Procedural Plant Generation and Simulated Plant Growth

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Information Science

in

Computer Science

at Massey University, Albany,

New Zealand.

Matthew Halen Crankshaw

2019

Acknowledgements

I would like to start by thanking my supervisor Dr Daniel Playne for all of your support, guidence and feedback during the course of this thisis. Additionally I would like to acknowledge Dr Martin Johnson. You have both been an inspiration not just to myself but to all of your students studying Computer Science.

A very special gratitude goes to my colleagues Dara and Richard in the center for parallel computing for their assistance and friendship.

I certainly would not be where I am today if it weren't for my siblings, mother and father for their continued love and support.

Finally, to my beloved partner Romana, thank you for your unwaivering support and encouragement throughout the last year of study, and in the writing process or this thesis. This accomplishment would not be possible without you.

Abstract

Contents

1 Introduction **9**

1.1 Motivations . . . 9

1.2 Introduction to Procedural Generation . . . 10

1.3 Introduction to Rewriting Systems . . . 11

1.4 Introduction to Formal Grammars . . . 12

1.5 Structure of Thesis . . . 12

2 Lindenmayer Systems **14**

2.1 Simple DOL-system . . . 16

2.2 Interpreting the DOL-system String . . . 17

2.3 Branching . . . 21

2.4 Parametric OL-systems . . . 24

2.4.1 Formal Definition of a Parametric 0L-system . . . 24

2.4.2 Defining Constants and Objects . . . 25

2.4.3 Modules With Special Meanings . . . 27

2.4.4 Representing L-system Conditions . . . 28

2.5 Randomness within L-systems . . . 29

2.6 Stochastic Rules within L-systems . . . 30

2.7 Computing L-systems . . . 32

2.8 Summary . . . 32

3 L-system Rewriter Implementation **34**

3.1 Environment and Tools . . . 35

3.2 The L-system as an Interpreted Grammar . . . 36

3.3 The Syntax of a Parametric L-system . . . 37

3.4 The L-system Lexical Analyser . . . 38

3.5 The L-system Parser . . . 40

3.5.1 Backus-Naur Form of the L-system Grammar . . . 41

3.5.2 Dealing with Constant Values and Objects . . . 43

3.5.3 Implementing Modules and Strings . . . 44

3.5.4 Implementing Arithmetic Expressions Trees . . . 44

3.5.5 Implementing Random Ranges . . . 45

3.5.6 Implementing Stochastic Rules . . . 46

3.6 The String Rewriter . . . 47

3.7 Summary . . . 52

4 Mathematics For 3D Graphics 53
4.1 Vectors . . . 53
4.2 Matrices . . . 56
4.3 Quaternions . . . 58
4.4 summary . . . 60

5 L-system String Interpreter Implementation 61
5.1 Turtle Graphics Interpreter . . . 62
5.2 Model Generator . . . 64
5.3 Renderer . . . 66
5.3.1 Models and Buffer Objects . . . 67
5.4 Shaders . . . 67
5.5 Summary . . . 67

6 Physics Simulation 68
6.1 Branch Physical Properties . . . 69
6.2 Hook's Law . . . 70
6.3 Equations of Motion . . . 70
6.4 Updating Branches . . . 71
6.5 Results . . . 71
6.6 Summary . . . 72

7 Discussion 73

8 Conclusions 74

A Appendix 75
A.1 Appendix 1 . . . 75
A.2 Bibliography . . . 75

List of Figures

1.1 Construction of the snowflake curve[Prusinkiewicz and Hanan, 2013]. 11

1.2 Diagram of the Chomsky hierarchy grammars with relation to the 0L and 1L systems generated by L-systems. 12

2.1 Diagram showing the relationships between the L-system grammar, language, rewriter, and interpreter. 15

2.2 Diagram of the 3D rotations of the turtle. 18

2.3 Diagram showing a turtle interpreting simple L-system string. 19

2.4 Koch Curve. 21

2.5 Sierpiński Triangles. 21

2.6 Diagram showing a turtle interpreting an L-system using the branching symbols. 22

2.7 Diagram showing a turtle interpreting an L-system with nested branching. . . . 23

2.8 Fifth generation of the fractal bush L-system. 23

2.9 Fifth generation of the fractal tree L-system 24

2.10 Diagram of an L-system Using Multiple Objects. 26

2.11 3D Parametric L-system. 27

2.12 Condition statements used to simulate the growth of a flower. 2nd generation on the left, 4th generation in the center and 6th generation on the right 29

2.13 Different Variations of the Same L-system with Randomness Introduced in The Angles. 30

2.14 Representation of an L-system with a probability stochastic with a 0.33 probability for each rule. 31

2.15 Diagram of the procedural generation process. 32

3.1 Diagram of the Parts of The Rewiting System. 35

3.2 Diagram syntax tree for an expression. 41

3.3 Diagram of an expression tree. 45

4.1 Table of common dot product tests between two vectors. 54

4.2 Diagram of the cross product of two vectors a and b. 55

5.1 Diagram of the stages of L-system interpretation and rendering 62

5.2 Diagram for the properties of a joint 63

5.3 Diagram of a simple plant skeleton with joint position and orientation. 64

5.4 Example of the continuity problem faced with stacked branching with a 25° bend per joint. 65

5.5 Example of linked branching with a 25° bend per joint. 66

5.6 Stacked Vs Linked. 66
5.7 Diagram showing the structure of a vertex buffer object. 67

6.1 Examples simulating gravity on different 3D models 71
6.2 Examples simulating gravity on different 3D models 72
6.3 Examples simulating gravity on different 3D models 72

List of Tables

2.1 Table of turtle graphics instructions symbols and their meaning to the interpreter 18

2.2 Table showing each instruction symbols and their interpretation for the L-system 2.3 19

3.1 Table of Valid Lexer Words 39

3.2 Table of turtle instruction symbols and their meaning to the interpreter 43

3.3 Table of turtle instruction symbols and their meaning to the interpreter 43

3.4 Table of the stochastic rules probabilities within a stochastic group. 47

4.1 Table showing the dot product tests and an example of their use. 55

5.1 Table of turtle instruction symbols and their meaning to the interpreter 63

Chapter 1

Introduction

Procedurally generating 3D models of plant-life is a challenging task, largely due to the complex branching structures and variation between different types of plant species. Up until recently, all assets within 3D graphics applications either had to be sculpted using 3D modeling software or scanned using photogrammetry, laser triangulation or some form of contact-based 3D scanning. These methods are still used today but tend to be very time consuming and extremely costly. With the increase in computational power over the last few decades more emphasis has been placed on the use of procedural generation. Which can be used to create complex structures such as terrain, architecture, sound and 3D models with far greater speed than previous techniques, and often much better realism than would be possible with artists. Plant-life stands as a challenge due to the thousands of species, each with their unique structure and features. It is difficult to define a system that can represent them all in a way that is simple, understandable and accurate. The Lindenmayer System (L-system) stands as a solution to this problem, it was originally developed by Aristed Lindenmayer as a method of representing the development of multicellular organisms [Lindenmayer, 1968]. This has since gained popularity in the area of procedural generation and has been adapted to represent different types of structures. L-systems have been adapted to represent plant-life, such as trees, flowers, algae, and grasses. Whilst still applying to non-organic structures such as music, artificial neural networks, and tiling patterns [Prusinkiewicz and Hanan, 1989].

1.1 Motivations

The L-system, in its most basic form, is a formal grammar that contains a set of symbols or letters that belong to an *alphabet*. A starting string or *axiom* is created using the alphabet, as well as a set of production rules. The production rules are applied to each symbol within the axiom string, and each rule dictates whether or not the symbol can be rewritten and what they will be rewritten with. In essence, an L-system uses the set of production rules to generate a resulting string of symbols which follow those production rules. What the resulting strings' is ultimately going to represent depends on how it is interpreted. In this case, the string of symbols can be interpreted to generate a model of a plant. This thesis develops upon the L-system concepts described by Przemyslaw Prusinkiewicz and Aristid Lindenmayer to procedurally generate structures of plant-life in real-time. The L-system grammar allows the construction of a plant to be described in a human-readable, formal grammar. The grammar can be used to specify variation in shape, size, and branching structure within a particular species. Furthermore, this thesis will also investigate the use of a parameterised L-systems to provide physical properties using string rewriting. Which in turn will enable the animation and physical behavior of the plant that it generates, thus making it possible to simulate external forces such as gravity and wind.

This chapter will provide an overview for how to improve the procedural generation of plant-life in 3D applications and the motivations doing so. It will then introduce the concepts of procedural generation, rewriting systems, and formal grammars. This chapter will briefly describe how to apply procedural generation to the development of plant-life, and will provide sufficient background as to the use of formal grammars as a means of describing complex L-system languages. Finally, there will be an outline as to the structure of this thesis.

1.2 Introduction to Procedural Generation

Procedural generation is used in many different areas and applications in computer graphics, particularly when generating naturally occurring structures such as plants or terrain. An effective procedural generator is capable of taking input in the form of a relatively simple description of what it should be generating; its job is then to computationally create the structure in a way that is accurate to the description given. Currently, there are three main methods for procedurally generating models of plant-life; these are genetic algorithms [Haubenwallner et al., 2017], space colonisation algorithms[Juuso, 2017], and L-systems. The genetic algorithm and space colonisation algorithms are similar in that they require the overall shape of the plant to be described by simple 3D shapes; the algorithm then creates a branching structure that matches these shapes. The limitation of these methods is that the 3D description is not very specific, and although it can get good results for trees, it may not be able to generate different types of plant-life, such as flowers. The L-system, on the other hand, relies on a method of string rewriting, whereby the rewriting is based on a set of production rules to generate a string of symbols that obey those rules. A separate system can later interpret this string to create the model. The L-system procedural generation, therefore, has two different systems within it, one of string rewriting and one of interpretation of the generated string. This makes it quite easy for the same L-system to generate very different results based upon the interpretation.

Plant-life can have very complex and seemingly random structures; however, with closer observation, trees of a similar species have distinct traits and features. For instance, a palm tree has long straight trunks with large compound leaves exclusively near the top, branching in all different directions. Comparatively, a pine tree has a long straight trunk with many branches coming off in different directions perpendicular to the ground, from its base to the top of the trunk. These are two very different species of trees; the palm belongs to the Arecaceae family, whereby the pine belongs to the Pinaceae family. They look different; however, they share very similar properties, such as their long straight trunks. The challenge behind the procedural generation of plant-life is providing a human-readable grammar that describes in sufficient detail, how to generate a 3D model. Whilst allowing for randomness and variety within the generation process, such that variations of a particular species can be created without repetition. The grammar for procedural generation should also be relatively straightforward and intuitive, and must accurately represent what it is going to generate. Furthermore, the description must not be limited to only known species of trees, as some graphics applications may require something other-worldly.

1.3 Introduction to Rewriting Systems

Rewriting systems are the fundamental concept behind L-systems. In their most basic form, rewrite systems are a set of symbols or states, and a set of relations or production rules that dictate how to transform from one state to the other [Prusinkiewicz and Lindenmayer, 2012]. These production rules can be used to generate complex structures by successively replacing parts of a simple initial object with more complex parts. Rewrite systems can be non-deterministic, meaning that there could be a transition that depends on a condition being met or on neighbouring states. The rewriting concept means that any next state can rely upon some conditions necessary for transformation. If the condition evaluates true, the state is rewritten; otherwise, it remains the same and is checked in the next rewriting stage. A graphical representation of an object defined in rewriting rules can be seen below in figure 1.1 below, called the snowflake curve proposed by Von Koch [Koch et al., 1906].

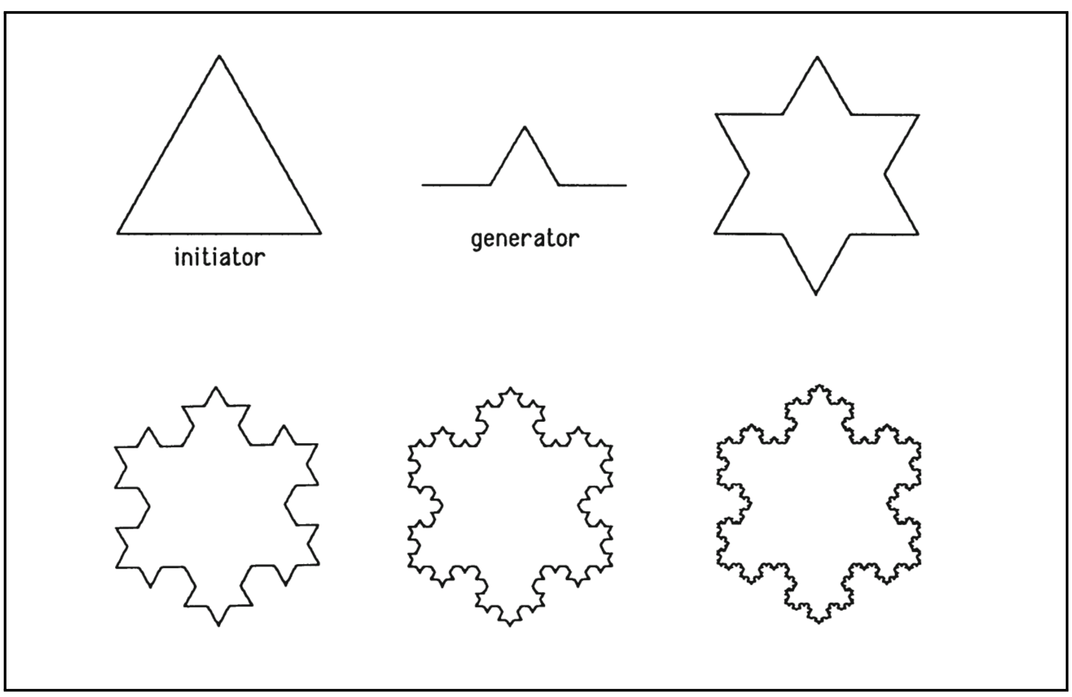


Figure 1.1: Construction of the snowflake curve[Prusinkiewicz and Hanan, 2013].

The snowflake curve starts with two parts, the initiator and the generator. The initiator is the initial set of edges forming a certain shape, whereas the generator is a set of edges that can be used to replace each edge of the initiator to form a new shape. That new shape then becomes the initiator for the next generation, where the generator again replaces each edge. The result is a complex shape similar to that of a snowflake. The initiator, generator concept, is a graphical representation of how rewriting systems operate; rather than the initiator and generator being a set of edges, a set of symbols or strings instead represents them.

1.4 Introduction to Formal Grammars

In the context of computer science, a grammar is defined as a set of rules governing which strings are valid or allowable in a language or text. They consist of syntax, morphology, and semantics. Formal languages have been defined in the form of grammars to suit particular problem domains. It is natural for humans to communicate a problem or solution in the form of language; it is intuitive to use a language to describe the desired outcome when dealing with the procedural generation of plant-life. In the past, formal grammars have been used extensively in computer science in the form of programming languages in which humans can provide a computer with a set of instructions to carry out to gain an expected result. The challenge is to procedural generation of plant-life by creating a grammar in the form of a rewriting system. A rewriting system such as the L-system operates in a way that is consistent with a context-free class of Chomsky grammar [Chomsky, 1956], similar to that of the programming language ALGOL-60 introduced by Backus and Naur in 1960[Backus et al., 1960]. In figure 1.2 below, two types of L-system grammars overlap the classes of Chomsky grammars, the OL-system, and the 1L-system. The details of these two systems will be discussed in detail chapter 2, but in summary, 0L-systems are grammars that can represent a context-sensitive Chomsky grammar but generally tend to be context-free, the main difference between the 0L-system and the 1L-system is that 1L-systems can be recursively enumerable. Furthermore, a 1L-system can represent any 0L-system, and 1L-system languages tend to be more complex and verbose when compared to 0L-systems. These two different types of L-systems each have trade-offs, 1L-systems are more powerful and complex, and 0L-systems are less powerful but make for a simpler language.

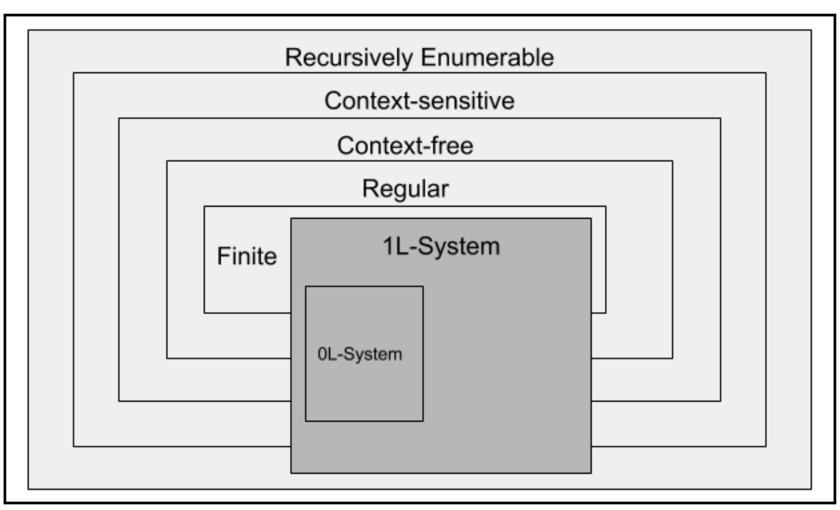


Figure 1.2: Diagram of the Chomsky hierarchy grammars with relation to the 0L and 1L systems generated by L-systems.

1.5 Structure of Thesis

This thesis begins by delving into the underlying concepts of L-systems. It makes sense to firstly start by defining the simplest type of L-system named the DOL-system. Then to provide details about how to interpret DOL-systems to produce graphical representations. The L-system chapter provides a formal definition for more complex types of L-systems, such as parametric L-systems. In conjunction, the L-system chapter talks about major features and improvements that aid the procedural generation of plant life. These include branching, conditionals, randomness, and stochastic rules.

The next major part of the thesis focuses on the L-system rewriter implementation. The string rewriter section includes the definition of the grammar and syntax for a parametric L-system. It also describes the process of string rewriting, and computationally understanding the L-system grammar using lexical analysis, parsing. The rewriter implementation describes a method of implementing the rewriting system and its connection to the string interpretation process.

The next chapter covers specific mathematics concepts necessary for working with 3D graphics. The chapter includes vectors, matrix transformations, and quaternions. The mathematics chapter is there to provide a brief overview of the mathematics concepts often used when rendering or animating 3D graphics.

Chapter 5 discusses the three main stages of L-system string interpretation with regards to the procedural generation of 3D plant-life. These three stages are the turtle graphics interpreter, model generator, and renderer. The turtle graphics interpreter goes into detail about the skeletal and joint structure of plants. The model generator talks about how to generate the vertex data for the 3D model of the plant using a skeletal structure, which can create a realistic-looking plant. Finally, the renderer covers the specifics of rendering models on the screen in the OpenGL framework.

The physics chapter which focuses on the physics behind the simulation of 3D generated plants. This chapter includes details of Hook's Law and the equations of motion within a 3D application.

Chapter 2

Lindenmayer Systems

An L-system at its core is a formal grammar. The term grammar refers to the structure or definition of a language. Grammars consist of syntax and semantics and allow the formalisation of a language. L-systems can be seen as a grammar for a language that can be used to describe the properties and structure of plant-life. The L-system grammar specifies an *alphabet* of characters which are concatenated together into collections of symbols, called strings. The L-system describes a starting string called an *axiom* and a set of production rules. The production rules decide whether or not another symbol or string should replace a symbol within the L-system string. This process of replacing symbols in a string depending on the production rules is called a rewriting step. The *axiom* is used in the first rewriting step. Each symbol within the axiom is matched to the production rules. If a match is found, the axioms symbol is replaced by the string described by that production rule. This process is carried out for each symbol in the *axiom* until the end of the string is reached. The resulting string created by the rewriting process then becomes the next string for rewriting, and the next rewritten step will begin. This process of rewriting using production rules is the mechanism for generating a structure of symbols that obey the production rules, similar to that of a context-free grammar. The symbols can represent plant-life because each symbol represents a particular state or feature of the plant-life. The resulting strings' symbols generated by the L-system can then be read by a different system called the interpreter. The interpreter understands the meaning of each symbol, and will use each symbol as an instruction to generate the plants structure in 3D space.

This chapter will go into detail about the L-system concept, the rewriting process and a simple interpreter. It will then discuss several different types of L-systems, and their features and limitations. This chapter focuses on the mechanics behind the rewriting system and different techniques that can be used to represent plant-life better. It will also provide sufficient background by briefly touching on how the resulting strings generated by the L-system can be interpreted. The interpretation of an L-system is a separate system to the L-system; however, it is essential to note that the L-system has no concept of what it is trying to represent, it is merely a string rewriting system. It is left up to the interpreter to carry out the L-systems' interpretation. The interpreter is responsible for interpreting the resulting string to create a suitable representation for that problem domain. For instance, the symbols for an L-system trying to represent a tree may be interpreted very different to the symbols trying to represent music; however, the L-systems may be identical. Although the interpreter is not necessarily part of the L-system, it is important to understand the reliance of the L-system on the string interpreter. The string interpreter will be explored in great detail in chapter 5.

The diagram below details the relationship between the L-system grammar and how it conforms to a number of different classes of grammars. In the L-system grammar the symbol "N" indicates the number of rewriting steps follow. "W" states that what follows is the axiom. Finally, "p1" and "p2" each indicate a production rule follows. It shows how the L-system is sent to the rewriter as input. There are three stages of rewritting, starting with the axiom. Each stage develops an increasingly complex string of symbols. The resulting string of symbols is then interpreted. In this example the symbol "A" draws a line and the symbol "B" draws a circle, resulting in an image that can be drawn on the screen.

