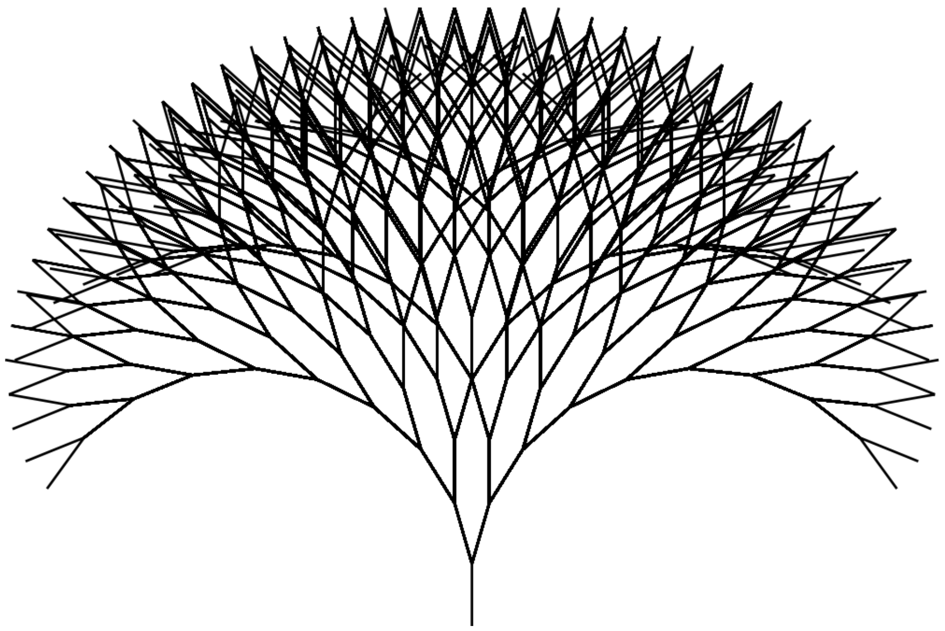
Procedural Plant Generation and Simulated Plant Growth

Massey University



Matthew Crankshaw

25 February 2019

Acknowledgements

Abstract

Contents

List of Figures

Chapter 1

Introduction

Here I will introduce the project and the thesis in general.

1.1 Context and Background

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.

The unique thing about plant life when compared with other types of graphics assets is that plant life is very random in the way it grows and it does not take an incredibly realistic model of a plant life to trick the human mind into believing that what it is seeing is some kind of plant. However what stands out like a sore thumb is when plants that are growing next to each other seemingly have no influence on the plant life around them.

1.2 Methodology

1.3 Timeframe and Scale

Chapter 2

Introducing Lindenmayer Systems

Aristid Lindenmayer is a well known biologist who started work on what would become known as the Lindenmayer System or L-system for short. Lindenmayer initially intended the L-systems to be to be used as a way to describe the development of simple organisms such as algae and bactaria. More recently the concept has been adapted to be used to describe larger organisms such as plants and trees. L-systems have also been used to describe non organic structures like music. [Worth and Stepney, 2005]

An L-system at its core is a definition of a starting point called the *Axiom*, the Axiom is a state or number of states which bear some kind or real world meaning. A single state can then have a number of rules describing what the next state will be. In section 2.1 below, I will be speaking about the most simple type of L-system called a Deterministic 0L-system. These types of simple D0L-systems serve as a good way to introduce the concept of an L-system.

2.1 Simple DOL-system

According to Prusinkiewicz and Hanan a simple type of L-systems are those known as deterministic 0L systems, where the string refers to the sequence of cellular states and '0L system' abbreviating 'Lindenmayer system with zero-sided interactions'. With 0L systems there are only three major parts. There is a set of symbols known as the (*alphabet*), the starting string (*Axiom*) and state transition rules (*rules*). The alphabet is a set of states. The starting string is a starting point containing one or more states. The transition rules are rules that dictate whether a state should remain the same or transition into a different state or even disappear. [Prusinkiewicz and Hanan, 2013].

