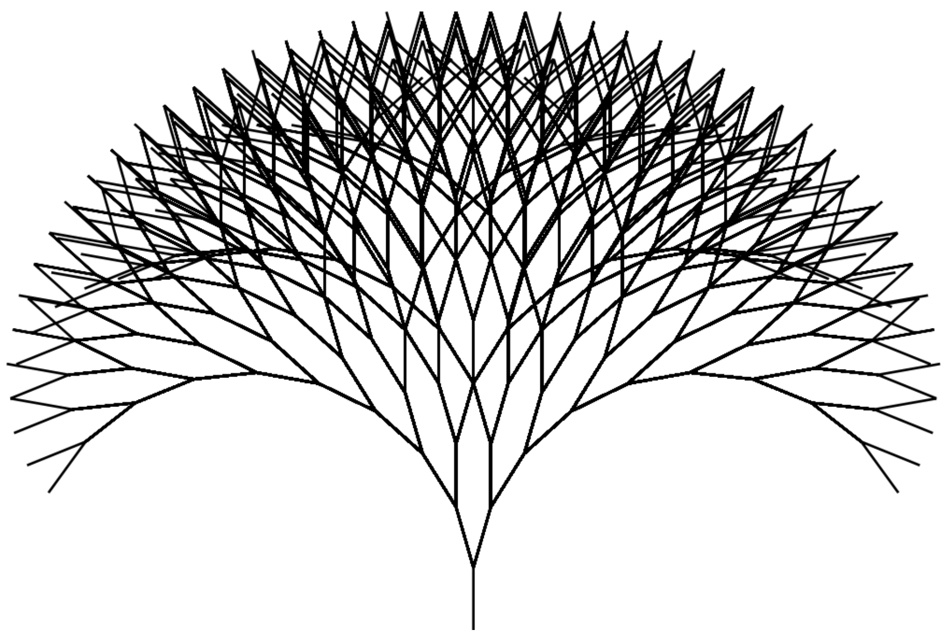
Procedural Plant Generation and Simulated Plant Growth

Massey University



Matthew Crankshaw

25 February 2019

Acknowledgements

Abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To procedurally generate realistic plant like structures in a way that can be used for modern graphics applications, as well as simulate outside forces such as gravity and wind on the generated structure in real time.

1.1 Motivations

One of the most time consuming parts for digital artists and animators is creating differing variations of the same basic 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. Creating a tree asset can take a skilled digital artist more than an hour of work by hand, The artist will then 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 you multiply this by the number of assets that a given artist will have to create and then modify, you are looking at an incredible number of hours that could potentially be put to use creating much more intricate and important assets.

In addition to the huge number of development hours required, 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. We could just store one specification or description of set of similar plants we would like to create, then procedurally generate the geometry during the running of the program.

1.2 Research Aims and Objectives

To develop upon the Lindenmayer System in order to procedurally generating the structure of plant life in real time, in a way that allows us to specify the species, or overall look of the plant as well as introduce variation in order to produce plants that look similar, but can vary in shape, size and branching structure.

I will also investigate using the Lindenmayer System to specify aspects of the plant life that enables the simulation of physical behaviour such as external forces like gravity and wind, and thus having a specification not only for the development of the plants structure but also of its physical behaviour.

Finally, I will be investigating a method of generating a 3D mesh for the given structure of plant, where the branches are seamlessly connected together, and textures are intelligently mapped onto the generated 3D mesh.

1.3 Scope and Limitations

For the purpose of this thesis, we will only be focusing on larger plant life, such as flowers, bushes and trees. We will not be focusing on algae or fungi as these types of plants are usually better represented with specialised texturing in modern 3D applications.

1.4 Timeframe

This research will be carried out over the period of a full year, from the 20$^{\text{th}}$ of February 2019 through to the 20$^{\text{th}}$ of February 2020.

1.5 Structure of Thesis

Chapter 2

Lindenmayer Systems

The L-system at its core is a formal grammar made up of an *alphabet* of characters which are concatonated together into strings. The L-system describes a starting string of one or more characters called an *axiom*. The L-system also describes a set of production rules, which determine whether a symbol in a string should be rewritten with another symbol or string. These rules are applied to each symbol in the *axiom*. Once the production rules have been applied to the all of the symbols in the *axiom* string, we may then use the resulting string as the new starting point, then iteratate over that string once again. This rewriting of strings, via the production rules is the mechanism for generating a structure of state transitions. Essentially the symbols represent a particular state of a system, and the production rules decide whether a state should transition based on a certain criteria, and if so, what the next state should be.

It is important to note that the L-system itself has no concept of what it is trying to represent, therefore, how the L-system is represented is left up to a separate system which is responsible for interpreting the resulting set of symbols in way that makes the most sense for that representation. 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-system itself may be identical. This chapter will discuss a number of different types of L-systems, as well as their features and limitations, whilst focusing on the mechanics behind the rewritting systems. I will then discuss the system which takes the resulting strings generated by the L-system, and interprets them in such a way that we can create 3D models of plant-life.

