

Final Report

GENERATING AND VISUALISING REALISTIC PLANTS



CM3203 One Semester Individual Project

Cardiff School of Computer Science and Informatics

Author: Yao Xiao

Supervisor: Professor Yukun Lai

Moderator: Dr Sylwia Polberg

Abstract

Plants come with complicated structures and abundant details . 3D virtual plant requires combination of plant morphology , computer science , and rendering techniques.

This report covers an implementation of plants modelling algorithm known as L-system. The project aims to create an application that visualize 3D realistic plants. It requires to abstract the morphological structure and to analyze the expanding and branching progress of real plants, as well as a rendering process that draws the plants on the screen. Furthermore, to generate plants with random structure which makes it more natural. The report will also discuss the success and major problems of the process of implementation, as well as failures of results.

Results will be analyzed and discussed by adjusting parameters that change the shape of 3D virtual plants. Ideas to expand the solution will be put forward, as well as the reflection on the learning during implementation, including the positives and negatives.

Acknowledgements

I would like to thank my supervisor, Professor Yukun Lai, for his help and invaluable suggestions and criticism at every stage of this project.

Table of Contents

Generating and visualising realistic plants	0
Abstract	1
Acknowledgements	2
Table of Contents	3
Table of figures	5
Table of code fields.....	7
1.Introduction	7
2.Background.....	8
2.1 Plants classification by stem type and branching description	8
2.2 A plant branching pattern: Sympodial branching	10
2.2.1 Related concepts of sympodial branching	10
2.3 Plants modelling based on L-system	12
2.3.1 Principle of L-system.....	12
2.3.2 Explanation by L- system turtle in 2D plane	12
2.3.2 Explanation by L- system turtle in 3D space	14
2.3.3 Parametric L-system	16
2.4 Brief introduction of Unreal Engine 4	16
2.4.1 Map.....	16
2.4.2 Actor class	17
2.4.2 Actor's component	18
2.4.3 Textures	19
2.4.4 Game modes	19
3.Specification and Design.....	19
3.1 Parametric L-system construction of sympodial tree component	19
3.1.1 Construction of sympodial tree	19
3.2 L-system turtle component	21
3.2.1 Basic information to describe the status of L-system turtle	21
3.2.2 Actions of L-system turtle	22
3.3 Mesh component.....	23
3.3.1 Tree trunk	23
3.3.2 Branch	24
3.3.3 Leaf.....	24
3.4 Working mechanism of the system.....	24
4.Implementation.....	25
4.1 UML Class diagram	26

4.2 Tree component	26
4.2.1 Generate random numbers	26
4.2.2 Generate sentence.....	27
4.3 L-system turtle component	28
4.3.1 Overall working process: Iteration.....	28
4.3.2 Rotation of L-system turtle	32
4.3.3 Forward.....	33
4.3.4 Push and pop	36
4.3.5 Delete tree	36
4.4 Mesh component.....	37
4.4.1 Scale of the mesh.....	37
4.5 User interface.....	39
4.5.1 Overall design of user interface	39
4.5.2 Implementation of user interface	39
4.6 Game mode.....	41
4.7 Important problems during implementation	42
4.7.1 Rough and bumpy surface of mesh component	42
4.7.2 Performance improvement.....	43
5.Results and Evaluation.....	45
5.1 User interface.....	45
5.1 Effect of each parameter.....	46
5.2 Different types of trees.....	49
5.3 Observation from different perspective.....	53
6. Future work.....	54
7. Conclusions.....	54
8. Reflection on Learning	55
9. References.....	56

Table of figures

Figure 1 The picture of a woody plant	9
Figure 2 An axial tree. (Prusinkiewicz P and Lindenmayer A, 1990)	9
Figure 3 Sympodial branching.....	10
Figure 4 Branching order	10
Figure 5 Branching angle.....	11
Figure 6 Numbers of branching	11
Figure 7 Interpretation of the control string. (Prusinkiewicz P and Lindenmayer A, 1990)	13
Figure 8 Demonstration of L-system turtle.....	14
Figure 9 Controlling the turtle in 3D space. (Prusinkiewicz P and Lindenmayer A, 1990).....	15
Figure 10 A map in Unreal Engine 4.....	17
Figure 11 Location and rotation.....	17
Figure 12 Location and rotation of an object	17
Figure 13 The world coordinate(left) and local coordinate system(right).....	18
Figure 14 A static mesh actor.....	18
Figure 15 Branches with different materials	19
Figure 16 Demonstration of direction vectors.....	22
Figure 17 Demonstration of a trunk and its information.....	22
Figure 18 Working process	23
Figure 19 Tree trunk component	24
Figure 20 Branch component.....	24
Figure 21 Leaf component	24
Figure 22 Working mechanism	25
Figure 23 2-generation tree	29
Figure 24 Demonstration of factors distribution	29
Figure 25 The actual render process.....	30
Figure 26 Demonstration of Moving.....	33
Figure 27 Demonstration of static mesh actors.....	34
Figure 28 Confirming location (Ideal design)	34
Figure 29 Actual confirmation of next location	35
Figure 30 A branch, trunk, and leaf (Original).....	38

Figure 31 Design of user interface	39
Figure 32 A widget blueprint class	40
Figure 33 Previous mesh of branch	42
Figure 34 Tree trunk.....	42
Figure 35 Branch	43
Figure 36 Cropping process on 3DS MAX 2019.....	43
Figure 37 Frame rate testing before improving.....	44
Figure 38 Default lighting setting in a map.....	44
Figure 39 Frame rate test after improving.....	45
Figure 40 User interface.....	45
Figure 41 Same type of tree with different generation (7,9).....	46
Figure 42 Same type of tree with different triple branching probability (0.4,0.6)	47
Figure 43 Same type of tree with different initial length (300,400).....	47
Figure 44 Same type of tree with different initial width (70,80)	48
Figure 45 Same type of tree with different rotation angles around z-axis (-45,45,105) (-30,30,90)	48
Figure 46 Same type of tree with different rotation angles around x-axis (-30,35, -40) (-20,30, -35)	49
Figure 47 Sympodial tree _1	49
Figure 48 Sympodial tree _2	50
Figure 49 Sympodial tree _3	51
Figure 50 Sympodial tree _4	51
Figure 51 Sympodial tree _5	52
Figure 52 Observe the tree from the right.....	53
Figure 53 Observe the tree from the bottom	53
Figure 54 Observe the tree from the top.....	53

Table of code fields

Code 1 Generate probability for different branching(SympodialTree_3.cpp)	27
Code 2 Generate random numbers sequence(SympodialTree.cpp)	27
Code 3 Rewrite character(SympodialTree.cpp)	28
Code 4 Double-branching (LsystemTurtleActor.cpp).....	31
Code 5 Angle distribution (LsystemTurtleActor.cpp).....	32
Code 6 Angle declaration of sympodial tree (SympodialTree_3.cpp)	32
Code 7 Rotation (LsystemTurtleActor.cpp).....	33
Code 8 Spawning static mesh actors (LsystemTurtleActor.cpp).....	35
Code 9 Store references (LsystemTurtleActor.cpp).....	37
Code 10 Iterate arrays (LsystemTurtleActor.cpp).....	37
Code 11 First scaling adjustment of branch (StaticTreeComponent.cpp)	38
Code 12 First scaling adjustment of trunk (TreeTrunkComponent.cpp)	38
Code 13 Method to adjust scale (StaticTreeComponent.cpp)	38
Code 14 Method to adjust scale (TreeTrunkComponent.cpp)	39
Code 15 An Example of connecting component (MyUserWidget.cpp)	40
Code 16 Declaration of widget object (IPGameModeBase.cpp)	40
Code 17 Example of data transmit and generating a tree (MyUserWidget.cpp)	41
Code 18 Create and assign L-system turtle object (IPGameModeBase.cpp)	41
Code 19 BeginPlay() in Game mode (IPGameModeBase.cpp).....	41

1.Introduction

With the huge development of 3D technology, the topological structure and geometrical morphology of plants has become a key researching point within related field. By using computer to generate realistic plants, time expenses can be saved from constructing 3D plants model manually. In terms of realness of plants, computer program can generate random shape of plants which is an advantage compared to manual construction.

The project aims to generate and visualize realistic virtual plants on computer. However, we will need to solve 2 aspects of problems. First, to learn and describe the structure of plants and made it into a digital term. Second, render the plants on computer based on the digital structure of plants.

Topological structure is a significant symbol for plants, which contains the structure and morphology of plant organs. There are several methods to simulate structure of plants, in this project, we will focus on L-system. L-system is a parallel rewriting system producing control string. Control string starts with simple symbol string called “axiom” string and by expanding each symbol into larger string of symbols, a complex control string can be used to describe the structure of plants. In this project, L-system will be implemented by C++.

To render the digital structure of plants, a 3D graphics tool called Unreal Engine 4 will be used in this project. Unreal Engine 4 is a game engine developed by Epic Games. It has been used in variety of genres of 3D games and has seen adoption by other industries like film and television industry. In comparison with other render method like using graphics API like OpenGL, Unreal Engine 4 can be efficient in generating more realistic result, in terms of lighting, model importing and texture attaching.

This report will cover the implementation of the L-system algorithm from beginning to end. It starts with introducing the mechanism of L-system as well as the Unreal Engine 4. Following with the implementation and result of L-system. Finally, the report will discuss the failure of the projects compared to the initial planning and the potential future work.

2. Background

2.1 Plants classification by stem type and branching description

Plant is classified as herbaceous plant and woody plant by stem type. Herbaceous plants are plants that, by definition, have non-woody stems. A woody plant is a plant that produces wood as its structural tissue and thus has a hard stem. Woody plants are usually either trees, shrubs, or lianas (Wikipedia, n.d.)[1]. Considering the time limit of this project, we will only research and discuss tree, a type of woody plant, in this project.



Figure 1 The picture of a woody plant

There are three forms of branching. The first one is between a trunk and branches, the second one is between branches and branches, and the last one is between branches and leaves. Considering the features of L-system, in this project, we will only focus on the ones which relate to a trunk and branches. In nature, leaves have more complex branching strategy that they tend to be soft and drooping. Therefore, we will not focus on branching form between branches and leaves.

The way we use to describe the branching is called “axial tree”[2]. A demonstration of an axial tree and its definition are as follows:

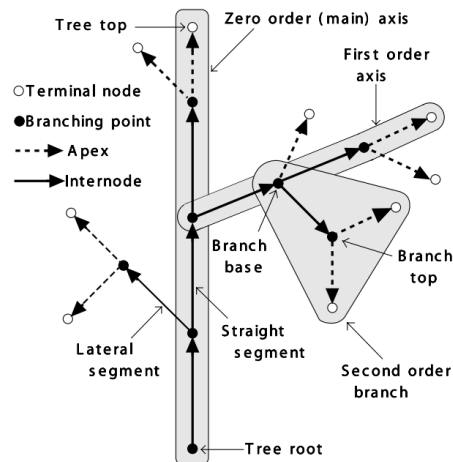


Figure 2 An axial tree. (Prusinkiewicz P and Lindenmayer A, 1990)

As shown in the figure, the trunk starts at tree root is called zero order axis, or main axis. Grows from the trunk, the next axis is called first order axis, and so on. The shape of each axis is the same, but child axis is smaller. Overall, the shape of each axis is basically same as the shape of the whole tree, which is an important feature of L-system.

2.2 A plant branching pattern: Sympodial branching

Sympodial branching is one of the plant branching patterns. It means that after a period time of growing, the main branch will stop growing, or even die and forming lateral branch to grow. This pattern will also be applied to the new branches repeatedly. In this project, sympodial branching will be implemented.

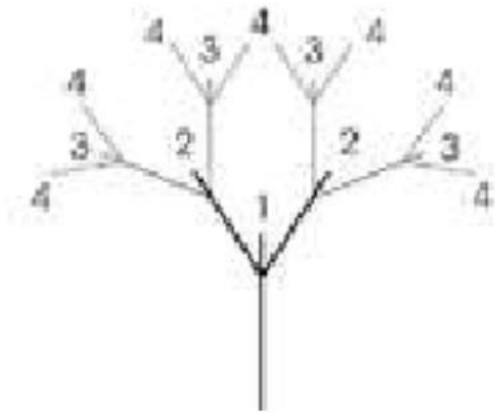


Figure 3 Sympodial branching

2.2.1 Related concepts of sympodial branching

Three concepts that make influences are given to explain sympodial branching.

