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

Abstract

Contents

List of Figures

List of Tables

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 produral 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 own 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, algea and grasses, whilst still being applicable to non-organic structures such as music, artificial neural networks and tiling patterns [Prusinkiewicz and Hanan, 1989].

The L-system in its most basic form, is a formal grammar which contains a set of symbols or letters that belong to an *alphabet*. The alphabet is used to create a starting string known as the *axiom*, as well as a set of production rules. The production rules are applied to each symbol within the axiom string, each rule dictates whether or not the symbol can be rewritten and furthermore, what they will be rewritten with. In essence, a L-system uses the set of production rules to generate a resulting string of symbols which follow those production rules. The resulting strings' meaning can then be interpreted in a way that best fits what it is trying to represent. In this case the string can 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 structure 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 behaviour of the plant that it generates, thus making it possible to simulate external forces such as gravity and wind.

This chapter will describe the motivations behind this research and how it can be used to improve the procedural generation of plant-life in 3D applications. It will then introduce the concepts of procedural generation, rewriting systems and formal grammars. Briefly describing how procedural generation can be applied to the development of plant-life through the use of rewriting systems. Furthermore, this chapter 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, and how this research will be conducted.

1.1 Motivations

L-systems have been talked about and researched since its inception in 1968 by Aristid Lindenmayer. Over the years it's usefulness in modeling different types of plant life has been very clear, however its presence has been quite absent from any mainstream game engines and graphics applications for the most part, these engines relying either on digital artists skill to develop individual plants or on 3rd party software such as SpeedTree. These types of software use a multitude of different techniques however their methods are heavily rooted in Lindenmayer Systems.

One of the most time consuming parts for digital artists and animators is creating differing variations of the same piece of artwork. In most games and other graphics applications environment assets such as trees, plants, grass, algae and other types of plant life make up the large majority of the assets within a game, and creating a plant asset can take a skilled digital artist more than an hour of work by hand, The artist will often have to create many variations of the same asset in order to obtain enough variation that a user of that graphics application would not notice that the asset has been duplicated, if this is multiplied by the number of assets that a given artist will have to create or modify, there is an incredible number of hours that could have potentially been put to use creating much more intricate assets. In addition to this, it is also important to note that graphics assets are then stored in large data files, describing the geometry and textures and other information. If we require three very similar plants, we have to store three separate sets of data. Procedurally generating plants can avoid this wasteful data storage entirely, instead a relatively small L-system description can be stored which can be used to procedurally generate all the required geometry and other information during the execution of the program.

The L-system can not only procedurally generate the geometry of the plant-life but can also generate parameters physical properties of the plant inself such as the weight and flexibility of branches as well as its wind resistance and many other important information that can be used to simulate or animate the motion of the plant under various forces.

1.2 Introduction to Procedural Generation

Procedural generation is used in many different areas and applications in computer graphics, particularly when generating naturally occuring 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 generate 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 a different types of plant-life, such as flowers. The L-system on the other hand relies on a mothod of string rewriting, whereby the rewriting is based on a set of production rules in order to generate a string of symbols that obay those rules. A separate system can later interpret this string to create the model. The L-system procedural generation therefore, has two separete 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 obvious traits and features. For instance, a palm tree has long stright trunks with long compound leaves exclusively near the top, branching in all different directions. Comparatively a pine tree has a long staight trunk with many branches coming off in different directions pupendicular 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 three dimensional model. Whilst allowing for randomness and variety within the generation process, such that variations of a particular species can be generated 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 that is other-wordly.

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]. Using these state transitions it is possible to generate complex structures by successively replacing parts of a initial simple object with more complex parts. Rewrite systems can be non-deterministic, meaning that there could be a transition which depends on a condition being met or on a neighbouring states. Using this rewriting concept any preceeding state can rely upon some conditions neccessary for transformation. If condition is true the state will be rewritten, otherwise it will remain the same, and will be 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].

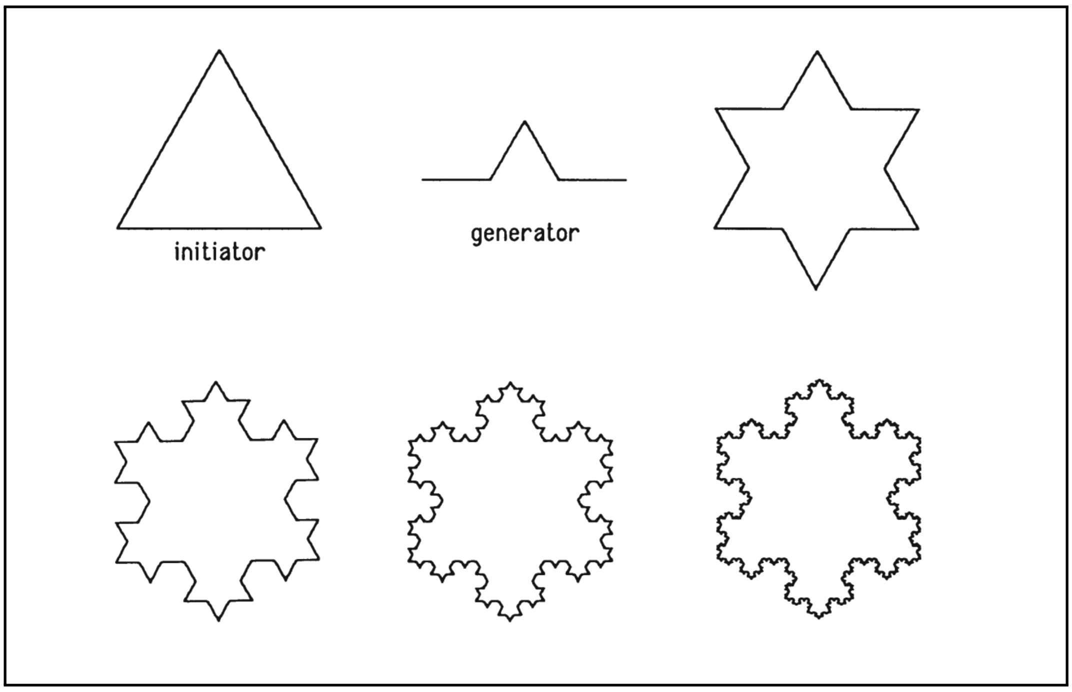


Figure 1.1: Construction of the snowflake curve[Prusinkiewicz and Hanan, 2013].

The snowflake curve starts with two parts, the initiator and the generator. The initiator is is the initial set of edges forming a certain shape, whereas the generator is a set of edges which 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 each edge is again replaced by the generator. 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 they are instead represented by a set of symbols or strings.

1.4 Introduction to Formal Grammars

In the context of computer science, grammars are 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 therefore 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 in order to gain an expected result. The challenge is therefore to create a grammar in the form of a rewriting system that facilitates the procedural generation of plant-life. A rewriting system such as the

L-system operates in a way that is consistant 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, there are two types of L-system grammars that 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, it is possible for a 1L-system to represent any 0L-system, therefore, 1L-system languages tend to be more complex and verbose when compared to 0L-systems, this creates a trade off between a more powerful and complex language or a less powerful but simpler language.

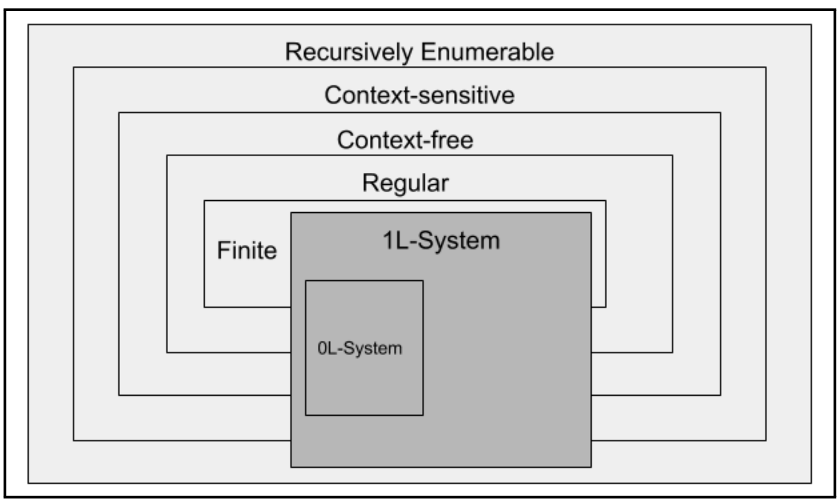


Figure 1.2: Diagram of the Chomsky hierarchy grammars with relation to the 0L and 1L systems generated by L-systems.

1.5 Structure of Thesis

This thesis is split into three major parts. Part 1 focuses on the L-system itself, it defines the various types of L-systems for modeling plant-life, the concept of a parametric L-system as well as some techniques for definiting randomness and stochasticisity within an L-system in order to create variaty. Part 2 talks about the L-system rewriter implementation, covering how the L-system generates the resulting string and structures relavant rendering. Part 3 focuses on the interpretation and renderer implementation and how to render a convincing model of a tree on the screen, as well as simulation and animation of the generated plants.

Chapter 2

Lindenmayer Systems

The L-system at its core is a formal grammar made up of an *alphabet* of characters which are concatenated together into collections of symbols, called strings. The L-system describes a starting string called the *axiom*, and a set of production rules. For each rewriting step the production rules determine whether a symbol within a string should be rewritten with another symbol or string. Each symbol within the *axiom* is matched against the production rules. If a match is found, the symbol within the axiom is replaced with a predecessor string described by the production rule. This process is carried out for each symbol in the *axiom.* The resulting string created by the rewriting process then becomes the axiom, which can then be rewritten once again. This process of rewriting using production rules is the mechanism for generating a structure of states that obay the production rules, similar to that of a context-free grammar. Essentially the symbols represent a particular state of the system, and the production rules decide whether that state should transition based on a certain criteria, and what the next state should be.

This chapter will discuss a number of different types of L-systems, as well as their features and limitations. It will focus on the mechanics behind the rewriting system and different techniques that can be used to better represent plant-life as an L-system. In order to provide sufficient background, this chapter will also touch briefly 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 important to note that the L-system itself has no concept of what it is trying to represent, it is simply a string rewriting system. The L-systems interpretation is left up to a separate system, responsible for interpreting the resulting string to create a suitable representation for that problem domain. For instance, the symbols for a L-system trying to represent a tree, may be interpreted very differently to the symbols trying to represent music, however, the L-systems may be identical. Although the interpreter is not neccessarily part of the L-system it is an important to understand the reliance of the L-system on the string interpreter. The string interpreter will be in great detail in chapter 5.