An example of a deterministic 0L system:

We are given the *alphabet*: A, B
and the *axiom*: A
and the *rule* set:
A $\rightarrow$ AB
B $\rightarrow$ A

The symbol $\rightarrow$ can be verbalised as "replaced by". Therefor it can be said that the string 'A' is replaced by string 'AB' and string 'B' is replaced by the string 'A'.
To start we take the *axiom* which is this case is 'A' and we run through all of the states in this

start string 'A' matches the rule: A $\rightarrow$ AB and is therefor replaced by 'AB'. 'AB' then becomes the new start string and we then match the rules once again. Below I have shown the resulting string up to six generations.

If we then apply the rules to the L-system we find it creates the following generation structure.

1.) A
2.) AB
3.) ABA
4.) ABAAB
5.) ABAABABA
6.) ABAABABAABAAB

This rewriting of initial string using a set of rules is ultimately the underlying concept behind L-systems. There are a number of improvements that can be made to this type of L-system in order to accommodate for more complex and intricate structure. One of which is the inclusion of *constants*. Constants can be considered any state that does not have a rule associated with it or remains the same from generation to generation and therefore holds a consistent value or meaning. These constants are used when the L-system is interpreted and thus holds a constant value during string rewriting, I will be covering this in section 2.2.

Prusinkiewicz and Lindenmayer simulated a blue-green bacteria known as *Anabaena catenula* [Prusinkiewicz and Lindenmayer, 2012]

The DOL-system representation is shown below in the grammar:

w : a_{r}
p1 : $a_{\mathrm{r}} \rightarrow a_{\mathrm{l}}b_{\mathrm{r}}$
p2 : $a_{\mathrm{l}} \rightarrow b_{\mathrm{l}}a_{\mathrm{r}}$
p3 : $b_{\mathrm{r}} \rightarrow a_{\mathrm{r}}$
p4 : $b_{\mathrm{l}} \rightarrow a_{\mathrm{l}}$

The value w is there to specify the axiom which is this case has the value of a_{r}. *p1*, *p2*, *p3*, *p4* are the names of the rules that follow after the semi-colon. In order to simulate Anabaena catenula we need four rules.
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]

This gives us a good real world demonstration of how even a simple DOL-system can represent something in the real world.

2.2 Interpreting L-systems

After we have iterated through each generation, using string rewriting we are left with a string of characters or states which represent a number of different instructions that need to be interpreted.

As with any grammar, there is a number of ways of interpreting the string that is generated by the L-systems rules. One method proposed by Przemyslaw Prusinkiewics is ”to generate a string of symbols using an L-system, and to interpret this string as a sequence of commands which control a ’turtle’” [Prusinkiewicz, 1986].

When talking about a turtle, prusinkiewicz is refering 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. A simple set of turtle instructions can be expressed in the form below and displayed as Figure 2.1.

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

In order to quickly and effectively interpret the L-systems resulting string we can define the turtle instructions in the form of the following characters below:

• F:	Move forward by a specified distance whilst drawing a line
• f:	Move forward by a specified distance without drawing a line
• +:	Rotate to the right specified angle.
• -:	Rotate to the left by a specified angle.
• [:	Save the current position and angle.
•]:	Load a saved position and angle.

Similarly a three dimensional L-system string may hold the following commands in the form of symbols.

• 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.
• [:	Save the current position and angle.
•]:	Load a saved position and angle.

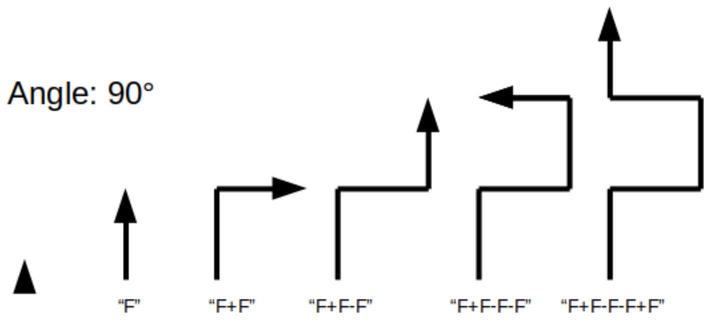


Figure 2.1: Diagram showing a turtle interpreting simple L-system string.

2.3 Branching

In the previous section there are two turtle commands in particular which were not covered. These are the square brackets "[", "]". The square bracket characters instruct the turtle object to save its position and current angle for use later on. This allows the turtle to jump back to a previous point facing the same direction as it was before, It could then branch 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 direction of the turtle is saved to the top of the stack. Conversely when the "]" is called we restore the turtles position back to whatever position and direction is stored on the top of the stack.

