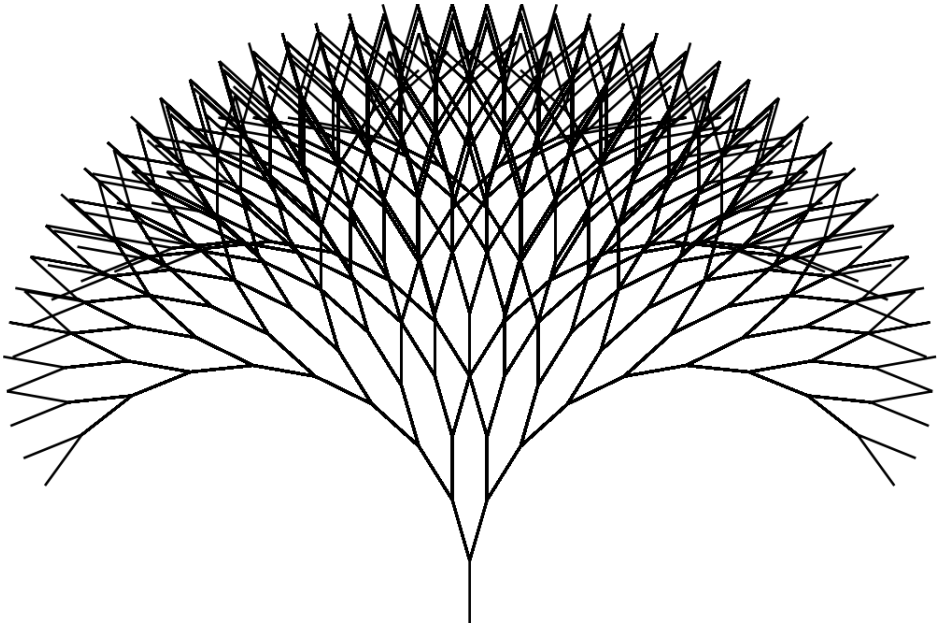
Procedural Plant Generation and Simulated Plant Growth

Massey University



Matthew Crankshaw

25 February 2019

Acknowledgements

Abstract

Contents

List of Figures

Chapter 1

Research Description

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^{th} of February 2019 through to the 20^{th} of February 2020.

1.5 Structure of Thesis

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 to describe the development of simple organisms such as algae and bacteria. 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 formal grammar made up of an *alphabet* of symbols which are put together into strings, a set of rules is used to determine whether a symbol in the string should be rewritten with another symbol or string. What we end up with is a string of symbols which we can refer to as a set of states, for each state the rules determine what symbols to rewrite and what they should be replaced with or if they should be replaced at all.

In section 2.1 below, I will be going into detail about a simple type of L-system called a Deterministic 0L-system. 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 is known as a deterministic 0L systems, where the string refers to the sequence of cellular states and the term '0L system' abbreviates 'Lindenmayer system with zero-sided interactions'. With D0L systems there are only three major parts. There is a set of symbols known as the (*alphabet*), the starting string or (*axiom*) and the state transition rules (*rules*). The alphabet is a set of states. The starting string or *axiom* is the starting point containing one or more states. The transition rules dictate whether a state should remain the same or transition into a different state, remain the same or even disappear. [Prusinkiewicz and Hanan, 2013].

Below is an example of a deterministic 0L system:

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

B → A

The symbol → can be verbalised as "replaced by". Therefore, the first rule is said to be, string 'A' is replaced by string 'AB' and the second rule is said to be 'B' is replaced by the string 'A'. To start we take the first state in the *axiom* which, in this case is the symbol 'A', we then check it against the first rule which is 'A', if the current state matches the rule state we replace 'A' with whatever the rules successor is, which is 'AB'. We would then move onto the next state in the axiom, however there is only one state in the axiom, 'A' so we are finished with the first generation. The states 'AB' then becomes the new starting string for the first generation. We can then continue by matching the rules once again to the new starting string. Below I have shown the string for each generation up to the sixth generation.