(1) Branching order

The axis originating at the root of the entire plant has order zero. A branch originating as a lateral segment of an n -order parent branch has order $n+1$ [2]. As shown in the figure 6, by its definition, the tree has at most 2 orders of branching.

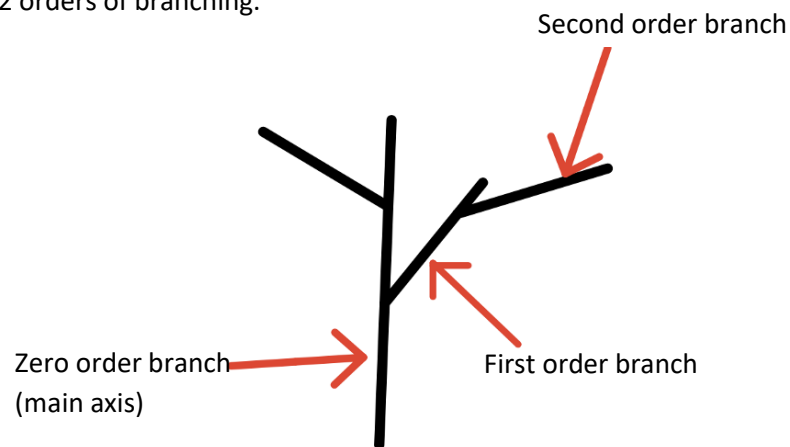


Figure 4 Branching order

(2) Branching angle

Branching angle is another important factor of trees' branching characteristic. Branching angle is the angle between a branch and its parent branch. It makes a significant influence on the overall structure of a tree. Multiple branching angles will also appear on one type of tree.

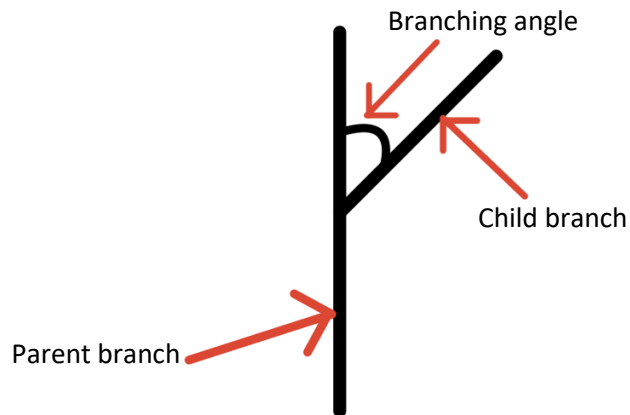


Figure 5 Branching angle

(3) Numbers of branching

Numbers of branching could be different on different order of branch. For example, there could be 2 child branches on the zero-order branch, 4 child branches on the first order branch and 6 child branches on the second order branch. In nature, the numbers of child branching are random. However, in this project, the numbers of branching have been manually set as 2 or 3, which means that there could be 2 child branches or 3 child branches on a parent branch.

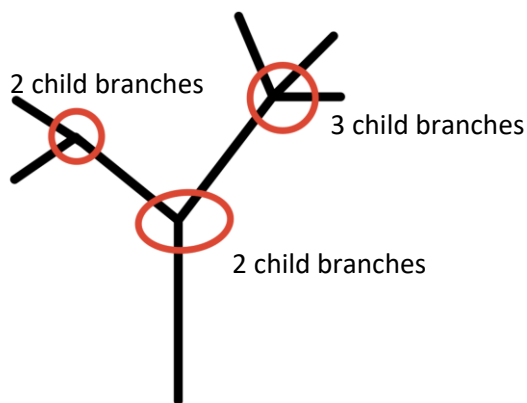


Figure 6 Numbers of branching

2.3 Plants modelling based on L-system

2.3.1 Principle of L-system

L-System is a unique type of iteration process. Its core concept is rewriting. L-system uses character from alphabet or strings that formed by character to generate the initial form of control string, which is Axiom. After that, it rewrites the string by replacing every character in Axiom with rewriting rules repeatedly. Finally, the result is the ultimate control string that is needed.

L-system are formalized as a tuple as followed:

$$G = \langle V, \omega, P \rangle$$

Within the tuple, V is the alphabet of system, or all potential symbols in the string. ω is the initial string of system, which is the Axiom. P is a finite set of rewriting rules.

For example, a specific L-system can be as follows:

$$V = \{a, b\}$$

$$\omega = ab$$

$$P = \{a \rightarrow b, b \rightarrow ab\}$$

There are 2 rewriting rules in P , if we replace a with b , replace b with ab , then the iteration process can be described as:

$$ab \rightarrow bab \rightarrow abbab \rightarrow bababbab \rightarrow abbabbababbab \rightarrow \dots\dots$$

When we assign actual meaning to each character and make a geometric explanation to the ultimate control string, we can get the corresponding images.

2.3.2 Explanation by L- system turtle in 2D plane

L-system turtle is one of the geometric interpretation ways of the string that L-system generates. L-system turtle is put forward by Prusinkiewicz[3] and Hana[4].

(1) Traditional L-system turtle

The state of L-system turtle can be defined by a triad (x, y, σ) . Within the triad, (x, y) represents the Cartesian coordinates of turtle, σ represents the heading direction of turtle. Now we define a L-system as following:

Alphabet $V = \{F, +, -\}$

Axiom $\omega = F$

Set of rewriting rules $P = \{F \rightarrow FFF - FF - F - F + F + FF - F - FFF\}$

Given the length of each move s and the angle increment is δ . The actions that turtle will perform for each character in alphabet are as following:

F : The turtle moves forward a step of length s . The new coordinate of turtle is (x', y') . Suppose the direction vector of turtle's forward is $(\cos\theta, \sin\theta)$, then $x' = x + s \cos\theta$, $y' = y + s \sin\theta$. The turtle draws a line between (x, y) and (x', y') .

$+$: Turn left by 90 degree, the next status of turtle is $(x, y, \sigma + 90)$.

$-$: Turn right by 90 degree, the next status of turtle is $(x, y, \sigma - 90)$.

In this case, if the iteration time is 1, the control string that is generated by L-system is $FFF - FF - F - F + F + FF - F - FFF$. Set initial angle $\sigma = 90^\circ$ and the movement unit s equals to the length of grid. Then we can get an explanation graph of this control string as follow:

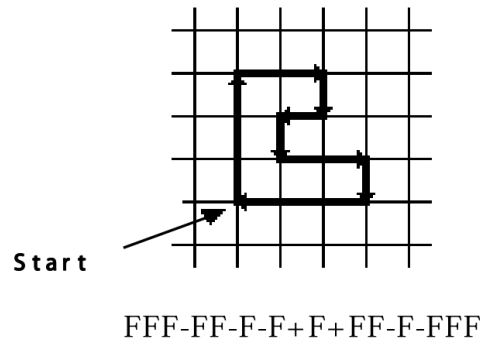


Figure 7 Interpretation of the control string. (Prusinkiewicz P and Lindenmayer A, 1990)

(2) L-system turtle with bracket structure

We will use a pair of brackets introduced by Lindenmayer to deal with tree's branching structure. Suppose a L-system as follows:

Alphabet $V = \{F, +, -, []\}$

Axiom $\omega = F$

Set of rewriting rules $P = \{F \rightarrow F[+F][-F]\}$

Given the length of each move s and the angle increment is δ . The actions that turtle will perform for each character in alphabet are as following:

F : The turtle moves forward a step of length s . The new coordinate of turtle is (x', y') . Suppose the direction vector of turtle's forward is $(\cos\theta, \sin\theta)$, then $x' = x + s \cos\theta$, $y' = y + s \sin\theta$. The turtle draws a line between (x, y) and (x', y') .

$+$: Turn left by 30 degree, the next status of turtle is $(x, y, \sigma + 30)$.

$-$: Turn right by 30 degree, the next status of turtle is $(x, y, \sigma - 30)$.

$[$: Push turtle's status into a stack. The information includes turtle's location, and degree of rotation.

$]$: Pop turtle's information out of a stack as turtle's status.

In this case, if the iteration time is 1, the control string that is generated by L-system is $F[+F][-F]$. Set initial angle $\sigma = 90^\circ$ and the movement unit equals to the length of grid. Then the process of drawing will be as follows:

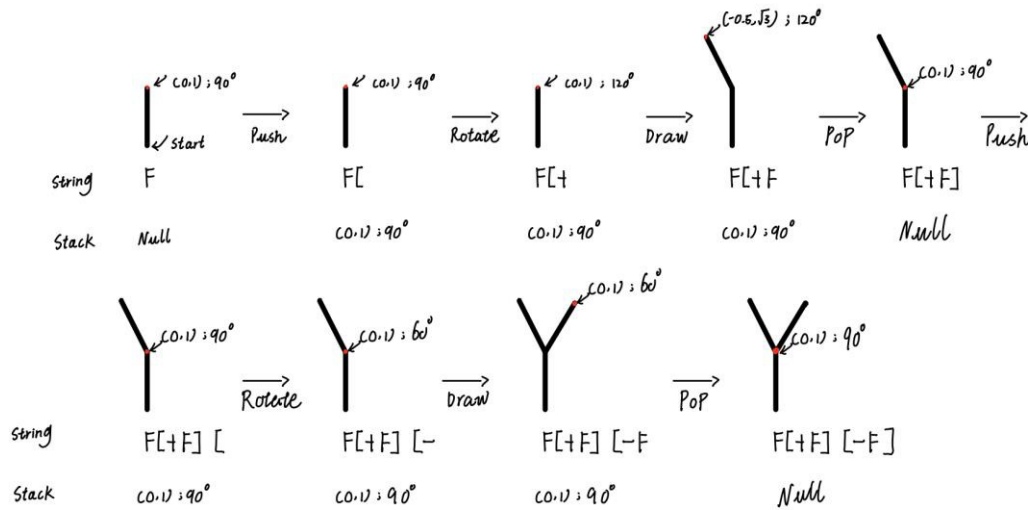


Figure 8 Demonstration of L-system turtle

As we can see, red dot represents the position of the turtle. The information of status includes the coordinate and the degree of rotation of the turtle.

2.3.2 Explanation by L- system turtle in 3D space

According to the theory of Abelson and diSessa[5], the explanation by L-system turtle can be extended to 3D space. The current orientation of turtle can be represented by three vectors, \vec{H} , \vec{L} , \vec{U} , indicating the turtle's heading, the direction to the left, and the direction up[2].

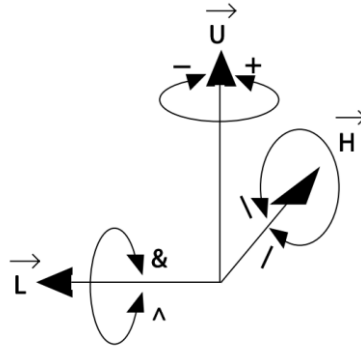


Figure 9 Controlling the turtle in 3D space. (Prusinkiewicz P and Lindenmayer A, 1990)

The orientation of the turtle can be represented by following equation:

$$[\vec{H}', \vec{L}', \vec{U}'] = [\vec{H}, \vec{L}, \vec{U}] R,$$

Where R is a 3×3 rotation matrix [6]. Specifically, rotation by each axis of direction should be represented by 3 matrixes as follows:

$$R_U(\sigma) = \begin{bmatrix} \cos\sigma & \sin\sigma & 0 \\ -\sin\sigma & \cos\sigma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_L(\sigma) = \begin{bmatrix} \cos\sigma & 0 & -\sin\sigma \\ 0 & 1 & 0 \\ \sin\sigma & 0 & \cos\sigma \end{bmatrix}$$

$$R_H(\sigma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\sigma & -\sin\sigma \\ 0 & \sin\sigma & \cos\sigma \end{bmatrix}$$

The following characters control the actions of the turtle:

- + Turn left by angle σ around U axis (Up direction), using rotation matrix $R_U(\sigma)$
- Turn right by angle σ around U axis (Up direction), using rotation matrix $R_U(-\sigma)$
- & Pitch down by angle σ around L axis (Left direction), using rotation matrix $R_L(\sigma)$
- ^ Pitch up by angle σ around L axis (Left direction), using rotation matrix $R_L(-\sigma)$
- \ Roll left by angle σ around H axis (Heading direction), using rotation matrix $R_H(\sigma)$
- / Roll right by angle σ around H axis (Heading direction), using rotation matrix $R_H(-\sigma)$

What needs to be mentioned is that this section is meant to explain the principle of L-system. In this project, the characters, and the rules we use will be based on the rules above. **However, in Unreal**

Engine 4, left-handed coordinate is being used instead of right-handed. During implementation, the action rules will be slightly different than these rules, which will be clarified in the following report.