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

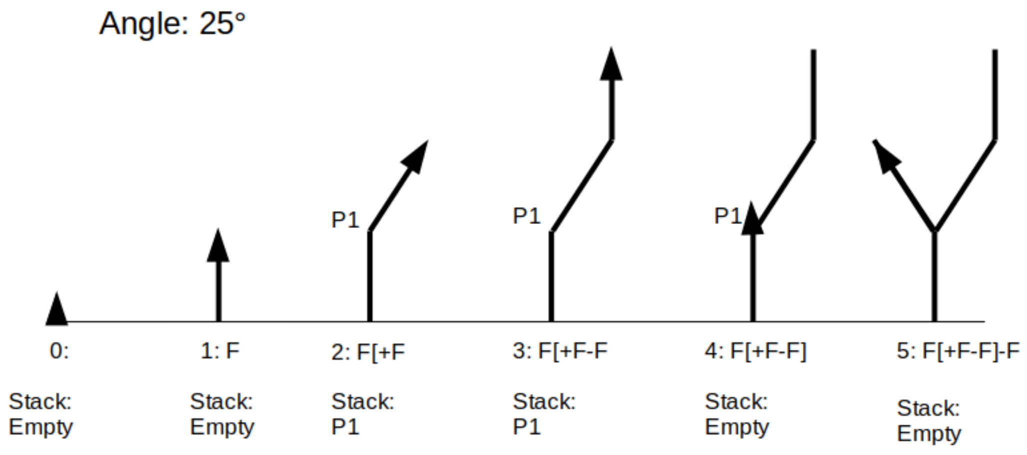


Figure 2.2: Diagram showing a turtle interpreting an L-system incorporating branching.

2.4 Parametric L-system

With simplistic L-systems like the algae representation above, there are a number of details that are skipped over when making this simplistic representation. (talk about the representation for both parameterized and non parameterised Algae systems). When it comes to representing trees as L-systems a simplistic approach would be to just assume that the width and length of each branch section is constant and will not vary depending on where in the tree it is. We can also assume that the angles at which a branch may split is also constant, say 25 degrees. The resulting representation of this L-system is a tree like structure, however it is not a very accurate representation of a real tree in nature.
In order to more accurately model trees we need to take into account the branch width, height, branching angles. There are two different approaches to solving this added complexity. One would be to increase the complexity of the L-system grammar and the other would be to increase the complexity of the interpretation of the L-system.
For instance defining an complex L-system grammar with a less complex interpreting system can give a huge amount of flexibility to define parameters that can accurately define exactly how the L-system should be interpreted, and because the complexity is with the L-system rewriting you also have the control of being able to change the L-system rules.

A parametric l-system be represented as the following:

```
n = 8

R 1.456
r1 85
wr 0.707

w : A(5)

p1 : A(w) : * : F(1)!(w)[/(r1)A(w*wr)][\(r1)A(w*wr)]
p2 : F(s) : * : F(s*R)
```

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

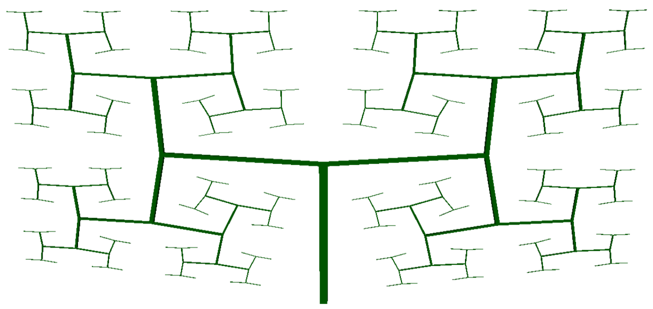


Figure 2.3: 3D Parametric L-system.