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

This rewriting of strings using a set of rules is ultimately the underlying concept behind L-systems. There are several improvements that can be made to this type of L-system in order to accommodate for more complex and intricate structures. I will be talking about these in more detail in the following sections, however some important improvements are: constants, variables, branching constructs, parametric l-systems, conditional rules and random values. //

An example of how an L-system can represent a real-life biological structure would be Prusinkiewicz and Lindenmayer's simulation of a blue-green bacteria known as *Anabaena catenula*

Prusinkiewicz and Lindenmayer created the following DOL-system representation shown below in the following grammar:

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

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 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]

The first five generations can be written as follows:

0.) a_r
1.) $a_l b_r$
2.) $b_l a_r a_r$
3.) $a_l a_l b_r a_l b_r$
4.) $b_l a_r b_l a_r a_r b_l a_r a_r$
5.) $a_l a_l b_r a_l a_l b_r a_l b_r a_l a_l b_r a_l b_r$

2.2 Constants and Variables

Constants are symbols or states which don't have any significant value during the rewriting process and therefore will remain the same between generations however they do have significance when the final string is being interpreted and furthermore represented. There are a number of constants that have a fixed meaning when interpreted and are therefore known as commands. These values are:

- 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.
- ^: Roll to the right specified angle.
- &: Roll to the left by a specified angle.

The question then remains why they should be interpreted using these commands and how are the instructions interpreted? As with any grammar, there are numerous ways of interpreting it. 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 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 2.1

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.

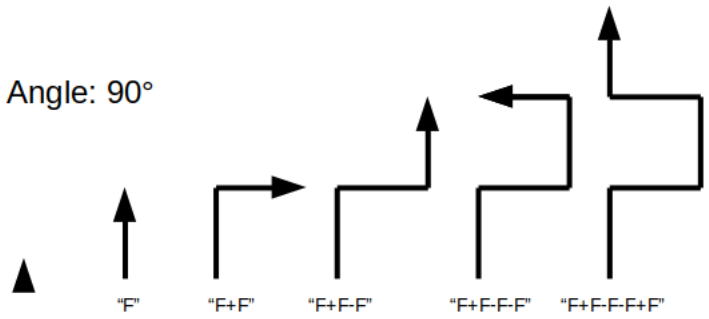


Figure 2.1: Diagram showing a turtle interpreting simple L-system string.

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.

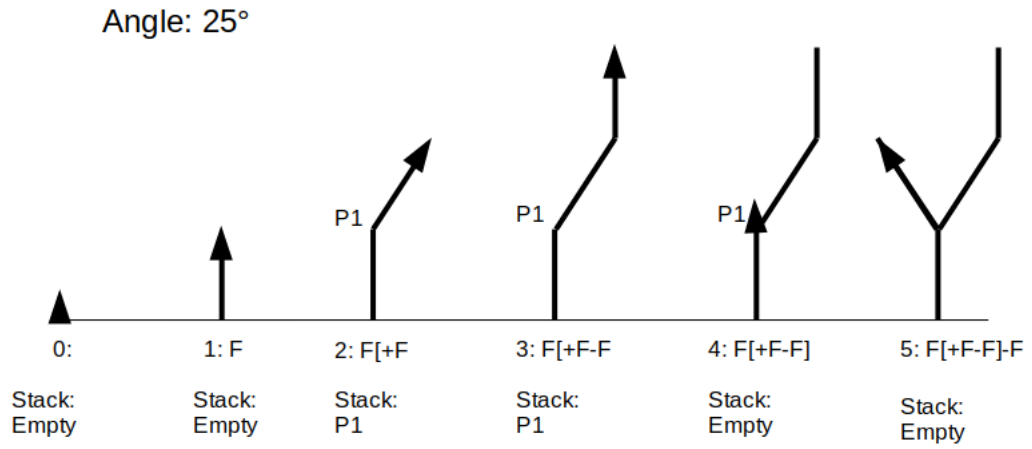


Figure 2.2: Diagram showing a turtle interpreting an L-system incorporating branching.