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 orgamisms such as algae and bacteria [Lindenmayer, 1968]. He later defined a formal grammar for simulating multicelular 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 non organic structures like music [Worth and Stepney, 2005].

There has also been studies to try to use an L-system as a method of creating and controlling 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 the human retina, this can be combined with evolutionary operators and be 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-systems is known as the D0L-system. The term 'D0L system' abbreviates 'Deterministic Lindenmayer system with zero-sided interactions'. It is deterministic as each symbol has an associated production rule and there is no randomness in determining which rule should be chosen. 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 direct 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*), the starting string or (*axiom*) and the state transition rules (*rules*). The alphabet is a set of characters which represent a particular state in a system. The starting string or *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 was originally created to serve as a context-free grammar, to represent the development of multicellular organisms. In the DOL-system 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 will be produced [Prusinkiewicz and Lindenmayer, 2012].

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

With the definition above, the DOL-system states $w : a_r$, the symbol w signifies that what follows is the starting point (axiom), therefore, the starting point is the cell a_r. The production rules then follow and are $p1, p2, p3$ and $p4$. The : symbol separates the axiom and production names from their values, furthermore the $\rightarrow$ can be verbalised as "is relaced by" or "rewritten with". In production rule 1 ($p1$) the cell a_r will be rewritten with cells $a_l b_r$, $p2$ states that a_l will be rewritten with cells $b_l a_r$, $p3$ 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. In order to simulate

Anabaena catenula we require these four rewritting rules, as there are four types of state transitions.

The resultant strings of five generations of the DOL-system rewritting process:

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

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 next symbol or a string that is specified by the production rule. For G_1 the previous generation resultant string is taken, which in this case is G_0, being a_r, the first symbol is compared with the production rules. In this case it matches rule $p1$ with the rule $p1 \;:\; a_r \;\rightarrow\; a_l b_r$ and therefore, a_r is rewritten with $a_l b_r$. 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 a if the symbol matches, then that symbol will be 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 do not supply very much information, for instance, how does the D0L-system indicate how many times a given string has been rewriten? The D0L-system is also deterministic, meaning that there is no randomness the rewriting process, therefore, it will always yield the same result with no variation.

2.2 Interpreting the DOL-system

Section 2.1 outlines a simple type of L-system known as the DOL-system, this L-system specifies a set of symbols, a starting point and a set of production rules, allowing us to represent a problem as a set of states. The production rules can express valid state transisions, which thereafter allows us to produce a resultant string of symbols that obey the L-systems production rules. This functionality is powerful in and of itself, however, the L-system's symbols are only useful if they represent some kind of meaning, furthermore the L-system does not supply this meaning, each symbol's meaning is interpreted after the rewriting process of the L-system by the interpreter. Due to this, there are two separate and very different systems involved in taking an L-systems input, such as the alphabet, axiom and production rules and turning it into something that is able to model plant-life. These two systems are the L-system rewriter, which is reposponsible for using L-system to rewrite a string and provide a resultant string of symbols. The L-system interpreter takes the resultant string from the L-system rewriter and interprets it in a way that is able to represent the model we are trying to create.

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 which 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 2.2. The turtle starts at the base or root of the tree and interprets a set of rotation and translation movements, which when all executed one after the other, trace the points which make up the plant 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 instruction symbols and their meaning to the interpreter

In the OL-system there are a number of symbols that 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 5.1.

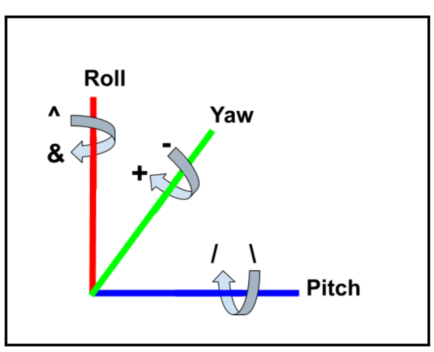


Figure 2.1: Diagram of the 3D rotations of the turtle.

The turtle instructions are presented in such a way that allows movement in three dimensions, the rotations in yaw, pitch and roll, where yaw is a rotation around the Z axis, pitch is rotation around the X axis and roll is rotation around the Y axis. We have two symbols for each rotation, which represent positive and negative rotations repectively. 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 translation to translation, and subsequent rotations are concatenated to maintain a global orientation, in this way when the turtle moves forward again, it will move in the direction of this global orientation. Diagram 3.2 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 5.1, can be used as the alphabet for the rewriting system in the 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 relevent to the rewriting system, so we can use the axiom and production rule to rewrite by one generation. The only piece of information which is relevant to the interpreter is the angle to rotate by when it comes across the symbols + and - this is specified in the definition of the L-system with Angle: 90°. The resulting string would be "F+F-F-F+F", this string is passed to the interpreter system which uses turtle graphics to execute a list of instructions. These instructions can be articulated in the list 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 meaning for the L-system 2.3

These instructions are carried out one after the other, moving the turtle around the screen in three dimensions, furthermore, tracing the structure which the 0L-system has generated, these instructions will generate the traced line shown in figure 2.2.

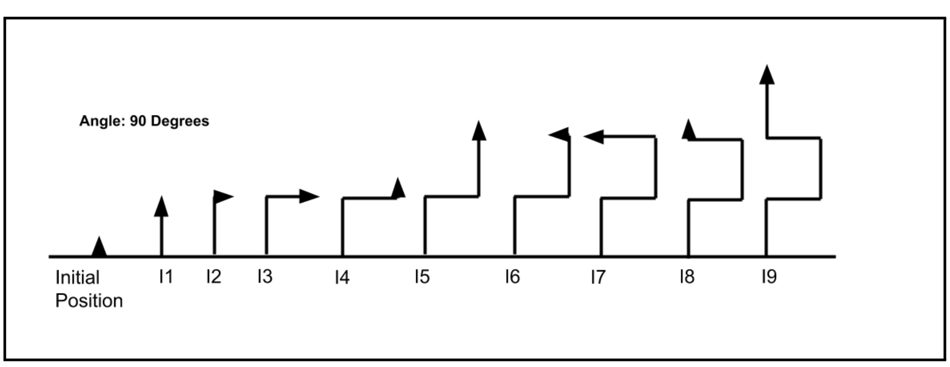


Figure 2.2: Diagram showing a turtle interpreting simple L-system string.

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 telling 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 a number of assumptions which the interpreter makes in order 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, as such it must be hardcoded into the interpretation, or assumed by some other means. This highlights one of the 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 is able to supply a large amount of information to the interpreter, the interpreter can be quite rudimentary and follow the instructions exactly. Conversely, we could have a system where the L-system rewriter is quite basic, but the interpreter is very complex and must be capable of representing the L-system despite the lack of information in the resultant string, or be able to obtain this information by other means.

It may be tempting to leave the complexity to the interpreter in order 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 even the type of objects that need to be represented have to be supplied to the interpreter in some way, and if not through the resulting string of information, how is this information meant to 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 which decrement in diameter, branching angles which are consistant and other aspects. This results in a very inflexible system which 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 that the L-system is responsible for some of the details of the interpretation such as angles, branch diameters and so on. In the next few sections, different types of L-systems will be discribed, explaining their benefits and limitations, as well as developing a system intergrating these separate systems into a single L-system grammar.

There are a number of fractal geometry that have become well known particularly with regards to how they can seemingly imitate nature [Mandelbrot, 1982]. Particularly with the geometry such as the Koch snowflake which can be represented using the following L-system.

2.3 Branching

The previous section covered a very basic 0L-system, which was capable of tracing a 3D pattern, the 0L-system allows us to move around a 3D space, however we are not able to branch off in two or more directions as plants do. Lindenmayer introduced two symbols which make branching much easier [Lindenmayer, 1968]. These are the square bracket commands '[', ']'. The square bracket characters instruct the turtle object to save its current position and orientation for the purpose of being able to go back to that saved position and orientation later. This allows the turtle to jump back to a previous position, facing the same direction as it was before. We can then change orientation and branch off in a different

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

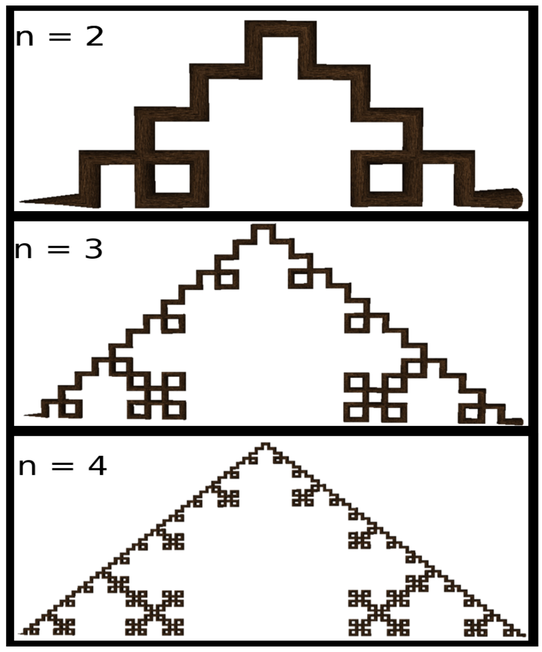


Figure 2.3: Koch Curve.