Similarly to the 2D L-system in section 2.1, n refers to the number of iterations that we would like to rewrite the L-system. w refers to the Axiom string, the #define R 1.456 states that there

is a constant number that will be used somewhere in the production rules, with the name R and the value 1.456. P1, P2 refer to the production rules. The 3D L-system introduces the concept of a module.
A module is an instruction or variable which has zero or more parameters. The Axiom A(5) is a module with one parameter which is the number 5. A parameter can either be a number, variable or even an expression of variables and numbers. For instance A(a + 1, a * b) is a valid module A with two parameters a + 1 and a * b, where a and b are variables, however this is only a valid module if the value of a and b can be determined. Each module is treated as a single instruction, it will either be overwritten when matched with a production rule or the expression of each parameter is evaluated and is left unchaged for use when interpretted.
Each production rule is made up of four parts the name, the predecessor module, the condition and the successor modules, each part is separated by a colon. Therefor the predecessor for p1 is A(w), the condition is a '*' which means that in this case there are no conditions, and the successor is F(1)!(w)[/(r1)A(w*wr)][\(r1)A(w*wr)]

Initially we iterate through the Axiom modules and compare them to the production rule. A match is determined if they meet three criteria.

• 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.

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.

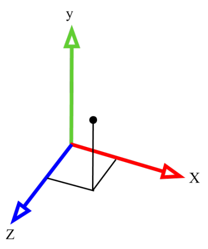


Figure 3.1: Point in 3D space shown using cartesian coordinates.

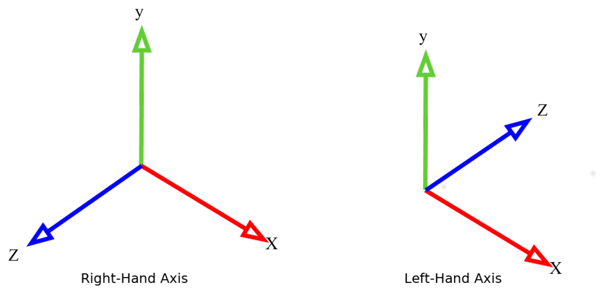


Figure 3.2: Right Hand and Left Hand Coordinate Systems.

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)

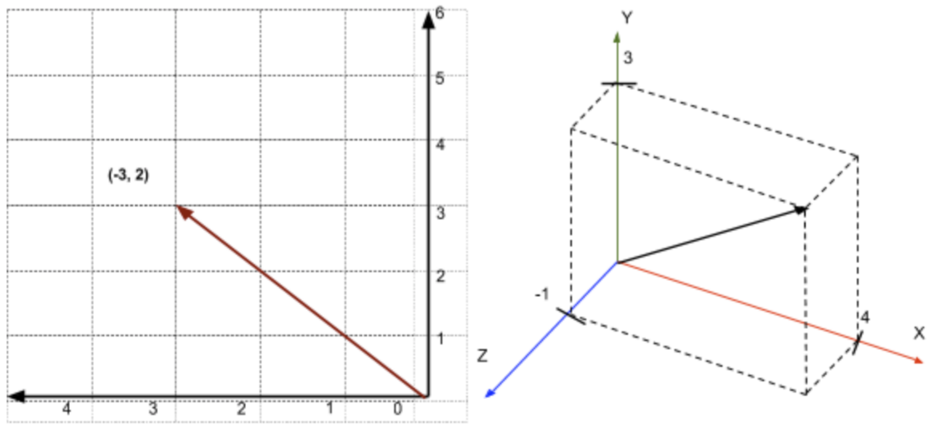


Figure 3.3: 2D Vector representing the vector at (-3, 2) And 3D Vector (4, 3, -1).

3.2.1 Vector Multiplication

3.2.2 Vector Addition and Subtraction

3.2.3 Dot and Cross Product

3.3 Matrices

3.4 Quaternions

3.4.1 Quaternion Multiplication

3.4.2 Conjugate and Inverse

Chapter 4

Implementation

Introduction to the implementation section

4.1 Language and environment

To understand what programming language and environment will be best suited for this project, I first provide the technical requirements that will need to be met. The programming language will need to be relatively fast when processing the rules of the L-systems and when rewriting these strings according to those rules.

The program will also need to interpret the strings generated by the L-system rules and be able to generate a three dimensional representation of that L-system. This representation will need to be intuative and will have to allow me to examine it from different perspectives. In order to make the representation intuative, the 3D representation should be rendered in real time at multiple frames per second. And the user should be able to use a computer mouse and keyboard to move around the 3D world.