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 $+,\ -,\ *,\quad \wedge$. 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:

$$G\ = (V, \Sigma, \omega, P) \tag{2.1}$$

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:

$$P\ \subset\ (V\ \times\ \Sigma^*)\ \times C(\Sigma)\ \times\ (V\ \times\ E(\Sigma))^* \tag{2.2}$$

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 2.4.3.
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 Controlling the Width and Length of Branch Segments

talk about the ! symbol for changing the width of a branch somewhere (not using the ' symbol as per text) and the use of the first parameter determining length
A parametric l-system be represented as the following:

$$
\begin{aligned}
& n = 8 \\
& \omega \ : A(5) \\
& p_1 \ : A(w) \quad : * \rightarrow \ F(1)!(w)[+A(w \ * \ 0.707)][-A(w \ * \ 0.707)] \\
& p_2 \ : F(s) \quad : * \rightarrow \ F(s \ * \ 1.456)
\end{aligned}
\tag{2.3}
$$

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

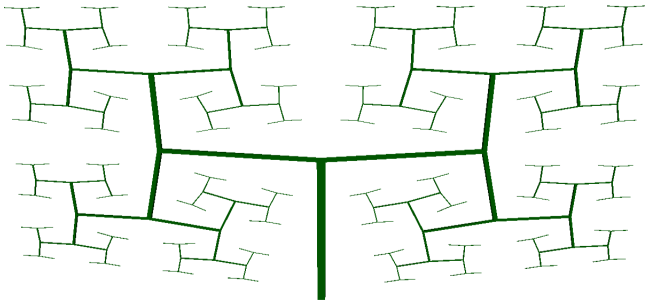


Figure 2.3: 3D Parametric L-system.

2.4.3 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:

$$
\begin{aligned}
& n = 5 \\
& \omega \ : A(0)B(0,4) \\
& p_1 \ : A(x) \quad : \ x \ > \ 2 \ \rightarrow \ C \\
& p_2 \ : A(x) \quad : \ x \ < \ 2 \ \rightarrow \ A(x \ + \ 1) \\
& p_3 \ : B(x, \ y) \ : \ x \ > \ y \ \rightarrow \ D \\
& p_4 \ : B(x, \ y) \ : \ x \ < \ y \ \rightarrow \ B(x \ + \ 1, \ y)
\end{aligned}
\tag{2.4}
$$

The L-system above in 2.4 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 2.5.

$$
\begin{aligned}
g_0 &: \ A(0)B(0,\ 4) \\
g_1 &: \ A(1)B(1,\ 4) \\
g_2 &: \ A(2)B(2,\ 4) \\
g_3 &: \ C\ B(3,\ 4) \\
g_4 &: \ C\ B(4,\ 4) \\
g_5 &: \ C\ D
\end{aligned}
\tag{2.5}
$$

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

Growth of Flower Using Conditions:

$$
\begin{aligned}
& n = 2,\ 4,\ 6 \\
& \#object\ F\ BRANCH \\
& \#object\ L\ LEAF \\
& \#object\ S\ SPHERE \\
& \#define\ r\ 45 \\
& \#define\ len\ 0.5 \\
& \#define\ lean\ 5.0 \\
& \#define\ flowerW\ 1.0 \\
& \omega \ \ :!(0.1)I(5) \\
& p_1 \ : I(x)\ :\ x\ >\ 0\ \ \rightarrow\ F(len) - (lean)[R(0,100)]F(len)[R(0,100)]I(x-1) \\
& p_2 \ : R(x)\ :\ x\ >\ 50\ \rightarrow\ \ -(r)/(20)!(2.0)L(2)!(0.1) \\
& p_3 \ : R(x)\ :\ x\ <\ 50\ \rightarrow\ \ -(r)\backslash(170)!(2.0)L(2)!(0.1) \\
& p_4 \ : I(x)\ :\ x\ <=\ 0\ \rightarrow\ F(len)!(flowerW)S(0.3)
\end{aligned}
\tag{2.6}
$$