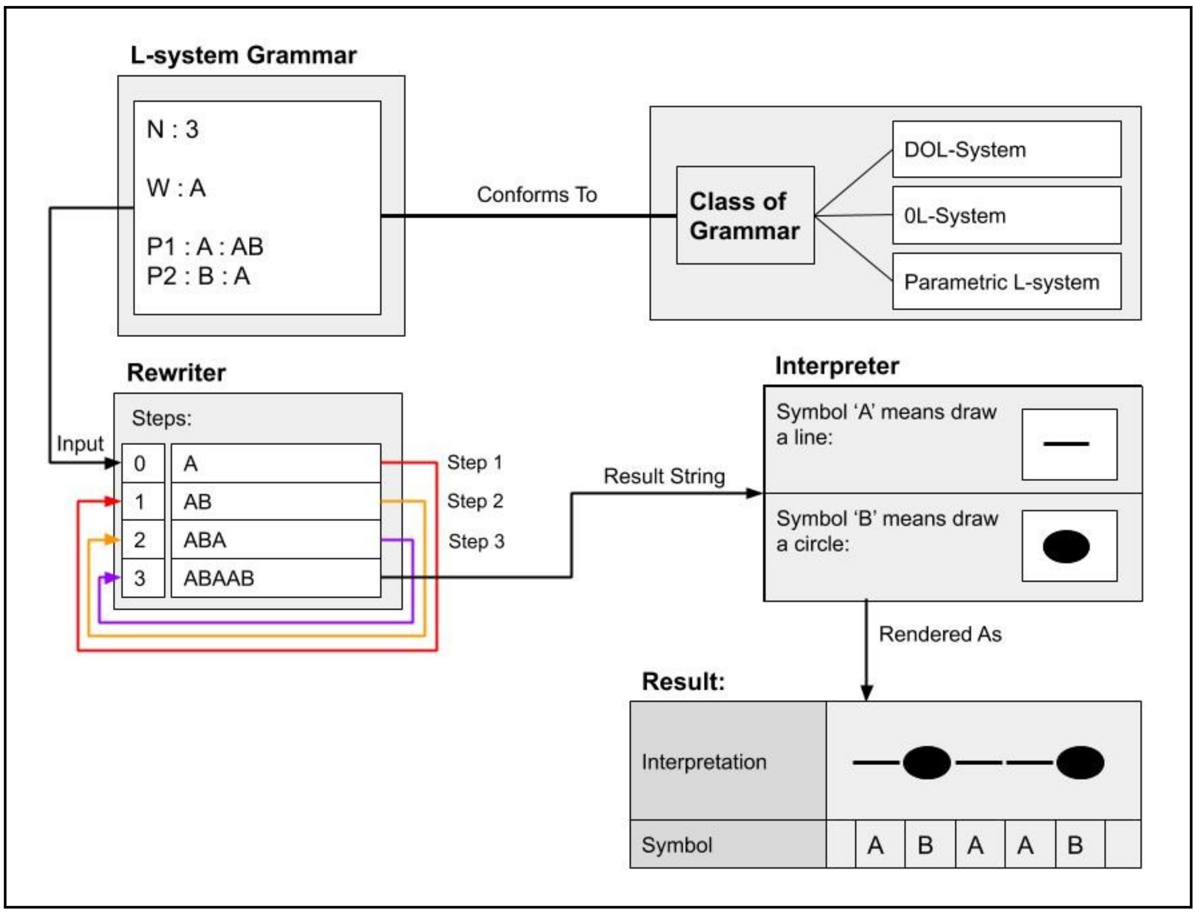


Figure 2.1: Diagram showing the relationships between the L-system grammar, language, rewriter, and interpreter.

A well-known biologist, Aristid Lindenmayer, started work on the Lindenmayer System or L-system in 1968, he sought to create a new method of simulating the growth in multicellular organisms such as algae and bacteria [Lindenmayer, 1968]. He later defined a formal grammar for simulating multicellular growth, which he called the 0L-system [Lindenmayer, 1971]. In the last twenty years, the concept has been adapted to be used to describe larger organisms, such as plants and trees, as well as other nonorganic structures like music [Worth and Stepney, 2005]. There have also been studies to use an L-system for creating and controlling the growth of a connectionist model to represent human perception and cognition [Vaario et al., 1991]. Similarly, Kókai et al. (1999) have created a method of using a parametric L-system to describe a human retina. This method can be combined with evolutionary operators and applied to patients with diabetes who are being monitored [Kókai et al., 1999].

2.1 Simple DOL-system

According to Prusinkiewicz and Hanan, the most simple type of L-system is known as the D0L-system. The term 'D0L system' abbreviates 'Deterministic Lindenmayer system with zero-sided interactions.' It is deterministic because each symbol has an associated production rule, and there is no randomness in determining the production rule. A zero-sided interaction refers to the multicellular representation of an L-system, where each symbol refers to a type of cell, each cell does not account for the state of its directly neighbouring cells, making it zero-sided. There are three major parts to a D0L system. Firstly there is a finite set of symbols known as the *alphabet*, a starting string or *axiom* and the state transition rules *production rules*. The alphabet is a set of characters that represent a state in a system. The axiom is the starting point of the system, which contains one or more characters from the alphabet. The transition rules dictate whether a state should remain the same, or transition into a different state, or even disappear completely. [Prusinkiewicz and Hanan, 2013].

The DOL-system serves as a context-free grammar, to represent the development of multicellular organisms. The DOL-system shown in 2.3 below is an example formulated by Prusinkiwicz and Lindenmayer to simulate Anabaena Catenula, which is a type of filamentous cyanobacteria which exists in plankton. According to Prusinkiewicz and Lindenmayer "Under a microscope, the filaments appear as a sequence of cylinders of various lengths, with a-type cells longer than b-type cells. The subscript l and r indicate cell polarity, specifying the positions in which daughter cells of type a and b are produced." [Prusinkiewicz and Lindenmayer, 2012].

$$
\begin{aligned}
\omega \quad &: a_r \\
p_1 \quad &: a_r \rightarrow a_l b_r \\
p_2 \quad &: a_l \rightarrow b_l a_r \\
p_3 \quad &: b_r \rightarrow a_r \\
p_4 \quad &: b_l \rightarrow a_l
\end{aligned}
\tag{2.1}
$$

With the definition above, the ":" symbol separates the axiom and production names from their values, furthermore, the $\rightarrow$ can be verbalised as "is replaced by" or "rewritten with". The DOL-system states that $w \; : \; a_r$, where the symbol w signifies that what follows is the axiom, therefore, the starting point is the cell a_r. The production rules then follow and are $p1, p2, p3$ and $p4$. In production rule 1 ($p1$) the cell a_r will be rewritten with cells $a_l b_r$. Production rule $p2$ states that a_l will be rewritten with cells $b_l a_r$. Production rule $p3$ states b_r will rewritten with cell a_r and finally production rule 4 ($p4$), states that b_l will be rewritten with cell a_l. To simulate Anabaena Catenula there are four rewriting rules required, due to the four types of state transitions. The resultant strings for five generations of the rewriting process can be seen in 2.2 below:

$$\begin{aligned}
G_0 &: a_r \\
G_1 &: a_l b_r \\
G_2 &: b_l a_r a_r \\
G_3 &: a_l a_l b_r a_l b_r \\
G_4 &: b_l a_r b_l a_r a_r b_l a_r a_r \\
G_5 &: a_l a_l b_r a_l a_l b_r a_l b_r a_l a_l b_r a_l b_r
\end{aligned} \tag{2.2}$$

During the rewriting process, generation zero (G_0) is the axiom. In subsequent generations, the resultant string of the previous generation is taken, and each symbol in the string is compared to the production rules. If they match the production rule, the symbol is rewritten with the successor symbol or string, which is specified by the production rule. For instance, the previous generation for G_1 is G_0, and the resultant string is for G_0 is a_r, the first symbol in this resultant string is compared with the production rules. In this case a_r matches rule $p1$ with the rule being $p1 \ : \ a_r \ \rightarrow \ a_l b_r$ and therefore, a_r is rewritten with $a_l b_r$. The resultant string of G_0 only has one symbol, so it can be concluded that the string of G_1 is $a_l b_r$, this string is stored for the next rewriting step and is later rewritten to produce generation two and so on, until the desired number of generations is reached.

The D0L-system is very simple and minimalist in design, which comes with some limitations. The D0L-system production rules merely state that if the symbol matches the production rule, then that symbol is rewritten. Often this is not the case; there may be some other conditions that may need to be checked before it can be concluded that a rewrite should take place. Furthermore, the symbols within a D0L-system does not supply much information. For instance, how does the D0L-system indicate how many times a given string has been rewritten? The D0L-system is also deterministic, meaning that there is no randomness in the rewriting process, and therefore, it always yields the same result with no variation. This can be seen as a limitation as variation within the system may be seen as a good thing, such as variation within the branching structure of plants.

2.2 Interpreting the DOL-system String

Section 2.1 outlined a simple type of L-system known as the DOL-system. This type of L-system specifies an alphabet, an axiom, and a set of production rules. This concept allows the representation of a problem as a set of states. The production rules can express valid state transitions, eventually produces a resulting string of symbols that obey the L-systems production rules.

This functionality is compelling; however, the L-system's symbols are only useful if they represent some meaning. Furthermore, the L-system does not supply this meaning; each symbol's meaning is interpreted after the rewriting process by the interpreter. Due to this, there are two separate systems involved in taking an L-systems input, such as the alphabet, axiom, and production rules, turning it into something that can model plant-life. These two systems are the L-system rewriter and the string interpreter. The L-system rewriter is responsible for using L-system to rewrite a string from its axiom by a certain number of generations, eventually providing a resulting string of symbols. The string interpreter takes the resulting string from the L-system rewriter and interprets it in a way that can represent the model we are trying to render.

A paper by Przemyslaw Prusinkiewicz outlines a method for interpreting the L-system in a way that can model fractal structures, plants, and trees. The method interprets the resultant string of the L-system. Where each symbol represents an instruction that is carried out one after the other to control a 'turtle' [Prusinkiewicz, 1986]. When talking about a turtle, Prusinkiewicz is referring to turtle graphics. Turtle graphics is a type of vector graphics that can be carried out with instructions. It is named a turtle after one of the main features of the Logo programming language. The simple set of turtle instructions listed below can be displayed as figure 2.3. The turtle starts at the base or root of the tree and interprets a set of rotation and translation movements. When all executed one after the other, they trace the points which make up the plants' structure. When these points are then joined together, the result is a fractal structure such as a plant or tree.

Instruction Symbol	Instruction Interpretation
F	Move forward by a specified distance whilst drawing a line
f	Move forward by a specified distance without drawing a line
+	Yaw to the right specified angle.
-	Yaw to the left by a specified angle.
/	Pitch up by specified angle.
\	Pitch down by a specified angle.
ˆ	Roll to the right specified angle.
&	Roll to the left by a specified angle.

Table 2.1: Table of turtle graphics instructions symbols and their meaning to the interpreter

In the OL-system, several symbols represent a particular meaning to the L-system interpreter. Whenever the interpreter comes across one of these symbols in the resultant string, it is interpreted as a particular turtle instruction, which can be seen in table 2.2.

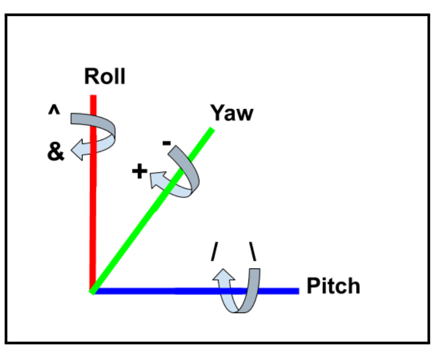


Figure 2.2: Diagram of the 3D rotations of the turtle.

The turtle instructions are presented in such a way that allows movement in three dimensions. The rotations are represented as yaw, pitch, and roll. Where yaw is a rotation around the Z-axis, the pitch is rotation around the X-axis, and roll is rotation around the Y-axis. There are two symbols for each rotation, which represent positive and negative rotations, respectively. Rotations are expected to be applied before a translation; that way, the rotations change the orientation of the turtle, and then the forward instructions move the turtle in the Y direction using the current orientation. The orientation is maintained from one translation to the next, and subsequent rotations are concatenated together to create a global orientation. In this way, when the turtle moves forward again, it moves in the direction of this global orientation.

Figure 2.2 shows the yaw, pitch and roll rotations as well as their axis and the instruction symbols for the L-system.

The turtle instructions in the table 2.1, can be used as the alphabet for the rewriting system defined in the L-system grammar below:

$$
\begin{aligned}
&\text{Generations: } 1 \\
&\text{Angle: } 90^\circ \\
&\omega \;\; : F \\
&p_1 \;\; : F \;\rightarrow\; F + F - F - F + F
\end{aligned}
\tag{2.3}
$$

This L-system makes use of the alphabet "F, +, -". The meaning of these symbols is not relevant to the rewriting system. The main piece of information that is relevant to the interpreter is the angle to rotate by when it comes across the symbols + and -. This value is specified in the definition of the L-system with the Angle: 90° statement. The resulting string would be "F+F-F-F+F"; this string is passed to the interpreter system, which uses turtle graphics to execute the list of instructions. These instructions can be articulated in table 2.2 below.

Instruction Number	Instruction Symbol	Instruction Interpretation
I1	F	Move forward by 1
I2	+	Yaw right by 90 degrees
I3	F	Move forward by 1
I4	-	Yaw left by 90 degrees
I5	F	Move forward by 1
I6	-	Yaw left by 90 degrees
I7	F	Move forward by 1
I8	+	Yaw right by 90 degrees
I9	F	Move forward by 1

Table 2.2: Table showing each instruction symbols and their interpretation for the L-system 2.3

These instructions are carried out one after the other, moving the turtle around the screen in three dimensions. Tracing the structure which the 0L-system has generated, these instructions generate the traced line shown in figure 2.3 below.

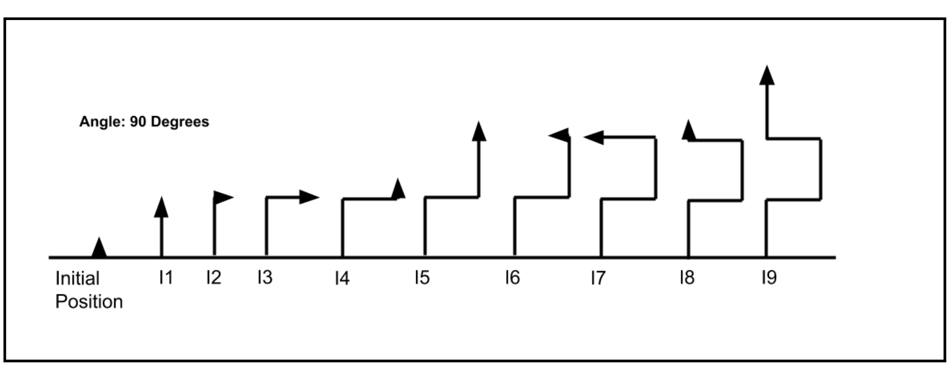


Figure 2.3: Diagram showing a turtle interpreting simple L-system string.

As we can see from the turtle interpretation above, the turtle moves around as if it is an entity within a 3D world following a set of instructions that tell it where to move. This is the basic concept of turtle graphics and how it is implemented in the interpreter system. What also becomes apparent is that there are several assumptions which the interpreter makes to produce the final image in I9. It is assumed that that the + and - symbols mean a change in yaw of 90 degrees, and the second assumption is that the F symbol means to move forward by a distance of 1 unit measurement. The angle and distance values are assumed because the resultant string does not explicitly define the angle or the distance; it leaves that up to the interpretation of the string.

In a simple DOL-system like the one above, there is no explicit way of providing this additional information to the interpreter. This means that it must be hardcoded into the interpretation or assumed by some other means. This highlights one of the primary considerations when creating an L-system. There is a difference in complexities between the L-system rewriter and the interpreter. It is possible to create a very complex rewriting system with extensive rule systems, which can supply a large amount of information to the interpreter. The interpreter, on the other hand, can be rudimentary and follow the instructions exactly. Conversely, we could have a system where the L-system rewriter is quite simple, but the interpreter is very complicated. The interpreter must be capable of representing the L-system, despite the lack of information in the resultant string. Alternatively, it should be able to obtain this information by other means.

It may be tempting to leave the complexity to the interpreter to make the L-system rewriter and its rules more simple. However, the drawback of this is that the information needed for modeling branch diameters, branching angles, and the type of objects that need to be rendered have to be supplied to the interpreter in some way. If not through the resulting string of information, how is this information meant to be provided to the interpreter? An answer may be to build a system within the interpreter that is capable of assuming the general look of a plant, for instance, branches that decrement in diameter and branching angles, which are consistent. This could result in a very inflexible system that may work for a portion of plant-life but might struggle to represent certain classes of plant-life. Therefore, the benefit of using a system with most of its complexity within the rewriting system is the L-system is responsible for some of the details of the interpretation, such as angles, branch diameters, and other details. In the next few sections, different types of L-systems are described, explaining their benefits and limitations, as well as developing a system integrating these separate systems into a single L-system grammar.

Several well-known fractal geometry patterns have been explored. They are particularly interesting because of how they seemingly imitate nature [Mandelbrot, 1982]. An example of this is the is with edge-rewriting patterns like the Koch curve and the Sierpiński gasket. The Koch curve can be represented using the L-system defined in 2.4 below. This is an adaption of the Koch snowflake, which can be generated by the 0L-system. It is important to note that as the number of rewrite generations increases, the complexity of the patterns becomes increasingly intricate.

Koch Curves:
Generations: 2,3,4
Angle: 90°
Distance: 1 cm
ω : F
$p1$: F $\rightarrow$ F+F-F-F+F

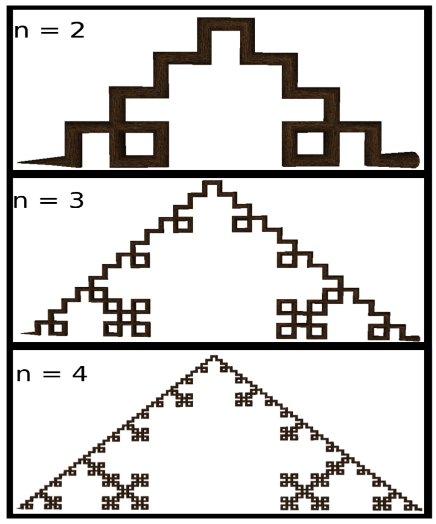


Figure 2.4: Koch Curve.

The Sierpińkski gasket is another example of an edge-rewriting pattern which can show the power of a rewriting system like the L-system. This example is interesting as with each generation, the even-numbered generations face left, and the odd-numbered generations face right.

Sierpiński Gasket:
Generations: 2,3,4,5
Angle: 60°
Distance: 1 cm
ω : F
$p1$: F $\rightarrow$ X-F-X
$p2$: X $\rightarrow$ F+X+F

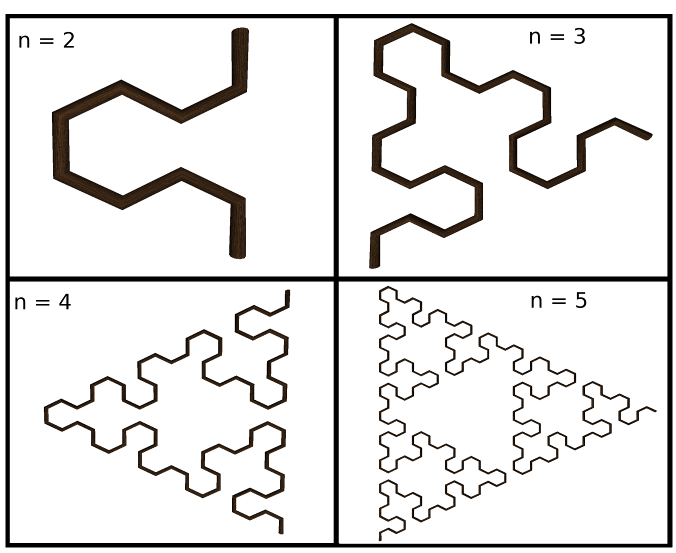


Figure 2.5: Sierpiński Triangles.

2.3 Branching

The simplistic D0L-system defined in previous sections can trace a 3D pattern. The D0L-systems interpretation provides a way of tracing a path or structure in 3D space. These types of L-systems are useful; however, to trace the branching structure of plants, there needs to be a way of branching off in one or more directions. A simple solution may be for the turtle object to trace its steps back to a particular branching point and then branch off in a different direction. Branching like this may get the desired result but is slow and inefficient.

Lindenmayer proposed a better solution to the branching problem. He introduced two symbols that have special meanings within the alphabet of the DOL-system, which make branching much easier [Lindenmayer, 1968]. These are the square bracket symbols "[", "]". The open square bracket "[" symbol instructs the turtle object to save its current state (position and orientation) and be able to go back to that saved state later. The close square bracket "]" instructs the turtle to load the saved state and continue from that position and orientation. The save and load states allow the turtle to jump back to a previously saved position, facing in the same direction as it was before. The orientation can later be changed, allowing the turtle to branch off in a different direction. This method was originally used by Lindenmayer to imitate the branching that occurs within algae but was later adapted by Smith to represent larger plant-life as well [Smith, 1984].