Due to these demands have decided to use the C and C++ programming language. Due to its and thoroughly tested in built standard template library, I can count on it to be reliable and fast enough for the purpose of this project. It will also allow me to use other very useful libraries for 3D graphics such as OpenGL which is also written in C/C++. I will be speaking in more detail about these details of these libraries in later sections.

In order to create a window and provide the environment for writing pixels to the screen I will make use of the Graphics Library Framework (GLFW). Some useful mathematics functions and facilities can be found in the OpenGL Matematics Library (GLM) and possibly the most important for rendering in 3D is the Open Graphics Library or OpenGL for short. All of these libraries together will provide me with a strong foundation of tools that I can use to approach the practical aspect of this project.

4.1.1 C/C++ Programming Language

The C programming language developed by Dennis Richie and Bell Labs in 1972 has been one of the most popular programming languages for a number of decades now [Ritchie et al., 1975]. The

C language was then extended upon by Bjarne Stroustrup in 1985 to create the C++ programming language [Stroustrup, 2000]. I have included both C and C++ as the programming languages that I will be using, as they are very closely related and C code can be compiled using the C++ compiler. For the most part I will be writing C++ code and making use of its object oriented features. However, their are instances when it will be more convenient to write C code or make use of a C library.

4.1.2 Standard Template Library (STL)

The STL in C/C++ provides a number of useful functions, data structures and algorithms that have been extensively tested for both reliability and efficiency. The most common features I will be using are strings, vectors, stacks and input and output. It is possible that in some cases it may be more efficient to use custom data structures for the most part the STL functions will be more than good enough. [Horton, 2015]

4.1.3 Open Graphics Library (OpenGL)

OpenGL is a 2D and 3D graphics API
talk about GLSL [Movania et al., 2017]

4.1.4 OpenGL Mathematics Library (GLM)

4.1.5 Graphics Library Framework(GLFW)

4.1.6 Git Version Control

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.

4.2 L-system Generator

The purpose of the L-system generator is to read a file containing 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.

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.

4.2.1 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> → number
<expression> → (<expression> )
<expression> → <expression> + <expression>
<expression> → <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> → 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 ∈ followed by an | states that it can either be nothing or another statement.

4.2.2 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_]*
```

4.2.3 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 productions. Once we have all of this information, we would like to interpret that information a number of times.
For instance we may firstly want to generate up to the fifth generation rewrite, this requires using the information to rewrite five times, but then we may want to instead generate up to the tenth generation. We do not want to have to throw all of this information away and start from scratch so instead we can go from the current state of the interpreter and just rewrite another five times, if we then would 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 4.2.4 and 4.2.5.

4.2.4 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.

4.2.5 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].

4.3 Interpreting the L-system Instructions

4.3.1 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);
```

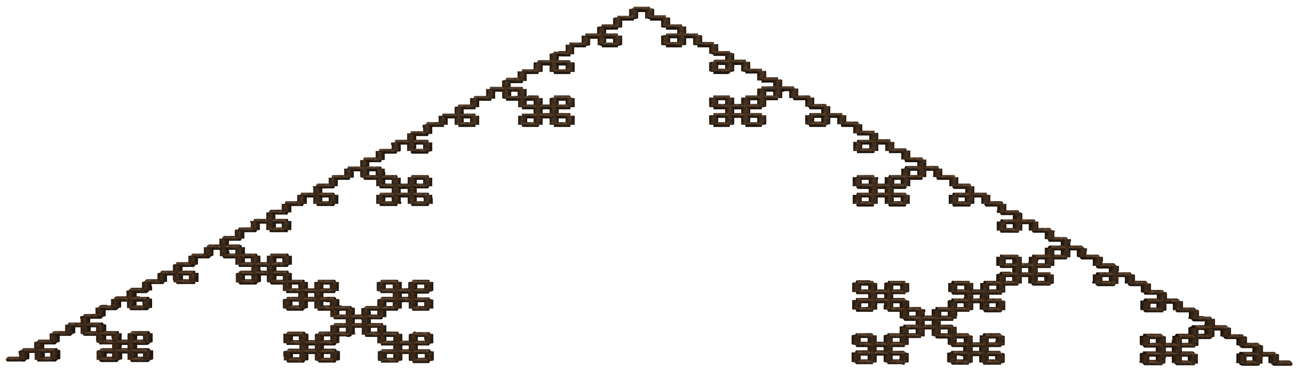


Figure 4.1: Koch Curve.