2.3.3 Parametric L-system

This section only aims to introduce the basic concepts of the construction of sympodial tree by parametric L-system. The actual implementation will be based on this theory and there are several modifications being made because of some features of Unreal Engine 4, which will be explained in the following report.

(1) Parametric L-system [7]

Based on L-system, parametric L-system introduces parametric symbols. A formalized parametric L-system is an ordered tetrad as follows:

$$G = \langle V, \Sigma, \omega, P \rangle$$

Specifically, V is the alphabet of system, or all potential symbols in the string. ω is the initial string of system, which is the Axiom. P is a finite set of rewriting rules. Σ is a set of parameters that control the shape of tree. In parametric L-system, a rewriting rule is made of three parts: Prerequisite, condition, and result.

$$prer : cond \rightarrow res$$

An example of a parametric L-system is as follow:

Alphabet $V = \{A, B, C, D\}$

Parameters set $\Sigma = \{t\}$

Axiom $\omega = A(9)$

Set of rewriting rules $P = \{A(t): t > 5 \rightarrow B(t + 1)CD(t - 1)\}$

According to the rules of rewriting, the prerequisite is $A(t)$, the condition is $t > 5$, and the result is $B(t + 1)CD(t - 1)$. As we can see, when $\omega = A(9)$, $t = 9$ and $t > 5$. Therefore, the result of rewriting string will be replaced by $B(t + 1)CD(t - 1)$, which will be $B(10)CD(9)$.

2.4 Brief introduction of Unreal Engine 4

2.4.1 Map

A map, or a scene in Unreal Engine 4 is a three-dimensional space where every object is placed. In a map, a left-handed cartesian coordinate system is used as the world coordinate system. This coordinate system is formed by x, y, z axis where the x axis indicates the front direction, the y axis indicates the right direction, and the z axis indicates the up direction by default.

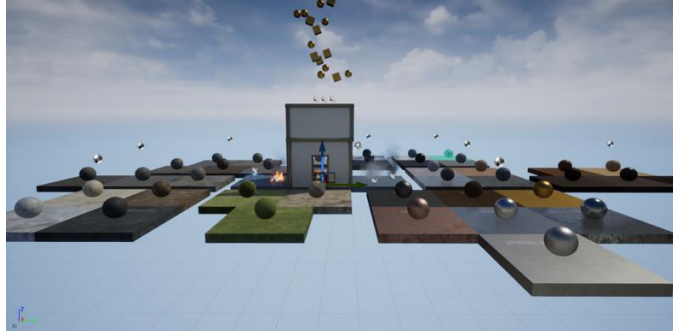


Figure 10 A map in Unreal Engine 4

2.4.2 Actor class

Actor is the base class for an Object that can be placed or spawned in a level. Actors may contain a collection of ActorComponents, which can be used to control how actors move, how they are rendered, etc. An actor object contains two 3D coordinate to represent the location and the degree of rotation.

Location ▾	X 275.0	Y -50.0	Z 32.0	↗
Rotation ▾	X 0.000004 °	Y -0.000013 °	Z 63.749893 °	↗

Figure 11 Location and rotation



Figure 12 Location and rotation of an object

The location and rotation of an actor object can be represented by two types of coordinate systems. One is the local coordinate system; the other is the world coordinate system. The world coordinate system is used when we try to describe a location or rotation of an object relative to the map. The local (or relative) coordinate system is used to describe a location or rotation relative to objects other than the map (It can be the object itself).

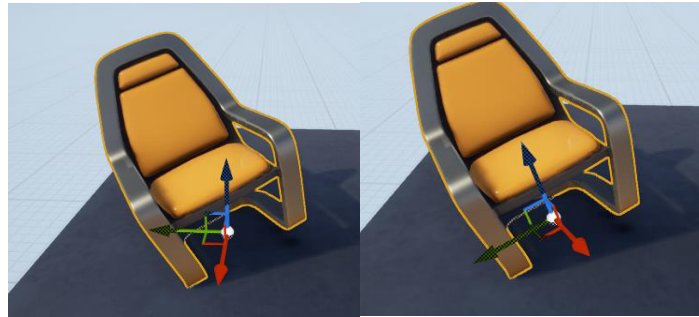


Figure 13 The world coordinate(left) and local coordinate system(right)

2.4.2 Actor's component

Actors can be thought of, in one sense, as containers that hold special types of Objects called Components. Different types of Components can be used to control how Actors move, how they are rendered, etc. The other main function of Actors is the replication of properties and function calls across the network during play [8].

Actors support having a hierarchy of SceneComponents. Each Actor also has a RootComponent property that designates which Component acts as the root for the Actor. Actors themselves do not have transforms, and thus do not have locations, rotations, or scales. Instead, they rely on the transforms of their Components; more specifically, their root Component.

In this project, an important type of component called “Static mesh component” is used frequently. The StaticMeshComponent is used to create an instance of a UStaticMesh. A Static Mesh is a piece of geometry that consists of a static set of polygons and are the basic unit used to create world geometry for levels in Unreal Engine 4. In addition to building levels, Static Meshes can be used for creating movers such as doors or lifts, rigid body physics objects, foliage and terrain decorations, procedurally created buildings, game objectives, and many more visual elements.

Below, a StaticMeshComponent representing a ceiling light mesh.

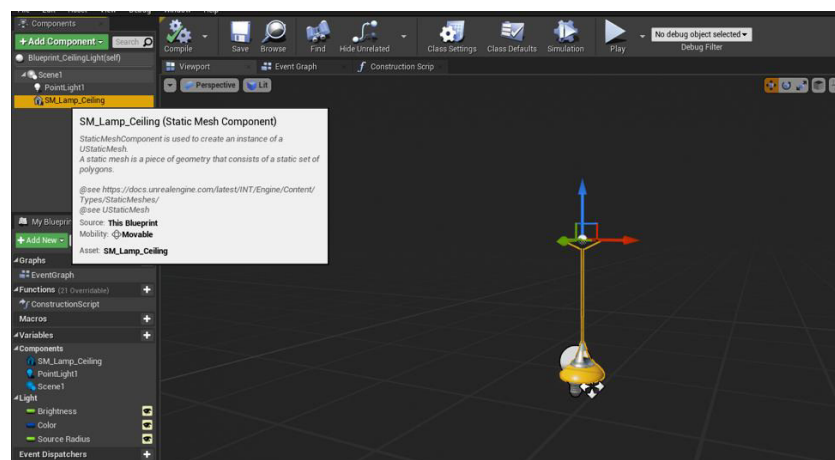


Figure 14 A static mesh actor

2.4.3 Textures

Textures are image assets that are primarily used in Materials but can also be directly applied outside of Materials, like when using an texture for a heads up display (HUD).

For Materials, textures are mapped to surfaces which the Material is applied to. Textures can be used for a variety of calculations within a Material by being applied directly to an input (such as, Base Color), used as a mask, or using the RGBA values for other calculations [9].



Figure 15 Branches with different materials

2.4.4 Game modes

While certain fundamentals, like the number of players required to play, or the method by which those players join the game, are common to many types of games, limitless rule variations are possible depending on the specific game you are developing. Regardless of what those rules are, Game Modes are designed to define and implement them. There are currently two commonly used base classes for Game Modes [10]. All Game Modes are subclasses of 'AGameModeBase', which contains considerable base functionality that can be overridden.

In this project, a 'AGameModeBase' class is used to create and spawn any objects that relate to L-system when program starts to run.

3.Specification and Design

In this section, we will focus on specifying the design of the system and how it works overall. There are three components in this system, a sympodial tree component explained by parametric L-system, a L-system turtle component that executes the render process, and finally, mesh component that provides visual structure of trees.

3.1 Parametric L-system construction of sympodial tree component

3.11 Construction of sympodial tree

Trees with sympodial branching have unfolding tree crowns without distinct tree trunks, compared to monopodial trees. As a sympodial tree grows, the width and length of branches become smaller. The shape of the tree crown is mainly decided by the order of branches and the angle of branches. The shape of a typical type of tree should be random to look natural.

Based on the features that are mentioned above, we set the order of branches and the angle of branches as parameter to control the iteration times and the direction of growing. Setting length of branch as parameter to control the length of each order of branch as well as the width of branch.

To generate a random shape, there are 2 types of branching conditions. One is double branching, which means that there will be 2 child branches on 1 parent branch. The other is triple branching, which means 3 child branches on 1 parent branch. Therefore, 2 rewriting rules are required.

A parametric L-system for sympodial trees is as follows:

$$G = \langle V, \Sigma, \omega, P, \pi \rangle$$

Alphabet $V = \{B, F, [], !, @, +, /\}$

Parameters set $\Sigma = \{L, w, L_1, L_2, L_3, L_4, L_5, W_1, W_2, W_3, W_4, W_5, \beta_1, \beta_2, \beta_3, \theta_1, \theta_2, \theta_3\}$

Axiom $\omega = B(L_0, W_0)$

Probability $\pi = \{\alpha, 1 - \alpha\}$

Set of rewriting rules

$$P = \begin{cases} B(L, w): L \geq \min \xrightarrow{\alpha} !(w)F(L)[+(\beta_1)/(\theta_1)B(L_1, W_1)][+(\beta_2)/(\theta_2)B(L_2, W_2)][+(\beta_3)/(\theta_3)B(L_3, W_3)] \\ B(L, w): L \geq \min \xrightarrow{1-\alpha} @(w)F(L)[+(\beta_1)/(\theta_1)B(L_4, W_4)][+(\beta_2)/(\theta_2)B(L_5, W_5)] \end{cases}$$

There are 2 rewriting rules within our parametric L-system. To specify them, we use P_1 and P_2 to represent them as follow:

$$P_1: B(L, w): L \geq \min \xrightarrow{\alpha} !(w)F(L)[+(\beta_1)/(\theta_1)B(L_1, W_1)][+(\beta_2)/(\theta_2)B(L_2, W_2)][+(\beta_3)/(\theta_3)B(L_3, W_3)]$$

$$P_2: B(L, w): L \geq \min \xrightarrow{1-\alpha} @(w)F(L)[+(\beta_1)/(\theta_1)B(L_4, W_4)][+(\beta_2)/(\theta_2)B(L_5, W_5)]$$

We will still use a pair of brackets introduced by Lindenmayer to deal with tree's branching structure.

[: Push turtle's status into a stack. The information includes turtle's location, direction vectors (up, right, forward) and degree of rotation.

]: Pop turtle's information out of a stack as turtle's status.

As we can see in this L-system, B and C can be seen as vertexes. The rewriting rule P_1 means that there is α percent chance for a vertex B being replaced by a branch and three new vertexes B, B, B, which indicates triple branching. P_2 means that there is $1 - \alpha$ percent for a vertex B being replaced by a branch and two new vertexes B, B, which indicates double branching. L_0 stands for the initial length of the tree trunk while W_0 represents the initial width of the tree trunk. L_i and L represent the length of branches, W_i and W stand for the width of the branches. β_i and θ_i represent the angle of branches, which control the direction of growing. \min gives out the threshold value of branches' length. When a branch is shorter than \min , it cannot be generated.

Next, we will introduce more detail information about branches' length and width.

(1) Branch length

To gradually shorten the branches, shrink factors r_i are being introduced. As we can see in the L-system demonstration, there are 5 different lengths being assigned to different branches. $L_1 = L * r_1$, $L_2 = L * r_2$, $L_3 = L * r_3$, $L_4 = L * r_4$, $L_5 = L * r_5$, while $L_i, i = 1,2,3$ represent the lengths for triple branching, $L_i, i = 4,5$ represent the lengths for double branching.

