VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

ENGR489 Final Report

Bryony Gatehouse

Supervisor: Zohar Levi

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

The purpose of this project is to demonstrate the skills I have acquired over the last four years by implementing a SIGGRAPH paper that describes a method to generate a realistic tree.

Contents

1 Introduction **1**

2 Previous Work **2**

3 Background **3**
3.1 Lindenmayer System 3
3.1.1 Turtle Interpretation of Strings 3
3.1.2 Branching Structures 4
3.1.3 3D L-systems 4
3.1.4 Stochastic Lindenmayer System 4
3.1.5 Context-Sensitive Lindenmayer System 5
3.2 Palubicki Tree 6
3.2.1 Terminology 6
3.2.2 Calculation of Environmental Input 6
3.2.3 Calculation of Bud Fate 7
3.2.4 Orientation of New Shoots 8
3.2.5 Shedding of Branches 8
3.2.6 Branch Diameter 9

4 Design **10**
4.1 3D Mesh 10
4.1.1 Modular Programming 10
4.2 Lindenmayer System 10
4.2.1 Model View Design Pattern 10
4.3 Palubicki Tree 10

5 Implementation **12**
5.1 3D Mesh 12
5.2 Lindenmayer System 13
5.3 Palubicki Tree 14
5.4 Godot 15

6 Evaluation **17**
6.1 Lindenmayer System 17
6.2 Palubicki Tree 18
6.3 Comparing Tree Models 23

7 Conclusion **25**
7.1 Future Work 25

Chapter 1

Introduction

Trees are critical to sustaining life on earth as they provide the oxygen we breathe. From trees, we can also acquire materials that have become a part of our everyday lives. In addition to these functions, they are an inherent part of a scene and thus are important when creating a realistic scene for a movie, game, or urban planning.

The purpose of this project is to demonstrate the technical skills that I acquired over the last 4 years at Victoria University of Wellington. To accomplish this, I chose, as a challenge, to implement a state-of-the-art SIGGRAPH paper that describes a method to generate a realistic tree. Implementing such a paper poses two challenges: the theoretical challenge of learning the background and comprehending the paper; and the technical challenge of implementing the paper.

Earlier methods for implementing tree models are built on the Lindenmayer system [1] which uses a recursive and repetitive process to build a simple branching structure. Thus, the algorithm is an important foundation for tree modelling. As part of the challenge was to learn about the background of computer-generated tree models, I chose to start my journey by implementing the Lindenmayer System described in [2].

This project will focus on the paper by Palubicki [3] which incorporates environmental conditions to lead to realism never achieved in previous works. This paper is important since later papers mostly describe methods at a high level, citing this paper as the basis, referring most implementation concerns to it [4], [5], [6].

It was discovered that the earlier tree modelling methods, which are based on the assumption that trees have a repetitive and recursive structure, only provide the branching patterns[7]. Another important feature that affects a tree's look is the distribution of buds from which branches arise with appears locally irregular but produce a well-balanced, recognizable tree form. Sachs [7] proposed an approach to producing a tree form emphasising the self-organising character of tree development in which each bud and branch considers the best options for the overall tree development. The paper by Palubicki introduces a method that considers three elements of tree development: local control of branch geometry, competition of buds and branches for space or light, and regulation of this competition through an internal signalling mechanism [3].

Chapter 2

Previous Work

One of the first algorithms for modelling trees was written by Lindenmayer [1] who described a tree as a self-generating, recursive structure that uses branching patterns and angles. This process, originally designed to represent structures of single multi-cellular organisms, has become a foundation for the development of tree modelling. It was extended by [8] who proposed a method to translating the symbols into movements for a turtle that could create a visual representation of the pattern by drawing a line behind them. This was later expanded again by [9] who introduced the third dimension.

An alternative concept, where trees are self-organising structures, was written by Ulam [10]. In this concept, the branch growth is determined by competition space. A unit is grown into if they are adjacent to a single existing unit of the previous generation. Other rules can be added to change the shape, including forbidding new growths from touching another growth (either already grown or in consideration) on either a side or corner. Originally described in terms of a 2D grid of cells, the idea was extended to 3D voxel spaces and augments with constructs needed for realistic image synthesis by [11], [12], [13], [14], and [15].

Algorithms using continuous representations of space were proposed by [16], and in computer graphics by [17] and [18]. Space propagation, proposed by [18] and expanded by [3], represents the empty space as attraction points with the optimal growth direction for a bud aiming towards the area with the most space (or attraction points).

[3] introduced the idea of shadow propagation as another form of environmental input. Using a 3D grid of voxels to represent the space, a bud casts a pyramidal shadow beneath it, with each layer of voxels receiving less shadow. As buds tend to grow towards the light, the optimal growth direction for a bud is towards the neighbouring voxel with the lowest shadow value.

Chapter 3

Background

3.1 Lindenmayer System

A common method for generating a tree is based on the Lindenmayer System [2] which is a mathematical theory that uses the concept of rewriting. Given a string of characters, a rewriting rule replaces a certain character with a new string, extending the string.

An example of rewriting is the Koch snowflake, one of the oldest known fractals (Figure 3.1). Each line represents a character in the string - or shape. The rewriting rule, each iteration, replaces each line segment with a pattern of four new lines, expanding the shape.

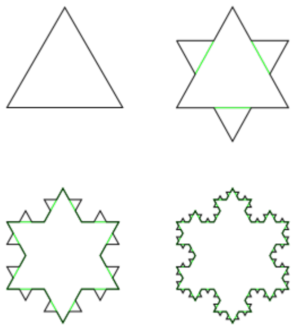


Figure 3.1: The first four iterations of the snowflake

An example of a simple Lindenmayer System is expanding a string built from the two letters a and b where the letters can appear multiple times in the string. For this System, there are two rules: $a \to ab$ and $b \to a$. Figure 3.2 shows the progression of the pattern, with both rules applied during an iteration. The third iteration, where the input pattern is ab, results in the a letter being replaced with the pattern ab while the b is replaced with an a.

An important feature of the Lindenmayer System is the application of rules simultaneously whereas previous rewriting mechanisms would apply the rules sequentially. If the rule for the Koch snowflake was applied sequentially, it would keep applying the rule to the first available line indefinitely.

b	a
a	ab
ab	aba
aba	$abaab$

Figure 3.2: The first 4 iterations of a Lindenmayer System.

3.1.1 Turtle Interpretation of Strings

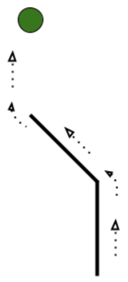


Figure 3.3: The turtle movements of $F + F - f$

An extension to the Lindenmayer System was proposed by [8] which translates the pattern created by a Lindenmayer System into a picture.

The turtle's state is defined as a triplet (x, y, α), where the coordinates (x, y) represent the turtle's position, and the angle α is the direction the turtle is facing. The turtle can then respond to commands represented by the following symbols (Figure 3.3):

- F Move forwards by the length d to $x' = x + d\cos\alpha$ and $y' = y + d\sin\alpha$. A line is drawn between the original position and the new position.
- f Move forwards by the length d without drawing a line.

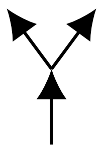
$+$	Turn left by angle δ, changing the state to $(x, y, \alpha + \delta)$.
$-$	Turn right by angle δ, changing the state to $(x, y, \alpha - \delta)$.

3.1.2 Branching Structures

A branch can be described as a point where a line splits in multiple directions. To replicate this in the Lindenmayer System, a method for returning to a prior state is implemented. Two new symbols are introduced which saves a state to the stack where it can be retrieved at a later date:

[	Push the current state to a stack.
]	Pop a state from the stack and make it the current state.

An example is shown in Figure 3.4. The first command moves the state forwards, drawing a line behind it. The state is then saved to a stack where it can be accessed later. Once the left branch is created, the state can be reverted to its previous state by replacing it with the state saved to the stack. This allows it to create the right branch.