Sierpinski Triangles:
Generations: 4
Angle: 60°
Distance: 1 cm
ω : F
$p1$: F $\rightarrow$ X-F-X
$p2$: X $\rightarrow$ F+X+F

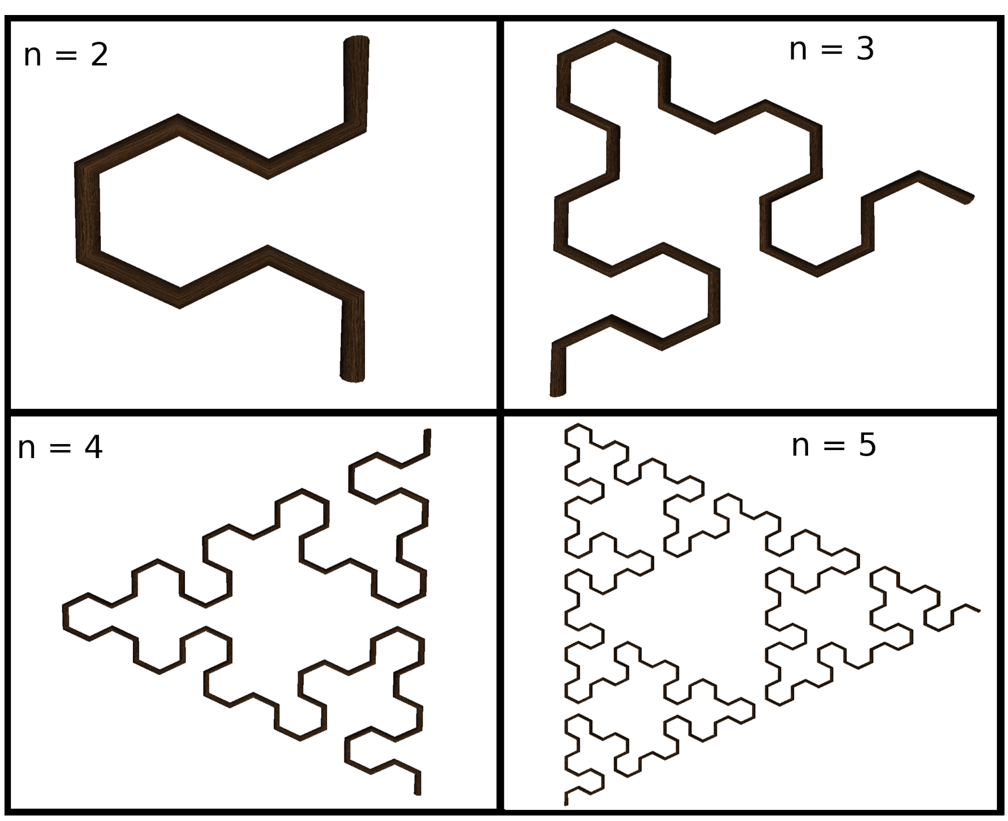


Figure 2.4: Sierpinski Triangles.

direction. This was originally used by Lindenmayer to develop the branching that occurs in algea, this idea was later used to represent plant-life by Smith [Smith, 1984].

The same method of interpreting the L-system string is applied, the symbols [and] are used in order the translate back to a previous position and orientation in order to branch off in a different direction. Each save state symbol [must have a corresponding load state symbol within the string. In order to have a branch off of another branch we can have nested save and load state symbols. For instance the resultant string "F[+F-F]-F" branches off the main branch once as seen in figure 2.5. Additionally using nested save and load states in the string "F[+F[+F]-F]-F" we are able to branch twice which can be shown in figure 2.6.

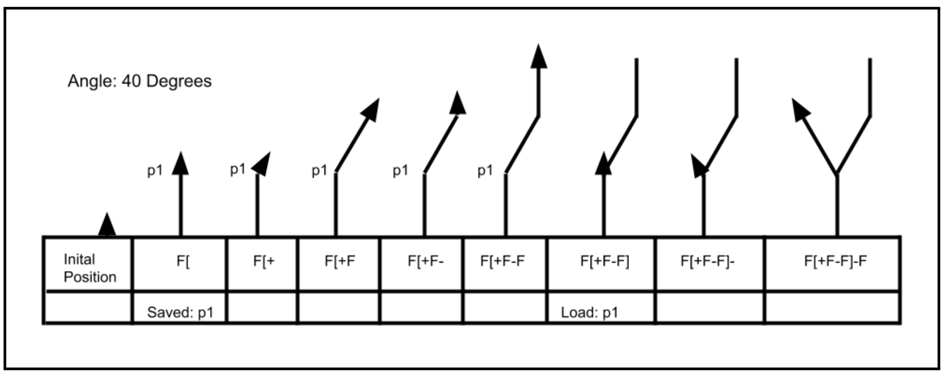


Figure 2.5: Diagram showing a turtle interpreting an L-system using the branching symbols.

Save and load operations are handled using the Last In First Out (LIFO) principle, meaning that when the save symbol is used [the current position and orientation at $p1$ is saved, the next load state] will restore $p1$'s position and orientation, unless another save takes place in which case that save will have to be loaded before $p1$ can be loaded. In this way, the position saves are stacked and the most recent save is loaded. This can be seen in figure 2.6 where we have nested branching.

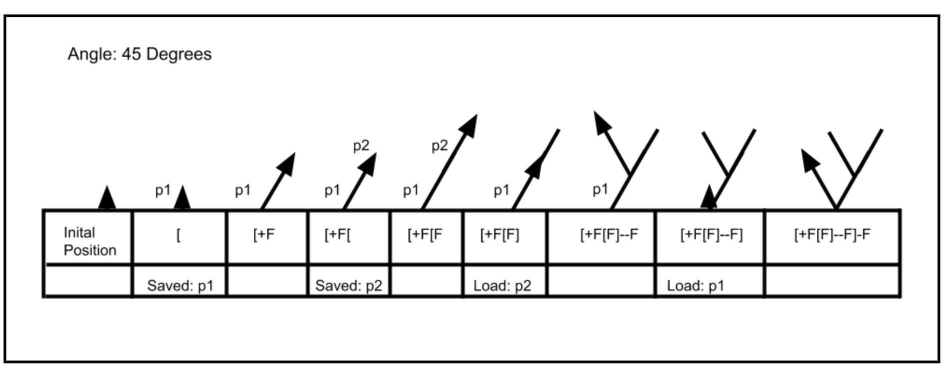


Figure 2.6: Diagram showing a turtle interpreting an L-system with nested branching.

The main advantage to using the save and load position functionality as a symbol within the alphabet of the L-system is that the rewiting system itself handles the branching. There is often no production rule for the save and load symbols and thus the symbols remain consistent from generation to generation.

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

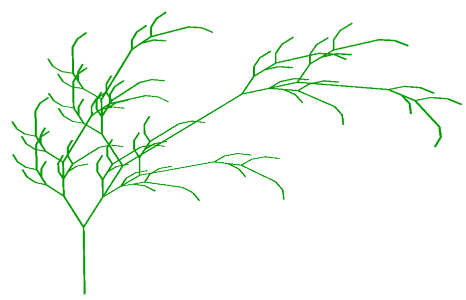


Figure 2.7: Fractal Plant.

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

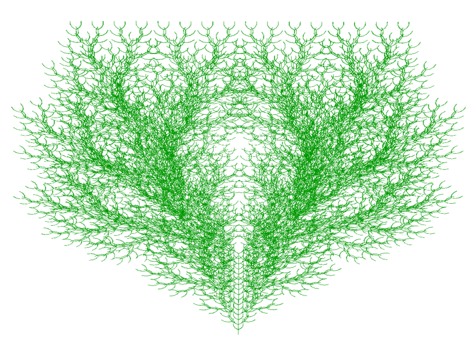


Figure 2.8: Fractal Bush.

2.4 Parametric OL-systems

Simplistic L-systems like the algae representation in section 2.1 above, give us enough information to create a very basic structure of plant life, there are many details that are not included which a simple OL-system will not be able to represent. With the simplistic approach we have assumed that the width and length and branching angles of each section is constant or predefined. The result of this was that all of the details such as width and length of branches is left up to the interpretation of the resultant L-system string. This begs the question as to how we should accurately interpret the L-system string when we are not provided the details by the L-system. The answer lies in parametric 0L-systems.

In this section I will outline the definition and major 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]. I will also be talking about some of the changes that I have made, and explaining why these changes are necessary for the purpose of 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 has a number of parameters associated with it. The module names belong 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 and $!=$ is the 'not equal' operator, and the symbols $>=$ and $<=$ are 'greater than or equal to' and 'less than or equal to' respectively. The parentheses () are also used in order to 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 will 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 ordererd 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 more 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" as well as the #include statement which specifies what type of shape to draw by refering to a library of predefined shapes [Prusinkiewicz and Lindenmayer, 2012]. For instance if we have an value for an angle that we would like to use within the production rules we can use the #define statement as follows:

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

Here you can see that the #define acts like a declaration, where we are going to be defining a variable which will be used later. Essentially we are replacing any occurences 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 would tell 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 hetrocist 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:

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

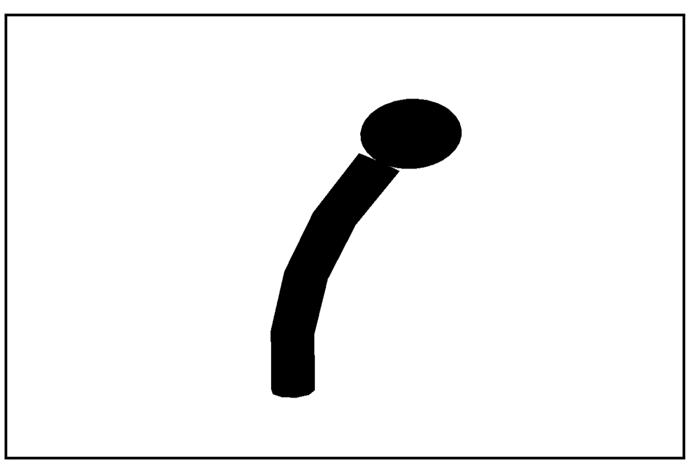


Figure 2.9: Diagram of an L-system Using Multiple Objects.

In the simple example in figure 2.7 above, you can see that the first three F modules render a branch segment with 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 bares no meaning to the rewriting system, it simply 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, this will be discussed more detail chapter 5 where the turtle graphics interpreter and renderer is defined.

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 I have not discussed yet, those 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 ‘ will still perform the same operation, however the ! and ‘ symbols are actually treated as a module that holds a 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 in a similar manner. 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.

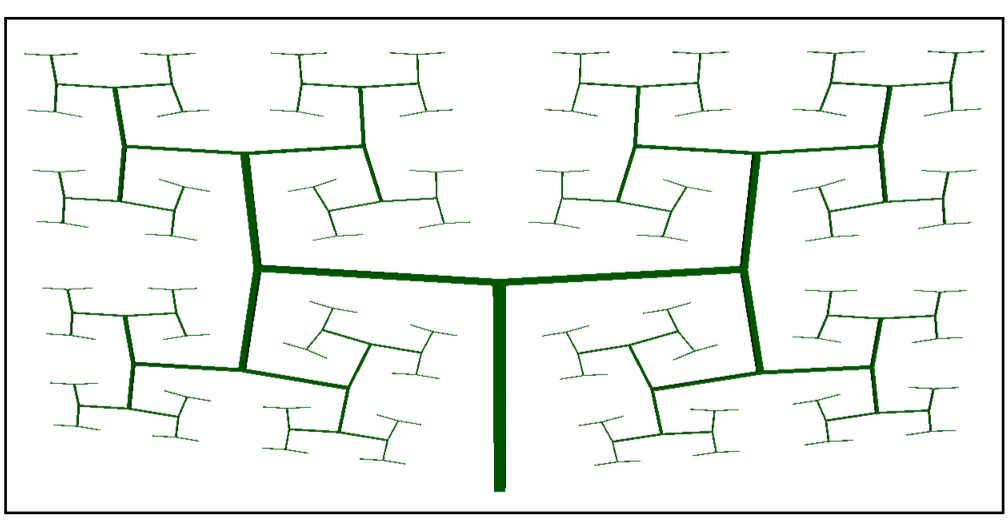


Figure 2.10: 3D Parametric L-system.