(2) Branch width

To make tree more natural, we will set the factors of width based on the Leonardo's Rule[11]. The rule says that when a tree's trunk splits into two branches, the total cross section of those secondary branches will equal the cross section of the trunk [12]. The formula of it is:

$$w^2 = \sum_{i=1}^N w_i^2$$

where the w_i is the width of the i^{th} child branch. Based on this rule, let $w_1 = w * p_1^e$, $w_2 = w * q_1^e$, $w_3 = w * (1 - p_1 - q_1)$, $w_4 = w * p_2^e$, $w_5 = w * q_2^e$, $p_1 + q_1 < 1$, $p_2 + q_2 = 1$. Here, p_i, q_i, e control the width of the branch. According to Leonardo's Rule, let $e = 0.5$ so that the total cross section of those secondary branches will equal the cross section of the parent branch. Therefore, we will get $w^2 = w_1^2 + w_2^2 + w_3^2$ for the rewriting rule P_1 , $w^2 = w_4^2 + w_5^2$ for the rewriting rule P_2 .

3.2 L-system turtle component

In this section, we will introduce how L-system turtle component works in this project. Overall, L-system turtle reads through the control string generated by tree component and perform different actions based on different character.

3.2.1 Basic information to describe the status of L-system turtle

L-system turtle is derived from Actor. The status of L-system turtle can be described by three direction vectors as well as the coordinates of location and rotation.

The direction vectors include up, right and front vectors. The reason for maintaining these vectors is that L-system turtle always moves toward "up direction". However, the "up direction" is not always vertical to the ground level. It changes as the turtle rotates to make certain angles between different branches.

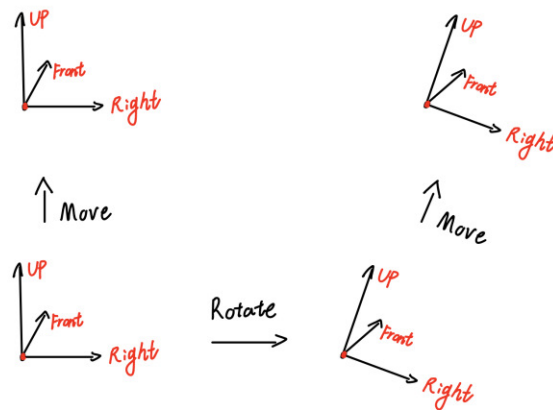


Figure 16 Demonstration of direction vectors

The location and rotation of L-system turtle is represented by **world coordinate**. The location and rotation of a tree component (like a trunk) are same as the ones of turtle.

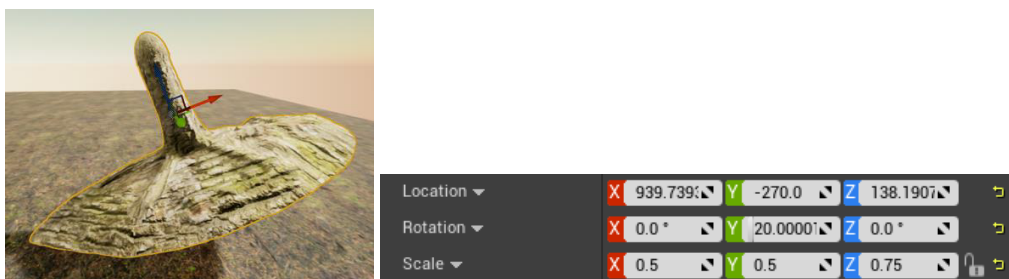


Figure 17 Demonstration of a trunk and its information

3.2.2 Actions of L-system turtle

F: The turtle moves towards the direction of up vector with the length of a branch.

+: Rotate the turtle actor around up vector.

/: Rotate the turtle actor around front vector.

[: Push turtle's status into a stack. The information includes turtle's location, rotation, direction vectors (up, right, forward).

]: Pop turtle's information out of a stack as turtle's status.

!: Prepare to render triple branches.

@: Prepare to render double branches.

(1) Working process of L-system turtle

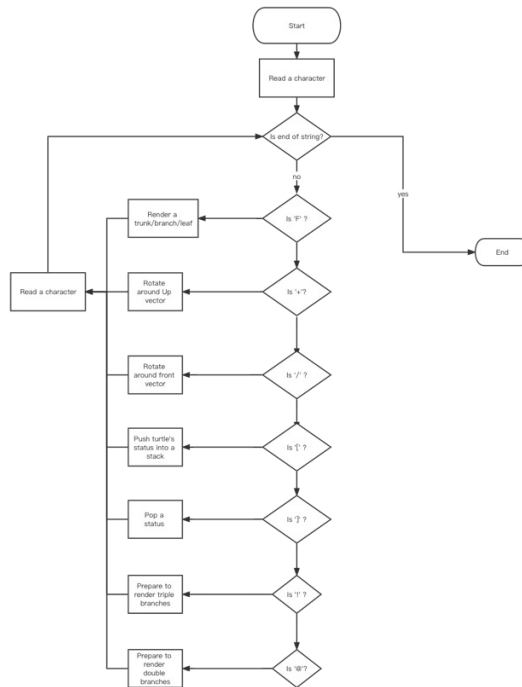


Figure 18 Working process

3.3 Mesh component

In this project, there are 3 types of mesh component: tree trunk, branch, and leaf. They will be used and placed by L-system turtle component. In Unreal Engine 4, they are static mesh actors with static mesh components.

3.3.1 Tree trunk

The tree trunk component is consisted of a tree trunk mesh and a sphere mesh. The sphere is located on the top of the trunk to act as a joint. This joint makes branching point more natural and materials of joint and trunk are consistent.

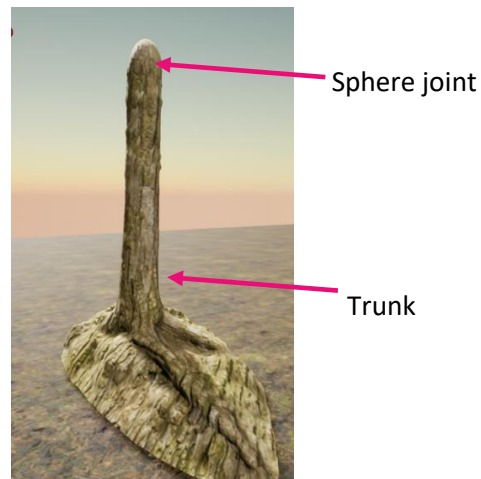


Figure 19 Tree trunk component

3.3.2 Branch

The branch component is consisted of a branch mesh and a sphere mesh. The sphere is located on the top of the branch to act as a joint. This joint makes branching point more natural and materials of joint and trunk are consistent.

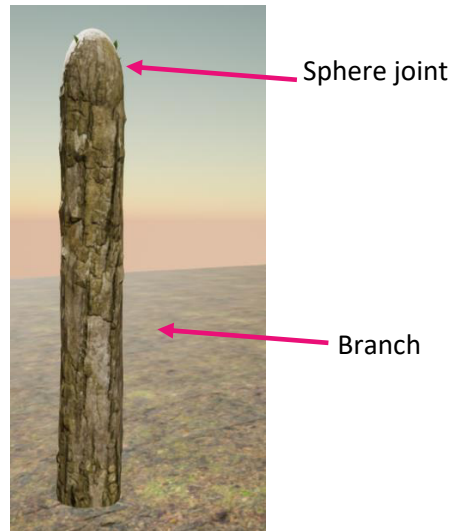


Figure 20 Branch component

3.3.3 Leaf

The leaf component is only consisted of a leaf mesh.



Figure 21 Leaf component

3.4 Working mechanism of the system

In this system, three components work together to generate a realistic sympodial tree. First, the tree component will generate a control string that represents tree's structure. It also provides every parameter that control the shape of the tree to the L-system turtle component. Second, the L-system turtle component is spawned at a specific location in a map with a preset rotation. It reads every

character in the control sentence and makes corresponding actions. L-system turtle is also responsible to spawn and adjust mesh component in the map to form the tree.

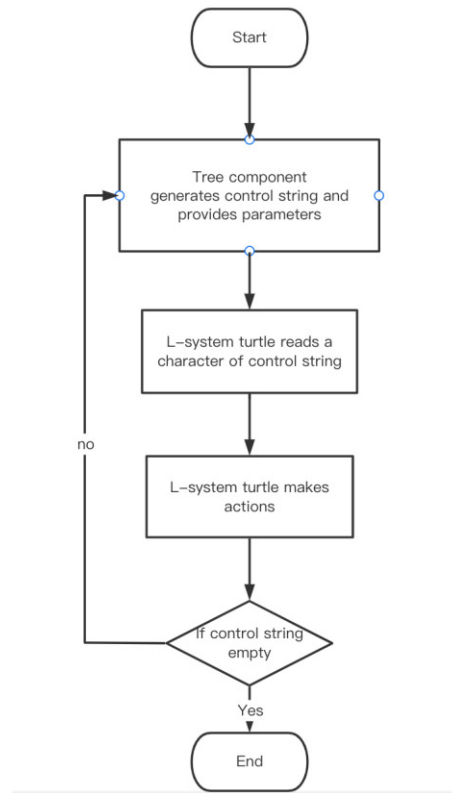
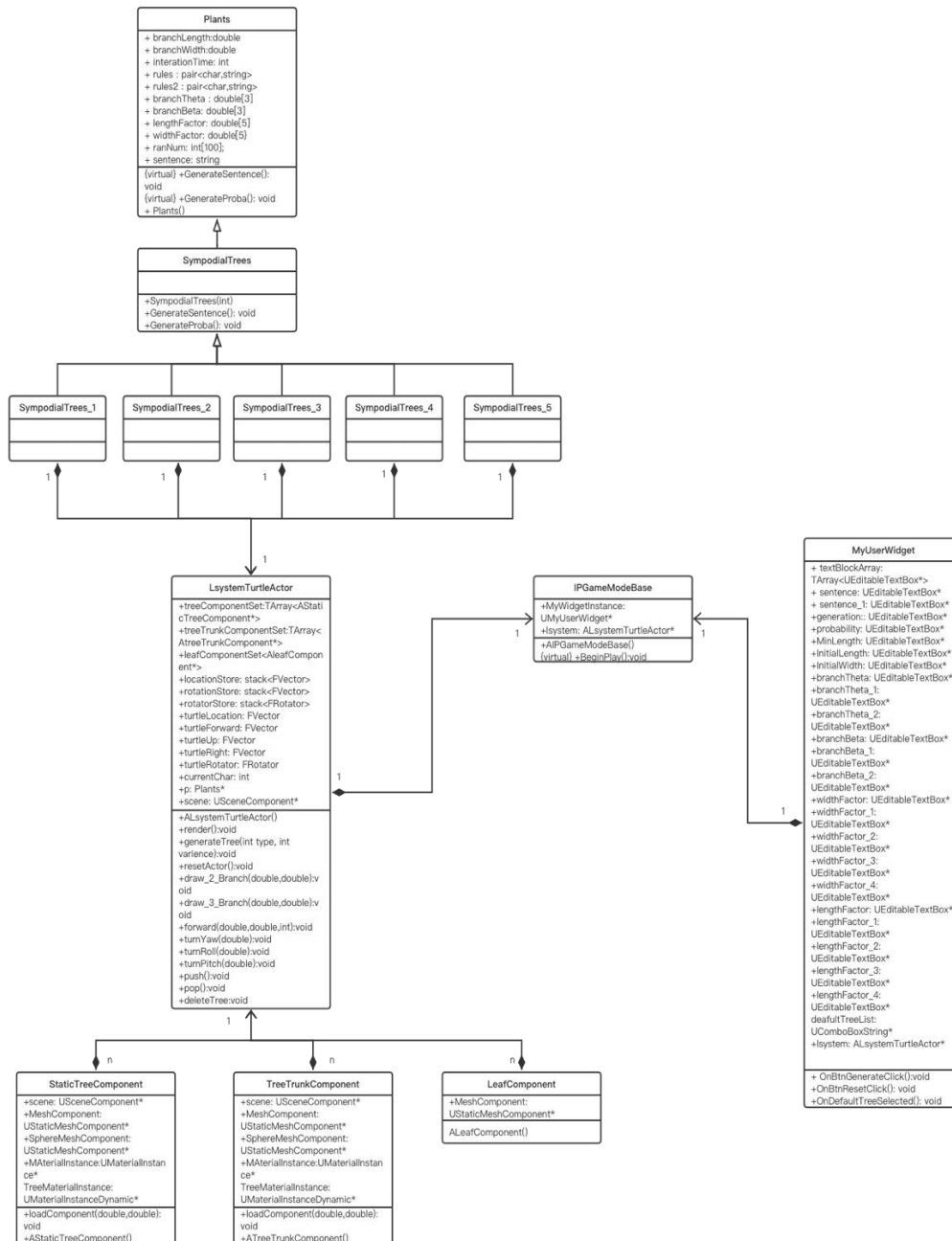


Figure 22 Working mechanism

4.Implementation

In this section, we will start with demonstrating a UML class diagram. Then coding details of each component will be introduced. Lastly, some problems that are difficult to solve during implementation will be demonstrated. The attempts to solutions are also included.