$F[+F][-F]$

Figure 3.4: Example of a Branching Structure built from a Lindenmayer System

3.1.3 3D L-systems

A method for extending the Turtle Interpretation of Strings to create a 3D image of a Lindenmayer System by [9]. This is achieved by representing the current orientation in space as three vectors: the direction the turtle is heading, its direction left, and its direction up. To transform these vectors, five new symbols are introduced that use the traditional rotational matrices to calculate the next location the turtle will travel to:

$+$	Turn left by angle δ, using rotational matrix $R_U(\delta)$
$-$	Turn right by angle δ, using rotational matrix $R_U(-\delta)$
&	Pitch down by angle δ, using rotational matrix $R_L(\delta)$
$\wedge$	Pitch down by angle δ, using rotational matrix $R_L(-\delta)$
$\backslash$	Roll left by angle δ, using rotational matrix $R_H(\delta)$
/	Roll right by angle δ, using rotational matrix $R_H(-\delta)$
$\vert$	Turn around, using rotational matrix $R_U(180^\circ)$

3.1.4 Stochastic Lindenmayer System

The Lindenmayer System described in this section will always produce the same result, given the same inputs and rules. A method was proposed that introduces variation to each rewriting step, called the Stochastic Lindenmayer System [2].

Unlike the original Lindenmayer System, the Stochastic Lindenmayer System allows each letter to have multiple rewriting rules, each with a probability of occurring. The probabilities of rules for each letter should add to 1, ensuring that a rule is picked for each rewriting step. As seen in Figure 3.5, the letter F has a $\frac{1}{3}$ of being rewritten by rule p_1.

$$\begin{aligned}
w:&\quad F\\
p_1:&\quad F \xrightarrow{.33} F[+F]F[-F]F\\
p_2:&\quad F \xrightarrow{.33} F[+F]F\\
p_3:&\quad F \xrightarrow{.34} F[-F]F
\end{aligned}$$

Figure 3.5: Example of a Stochastic Lindenmayer System.

3.1.5 Context-Sensitive Lindenmayer System

Another extension to the original Lindenmayer System considers the surrounding letters in the pattern when choosing a rewriting rule [2]. The rules $0 < 0 > 1 \rightarrow 1$ states that if a 0 is surrounded by 0 on the left and 1 on the right, it will be replaced by 1.

The method assumes that at each end of the string is an invisible 1 so that the first and last letter can be rewritten. Another assumption is that, with a bracketed system, the first letter inside the brackets considers the letter to its left outside the brackets as its neighbour. However, the last letter inside the brackets considers the invisible 1 at the end of the string rather than the letter outside the brackets as it is the end of a branch.

$$\begin{aligned}
0<0>0 &\;\rightarrow\; 0\\
0<0>1 &\;\rightarrow\; 1[1]\\
0<1>0 &\;\rightarrow\; 1\\
0<1>1 &\;\rightarrow\; 1\\
1<0>0 &\;\rightarrow\; 0\\
1<0>1 &\;\rightarrow\; 11\\
1<1>0 &\;\rightarrow\; 1\\
1<1>1 &\;\rightarrow\; 0
\end{aligned}$$

n	L(n)
0	1
1	0
2	11
3	00
4	01[1]
5	111[0]
6	000[11]
7	001[1][10]
8	01[1]1[0][111]
9	111[0]0[11][000]
10	001[11]11[10][001[1]]

Figure 3.6: Example of a Context-Sensitive Lindenmayer System.

3.2 Palubicki Tree

The algorithm described by Palubicki [3] incorporates environmental conditions to produce a more realistic tree model.

3.2.1 Terminology

The algorithm proposed by [3] grows the tree in levels. Starting from the root, branches are grown from buds, though the decision of when and how the branches are produced depends on the environment.

I describe a tree as a branching structure. The point where a bud was - or is - attached to the stem is termed a *node*. The part of the stem between two nodes is an *internode* and is of length L. A node can grow two nodes (depending on environmental conditions): the *terminal bud* continues the main stem, and the *lateral bud* produces a new branch. An internode with attached leaf and bud forms a *metamer*. A sequence of metamers grown in one season is called a *shoot*. A collection of internodes, which are created by terminal buds (excluding the original bud), is called an *axis* or a *branch*.

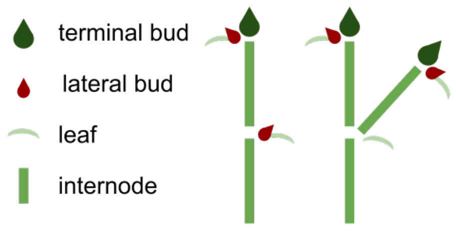


Figure 3.7: Diagram of a shoot

3.2.2 Calculation of Environmental Input

In each iteration, the environmental process estimates the availability and quantity of the space surrounding each bud (Q) and the optimal direction of growth ($\vec{V}$).

Space Colonisation

The method described by [3] is an extension on the space colonisation algorithms for generating leaf venation patterns [19] and trees [18]. In this algorithm, only one bud per node is considered, as the terminal bud must grow before the lateral bud.

In this method, each bud is surrounded by a spherical occupancy zone of radius ρ and has a conical perception volume characterised by the perception angle θ and distance r. The empty space is populated with attraction points kept in a set S which represent the space available for tree growth. When a new bud is created, any attraction points within the bud's occupancy zone are deleted from the set S. At the beginning of an iteration, the buds compete for the remaining attraction points. An attraction point within the perception volume of a single bud becomes associated with that bud. Any attraction points within the perception volume of multiple buds becomes associated with the closest bud. A bud can grow if it is associated with at least one attraction point ($Q = 1$) otherwise the bud has no available space ($Q = 0$). Finally, the optimal growth direction $\vec{V}$ is calculated as the normalised sum of

the normalised vectors towards all markers associated with the bud.

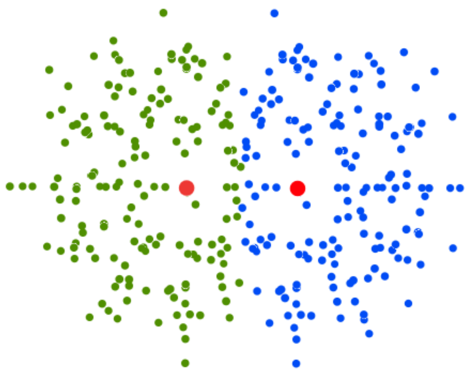


Figure 3.8: Two Buds competing for Resources

Space Colonisation: Reducing Attraction Points

The space colonisation method expects the space the tree will grow into to be populated with attraction points at the beginning of the simulation. I chose to implement an adaptation of this method, adding more attraction points with each new bud.

Each bud is associated with a 'cloud' of attraction points. These points are created at the same time the bud is created. The cloud is then trimmed to remove any outside the perception volume or within the occupancy zone. Finally, after creation, the new bud is compared to its neighbours. During this process, any attraction points that fall within another bud's occupancy zone is removed and an attraction point that is closer to another bud is removed.

An example can be seen in Figure 3.8 where two buds (the red dots) are competing for resources. The blue dots are attraction points associated with the right bud while the green dots are attraction points associated with the left bud. This figure shows how the cloud of attraction points is leaning away from the other bud, resulting in the optimal growth direction tending away from other buds.

The benefit of this method is that an initial shape doesn't need to be defined, fewer attraction points need to be compared each iteration, and the tree can keep growing infinitely. Finally, it reduces the computational load on the machine running the program.

Shadow Propagation

Phototropism is the orientation of plants in response to light. To implement this, [14] proposed a method of space propagation that computes a coarse estimate of the exposure of each bud to light.

The space is divided into a grid of voxels. Each voxel has an associated shadow value s. A bud located in voxel (I, J, K) casts a pyramidal shaped shadow to the voxels underneath. The affected voxels have indices $(i, j, k) = (I \pm p, J - q, K \pm p)$, where $q = 0, 1, ...q_{max}$ and $p = 0, 1, ..., q$. An affected voxel's shadow value is increased by $\Delta s = ab^{-q}$, where $a > 0$ and $b > 1$ are user-defined parameters. The light exposure Q of a bud in voxel (i, j, k) with shadow value s can be calculated as $Q = max(C - s + a, 0)$, where C is a constant representing full exposure. The addition of the term a corresponds to the assumption that a bud does not cast a shadow on itself. However, I found that it increased the light exposure past the full exposure represented by C. Thus, the equation I implemented was $Q = max(C - s, 0)$. The optimal growth direction $\vec{V}$ is calculated by considering the neighbouring voxels within the perception volume of the bud and selecting the voxel with the lowest shadow value.

3.2.3 Calculation of Bud Fate

The environmental inputs determine which buds will produce new shoots and how large they'll be. The methods described in [3] focused on implementing methods that would allow apical control, where resources are allocated with a bias towards a certain axis (non, main, or lateral).