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 growth in multicelular orgamisms such as algae and bacteria [Lindenmyer, 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 and use an L-system as a method of creating and controlling growth of a connectionist model to represent human perception and cognision [Vaario et al., 1991]. More towards real human biology 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 a simple type of L-systems is known as a Deterministic 0L systems. It is deterministic as each symbol has an associated rule and there is no randomness in determining which rule should be chosen. The term '0L system' abbreviates 'Lindenmayer system with zero-sided interactions'. A zero-sided interactions refers to the multicellular representation of an L-system, where each symbol refers to a type of cell, and a 0L-system is a type of L-system that does not account for the state of its direct neighbouring symbols. There are three major parts to a D0L system. 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, which allows the multicellular organisms to be represented. In this case the each type of cell in represented in the L-system by a character in the alphabet, and the production rules represent the type of cell transitions that take place. In the DOL-system below, there 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. And 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 deinition above, the DOL-system states $w \ : \ a_r$, w signifies that what follows is the starting point (axiom), ergo, 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 value, 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 a rule the symbol is rewritten with the next symbol or string that is specified by the production rule. For G_1 we take the previous generations resultant string which in this case is the resultant string of G_0, being a_r, the first symbol is compared with the production rule until it matches one. 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 $a_l b_r$. The G_0 resultant string only has one symbol it can be concluded that the resultant string of G_1 is $a_l b_r$. The resultant string of generation one is then rewritten to produce generation two and so on, untill we reach the desired number of generations.

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 prodcution rules can express valid state transisions, which thereafter allows us to produce a resulatant string of symbols that obay the L-systems production rules. This functionality is powerful in and of itself, however, the L-system's symbols are only usefull if they are capable of representing something meaningful in the real world, furthermore the L-system does not supply this meaning, each symbols meaning is interpreted after the rewriting process of the L-system. 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 is the system that is reposponsible for using the L-system definition in order 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 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 instructions 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.

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 the table below:

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

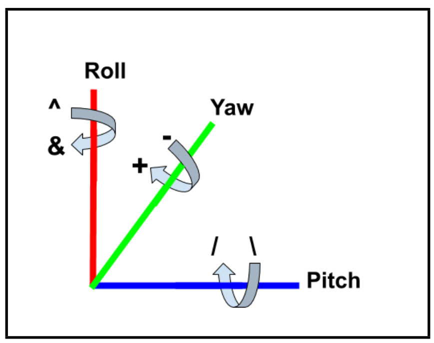


Figure 2.1: Diagram of the 3D rotations of the turtle.

The turtle instructions are presented in such a way that allows movement in three dimensions, the rotations in yaw, pitch and roll are multiplied together and then if the turtle moves forward again it will move in the direction of this new orientation.

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

Instruction 1. Move forward by 1.
Instruction 2. Rotate right by 90 degrees.
Instruction 3. Move forward by 1.
Instruction 4. Rotate left by 90 degrees
Instruction 5. Move forward by 1.
Instruction 6. Rotate left by 90 degrees.
Instruction 7. Move forward by 1.

Instruction 8. Rotate right by 90 degrees.
Instruction 9. Move forward by 1.

talk about the complexities of each system.

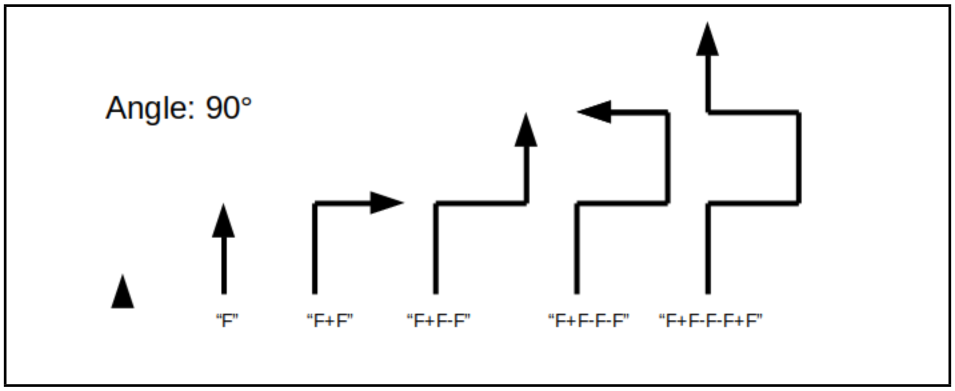


Figure 2.2: Diagram showing a turtle interpreting simple L-system string.

There are a further two commands which I will be covering in detail in section 2.3. We can also have constant numerical values that can be used. For instance, we could pass in a constant value of 1.0 as a parameter to the forward instruction as follows.

F(1.0)+F(1.0)-F(1.0)+F(1.0)

In doing this, we can specify that we would like to move forward by a specified amount. In this case we would like to move forward by 1.0 unit length. We will be covering parametric L-systems in detail in section 2.4.

2.3 Branching

In the previous section there are two turtle commands in particular which were not covered. These are the square bracket commands '[', ']'. The square bracket characters instruct the turtle object to save its position and rotation for the purpose of being able to restore that saved position and rotation later on. This allows the turtle to jump back to a previous position, facing the same direction as it was before. We can then branch off in a different direction.

A way to keep track of these saved locations, is in the form of a stack structure. Each time the '[' is called the current position and orientation of the turtle is saved to the top of the stack. While conversely when the ']' is called we restore the turtles position back to whatever position and orientation is stored on the top of the stack.

An example of this can be shown below in figure 2.2.