The main advantage of using the save and load position functionality within the alphabet is that the rewriting system itself handles branching. The production rules often contain the next generations branching structure by using the save and load symbols, and thus the branching structure becomes more intricate from one generation to the next.

Each save state symbol must have a corresponding load state symbol within the string. This is not a requirement by the L-system language, but a requirement during interpretation because the load and save state symbols have no special meaning to the rewriter. It is treated the same as any other symbol in the alphabet. This being said, during interpretation, for the turtle object to jump back to a saved state, those save and load states should correspond. For instance, the resultant string "F[+F-F]-F" has both a load, and a save state, meaning there is a single branch off the main branch. An example of this can be seen in figure 2.6 below. Additionally, using nested save and load states in the string, for instance, "F[+F[+F]-F]-F", there can be two branches off the main branch twice as seen in figure 2.7.

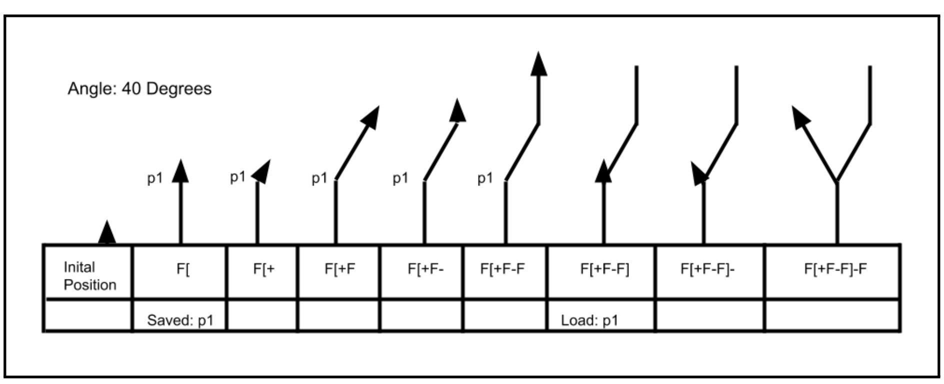


Figure 2.6: Diagram showing a turtle interpreting an L-system using the branching symbols.

Save and load operations are handled using the Last In First Out (LIFO) principle. LIFO states that when using the save symbol, it saves the current position and orientation at $p1$. The next load state restores $p1$'s position and orientation. Unless there is another save that takes place before the load state, in which case the most recent save has to be loaded before $p1$ can be loaded. In this way, the position saves are stacked, and the most recent save is always loaded first. An example of this can be seen in figure 2.7 below:

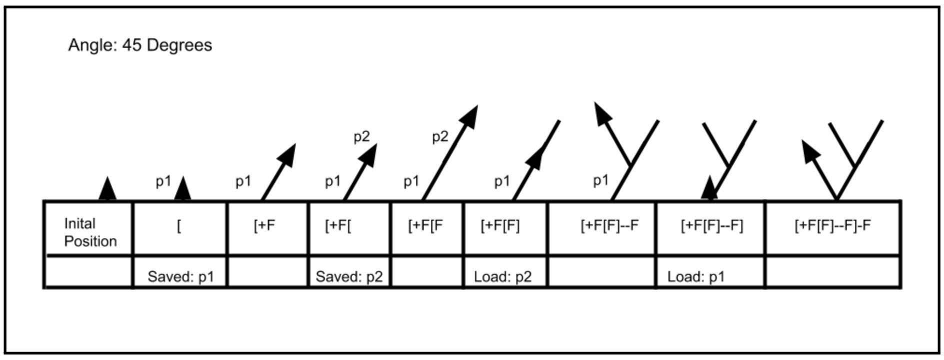


Figure 2.7: Diagram showing a turtle interpreting an L-system with nested branching.

The save and load state symbols can be used within a simple L-systems to create a more complex plant-like fractal pattern. In the following examples, there are two L-systems. One can generate a fractal pattern similar to that of a bush, and the other a fractal representing a tree. In figure 2.8, the F symbol can be rendered as a branch segment. The L-system only consists of a single rewriting rule; thus, each generation results in exponentially more branches. Each generation results in eight times more branches than the previous generation.

Fractal Bush:
Alphabet: F, +, -, [,]
Axiom: F
Angle: 25°
Rules:
F → FF+[+F-F-F]-[-F+F+F]

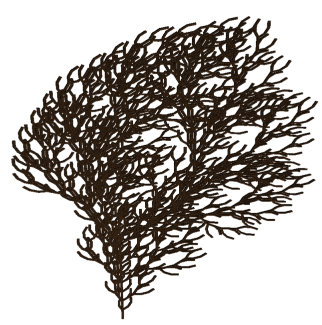


Figure 2.8: Fifth generation of the fractal bush L-system.

In figure 2.9 below, there are two different rewriting rules. One for the symbol F and the other for symbol X. Symbol X is the axiom; however, it is not a rendered symbol meaning the interpreter ignores it. Unlike the symbol F, which is rendered as a branch. Instead, symbol X stands as a placeholder for the next rewriting step, where it is rewritten with "F-[[X]+X]+F[+FX]-X". The symbol F is replaced by FF, this means that existing branches get longer each generation, but new branching structures are created at the end "leaves" or ends of the branches due to the production rule for symbol X.

<u>Fractal tree:</u>
Alphabet: X, F, +, -, [,]
Axiom: X
Angle: 25°
Rules:
X → F-[[X]+X]+F[+FX]-X
F → FF

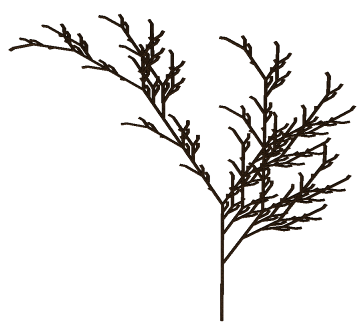


Figure 2.9: Fifth generation of the fractal tree L-system

2.4 Parametric OL-systems

Simplistic L-systems, like the algae representation in section 2.1, give enough information to create the fundamental structure of plant life. Many details necessary for rendering the plant are not included with a simple OL-system. Things like the width, length, and branching angles of each section. These details have to be assumed or are defined somewhere as a constant value. The interpreter is left to find the details of the branching structure. The question becomes, is there a type of L-system that is capable of providing these details? The answer lies with parametric 0L-systems.

This section will outline the definition and significant concepts of the parametric L-system formulated by Prusinkiewicz and Hanan in 1990 [Prusinkiewicz and Hanan, 1990], and developed upon in 2012 by Prusinkiewicz and Lindenmayer [Prusinkiewicz and Lindenmayer, 2012]. This section talks about the changes and improvements to the parametric L-system. As well as explains why these changes are necessary for this thesis.

2.4.1 Formal Definition of a Parametric 0L-system

Prusinkiewicz and Hanan define the parametric 0L-systems as a system of parametric words, where a string of letters make up a module name A, each module can have several parameters associated with it. The module names belong to an alphabet V; therefore, $A \in V$, and the parameters belong to a set of real numbers $\Re$. If $(a_1,\ a_2,\ ...,\ a_n) \in R$ are parameters of A, the module can be stated as $A(a_1,\ a_2,\ ...,\ a_n)$. Each module is an element of the set of modules $M = V \times \Re^*$. $\Re^*$ represents the set of all finite sequences of parameters, including the case where there are no parameters. We can then infer that $M^* = (V \times \Re^*)^*$ where M^* is the set of all finite modules.

Each parameter of a given module corresponds to a formal definition of that parameter defined within the L-system productions. Let the formal definition of a parameter be Σ. $E(\Sigma)$ can be said to be an arithmetic expression of a given parameter.
Similar to the arithmetic expressions in the programming languages C/C++, we can make use of the arithmetic operators $+,\ -,\ *,\quad \wedge$. Furthermore, we can have a relational expression $C(\Sigma)$, with a set of relational operators. In the literature by Prusinkiewicz and Hanan the set

of relational operators is said to be $<, \ >, \ =$, I have extended this to include the relational operators $>, \ <, \ >=, \ <=, \ ==, \ !=$. Where $==$ is the 'equal to' operator, $!=$ is the 'not equal' operator, the symbols $>=$ and $<=$ are 'greater than or equal to' and 'less than or equal to' respectively. The parentheses () specify precedence within an expression. A set of arithmetic expressions can be said to be $\hat{E}(\Sigma)$, these arithmetic expressions can be evaluated and result in the real number parameter $\Re$, and the relational expressions can be evaluated to either true or false.

The parametric 0L-system can be shown as follows as per Prusinkiewicz and Hanan's definition:

$$G = (V, \Sigma, \omega, P) \tag{2.4}$$

G is an ordered quadruplet that describes the parametric OL-system. V is the alphabet of characters for the system. Σ is the set of formal parameters for the system. $\omega \in (V \times \Re^*)^+$ is a non-empty parametric word called the axiom. Finally, P is a finite set of production rules which can be fully defined as:

$$P \subset (V \times \Sigma^*) \times C(\Sigma) \times (V \times \hat{E}(\Sigma))^* \tag{2.5}$$

Where $(V \times \Sigma^*)$ is the predecessor module, $C(\Sigma)$ is the condition and $(V \times E(\Sigma))^*$ is the set of successor modules. For the sake of readability we can write out a production rule as $predecessor : condition \rightarrow successor$. I will be explaining the use of conditions in production rules in more detail in section 2.4.4. A module is said to match a production rule predecessor if they meet the three criteria below.

- The name of the axiom module matches the name of the production predecessor.
- The number of parameters for the axiom module is the same as the number of parameters for the production predecessor.
- The condition of the production evaluates to true. If there is no condition, then the result is true by default.

In the case where the module does not match any of the production rule predecessors, the module is left unchanged, effectively rewriting itself.

2.4.2 Defining Constants and Objects

There are some other features covered by Prusinkiewicz and Lindenmayer that are not specific to the parametric L-systems definition itself but serve as quality of life. In the literature, they refer to the #define, which is said: "To assign values to numerical constants used in the L-system." The #include statement specifies what type of shape to draw by referring to a library of predefined shapes [Prusinkiewicz and Lindenmayer, 2012]. For instance, if we have a value for an angle that we would like to use within the production rules, we can use the #define statement as follows:

$$\begin{aligned}
&n = 4\\
&\text{\#define angle 90}\\
&\omega \quad : F(5)\\
&p_1 \quad : F(x) \qquad : * \rightarrow \; F(w) + (angle)F(w) + (angle)F(w) + (angle)F(w)
\end{aligned} \tag{2.6}$$

Here you can see that the #define acts like a declaration, where a variable is going to be defined, which is used later. Essentially we are replacing any occurrences of the variable *angle* with the value of 90 degrees. The define statement is written as #define *variable_name value.*

With regards to the #include statement, In the literature, the #include may be used by stating "#include H". This tells the turtle interpreter that the symbol "H" is a shape in a library of predefined shapes which should be rendered instead of the default shape. This functionality has been slightly modified, instead of the #include statement, the #object is used and serves a similar purpose, however, instead importing the symbol "H", denoting to the heterocyst object from a library of predefined shapes, The statement "#object H HETEROCYST" specifies that we are associating the symbol or module "H" with the object HETEROCYST. The HETEROCYST object is still stored in a predefined library; however, the advantage is that the object can be associated with multiple different symbols, it also does not limit us to a predefined name for an object. Below is an example using the #object statement:

$$\begin{aligned}
&n = 1\\
&\text{\#object F BRANCH}\\
&\text{\#object S SPHERE}\\
&\omega \quad : F(1)\\
&p_1 \quad : F(x) \qquad : * \rightarrow \; F(w)F(w)F(w)F(w)S(w)
\end{aligned} \tag{2.7}$$

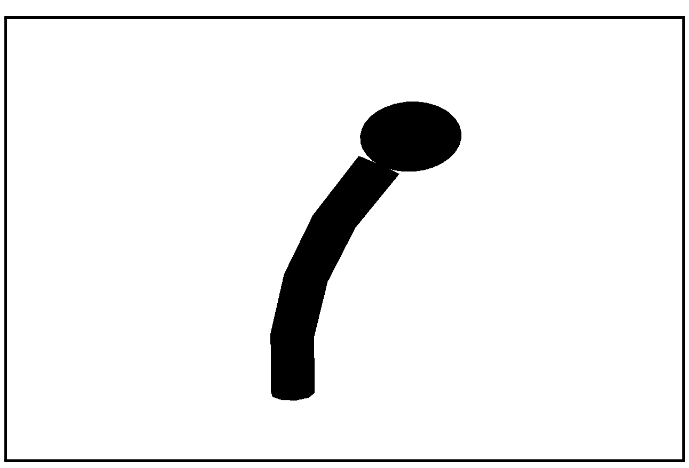


Figure 2.10: Diagram of an L-system Using Multiple Objects.

In the simple example in figure 2.7 above, you can see that the first three F modules render a branch segment with a length of 1.0; however, for the final S module renders a sphere of diameter 1.0. The geometric shape that is eventually rendered does not affect the L-system in any way, and the #object feature bears no meaning to the rewriting system, it merely stands as an instruction to the interpreter which instructs that each time the symbols F or S are interpreted, a specific object should be rendered, such as BRANCH and SPHERE respectively. The position of the next object or branch can then be determined by moving forward by the diameter of the object and rendering the next object from that point. The details of the interpreter are discussed in more detail chapter 5.

2.4.3 Modules With Special Meanings

In the above section, I defined the details of a parametric 0L-system. In the paper by Prusinkiewicz and Lindenmayer, there are two operators which have not been discussed yet. These operators are the ! and the ‘. Prusinkiewicz and Lindenmayer state that “The symbols ! and ‘ are used to decrement the diameter of segments and increment the current index to the color table respectively” [Prusinkiewicz and Lindenmayer, 2012]. We have decided to modify this to work slightly differently, the ! and ‘ still performs the same operation; however, the ! and ‘ symbols are treated as a module that holds particular meaning to the interpreter, rather than a single operator. Furthermore, they share the same properties with modules; they can contain multiple parameters, and depending on the number of parameters, they can be treated differently. The module ! with no parameters could mean decrement the diameter of the segment by a default amount, whereas !(10) means set the diameter of the segment to 10. The length can also be manipulated similarly. The module with the name F has a default meaning to create a segment in the current direction by a default amount. If we provide the module F(10) we are specifying to create a segment of length 10.

Using the L-system below, we can create figure 2.8, the concepts discussed above have been used by decrementing the segment diameter during the rewriting process as well as by incrementing the branch length.

$$
\begin{aligned}
&n = 8 \\
&\omega \quad : A(5) \\
&p_1 \quad : A(w) \qquad : * \rightarrow \; F(1)!(w)[+A(w \; * \; 0.707)][-A(w \; * \; 0.707)] \\
&p_2 \quad : F(s) \qquad : * \rightarrow \; F(s \; * \; 1.456)
\end{aligned}
\tag{2.8}
$$

The above l-system gives the resulting representation shown below in figure 3.8.

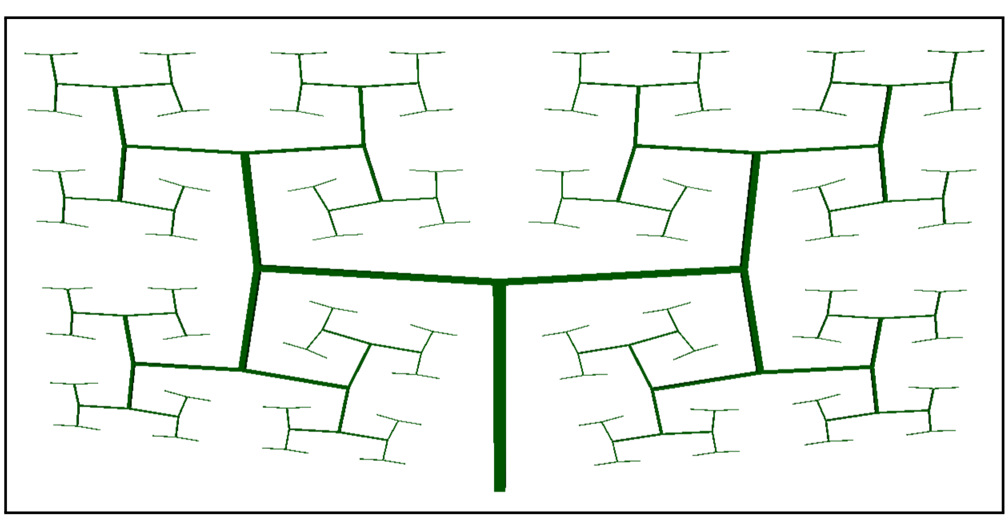


Figure 2.11: 3D Parametric L-system.

This gives a much more realistic looking tree structure as the branch segments become shorter but also become thinner in diameter as they get closer to the end of the branch as a whole.

2.4.4 Representing L-system Conditions

A condition allows multiple production rules that are the same in terms of their module name and number of parameters. Furthermore, they require a particular condition to be met for the module to match that rule.

This section will detail the use of the condition statement, which lies between the predecessor and the successor in a production rule. It can be seen as a mathematical expression on either side of a relational operator. During the rule selection process, the expressions are evaluated, and the results are compared using the condition operator. If the result of the condition evaluates as true, then that rule is selected for rewriting; otherwise, it will check the next rule.

Below is an example of a parametric 0L-system using condition statements:

$$\begin{aligned}
&n = 5 \\
&\omega \;\; : A(0)B(0,4) \\
&p_1 \; : A(x) \qquad : x \; > \; 2 \; \rightarrow \; C \\
&p_2 \; : A(x) \qquad : x \; < \; 2 \; \rightarrow \; A(x \; + \; 1) \\
&p_3 \; : B(x,\; y) \; : \; x \; > \; y \; \rightarrow \; D \\
&p_4 \; : B(x,\; y) \; : \; x \; < \; y \; \rightarrow \; B(x \; + \; 1,\; y)
\end{aligned} \tag{2.9}$$

The L-system above in 2.9 is rewritten five times using the axiom specified by the symbol ω, as well as the four production rules p_1, p_2, p_3, p_4. Each generation of the rewriting process can be seen below in 2.10.

$$\begin{aligned}
g_0 &: \; A(0)B(0,\; 4) \\
g_1 &: \; A(1)B(1,\; 4) \\
g_2 &: \; A(2)B(2,\; 4) \\
g_3 &: \; C \; B(3,\; 4) \\
g_4 &: \; C \; B(4,\; 4) \\
g_5 &: \; C \; D
\end{aligned} \tag{2.10}$$

The practical use of the condition statement might be to simulate different stages of growth. The condition statement is best illustrated using the L-system below:

$n = 2,\ 4,\ 6$

#object F BRANCH

#object L LEAF

#object S SPHERE

#define r 45

#define len 0.5

#define lean 5.0 (2.11)

#define flowerW 1.0

$\omega \ : \ !(0.1)I(5)$

$p_1 \ : I(x) \ : \ x \ > \ 0 \ \rightarrow \ F(len) - (lean)[R(0,100)]F(len)[R(0,100)]I(x-1)$

$p_2 \ : R(x) \ : \ x \ > \ 50 \ \rightarrow \ -(r)/(20)!(2.0)L(2)!(0.1)$

$p_3 \ : R(x) \ : \ x \ < \ 50 \ \rightarrow \ -(r)\backslash(170)!(2.0)L(2)!(0.1)$

$p_4 \ : I(x) \ : \ x \ <= \ 0 \ \rightarrow \ F(len)!(flowerW)S(0.3)$

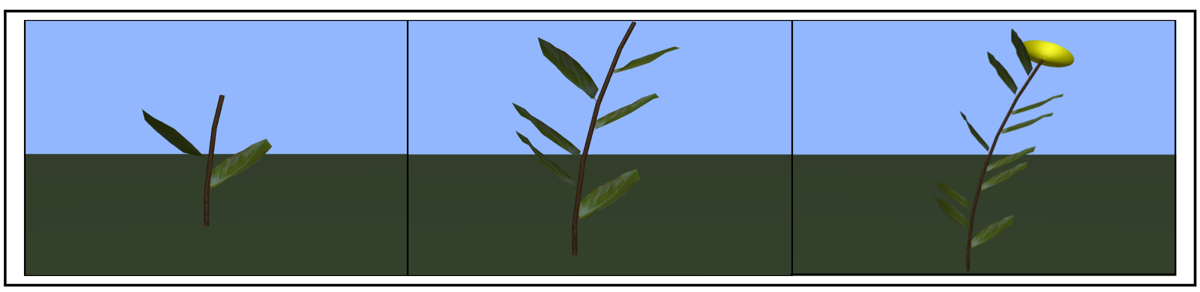


Figure 2.12: Condition statements used to simulate the growth of a flower. 2nd generation on the left, 4th generation in the center and 6th generation on the right

2.5 Randomness within L-systems

Randomness is an essential part of nature. If there is no randomness in plant life, it will end up with very symmetric and unrealistic. Randomness is also responsible for creating variation in the same L-system. An L-system essentially describes the structure and species of a plant. It describes how large the trunk of the tree is, how many leaves are on the end of a branch, or even if it has flowers or not. However, if there is no capability to have randomness in the generation of the L-system, then it will always end up with the same structure. Below is a simple example of how randomness can be used to create variation.