Extended Borchert-Honda (BH) model

The BH model was originally proposed to regulate the extent of branching by controlling the distribution of growth-inducing resources to buds [20]. It was adapted in [3] to self-organising trees by using the amount of light received by the buds to guide the distribution of the resource.

The algorithm passes over the tree twice. The first pass gathers information about the amount of light Q that reaches the buds and stores the cumulative values in the nodes. The cumulative amount at the base determines the amount of resources that will be distributed around the tree. The amount of resource v reaching a branching point is distributed between the continuing main axis (v_m) and the lateral branch (v_l) using the equations:

$$v_m = v\frac{\lambda Q_m}{\lambda Q_m + (1-\lambda)Q_l} \quad \text{and} \quad v_l = v\frac{(1-\lambda)Q_l}{\lambda Q_m + (1-\lambda)Q_l}$$

The parameter $\lambda \in [0, 1]$ controls whether resource allocation is biased towards the main axis ($\lambda > 0.5$), not biased ($\lambda = 0.5$), or biased towards the lateral branches. The amount of resource v the bud receives determines the number of metamers produced $n = \lfloor v \rfloor$ and the length of the next internode is calculated as $l = L * (v/n)$ where L is the default length.

Priority Model

The priority model was proposed by [3] as another approach to resource allocation.

Similar to the BH model, information representing the amount of light received by the buds propagates towards the root. However, this information is stored at the base of each axis along with the number of buds along the axis. Supported branches along the axis are represented by their average amount of light (total light divided by the number of buds in a branch). The amount of light received by each bud or branch along the axis is then sorted, yielding order priority lists for each axis. Apical control is simulated by placing the terminal bud (of which there should only be one per branch) at the beginning of this list, irrespective of its light exposure. The resource for an axis is distributed between the supported branches and buds using weights based on their position in the priority list

$$v_i = v\frac{Q_i w_i}{\sum_{j=1}^{N} Q_j w_j}, i = 1, 2, ...N$$

where v is the amount of resource flowing into the axis, N is the number of buds or branches supported by this axis, v_i is the amount of resource allocated to the bud or branch i, and $w_1, ..., w_N$ are the weights (Figure 3.9).

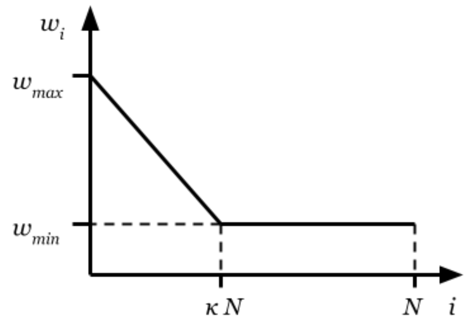


Figure 3.9: The form of the weight function

3.2.4 Orientation of New Shoots

The orientation of a new shoot is, by default, the same as the bud. To represent the response to environmental inputs, the optimal growth direction determined by the environment ($\vec{V}$ in Section 3.2.2) modifies the orientation. Finally, [3] proposed a method to include the effect of gravity on new shoots by introducing a tropism vector. Thus, the actual orientation of a new shoot is calculated as the sum of three vectors: the default orientation, the optimal growth direction, and the tropism vector.

3.2.5 Shedding of Branches

As the shedding of branches is an important component of crown self-organising, I used a method based on the light exposure of buds which was proposed by [21] and brought to computer graphics by [22]. The total amount of light gathered by a branch to number is compared to internodes in a branch. If the ratio drops below the threshold, the branch is considered a liability and is shed.

3.2.6 Branch Diameter

The formula for calculating the branch diameter is a version of a pipe model [23] where each leaf adds a pipe to the tree which extends down the branches to the root. Thus, each leaf contributes an initial diameter value. At a branching point, the diameters are accumulated using the formula $d^n = d_1^n + d_2^n$, where d is the diameter below the branching point, d_1 and d_2 are diameters above the branching point, and n is a user-defined parameter [24].

Chapter 4

Design

4.1 3D Mesh

4.1.1 Modular Programming

Modular Programming was used in the implementation of converting a tree structure to a 3D mesh. The process of creating a 3D mesh contains many layers which are loosely coupled together. One could be replaced or edited with very little impact on another.

For example, the method to create a branch 3D mesh is a static method that chooses which implementation of the branch object it will return. This allows a new branch object to be switched in by changing one method.

4.2 Lindenmayer System

4.2.1 Model View Design Pattern

I chose to implement the Model-View-Controller when creating the 2D representation of the Lindenmayer System. The logic for creating the Lindenmayer System and interpreting the symbols into movements is kept in the Model. Each new position the turtle reaches is passed to the View where it is saved and processed to be shown to the user. Keeping the positions in the View reduces the amount of information being passed from the Model, especially as the Model will never use the positions once calculated. Finally, there is no user interaction with the program, removing the need for a Controller.

The result of this implementation meant converting the Lindenmayer System implementation to a 3D mesh meant just changing the View that it uses to one that creates a 3D mesh rather than a 2D representation.

4.3 Palubicki Tree

Inheritence

Nodes have various behaviours based on their circumstances. Terminal buds, which are situated at the end of branches, will always extend the main branch. Lateral buds, located along the branches, create a new lateral branch. Branching points are the nodes between a lateral branch and the main branch. Finally, a branching point can shed the lateral branch if it is deemed a liability to the tree, retaining the diameter information about the branch.

To implement this behaviour, I created an abstract class that will contain the behaviour of an average node (*Bud*). This behaviour would include the bud's location, the cloud of attraction points and logic for converting them into a direction vector, and the shadow value. Later, I extracted the shadow propagation logic and coordinate information into a separate class, *Marker*, which *Bud* extends. This allows the same logic to be used for both attraction points and nodes.

More specific behaviour was added by extending *Bud* (see Figure 4.1. The *RootBud* class contains the logic for gathering and spreading resources, and other logic required for the tree root. The *ChildBud* class and its sub-classes represent the behaviour of every other bud, including casting a shadow and propagating the diameter.

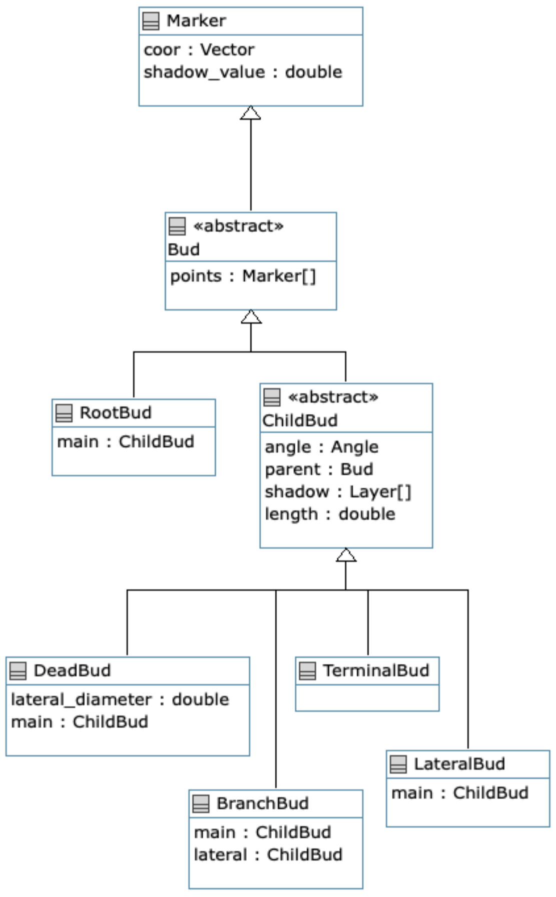


Figure 4.1: UML Diagram of the Bud classes.

Polymorphism

To separate the behaviours of the different types of nodes, I chose to use polymorphism. This was implemented by creating four classes that extend the abstract class *ChildBud* allowing a node to hold a *ChildBud* object as a child and not need to know which behaviour it implements.

Creating a new child always create a *TerminalBud* as the new child should not contain any children of its own. This means that the code for calculating the location of the new bud only need to be in the *TerminalBud* class. When this object is passed a new child of its own, it transforms into a *LateralBud*. The expected vector for the lateral branch is different from the main branch, so the *LateralBud* class overrides the default method in *Bud* to return the correct vector. Adding a child to a *LateralBud* transforms it into a *BranchBud* which holds two children and can no longer grow. The *BranchBud* is the only class that holds a lateral child; thus, it also contains the logic for calculating whether the lateral child is a liability to the tree. If the lateral child is shed, the *BranchBud* transforms into a *DeadBud* which keeps the memory of the lateral branch by remembering its diameter and cannot grow.