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 us to have multiple production rules that are the same in terms of the module name and the number of parameters that they have, furthermore, they require a particular condition to be met in order for the module to match that rule.
In this section I will be detailing the use of the condition statement, which lies between the predecessor and the successor in a production rule, and can be seen as an 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 is true then that rule is selected for rewriting, if the result is false then it moves on to check the next rule.

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

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

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 rewritting process can be seen below in 2.10.

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

A practical use of the condition statement might be to simulate different stages of growth. This 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

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

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

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

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

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

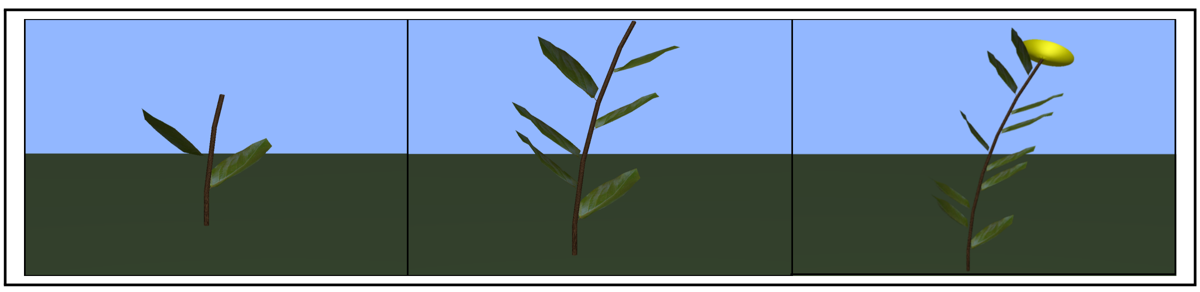


Figure 2.11: Condition statements used to simulate the growth of a flower. 2nd generation on the left, 4th generation in the center and 6th generation on the right

2.5 Randomness within L-systems

Randomness is an essential part of nature. If there was no randomness in plant life, it would end up with very symetric and unrealistic. Randomness is also responsible for creating variation in the same L-system. A L-system essentially describes the structure and species of a plant. It describes everything from how large the trunk of the tree is, to how many leaves there are on the end of 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 we will always end up with the exact same structure. Below is a simple example of how randomness can be used to create variation.

$n = 2$

#define r 25

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

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

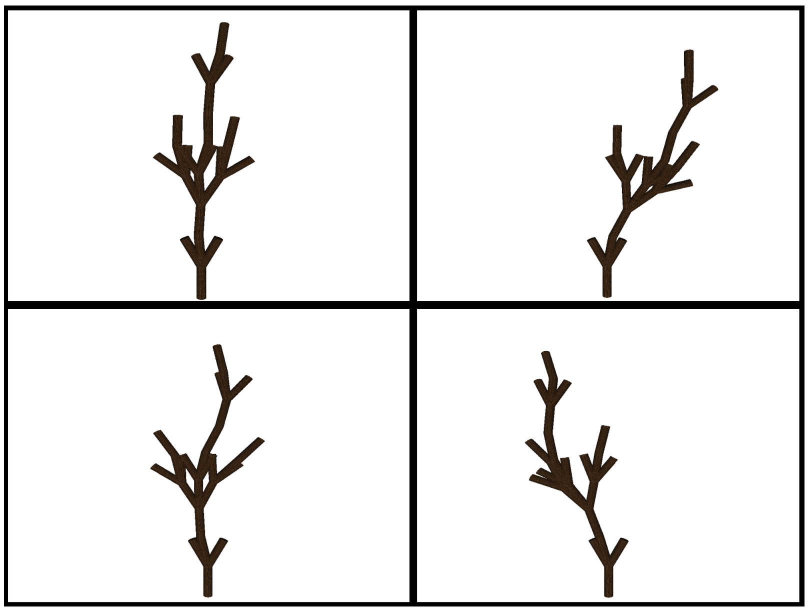


Figure 2.12: Different Variations of the Same L-system with Randomness Introduced in The Angles.

In figure 2.12 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 what is contained is a random number range, ranging from the minimum value as the first floating point value and the maximum value as the second floating point value separated by a comma. If both values are the same for instance +({10.0, 10.0}) this is equivilant to +(10.0).

2.6 Stochastic Rules within L-systems

Similar to the previous section about randomness in L-systems, stochastic L-systems fulfill a similar goal. 0L-systems on their own are incapable of creating any variation, they simply follow a strict set of production rules which gives the same result. Introducing randomness to an 0L-system for width, length and other parameters can result in a plant that looks slightly different but does not change to overall structure of the plant or any branching. In order to create a different structure for a plant we must introduce stochastic probability within the selection of the production rules, thus effecting the rewriting of the structure itself.

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 in order to select the starting point at the start of the rewriting process. Secondly, the S0L-system allows you to define a finite number of production rules which have a probability distribution in order 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. In this paper, I 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 neccessary.

Similarly to the 0L-sysstem, 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 probabilies 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 an 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 you can see the module F(x) above, is the predecessor for all three of the production rules, each rule has a probability which is defined using the $\sim$ symbol followed by probability from 0 to 1. In the above example each probability isapproximately one third, they are approximate in order to total a an exact probability of 1.0. During the rewriting process, when the module F with one parameter is found, a production rule is randomly selected using the probability distribution described within the production rule. The predecessor from the selected rule with then rewrite that module.

Figure 2.13: 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 2.13 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. This gives a different number of translation instructions, can result in the plants having branches of different lengths, for example $p1$ has two extra F instructions. Resulting in some branches being much longer than others, as well as possibly producing plants of different sizes.

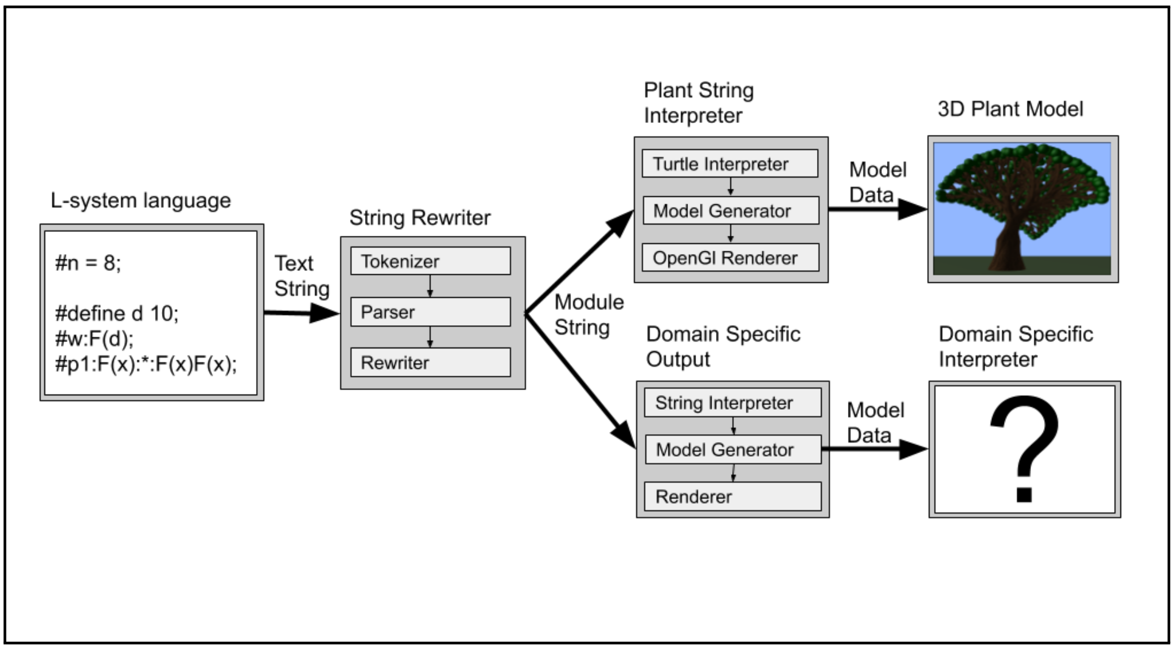


Figure 2.14: Diagram of the procedural generation process.

2.7 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 symbols or modules within an L-system do not change the behaviour of the L-system but matter only to the interpreter. Additionally the complexity of the L-system rewriter decides the complexity of the interpreter, if a L-system provides a large amount of information to the interpreter, less assumptions need to be made during the interpretation and therefore, providing the able to more accurately describe the plant-life it is representing.

By using the parametric 0L-system we can build in a number of 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 as well as branch lengths, branch widths and production rule conditions can give control over stages of growth.

Chapter 3

L-system Rewriter Implementation

There are two major parts necessary to procedurally generate plant-life using L-systems. These are the L-system rewriter and the L-system interpreter. The L-system rewriters' purpose is to take a L-system file as input. The input is read, processed and understood generating the necessary structures and information in order to carry out the final rewriting process. This eventually gives a resulting string of modules which can be passed to the interpreter. The interpreter goes through each module and processes it as an instruction for the turtle graphics interpreter. Which will eventually result in a model of a plant on the screen.

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 single characters. As the D0L-system is deterministic, there is no randomess when determining which rule matches. It is therefore quite easy to create a rewriting system for the D0L-system. All the rewriter needs to do is store the starting string and production rule predecessors and successors. And then iterate over a string of symbols and replace them with the successor. This can be implemented relatively simply. Conversely, the implementation of a more sophisticated parametric 0L-system is much more complex. For example 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. For the rewriting system to rewrite a complex system such as this, it needs to better understand what each part of the L-system is specifying, based on each symbols context within the L-system.

This chapter will focus on the each part of the string rewriters' implementation and will introduce the technique of processing the L-systems' input, similar to how computer languages are compiled. Due to the complexity of the L-system grammar, it is difficult for a computer tell the syntactic and semantic properties of the each part of the L-system input, which in turn makes it difficult to carry out the rewriting process. Using a system similar to a "compiler" to process an L-system means that a L-system "program" can be broken down into a three stage process, as seens in figure 3.2 below. The first stage is the *lexical analysis* of the L-system input, then a process called *parsing* and finally the string rewriting stage. The lexical analyser is responsible for splitting the input into words and then assigning the word into its syntactic category. Any word within the L-system that does match any syntactic category will result in a lexical error. Given there are no errors the words and their syntactic categories are then sent to the parser, which 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 occured and what was grammatically incorrect. The parser also creates a syntax tree as well as any data structures neccessary for the rewriting process. These structures can then be used to carry out string rewriting. String rewriting is the final stage and will eventually provide the resultant string to the interpreter.

