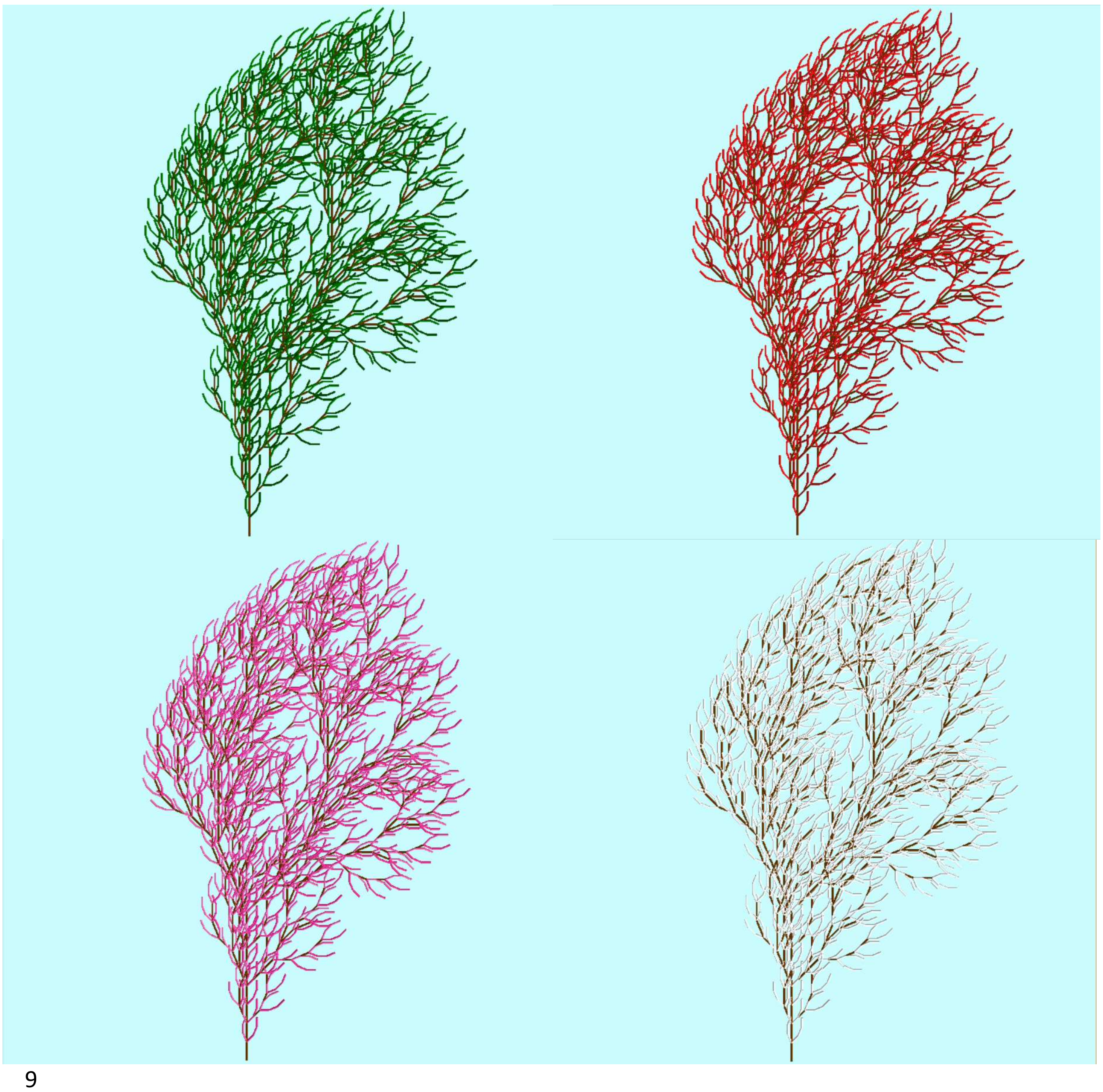


Figure 8 – Different leaf colour patterns according to season on Example 3

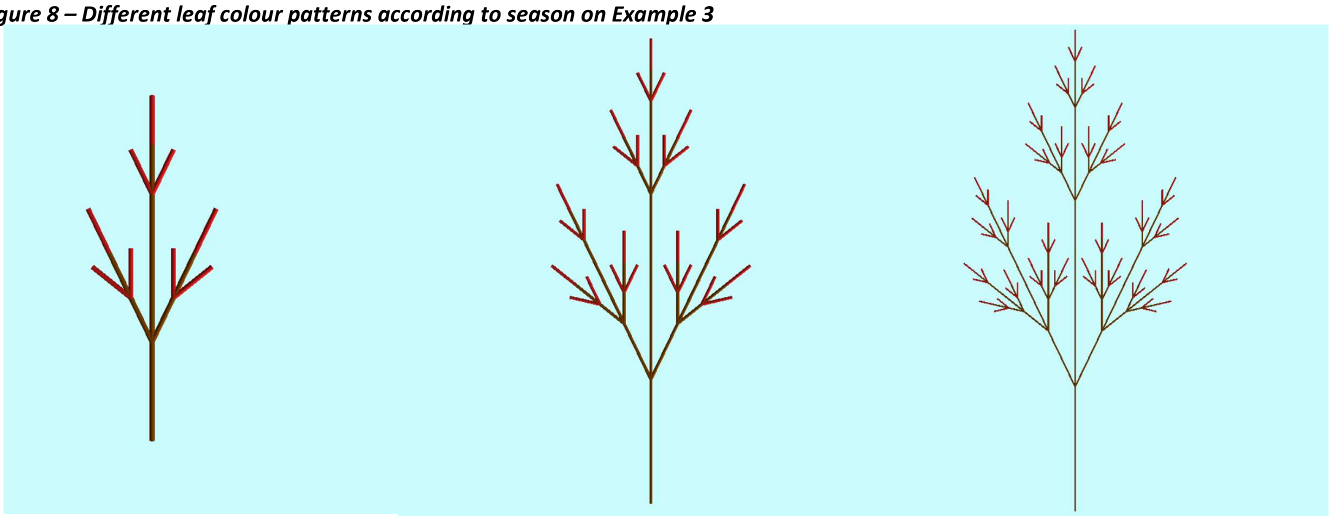


Figure 9 – Subsequent iterations in Example 5

CONCLUSION and CREDITS

It is important to note that given the flexibility of the text file reading method, virtually every basic 2D deterministic or Stochastic L-system could be rendered.

Future Improvements could add more complex 3D structures, methods for context-sensitive L-Systems and optimization for speed and maximum number of iteration achievable. Also, a more user friendly user interface on screen is desirable instead of using the console.

A method to devolve the second type of stochastic system has been partially developed but not optimized. The relevant code can be found on github/eliodeberardinis on the branch "trying_stochastic_type_2". Fork the project, run the solution in Visual Studio and select example 9 for testing.

This complete project is also present on github/eliodeberardinis on the branch "Math_Graphics_Assignment_1_LSystems_Complete".

A demo video of the project can be found on youtube: https://youtu.be/FZDgwvCKaiw

Finally I would like to credit my colleague Mircea Catana whose methods for reading the text files, storing the data and create the basic geometry for DOL systems have been derived, modified and improved in this work.

REFERENCES

[1] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. The AlgorithmicBeauty of Plants. Springer-Verlag, 1990. Electronic version (2004): http://algorithmicbotany.org/papers/#abop.

[2] Andy Thomason's OCTET: https://github.com/andy-thomason/octet