Chapter 5

Implementation

This section will go into some more technical details on the implementation and design for this project.

5.1 3D Mesh

Once a tree structure, either the Lindenmayer System or Palubicki Tree, is completed it is transformed into a 3D mesh.

Branch Rotation

The *Branch* class creates a default shape of the given width and length - currently a hexatriacontagon prism - which is then rotated and translated to the correct location. Each default shape is made pointing straight upwards from the origin point; thus, it will rotate and translate around the origin point.

While this method makes creating a mesh branch much easier, it requires a rotation which will rotate the given branch to the correct location. The axis of rotation is calculated as:

$$A = (target - origin) \times (0, 1, 0)$$

and the angle using the equation:

$$\alpha = -\cos^{-1}\left(\frac{target - origin}{|target - origin|} \cdot (0, 1, 0)\right)$$

Branch Connections

If a branch is created made of two segments, with the second at an angle from the first, creating a hexatriacontagon prism and transforming each to the correct location results in a wedge-shaped gap between the two branch segments. This is because there is no connection between the two segments.

I implemented a connection by creating a base for the parent, a base for the child, and joining them together to create a branch segment. Each base is a circle of points that is rotated and translated. Thus, the first branch segment's child base has the same set of points as the parent base for the second branch segment.

To use this method, each branch segment needs knowledge about its parent's coordinates and transformations. Therefore, the information about branch segments is passed in a tree data structure, so a parent can create the mesh segments for its children.

File Format

I chose to export the 3D mesh in an OBJ file format.

The OBJ file format is open source and is widely used for both 3D printing and non-animated 3D models. It has wide export and import capabilities as most CAD software can interpret it correctly and consistently. This means that others using my code would be able to view and use the resulting files easily. Finally, the format is in an ASCII file format which can be read as a text file, making it easier to debug and to create an exporter. This is important as there aren't many exporters for 3D meshes written in Java.

5.2 Lindenmayer System

The Lindenmayer System has two important steps: rewriting the pattern and converting the pattern to a picture.

Each rewriting rule is a relationship between a character and a string ($b \rightarrow ab$). I implemented this using a Map with the character as the key. As each character is only associated with one rule, this implementation could quickly find the corresponding string pattern to replace the character with.

The turtle's state was held in an object, which could be cloned to create a replica to put on the Stack when branching. The resulting pattern is iterated over, with each character being parsed to a specific movement. The state's position and angle can be updated by a movement.

Finally, the resulting points are saved to an external structure which is passed to the View, separating the Logic from the Visuals.

Rotation

In the 3D Lindenmayer System, the turtle is rotated around itself - rather than around the environment origin. Thus, each rotation assumes that the turtle is the origin. However, in relation to the environment, the turtle is not the origin point as each rotation it completes changes its position. Thus, when moving forwards, the location is calculated by performing all rotations in order of most recent. To implement this method, each rotate movement is saved in a history which can be iterated through.

Diameter

Two branch diameter methods were implemented, one for the 2D representation of the L-System and one for the 3D Mesh representation. The method, as described in [2], decrements the diameter of branch segments along the branch.

The logic is performed separately from the *View* class which creates the 2D visual representation of the L-System. This means the *View* object does not know about the relationships between branch segments. Thus, a default value is set for the width. This means that the tree diameter does not respond to the change in iterations.

As described in Section 5.1, the 3D mesh creation process requires information about the branch segments in a tree data structure. Therefore, the L-System implementation must be updated to create such a structure rather than a flat list of branch segments. This structure can then be used to calculate the width of the root by saving the width changes for each branch segment and propagating this information back to the root. The width of a branching point can be calculated as:

$$w = max(0, ..., w_i) + \Delta w$$

where i is the number of children, w_i is the width of child i, and δw is the weight change for that branch segment.

5.3 Palubicki Tree

The Palubicki Tree had a different structure to the Lindenmayer System: it builds from a single point using external factors to decide on the angle and length of the growth.

Shadow Propagation

Shadow propagation is described as spreading through a grid of voxels. Starting from the single voxel wide layer containing the bud, each layer is expanded by a voxel in all directions (apart from the y axis) with the shadow value for each layer decreasing. I chose to implement this by creating a system to create a shadow pyramid under each bud and store the resulting changes in shadow value into the buds and attraction points themselves.

Rotation

As mentioned in Section 5.1 the branches are placed using a rotation calculation between the target and origin points. This method is reused to calculate the location of a new child given an optimal growth direction and a length. Using this calculation allows the length to be based on the resources allocated to the parent, see Section 3.2.3, rather than the length of the optimal growth vector which, as the sum of three vectors, can become quite long.

Seasons

A season is an iteration of growth. Before each season, a list of buds that can grow is collected, and it is this list that grows in that season. However, adding a child to a bud transforms it (see Section 4.3). While the buds are updated, the list of buds that were collected at the start of the iteration is not updated.

As each bud is kept and updated in a class level list, this list could be used to access the correct bud. However, when updating a bud, the previous version is removed and an updated version added to the end, changing the order of the list.

Thus, the solution that I implemented saves the location of the fertile buds as, while the class is updated, the location of a bud always stays the same. When a new location is taken from the list, the corresponding bud is found from the class level list which then grows.

Obstacles

An Obstacle can be described as an area of space where the tree is not allowed to grow. Thus, a method should return whether the location is within or outside the obstacle.

I implemented a Box obstacle which is used in Figure 6.7 and 6.9 by checking whether a point is between two corners of the box. For this method to work, the corners should be the front, right, top corner and the back left, bottom corner.

A Sphere obstacle was easier as any point within the radius distance from the centre point of the sphere is considered within the sphere. An example can be seen in Figure 6.8.

To prevent the Tree from growing through the obstacles, any attraction points within the obstacle should be removed, encouraging the point to grow away. However, there are times when this fails, for example, if the attraction points pass through the obstacle, resulting in a point within the obstacle. The solution I implemented was to stop any buds which attempt to grow into an obstacle from growing any further.

5.4 Godot

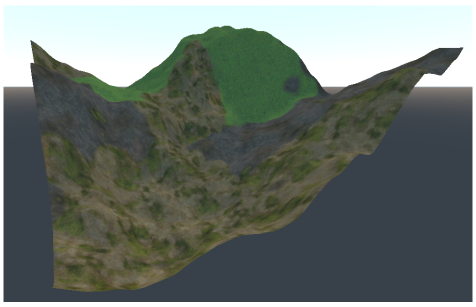


Figure 5.1: The Terrain from above.

At the beginning of the project, I endeavoured to create a mesh that could be incorporated into a 3D environment, specifically Godot. This game engine was chosen because it provides a shallow learning curve, supporting someone like me who has no graphical background, and is popular within the graphics community.

As preparation for this, I implemented a 3D terrain (see Figure 5.1), a first-person player to move around the environment, and a method for placing an object at a mouse click.

First Person Player

The basics of the Godot interface were learnt through a 5hr tutorial on YouTube [25] which lead the viewer through the steps to create a simple 3D game involving a rolling ball, enemies, and coin collection. The important things learnt from the tutorial was the ability to move a camera around a scene, attaching a script to an object, and kinematic bodies.

Each Node can have a single script attached. A script contains Godot code which includes the ability to call internal functions and alert other Nodes of an event. A Kinematic Body is an object which interacts with other objects following the laws of physics [26], like falling and rolling with gravity. Combining this knowledge can lead to a movable object. The Kinematic body allows it to react naturally to the ground and other obstacles, while the script can include code to move the object based on the user inputs:

```
func _physics_process(delta):
    if Input.is_action_pressed("ui_right"): velocity.x = SPEED
    elif Input.is_action_pressed("ui_left"): velocity.x = -SPEED
```

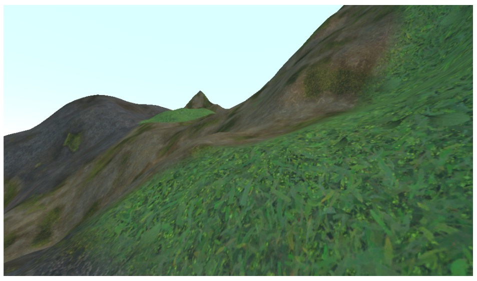


Figure 5.2: The Terrain from the First Person Perspective.