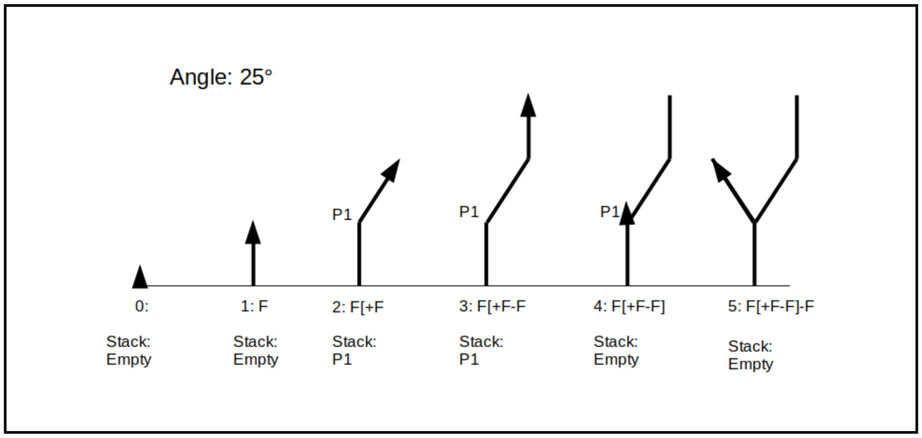


Figure 2.3: Diagram showing a turtle interpreting an L-system incorporating branching.

2.4 Parametric OL-system

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 branchesis 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 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 $+, -, *, \wedge$. Furthermore, we can have the 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. The 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:

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

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

- 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.

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:

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

Here you can see that the #define acts like a declaration, where 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. We have decided modify this functionality, instead of the #include statement, we have provided the #object statement. The #object statement serves a similar purpose however instead of import the symbol 'H' we can specify, #object H HETEROCYST, which specifies that we are associating the symbol or module 'H' with the object HETEROCYST. The HETEROCYST object will be stored in a predefined library. This way we can associate the same object to multiple symbols. It also does not limit us to a predefined name for an object. Below is an example using the #object statement:

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

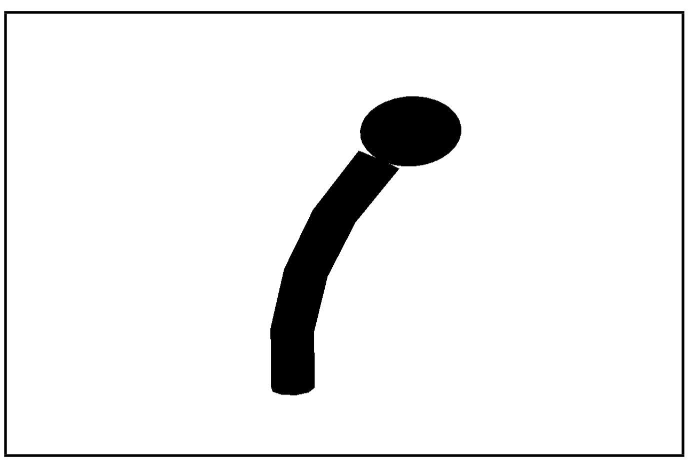


Figure 2.4: Diagram of an L-system Using Multiple Objects.

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 module the S module renders a sphere of diameter 1.0. It is possible

I will be going into more detail about this and other features in section 5.2.

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.

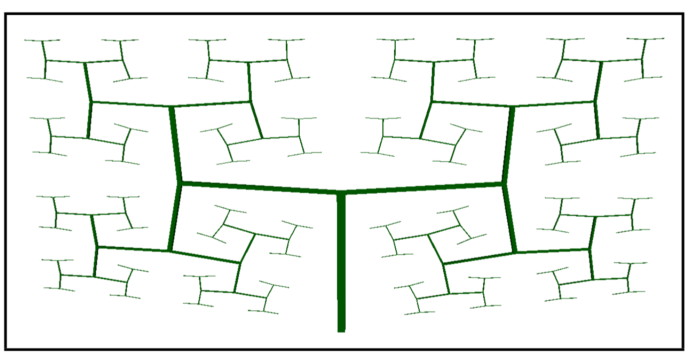


Figure 2.5: 3D Parametric L-system.

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 particalar 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:

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

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:

$$
\begin{aligned}
&n = 2,\ 4,\ 6\\
&\#\text{object F BRANCH}\\
&\#\text{object L LEAF}\\
&\#\text{object S SPHERE}\\
&\#\text{define r 45}\\
&\#\text{define len 0.5}\\
&\#\text{define lean 5.0}\\
&\#\text{define flowerW 1.0}\\
&\omega \ \ :\ !(0.1)I(5)\\
&p_1\ :I(x)\ :\ x\ >\ 0\ \ \rightarrow\ F(len)-(lean)[R(0,100)]F(len)[R(0,100)]I(x-1)\\
&p_2\ :R(x)\ :\ x\ >\ 50\ \rightarrow\ -(r)/(20)!(2.0)L(2)!(0.1)\\
&p_3\ :R(x)\ :\ x\ <\ 50\ \rightarrow\ -(r)\backslash(170)!(2.0)L(2)!(0.1)\\
&p_4\ :I(x)\ :\ x\ <=\ 0\ \rightarrow\ F(len)!(flowerW)S(0.3)
\end{aligned}
\tag{2.11}
$$