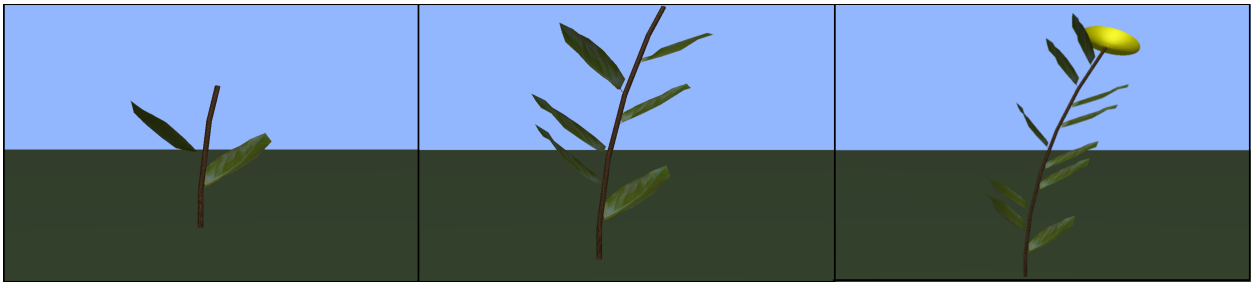


Figure 2.4: Condition Statements Used to Simulate the Growth of a Flower. 2nd Generation on the Left, 4th Generation in the Center and 6th Generation on the Right

2.4.4 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);

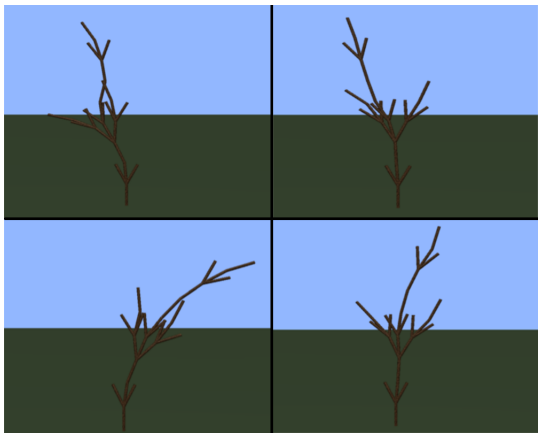


Figure 2.5: Different Variations of the Same L-system with Randomness Introduced in The Angles.

In figure 2.5 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).

2.4.5 Stochastic Rules in the L-system

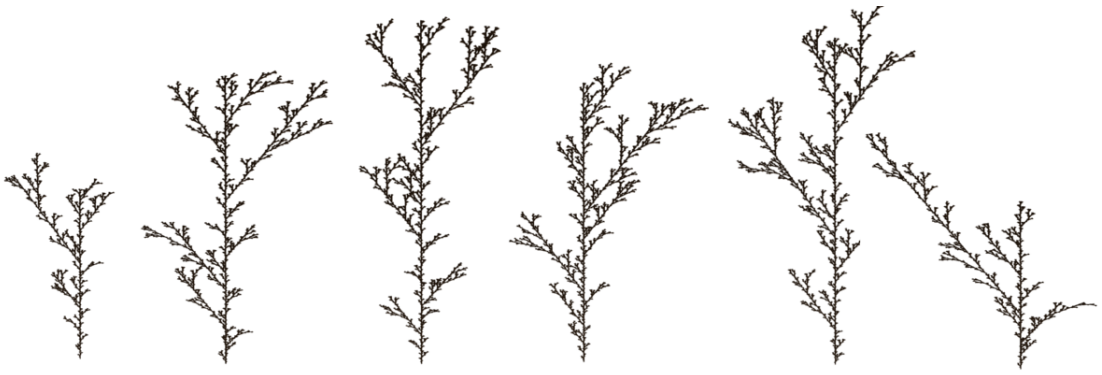


Figure 2.6: Representation of an L-system with a probability stochastic with a 33.33% chance for each rule.

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.