The function `_physics_process` is called every time the engine draws a frame. `Input` is an object which holds any user input made since the last frame. If the right key (`ui-right`) or left key (`ui-left`) is pressed, the x variable of the velocity vector is edited, moving right, or left.

Finally, without a camera, a 3D scene would appear as a black screen during run time. Connecting the camera to the Kinematic body will cause it to move with the body as the user moves the body.

Using this knowledge, I implemented gravity into the `_physics_process`, allowing the player to move across the hilly terrain realistically, rather than through.

GLTF File Format

When first converting the Lindenmayer System to a 3D mesh, I attempted to use the file format GLTF. This file format is a standard file format for 3D scenes and models and is the preferred format for importing models into Godot. When I started working on the Palubicki Tree, I discovered that the GLTF file format was causing too many problems. As displaying the final result in a 3D environment wasn't required for the project, I chose to export to the OBJ file format as explained in Section 5.1.

Godot is unable to import OBJ files. Thus, Godot isn't a part of the final product but was a crucial part of developing my understanding of computer graphics.

Chapter 6

Evaluation

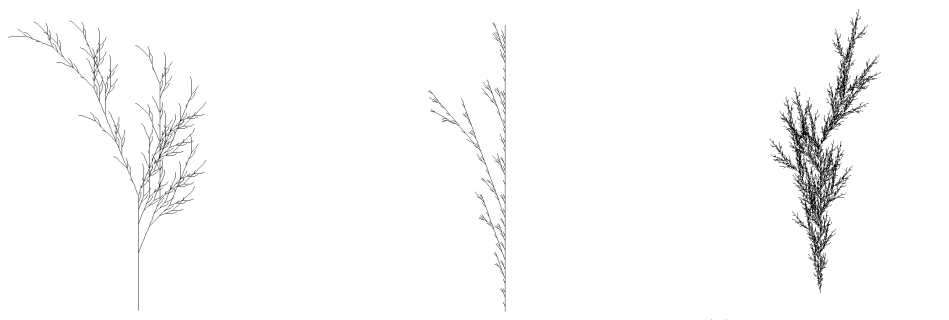
The inputs and OBJ files of the Figures shown in this chapter can be found in the appendix [27].

6.1 Lindenmayer System

2D L-Systems

The paper I followed on the Lindenmayer System [2] introduced a variety of variations on the system. The examples displayed in Figure 6.1 were created using my implementation of the Lindenmayer System. As the paper provided the rules used to create each tree, I was able to compare my results to their results.

In this report	In the paper [2]
Figure 6.1a	Figure 1.24f
Figure 6.1b	Figure 1.27
Figure 6.1c	Figure 1.31a



(a) Normal L-system (b) Stochastic L-system (c) Context Sensitive L-system

Figure 6.1: Examples of 2D Lindenmayer Systems.

The result for the Normal Lindenmayer System is identical to the image in the paper, confirming that my implementation was correct. The Stochastic Lindenmayer System is based on probability, meaning that no two results are identical. However, the structure of the tree model produced by my implementation is very similar to the various structures displayed in the paper. My Context-Sensitive Lindenmayer implementation produced slightly different results, indicating that there were some differences in the implementation.

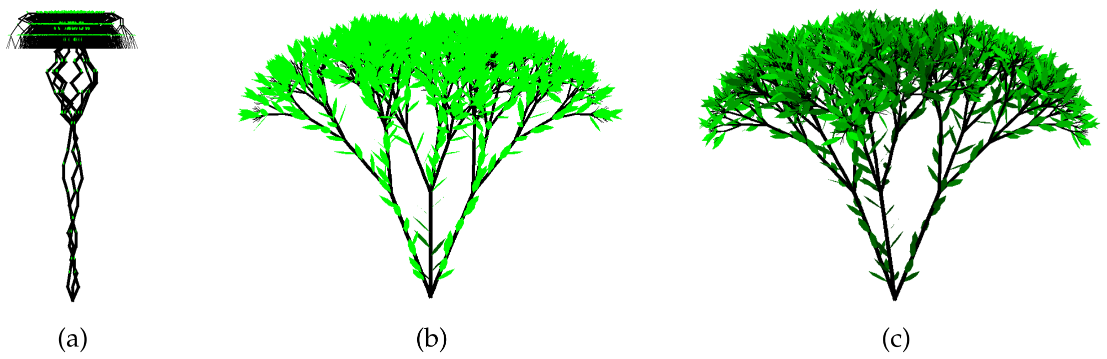


Figure 6.2: Examples of 3D Lindenmayer Systems.

3D L-Systems

The examples shown in Figure 6.2 are displayed as 2D representations of the tree. I chose to use this method while implementing the 3D L-System logic as it was quicker to test the output than building a mesh, it could use a similar implementation as the 2D L-System, and the results in [2] were presented in a similar method.

As mentioned in Section 5.2, the 3D system had some difficulties in the implementation, specifically around calculating the position of the next growth. Figure 6.2a was my first attempt at recreating the 3D Lindenmayer System shown in Figure 1.25 of [2]. The problem was the implementation of rotation which was based on the simpler 2D system. Each new rotation would just add to the angle, which was used to calculate the resulting position. As can be seen in Figure 6.2a, this approach was wrong.

Figure 6.2b is the improved version where, when the turtle moved forwards, its current angle would be stored into history and used in later movements. This produces a much more realistic tree.

Figure 6.2c is the final version which saves every angle change to history. It also incrementally edits the leaf colour by increasing the green parameter of the colour as the branch grows. Changing how the angles are stored exposes the branches but results in more accurate results in the later branches (further from the root). Finally, incrementing the leaf colour greatly increases the realism of the resulting tree, as it replicates the effect of shadow and prevents the leaves from looking like a blob of green. This final version, when compared to Figure 1.25 in [2] appear structurally identical.

Mesh L-Systems

The 3D Mesh of the L-System will be compared to the result of the Palubicki as the Lindenmayer paper only implements the 2D representation.

6.2 Palubicki Tree

While the paper I followed to create the Palubicki Tree [3] presented examples of its results, it didn't provide all of the inputs required to create such a tree model. Thus, I will instead consider the changes created by changing a parameter or resource allocation method.