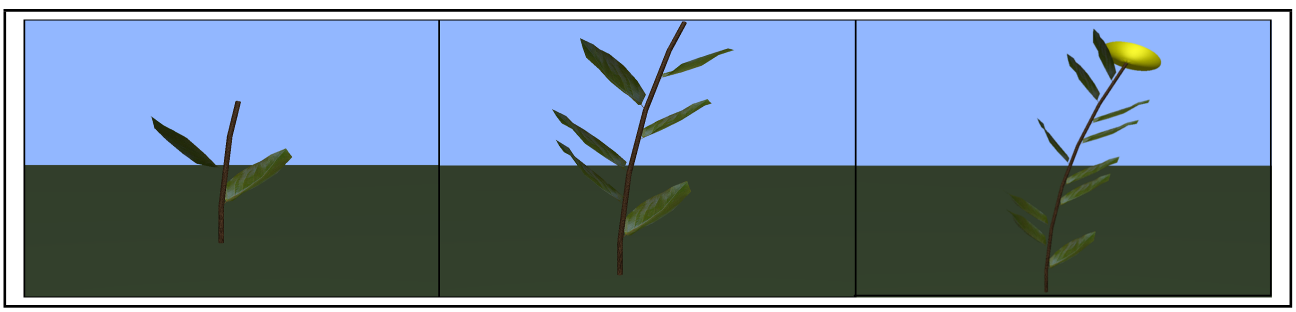


Figure 2.6: Condition Statements Used to Simulate the Growth of a Flower. 2nd Generation on the Left, 4th Generation in the Center and 6th Generation on the Right

2.4.5 Representing Randomness

Randomness is an essential part of nature. If there was no randomness in plant life, we would end up with very symetric and unrealistic plants. 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.

Random Fractal:
#n = 2;
#w : !(0.2)F(1.0);
#p1 : F(x) : * : F(x)[+(25)F(x)][-(25)F(x)]+({-20.0, 20.0})F(x)-({-20.0, 20.0})F(x);

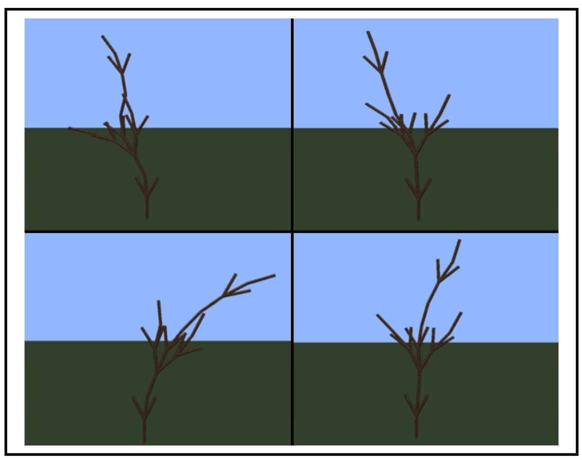


Figure 2.7: Different Variations of the Same L-system with Randomness Introduced in The Angles.

In figure 2.7 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).

2.4.6 Stochastic Rules in the L-system

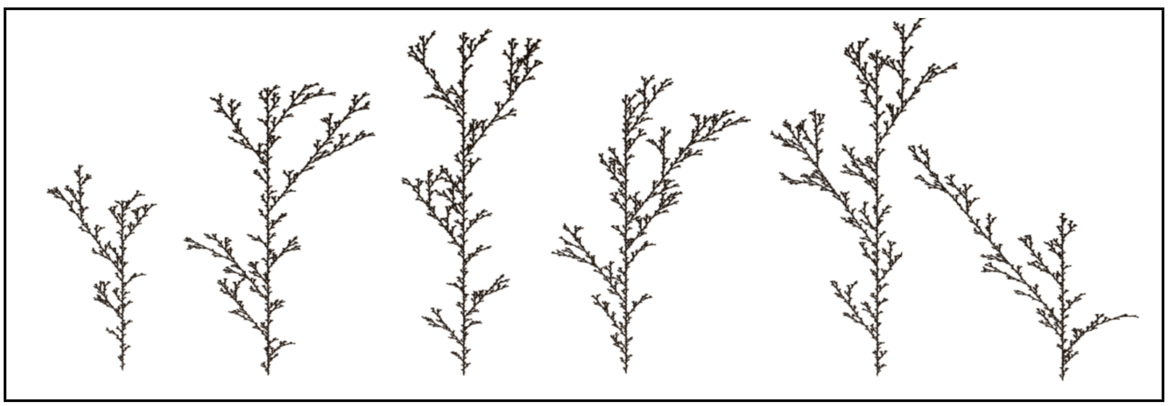


Figure 2.8: Representation of an L-system with a probability stochastic with a 33.33% chance for each rule.

Chapter 3

3D Mathematics

In any 3D application we are creating a mathematical model to represent the objects within a given scene. Therefore mathematics plays a very large part in many areas of 3D graphics. Three dimensional mathematics makes use of trigonometry, algebra and even statistics, however in the interest of time, I will be mainly focusing on the specific concepts used in 3D vector, quaternion and matrix mathematics. These concepts contribute to the representation of 3D models as well as model transformations such as: translation, rotation and scaling.

When representing objects we need to keep track of where they are in the virtual world. In a 2 dimensional world this may be represented by two numbers, an X and a Y position. In 3 dimesions we can represent this with X, Y and Z positions. 3D objects are will usually have a point of origin or global position but in most 3D applications, points or vectors make up triangles. One triangle can be said to be a face and multiple faces will make up a 3D object. In this section we will be looking at the use of points and vectors in 3D graphics.