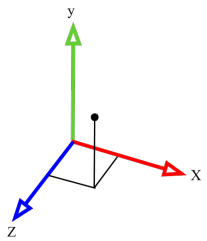


Figure 3.1: Point in 3D space shown using cartesian coordinates.

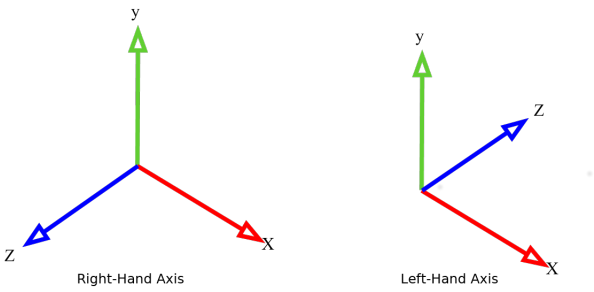


Figure 3.2: Right Hand and Left Hand Coordinate Systems.

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)

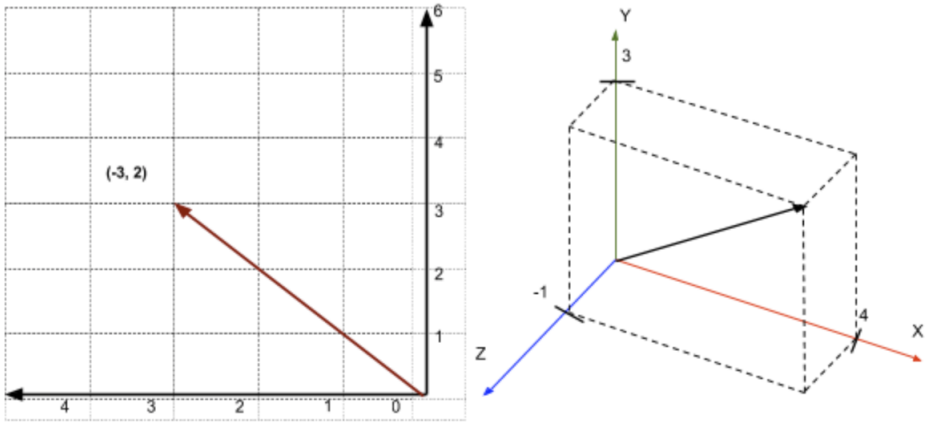


Figure 3.3: 2D Vector representing the vector at (-3, 2) And 3D Vector (4, 3, -1).

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 × 3 matrix we tend use a 4 × 4 matrix known as an Atomic Transform Matrix. An atomic Transfom matrix is a concatination of four 4 × 4 matrices: translations, rotations, scale and shear transforms. Resulting in a 4 × 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 × 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

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
$M = \Pi\ r^2 h$
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
$I = \frac{1}{3} M L^2$
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

Implementation

Introduction to the implementation section

5.1 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.

5.1.1 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.

5.1.2 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]

5.1.3 Open Graphics Library (OpenGL)

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

5.1.4 OpenGL Mathematics Library (GLM)

5.1.5 Graphics Library Framework(GLFW)

5.1.6 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.

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> → 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 $\in$ 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> | "+" | "−" | "/" | "\" | "^" | "&" | "!"
| <variable> (<paramAx> <paramListAx>)
| +(<paramAx> <paramListAx>)
| -(<paramAx> <paramListAx>)
| /(<paramAx> <paramListAx>)
| \(<paramAx> <paramListAx>)
| ^(<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);

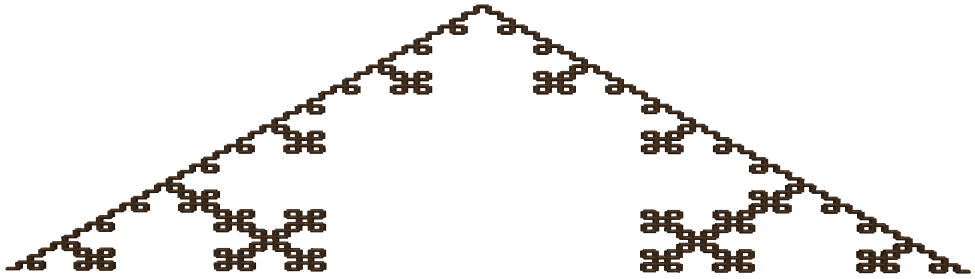


Figure 5.1: Koch Curve.