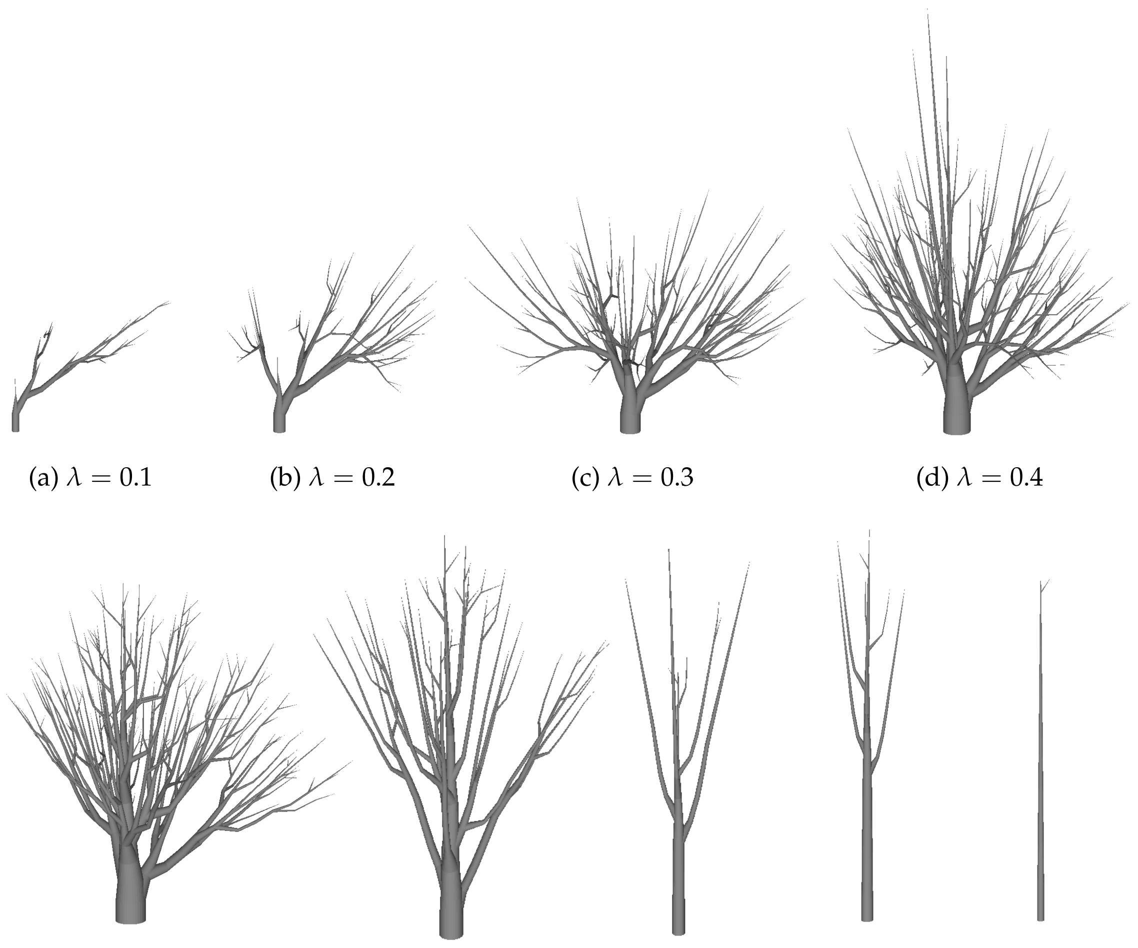
Extended Borchert-Honda Model

One of the methods for resource allocation, the extended Borchert-Honda model has one parameter, λ, that controls whether the resource allocation is biased to the main axis, biased towards the lateral branches, or not resource (see Section 3.2.3).

Having a bias to the lateral branches results in a sprawling tree with very little upwards growth. Increasing the bias towards the main axis, the tree form tends towards a tighter crown with most of the growth-focused on growing upwards. This can be recognised in Figure 6.3, which presents a progression of tree forms created by the extended Borchert-Honda model.

Setting the parameter λ to a value under 0.5 caused the model to be biased towards the lateral branches. Figure 6.3a has the lowest λ value and is a widespread tree whose growth is focused on the lateral branches. In the following Figures, where the λ value is increasing, more branches are growing upwards, creating a more upright tree. Finally, Figure 6.3i has the highest λ value and is concentrated on growing the main axis.

In the paper, [3], Figure 7 presents a similar progression. As the paper used a different set of values for the parameter λ and it has yet to implement the shedding of branches, our results are not identical. However, it can be observed that like my results, the tree form goes from a wide short form to a tall thin form.



(a) $\lambda = 0.1$ (b) $\lambda = 0.2$ (c) $\lambda = 0.3$ (d) $\lambda = 0.4$

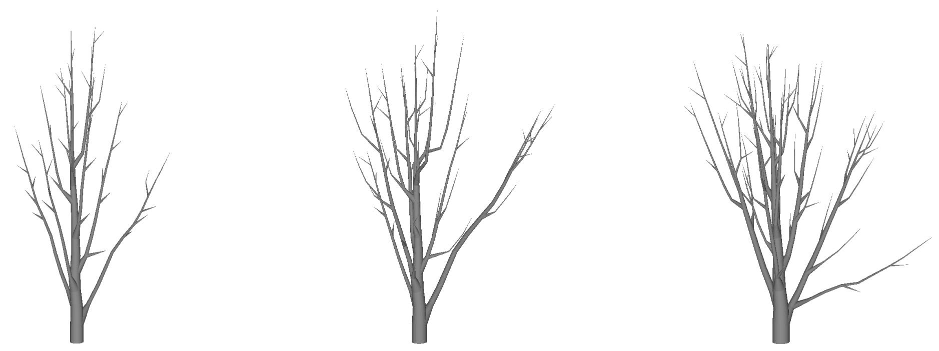
(e) $\lambda = 0.5$ (f) $\lambda = 0.6$ (g) $\lambda = 0.7$ (h) $\lambda = 0.8$ (i) $\lambda = 0.9$

Figure 6.3: Sample Trees built using the extended Borchert-Honda model.

Priority Model

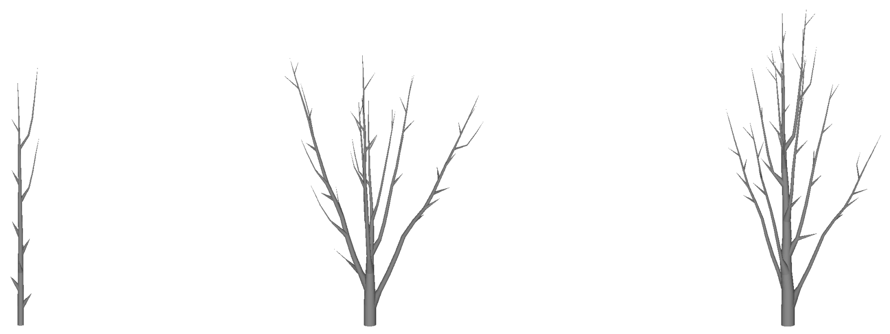
The Priority model is another resource allocation model described in the paper [3]. It uses a weight system to spread the resources to the buds on a branch to spread the resources to the most productive buds.

The weight system, shown in Figure 3.9, is described by three parameters: K, w_{max}, and w_{min}. To confirm the behaviour of my implementation of the priority model, I created tree models using a variety of values for these parameters.



(a) $K = 0.25$ (b) $K = 0.5$ (c) $K = 0.75$

Figure 6.4: Sample Trees built with the Priority Model.



(a) $w_{min} = 0.1$ (b) $w_{min} = 0.3$ (c) $w_{min} = 0.5$

Figure 6.5: Sample Trees built with the Priority Model.

The parameter K designates the percentage of buds on a branch that will be weighted, with the rest receiving a default minimum value ($w_m in$). Assigning weights to the more productive branches (at the beginning of the priority list) results in a more excurrent tree, where the main axis is kept as a central leader rather than separating into smaller trunks. Figure 6.4 shows the effect of changing the K value. It can be observed that Figure 6.4a keeps the main axis to the top, with the lateral branches keeping thin. Increasing the K value does increase the size of the lateral branches, meaning fewer resources are being kept in the main axis. Finally, Figure 6.4c has a K value of 0.75. Near the top, the main branch is separating into multiple branches.

If a bud is not weighted, it is assigned the parameter w_{min}. Thus, having a smaller w_{min} reduces the resources being allocated to the buds lower in the priority list. As the ability to grow is based on the resource income, if the number of resources coming into a bud is too small, it won't be able to grow. This effect is shown in Figure 6.5. Figure 6.5a has a very low w_{min} meaning the resources allocated to lateral branches is not enough for them to grow properly. Interestingly, using the value of 0.3 for the w_{min} in Figure 6.5b resulted in more growth in a select few lateral branches along with the main axis. Finally, Figure 6.5c has reached a balance where the main axis has the most resources but lateral branches have enough resources to grow.

In the paper, [3], Figure 9 shows two trees with different K values. The tree which was produces with a larger K value (left) has wider spread branches and the main axis splits early on while the tree with the smaller K value (right) maintains the main axis for much longer. These results match the tree forms created by my implementation.

Shedding Branches

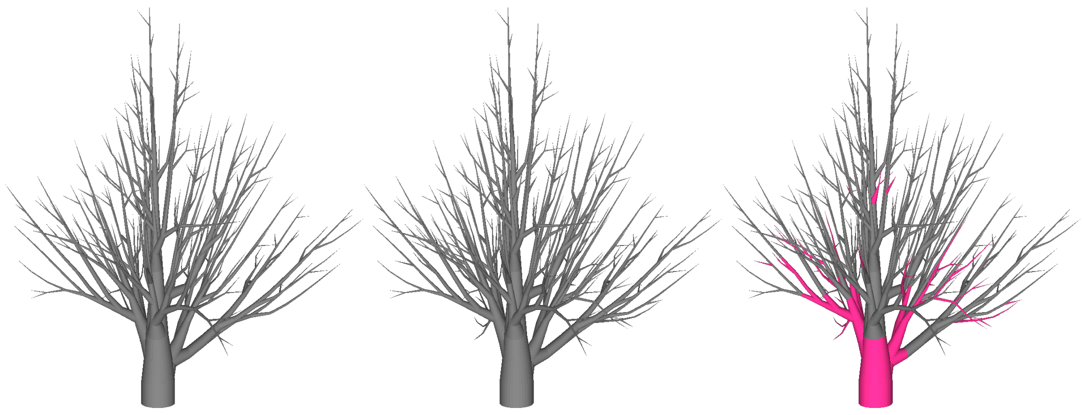

(a) Shedded Liable Branches (b) Kept all Branches (c) Comparison

Figure 6.6: A Tree Model grown around an obstacle.