$n = 2$

#define r 25

$\omega \ : \ !(0.2)F(1.0)$

$p_1 \ : F(x) \ : \ * \ \rightarrow \ F(x)[+(r)F(x)][-(r)F(x)] + (\{-20,20\})F(x) - (\{-20,20\})F(x)$ (2.12)

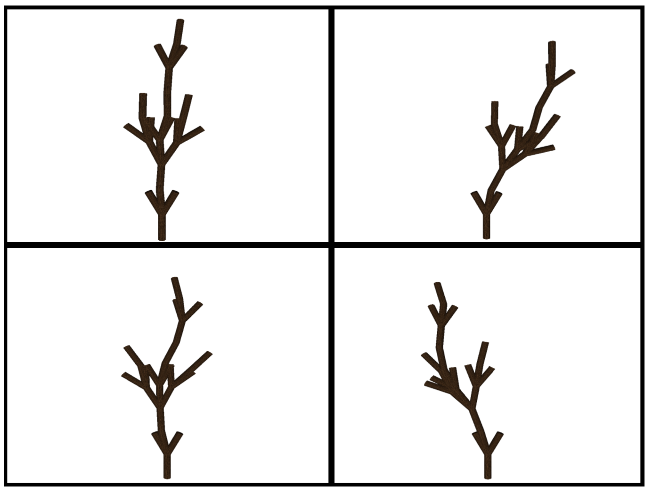


Figure 2.13: Different Variations of the Same L-system with Randomness Introduced in The Angles.

In figure 2.13, there are four variations of the same L-system using randomness. We can specify that we would like to create a random number by using the expression {-20.0, 20.0}. The curly braces signify that a random number range contains a number ranging from the minimum value, being the first floating-point value and the maximum value, being the second floating-point value, separated by a comma. If both values are the same for instance +({10.0, 10.0}) this is equivalent to +(10.0).

2.6 Stochastic Rules within L-systems

Similar to the previous section, stochastic L-systems fulfill a similar goal. On their own, 0L-systems are incapable of creating any variation. They follow a strict set of production rules that give the same result. Introducing randomness to an 0L-system for the width, length, and other parameters can result in a plant that looks slightly different but does not change to the overall structure of the plant. To create a different structure for a plant, we must introduce stochastic probability within the selection of production rules, thus effecting the rewriting of the plant's structure.

Eichhorst and Savitch introduced a new type of 0L-system called the S0L-system, this added two features to the existing 0L-system, firstly the S0L-system is not limited to defining a single axiom (starting point), a finite number of starting points can be defined, and a probability distribution is used to select the starting point at the start of the rewriting process. Secondly, the S0L-system allows the definition of a finite number of production rules which have a probability distribution to decide which rule should be chosen for rewriting [Eichhorst and Savitch, 1980]. Similarly, an article by Yokomori proposes a stochastic 0L-system which also proposes a measure of the entropy of a string generated by a 0L-system [Yokomori, 1980].

Later, Prusinkiewicz and Lindenmayer built upon this by creating a definition of a stochastic L-system, that makes use of the stochastic nature of the production rules from the SOL-system. This paper will be using the definition of the stochastic 0L-system defined by Prusinkiewicz and Lindenmayer and developing them into the existing parametric 0L-system. This paper will not allow multiple starting points as defined by Eichhorst and Savitch in the SOL-system, as it does not seem necessary and could overcomplicate the 0L-system. However, this functionality could be added in the future if it is seen to be necessary.

Similarly to the 0L-system, the stochastic 0L-system is an ordered quadruplet, represented as $G_\pi = (V, \omega, P, \pi)$, where V is the alphabet of the 0L-system, ω is the axiom, P is the finite set of productions, and π represents a probability distribution for a set of production probabilities this can be shown as $\pi : P \rightarrow (0, 1)$ the production probabilities must be between 0 and 1 and the sum of all production probabilities must add up to 1.

The following L-system definition created by Prusinkiewicz and Lindenmayer states three production rules with each rule having a probability of 0.33 out of one. For a finite set of production rules to be stochastic, the production rules must share the same module name and the same number of parameters. There must be two or more production rules, and the total probability distribution must add up to 1.0 [Prusinkiewicz and Lindenmayer, 2012].

$$
\begin{aligned}
&n = 5 \\
&\#\text{define r } 25 \\
&\omega \ : F(1) \\
&p_1 \ : F(x) \ : \ \sim 0.33 \ \rightarrow \ F(x)[+(r)F(x)]F(x)[-(r)F(x)]F(x) \\
&p_2 \ : F(x) \ : \ \sim 0.33 \ \rightarrow \ F(x)[+(r)F(x)]F(x) \\
&p_3 \ : F(x) \ : \ \sim 0.34 \ \rightarrow \ F(x)[-(r)F(x)]F(x)
\end{aligned}
\tag{2.13}
$$

As seen above, the module F(x) is the predecessor for all three of the production rules, each rule has a probability which is defined using the $\sim$ symbol followed by a probability from 0 to 1. In the above example, each probability is approximately one third, and they are approximate to total an exact probability of 1.0. During the rewriting process, when module F with one parameter is found, a production rule is randomly selected using the probability distribution described within the production rules. The predecessor from the selected rule will then rewrite that module.

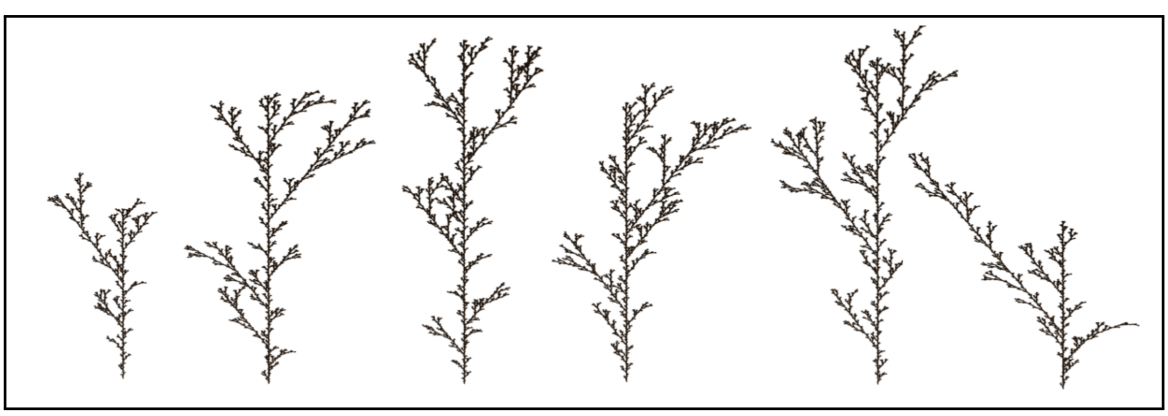


Figure 2.14: Representation of an L-system with a probability stochastic with a 0.33 probability for each rule.

The stochastic L-system definition in 2.13, produces the following fractal structures seen in figure 2.14 below. The stochastic L-system will get a slightly different resultant string each time it is run, depending on which rules were selected for rewriting. The difference in resulting strings gives a different number of translation instructions, resulting in the plant having branches of different lengths. $p1$ has two extra F instructions, this results in some branches being much longer than others, and possibly producing plants of different sizes.

2.7 Computing L-systems

This thesis focuses on the different levels of complexity between the L-system rewriting and the L-system interpretation. It is essential to distinguish these two systems by their components, and how these components interact. The two systems will be called the L-system rewriter and the L-system interpreter. As discussed at the begining of this chapter, the L-system rewriter takes L-system language as input in the form of a text file. The rewriter has three significant parts the tokenizer, parser, and the rewriter. The tokenizer breaks the language into individual words, then checks the syntax of the language according to the grammar. The parser then uses these words to check the validity of the semantic structure of the language as well as build relevant data structures for the rewriter. Finally, the rewriter uses these data structures to rewrite the axiom string several times according to the production rules. The result of the string rewriter is a module string, as well as other bits of information that will be used by the interpreter. The interpreter must now assume any information that has not provided by the string rewriter.

The string interpreter also has three significant parts; however, the functions of these parts are very dependant on what the L-system is trying to represent. For the procedural generation of plant life, there is the turtle graphics interpreter, model generator, and the OpenGL renderer. The turtle graphics renderer takes each module from the module string and interprets its meaning as a set of instructions carried out by a turtle object. The model generator takes the information generated by the turtle graphics interpreter and generates the 3D branching model as well as leaves and other objects. Lastly, the OpenGL renderer takes the models generated and renders them on the screen for the user. This process can be visualised in figure 2.7 below.

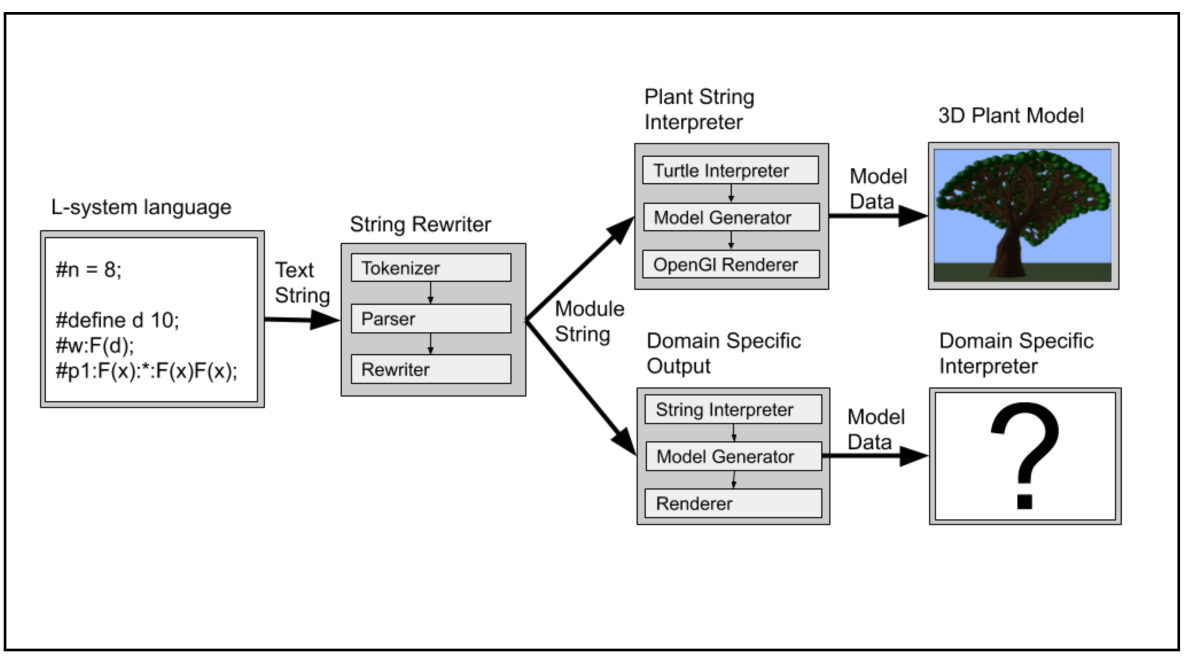


Figure 2.15: Diagram of the procedural generation process.

2.8 Summary

L-systems represent a set of state transitions based upon the production rules provided. These rules dictate how a string will be rewritten, which in turn determines the overall structure of the plant it is trying to represent. The symbols in D0L-systems or modules in parametric 0L-systems represent particular instructions to be carried out by turtle graphics within the interpreter. The modules within an L-system do not change the behaviour of rewriting but instead matter to the interpreter. Additionally, the complexity of the L-system rewriter decides the complexity of the interpreter. If an L-system provides a large amount of information to the interpreter, fewer assumptions need to be made during the interpretation and, therefore, providing the ability to describe the plant-life it is representing accurately.

By using the parametric 0L-system, we can build in several features, otherwise used in other L-systems, such as branching, conditional production rules, randomness in parameters, stochasticity. These features allow the parametric 0L-system to represent plant-life with varying structures, branch lengths, branch widths, and production rule conditions, which gives further control over stages of growth.

Chapter 3

L-system Rewriter Implementation

There are two major parts necessary to procedurally generate plant-life using an L-system. These are the rewriter and the interpreter. The purpose of the L-system rewriter is to take an L-system file as input, and generate the resulting string that fits the L-system grammar. It does this by syntactically and semantically analysing the L-system input, and generating the structures and information necessary to carry out the rewriting process. The rewriting process uses the structures and information, such as the string of modules and the production rules, to step through each string and rewrite the symbols. This chapter focuses on each part of the string rewriters' implementation and will introduce a technique of processing the L-systems' input, similar to how computer languages are compiled. This chapter will also formally define the L-system grammar in Backus-Naur Form, and provide the pseudocode for the L-system rewriter.

For a simple D0L-system, like the one seen in section 2.3. Each symbol within the alphabet is made up of a single character, the productions rules then match against those characters. As the D0L-system is deterministic, there is no randomness when determining the matching rule. The simplicity of the L-system makes it quite easy to create a rewriting system for the D0L-system. All the rewriter must do is store the starting string and production rule predecessors and successors. It then iterates over a string of symbols and replace them with the successor. The implementation of a more sophisticated L-system, like the parametric 0L-system, is much more complex. A parametric L-system can have multiple modules that make up a string, where each module may have multiple parameters, and each parameter could be a mathematical expression. The added complexity makes developing a rewriting system considerably more difficult. The rewriter must better understand what the syntax of the L-system is specifying, based on the context of each symbol within the L-system.

Due to the complexity of the L-system grammar, it is difficult for a computer to tell the syntactic and semantic properties of each part of the L-system input, which makes it difficult to carry out the rewriting process. Using a system similar to a "compiler", an L-system "program" can be broken down into a three-stage process, as seen in figure 3.3 below. The first stage is *lexical analysis*, then a process called *parsing* and finally the string rewriting stage. The lexical analyser is responsible for splitting the input into syntactic words, and then assigning each word into its syntactic category. Any word within the L-system that does match a syntactic category will result in a lexical error. If there are no lexical errors the words and their syntactic categories are sent to the parser. The parser matches the syntactical categories of each sentence in the language against a grammatical model. If any of the sentences within the language do not match the grammatical model, an appropriate error message can be displayed, similar to that of the lexical error. The error states where the syntax error occurred and what was grammatically incorrect. The parser also creates a syntax tree along with any data structures necessary for the rewriting process. These structures can then be used to carry out string rewriting or provide information to the interpreter.

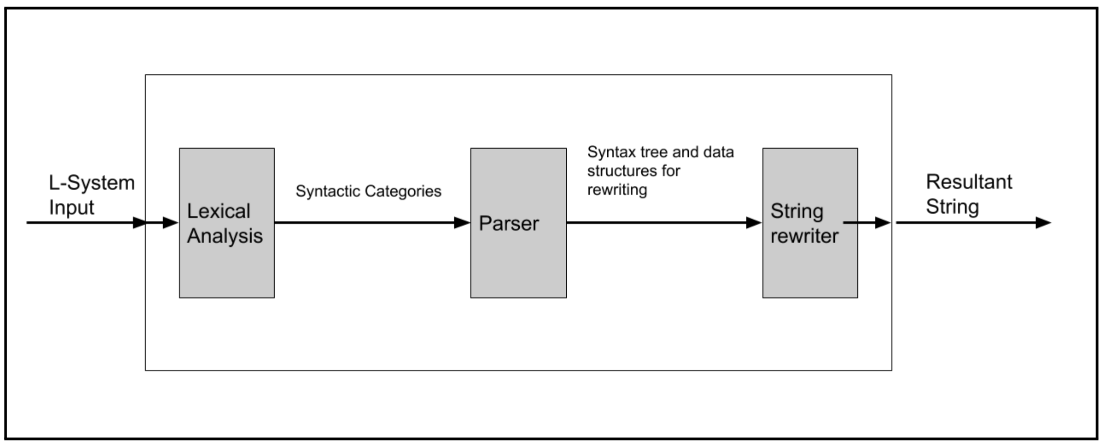


Figure 3.1: Diagram of the Parts of The Rewiting System.

3.1 Environment and Tools

The implementation of the string rewriter, and the string interpreter, is written in the C and C++ programming languages [Stroustrup, 2000]. The C and C++ languages are two of the most common programming languages that have stood the test of time with the first version of C being released in 1974. These languages are frequently used within computer graphics, with some of the most popular game engines supporting either C or C++. Such as CryEngine, Unreal Engine, Source Engine, and more. The main reason for this is the high performance and low-level memory management that C and C++ provide, and the graphics programming frameworks such as OpenGL, Vulkan, and DirectX all having direct support for either C or C++. The C and C++ languages also have a large number of useful libraries that provide extra functionality.

The implementation of the rewriter and the interpreter will use the modern Open Graphics Library (OpenGL). The OpenGL framework is one of the industry standards for creating 3D graphics applications. It is a cross-platform API for interacting with the GPU in a low-level way. The high-performance nature of OpenGL is essential, as displaying and simulating the L-system can be very graphically intensive [Sellers et al., 2013] [Movania et al., 2017]. OpenGL was initially intended to be an API for the C and C++ programming languages. Therefore, both the programming language and graphics API have a strong emphasis on performance, which is necessary when procedurally generating and simulating plant-life.

For more specialised mathematics capabilities, the OpenGL Mathematics Library (GLM) library holds many mathematics classes and functions for conveniently dealing with structures such as vectors, matrices, and quaternions. This thesis will cover these mathematical concepts in chapter ; however, it is convenient to have these implemented and tested within a C++ library. Another important library is Graphics Library Framework (GLFW) which is a multi-platform API for creating an managing user interface windows, events, and user-input

[GLFW development team, 2019]. To keep track of changes and manage versions. Git is a free and open-source version control software. It can keep track of changes that have been made to the files within a project folder as well as keep previous versions of the project throughout the development process. In conjunction with Git, Github is an online web application that stores git repositories. Git acts as a backup as well as containing all previous versions of the project [Torvalds,].

3.2 The L-system as an Interpreted Grammar

Traditionally an interpreter in computing is a program that takes program code as input. It is then analyzed and interpreted as it is encountered in the execution process. All of the previously encountered information is kept for later interpretations. The information about the program can be extracted by inspecting the program, such as the set of declared variables in a block or a function [Wilhelm and Seidl, 2010]. In essence, the L-system rewriter contains a type of interpreter. This should not be confused with the interpreter that processes the resultant string using turtle graphics. Due to this confusion of terms, the system containing the lexical analyser, L-system parser, and the string rewriter will be referred to as the L-system rewriter, instead of the interpreter in the computational sense.

A similarity can be drawn between traditionally interpreted languages and the L-system rewriter. The L-system rewriter defines a set of constant variables, a starting point, and then some production rules. This information can then be used to rewrite the starting string several times. Later on, it may be decided that, instead of five generations of rewriting, the rewriter should instead generate ten. Some information about the L-system is still valid, the production rules, axiom, and constants have not changed, and therefore this information can be used to interpret to the tenth generation. This concept can be used to go from the current state of the L-system rewriter and rewrite another five times. Instead of throwing all the information away and starting from scratch. Furthermore, if we would like to retrieve the resultant string, this can be requested from the L-system rewriter.

The lexical analyser and parser are a necessary part to carry out rewriting. Without the lexical analyser or parser, it would not be straightforward to find the syntactic roles of each part of the L-system. Take the example of the module: F(2*3, x * (2 + y)). Here there is a single module with two parameters, one parameter has the expression (2 * 3), and the other has the expression (x * (2+y)). These complex structures within a grammar require knowledge about the grammatical model it represents. The lexical analyser firstly makes sure that all the syntax within the L-system is correct and assigns each word or symbol to a syntactic category, the parser then splits the L-system into its components and is describes each parts syntactic roll. The lexical analyser provides the understanding that x and y are variables within a module and do not represent something else. It also provides knowledge about how to find the values of x and y.

The difficulty of creating an L-system with more complexity in the grammar is that it becomes more challenging to write a valid L-system to represent a particular structure. For example, imagine trying to write a C program where the compiler does specify why the program is incorrect. The advantage of using a rewriter similar to a compiler is that it makes it simpler to debug any syntactic errors, as well as make the string rewriting much faster. This means that writing an L-system becomes similar to rewriting a recursive program, where any syntactic mistakes will result in a meaningful error describing what was incorrect.

3.3 The Syntax of a Parametric L-system

This section will specify the valid syntax for the parametric L-system rewriter. The syntax is similar to the definition of the parametric L-system definition given by Prusinkiewicz and Lindenmayer in section 2.4.1. There are some additions and modifications to the syntax definition provided by Prusinkiewicz and Lindenmayer to construct an L-system that includes branching, constant variable definitions, object specifications, parametric L-system concepts, randomness, and stochastic L-systems [Prusinkiewicz and Lindenmayer, 2012].

This L-system has five major parts. Each part is categorised as a statement. Valid statements are the *defines* , the *includes*, a single generation statement, a single axiom statement, and one or more production rules [Prusinkiewicz and Hanan, 2013]. All of these statements collectively form an L-system. Each statement starts with a '#' character and ends with a ';' symbol. These are used to indicate the start and end of a statement, even if multiple statements are written on the same line.

The order that statements should be listed is as follows:

$$\begin{aligned}
&\#\text{generations statement;}\\
&\#\text{define statements;}\\
&\qquad ...\\
&\#\text{include statements;}\\
&\qquad ...\\
&\#\text{axiom statement;}\\
&\#\text{production statements;}\\
&\qquad ...
\end{aligned} \tag{3.1}$$

The order for the statements does not always matter; for instance, the generation statement can be defined anywhere within the L-system. However, some parts are required to be in a particular order, such as the define and include statements, which must appear above the axiom and production rule statements as they define values used within the axiom and production rules. It is best practice to specify the L-system in the above order as to avoid any conflictions or errors.