Sierpinski Triangle:
Alphabet: A, B
Constants: +, -
Axiom: A
Angle: 60°
Rules:
A → B-A-B
B → A+B+A

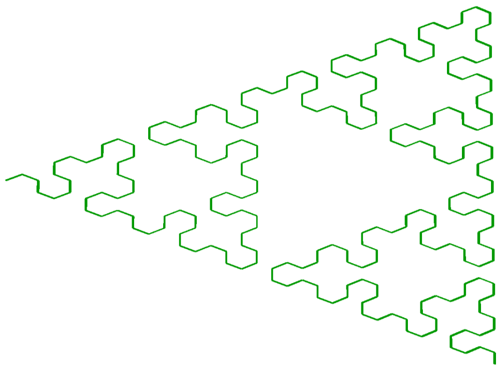


Figure 4.2: Sierpinski Triangle.

Dragon Curve:
Alphabet: F, X, Y
Constants: +, -
Axiom: FX
Angle: 90°
Rules:
X → X+YF+
Y → -FX-Y

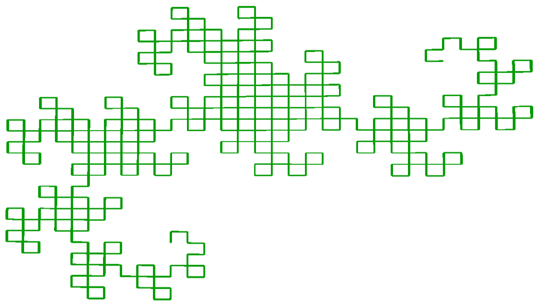


Figure 4.3: Dragon Curve.

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

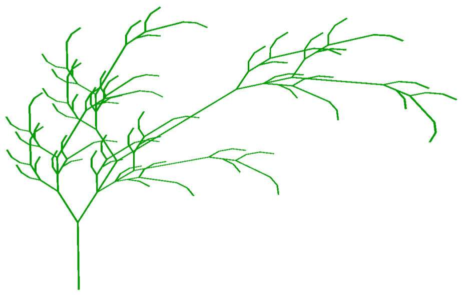


Figure 4.4: Fractal Plant.

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

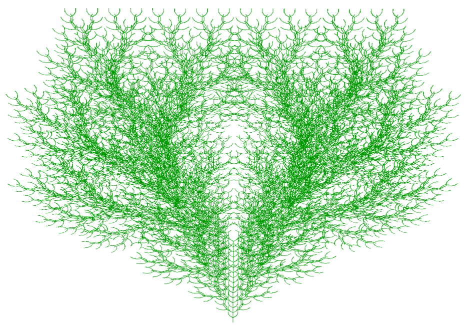


Figure 4.5: Fractal Bush.

4.3.2 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 5

Findings and Data Analysis

Chapter 6

Discussion

Chapter 7

Conclusions

Acronyms

2D	Two Dimensional. 13
3D	Three Dimensional. 13, 14
API	Application Programming Interface. 17
BNF	Backus-Naur Form. 3, 18, 19
CFG	Context-Free Grammar. 18, 21
Flex	Fast Lexical Analyzer Generator. 21
GLFW	Graphics Library Framework. 3, 16, 17
GLM	OpenGL Mathematics Library. 3, 16, 17
GLSL	OpenGL Shading Language. 17
STL	Standard Template Library. 3, 17

Glossary

C/C++ Refers to the C and C++ programming languages. 16

Lexer A computer program that performs lexical analysis. 21

OpenGL The Open Graphics Library is a cross-platform, cross-language application programming interface used in creating graphics applications. 16, 17

Parser A computer program that performs parsing. 21

Appendix A

Appendix

A.1 Appendix 1

A.2 Bibliography

Bibliography

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

[Horton, 2015] Horton, I. (2015). *Using the C++ Standard Template Libraries*. Apress.

[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, 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.

[Ritchie et al., 1975] Ritchie, D. M., Kernighan, B. W., and Lesk, M. E. (1975). *The C programming language*. Bell Laboratories.

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

[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.