4.1 UML Class diagram



4.2 Tree component

4.2.1 Generate random numbers

There are 2 types of branching, which are double branching and triple branching as we mentioned above. However, to make the tree more thriving, the probability for generating triple branching is higher.

Therefore, the system will generate a random integer between 6 and 8, let it be A. When system rewrites a character, it generates an integer between 1 to 10 again. If this new integer is between 1 to A, then a triple branching is generated. If it is between A to 10, then a double branching is generated. Overall, the probability of generating triple branching is between 60% to 80% and the probability for double branching is between 20% to 40%.

In C++, function rand() requires a “seed” to generate different sequence of random numbers. In practice, we usually use the current time of system as the seed. However, there could be more than 10k branches on a tree, which means the systems will get system time constantly in a short time. The time gap between several access of system time could be too short to make the seed changed. Therefore, the rand() function will keep generating the same random number sequence. In this project, system will generate 100 random numbers each time to avoid repetition of random numbers sequence.

```
srand((unsigned)time(NULL));  
branchProbability = (rand() % (8 - 6 + 1)) + 6;
```

Code 1 Generate probability for different branching(SympodialTree_3.cpp)

```
void SympodialTrees::GenerateProba()  
{  
  
    srand((unsigned)time(NULL));  
  
    for (int i = 0; i < 100; i++) {  
        ranNum[i] = (rand() % (10 - 1 + 1)) + 1;  
    }  
}
```

Code 2 Generate random numbers sequence(SympodialTree.cpp)

4.2.2 Generate sentence

To generate sentence, the system will read the sentence character by character, rewriting character that fulfill the rewriting rule. More specifically, every time the system rewrites a sentence, a new ‘string’ will be created. If the system reads a character that does not fulfill the rule, it duplicates the character and append it to the end of the new string. If the system reads a character that fulfill the rule, it appends the corresponding replacement to the end of the new string.


```

GenerateProba();
for (int i = 0; i < iterationTime; i++) {
    string nextSentence = "";
    for (int32 k = 0; k < sentence.size(); k++) {

        char currentChar = sentence.at(k);
        //take one random number from the sequence
        int probability = ranNum[ranCounter++];

        if (currentChar == rules.first && (probability >= 1 && probability <= branchProbability) ) {

            nextSentence.append(rules.second);
            branch_3_count++;
        }
        else if (currentChar == rules2.first && (probability > branchProbability)) {
            nextSentence.append(rules2.second);
            branch_2_count++;
        }
        else {
            nextSentence += currentChar;
        }

        // The sequence only contains 100 random numbers, if system has already
        //rewrote 100 character,
        // update the sequence
        if (ranCounter == 100)
        {
            GenerateProba();
            ranCounter = 0;
        }
    }
    sentence = nextSentence;
}

```

Code 3 Rewrite character(SympodialTree.cpp)

4.3 L-system turtle component

4.3.1 Overall working process: Iteration

First, we will explain why L- system turtle needs an iteration program to read and render the tree.

An example of a 2-generation tree and its sentence are as follow:

$$\begin{aligned} & !F[+/!F[+/!F[+/B][+/B][+/B]][+/!F[+/B][+/B][+/B]][+/!F[+/B][+/B][+/B]][+/!F[+/@F[+/B][+/B]][+/!F[+/B][+/ \\ & B][+/B]][+/!F[+/B][+/B][+/B]][+/@F[+/!F[+/B][+/B][+/B]][+/!F[+/B][+/B][+/B]]] \end{aligned}$$



Figure 23 2-generation tree

As we know in the previous sections, the width and length shrinking factors of a tree are fixed. They are used repeatedly on different triple branching and double branching. Therefore, the distribution of those factors is a breath first process as following picture (First number denotes the sequence number of the branch, 'W' means Width factor, 'L' means Length factor)

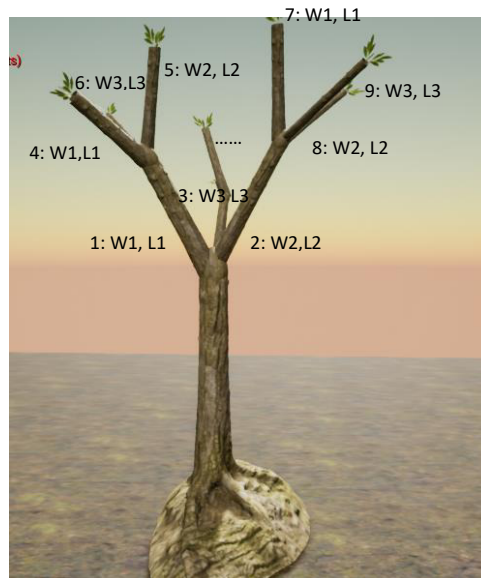


Figure 24 Demonstration of factors distribution

However, in practice, the sentence of the tree is generated in a depth-first way. The system can only read through the string from the beginning to the end. Therefore, the actual render process of the L-system turtle should be as follows:

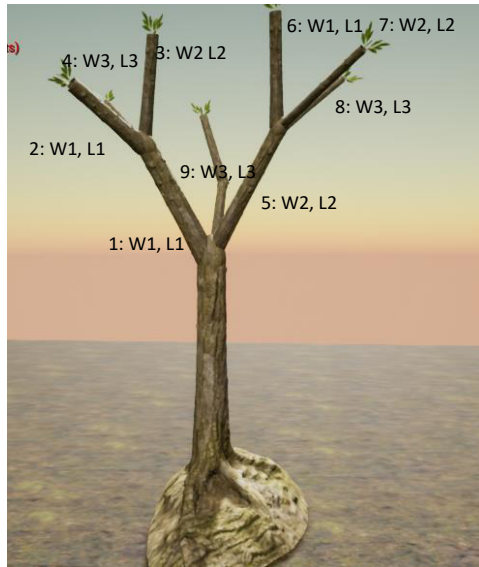


Figure 25 The actual render process

As a result, before the L-system turtle starts to render a child branch, the index for width and length factors should be reserved and an iteration process is needed. If system reads a '!' or '@', it will start a new render function for triple branching or double branching where the index for length and width factors can be reset. Current render function waits until the inner iterations end.

An iteration ends when system finishes reading a substring representing a triple branching or double branching. However, for the first iteration, it ends when system finishes reading the whole sentence.

```
void ALsystemTurtleActor::draw_2_Branch(double branchLength, double branchWidth)
{
    int count = 0;
    int length = p->rules2.second.size() - 1;
    int angleIndex = 0;
    int lengthFactorIndex = 3;
    int lable = 0;

    double rotateX = 0;
    branchLength = (branchLength < p->minLength) ? p->minLength : branchLength;
    lable = (currentChar == 1) ? 1 : lable;

    bool endingCondition = true;

    while (endingCondition) {
        char current = p->sentence.at(currentChar++);

        if (current == '!')
        {
            double lengthFactor = p->lengthFactor[lengthFactorIndex];
            double widthFactor = p->widthFactor[lengthFactorIndex++];
            draw_3_Branch(branchLength * lengthFactor, branchWidth * widthFactor);
        }
    }
}
```

```

    }

    else if (current == '@')
    {

        double lengthFactor = p->lengthFactor[lengthFactorIndex];
        double widthFactor = p->widthFactor[lengthFactorIndex++];
        draw_2_Branch(branchLength * lengthFactor, branchWidth * widthFactor);
    }
    else if (current == 'F')
    {

        forward(branchLength, branchWidth, 0, rotateX);
    }
    else if (current == '+')
    {

        turnYaw(p->branchBeta[angleIndex]);
    }
    else if (current == '/')
    {

        rotateX = p->branchTheta[angleIndex];
        turnRoll(p->branchTheta[angleIndex++]);
    }
    else if (current == '&')
    {

        turnPitch(p->branchBeta[angleIndex++]);
    }
    else if (current == '[')
    {

        push();
    }
    else if (current == ']')
    {

        pop();
    }
    else
    {

        forward(branchLength, branchWidth, 1, rotateX);
    }
    count++;
    if (lable == 1)
    {

        endingCondition = ((!(count >= length)) || !(currentChar >= p->sentence.size() - 1));
    }
    else {

        endingCondition = (!(count >= length));
    }
}
}

```

Code 4 Double-branching (LsystemTurtleActor.cpp)

4.3.2 Rotation of L-system turtle

There are 2 types of rotation actions for L-system turtle. One is rotating around “Up” vector, the other is rotating around “Front” vector. Like the distribution of length and width factors, the rotation angles around “up” vector and “front” vector are the same among double branching or triple branching.

```
        else if (current == '+')
        {
            turnYaw(p->branchBeta[angleIndex]);
            count++;
        }
        else if (current == '/')
        {
            turnRoll(p->branchTheta[angleIndex++]);
            count++;
        }
    }
```

Code 5 Angle distribution (LsystemTurtleActor.cpp)

```
SympodialTrees_3::SympodialTrees_3(int iterationTime) : SympodialTrees(iterationTime)
{
    //z
    branchBeta[0] = -30.0f;
    branchBeta[1] = 30.0f;
    branchBeta[2] = 90.0f;

    //x
    branchTheta[0] = -30.0f;
    branchTheta[1] = 35.0f;
    branchTheta[2] = -40.0f;
}
```

Code 6 Angle declaration of sympodial tree (SympodialTree_3.cpp)

In Unreal Engine 4, to rotate around an actor’s “local coordinate axis”, (in this case, the local coordinate axis is same as actor’s Up, Front or Right vector), a useful function “AddActorLocalRotation” can be used. The parameter for this function is a Vector structure called “FRotator”. It denotes the rotation angles around Z axis (Up), Y axis (Right), and X axis (Front). For example, AddActorLocalRotation (FRotator(100,0,0)) means rotate the actor around local Z axis by 100 degrees.

```

void ALsystemTurtleActor::turnYaw(double branchAngle)
{
    turtleRight = turtleRight.RotateAngleAxis(branchAngle, turtleUp);
    turtleFoward = turtleFoward.RotateAngleAxis(branchAngle, turtleUp);

    AddActorLocalRotation(FRotator(0, branchAngle, 0));
}

void ALsystemTurtleActor::turnRoll(double branchAngle)
{
    turtleRight = turtleRight.RotateAngleAxis(-branchAngle, turtleFoward);
    turtleUp = turtleUp.RotateAngleAxis(-branchAngle, turtleFoward);

    AddActorLocalRotation(FRotator(0, 0, branchAngle));
}

```

Code 7 Rotation (LsystemTurtleActor.cpp)

In the previous parts of the report, we mentioned that the L-system turtle actor needs to maintain and update three direction vectors up, right and front vector. The reason why we need to keep them is that Unreal Engine 4 does not provide real time direction vectors of an actor. Therefore, if an actor has rotated before, we cannot get the current direction vectors of it. The front actor is necessary for an L-system actor to confirm the next location.

Therefore, as we showed above, the direction vectors are also rotated by using RotateAngleAxis() function. This function allows a vector to rotate around another vector, which is suitable in this case.

4.3.3 Forward

In L-system, turtle will move to next location after several rotations. In nature, tree always grow vertically. Therefore, in this project, the turtle will always move towards the direction of 'Up' vector. Here, we will first introduce how L-system turtle moves to the next location.

As we know in the previous section, the length of a branch is acknowledged. Therefore, we can get:

$$NextLocation = currentLocation * Up$$



Figure 26 Demonstration of Moving

To spawn a static mesh actor, we need to input a location as well as a rotation. We have already known the rotation for it, which is the rotation of L-system actor. Now, we will focus on the location for this newly created mesh component.