3.1 Points

A point is a position in space of n-dimensions. In computer graphics applications we usually deal with 2D or 3D coorinate systems. The most common coorinate system used in computer graphics is cartesian coorindates. Cartesian coordinates of a 2D system can be represented by an ordered pair of perpendicular axes, which can be represented as (p_x, p_y). Similarly a point in 3D space can be represented by an ordered triple of perpendicular axes, represended in the form (p_x, p_y, p_z). This can be represented in figure 3.3 below.

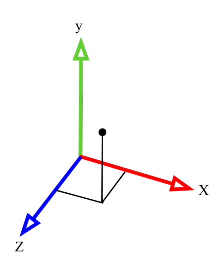


Figure 3.1: Point in 3D space shown using cartesian coordinates.

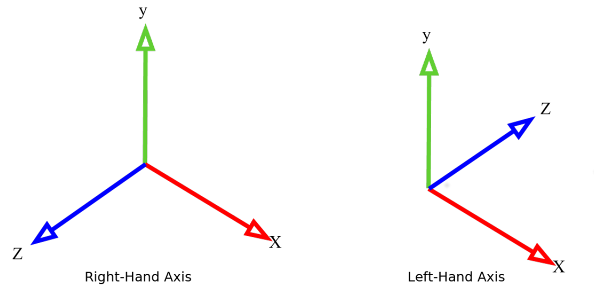


Figure 3.2: Right Hand and Left Hand Coordinate Systems.

3.2 Vector

Vectors have many meanings in different contexts, In 3D computer graphics, when we talk about vectors we are talking about 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 direction with a certain length. A 3D vector can be written as a triple of scalar values eg: (x, y, z)

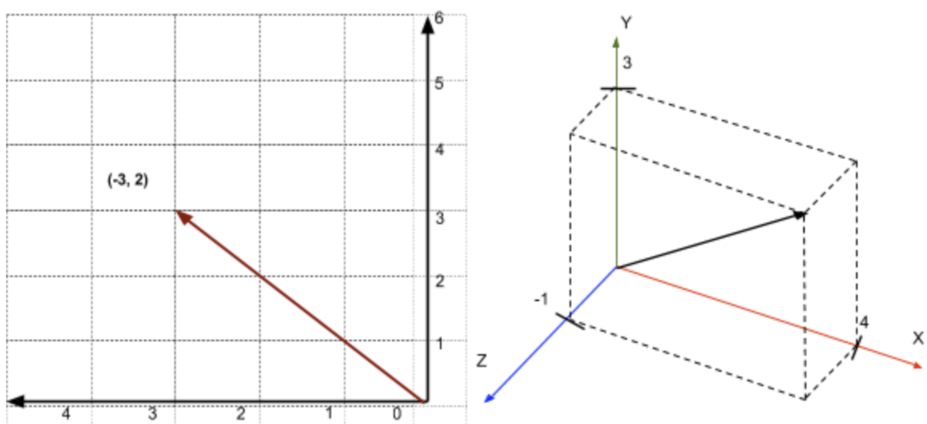


Figure 3.3: 2D Vector representing the vector at (-3, 2) And 3D Vector (4, 3, -1).

3.2.1 Vector Multiplication

3.2.2 Vector Addition and Subtraction

3.2.3 Dot and Cross Product

3.3 Matrices

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \tag{3.1}$$

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \tag{3.2}$$

When it comes to matrices in 3D graphics, instead of using a 3 $\times$ 3 matrix we tend use a 4 $\times$ 4 matrix known as an Atomic Transform Matrix. An atomic Transfom matrix is a concatination of four 4 $\times$ 4 matrices: translations, rotations, scale and shear transforms. Resulting in a 4 $\times$ 4 matrix in the following representation:

$$\mathbf{M} = \begin{bmatrix} RS_{11} & RS_{12} & RS_{13} & 0 \\ RS_{21} & RS_{22} & RS_{23} & 0 \\ RS_{31} & RS_{32} & RS_{33} & 0 \\ T_1 & T_2 & T_3 & 1 \end{bmatrix} \tag{3.3}$$

Where RS is a 3 $\times$ 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.

3.3.1 Matrix Multiplication

$$\mathbf{AB} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} \tag{3.4}$$

$$= \begin{bmatrix} A_{row1} \cdot B_{col1} & A_{row1} \cdot B_{col2} & A_{row1} \cdot B_{col3} \\ A_{row2} \cdot B_{col1} & A_{row2} \cdot B_{col2} & A_{row2} \cdot B_{col3} \\ A_{row3} \cdot B_{col1} & A_{row3} \cdot B_{col2} & A_{row3} \cdot B_{col3} \end{bmatrix} \tag{3.5}$$

Matrix multiplication is non-commutative, Meaning order matters.

$$AB \neq BA \tag{3.6}$$

3.3.2 Translation

3.3.3 Rotation

3.3.4 Scale

3.4 Quaternions

We are able to express 3D rotations in the form of a matrix, however in many ways a matrix is not the optimal way of representing a rotation. Matrices are represented by nine floating point values and can be quite 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, this is possible using matrices but ican be very complicated. 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 $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$

3.4.1 Unit Quaternion

Unit quaternions are what 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:
$q = [qx, qy, qz]$
where
$q_x = a_x sin\frac{\theta}{2}$
$q_y = a_y sin\frac{\theta}{2}$
$q_z = a_z sin\frac{\theta}{2}$
$q_w = cos\frac{\theta}{2}$
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.