All numbers within the L-system are represented as floating-point numbers. Using a single data-type keeps all numbers consistent. Other data types could be added in the future; however, there are added complexities in doing so, such as the conversion from one type to another, or having to specify which data type a variable represents. The floating-point data type provides all the necessary functionality needed for the L-system; therefore, it seems unnecessary to add more data types.

3.4 The L-system Lexical Analyser

In computer science, specifically the study of programming language compilers, the program responsible for carrying out lexical analysis is the lexer. Depending on the literature the lexer can also be known as the tokenizer or scanner. D. Cooper and L. Torczon write that "The scanner, or lexical analyser, reads a stream of characters and produces a stream of words. It aggregates characters to form words and applies a set of rules to determine whether each word is legal in the source language. If the word is valid, the scanner assigns it a syntactic category or part of speech" [Cooper and Torczon, 2011]. This is no different for the parametric 0L-system rewriter. For the rewriter to have enough information to carry out rewriting, it must first understand what each word or token within the L-system means, this requires assigning a syntactic category to each token, and whether or not the token is valid or not within the L-system grammar.

The scanner itself is quite complex, its main goal is to match the characters or strings within the language, to either a word or a regular expression defined in the grammar. When the match is made the token is given a syntactic category. The mechanism by which it achieves this is known as *finite automata* [Wilhelm et al., 2013]. It is possible to write custom lexer, however, it can be quite complicated and time-consuming to design and implement, and once a custom lexer has been created it is also difficult to change functionality at a later stage. There is a well known program known as the Fast Lexical Analyzer Generator (Flex). Flex takes in a file which contains the lexical rules of the language, this being the strings as well as the regular expression as well as its associated syntactic category. When Flex is executed it will create a lexer in the form of a C program. To create a lexer with Flex, the lexical rules must be defined. Below are the characters, strings and regular expressions and their associated syntactic categories, as well as a description as to its use in the parametric 0L-system.

Syntactic Word	Syntactic Category	Description
,	T_COMMA	Separation between module parameters
:	T_COLON	Separation between production rule parts
;	T_SEMI_COLON	End of a statement
#	T_HASH	Beginning of a statement
(	T_PARENL	Start of a modules parameters or specifies presidence in an expression
)	T_PARENR	End of a modules parameters or specifies presidence in an expression
{	T_BRACKETL	Start of a random range
}	T_BRACKETR	End of a random range
∼	T_TILDE	Stochastic operator
==	T_EQUAL_TO	Relational operator stating equal to
! =	T_NOT_EQUAL_TO	Relational operator for not equal to
<	T_LESS_THAN	Relational operator for less than
>	T_GREATER_THAN	Relational operator for greater than
<=	T_LESS_EQUAL	Relational operator for greater or equal
>=	T_GREATER_EQUAL	Relational operator for greater or equal
[	T_SQUARE_BRACEL	Module name (branching save state)
]	T_SQUARE_BRACER	Module name (branching load state)
+	T_PLUS	Arithmetic operator for addition, or Module name (Yaw right)
-	T_MINUS	Arithmetic operator for subtraction, or Module name (Yaw left)
/	T_FORWARD_SLASH	Arithmetic operator for division, or Module name (Pitch up)
\	T_BACK_SLASH	Module name (Pitch down)
*	T_STAR	Arithmetic operator for multiplication, or Condition in a production rule which is true
∧	T_HAT	Arithmetic operator for and exponent, or Module name (Roll right)
&	T_AMPERSAND	Module name (Roll left)
!	T_EXCLAMATION	Module name (Set size of branch)
$	T_DOLLAR	Module name
=	T_ASSIGN	Assignment operator used to set generations
#n	T_GENERATIONS	Declaration of the number of generations
#w	T_AXIOM	Declaration of the axiom
#define	T_DEFINE	Declaration of the define
#object	T_OBJECT	Declaration of the object
[0-9]+.[0-9]+|[0-9]+	T_FLOAT	Regular expression for a floating point number
[a-zA-Z_][a-zA-Z0-9_]*	T_VAR_NAME	Regular expression for a module or variable name

Table 3.1: Table of Valid Lexer Words

From the table above, several syntactic categories contain more than one meaning; for instance, the open and close parentheses have two meanings. They are used to either specify a modules' parameters or to specify precedence within an expression. It is not up to the scanner to determine what each parenthesis means, or that it has a meaning at all, the lexer only recognises that it falls into the syntactic categories, T_PARENL and T_PARENR. Deriving the meaning of a given token or syntactic category is decided by the parser. The parser is more aware of the context of each syntactic word. Similarly, the symbols [,],+,-,/,\, ∧, &, !, $, and T_VAR_NAME are valid module names. These symbols need to be specifically defined as their syntactic category, as they not only represent a module name but can also represent a different meaning depending on their context. For instance, the +, -, / are valid module names, but they also are mathematical symbols used within arithmetic expressions. The scanner must separate these symbols and keep them in their syntactic category for the parser to be able to understand the same symbol in multiple contexts.

It is also important to note that there are two unique types of tokens. These are the T_FLOAT and T_VAR_NAME. The regular expression for T_FLOAT will match any floating-point value, and the regular expression for T_VAR_NAME will match with any valid variable name. These unique tokens are valid syntactic categories but also contain an associated value. For instance, T_FLOAT has a floating-point value associated with it, and T_VAR_NAME has a string value associated with it. These values must be kept and provided to the parser for use later on.

3.5 The L-system Parser

The parsers' job is to find out if the input stream of words from the lexer is a valid sentence according to the grammar. If the syntactical categories from the lexer match the grammatical model, then the syntax is seen to be correct. If the syntax of the language is correct, the parser will generate a syntax tree and build the relevant data structures for use later on in the compilation process [Cooper and Torczon, 2011]. For the L-system rewriter, the syntax tree and data structures are not used for compilation but rather for the string rewriting process.

In order to describe a grammar, a suitable notation is necessary to express its syntactic structure and grammatical model. According to Cooper, the Backus-Naur Form(BNF) has traditionally been used by computer scientists to represent context-free grammars such as programming languages. Its origins are from the late 1950s and early 1960s. The BNF notation represents the context-free grammar by defining a set of non-terminal symbols that derive from a set of terminal or non-terminal symbols. Terminal symbols are elementary symbols of the language defined by the formal grammar. A terminal symbol will eventually appear in the resulting formal language. On the other hand, a non-terminal symbol exists only as a placeholder for patterns of terminal symbols but does not appear within the formal language itself. The syntactic convention for a BNF is for non-terminal symbols to be surrounded by angled brackets. For instance, <expression> and terminal symbols, such as the symbol for addition "+" to be underlined, but nowadays, it is not often underlined. The symbol ϵ represents an empty string, the ::= means "derives" and the | means "also derives" but is often articulated as an "or" [Cooper and Torczon, 2011]. The very first derivation must be a non-terminal symbol called the goal symbol. The goal symbol is a set of all valid derived strings. This means that the goal symbol is not a word within the language, but rather a syntactic variable in the form of a non-terminal symbol. The BNF notation below can be used to represent a simple grammar for arithmetic expressions, where the terminal "number" is any valid integer, and the goal symbol is <expression>. Below is the BNF notation for the syntax of an arithmetic expression that can represent addition and subtraction.

$$
\begin{array}{rcl}
\langle \text{expression} \rangle & ::= & \texttt{number} \\
& | & \texttt{(}\langle \text{expression} \rangle\texttt{)} \\
& | & \langle \text{expression} \rangle\texttt{+}\langle \text{expression} \rangle \\
& | & \langle \text{expression} \rangle\texttt{-}\langle \text{expression} \rangle
\end{array}
$$

The BNF above states that the goal symbol, <expression> derives from one of four states. Either a terminal number, or an expression contained within two parentheses, or two expressions either side of an addition or subtraction terminal symbol. This type of notation is recursive and allows the formal language to write expressions that exist within other expressions. For example the expression “5 + 10 - (20 + 2)” can be broken down into using the BNF production rule forming a syntax tree as seen in figure 3.2 below. In this case, the whole expression fits the grammatical model of the language. Thus it can be parsed, forming the syntax tree. Computationally, when parsed, this expression will create a data structure, which will be discussed in more detail in section 3.5.4.

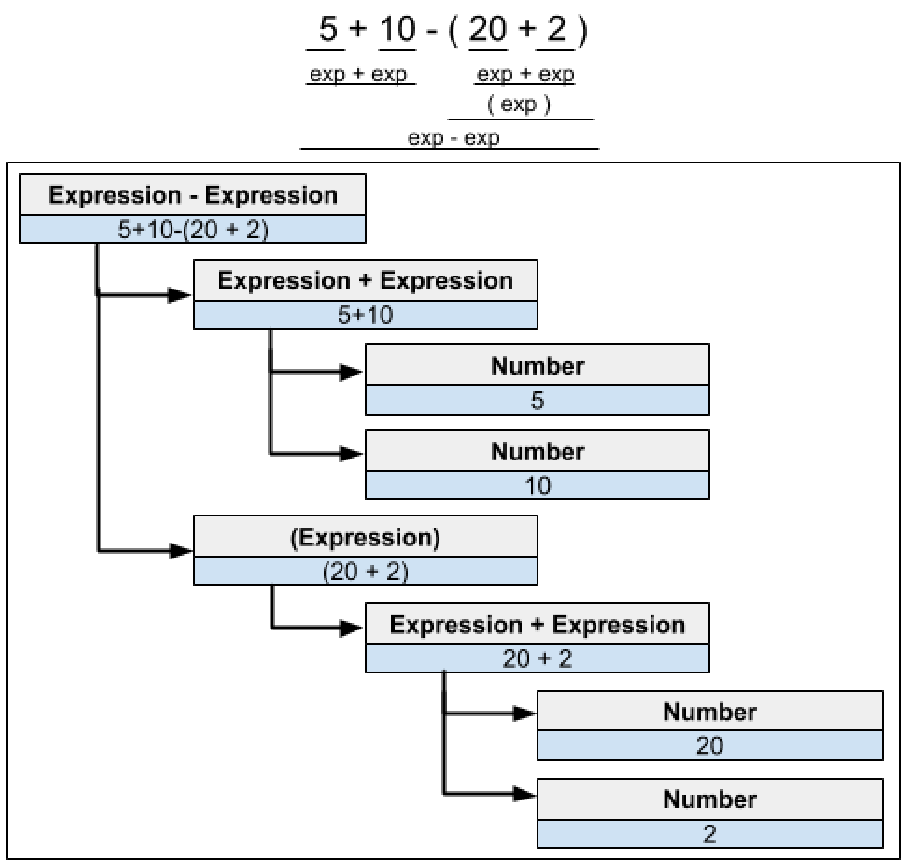


Figure 3.2: Diagram syntax tree for an expression.

Similar to the scanner, the parser program can be quite complex. It needs to find the associated terminal and non-terminal symbols and comply with the grammatical model. Furthermore, if there is a change in the grammar or there is a need to add features at a later date, it is frequently difficult to change the parser. Many studies have been conducted on creating a parsers; however this is beyond the scope of this thesis. Therefore, a program called a parser generator can be used to create the parser program. It uses a specification of the grammar similar to that of the BNF to generate a C program capable of parsing a given language. A popular implementation of a parser generator is called Bison.

3.5.1 Backus-Naur Form of the L-system Grammar

A BNF below is used to describe any possible valid L-system. The Bison program takes a definition similar to this one and creates the parser program. The parser takes in an L-system as input and will process and output the appropriate data structures and information necessary to carry out rewriting.

```
⟨lSystem⟩ ::= ϵ | ⟨statements⟩ EOF
⟨statements⟩ ::= ϵ | ⟨statement⟩⟨statements⟩
⟨statement⟩ ::= EOL | ⟨generation⟩ | ⟨definition⟩ | ⟨object⟩ | ⟨axiom⟩ | ⟨production⟩
⟨generation⟩ ::= #define = ⟨float⟩;
⟨float⟩ ::= [0-9]+.[0-9]+|[0-9]+
⟨variable⟩ ::= [a-zA-Z_][a-zA-Z0-9_]*
⟨number⟩ ::= ⟨float⟩ | - ⟨float⟩
⟨range⟩ ::= {⟨number⟩,⟨number⟩}
⟨definition⟩ ::= #define ⟨variable⟩ ⟨number⟩;
⟨object⟩ ::= #object ⟨variable⟩ ⟨variable⟩;
⟨module⟩ ::= ⟨variable⟩ | + | - | / | \ | ∧ | & | $ | [ | ] | !
               | +(⟨param⟩, ⟨paramList⟩)
               | -(⟨param⟩, ⟨paramList⟩)
               | /(⟨param⟩, ⟨paramList⟩)
               | \(⟨param⟩, ⟨paramList⟩)
               | ∧(⟨param⟩, ⟨paramList⟩)
               | &(⟨param⟩, ⟨paramList⟩)
               | $(⟨param⟩, ⟨paramList⟩)
               | [(⟨param⟩, ⟨paramList⟩)
               | ](⟨param⟩, ⟨paramList⟩)
               | !(⟨param⟩, ⟨paramList⟩)
⟨axiom⟩ ::= #w : ⟨axiomStatementList⟩;
⟨axiomStatementList⟩ ::= ϵ | ⟨axiomStatement⟩⟨axiomStatementList⟩
⟨axiomStatement⟩ ::= ⟨module⟩
⟨paramList⟩ ::= ϵ | ⟨param⟩⟨paramList⟩
⟨param⟩ ::= ⟨expression⟩
⟨expression⟩ ::= ⟨variable⟩ | ⟨number⟩ | ⟨range⟩
               | ⟨expression⟩+⟨expression⟩
               | ⟨expression⟩-⟨expression⟩
               | ⟨expression⟩*⟨expression⟩
               | ⟨expression⟩/⟨expression⟩
               | ⟨expression⟩∧⟨expression⟩
               | (⟨expression⟩)
⟨production⟩ ::= #⟨variable⟩ : ⟨predecessor⟩ : ⟨condition⟩ : ⟨successor⟩;
⟨predecessor⟩ ::= ⟨predecessorStatementList⟩
⟨predecessorStatementList⟩ ::= ϵ | ⟨predecessorStatement⟩⟨predecessorStatementList⟩
⟨predecessorStatement⟩ ::= ⟨module⟩
⟨condition⟩ ::= *
               | ∼⟨float⟩
               | ⟨leftExpression⟩⟨operator⟩⟨rightExpression⟩
⟨leftExpression⟩ ::= ⟨expression⟩
⟨rightExpression⟩ ::= ⟨expression⟩
⟨operator⟩ ::= == | != | <= | >= | > | <
⟨successor⟩ ::= ⟨successorStatementList⟩
⟨successorStatementList⟩ ::= ϵ | ⟨successorStatement⟩⟨successorStatementList⟩
⟨successorStatement⟩ ::= ⟨module⟩
```

As seen above in the BNF notation for a L-system, the goal state is <lSystem>. The <lSystem> can be made up of <statements> beginning with the symbol "#" and ending with the symbol ";", or the End of File (EOF) character signifying the end of the L-system. Each non-terminal <statements> is made up of a <statement> followed by more <statements>, or an empty string (ϵ). The <statement> itself can either be an End of Line (EOL) character or a <generation>, <definition>, <object>, <axiom> or <production> statement. The non-terminal symbols <float> and <variable> specify a regular expression. Each statement then has a number of terminal and non-terminal derivatives that allow the production of all valid L-systems that follow this grammar.

In the previous chapter, the scanner defined the syntactic categories. These syntactic categories are all the valid terminal symbols within the L-system grammar. In essence, the parser takes these syntactic categories and finds if they fit the above BNF, and if so, it extracts the information from the L-system and generates the relevant data structures and syntax tree.

3.5.2 Dealing with Constant Values and Objects

Defining constants and objects is essential as it allows the specification of named variables and module names that have a particular meaning. To define a constant or an object is syntactically similar. The keyword define or include is used, then a variable name followed by a value. The value for a constant is a floating-point number, and the value for an *include* is a name of an object within the predefined object library. Seen below is an example of defining a constant and an object:

$$\begin{array}{l}\text{\#define num 10;}\\ \text{\#define pi 3.1415;}\\ \\ \text{\#include F BRANCH;}\\ \text{\#include S SPHERE;}\end{array} \tag{3.2}$$

The definition variables can be stored as a table, called a constants table, which keeps track of all of the constant variable names as well as their values defined by the L-system, as seen in the table below:

Variable Name	Value
num	10.0
pi	3.1415

Table 3.2: Table of turtle instruction symbols and their meaning to the interpreter

The object table structure is very similar to the constants table. The object table holds the module name, and name of the object in the predefined object library. The object table is not used during rewriting, but it is necessary to provide information during the interpretation of the resulting string.

Module Name	Object Name
F	BRANCH
S	SPHERE

Table 3.3: Table of turtle instruction symbols and their meaning to the interpreter

3.5.3 Implementing Modules and Strings

For the rewriter, it is crucial to understand that there are three significant parts of a module. There is a module name, which is a symbol or string of symbols. Secondly, there is a list of zero or more parameters signified by the open and close parenthesis. If there are no parameters for a module, it can be specified without parenthesis. However, if there are no parameters, there should then be a space between the current module and the next module. Thirdly, each parameter can either contain a number, variable, random number range, or a mathematical expression containing numbers, variables, and parentheses signifying precedence.

It is important to note that there are two types of modules. One being a module definition and the other a module call. Although these are two different types of modules, they can refer to the same thing. The module definition stands as a template for a module within a production rule. These templates do not have to hold actual values but rather the variable names or random ranges, which will be substituted during the rewriting process. Module calls, on the other hand, would appear either in the axiom or in the resultant string. The parameters of a module call will always hold actual numerical values. Below is an example outlining the difference between the module definition and module calls.

$$\begin{aligned} &\#\mathrm{w} : \mathrm{A}(10,\ 20); \\ &\#\mathrm{p1} : \mathrm{A}(\mathrm{x},\ \mathrm{y}) : * : \mathrm{A}(\mathrm{x{+}y},\ \mathrm{y}); \end{aligned} \tag{3.3}$$

In the example 3.3 above, module A(10, 20) within the axiom is a module call, as it contains two numerical values of 10 and 20. In the production rule p1, the predecessor is the module A(x, y), this is a module definition, it states that module A's first parameter has a local variable x, and its second parameter has the local variable y. The calling modules values 10 and 20 will substitute x and y anywhere within the successor statement. The production rule p1's successor has a single module A(x+y, y). This is also a module definition; however, the variables will be substituted during rewriting with the calling modules value. When substituted, the successor will be A(10+20, 20). This module can be further evaluated to A(30, 20). After the successor module has been substituted and evaluated, the successors' modules must have a numerical value. They then become module calls within the resultant string ready for the next stage of rewriting.

A string in the context of a parametric L-system is a list of modules. The modules are linked one after the other, creating a type of string.

3.5.4 Implementing Arithmetic Expressions Trees

As stated previously within the L-system BNF, an expression is either a variable name, a number, or a random range. It is also possible that an expression is part of another expression. Take the example: $5 \times 4 + n$, here there are three expressions 5, 4 and n however, 5×4 is also an expression, as well as $4 + n$. An expression can also be described as any of the expressions above between a set of parentheses, such as $(4 + n)$. The result of the expression is calculated from left to right unless parentheses are used, which prioritises the encapsulated expression to be calculated first. We can represent this expression as an expression tree in the diagram below:

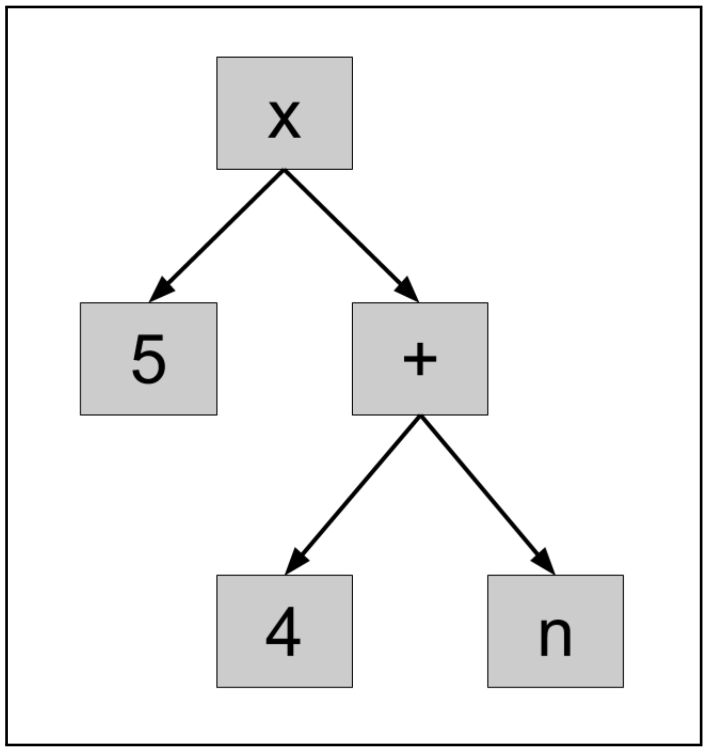


Figure 3.3: Diagram of an expression tree.

The parser provides a syntax tree, which makes it easy to generate the above expression tree. The expression tree can be made up od four types of nodes: a variable, number, random range, or an operator. The leaf nodes of the expression tree must be either a number, variable or random range; moreover, a connecting node within the tree must be an operator. We can then traverse the generated tree and replace the variables with their associated value. For random ranges, the random value can be generated and assigned to the node. A second traversal during the rewriting process can then computes the result of the expression.