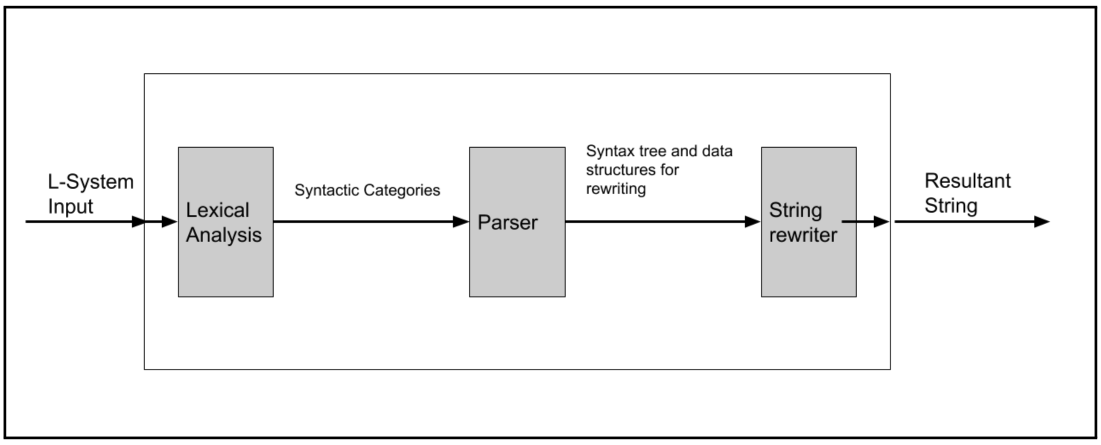


Figure 3.1: Diagram of the Parts of The Rewiting System.

3.1 Environment and Tools

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

The implementation for this thesis will be using the modern Open Graphics Library otherwise known as OpenGL. The OpenGl framework is one of the industry standards for creating 3D graphics applications, and is a cross platform API for interacting with the GPU in a low-level way. The high performance nature of OpenGL is important as displaying and simulating the L-system can be very graphically intensive [Sellers et al., 2013] [Movania et al., 2017]. OpenGL was originally intended to be an API for the C and C++ programming languages, and therefore we can have a programming language and graphics API which have a strong emphasis on performance.

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. Furthermore, 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]. In order to keep track of changes and manage versions Git is a free and open source version control software. It is is able to 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. Git can be used in conjuction with Github, which is a online web application that stores git repositories. This 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, where 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 inspection of the program as a whole, such as the set of declared variables in a block, a function, etc [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 I will refer the system containing the lexical analyser, L-system parser and the string rewriter as the L-system rewriter, instead of an interpreter in the computational sense.

A similarity can be drawn between traditional 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 a number of times. Later on, it may be decided that, instead of five generations of rewriting, the rewriter should instead generate ten rewrites. 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 in order to interpret to the tenth generation instead. This can be used to go from the current state of the L-system rewriter and just 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 simply be asked for from the L-system rewriter.

The lexical analyser and parser are a neccessary part in order to carry out rewriting. Without them it would be difficult to find the syntactic roles of each part of the L-system, take the module: F(2*3, x * (2 + y)) as an example, 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 structure of the grammar 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 component parts and is describes each parts syntactic roll. This provides the understanding that x and y are variables within a module and do not represent another module, as well as where the values of x and y could be found.

The trade off of creating an L-system with more complexity within the grammar itself is that it become more difficult to write a valid L-system to represent a particular structure, the advantage of using an rewriter specifically designed for a CFG like the parametric 0L-system grammar is that it can make it simpler to debug any syntactic errors within an L-system, as well as make the string rewriting much faster. This means that writing a L-system becomes simlar to rewriting a recursive program and any syntactic mistakes made in writing this "program" results in a meaningful error describing what was incorrect.

3.3 The Syntax of a Parametric L-system

This section will cover the valid syntax for the parametric L-system rewriter, the syntax for the parametric 0L-system is similar to the definition of the L-systems given by Prusinkiewicz and Lindenmayer in section 2.4.1, this is to keep consistant with how most L-systems are defined. There are some additions and modifications to the syntax definition provided by Prusinkiewicz and Lindenmayer in order to construct a L-system that includes branching, constant variable definitions, object specifications, parametric L-system concepts, randomness as well as stochastic L-systems [Prusinkiewicz and Lindenmayer, 2012].

This parametric 0L-system is made up of five major parts, each part can be catagorised as a statement, these statements are the define statements, include statements, a single generation statement, a single axiom statement and a one or more production rule statements [Prusinkiewicz and Hanan, 2013]. All of these statements collectively form a parametric 0L-system. Each statement starts with a # character and ends with a ; character, this is useful to the lexer and parser and allows two statements to be written on the same line.

The order of statements should be listed as follows:

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

The order for some of statments does not necessarily matter, such as the generations statement which can be defined anywhere within the L-system, however, there are some parts that are required to be in a particular order, for instance, the define and include statements must appear above the axiom and productions 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.

Another design decision that has been made, is that all numbers within the L-system are repersented as floating point numbers. Other data types could be added to the definition of the L-system in the future, however, there is some added complexities in doing so, such as the conversion from one type to another, or having to specify which data type a variable represents. For all intents and purposes, the floating point data type provides all the neccessary functionality needed for the L-system, therefore, it seems unnecessary to add extra data types.

3.4 The L-system Lexical Analyser

In computer science, particularly in the study of computer 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 a parametric 0L-system. For the rewriter to have enough information to rewrite the L-system a string 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, thise 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. We can use the generated Lexer program by providing the L-system input. This takes each word within the L-system and assigns a syntactic category to it. In order 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 there are a number syntactic categories which contain more than one meaning to the grammar, for instance the (and) parenthesis have two meanings, it is either to specify the begining and end of a modules parameters or it specifies presidence within an expression. It is not up to the scanner to determine what the each particular parentheses 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 sytactic category is left up to the parser which is more aware of the context of each syntactic word. Similarly, the symbols [,],+,-,/,\, ∧, &, ! and $ are valid module names; Moreover, it is possible for a T_VAR_NAME to also be a module name, these symbols need to be specifically defined as their own 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 an arithmetic expression. The scanner must separate these symbols and keep them in their own syntactic category in order for the parser to be able to understand the same symbol in multiple contexts.

It is also important to note that there are two special types of tokens, these being the T_FLOAT and T_VAR_NAME which not only are part of a syntactic category but also contain a value, for instance T_FLOAT has a floating point value and T_VAR_NAME has a string value. These values must be kept and provided to the parser.

3.5 The L-system Parser

The parsers job is to find out if the input stream of words from the lexer makes up a valid sentence in the language. The parser fits the syntactical category to the grammatical model of the language . The parser is able to fit the syntactical categories from the lexer to the grammatical model of the language, if the syntactical categories match the grammatical model then the syntax is seen to be correct. If all of the syntax is correct the parser will generate a syntax tree and build the 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, there needs to be a suitable notation to express its syntactic structure or grammatical model. According to Cooper the BNF has traditionally been used by computer scientists to represent context-free grammers such as programming languages, its origins are from the late 1950s and early 1960s. The Backus-Naur Form (BNF) notation represents the context-free grammar by defining a set of non-terminal symbols that derives from a set of terminal or non-terminal symbols. Terminal symbols are elementary symbols of the language definied 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 convension 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]. In order to derive a sentence of text within a language 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 above BNF states that the goal symbol non-terminal <expression> derives from one of four states. Either a terminal number, or an expression contained within two parenthesis, or two expressions either side of an addition terminal symbol, or two expressions either side of a subtraction terminal symbol. This type of notation is recursive in nature and allows the formal language to write expressions which exist within other expressions. For example the expression "5 + 10 - (20 + 2)" can be broken down into the following syntax tree.

have a diagram of the syntax tree for 5+10-(20+2)

Similar to the scanner, the parser program can be quite complex and difficult to write. If there is a change in the grammar or add features at a later date, it is often times difficult to change the parser to account for these changes. To account for this, there is a program called the parser generator, an implementation of which is called Bison. Bison takes in the context-free grammar definition, in a similar format to the BNF, and using the deterministic pushdown automata, generates a parser program.

3.5.1 Backus-Naur Form of the L-system Grammar

```
⟨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⟩;
```

$$
\begin{array}{rcl}
\langle\text{predecessor}\rangle & ::= & \langle\text{predecessorStatementList}\rangle \\
\langle\text{predecessorStatementList}\rangle & ::= & \epsilon \mid \langle\text{predecessorStatement}\rangle\langle\text{predecessorStatementList}\rangle \\
\langle\text{predecessorStatement}\rangle & ::= & \langle\text{module}\rangle \\
\langle\text{condition}\rangle & ::= & \texttt{*} \\
 & & \mid\ \sim\langle\text{float}\rangle \\
 & & \mid\ \langle\text{leftExpression}\rangle\langle\text{operator}\rangle\langle\text{rightExpression}\rangle \\
\langle\text{leftExpression}\rangle & ::= & \langle\text{expression}\rangle \\
\langle\text{rightExpression}\rangle & ::= & \langle\text{expression}\rangle \\
\langle\text{operator}\rangle & ::= & \texttt{==} \mid \texttt{!=} \mid <= \mid >= \mid > \mid < \\
\langle\text{successor}\rangle & ::= & \langle\text{successorStatementList}\rangle \\
\langle\text{successorStatementList}\rangle & ::= & \epsilon \mid \langle\text{successorStatement}\rangle\langle\text{successorStatementList}\rangle \\
\langle\text{successorStatement}\rangle & ::= & \langle\text{module}\rangle
\end{array}
$$

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 definined the syntactic categories, these syntactic categories are in fact 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 relavent data structures and syntax tree.

3.5.2 Dealing with Constant Values and Objects

Defining constants and objects is similar syntactically, the keyword define or include is used, followed by 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. An example of the defining a constant and an object can be seen below:

#define num 10;

#define pi 3.1415;

(3.2)

#include F BRANCH;

#include S SPHERE;

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 will not be used during rewriting but will be necessary to provide information during the interpretation of the resulting string about which objects each module should render.

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 purposes of the rewriter it is important to understand that there are three major parts of a module, there is a module name, which is a string of characters or a symbol, secondly there is a list of parameters signified by open and close parenthesis, there can be zero or more parameters listed. If there are no parameters for a module you can specify it without parenthesis, however, there should then be a space between the module without parenthesis and the next module. Thirdly, each parameter can either be made up of a number, variable, random number range or a mathematical expression containing numbers, variables and parentheses signifying precedence.