Shedding branches is simulated by calculating the worth of a branch and shedding branches that are considered a liability to the tree.

Figure 6.6a shows a tree model that implements shedding branches. Figure 6.6b shows a tree model that keeps all its branches. As most shedded branches are fairly small, Figure 6.6c shows a comparison between the two trees, with the pink parts present in Figure 6.6b but not in Figure 6.6a. The branches that are shedded in Figure 6.6a keep growing in Figure 6.6b, increasing the diameter along the branch to the trunk, explaining why the trunk is pink.

Reacting to Environment

The Palubicki Tree is designed to react correctly to changes in the environment. To test I implemented a method to add an obstacle to the environment to which the growing tree should react.

The three examples in Figures 6.7, 6.8, and 6.9 are built using the same inputs but are influenced by different obstacles in their environment. The obstacle types include a wall beside the tree (Figure 6.7), a sphere hovering above the tree (Figure 6.8), and two walls on either side of the tree (Figure 6.9). These tree models confirm that my implementation correctly avoids growing through an obstacle but can grow tight around it, as seen in Figure 6.8.

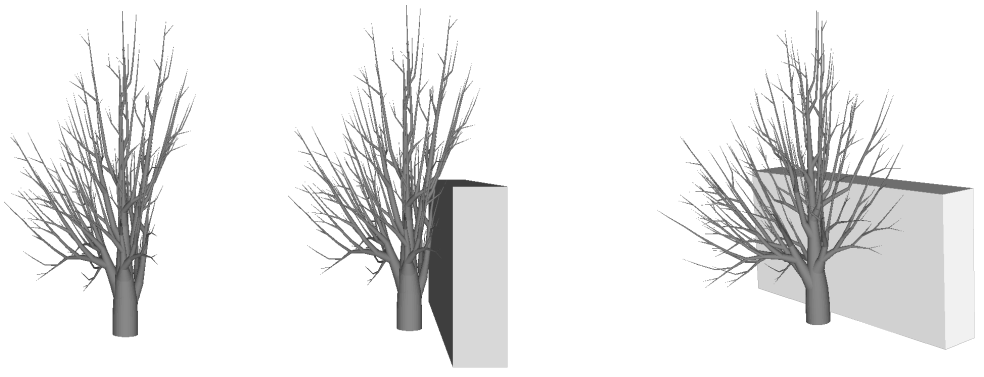

(a) (b) (c)

Figure 6.7: A Tree Model grown around an wall.

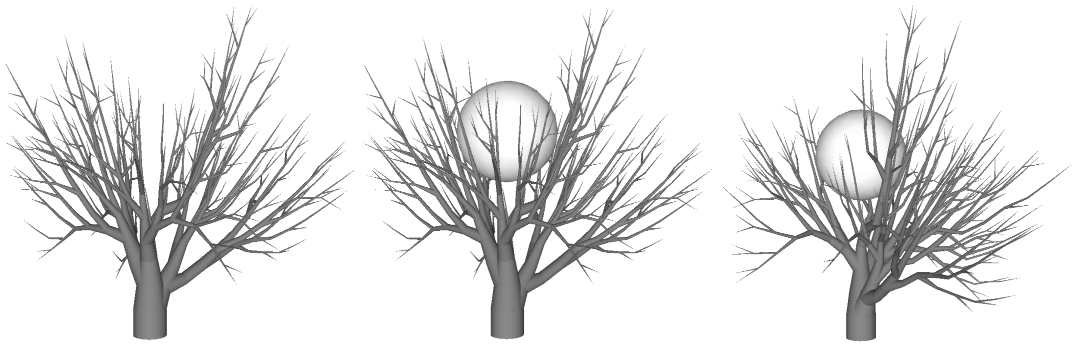

(a) (b) (c)

Figure 6.8: A Tree Model grown around an ball.

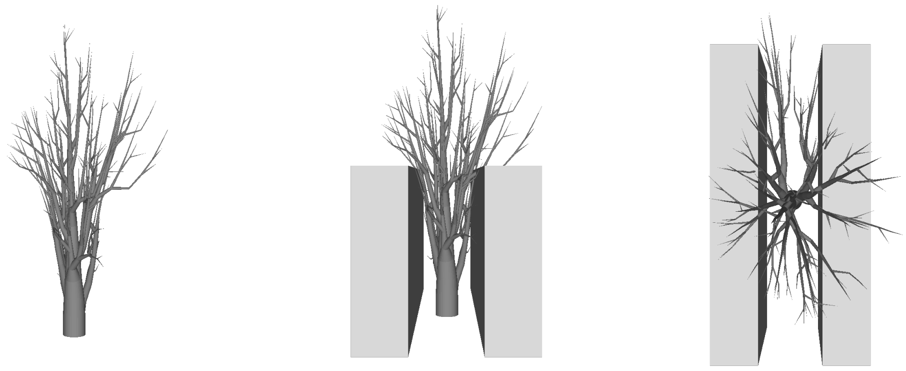

(a) (b) (c)

Figure 6.9: A Tree Model grown between two walls.

6.3 Comparing Tree Models

During this project, I implemented two types of tree modelling methods, one that uses a repetitive, recursive process to build a branching structure, and one that incorporates environmental conditions to calculate the best position for the next growth. Thus, this section of the evaluation will compare the results of the two modelling methods: the Lindenmayer System and the Palubicki Tree.

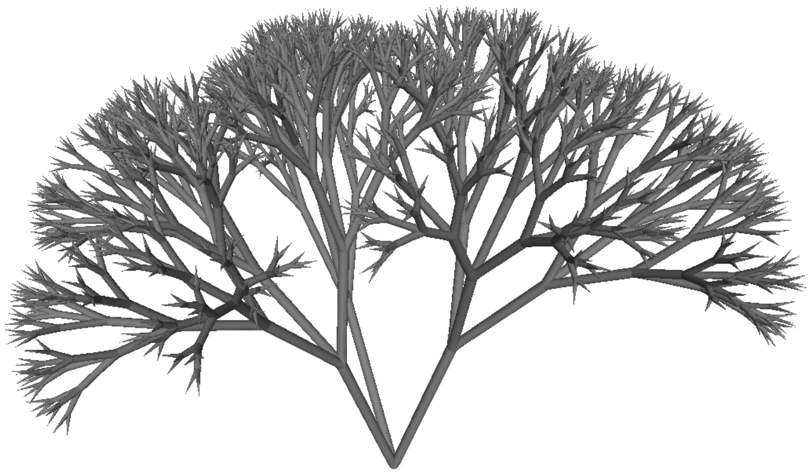
Figure 6.10a shows a rendered version of the 3D mesh created by the Lindenmayer System as shown in Figure 6.2c though without the leaves (to see the distribution of branches). Figure 6.10b displays a rendered version of the 3D mesh created by the Palubicki Tree. Both tree structures were converted to 3D meshes using the same process.

The challenge for this project was to implement a paper that could produce a realistic tree.

The Lindenmayer System, while creating an impressive branching structure, fails to meet the goal in a few aspects. It tends to produce repetitive structures which appear too perfect for the real world. But, more importantly, it is unable to react to obstacles or changes in the environment. This is because the pattern is built without considering any external factors.

The Palubicki Tree focuses on implementing the self-organising character of tree development which produces a more realistic tree. The branches aren't perfectly straight, they tend to grow towards the light or away from other branches forming a more crooked structure. Some parameters control certain aspects of the tree, through which modifications can be made to change the tree form.

The largest difference is the highly controlled order of the Lindenmayer System vs the more random structure of the Palubicki Tree.


(a) Lindenmayer System.