3.4.2 Quaternion Multiplication

3.4.3 Conjugate and Inverse

Chapter 4

Physics Simulation

In order to generate realistic models of trees, there are a number of characteristics which are needed in order to provide a realistic looking tree. The motion of trees is not something really think too much about on an everyday basis. A plants movement is very subtle, however, when completely devoid of movement, the plant starts looking very unnatural. We will now be looking into simulating the animation of plants, and building this functionality into the parametric L-system which we covered in previous sections. Not all 3D applications may need to make use of simulating the plants, so the solution will allow us to declare the parameters of a 3D animation, or if not neccessary, leave them out entirely.

[Barron et al., 2001]

4.1 Motion Equations

Torque
$\tau = I\alpha$
Where τ is the torque, I is the moment of inertia and α is the angular acceleration.

$\tau = f \otimes R$
Where τ is the torque, f is the force acting on the end of the branch and R is the vector representing the length and orientation of the branch.

Mass
$M = \Pi\ r^2h$
Where M is the mass represented in kg, r is the radius of the branch in meters and h is the height of the branch in meters.

Inertia
$I = \frac{1}{3}ML^2$
Where I is the inertia of the branch, M is the Mass of the branch and h is the height of the branch.

Angular Acceleration
$\omega = \omega_0\alpha t$
Where ω is the angular velocity, ω_0 is the previous angular velocity, α is the angular acceleration and t is the change in time.

Next angle equation
$\theta = \theta_0 + \omega_0 t + \frac{1}{2}\alpha t^2$
Where θ is the angle, θ_0 is the previous angle, w_0 is the previous angular velocity, t is the change in time and α is the angular acceleration.

4.2 Hook's Law

$f = -k_s x + k_d v$
Where f is the force exerted by the spring, k_s is the spring constant and x is the total displacement of the spring. We then would also like to add a dampening force which is the $k_d v$ part where k_d is the dampening constant and v is the velocity at the end of the spring.

Chapter 5

L-system Parser Implementation

5.1 Environment and Tools

The implementation of this thesis will be in the C and C++ programming languages [Stroustrup, 2000], and I will be using the modern Open Graphics Library or 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 low level nature of OpenGL is important as some of the structures and models we are going to be displaying and simulating can be 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.

There are also a number of libraries which will provide some extra functionality. The standard library for C and C++ provides many usefull structures and functions which will be incredibly usefull during the development process. For more specialised mathematics capabilities the OpenGL Mathematics Library (GLM) library holds many mathematics classes and functions for conveniently dealing with some 3D mathematics such as vectors, matrices and quaternions. Another important library is the Graphics Library Framework (GLFW) which is a multiplatform 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, that is able to keep track of changes that have been made to the files within a project folder. It will be used to keep track of 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,].

5.2 L-system Generator

The purpose of the L-system generator is to read a file, called the L-system discriptor which contains any information that might be necessary for the string rewriting process. This file must contain the number of times the string will be rewritten (number of generations), a starting point (axiom) and at least one production rule, it may also contain some constant variables and other information.

For simple L-systems, the generator need not be too complicated. The Koch Curve L-system stated below is a good example of this.

Angle: 90
Axiom: F
Rules:
F $\rightarrow$ F+F-F+F

Here we have a constant value of 90 degrees, the starting point of 'F' and one rule F $\rightarrow$ F+F-F+F. This type of system is very simple to rewrite computationally.

Here we describe some pseudocode

When we move onto some more complicated L-systems, such as those that use parameters which have expressions with both variables and numbers. We end up with an L-system file that is quite difficult to process and rewrite. In order to compute these complex L-systems we need to first develop a formal grammar that describes how L-system files are defined. Once we have a formalization of how to define an parametric l-system we can create a system to carry out the rewriting.

5.2.1 Building a Generalised L-system Grammar

We are now able to represent complex three dimensional tree structures in the form of a L-system rule set. In a computing sense this rule set can be seen as a type of program. In the program we define the number of generations we would like to generate, the starting point (Axiom) some constant varables (#define) and the set of production rules.
Using this information, we iterate through the from generation to generation rewritting the strings and then at the end provide a resulting string which will then be interpreted and displayed on the screen.
We can represent the languages grammar in the form of a Backus-Naur Form, BNF is a notation for Context-Free Grammars used to describe the syntax of different languages. In this case the BNF is used below to describe the syntax of the parametric L-system grammar.

For example:

<expression> $\rightarrow$ number
<expression> $\rightarrow$ (<expression>)
<expression> $\rightarrow$ <expression> + <expression>
<expression> $\rightarrow$ <expression> - <expression>

The first line states that an <expression> can be any number, line two states that an <expression> can also be an expression that is inside parenthesis. Line three states that an <expression> can be an <expression> added to another <expression> , furthermore line four states that an <expression> can also be an <expression> subtracted by another <expression> .

The above grammar can be expressed as follows:

<expression> $\rightarrow$ number

| (<expression>)
| <expression> + <expression>
| <expression> - <expression>

Here the | symbol can be articulated as an OR, therefore it can be said that an <expression> can be a number OR an <expression> surrounded by parenthesis, OR an <expression> added to another <expression> OR an <expression> subtracted by another <expression> .
In addition to this, any statement that is not surrounded by <>, states it must match that particular string. The $\in$ followed by an | states that it can either be nothing or another statement.