There are two types of modules, one being a module definition and the other a module call. The module definition stands as a type of template of a module within a production rule, these templates do not have to hold actual values but the values will be substituted during the rewriting process. The module calls would appear either in the axiom or in the resultant string, the parameters of a module call will hold an actual numerical value. Below is an example outlining the difference between the module definition and module calls.

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

In example 3.3 above, the module A(10) within the axiom, is a module call, as it contains the numerical value of 10 in the first parameter. In the production rule p1 the module A(x) within the predecessor is a module definition, it states that module A's first parmeter has a local variable x, the calling modules value will substitute x and will replace the value of x anywhere within the successor statement, p1's successor has two modules A(x)A(x), also module defintions however the value x will be subtituted with the calling modules value. When a production rules successor rewrites the calling module, the successors modules must have a numerical value, they then become module calls within the resultant string.

A string in the context of a parametric L-system is a vector of modules, the modules are linked one after the other creating a type of string, but instead of characters or symbols we have a string of modules.

3.5.4 Implementing Arithmetic Expressions Trees

As stated within the BNF of the L-system grammar 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 aforementioned expressions between a set of parenthesis such as $(4 + n)$. The result of the expression is calculated from left to right unless the parenthesis are used which prioritises the encapsulated expression to be calculated first. We can represent this expression as an expression tree in the diagram below:

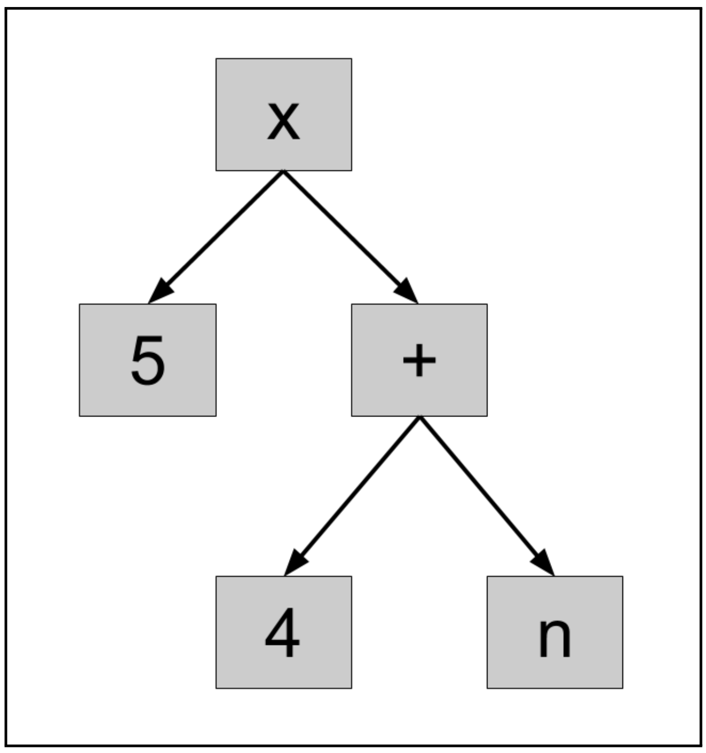


Figure 3.2: Diagram of an expression tree.

The parser provides a syntax tree, which makes it easy to generate the above expression tree, this can have four types of nodes, a variable, number, random range or a operator. The end 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, and for random ranges we can generate the random value and assign it 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 can be quite limited in the amount of variation that can be acheived from rewriting alone. In reality the variation between two plants depends on an enormous number of factors. Regardless of the cause, the factors ultimately change the variation mainly within the branching structure as well as slight variation in the features of the branches themselves. These features include but are not limited to branching angles, branch width, branch length and branch weight. To introduce variation in the braching structure the which rules are chosen needs to be chosen at random every now and then, which is discussed in section 3.5.6. This section introduces a method of providing variation in the features of branch segments, which will known as random ranges.

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

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

There are a number of other types of pseudo-random number generators which can be used to generate numbers according to certain distributions; such as normal, binomial, poisson among others. For the purposes of generating plant-life a uniform distribution should be sufficient.

A random range can be declared in the a define statement, axiom parameter or a production rule successor parameter. If it is declared within a define statement, it will generate a random number when that constant variable is added to the constants table. A random range declared within the axiom will generate random number before the string rewriting process begins, this ensures that the number. And finally, if a random range is defined within the successor of a production rule, the number should be generated during the rewriting process when the current module within the string is successfully matched to the predeccessor at the same time as the expressions within the successors are being evaluated. The values are generated during the rewriting process rather than prior is so that each time a module is matched to the rule, the successor will generate a different value.

3.5.6 Implementing Stochastic Rules

Each rule belonging to a stochastic group of rules provides a probabity value of how likely it is that the particular rule is selected during the rewriting process. For production rules to be part of the same stochastic group they are required to meet 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.

Each time a rule is added to a stochastic group an entry a stochastic rule table is created in order to keep track of which rules are associated with which stochastic group as well as the probability of each rule. Using the stochastic rules below, we can generate a stochastic rule table 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 can be generated by using the module name of the predecessor of 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 serves as a unique identifier for the stochastic group. Once all of the production rules have been processed and added to the stochastic rule table, each groups probabilities should be added together and the total should equal 1.0, certain tolerences should put in place to account for floating point error.

During the rewriting process the module that is to be rewritten will be matched to a particular stochastic group. A uniformly distributed random number is then generated between 0.0 and 1.0. A range for each rule will then be 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 with the range that the random number falls between is then selected and used for rewriting.

3.6 The String Rewriter

Once the L-system has been processed by both the lexical analyser and the parser, the data structures and information, such as the starting string, constant variables and production rules are set up for the string rewriter. The string rewriter, is the final stage which uses this data by starting with a current string of modules which is originally set to the axiom string. The string rewriter will then iterate over each module within the current string carrying matching it to the production rules and rewriting the module with the successor if the production rule matches. Once all of the modules have been rewritten, the current string is replaced by the result string for that iteration. This process is carried out for the number of generations specified within the L-system and will eventually provide the resultant string of modules.

```
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(call, def)
end function
```

3.7 Summary

The L-system rewriter is one of two major systems within the process or procedurally generating plant-life. The rewriter defined in this thesis acts as a type of compiler similar to that of a computer language. In a sense the L-system itself becomes a language, that the L-system rewriter is able to understand and generate meaningful data structures with. The L-system rewriter follows the grammatical structure of the language very closely. Due to the lexer and the paser, it is able to give informative messages if there is a mistake either grammatically or syntactically. If all of these requirements are met, the rewriter is able to use this data to very quickly generate the resultant string of modules. The person writing that L-system, can know indefinitely that the result provided by the rewriting system is correct according the grammar of the L-system.

The L-system languages can be used for many different applications and is not limited to that of procedural plant generation. The interpretation of the resultant string is really what creates the physical result such as the plant model. This means that this L-system rewriter need not change if the L-system is used for a different purpose, only the interpretation will need to change. 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 too much that interpretation becomes reliant on the string rewriter.

Chapter 4

Physics Simulation

The motion of plants is an important 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, layed 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 main technique discussed by Barron et al for simulating the motion of a system like a tree or 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 simaltaniously. 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.

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

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 needs to be calculated. This can be done by multiplying π by the radius r squared and the length l as sees below.

$$v = \pi r^2 l \tag{4.1}$$

The volume can now be used to calculate the mass, this requires knowing the density of the material that the plant is made of. 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 hard wood being about 1000kg/m^3.

$$m = v \times d \tag{4.2}$$

The mass is mainly used to calculate the branch segments moment of inertia, each branch can be simply seen as a long thin cylinder.

$$I = \frac{1}{3}ml^2 \tag{4.3}$$

Where I is the inertia of the branch, m is the Mass of the branch and l is the lenght of the branch. Similarly an inertia tensor can be used for the sake of convenience and to better describe the objects rotational inertia as it is either to use within vector and matrix calculations. The intertia will be used when calculating the velocity of each segment.

$$I = \begin{bmatrix} \frac{1}{12}m(3r^2 + l^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2 + l^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix} \tag{4.4}$$

The forward vector of the branch, this being the vector in the direction that the branch is pointing towards, can be used to calcuate the direction the torque is acting on the branch V, 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.

$$V = v \otimes w \tag{4.5}$$

The dispacement can be calculated by keeping track of the starting local rotation of the branch p as well as the current rotation of the branch q in the form of two quaternions. We can then calculate a quaternion d for the differece of the two quaternions, by taking p and multiplying it by the inverse of q.

$$d = p \times q^{-1} \tag{4.6}$$

Using the displacement vector of the branch we can apply it to Hook's Law, to get the force of the spring.

$$f = -k_s d + k_d v \tag{4.7}$$

Where f is the force exerted by the spring, k_s is the spring constant and x 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. The forces can then be multiplied together to get the net force f_net acting on the spring, this can be used to calculate the momentum and furthermore the velocity of the the branch. T_{delta} is that change in time between physics calculations.

$$M = M_0 + f_{net} * T_{delta} \tag{4.8}$$

The velocity v can be calculated by taking the inverse of the inertia tensor I and multiplying that by the momentum vector M.

$$v = I^{-1} * M Q_v = [0, v] \tag{4.9}$$

The velocity vector can be converted to its quaternion form Q_v in order to make the last step simpler. The scalar part of 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.

$$R = R_0 + (\frac{1}{2} * Q_v * R_0 * T_{delta}) \tag{4.10}$$

Where 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.

Chapter 5

L-system String Interpreter Implementation

The string interpreter is one of the major components of plant generation and it is the final step in the process of procedural generation. The output of this stage of processing is dependant on what the L-system is representing, in this case it is responsible for interpreting the resulting string of modules provided by the L-system rewriter, and uses this to generate the 3D models, structures and data of the resulting plant, which is then rendered and simulated on the screen using the OpenGL framework. The generation of plant-life has three main stages, the first part consists of a turtle graphics interpreter which takes the string of modules as a set of instructions, it starts from the root of the tree and generates a skeleton made up of joints, this is similar to the techniques used in skeletal rigging in animation [Gregory, 2014]. The joints within the tree skeleton each represents a branch segment which has some information about the properties of that segment. These segements can be used to generate the vertex, index and other data that make up the 3D models of the plant. These models can finally be passed to the final part of string interpreter which is the renderer, the renderer is reponsible for taking all the vertices, indices, textures and shaders and organising it in an optimal way that enables rendering the plant on the screen, as well as the physical simulation of the tree skeleton. The stages of string interpretation can be seen in figure 5.1 below.