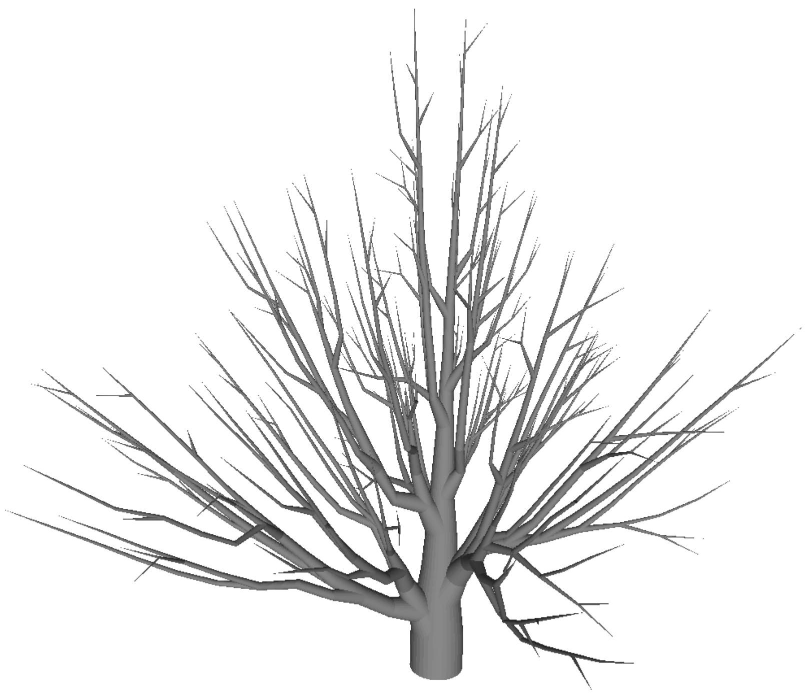

(b) Palubicki Tree.

Figure 6.10

Chapter 7

Conclusion

Over the last 4 years at Victoria University of Wellington, I acquired many skills based in the area of Software Engineering. To challenge myself, I chose to implement a Computer Graphics paper as I had little experience in this area. Thus, I could demonstrate the skills I've already learnt while expanding my field of knowledge.

During this project, I implemented two papers. The Lindenmayer System is described in [2] to produce a repetitive and recursive branching structure and has become a foundation for tree modelling while the Palubicki Tree described in [3] aimed to create a method for generating self-organising tree models which incorporate environmental conditions. This was done to familiarise myself with the background of tree modelling and so the results could be compared.

In Chapter 6 I was able to demonstrate that my implementation of the Palubicki Tree responded correctly to changes in parameters including the addition of various types of obstacles. The Lindenmayer System implementations produced similar results to the source paper.

The conclusion of the evaluation was the 3D mesh created by the Palubicki Tree implementation appeared visually closer to a real tree than the model created by the Lindenmayer System as it produced varying branch shapes unlike the repetitive structure of the Lindenmayer System.

7.1 Future Work

Currently, the code produced contains a large number of **static final** fields and methods which are used in multiple places in the code. These include methods for calculating resources, fields for the perception volume and occupancy zone, and fields for the resource allocation models. This prevents the program from editing the parameters to change the structure of the tree. Thus, some future work should be invested into refactoring this code so that some fields can be editable while keeping them available to the classes and methods that need access.

As mentioned in Section 5.4, I attempted to create a mesh that can be incorporated into a 3D environment. However, exporting the mesh to a file format that Godot could read proved too much work for the result. Thus, future work could include developing a method to convert the 3D mesh created inside the program into a file format that Godot can import.

Finally, a challenge for the future is to implement the texture of the bark on the branches. This could be either implemented after the tree is created, so working with the 3D mesh, or during the tree construction where the branches are considered separate objects. In [28], they explore the cracking of the tree bark from the stress of the branch growing. The result of this would be a more realistic tree model.

Bibliography

[1] A. Lindenmayer, “Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs,” *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 300–315, 1968. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022519368900805

[2] A. Lindenmayer and P. Prusinkiewicz, *The Algorithmic Beauty of Plants*. Springer New York, Mar. 1990.

[3] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, and P. Prusinkiewicz, “Self-organizing tree models for image synthesis,” *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 1–10, Jul. 2009.

[4] S. Pirk, O. Stava, J. Kratt, M. A. M. Said, B. Neubert, R. Měch, B. Benes, and O. Deussen, “Plastic trees: Interactive self-adapting botanical tree models,” *ACM Transactions on Graphics*, vol. 31, no. 4, pp. 1–10, Aug. 2012.

[5] S. Pirk, T. Niese, O. Deussen, and B. Neubert, “Capturing and animating the morphogenesis of polygonal tree models,” *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 1–10, Nov. 2012.

[6] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, and B. Benes, “Inverse procedural modelling of trees,” *Computer Graphics Forum*, vol. 33, no. 6, pp. 118–131, Mar. 2014.

[7] T. Sachs and A. Novoplansky, “Tree form: architectural models do not suffice,” *Israel Journal of Plant Sciences*, vol. 43, no. 3, pp. 203–212, 1995.

[8] P. Prusinkiewicz, “Graphical applications of l-systems,” in *Proceedings of graphics interface*, vol. 86, no. 86, 1986, pp. 247–253.

[9] H. Abelson, *Turtle geometry : the computer as a medium for exploring mathematics*. Cambridge, Mass: MIT Press, 1981.

[10] S. Ulam *et al.*, “On some mathematical problems connected with patterns of growth of figures,” in *Proceedings of Symposia in Applied Mathematics*, vol. 14. Am. Math. Soc. Vol. 14, Providence, 1962, pp. 215–224.

[11] J. Arvo, D. Kirk *et al.*, “Modeling plants with environment-sensitive automata,” in *In Proceedings of Ausgraph’88*. Citeseer, 1988.

[12] N. Greene, “Voxel space automata: Modeling with stochastic growth processes in voxel space,” in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, 1989, pp. 175–184.

[13] B. Benes and E. U. Millán, "Virtual climbing plants competing for space," in *Proceedings of Computer Animation 2002 (CA 2002)*. IEEE, 2002, pp. 33–42.

[14] W. Palubicki, *Fuzzy Plant Modeling with OpenGL: Novel Approaches in Simulating Phototropism and Environmental Conditions*. Saarbrücken: VDM Verlag Dr. Müller, Oct. 2007.

[15] S. Bornhofen and C. Lattaud, "Competition and evolution in virtual plant communities: a new modeling approach," *Natural Computing*, vol. 8, no. 2, pp. 349–385, 2009.

[16] D. Cohen, "Computer simulation of biological pattern generation processes," *Nature*, vol. 216, no. 5112, pp. 246–248, 1967.

[17] Y. Rodkaew, P. Chongstitvatana, S. Siripant, and C. Lursinsap, "Particle systems for plant modeling," *Plant growth modeling and applications. Proceedings of PMA03, Hu B.-G., Jaeger M.,(Eds.). Tsinghua University Press and Springer, Beijing*, pp. 210–217, 2003.

[18] A. Runions, B. Lane, and P. Prusinkiewicz, "Modeling trees with a space colonization algorithm." *NPH*, vol. 7, pp. 63–70, 2007.

[19] A. Runions, M. Fuhrer, B. Lane, P. Federl, A.-G. Rolland-Lagan, and P. Prusinkiewicz, "Modeling and visualization of leaf venation patterns," in *ACM SIGGRAPH 2005 Papers*, 2005, pp. 702–711.

[20] R. Borchert and H. Honda, "Control of development in the bifurcating branch system of tabebuia rosea: a computer simulation," *Botanical Gazette*, vol. 145, no. 2, pp. 184–195, 1984.

[21] A. Takenaka, "A simulation model of tree architecture development based on growth response to local light environment," *Journal of Plant Research*, vol. 107, no. 3, pp. 321–330, 1994.

[22] R. Měch and P. Prusinkiewicz, "Visual models of plants interacting with their environment," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 397–410.

[23] K. Shinozaki, K. Yoda, K. Hozumi, and T. Kira, "A quantitative analysis of plant form-the pipe model theory: I. basic analyses," *Japanese Journal of ecology*, vol. 14, no. 3, pp. 97–105, 1964.

[24] N. MacDonald, "Trees and networks in biological models," 1983.

[25] BornCG, "Godot 3.1: Creating a simple 3d game," YouTube, Mar. 2021. [Online]. Available: https://www.youtube.com/playlist?list=PLda3VoSoc_TSBBOBYwcmlamF1UrjVtccZ

[26] J. Linietsky, A. Manzur, and the Godot Community, "Kinematicbody," *Godot Docs*, 2021. [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_kinematicbody.html

[27] B. Gatehouse, "Appendix," 2021.

[28] J. Kratt, M. Spicker, A. Guayaquil, M. Fiser, S. Pirk, O. Deussen, J. C. Hart, and B. Benes, "Woodification: User-controlled cambial growth modeling," *Computer Graphics Forum*, vol. 34, no. 2, pp. 361–372, May 2015.