As we can see in the following pictures, the branch and trunk can be approximately seen as cylinders. In Unreal Engine 4, the position of a static mesh is confirmed by the center point of it. Therefore, we can align the center of the branch (or trunk) to the middle point between turtle's current location and the next location. It can be represented as:

$$\text{MeshLocation} = (\text{currentLocation} + \text{nextLocation})/2$$



Figure 27 Demonstration of static mesh actors

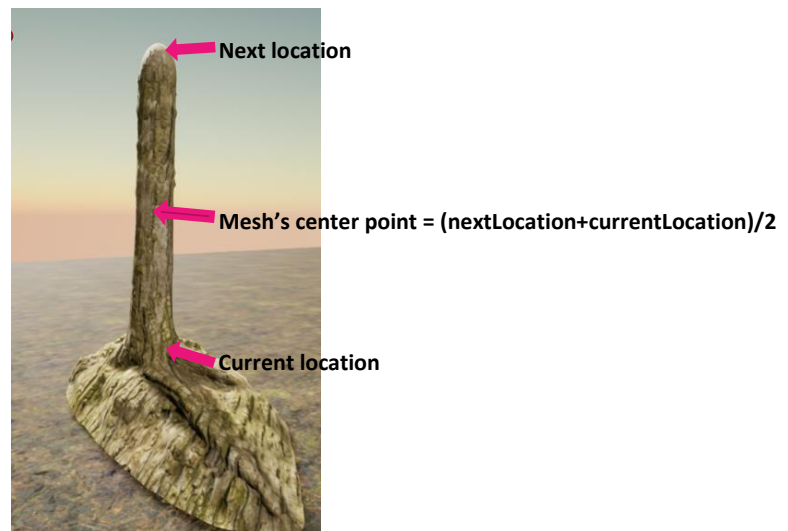


Figure 28 Confirming location (Ideal design)

This solution is not perfect. Ideally, the vertical length of the mesh should **be the same as** the length of branch that we preset. Yet, they are slightly different. The length of the branch could be longer than the actual length of the mesh. The scale of the mesh is not defined by concrete numbers. It needs to be adjusted by multiplying constants in different dimensions (x, y, or z). The length and width of branches are only the parameters that affect the actual length of width of the mesh. Here, we call the location calculated by the equation below as 'Preset next location'

$$\text{Preset next location} = \text{currentLocation} * Up$$

As a result, we need to manually set the next location as **the location of sphere component** of branches or trunk after the mesh actor is spawned. So that the child branches will be generated at correct location.

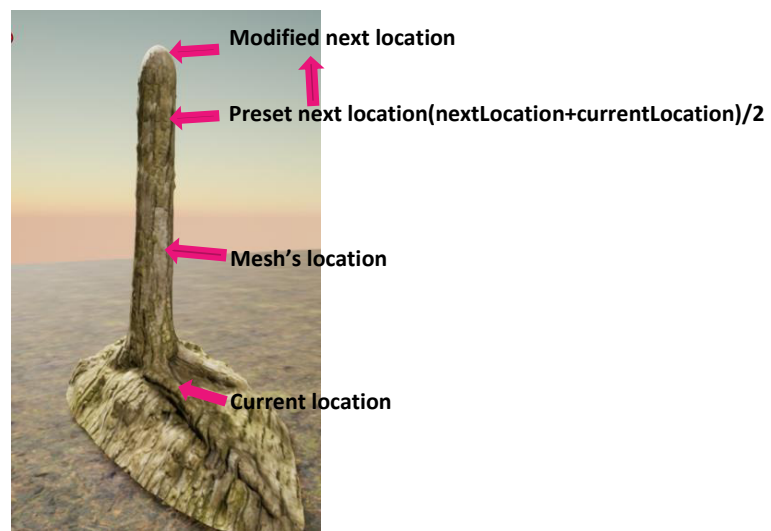


Figure 29 Actual confirmation of next location

```

if (currentChar == 2)
{
    ATreeTrunkComponent* treeComponent = GWorld->GetWorld()-
>SpawnActor<ATreeTrunkComponent>(treeComponent->StaticClass(), (turtleLocation + nextLocation) / 2, GetActorRotation());
    treeComponent->loadComponent(branchWidth, branchLength);
    treeTrunkComponentSet.Add(treeComponent);
    turtleLocation = treeComponent->SphereMeshComponent->GetComponentLocation();
}
else
{
    AStaticTreeComponent* treeComponent = GWorld->GetWorld()-
>SpawnActor<AStaticTreeComponent>(treeComponent->StaticClass(), (turtleLocation + nextLocation) / 2, GetActorRotation());
    treeComponent->loadComponent(branchWidth, branchLength);
    treeComponentSet.Add(treeComponent);
    turtleLocation = treeComponent->SphereMeshComponent->GetComponentLocation();
}

```

Code 8 Spawning static mesh actors (LsystemTurtleActor.cpp)

4.3.4 Push and pop

The status of L-system turtle is consisted of location, rotation and direction vector (up, forward, right). Therefore, 2 stacks that store FVector and 1 stack that stores FRotator are needed.

```
void ALsystemTurtleActor::push()
{
    locationStore.push(turtleLocation);
    rotationStore.push(turtleFoward);
    rotationStore.push(turtleUp);
    rotationStore.push(turtleRight);

    rotatorStore.push(GetActorRotation());
}

void ALsystemTurtleActor::pop()
{
    turtleLocation = locationStore.top();
    locationStore.pop();

    turtleRight = rotationStore.top();
    rotationStore.pop();

    turtleUp = rotationStore.top();
    rotationStore.pop();

    turtleFoward = rotationStore.top();
    rotationStore.pop();

    SetActorRotation(rotatorStore.top());
    rotatorStore.pop();
}
```

4.3.5 Delete tree

To change the shape of the tree in real time, we need a delete function for L-system turtle. L-system turtle should delete all components of a tree and starts to create a new one.

In Unreal Engine 4, an object of actor class is declared as a reference. Therefore, we can save all references of component objects into arrays. Iterating these arrays and delete every component object by using function Destroy(). The Destroy() function is an embedded function of Actor.

```
ALeafComponent* leafComponent = GWorld->GetWorld()-
>SpawnActor<ALeafComponent>(leafComponent->StaticClass(), turtleLocation,
GetActorRotation());
leafComponent->SetActorRelativeScale3D(FVector(1, 1.5, 1));

if (treeComponentSet.Num() != 0)
{
    treeComponentSet.Last()->SphereMeshComponent->SetVisibility(false);

    //Add this component to the array
    leafComponentSet.Add(leafComponent);
}
```

```

        if (currentChar == 2)
        {
            ATreeTrunkComponent* treeComponent = GWorld->GetWorld()-
>SpawnActor<ATreeTrunkComponent>(treeComponent->StaticClass(), (turtleLocation + nextLocation) / 2, GetActorRotation());
            treeComponent->loadComponent(branchWidth, branchLength);

            //Add this component to the array
            treeTrunkComponentSet.Add(treeComponent);
            turtleLocation = treeComponent->SphereMeshComponent->GetComponentLocation();
        }
        else
        {
            AStaticTreeComponent* treeComponent = GWorld->GetWorld()-
>SpawnActor<AStaticTreeComponent>(treeComponent->StaticClass(), (turtleLocation + nextLocation) / 2, GetActorRotation());
            treeComponent->loadComponent(branchWidth, branchLength);
            //Add this component to the array
            treeComponentSet.Add(treeComponent);
            turtleLocation = treeComponent->SphereMeshComponent->GetComponentLocation();
        }
    }
}

```

Code 9 Store references (LsystemTurtleActor.cpp)

```

void ALsystemTurtleActor::deleteTree()
{
    for (auto n : treeComponentSet) {
        n->Destroy();
    }

    for (auto n : leafComponentSet) {
        n->Destroy();
    }

    for (auto n : treeTrunkComponentSet) {
        n->Destroy();
    }
}

```

Code 10 Iterate arrays (LsystemTurtleActor.cpp)

4.4 Mesh component

4.4.1 Scale of the mesh

In the previous section, we know that the parameters ‘width’ and ‘length’ of branches are the factors that affect the actual width and length of branch meshes. In this section, we will focus on the implementation of adjusting scale of meshes.

First, we need to clarify that the original meshes of three components (branch, trunk, leaf), are out of shape. They need to be adjusted in the first place of spawning.



Figure 30 A branch, trunk, and leaf (Original)

Specifically, the first phase of scale adjustment takes place in the construction function.

```
FVector scale = MeshComponent->GetRelativeScale3D();
scale.X *= 4.056;
scale.Y *= 5.131;
MeshComponent->SetRelativeScale3D(scale);
```

Code 11 First scaling adjustment of branch (StaticTreeComponent.cpp)

```
FVector scale = MeshComponent->GetRelativeScale3D();
scale.X *= 3.556;
scale.Y *= 4.031;
MeshComponent->SetRelativeScale3D(scale);
```

Code 12 First scaling adjustment of trunk (TreeTrunkComponent.cpp)

Second phase of scaling adjustment happens after mesh actors are spawned. The mesh component actors provide a method to input the branch length as parameter.

```
void AStaticTreeComponent::loadComponent(double width, double length)
{
    SetActorScale3D(FVector(width / 82.0f, width / 82.0f, length / 60.0f));
    //MeshComponent->SetWorldScale3D(FVector(width / 100.0f, width / 100.0f, length / 100.0f));
    SphereMeshComponent->SetWorldScale3D(FVector(width / 140.0f, width / 140.0f, width / 80.0f));
}
```

Code 13 Method to adjust scale (StaticTreeComponent.cpp)

```

void ATreeTrunkComponent::loadComponent(double width, double length)
{
    SetActorScale3D(FVector(width / 137.0f, width / 137.0f, length / 150.0f));
    //MeshComponent->SetWorldScale3D(FVector(width / 100.0f, width / 100.0f, length / 100.0f));
    SphereMeshComponent->SetWorldScale3D(FVector(width / 130.0f, width / 128.0f, width / 73.0f));
}

```

Code 14 Method to adjust scale (TreeTrunkComponent.cpp)

4.5 User interface

In this section, we will not discuss the declaration of every component in this interface. Instead, we will introduce how input data transmits from user interface to L-system turtle.

4.5.1 Overall design of user interface



Figure 31 Design of user interface

Users can input the parameters that control the shape of the tree. Five default settings of trees are provided. When users click generate, the system will generate a tree in the middle of the screen. If user click reset, all input content can be erased.

4.5.2 Implementation of user interface

In Unreal Engine 4, a user interface can be designed by a visual development tool called 'Blueprint'. The design above is done by blueprint. Yet, the function of user interface should be developed in C++ in this project. Therefore, a link between 'blueprint' design and C++ class is necessary.

First, we create a C++ widget class called "MyWidget". Then we can create a blueprint widget class derived from "MyWidget".