3.5.5 Implementing Random Ranges

L-systems are limited in the amount of variation they produce during the rewriting stage. In nature, the variation between the two plants depends on an enormous number of factors. These factors ultimately create variation within the branching structure and in the features of the branches, leaves, and flowers. These features include but are not limited to, branching angles, width, length, height, and weight. When introducing variation in the L-system branching structure, there must be randomness in how rules are chosen. This topic is discussed in section 3.5.6. However, this section introduces a method of providing variation in the features of branch segments, called random ranges.

A random range is a method of declaring a variable that represents a number that is randomly generated between two bounding numbers. The bounding numbers are the minimum and a maximum, respectively. The primary method used for generating a pseudo-random number using a uniform distribution within a range can be seen below.

```
procedure Random Range(min, max)
    n ← (rand() % (max - min + 1)) + min
    return n
end procedure
```

Several other types of pseudo-random number generators could generate numbers according to different distributions, such as normal, binomial, Poisson, among others. When generating plant-life, a uniform distribution should be sufficient for most features and plant-life.

A random range can be declared in three different places within the L-system. It can be declared in the define statement, as an axiom parameter, or a production rule successor parameter. If the random range is declared within a define statement or the axiom, it will generate the random value during the parsing stage. However, if the range is defined in the successor, the number is generated during the rewriting process. More specifically, it is generated when the expressions within the successors are being evaluated. The values are generated during the rewriting process, rather than during parsing because each time a module is rewritten, the number should be a new random number. Generating the numbers during parsing means that the random number is only generated once, and then kept for use later. Conversely, generating the number during rewriting means that a new number will be generated every time rewriting takes place.

3.5.6 Implementing Stochastic Rules

The term "stochastic" refers to a randomly determined process. This could be by a uniform distribution or some random probability distribution.

One of the important factors of generating plant-life is being able to simulate randomness in the generation process. Section 3.5.5 covers a method of generating random numbers that can be used for the features within an L-system. This section covers a different type of randomness that affects the way the rewriter selects a rule for rewriting. In this way, rules can be selected randomly instead of meeting certain conditions. Randomly selecting rules provides randomness within the structure of the plant-life rather than the features.

In order to achieve stochastic rules, each rule must belong belonging to a stochastic group of rules that provides a probability value. The probability indicates how likely it is that rule is selected during the rewriting process. For production rules to be part of the same stochastic group, they are required to meet the following four criteria:

- The stochastic operator $\sim$ must be used with a probability between 0.0 and 1.0.
- The predecessor module name must match the other predecessor module names within that stochastic group.
- The number of parameters within the predecessor must match the number of parameters of other production rules within that stochastic group.
- The total probability of all of the production rules within the stochastic group must not exceed 1.0 or be less than 0.0.

During the parsing phase, if the rule has the stochastic operator, the probability of the rule must be kept for later use within a stochastic probability table. The table also keeps track of which rules are associated with which stochastic groups. A stochastic probability table can be generated from the rules below, as seen in table 3.4.

$$
\begin{aligned}
p_1 &: F(x) : \sim 0.33 : F(x)[+(r)F(x)]F(x)[-(r)F(x)]F(x) \\
p_2 &: F(x) : \sim 0.33 : F(x)[+(r)F(x)]F(x) \\
p_3 &: F(x) : \sim 0.34 : F(x)[-(r)F(x)]F(x)
\end{aligned} \tag{3.4}
$$

Stochastic Group	Rule Name	Probability
F1	p1	0.33
	p2	0.33
	p3	0.34

Table 3.4: Table of the stochastic rules probabilities within a stochastic group.

The stochastic name used within the stochastic table is generated by using the predecessor module name in the production rule, as well as the number of parameters within the predecessor module. In the example above, we can use the predecessor name F, which has a single parameter, making the stochastic name F1. This method of naming serves as a unique identifier for the stochastic group. Once all of the production rules are processed, each groups' probabilities are added together. The total probability should equal 1.0. A tolerance should put in place to account for floating-point error.

During the rewriting process, the module that is being rewritten is matched to a particular stochastic group. A uniformly distributed random number is generated between 0.0 and 1.0. A range for each rule is generated, for instance, p1 will be between 0.0 and 0.33, p2 will be between 0.33 and 0.66, and finally, p3 will be between 0.66 and 1.0. The production rule will be chosen where the random number falls between. For example, if the random number is 0.456, p2 will be chosen as 0.456 falls between 0.33 and 0.66.

3.6 The String Rewriter

Once processed by the lexical analyser and the parser, the L-systems' data structures and information are ready for string rewriting. The string rewriter is the final stage. It starts by starts by using the axiom as the current string of modules. The string rewriter will then iterate over each module within the current string, matching it to the production rules. If they match, the module will be rewritten with the production rules successor, after each parameter in each module is evaluated. Once all the modules are rewritten, the result string replaces the current string of that iteration. This process is carried out for the number of generations specified within the L-system and will eventually provide the final result string of modules.

Below is the pseudocode for the rewriting procedure as well as several useful functions for finding the matching production rule, replacing variables, evaluating expressions, and adding variables to the local table.

```
struct node{
  enum Type {VARIABLE, OPERATOR, NUMBER, RANGE} type;
  union{
        string *variable;
        string *operator;
        float number;
        float range[3];
  };
  node *left;
  node *right;
};
```

```
struct condition{
    enum Type {EQUAL_TO, NOT_EQUAL_TO, LESS_THAN, GREATER_THAN,
        LESS_EQUAL, GREATER_EQUAL, STOCHASTIC, NO_CONDITION} type;
    node * leftExp;
    node * rightExp;

    float stochasticValue;
};
```

```
struct module{
  string name;
  int numParam;
  enum Type {CALL, DEFINITION} type;
  string object;
  vector<struct node*> params;
};
```

```
struct production{
    string name;
    module *predecessor;
    condition *condition;
    vector<module*> successor;
};
```

```
procedure REWRITER(N, A)
Ensure: N > 0                                  ▷ The number of generations to rewrite
Ensure: A ≠ empty                              ▷ A non empty Axiom, a list of modules
    n ← 0
    current ← A                                ▷ Current string of modules
    while n < N do                             ▷ For each generation
        next ← empty list
        for each mod in current do             ▷ call is the calling module in current
            P ← FindProductionMatch(mod)       ▷ P is the matching production rule
            if P ≠ NULL then
                pred ← P.predecessor           ▷ def is the defining module in predecessor
                for each succ in P.successor do
                    index ← 0
                    while index < number of predecessor parameters do
                        AddLocalVar(pred.param[index], mod.param[index])
                        index ← index + 1
                    end while
                    copy ← succ                ▷ Create a deep copy
                    for each parameter in copy do   ▷ parameter is an expression tree
                        ReplaceVariables(parameter)
                        EvaluateExpression(parameter)
                    end for
                    next ← next + copy
                end for
            else
                next ← next + mod
            end if
        end for
        n ← n + 1
        current ← next
    end while
    return current
end procedure
```

```
function FindProductionMatch(Module)
    for each P in productionTable do                                          ▷ P is a production
        predecessor ← P.predecessor                          ▷ predecessor is a single module
        if predecessor.name ≠ Module.name then
            continue
        end if
        if predecessor.numParam ≠ Module.numParam then
            continue
        end if
        if P has no condition then
            return P.name                                                         ▷ match found
        else if P has a stochastic condition then
            rand ← random float between 0.0 and 1.0
            total ← 0.0
            S ← list of pairs                          ▷ pair(production name, probability value)
            for each s in S do                   ▷ Loop through each tuple in the stochasic list
                if first item then
                    if rand ≥ 0.0 AND rand < s.value then
                        return s.name
                    end if
                else if last item then
                    if rand ≥ total AND rand ≤ 1.0 then
                        return s.name
                    end if
                else
                    if rand ≥ total AND rand < total + s.value then
                        return s.name
                    end if
                end if
                total ← total + s.value
            end for
        else                                                                  ▷ Regular condition
            left ← P.condition.left                          ▷ Deep copy left expression tree
            right ← P.condition.right                       ▷ Deep copy right expression tree
            replaceVariables(left)
            replaceVariables(right)
            evaluateExpression(left)
            evaluateExpression(right)
            if left P.condition.op right then          ▷ Apply operator (==, ≠, <, >, ≤, ≥)
                return P.name
            end if
        end if
    end for
end function
```

```
function EvaluateExpression(TreeNode)   ▷ Recursively evaluate the expression tree
    left ← 0.0
    right ← 0.0
    if TreeNode.left == NULL OR TreeNode.right == NULL then
        return TreeNode.value
    end if
    left ← ReplaceVariables(TreeNode.left)
    right ← ReplaceVariables(TreeNode.right)
    if TreeNode.type is an operator then
        return left TreeNode.operator right    ▷ Apply arithmetic operator (+, -, *, /, ∧)
    end if
end function

function ReplaceVariables(TreeNode) ▷ Recursively replace expression tree variables
    if TreeNode == NULL then
        return
    end if
    if TreeNode.type is a variable then
        if TreeNode.value is in constants table then
            TreeNode.value ← numeric value in constants table
        end if
        if TreeNode.value is in local table then
            TreeNode.value ← numeric value in local table
        end if
    end if
    ReplaceVariables(TreeNode.left)
    ReplaceVariables(TreeNode.right)
end function

function AddLocalVar(TreeNodeCall, TreeNodeDef)
    if TreeNodeCall child nodes == NULL OR TreeNodeDef child nodes == NULL then
        if TreeNodeCall.type == Number AND TreeNodeDef.type == Variable then
            Add variable name and value to local table
        else if TreeNodeCall.type == Range AND TreeNodeDef.type == Variable then
            Add variable and generated random range value to local table
        end if
    end if
end function
```

3.7 Summary

The L-system rewriter is the first of two major systems within the process of procedurally generating plant-life. The second system is the interpreter. The rewriters' purpose is to take an L-system input and understand its grammatical structure and carry out string rewriting.

The rewriter system defined in this thesis acts as a type of compiler, which is similar to a computer language. The L-system becomes a type of language that the L-system rewriter can understand. This understanding allows the creation of data structures, which allow the rewriting process to be carried out. The lexical analysis stage and the parser stage, give informative messages if there is a mistake, either grammatically or syntactically. If the language meets all of the grammatical and syntactic requirements, the rewriter can use this data to generate the resultant string of modules. The result string produced by the rewriting system will always be a valid string according to the L-system grammar.

The L-system languages can be used for many different applications and are not limited to that of procedural plant generation. The interpreter uses the resultant string to create the final rendered representation, such as the plant model. The advantage of having the rewriter be complex is that the rewriting system does not need to change, even if the L-system is used for a different purpose. Only the interpretation will need to change in order to understand the resulting string. This is the main reason behind using a compiler-like process to govern the string rewriting. It allows the L-system enough complexity to provide information to the interpreter, but not so much that interpretation becomes reliant on the string rewriter.

Chapter 4

Mathematics For 3D Graphics

In any 3D application, mathematical models are used to represent the positions, rotations, and scale of objects within a given scene. It is crucial for this thesis to briefly touch on some of the core concepts, particularly for representing and manipulating 3D objects.

All objects within a 3D application are made up of a set of vertices or points, which are represented with X, Y, and Z coordinates. Three vertices can make up one triangle, also called a face, multiple faces will then make up a whole 3D object. The use of mathematical methods in 3D graphics is to manipulate the vertices within an object consistently. These methods include: rotating, translating, or scaling objects within a scene.

This section will provide sufficient background on some of the essential concepts of 3D Mathematics, such as vectors, matrices, and quaternions, that are used widely in the turtle graphics interpreter as well as the model generator.

4.1 Vectors

Vectors have many meanings in different contexts, in 3D computer graphics, vectors often refer to the Euclidean vector. The Euclidean vector is a quantity in n-dimensional space that has both magnitude and direction. Vectors can be represented as a line segment pointing in a direction, with a certain length. A 3D vector can be written as a triple of scalar values eg: (x, y, z).

The most common operations on vectors are multiplication by a scalar, addition, subtraction, normalisation and the dot and cross product. The multiplication by a scalar value can be seen as scaling the magnitude of the vector. This operation can be done uniformly or non-uniformly, as seen in the equation below:

$$a \otimes s = (a_x s_x, a_y s_y, a_z s_z) \tag{4.1}$$

Where $\otimes$ is the component-wise product of vector a, and the scaling vector s. Similar to the scalar product of a vector, the addition and subtraction of two vectors are the component-wise sum or difference. The equation for the sum and difference of a vector with a scalar value can be seen below.

$$\begin{aligned} a \oplus b &= [(a_x + b_x), (a_y + b_y), (a_z + b_z)] \\ a \ominus b &= [(a_x - b_x), (a_y - b_y), (a_z - b_z)] \end{aligned} \tag{4.2}$$

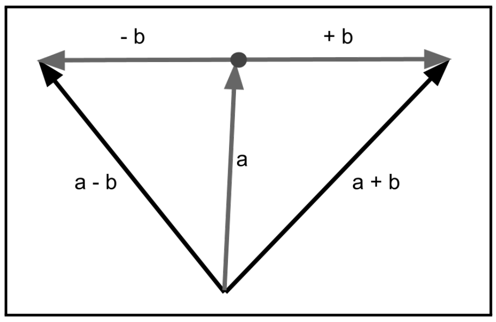


Figure 4.1: Table of common dot product tests between two vectors.

A type of vector that is used very often in 3D graphics is known as a unit vector. This is a vector that has a magnitude of 1. Unit vectors are used extensively, particularly with shaders. Take the vector v its magnitude α can be calculated by taking the square root of the product of its components squared, as seen below.

$$\alpha = \mid \mathbf{v} \mid = \sqrt{\mathbf{v}_x^2 + \mathbf{v}_y^2 + \mathbf{v}_z^2} \tag{4.3}$$

The unit vector can then be calculated by taking the product of v and the inverse of its magnitude shown in the following equation.

$$\upsilon = \frac{\mathbf{v}}{\alpha} = \frac{1}{\alpha}\mathbf{v} \tag{4.4}$$

There are many different ways to multiply vectors. The two main multiplications being the dot and cross product. The dot product yields a scalar value by adding the products of the vector product components. The cross product, on the other hand, is the product of two vectors, which gives a vector that is perpendicular. The dot product can be calculated using the formula below.

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z = d \tag{4.5}$$

Some of the primary uses for dot products within 3D graphics is to find whether two vectors are collinear, perpendicular, or if they are in the same direction or opposite directions. One possible use for this is to find if two branches are growing in the same direction or in opposite directions. In the table 4.1 below, there are all of the dot product tests as well as its equation. Please note that $ab =\mid a \mid\mid b \mid= a \cdot b$.

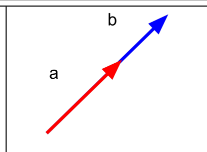
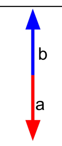
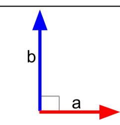
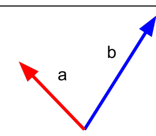
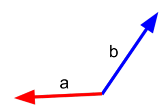
Test	Equation	Example
Collinear	$(a \cdot b) = ab$	
Opposite Collinear	$(a \cdot b) = -ab$	
Perpendicular	$(a \cdot b) = 0$	
Same Direction	$(a \cdot b) > 0$	
Opposite Direction	$(a \cdot b) < 0$	

Table 4.1: Table showing the dot product tests and an example of their use.

The cross product, also known as the outer product, takes two different vectors and finds the perpendicular vector of those two vectors. The cross product is only possible in 3D space and can be expressed in the following formula using the left-hand rule.

$$a \times b = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)] \tag{4.6}$$

The result of a cross product can be seen in figure 4.1 below. Where vectors a and b give the perpendicular vector $a \times b$. The cross product is beneficial within physics calculations when it's necessary to find the rotational motion of objects. It is also used in the graphics shader when finding the normal vector in light calculations.

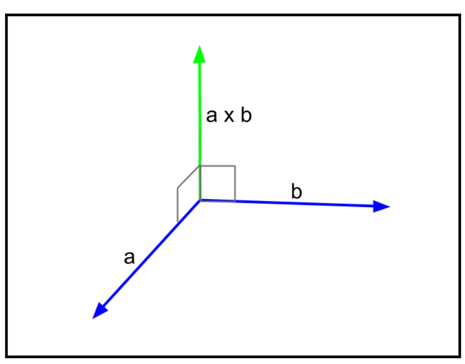


Figure 4.2: Diagram of the cross product of two vectors a and b.

Some properties of the cross product are as follows:

- It is non-commutative, meaning order matters($a \times b \neq b \times a$).
- It is anti-commutative ($a \times b = -(a \times b)$).
- It is distributive with addition ($a \times (b + c) = (a \times b) + (a \times c)$).

4.2 Matrices

A model in 3D space exists as a set of position vertices, often represented as vectors. Moving the model requires moving all of the vertices of that model without distorting it in any way. Moving a model like this is called a model transform. There are four main types of transforms, these being: translation, rotation, scale, and shear. Matrices are a single mathematical construct capable of carrying out all four of these transformations. This section will only cover the first three as the shear transformation only used in certain circumstances and will not be useful in this thesis.

A matrix is a 2D array of numbers arranged into rows and columns, which can come in many different sizes. In 3D graphics, matrices used for transformations are 3×3 and 4×4 matrix, as seen below.

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \tag{4.7}$$

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \tag{4.8}$$

A 3×3 matrix is used for linear transforms such as scaling and rotation. Furthermore, a linear transform that contains translation is known as an affine transform and is represented as a 4×4 matrix known as an Atomic Transform Matrix. An atomic Transform matrix is the concatenation of four 4×4 matrices, one for translation, rotation, scale, and shear transforms, which results in a 4×4 matrix, as shown below.

The affine matrix can be shown in the expression below where RS is a 3×3 matrix containing the rotation and scale where the 4^th elements are 0. The T elements represent the translation, with the 4th element being 1.

$$\mathbf{M} = \begin{bmatrix} RS_{11} & RS_{12} & RS_{13} & 0 \\ RS_{21} & RS_{22} & RS_{23} & 0 \\ RS_{31} & RS_{32} & RS_{33} & 0 \\ T_1 & T_2 & T_3 & 1 \end{bmatrix} \tag{4.9}$$

The product of two linear transform matrices will be another linear transform matrix where both of the transformations have taken place. This is true for the multiplication of two affine transform matrices as well, and is why matrix multiplication is so powerful in 3D graphics. Take the two matrices, A and B, which give the product P. To multiply A and B, the dot product of the row and the column is must be calculated, which can be seen in the equation below. It is also important to know that matrix multiplication is non-commutative ($AB \neq BA$).

$$\mathbf{AB} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} (A_{row1} \cdot B_{col1}) & (A_{row1} \cdot B_{col2}) & (A_{row1} \cdot B_{col3}) \\ (A_{row2} \cdot B_{col1}) & (A_{row2} \cdot B_{col2}) & (A_{row2} \cdot B_{col3}) \\ (A_{row3} \cdot B_{col1}) & (A_{row3} \cdot B_{col2}) & (A_{row3} \cdot B_{col3}) \end{bmatrix} \tag{4.10}$$

Translating a vertex in 3D space using matrices is is relatively straightforward. The vertex can be is added to the matrix as seen in the equation below. Where V is the vertex and the T is the translation matrix. To rotate an entire model, the same translation matrix can be applied to all vertices.

$$V + T = \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} = \begin{bmatrix} (V_x + T_x) \\ (V_y + T_y) \\ (V_z + T_z) \\ 1 \end{bmatrix} \tag{4.11}$$

To rotate a vertex in 3D space, the vertex position and rotation angle can be applied to the matrix differently depending on the axis about which it is rotating. Similar to the translation matrix, rotation matrices are applied to each vertex, to gain the new position of the vertex. The rotation matrices below rotate a vertex around the x, y, and z axes, repectively.

$$R_x(\theta) = \begin{bmatrix} (v_x) \\ (v_y) \\ (v_z) \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.12}$$

$$R_y(\theta) = \begin{bmatrix} (v_x) \\ (v_y) \\ (v_z) \\ 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.13}$$

$$R_z(\theta) = \begin{bmatrix} (v_x) \\ (v_y) \\ (v_z) \\ 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.14}$$

Similarly, the scale transform takes a vertex and multiplies it by the scale matrix. If there are a large number of vertices making up an entire model if all of the points are scaled using the scale transform, the result will be the model either increasing or decreasing in size.

$$VS = \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (V_x S_x) \\ (V_y S_y) \\ (V_z S_z) \\ 1 \end{bmatrix} \tag{4.15}$$

4.3 Quaternions

In computer graphics, there are several ways to represent 3D rotations. One method is to use matrix affine transforms, that is spoken about in the previous section. Matrices are a common way of representing rotation; however, there are some limitations. Matrices are represented by nine floating-point values and can be computationally expensive to store and process, mainly when doing a vector to matrix multiplication. There are also situations where it is necessary to interpolate from one rotation to another, smoothly, or to find the rotation somewhere between two different rotations. It is possible to make these calculations using matrices, but it can become very complicated and even more computationally expensive. Quaternions are the answer to these challenges.