5.2.2 Backus-Naur Form of the L-system Grammar

<prog> → ∈
| <stmts> EOF

<stmts> → ∈
| <stmt> <stmts>

<stmt> → EOL
| <generations>
| <definition>
| <axiom>
| <production>

<generations> → #n = <float> ;

<definition> → #define <variable> <float> ;
| #define <variable> + <float> ;
| #define <variable> -<float> ;

<axiom> → #w : <axiom statement list> ;

<axiom statement list> → ∈
| <axiom statement> <axiom statement list> ;

<axiom statement> → <moduleAx>

<moduleAx> → <variable> | ”+” | ”−” | ”/” | ”\” | ”ˆ” | ”&” | ”!”
| <variable> (<paramAx> <paramListAx>)
| +(<paramAx> <paramListAx>)
| -(<paramAx> <paramListAx>)
| /(<paramAx> <paramListAx>)
| \(<paramAx> <paramListAx>)
| ˆ(<paramAx> <paramListAx>)
| &(<paramAx> <paramListAx>)

<paramAxList> → ∈
| , <paramAx> <paramAxList>

<paramAx> → <expression>

<production> → # <variable> : <predecessor> : <condition> : <successor> ;

<predecessor> → <pred statement list>

<pred statement list> → ∈
| <pred statement> <pred statement list>

<pred statement> → <module>

<condition> → *
| <left expression> <conditions statement> <right expression>

<left expression> → <expression>

<right expression> → <expression>

<condition statement> → == | != | <| >| <= | >=

<successor > → <successor statement list >

<successor statement list> → ∈
| <successor statement> <successor statement list >

<successor statement> → <module>

<module> → <variable> | + | − | / | \ | ˆ| & | !
| <variable> (<param> <paramList>)
| +(<param> <paramList>)
| -(<param> <paramList>)
| /(<param> <paramList>)
| \(<param> <paramList>)
| ˆ(<param> <paramList>)
| &(<param> <paramList>)

<paramList> → ∈ | : <param> <paramList>

<param> → <expression>

<expression> → <variable>
| <float>
| <expression> + <expression>
| <expression> - <expression>
| <expression> * <expression>
| <expression> / <expression>
| <expression> ˆ<expression>
| ”(” <expression>)

<float> → [0-9]+.[0-9]+

<variable> → [a-zA-Z_][a-zA-Z0-9_]*

5.3 The L-system Interpreter

Traditionally an interpreter 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].

A similarity can be drawn between traditional interpreted languages and the L-system descriptors. With the L-system descriptors we are defining a set of constant variables, a starting point and then some production rules. Once we have all of this information, we would like to interpret that information a number of times.
For instance, we may want to rewrite five generations of the L-system, but later on we may want to instead generate up to the tenth generation. So we don't want to have to throw all the previous information away and start from scratch, instead, we can go from the current state of the interpreter and just rewrite another five times. If we would then like to get the resulting string we can just ask for it from the interpreter.

To make the most of a CFG like the L-system grammar, creating an interpreter specifically designed to interpret the L-system descriptors can not only make it simpler to debug any syntactic errors, but also make the string rewriting much faster.

In compilers and interpreteres there is usually a three step process in order to understand the input program. The first is the scanner or lexical analyser, the output of the scanner is then processed using the parser, this generates a syntax tree which is then further processed by a context-sensitive analyser. I will be elaborating on each of these processes in sections 5.3.1 and 5.3.2.

5.3.1 Scanner - Flex

D. Cooper and L. Torczon write that "The scanner, or lexical analyser, reads a stream of characters and produces a streeam of words. It aggregates characters to form words and applies a set of rules to determin wheter or not 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].

Writing a custom Lexer can be quite complicated and time consuming to design and implement, and once a custom Lexer has been created it can be difficult to change some functionality at a later stage. Luckely there is a well known program known as the Fast Lexical Analyzer Generator (Flex), Flex takes a .lex file which contains the lexical rules of the language, it uses these rules to create a Lexer program. When Flex is run it will create a Lexer in the form of a C program.

5.3.2 Parser - Bison

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. If the Parser is able to fit the syntactical category of the word to the grammatical model of the language then the syntax is seen to be correct. If all of the syntax is correct the Parser will output a syntax tree and build the structures for use later on during the compilation process [Cooper and Torczon, 2011].

5.4 Displaying the L-system Instructions

5.4.1 Basic 2D L-systems

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.

Koch Curve:
#n = 4;
#define r 90;
#w : F(1);
#p1 : F(x) : * : F(x)+(r)F(x)-(r)F(x)-(r)F(x)+(r)F(x);

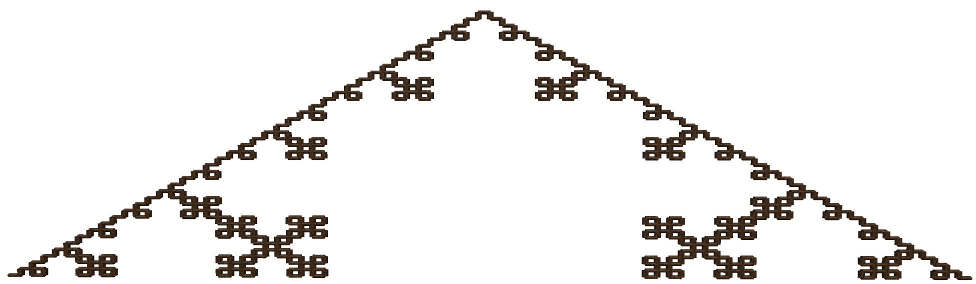


Figure 5.1: Koch Curve.