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);

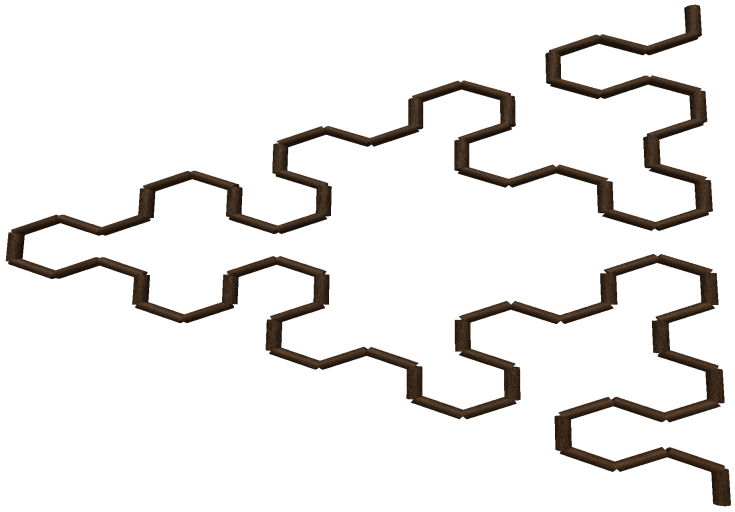


Figure 5.2: Sierpinski Triangle.

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]

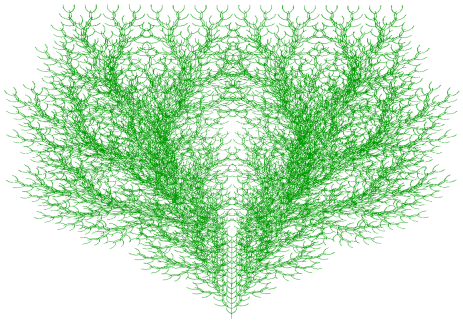


Figure 5.4: Fractal Bush.

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.

5.5 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:

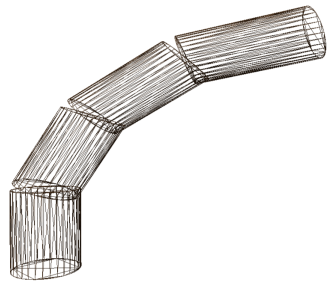


Figure 5.5: Example of the continuity problem faced with stacked branching with a 25° bend per joint.

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.

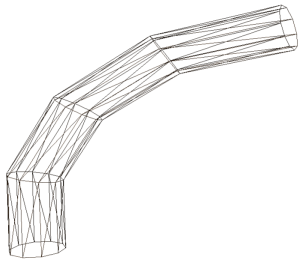


Figure 5.6: Example of linked branching with a 25° bend per joint.

Chapter 6

Findings and Data Analysis

Chapter 7

Discussion

Chapter 8

Conclusions

Acronyms

2D Two Dimensional. 18

3D Three Dimensional. 18, 19

API Application Programming Interface. 25

BNF Backus-Naur Form. 4, 26, 27

CFG Context-Free Grammar. 26, 29

Flex Fast Lexical Analyzer Generator. 29

GLFW Graphics Library Framework. 4, 24, 25

GLM OpenGL Mathematics Library. 4, 24, 25

GLSL OpenGL Shading Language. 25

STL Standard Template Library. 4, 25

Glossary

C/C++ Refers to the C and C++ programming languages. 24

Lexer A computer program that performs lexical analysis. 29

OpenGL The Open Graphics Library is a cross-platform, cross-language application programming interface used in creating graphics applications. 24, 25

Parser A computer program that performs parsing. 29, 30

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.

[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, 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.

[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.