Quaternions look similar to a 4D vector. They contain four axes $q = [q_x, q_y, q_z, q_w]$, these are represented with a real axis (q_w) and three imaginary axes (q_x, q_y, q_z). A quaternion can be represented in the complex form below:

$$q = (iq_x + jq_y + kq_z + qw) \tag{4.16}$$

For this thesis, it is not essential to understand the derivation of quaternions mathematically. However, it is essential to understand that a quaternion obeying the rule in 4.17 below is known as a unit quaternion.

$$q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1 \tag{4.17}$$

Unit quaternions can be used for rotations, and it is possible to convert a quaternion to a unit quaternion by taking the angle and the axis of rotation and applying it to the quaternion as seen in 4.18 below.

$$\begin{aligned}
&q = [q_x, q_y, q_z, q_w] \\
&\text{where} \\
&q_x = a_x sin\frac{\theta}{2} \\
&q_y = a_y sin\frac{\theta}{2} \\
&q_z = a_z sin\frac{\theta}{2} \\
&q_w = cos\frac{\theta}{2}
\end{aligned} \tag{4.18}$$

The scalar part (q_w) is the cosine of the half-angle, and the vector part ($q_x q_y q_z$) is the axis of the rotation, scaled by the sine of the half-angle of rotation.

Some of the most useful features of a quaternion are the ability to rotate vectors, interpolate between two rotations, and concatenate rotations together.

The first operation for quaternions is addition. The addition of two quaternions is quite simple. It involves taking each component of each quaternion and adding them together. This method is similar to matrix addition and can be expressed as follows.

$$p + q = [(p_w + q_w), (p_x + q_x), (p_y + q_y), (p_z + q_z)] \tag{4.19}$$

The multiplication of quaternions is also incredibly powerful and can be used to concatenate many rotations together without any gimbol lock. There are several different types of quaternion multiplication. However, the one most commonly used for quaternion rotation is called the Grassmann product.

The Grassmann product can be expressed in the formula below. Where p and q are quaternions, and the subscript w indicates the scalar part and subscript x, y, z indicates the vector components of each quaternion.

$$\begin{aligned}
&R = r_w + r_x + r_y + r_z \\
&\text{where} \\
&r_w = p_w q_w - (p_x q_x + p_y q_y + p_z q_z) \\
&r_x = p_w q_x + p_x q_w + p_y q_z - p_z q_y \\
&r_y = p_w q_y + p_y q_w - p_x q_z + p_z q_x \\
&r_z = p_w q_z + p_z q_w + p_x q_y - p_y q_x
\end{aligned} \tag{4.20}$$

Multiplying two quaternions together is important for multiple rotations taking place one after the other. However, rotate a quaternion by a vector, the vector will need to be converted into its quaternion form. This requires taking the unit vector v and using it as the vector part of the quaternion with a scalar part being equal to zero. This can be written as $Q_v = [v, 0] = [v_x, v_y, v_z, 0]$. The Grassmann product can be used to apply the rotation, by taking the product of the rotation quaternion q and the vector form quaternion v and the inverse of the rotation quaternion q^-1.

$$V_q = qvq^{-1} \tag{4.21}$$

The conjugate and the inverse of a unit quaternion are identical. The conjugate or inverse of a unit quaternion can be calculated by negating the vector components 'q_v' of the quaternion while leaving the scalar component 'q_s' the same. The inverse of a unit quaternion can be expressed as follows.

$$q^{-1} = [-q_v, q_s] \tag{4.22}$$

Concatenating quaternion rotations together is similar to how matrix affine transformations can be multiplied together. The Grassman product can be used. The Grassman product is noncommutative and, therefore, order matters. The quaternion multiplication would result in the rotation that represents all rotations, if they were to happen one after the other. This can be expressed in the equation below.

$$Q_{net} = Q_3 Q_2 Q_1 \tag{4.23}$$

The order the quaternions Q_1, Q_2, and Q_3 is applied is: Q_3, Q_2, and then Q_1. The product of three quaternions can be applied to a vector by multiplying the product of the quaternions to the vector, then multiplying the product of the inverse of the quaternion as seen below.

$$v' = Q_3 Q_2 Q_1 \ v \ Q_1^{-1} Q_2^{-1} Q_3^{-1} \tag{4.24}$$

Another incredibly useful mathematical function is called rotational linear interpolation, also known as LERP. The LERP function takes two quaternions, Q_1 and Q_2, and linearly interpolates between those two rotations by a given percentage β. The LERP function can be defined as follows.

$$\begin{aligned} Q_{\text{LERP}} = \text{LERP}(Q_1, Q_2, \beta) &= \frac{(1-\beta)Q_1 + \beta Q_2}{\mid (1-\beta)Q_1 + \beta Q_2 \mid} \\ &= \text{normalize}\left(\begin{bmatrix} (1-\beta)Q_{1x} + \beta Q_{2x} \\ (1-\beta)Q_{1y} + \beta Q_{2y} \\ (1-\beta)Q_{1z} + \beta Q_{2z} \\ (1-\beta)Q_{1w} + \beta Q_{2w} \end{bmatrix} \right) \end{aligned} \tag{4.25}$$

Using the linear interpolation function will result in a rotation between Q_1 and Q_2 at a given percentage of β, which can be specified between 0 and 1. Where 0 is the rotation of Q_1 and 1 is rotation of Q_2. LERP is very helpful in many areas of 3D graphics and is used extensively within the physics simulation of branches covered in chapter 6.

4.4 summary

This chapter covers the three major mathematical concepts used for representing a 3D objects' position, rotation, and scale within 3D graphics applications. This includes moving objects around a scene, or for animation or simulation of an object. It is essential to understand these concepts when implementing the L-system interpreter, as it is used to manipulate the branches or objects. These concepts are also useful in the implementation of the physics simulations. The OpenGL Mathematics Library (GLM) library provides a large number of useful classes and functions for working with vertices, matrices, and quaternions and can be used instead of re-implementing these mathematical functions.

Chapter 5

L-system String Interpreter Implementation

The string interpreter is one of the significant components of plant generation. It is the final step in the process of procedural generation. The output of this stage is dependant on what the L-system is representing. In this case, it is responsible for creating the final plant models and information and then rendering it on the screen, using the OpenGL framework.

The generation of plant-life has three main stages. The first part is the turtle graphics interpreter, then the model generator, and finally, the renderer. The turtle graphics interpreter takes the string of modules provided by the rewriter, as a set of instructions. It starts from the root of the tree and generates a skeletal structure made up of joints. This is similar to the techniques used in skeletal rigging in animation [Gregory, 2014]. The tree skeleton joints each represent a branch segment or part of the tree. These joints have some information about the properties of that segment. The joint data is then used to generate the model data in the second stage of processing. The model generator creates the 3D points that make up the plant, as well as texture and lighting information. The models can finally be passed to the final part of the string interpreter, which is the renderer. The renderer is responsible for taking all the vertex, texture, and lighting data and renders the final plant on the screen. The renderer will also handle any physical simulation of the tree skeleton. The stages of string interpretation can be seen in figure 5.1 below, as well as the information needed at each stage.

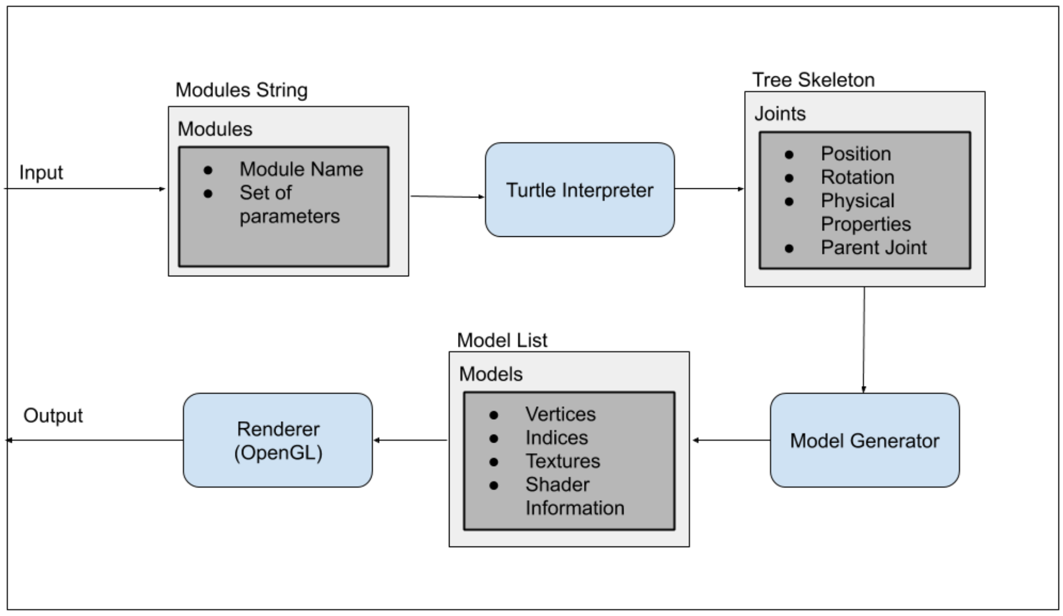


Figure 5.1: Diagram of the stages of L-system interpretation and rendering

This chapter will cover each stage of the string interpreter implementation in detail. This chapter will also introduce how the interpreter can simulate and animation the plants' movements under forces such as gravity and wind in real-time.

5.1 Turtle Graphics Interpreter

The primary purpose of the turtle graphics interpreter is to take the string of modules from the L-system rewriter and interpret each module as a turtle graphics instruction. Each instruction carries out a particular job in creating the overall structure of the plant. This stage is purely to follow the turtle graphics instructions and generate the skeletal data of the plant for the next stage.

This section is an implementation of what was covered in section 2.2. Where a turtle graphics interpreter was defined for a simple DOL-system, however, there are several differences. The main difference is that the L-system string being interpreted is a parametric L-system and, therefore, consists of modules and parameters. Despite these differences, the overall concept remains the same. Each module name within the L-systems resultant string represents a particular meaning to the turtle graphics interpreter. The meaning of the module names are predefined in the string interpreter system and are dependant on what the L-system is trying to represent. The L-system defined for this thesis allows each module to provide optional parameters. These parameters may also carry particular meanings for the interpreter. For instance, the forward instruction or module name "F" can have three parameters. The value of the first parameter is the distance to move forward. The second and third parameter is the spring constant of the branch and the mass of the branch, respectively. These are useful in a physics simulation in order to animate the plant.

Below is a table describing the L-system module names as well as the parameter meanings for the turtle graphics interpreter. In all of the instructions, there is also the case where no parameter is provided. This is still valid; however, if no parameter value is provided, a default value will be used.

Instruction Name	Meaning	Parameter 1	Parameter 2
F	Forward (Render)	Length	Spring Constant
f	Forward (Don't render)	Length	Spring Constant
+	Yaw Right	Angle	N/A
-	Yaw Left	Angle	N/A
/	Pitch Up	Angle	N/A
\	Pitch Down	Angle	N/A
∧	Roll Right	Angle	N/A
&	Roll Left	Angle	N/A
!	Change Width	Branch Width	Resolution of Branch
[	Save State	N/A	N/A
]	Load State	N/A	N/A

Table 5.1: Table of turtle instruction symbols and their meaning to the interpreter

Each modules' instruction is carried out one by one to generate the plants' skeletal structure. The skeleton starts without any joints at the root location. All of the rotation instructions change the current angle of the turtle, and the change width instruction changes the value of its width. When one of the forward instructions is reached, a joint is created and added to the plants' skeleton. The joints hold information about the properties of each particular segment or object of the plant. The joints properties are the position, orientation, scale, parent joint, as well as its physical characteristics. It is important to note that all of the rotation and scale transforms must happen before the forward movement. A joint holds the properties seen in the diagram below.

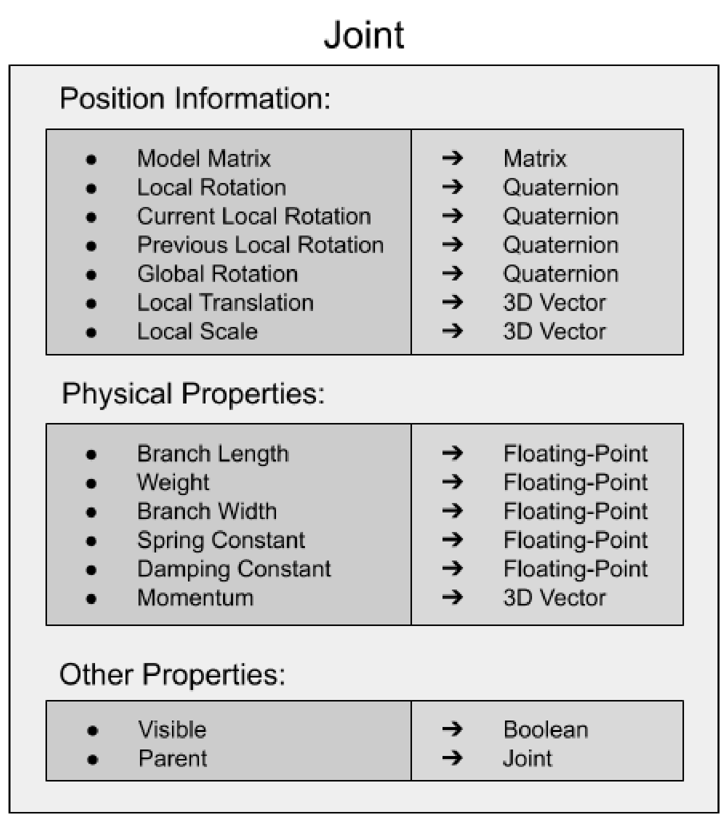


Figure 5.2: Diagram for the properties of a joint

Figure 5.2 shows that there is a large amount of information stored for the position and orientation of each joint. This is because the rotation of the joint is stored in both a local and global space. Local space refers to the rotation of the joint relative to its parent rotation. This is useful as it allows the manipulation of subsequent child joints while leaving other joints local rotation unchanged. Global space, also known as world space, is the rotation of each joint relative to the world itself. This is useful for understanding the current rotation of the joint relative to the world. It is essential to store both the current and previous rotations as they are used to calculate the rate of change for physics calculations.

The physical properties for each joint are the parts that will affect model generation as well as physics simulations. These properties include the length, width, spring constant, damping constant as well as the current momentum of the branch.

Take the following string of modules "F(1)[/(90)F(1)\ (90)F(1)]-(90)F(1)+(90)F(1)", the alphabet is made up of seven unique modules F, /, \, [,], + and -. As discussed in previous chapters the "F" symbol represents a move forward, and "+", "-", "/", "\" symbolize different rotations, and the "[" and "]" represent save and load state respectively. The symbols in the string above each have a single parameter except the load and save state. It is the turtle graphics interpreters' job to understand what these parameters are and how to interpret them. In this case, all of the "F" modules have the parameter value of 1, and all of the rotation modules have the parameter of 90. These are interpreted as the length to move forward and the change in angle from the previous joint. This interpretation can be displayed with the joint structure shown in figure 5.3 below.

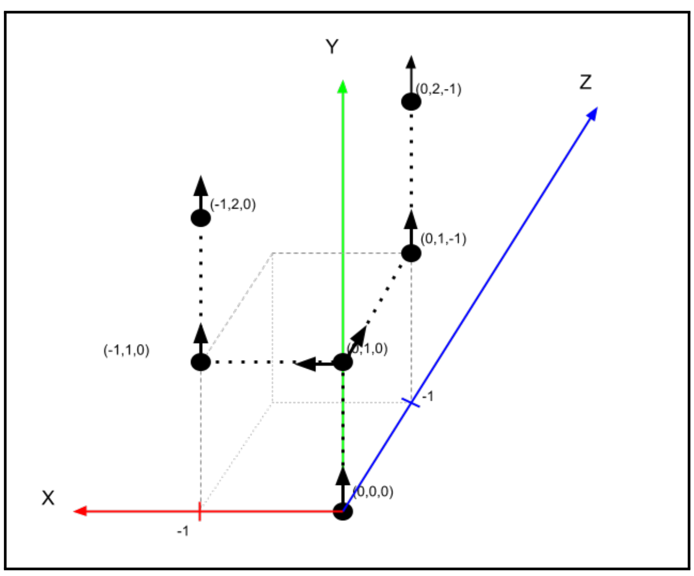


Figure 5.3: Diagram of a simple plant skeleton with joint position and orientation.

5.2 Model Generator

Modeling the branches of a plant is one of the most critical parts of the look and feel of the plant being generated. The plant skeleton and joints describe details about the plants' structure. The job of the model generator is to take the skeleton information and intelligently generate the 3D models' that make up the plants' branches, leaves, or flowers. The models of these objects are made up of vertices, normals, texture coordinates, and other low-level information that can then be provided to the OpenGL renderer and finally displayed on the screen using the GPU.

There are many different ways of procedurally modeling the branching branches of a plant. The simplest would be to take several cylinders, rotate and stack them according to each joints position in 3D space. The upside to this approach that it is very efficient, as every branch within the plant shares the same object model, which is a cylinder. This method can approximate the branching structure of the plant. However, there is a problem, which was pointed out by Baele and Warzée “The branches junction causes a continuity problem: to simply stack up cylinders generates a gap” [Baele and Warzee, 2005]. The continuity problem can be seen in the figure below.

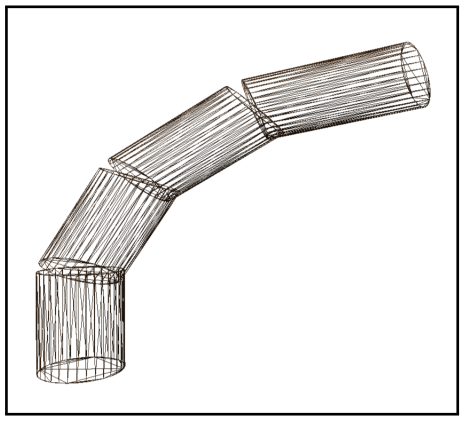


Figure 5.4: Example of the continuity problem faced with stacked branching with a 25° bend per joint.

The simple method of stacking cylinders gives an approximation of the tree structure. It is usually a good enough representation when the branch angles are not more than 25°, and the size of the branches do not change. However, for a much more convincing tree structure, there will need to be a better solution.

An improvement would be to link all of the branch segments together to make the entire branching structure seamless. The top vertices from the parent branch must be linked with the bottom of its child branch. The vertices that make up the top and bottom of a branch are circles of vertices, which are linked together using indexing. These circles will have to rotate depending on the bending direction of the branch. This means that the final model will not be made up of a large number of the same model but rather a single large model.

There are several points to keep in mind for linked branching. The first is that this process is much less efficient than rendering the same cylindrical object many times. The reason for this is that every vertex within the tree needs to be calculated, generated, and finally linked. The second point is what happens when there are multiple branches off a single joint. This will be covered in more detail later. The final point has to do with the resolution of the branch. The resolution is the number of points making up the circumference of the branch. The resolution can be increased or decreased as needed. A higher resolution plant might look better but will also be more resource-intensive to render. Conversely, a lower resolution plant might look a bit more jagged, but be far less resource-intensive to render. An example of the linked branching can be seen below.

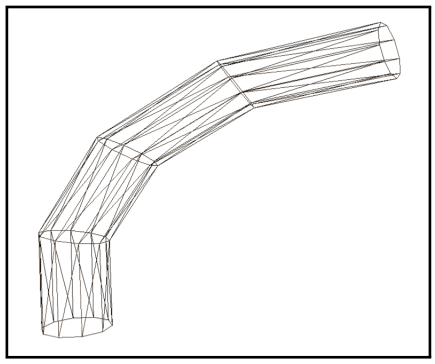


Figure 5.5: Example of linked branching with a 25° bend per joint.

This method of branch generation, at first glance, gives a very similar result to that of stacking cylinders. Although it does have a few advantages, firstly, it completely avoids the branch gap problem when there are larger angle changes, as well as branch size changes. As discussed previously, the second advantage is that the resolution is dynamic. This can be seen in figure 5.2 below, where a similar-looking branch can be achieved using less than half the number of vertices, with joined branches instead of stacked branches.

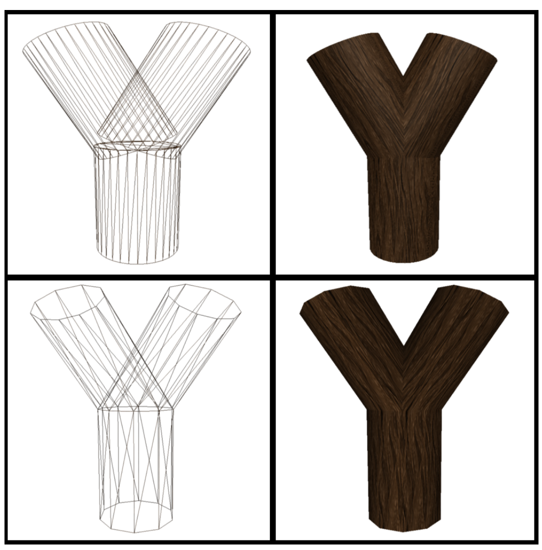


Figure 5.6: Stacked Vs Linked.

5.3 Renderer

The renderer is the final stage in the procedural generation pipeline. It takes all of the 3D models generated by the model generator, such as leaves, branches, flowers, and renders them on the screen. For this thesis, OpenGL is use used to render the models on the screen using the GPU efficiently.