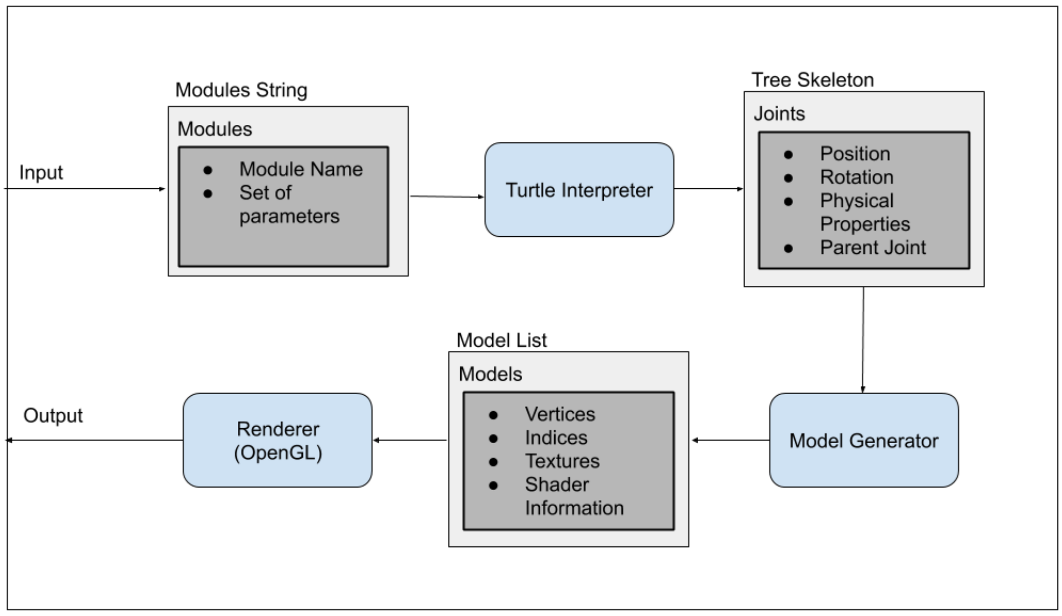


Figure 5.1: Diagram of the stages of L-system interpretation and rendering

This chapter will cover each stage of the string interpreter implementation in detail, as well as well as talk about how the interpreter is able to simulate and animation the plants movements under forces such as gravity and wind in real time.

5.1 Turtle Graphics Interpreter

The main purpose of the turtle graphics interpreter is to take the string of modules from the L-system rewriter, and interpret it as a list of turtle graphics instructions. As briefly covered in chapter 2, 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 and are dependant on what the L-system is trying to represent. The L-system defined for this thesis is a parametric L-system, which allows each module to also provide a number of optional parameters. These may also carry a particular meaning 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, this can also be seen as the length of the branch segment. The second and third parameter is the spring constant of the branch and the mass of the branch repectively. These give some information to the 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.

Instruction Name	Parameter 1	Parameter 2	Parameter 3
F	Distance	Spring Constant	Branch Mass
f	Distance	Spring Constant	Branch Mass
+	Angle of rotation		
-	Angle of rotation		
/	Angle of rotation		
\	Angle of rotation		
∧	Angle of rotation		
&	Angle of rotation		
!	Branch width		

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, which is made up of joints. The joints hold information about the properties of each particular segement 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 scales and rotations must happen before the forward movement. As the rotations change the orientation of the brach and then the movement generates the joint itself. A joint is defined by the figure below:

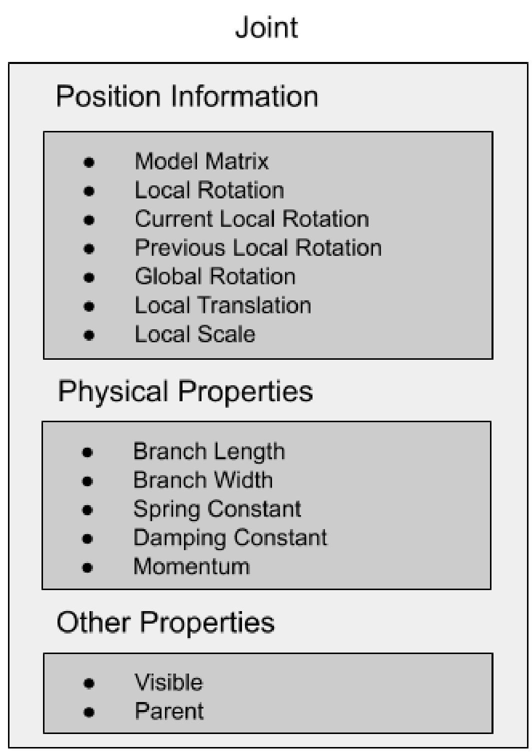


Figure 5.2: Diagram for the properties of a joint

As you can see from figure 5.2, there is 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, whilst 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 for instance calculating the torque or force calculations due to gravity. It is important 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 are what 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 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 -. According to the as discussed in previous chapters the "F" symbol represents a move forward, and "+", "-", "/", "\" symbolize different rotations, and the "[" and "]" represent save and load state respecively. The aforementioned symbols 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 distance to move forward and the change in angle from the previous joint in degrees. This interpretation can be represented with the joint structure shown in figure 5.3 below:

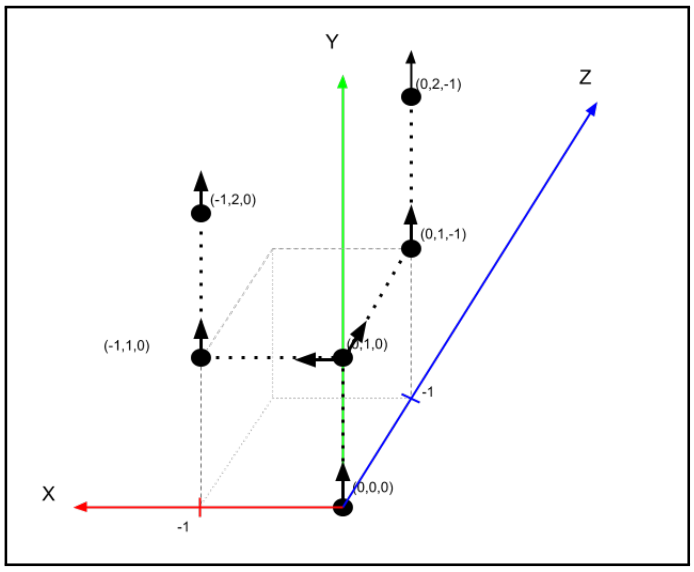


Figure 5.3: Diagram of a simple plant skeleton with joint position and orientation.

5.2 3D Mathematics

In any 3D application, mathematical models are used to represent the positions, rotations and scale of objects within a given scene. All objects within a 3D application are represented by a set of vertices or points, which can be 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 be able to manipulate all vertices of an object in a consistant way, thus rotating, translating or scaling the object within the scene. This section will provide sufficient background on some of most important concepts of 3D Mathematics, such as vectors, matrices and quaternions, which are widely used in the turtle graphics interpreter as well as the model generator.

5.2.1 Vectors

Vectors have many meanings in different contexts, in 3D computer graphics, often vectors are refering to the Euclidean vector. The Euclidean vector is a quantity in n-dimensional space that has both magnitude (the length from A to B) and direction (the direction to get from A to B). 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 simply seen as scaling the magnitude of the vector itself, this can be done uniformly or non-uniformly as seen in the equation below:

$$a \otimes s = (a_x s_x, a_y s_y, a_z s_z) \tag{5.1}$$

Where $\otimes$ is the component-wise product of a vector a and the scaling vector s. Similar to the scalar product of a vector the addition and subtraction of two vectors is the component-wise sum or difference.

$$a \oplus b = [(a_x + b_x), (a_y + b_y), (a_z + b_z)]$$
$$a \ominus b = [(a_x - b_x), (a_y - b_y), (a_z - b_z)] \tag{5.2}$$

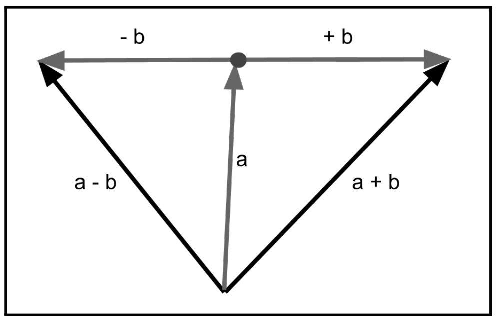


Figure 5.4: Table of common dot product tests between two vectors.

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

$$\alpha = \mid \mathbf{v} \mid = \sqrt{\mathbf{v}_x^2 + \mathbf{v}_y^2 + \mathbf{v}_z^2} \tag{5.3}$$

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

$$\upsilon = \frac{\mathbf{v}}{\alpha} = \frac{1}{\alpha}\mathbf{v} \tag{5.4}$$

There are many different ways to multiply vectors, however, in 3D graphics there two main multiplications. These being the dot and cross product. The dot product yields a scalar 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 which is perpendicular. The dot product can be calculated using the formula below:

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z = d \tag{5.5}$$

Some of the main uses for dot products within 3D graphics is to find whether two vectors are collinear, perpendicular, in the same direction or opposite directions. One possible use for this is to find the dot product of two branches directions in order to find out if they growing in the same direct or in opposite directions. In the table 5.2 below, there are each of the dot product test diagrams as well as the test equation where $ab = \mid a \mid\mid b \mid = a \cdot b$.

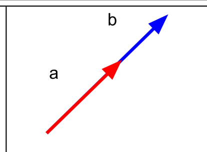
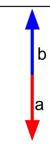
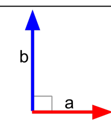
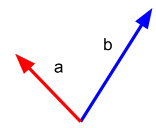
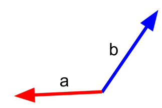
Test	Equation	Example
Collinear	$(a \cdot b) = ab$	
Opposite Collinear	$(a \cdot b) = -ab$	
Perpendicular	$(a \cdot b) = 0$	
Same Direction	$(a \cdot b) > 0$	
Opposite Direction	$(a \cdot b) < 0$	

Table 5.2: Table of turtle instruction symbols and their meaning to the interpreter

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

$$a \times b = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)] \tag{5.6}$$

The result of a cross product can be seen in figure 5.2.1 below. Where vectors a and b give the perpendicular vector $a \times b$. The cross product is very useful within physics calculations when it is necessary to find the rotational motion.
Some of the properties of the cross product are as follows:

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

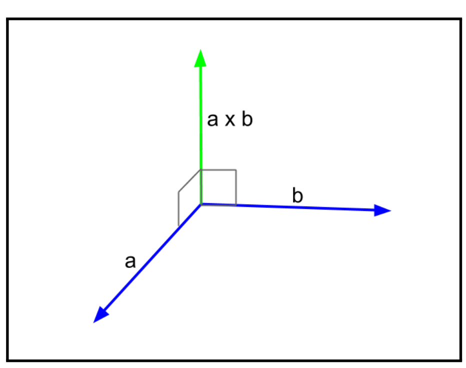


Figure 5.5: Diagram of the cross product of two vectors a and b.