Figure 32 A widget blueprint class

After finishing the visual design of the widget, we can connect the blueprint widget class to the C++ widget class each component by each component with the following code. This is only an example of implementation of button, there are many other components in this widget class.

```
if (UButton* btn = Cast<UButton>(GetWidgetFromName("Generate")))
{
    FScriptDelegate Del;
    Del.BindUFunction(this, "OnBtnGenerateClick");
    btn->OnClicked.Add(Del);
}
```

Code 15 An Example of connecting component (MyUserWidget.cpp)

Next, in the game mode class, we need to instantiate the widget class like below:

```
// Check if there is already a widget class exists
(MyWidgetInstance)
{
    MyWidgetInstance->RemoveFromViewport();
    MyWidgetInstance = nullptr;
}
//Load the class of widget
if (UClass* MyWidgetClass = LoadClass<UMyUserWidget>(NULL,
TEXT("WidgetBlueprint'/Game/NewWidgetBlueprint.NewWidgetBlueprint_C'")))
{
    //Get user controller
    if (APlayerController* PC = GetWorld()->GetFirstPlayerController())
    {
        //Create a widget object
        MyWidgetInstance = CreateWidget<UMyUserWidget>(PC, MyWidgetClass);
        if (MyWidgetInstance)
        {
            MyWidgetInstance->AddToViewport();
        }
    }
    UE_LOG(LogTemp, Warning, TEXT("Added"));
}
```

Code 16 Declaration of widget object (IPGameModeBase.cpp)

In widget class, we declare a L-system turtle pointer. When the widget class is instantiated, we assign the reference of L-system turtle object to this pointer. So that we can transmit all input data to L-system turtle.

```

    Lsystem->p->lengthFactor[0] = FString::Atod(*lengthFactor->GetText().ToString());
    Lsystem->p->lengthFactor[1] = FString::Atod(*lengthFactor_1->GetText().ToString());
    Lsystem->p->lengthFactor[2] = FString::Atod(*lengthFactor_2->GetText().ToString());
    Lsystem->p->lengthFactor[3] = FString::Atod(*lengthFactor_3->GetText().ToString());
    Lsystem->p->lengthFactor[4] = FString::Atod(*lengthFactor_4->GetText().ToString());
    UE_LOG(LogTemp, Warning, TEXT("lengthFactor: %f,%f,%f,%f,%f"), Lsystem->p->lengthFactor[0], Lsystem->p->lengthFactor[1], Lsystem->p->lengthFactor[2], Lsystem->p->lengthFactor[3], Lsystem->p->lengthFactor[4]);

    Lsystem->p->GenerateSentence();
    Lsystem->render();

```

Code 17 Example of data transmit and generating a tree (MyUserWidget.cpp)

```

    Lsystem = GWorld->GetWorld()->SpawnActor<ALsystemTurtleActor>(Lsystem->StaticClass(), FVector(0, 0, 70), FRotator(0, 0, 0));

    MyWidgetInstance->loadTurtle(Lsystem);

```

Code 18 Create and assign L-system turtle object (IPGameModeBase.cpp)

4.6 Game mode

Game mode is like the Main function in Java. It is the entry point of the program in Unreal Engine 4 project. In this project, the instantiation of the L-system turtle class and the widget class happen within the BeginPlay() function of Game mode class. BeginPlay() function will be run when the program starts.

```

if (MyWidgetInstance)
{
    MyWidgetInstance->RemoveFromViewport();
    MyWidgetInstance = nullptr;
}
if (UClass* MyWidgetClass = LoadClass<UMyUserWidget>(NULL,
TEXT("WidgetBlueprint'/Game/NewWidgetBlueprint.NewWidgetBlueprint_C'")))
{
    if (APlayerController* PC = GetWorld()->GetFirstPlayerController())
    {
        MyWidgetInstance = CreateWidget<UMyUserWidget>(PC, MyWidgetClass);
        if (MyWidgetInstance)
        {
            MyWidgetInstance->AddToViewport();
        }
    }
}
Lsystem = GWorld->GetWorld()->SpawnActor<ALsystemTurtleActor>(Lsystem->StaticClass(), FVector(0, 0, 70), FRotator(0, 0, 0));
MyWidgetInstance->loadTurtle(Lsystem);

```

Code 19 BeginPlay() in Game mode (IPGameModeBase.cpp)

4.7 Important problems during implementation

4.7.1 Rough and bumpy surface of mesh component

In the early stage of development, the mesh of branch is made of a standard cylinder and a standard circle. It will come up with a problem that the surface of branch is too smooth to look natural.



Figure 33 Previous mesh of branch

To solve this problem, realistic models for branch and trunk are needed. After searching a model library called Quixel Bridge, a better model for tree trunk has been found. As we can see in the demonstration figures, the surface of this trunk is rough and bumpy. At the same time, the upper part of the trunk is close to a cylinder. Therefore, we can crop the upper part of the trunk and take it as the model for branch.



Figure 34 Tree trunk



Figure 35 Branch

What needs to be mentioned is that the material for those meshes are different than the material we use on the mesh component. The cropping process is done on 3DS MAX 2019

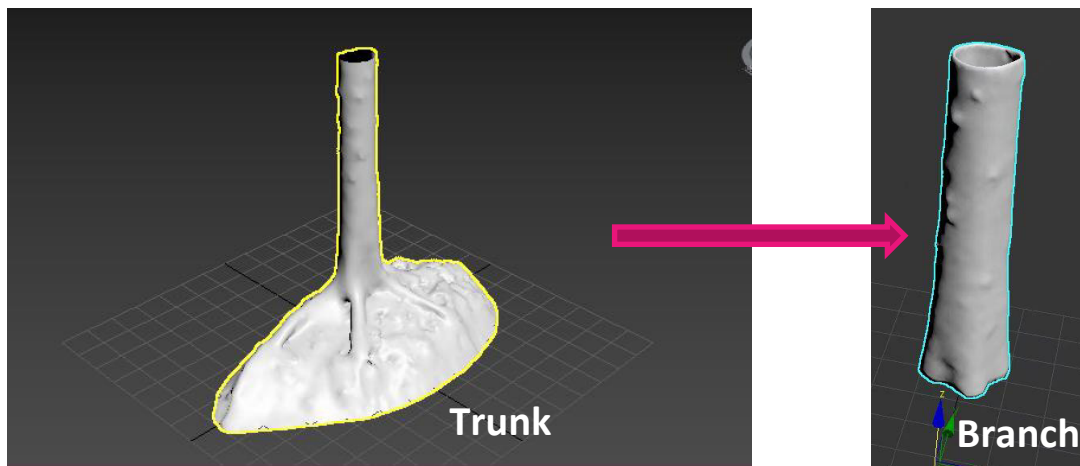


Figure 36 Cropping process on 3DS MAX 2019

4.7.2 Performance improvement

One significant issue that affects the performance is lighting in the scene. There are two of the lighting types in Unreal Engine 4 that are needed to introduce.

(1) Stationary lights [13]

Stationary Lights are lights that are intended to stay in one position, but are able to change in other ways, such as their brightness and color. This is the primary way in which they differ from Static Lights, which cannot change in any way during gameplay. However, it should be noted that runtime changes to brightness only affect the direct lighting. Indirect (bounced) lighting, since it is pre-calculated by Lightmass, will not change.

Of the three light mobilities, Stationary lights tend to have the highest quality, medium mutability, and medium performance cost.

(2) Static light [14]

Static Lights are lights that cannot be changed or moved in any way at runtime. They are calculated only within Lightmaps, and once processed, have no further impact on performance. Movable objects cannot integrate with static lights, so the usefulness of static lights is limited.

Of the three light mobilities, Static lights tend to have medium quality, lowest mutability, and the lowest performance cost.

By default, the light in a map will be a stationary light. It causes significant impact on performance because in this project, tens of thousands of static mesh actors are created. It means that the system must calculate real-time lighting for all those actors. The frame rate in inspecting window is very low.

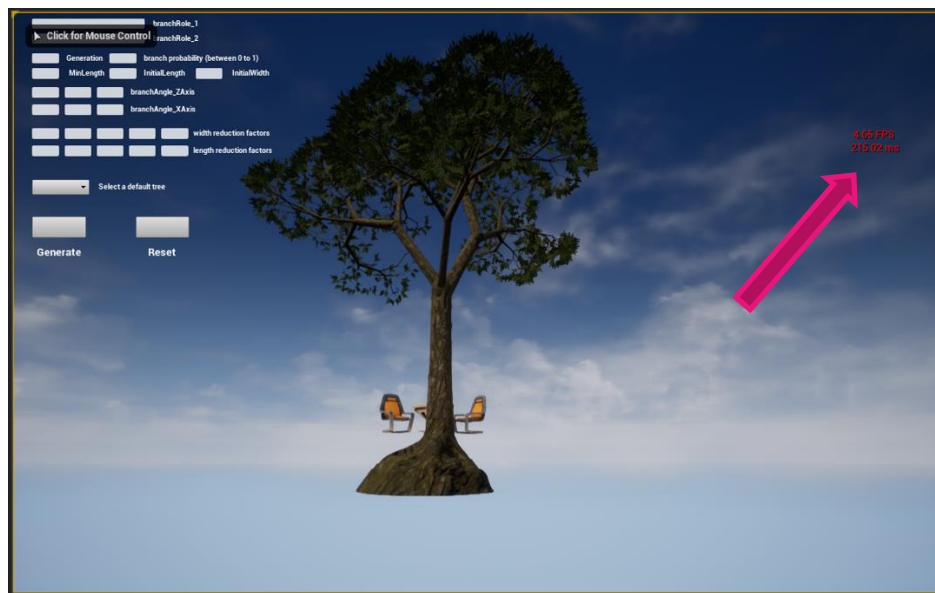


Figure 37 Frame rate testing before improving

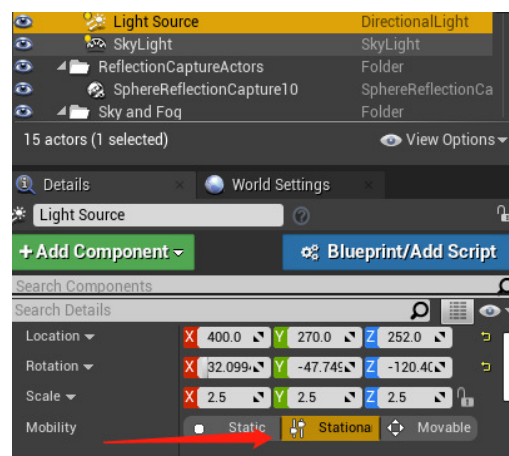


Figure 38 Default lighting setting in a map

By changing the light into static, the performance of program has improved. The frame rate has increased significantly. However, the quality of light is also lower.

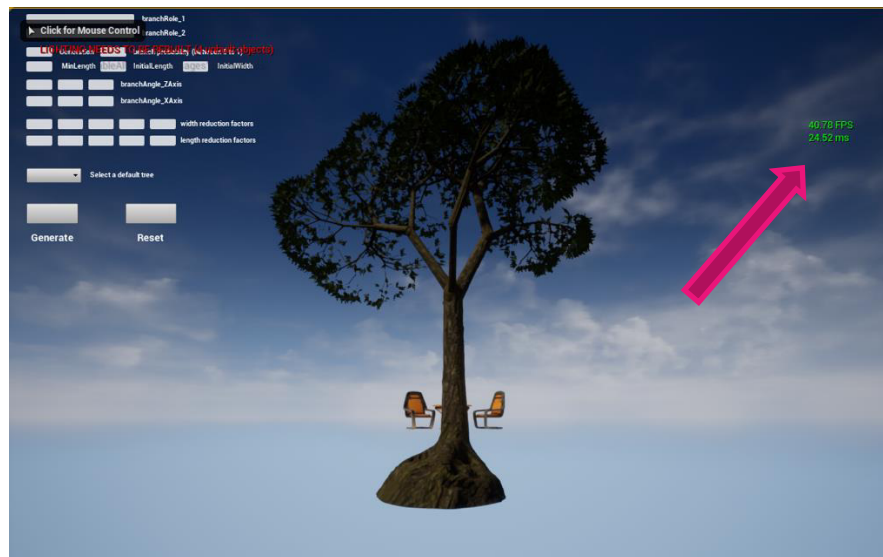


Figure 39 Frame rate test after improving

5.Results and Evaluation

5.1 User interface

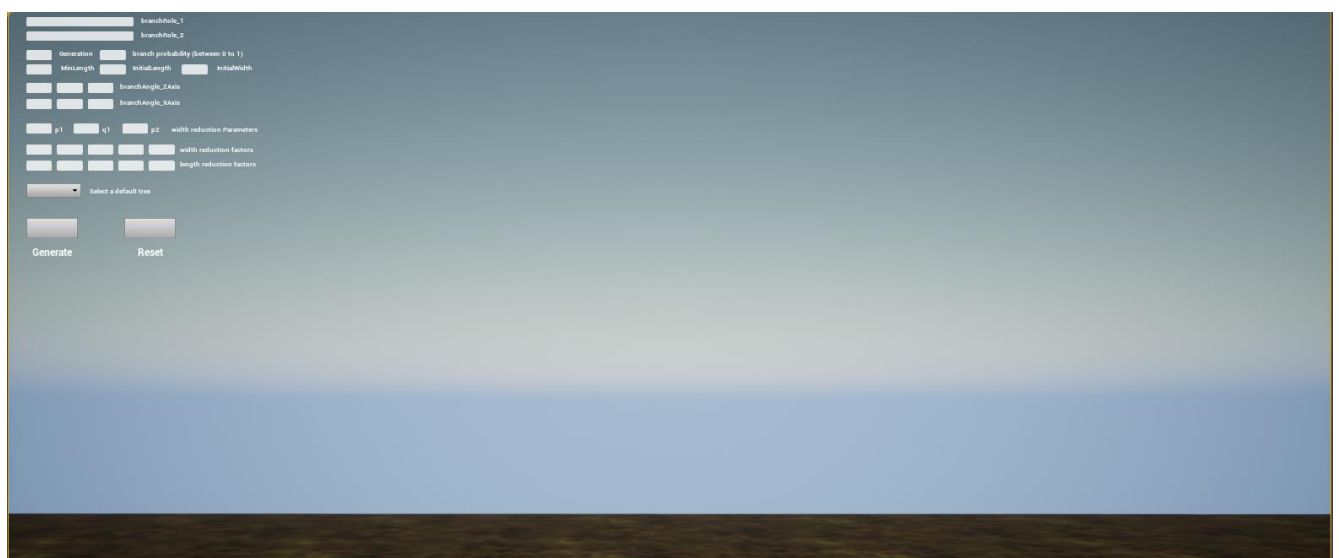


Figure 40 User interface

5.1 Effect of each parameter

Trees are controlled by parameters that are mentioned above. Therefore, changing in each parameter of one type of tree is necessary.