The GPU is a specially designed piece of hardware for processing computer graphics and image processing. It has hundreds or even thousands of individual compute cores that can be used in parallel. Due to the highly parallel nature of the GPU, the OpenGL framework helps to abstract the hardware and create an interface to interact with the GPU in a more straightforward way. There are several other types of graphics API, such as Vulkan, Metal, or DirectX. These APIs all provide a way of interacting with the hardware behind the scenes. However, they each have a different approach. Therefore, this section will not be going into great detail about the specifics of OpenGL but rather the general concepts required for rendering the plant model on the screen. The main parts of the rendering stage have to do with how model data is stored into buffer objects, secondly how textures are stored and then mapped to a specific object and thirdly how lighting can be calculated for a procedurally generated object.

5.3.1 Models and Buffer Objects

The model generator produces all of the information necessary for the renderer to produce the result on the screen. In general, the model data will consist of vertex data, texture coordinates, and vertex normals. The vertex data is simply the position of a point within the model, three vertices make up a face, and the faces are ultimately rendered on the screen. The texture coordinates are the locations on a texture image that maps directly to the model vertices, in order to have a textured object in the scene. Finally, the vertex normals, known as normals, are the average normal vector. A normal vector is a vector that is perpendicular to the surface at a given point and can be used for Phong shading or other types of lighting techniques.

One of the most important parts of the rendering process is buffering the model data onto the GPU. The Vertex Buffer Object (VBO) is a data structure within the OpenGL library which can be used to store this data on the GPU. Generally, the data is stored as a single buffer or array with the first three values being a vertex position, the second two being a texture coordinate, and the last three being a vertex normal.

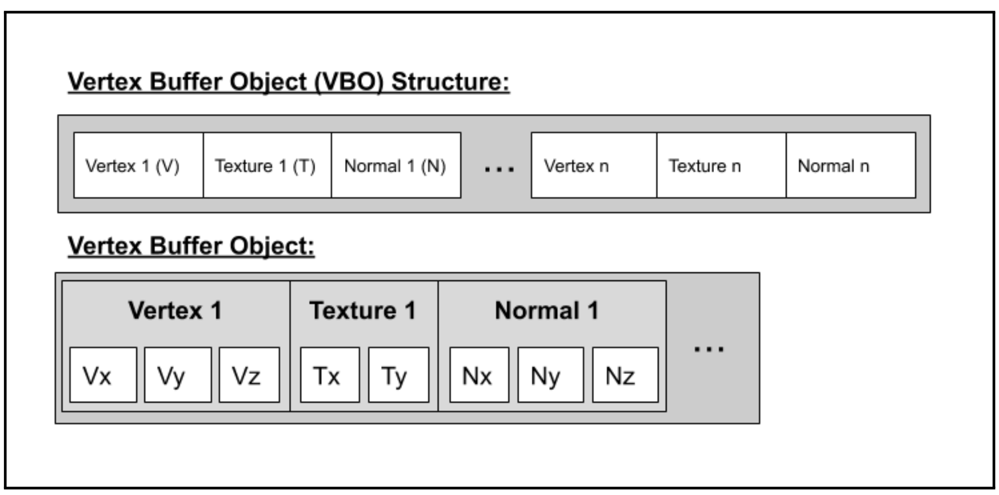


Figure 5.7: Diagram showing the structure of a vertex buffer object.

Buffer objects can be created not only for the plant branching structure but potentially for different parts of the plant, for instance, leaves or flowers. The leaves and flowers on a single tree tend to be very similar, so there is no need to have thousands of copies of a leafs' model or texture. This would be highly wasteful and unnecessary. Instead, there could be one copy of the vertex data, and texture data and instanced rendering can be used to render many copies of this single object in different places on the plant.

5.4 Shaders

5.5 Summary

Chapter 6

Physics Simulation

The motion of plants is a crucial factor when looking to create realistic looking plant-life. It has been a topic of discussion and research for many years now, particularly with regards to grass, bushes, and trees within video games. The movement is usually very subtle, but if it is missing, a scene can start looking very unnatural, making the user feel uncomfortable. This chapter will discuss a method of simulating the physical motion of plant-life, laid out by Barron et al. [Barron et al., 2001]. This method will be built into the parametric L-system itself in such a way that the L-system can provide the physical parameters for the simulation. This will allow a physics simulation to be run on any plant generated by the L-system.

The primary technique discussed by Barron et al. for simulating the motion of a system like a tree or a plant, is taken from that of a particle system, first described by Reeves [Reeves, 1983]. Particle systems can be applied to simulate phenomena like clouds, smoke, water, and fire. The main advantage of particle systems is that the motion for each particle can be updated simultaneously. This technique can be applied to the L-system representation of plant-life, where branches are split into segments that make up a skeleton of segments or joints. Each joint can represent a "particle" within the system, which has a dependency on all of its parent branches.

The particle system concept can be used to simulate the motion of the plant by having each joint within the plant skeleton provide some basic physical properties. These properties include but are not limited to the width, length, direction vector, spring consistent, and dampening constant. The direction vector is the global direction that the branch is pointing in 3D space. The spring constant and the dampening constant are used in Hook's Law calculations. The spring force of the branch tries to prevent it from bending. Whereas gravity, wind, and other forces generate torque, which generally acts against this spring force, causing the branch to bend.

This chapter defines a method of calculating the physical motion of a plant, where the properties can be defined in an L-system. The chapter starts by explaining how the volume, mass, inertia, and displacement of a given branch can be calculated. It then moves on to explain Hook's Law and its role within the simulation of plant movement. The equations of motion in a 3D setting are then explained. Finally, this chapter talks about the challenges faced with efficiency when updating branches and some of the results that can be achieved by using this method.

6.1 Branch Physical Properties

The mass of each branch segment can be simply calculated by taking the volume of each branch and multiplying it by the density of the wood or material. To do this the volume of each branch needs to be calculated. This can be done by multiplying π by the radius r squared and the length l as sees below.

$$v = \pi r^2 l \tag{6.1}$$

The volume of the branch segment is not often a clean cylindrical shape, particularly if the branch segment is decreasing in size. However, it gives a good indication as to the volume of the branch. The volume can now be used to calculate the mass. Calculating the mass also requires the density of the material. For instance the density of pine wood is between 400 - 420 kg/m^3. Some woods being less dense at about 200 kg/m^3, and other hardwood being as dense as 1000kg/m^3. The denser the wood, the higher the mass, and ultimately the greater its resistance to its change in velocity.

$$m = v \times d \tag{6.2}$$

The mass can be used to calculate each branch segments' moment of inertia. The moment of inertia is the branches' resistance to angular momentum. As the object is 3D, the shape of the object needs to be taken into account. Each branch can be seen as a long cylinder, which can be expressed in the following equation.

$$I = \frac{1}{3} m l^2 \tag{6.3}$$

Where I is the inertia of the branch, m is the mass, and l is the length. Similarly, an inertia tensor can be used for the sake of convenience to describe better the objects' rotational inertia, which is used within vector and matrix calculations. The inertia will be used when calculating the velocity of each segment in section 6.3. Below is an inertia tensor for a shape that is similar to that of a branch segment.

$$I = \begin{bmatrix} \frac{1}{12}m(3r^2 + l^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2 + l^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix} \tag{6.4}$$

The next vital piece of information needed is the direction that torque is acting on the branch (V), depending on the forces that are acting on it. The vector that represents the direction that the branch is pointing is known as the forward vector (v). The torque can be calculated by taking the cross product of the forward vector v and the force vector w. This can be visualised using the right-hand rule, where the index finger is the forward vector, and the middle finger is the force vector. The direction of the thumb then points in the direction of the torque. The angular velocity is produced as spin in the direction around the torque vector.

$$V = v \otimes w \tag{6.5}$$

The displacement is the change of angle of a branch from its resting position and is used to calculate the spring force of the branch in Hook's Law. The displacement can be calculated by keeping track of the starting local resting rotation of the branch p as well as its current rotation q in the form of two quaternions. The difference quaternion d is calculated by taking the local resting rotation p and multiplying it by the inverse of its current rotation q.

$$d = p \times q^{-1} \tag{6.6}$$

6.2 Hook's Law

Hook's law is a law of physics that states that the resultant force from compressing or extending a spring is equal to the product of the spring constant and the displacement of the spring. Each branch in a plant structure can be seen as a type of semi-rigid spring where external forces like gravity or wind bend the spring. Hook's law is used to calculate the reaction force due to the displacement of the spring. Hook's Law can be expressed in the equation below.

$$f = -k_s d + k_d v \tag{6.7}$$

Where f is the force exerted by the spring, k_s is the spring constant, and d is the total displacement of the spring. The dampening force can be calculated as $k_d v$ part where k_d is the dampening constant, and v is the velocity at the end of the spring or branch.

6.3 Equations of Motion

All of the forces such as gravity, wind, and spring forces can then be multiplied together to get the net force f_{net} acting on the spring. This is used to calculate the momentum of the branch, where T_{delta} is the change in time between physics calculations.

$$M = M_0 + f_{net} * T_{delta} \tag{6.8}$$

The velocity v is the current speed of the branch. In 3D graphics, the velocity is represented as a 3D vector and can be calculated by taking the inverse of the inertia tensor I and multiplying that by the momentum vector M.

$$v = I^{-1} * M Q_v = [0, v] \tag{6.9}$$

The velocity vector can be converted to its quaternion form Q_v in order to make the last step simpler. The scalar part of a quaternion can be set to 0, and the vector part can be set to v. This allows the next rotation quaternion R to be calculated. The last part involves taking the previous rotation quaternion, the velocity of the branch, and the change in time, to calculate the next quaternion rotation of the branch.

$$R = R_0 + (\frac{1}{2} * Q_v * R_0 * T_{delta}) \tag{6.10}$$

R is the next local rotation quaternion, R_0 is the previous local rotation quaternion, Q_v is the velocity quaternion, and finally, T_{delta} is the change in time since the previous physics update. This new rotation quaternion can then replace the current local rotation of the branch, in turn, simulating the motion of the branch.

6.4 Updating Branches

The particles in this system are the joints within the trees' skeleton. All of the joints have to be updated in each update step. The updates can happen as frequently as needed. A consideration is that if the branches are not updated frequently enough, the animations will not look smooth. Effectively each update step needs to take the forces acting on each branch, its current position, and rotation and calculate the next position and rotation of that branch. This information is then used to generate the model of the tree once again. This position and rotation are passed to the renderer, which will render the result.

6.5 Results

The result of having an L-system that contains the parameters for physics simulations is that it is straightforward to apply physics to almost any L-system. Whether it be for a 2D structure or a 3D one. The examples below show different types of L-systems and the rendered image both with and without gravity enabled. The acceleration due to gravity is kept at a constant value of $9.8m/s^2$.

#n = 6;
#define r 20; #define d 0.4; #define w 0.5;
#w : !(w)Z; (6.11)
#p1 : Z : * : F(d, 30.0)[-(r)Z]F(d, 30.0)[+(r)Z]-(r)Z;
#p2 : F(s, x) : * : F(s, x)F(s, x);

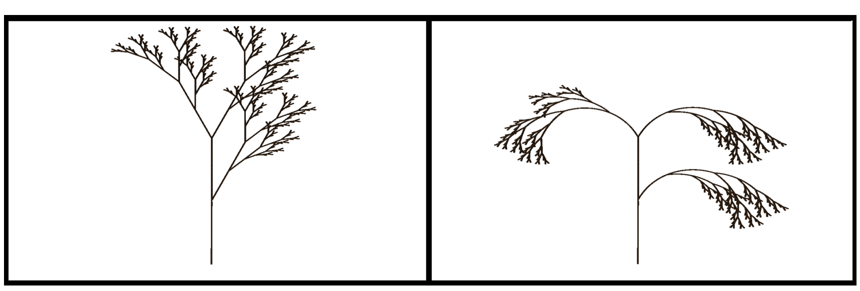


Figure 6.1: Examples simulating gravity on different 3D models

#n = 5;
#object F BRANCH; #object X SPHERE;
#define r 25.7; #define d 0.5; #define w 1;
#w : !(1.707)X; (6.12)
#p1 : X : * : F(d)[!(w)/(r)+(r)X][!(w)-(r)X][!(w)∧(r)X][!(w)&(r)X]!(w)F(d)X;
#p2 : F(s) : * : F(s)F(s);

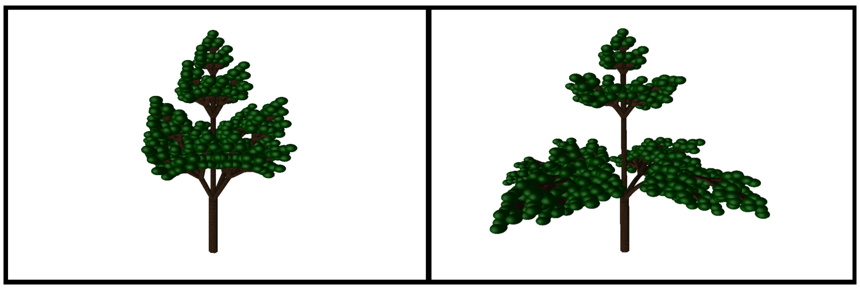


Figure 6.2: Examples simulating gravity on different 3D models

#n = 6;
#object F BRANCH; #object X SPHERE;
#define d1 112.5; #define d2 157.5; #define a 22.5;
#define lr 1.1; #define vr 1.4;
#w : !(1.4)F(2.0)/(45)A;
#p1 : A : * : !(vr)F(2)[&(a)F(2)AS(1)!(vr)]/(d1)[&(a)F(2)A S(1)!(vr)]/(d2)[&(a)F(2)A S(1)!(vr);
#p2 : F(l) : * : F(l*lr);
#p3 : !(w) : * : !(w*vr);

(6.13)

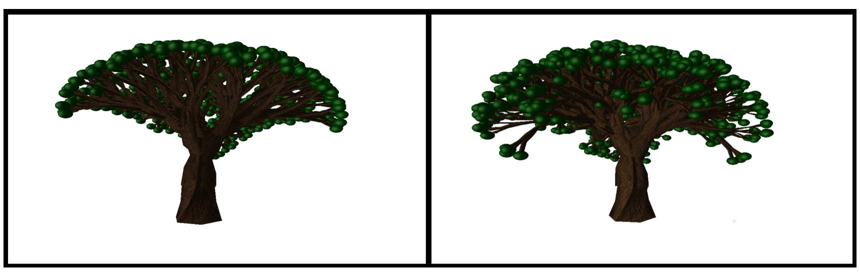


Figure 6.3: Examples simulating gravity on different 3D models

6.6 Summary

This chapter outlined a method of simulating and animating a procedurally generated plant by representing each branch as a particle in a larger particle system. The trees' skeleton provides the information about the location, rotation, dimensions, and properties of each branch. The skeleton is created during the interpretation stage, which is ultimately provided by the L-system. The properties of each branch can be simulated as a particle within the entire tree system. Due to the embarrassingly parallel nature of the system, each branch update can be computed in parallel, either on the CPU or GPU.

Chapter 7

Discussion

The relationship between the string rewriter and the string interpreter systems cannot be independent of one another. The rewriter is responsible for creating the overall structure and information for the interpreter. The interpreter takes this information and models it to the best of its ability. They are systems that share complexity. If one system is highly complex, the other one need not be as complex. The goal is to create a procedural generation system, where a user must specify a plants' structure using the L-system language. The L-system, and therefore the rewriter, cannot become so complex that it becomes unrealistically challenging to write an L-system for a plant. It is also not reasonable to create an L-system that is overly simplistic, like a DOL-system, because then the interpreter will need to assume certain features of the plant. It can be challenging to determine where the line of complexity should lie between the rewriter and the interpreter. Depending on the L-system being represented, there may be a need for emphasis on either side. The implementation contained within this thesis puts emphasis on providing a large amount of information in a way that is interpreter independent. This means that the information is provided through the parameters of modules and through declarations like the #object declaration. If more specific instructions are required, they can be provided through the use of the #object defined module, which will point to a meaning defined within the interpreter. This allows an L-system to provide specific information to the interpreter without the L-system dictating how it should be interpreted.

The advantage of this approach is the L-system grammar does not need to change, regardless of how it is interpreted. Additionally, the rewriting process is also kept independent of the interpretation. The interpretation of a particular module is defined by the interpreter, but can also be modified by the L-system using specific modifiers. For instance, the module name "F" can be interpreted as a turtle graphics instruction to move forward. However, the statement "#object F BRANCH" can be used to modify the meaning of this within the interpreter. Such that the meaning now suggests a move forward while also indicating that a BRANCH object should be rendered at that position. This makes it clear to the interpreter and the person writing the L-system, what is going to be rendered.

Chapter 8

Conclusions

This thesis aimed to explore the relationship between the two major systems within the procedural generation of plant-life using L-systems. These systems being the L-system rewiter and the interpreter.

Appendix A

Appendix

A.1 Appendix 1

A.2 Bibliography

Bibliography

[Backus et al., 1960] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1960). Report on the algorithmic language algol 60. *Numerische Mathematik*, 2(1):106–136.

[Baele and Warzee, 2005] Baele, X. and Warzee, N. (2005). Real time l-system generated trees based on modern graphics hardware. In *International Conference on Shape Modeling and Applications 2005 (SMI'05)*, pages 184–193. IEEE.

[Barron et al., 2001] Barron, J. T., Sorge, B. P., and Davis, T. A. (2001). *Real-time procedural animation of trees*. PhD thesis, Citeseer.

[Chomsky, 1956] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

[Cooper and Torczon, 2011] Cooper, K. and Torczon, L. (2011). *Engineering a compiler*. Elsevier.

[Eichhorst and Savitch, 1980] Eichhorst, P. and Savitch, W. J. (1980). Growth functions of stochastic lindenmayer systems. *Information and Control*, 45(3):217–228.

[GLFW development team, 2019] GLFW development team (2019). Glfw documentation. https://www.glfw.org/documentation.html.

[Gregory, 2014] Gregory, J. (2014). *Game engine architecture*. AK Peters/CRC Press.

[Haubenwallner et al., 2017] Haubenwallner, K., Seidel, H.-P., and Steinberger, M. (2017). Shapegenetics: Using genetic algorithms for procedural modeling. In *Computer Graphics Forum*, volume 36, pages 213–223. Wiley Online Library.

[Juuso, 2017] Juuso, L. (2017). Procedural generation of imaginative trees using a space colonization algorithm.

[Koch et al., 1906] Koch, H. et al. (1906). Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta mathematica*, 30:145–174.

[Kókai et al., 1999] Kókai, G., Ványi, R., and Tóth, Z. (1999). Parametric l-system description of the retina with combined evolutionary operators. *Banzhaf et al.[3]*, pages 1588–1595.

[Lindenmayer, 1968] Lindenmayer, A. (1968). Mathematical models for cellular interaction in development, i. filaments with one-sidedinputs, ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18:280–315.

[Lindenmayer, 1971] Lindenmayer, A. (1971). Developmental systems without cellular interactions, their languages and grammars. *Journal of Theoretical Biology*, 30(3):455–484.

[Mandelbrot, 1982] Mandelbrot, B. B. (1982). *The fractal geometry of nature*, volume 2. WH freeman New York.

[Movania et al., 2017] Movania, M. M., Lo, W. C. Y., Wolff, D., and Lo, R. C. H. (2017). *OpenGL – Build high performance graphics*. Packt Publishing Ltd, 1 edition.

[Prusinkiewicz, 1986] Prusinkiewicz, P. (1986). Graphical applications of l-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253.

[Prusinkiewicz and Hanan, 1989] Prusinkiewicz, P. and Hanan, J. (1989). *Other applications of L-systems*. Springer New York, New York, NY.

[Prusinkiewicz and Hanan, 1990] Prusinkiewicz, P. and Hanan, J. (1990). Visualization of botanical structures and processes using parametric l-systems. In *Scientific visualization and graphics simulation*, pages 183–201. John Wiley & Sons, Inc.

[Prusinkiewicz and Hanan, 2013] Prusinkiewicz, P. and Hanan, J. (2013). *Lindenmayer systems, fractals, and plants*, volume 79. Springer Science & Business Media.

[Prusinkiewicz and Lindenmayer, 2012] Prusinkiewicz, P. and Lindenmayer, A. (2012). *The algorithmic beauty of plants*. Springer Science & Business Media.

[Reeves, 1983] Reeves, W. T. (1983). Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions On Graphics (TOG)*, 2(2):91–108.

[Sellers et al., 2013] Sellers, G., Wright Jr, R. S., and Haemel, N. (2013). *OpenGL superBible: comprehensive tutorial and reference*. Addison-Wesley.

[Smith, 1984] Smith, A. R. (1984). Plants, fractals, and formal languages. *ACM SIGGRAPH Computer Graphics*, 18(3):1–10.

[Stroustrup, 2000] Stroustrup, B. (2000). *The C++ programming language*. Pearson Education India.

[Torvalds,] Torvalds, L. Git documentation. https://git-scm.com/doc.

[Vaario et al., 1991] Vaario, J., Ohsuga, S., and Hori, K. (1991). Connectionist modeling using lindenmayer systems. In *In Information Modeling and Knowledge Bases: Foundations, Theory, and Applications*. Citeseer.

[Wilhelm and Seidl, 2010] Wilhelm, R. and Seidl, H. (2010). *Compiler design: virtual machines*. Springer Science & Business Media.

[Wilhelm et al., 2013] Wilhelm, R., Seidl, H., and Hack, S. (2013). *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media.

[Worth and Stepney, 2005] Worth, P. and Stepney, S. (2005). Growing music: musical interpretations of l-systems. In *Workshops on Applications of Evolutionary Computation*, pages 545–550. Springer.

[Yokomori, 1980] Yokomori, T. (1980). Stochastic characterizations of eol languages. *Information and Control*, 45(1):26–33.