5.2.2 Matrices

A model in 3D space will exist as a set of vertices which each have a position. Moving the model requires moving all of the the vertices of that model without distorting it in any way, this is called a model transform. There are four main types of transforms; translation, rotation, scale and shear. Matrices are a single matematical construct which is capable of carrying out all four of these transformations. This sections will only cover the first three as the shear transformation is likely not going to be useful for this thesis.

A matrix is an 2D array of numbers, arranged into rows and columns, which can come in many different sizes. In 3D graphics, matrices used for transformations are the 3 × 3 and 4 × 4 matrix as seen below. A 3 × 3 matrix can be used for linear transorms such as scaling and rotation, furthermore, a linear transform which contains translation is known as an affine transform and can be represented by a 4 × 4 matrix known as an Atomic Transform Matrix. An atomic Transfom matrix is the concatination of four 4 × 4 matrices, one for translations, rotations, scale and shear transforms. Resulting in a 4 × 4 matrix as shown below.

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \tag{5.7}$$

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \tag{5.8}$$

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.

$$\mathbf{M} = \begin{bmatrix} RS_{11} & RS_{12} & RS_{13} & 0 \\ RS_{21} & RS_{22} & RS_{23} & 0 \\ RS_{31} & RS_{32} & RS_{33} & 0 \\ T_1 & T_2 & T_3 & 1 \end{bmatrix} \tag{5.9}$$

The product of two linear transform matrices will be another linear transform matrix that carries out both of those tranformations. 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. In order to multiply A and B together, the dot product of the row and the column is calculated as seen below. It is also imporant to know that matrix multiplication is non-commutative ($AB \neq BA$).

$$\mathbf{AB} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} (A_{row1} \cdot B_{col1}) & (A_{row1} \cdot B_{col2}) & (A_{row1} \cdot B_{col3}) \\ (A_{row2} \cdot B_{col1}) & (A_{row2} \cdot B_{col2}) & (A_{row2} \cdot B_{col3}) \\ (A_{row3} \cdot B_{col1}) & (A_{row3} \cdot B_{col2}) & (A_{row3} \cdot B_{col3}) \end{bmatrix} \tag{5.10}$$

To translate a vertex in 3D space without causing any distortion. The vertex can be added the the matrix below as follows. These translations can be carried out on all vertices in order to translate a whole object model.

$$V + T = \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} = \begin{bmatrix} (V_x + T_x) \\ (V_y + T_y) \\ (V_z + T_z) \\ 1 \end{bmatrix} \tag{5.11}$$

In order to rotate a vertex in 3D space the vertex position and the rotation angle can be applied to the as a matrix depending on the axis about which it is rotating. These rotation matrices can be applied to the vertex itself in order to gain the new position of the vertex.

$$R_x(\theta) = \begin{bmatrix} (v_x) \\ (v_y) \\ (v_z) \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.12}$$

$$R_y(\theta) = \begin{bmatrix} (v_x) \\ (v_y) \\ (v_z) \\ 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.13}$$

$$R_z(\theta) = \begin{bmatrix} (v_x) \\ (v_y) \\ (v_z) \\ 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.14}$$

$$ST = \begin{bmatrix} S_x \\ S_y \\ S_z \\ 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (S_x R_x) \\ (S_y R_y) \\ (S_z R_z) \\ 1 \end{bmatrix} \tag{5.15}$$

5.2.3 Quaternions

In computer graphics there are a number of ways to represent 3D rotations. As spoken about previously it is possible to use a matrix, however, a matrix has a number of limitations when dealing with rotation. Matrices are represented by nine floating point values and can be computationally expensive to process particularly when doing a vector to matrix multiplication. There are also situations where we would like to smoothly transition from one rotation to another, or find the rotation between two rotations. For example, when calculating the rotation of branch segments between time intervals in physics calculations. It is possible to make these calculations using matrices but it can become very complicated and even more expensive. Quaterions are the miraculous answer to all of these difficulties.

Quaternions look similar to a 4D vector $q = [qx, qy, qz, qw]$, and are represented with a real axis (qw) and three imaginary axis qx, qy, qz.

A quaternion can be represented in complex form as follows: $q = (iq_x + jq_y + kq_z + qw)$.

We will not get into too much detail as to the derivation of quaterions in mathematics however it is important to understand that any unit length quaternion which obays $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$

Unit quaternions are quaternions that are used for rotations, here we can take the angle and the axis of a rotation and convert it to a unit quaternion using the following formula:

$$
\begin{aligned}
& q = [qx, qy, qz] \\
& \text{where} \\
& q_x = a_x sin\frac{\theta}{2} \\
& q_y = a_y sin\frac{\theta}{2} \\
& q_z = a_z sin\frac{\theta}{2} \\
& q_w = cos\frac{\theta}{2}
\end{aligned}
\tag{5.16}
$$

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 that rotation, scaled by sine of the half angle of rotation.

5.3 Model Generator

Modeling the branches of a plant is one of the most important parts for the overall look and feel of that plant that is being generated. The L-system described in the previous sections is able to describe the details about the plants structure, for instance the position, width, length, weight and other important information. The job of the model generator is to take this information and intelligently generate the models vertices, normals, texture coordinates and other information that can then be provided to the OpenGL renderer and finally to the GPU to be rendered on the screen.

The simplest way to generate a model for a branching structure of a plant would be to take a number of cylinders, and to rotate and stack them according to each joints position in 3D space. The up side to this approach is that every branch within the plant shares the same object model, depending on the position, rotation and scale of the branch the relavent matrix transforms can be applied. In this way we are able to represent the overall branching structure of the plant. However, there is a problem which is 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]. This can be shown in the figure below:

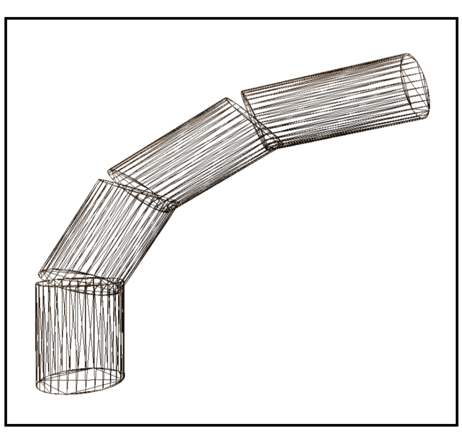


Figure 5.6: Example of the continuity problem faced with stacked branching with a 25° bend per joint.

This simple method of stacking cylinders gives a reasonable looking tree structure and it is usually good enough when the angles of branches are not more than about 25° and the size of the branches do not change. However for a much more convincing tree structure we will want to do better than this. The logical next step would be to actively link the branch segments together.

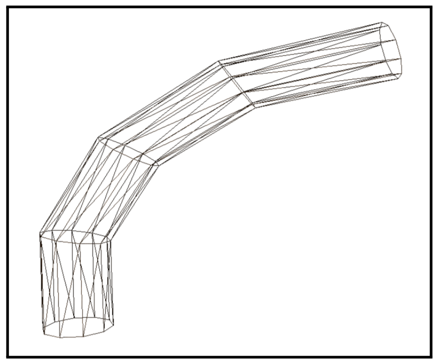


Figure 5.7: Example of linked branching with a 25° bend per joint.

5.4 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 and flowers and renders them on the screen. For this thesis, the Open Graphics Library (OpenGL) application programming interface is use used to efficiently render the models on the screen using the Graphics Processing Unit (GPU).

The GPU is a specially designed piece of harware for processing computer graphics and image processing, it has hundereds of individual compute cores which 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 simpler way. There are a number of 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 be going into great detail about about the specifics of OpenGL but rather the general concepts required for rendering the plant model on the screen.

5.4.1 Buffer Objects

One of the most important parts of the rendering process is buffering the model data onto the GPU. In general the model data will consist of vertex data, texture co-ordinates and vertex normals. The vertex data is simply position of a point within a model, three vertices make up a face and the faces are what are ultimately rendered on the screen. The texture co-ordinates are the locations on a texture image which maps directly to the model vertices. Finally the vertex normals simply known as normals are the average normal vector. A normal vector being the vector that is purpendicular to the surface at a given point, and can be used for Phong shading or other types of lighting techniques.

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 3 values being a vertex position, the second two being a texture co-ordinate and the last three being a vertex normal.

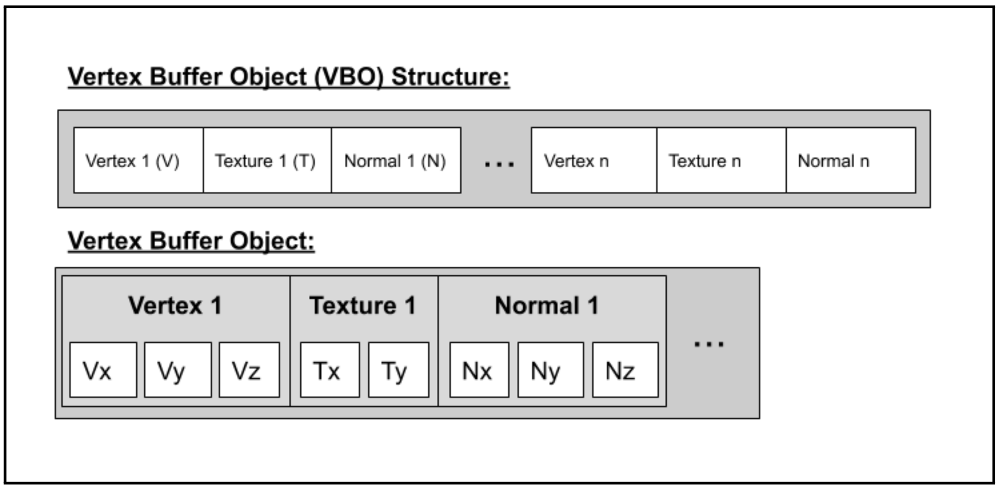


Figure 5.8: Diagram showing the structure of a vertex buffer object.

Chapter 6

Findings and Data Analysis

Chapter 7

Discussion

Chapter 8

Conclusions

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.