Basic parameter of a type of tree:

Axiom $\omega = B(300,70)$, Probability $\pi = \{0.6,0.4\}$, generation = 9, initial length = 300,

Initial width = 90,

Branch angle around Z axis $\{-30^\circ, 30^\circ, 90^\circ\}$, Branch angle around X axis $\{-30^\circ, 35^\circ, -90^\circ\}$

Width parameters: $p_1 = 0.5$ $q_1 = 0.3$ $p_2 = 0.3$

Length parameters: $\{0.68, 0.78, 0.58, 0.68, 0.78\}$

Rewriting rules: $\{B \rightarrow !F[+/B][+/B][+/B]\}, \{B \rightarrow !F[+/B][+/B]\}$



Figure 41 Same type of tree with different generation (7,9)



Figure 42 Same type of tree with different triple branching probability (0.4,0.6)



Figure 43 Same type of tree with different initial length (300,400)



Figure 44 Same type of tree with different initial width (70,80)



Figure 45 Same type of tree with different rotation angles around z-axis (-45,45,105) (-30,30,90)



Figure 46 Same type of tree with different rotation angles around x-axis (-30,35, -40) (-20,30, -35)

5.2 Different types of trees

Axiom $\omega = B(300,60)$, Probability $\pi = \{0.8,0.2\}$, generation = 8, initial length = 300,

Initial width = 60,

Branch angle around Z axis $\{-30^\circ, 30^\circ, 90^\circ\}$, Branch angle around X axis $\{27^\circ, -68^\circ, 60^\circ\}$

Width parameters: $p_1 = 0.5$ $q_1 = 0.3$ $p_2 = 0.3$

Length parameters: $\{0.65, 0.71, 0.55, 0.65, 0.71\}$

Rewriting rules: $\{B \rightarrow !F[+/B] [+/B][+/B]\}, \{B \rightarrow !F[+/B] [+/B]\}$



Figure 47 Sympodial tree _1

Axiom $\omega = B(200,50)$, Probability $\pi = \{0.6,0.4\}$, generation = 9, initial length = 200,

Initial width = 50,

Branch angle around Z axis $\{-30^\circ, 30^\circ, 90^\circ\}$, Branch angle around X axis $\{25^\circ, -25^\circ, 60^\circ\}$

Width parameters: $p_1 = 0.45$ $q_1 = 0.3$ $p_2 = 0.5$

Length parameters: $\{0.5, 0.85, 0.55, 0.5, 0.85\}$

Rewriting rules: $\{B \rightarrow !F[+/B][+/B][+/B]\}, \{B \rightarrow !F[+/B][+/B]\}$



Figure 48 Sympodial tree _2

Axiom $\omega = B(300,70)$, Probability $\pi = \{0.7,0.3\}$, generation = 9, initial length = 300,

Initial width = 70,

Branch angle around Z axis $\{-30^\circ, 30^\circ, 90^\circ\}$, Branch angle around X axis $\{-20^\circ, 35^\circ, -40^\circ\}$

Width parameters: $p_1 = 0.5$ $q_1 = 0.3$ $p_2 = 0.3$

Length parameters: $\{0.65, 0.71, 0.55, 0.65, 0.71\}$

Rewriting rules: $\{B \rightarrow !F[+/B][+/B][+/B]\}, \{B \rightarrow !F[+/B][+/B]\}$



Figure 49 Sympodial tree _3

Axiom $\omega = B(120,50)$, Probability $\pi = \{0.7,0.3\}$, generation = 9, initial length = 120,

Initial width = 50,

Branch angle around Z axis $\{-90^\circ, 90^\circ, -5^\circ\}$, Branch angle around X axis $\{5^\circ, -30^\circ, -40^\circ\}$

Width parameters: $p_1 = 0.5$ $q_1 = 0.3$ $p_2 = 0.6$

Length parameters: $\{0.95, 0.75, 0.68, 0.95, 0.75\}$

Rewriting rules: $\{B \rightarrow !F[+/B][+/B][+/B]\}, \{B \rightarrow !F[+/B][+/B]\}$



Figure 50 Sympodial tree _4

Axiom $\omega = B(100,20)$, Probability $\pi = \{0.6,0.4\}$, generation = 9, initial length = 120,

Initial width = 20,

Branch angle around Z axis $\{137^\circ, 137^\circ, -5^\circ\}$, Branch angle around X axis $\{-5^\circ, 25^\circ, -40^\circ\}$

Width parameters: $p_1 = 0.4$ $q_1 = 0.3$ $p_2 = 0.4$

Length parameters: $\{0.95, 0.75, 0.68, 0.55, 0.95\}$

Rewriting rules: $\{B \rightarrow !F[+/B] [+/B]\}, \{B \rightarrow !F[+/B] [+/B]\}$



Figure 51 Sympodial tree _5

5.3 Observation from different perspective



Figure 52 Observe the tree from the right

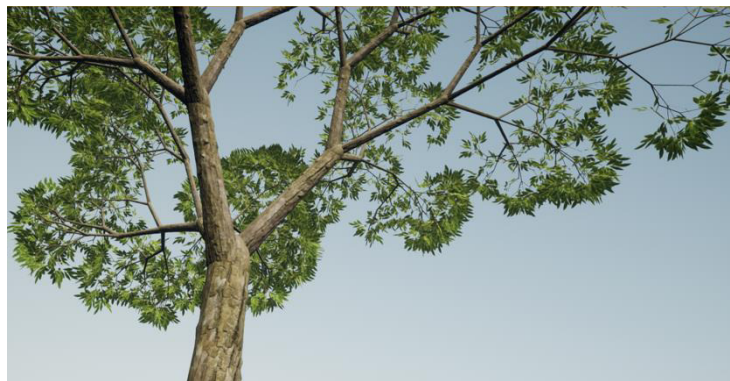


Figure 53 Observe the tree from the bottom



Figure 54 Observe the tree from the top

6. Future work

Due to time constraints during this project, the program did not implement the export 3D model function. In this project, trees are formed by tens of thousands of static mesh components, and they are independent from each other. To export the tree, we need to combine all components as one static mesh actor. However, there is not a simple way to automatically combine actors when the programming is running. A potential solution is to extract the mesh data from the lower level of render pipeline and combine those data, which requires longer time to study.

As mentioned in section 4.7.1, one important feature that makes tree look realistic is the surface of trunk and branch. In nature, the surface of a tree is bumpy and rough. Although in this project, this problem can be solved by using a better 3D model, the branch and trunk of the tree will be completely identical. Because they are using a fixed 3D model. This can be potentially fixed by using 'bump map' and apply random bump map onto the surface of branches and trunk.

As described in initial plan, there are more than one type of tree in nature. A potential future work can be studying and implementing other types of trees with different rewriting rules. For example, monopodial tree.

Another area for potential improvement is that the branches of a tree are all straight. In nature, branches are usually curving. Although this can be solved by using B Spline [16]. B Spline curve line simulation is broadly used in many areas, including plants modeling. In this project, B Spline can be used to simulate a curved branch by combining several straight branches with certain angles. However, the theory of B Spline is complicated, and it requires more time to understand and to be implemented in practice.

7. Conclusions

The primary aim of this project is to research and implement L-system on Unreal Engine 4, researching plants morphology and generate a realistic plant. The L-system was successfully implemented in C++ and a system based on C++ has been designed and it has been successfully run on Unreal Engine 4. The results of the evaluation have shown that a realistic 3D tree can be generated. User can change the shape of it by modifying the parameters. Although the parameters that are extracted from plants cannot perfectly simulate and describe the plants perfectly, it has left a research direction for the future work.

The secondary aim was to develop a system with a user interface that allows user to adjust the parameters directly and import customized 3D model and material. An export functionality is also expected, it provides a way for user to export the tree that has been generated as an FBX. File for future use. The secondary aim has partly achieved. A user interface has been implemented and user can change tree's parameter. However, the export and import functionality are missing. As discussed briefly in Section 7, this is potential for improvement.

8. Reflection on Learning

The project has achieved its practical aims with deliverable results. In addition to it, I have also benefit from completing the project. By designing and developing L-system on Unreal Engine 4 by C++, I have expanded my programming skills and the familiarity of Unreal Engine 4 platform, as well as computer graphics. I am deeply interested in them, and I hope to keep researching in field of computer graphics and game engineering. This project has also provided an opportunity to explore mathematical plants simulations. Although I cannot invent a simulation method, it is also very helpful to implement an existing method.

The project has also been very helpful to my time management skills and carrying out a project from start to finish. By completing the project, I have had the opportunity to research and to learn how to use Unreal Engine 4, which is my intended future area to work in.

9. References

- [1] En.wikipedia.org. 2022. *Woody plant* - *Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Woody_plant> [Accessed 12 May 2022].
- [2] Prusinkiewicz, P. and Lindenmayer, A., n.d. The algorithmic beauty of plants.
- [3] Prusinkiewicz P. Graphical applications of L-systems[C]. Proceedings of Graphics Interface. 1986,86:247-253
- [4] Hanan J S. Parametric L-systems and their application to the modeling and visualization of plants[D]. Regina: University of Regina, 1992
- [5] Abelson H, diSessa A A. Turtle geometry[M]. Cambridge:M.L.T. Press, 1982.
- [6] J. D. Foley and A. Van Dam. Fundamentals of interactive computer graphics. Addison-Wesley, Reading, Massachusetts, 1982.
- [7] Honda, H., 1971. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. Journal of Theoretical Biology, 31(2), pp.331-338.
- [8] Docs.unrealengine.com. 2022. *Actors*. [online] Available at: <[https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/#:~:text=An%20Actor%20is%20any%20object,code%20\(C%2B%2B%20or%20Blueprints\).](https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/#:~:text=An%20Actor%20is%20any%20object,code%20(C%2B%2B%20or%20Blueprints).>)> [Accessed 12 May 2022].
- [9] Docs.unrealengine.com. 2022. *Textures*. [online] Available at: <<https://docs.unrealengine.com/5.0/en-US/textures-in-unreal-engine/>> [Accessed 12 May 2022].
- [10] Docs.unrealengine.com. 2022. *Game Mode and Game State*. [online] Available at: <<https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/GameMode/>> [Accessed 12 May 2022].
- [11] Eloy, C., 2011. Leonardo's Rule, Self-Similarity, and Wind-Induced Stresses in Trees. Physical Review Letters, 107(25).
- [12] 2022. [online] Available at: <<https://www.wired.com/2011/11/branching-tree-physics/#:~:text=Leonardo's%20rule%20holds%20true%20for,cross%20section%20of%20the%20trunk.>>> [Accessed 12 May 2022].
- [13] Docs.unrealengine.com. 2022. *Stationary Lights*. [online] Available at: <<https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/LightMobility/StationaryLights/>> [Accessed 12 May 2022].
- [14] Docs.unrealengine.com. 2022. *Static Lights*. [online] Available at: <<https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/LightMobility/StaticLights/>> [Accessed 12 May 2022].

[15] de Boor, C., 1972. On calculating with B-splines. *Journal of Approximation Theory*, 6(1), pp.50-62.