Sierpinski Triangle:
#n = 4;
#define r 60;
#w : F(1);
#p1 : F(x) : * : X(x)-(r)F(x)-(r)X(x);
#p2 : X(x) : * : F(x)+(r)X(x)+(r)F(x);

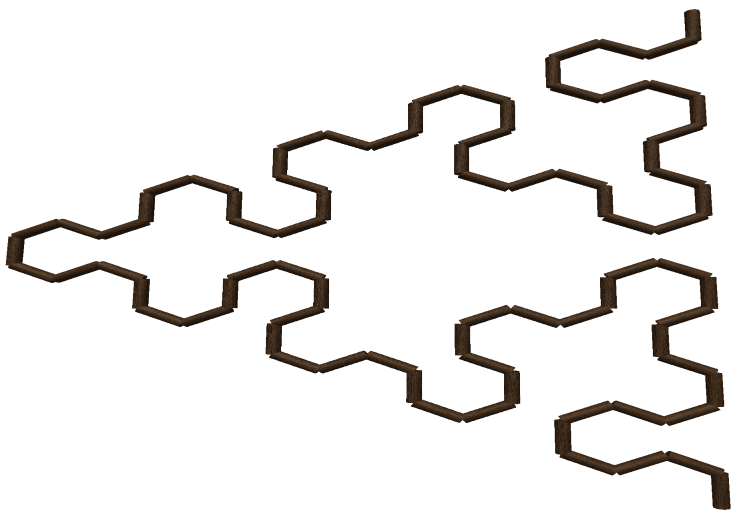


Figure 5.2: Sierpinski Triangle.

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

Figure 5.3: Fractal Plant.

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

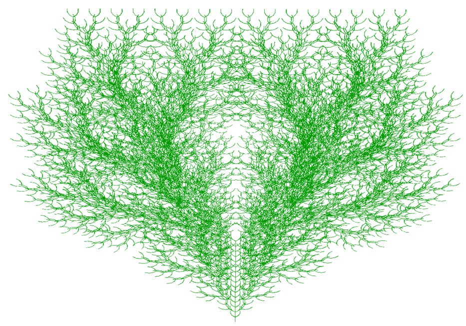


Figure 5.4: Fractal Bush.

5.4.2 The Use of L-systems in 3D applications

L-systems have been talked about and researched since its inception in 1968 by Aristid Lindenmayer. Over the years it's usefulness in modelling different types of plant life has been very clear, however its presence has been quite absent from any mainstream game engines 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.

Chapter 6

Turtle Interpreter Implementation

6.1 Modeling Seamless Branches

Modeling the branches of a plant is the most important part behind the overall look and feel of that plant. The L-system described in the previous sections is able to describe the most important details about the plants structure. For instance the width, length, weight and other important information. Our job now is to take this information and intelligently generate a model consisting of vertices, normals, texture coordinates and other information that can then be provided to the GPU and then rendered on the screen.

The most obvious way to generate a model for a branching structure would be to take a number of cylinders and to rotate and stack them according to the branching structure. In this way we are able to represent the overall branching structure of the tree. However there is a problem 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:

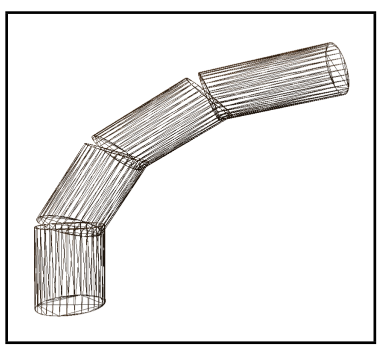


Figure 6.1: Example of the continuity problem faced with stacked branching with a 25° bend per joint.

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.

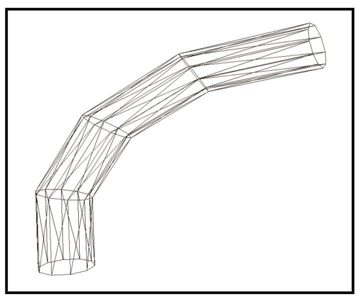


Figure 6.2: Example of linked branching with a 25° bend per joint.

Chapter 7

Findings and Data Analysis

Chapter 8

Discussion

Chapter 9

Conclusions

Acronyms

2D Two Dimensional. 22

3D Three Dimensional. 22, 23

API Application Programming Interface. 28

BNF Backus-Naur Form. 4, 29, 31

CFG Context-Free Grammar. 29, 33

Flex Fast Lexical Analyzer Generator. 33

GLFW Graphics Library Framework. 28

GLM OpenGL Mathematics Library. 28

GPU Graphics Processing Unit. 28

Glossary

Lexer A computer program that performs lexical analysis. 33

OpenGL The Open Graphics Library is a cross-platform, cross-language application programming interface used in creating graphics applications. 28

Parser A computer program that performs parsing. 33, 34

Appendix A

Appendix

A.1 Appendix 1

A.2 Bibliography

Bibliography

[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.

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

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

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

[Lindenmyer, 1968] Lindenmyer, 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.

[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, 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.

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

[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.

[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.