Plant Growth Visualiser in Unreal Engine 4

MSc Computer Animation and Visual Effects—Master's Project

Ollie Nicholls

22/08/2021

Abstract

When trying to recreate realistic ecosystems for video games and films, artists must consider all complexities and intricacies of the biological phenomena present in real-world ecosystems. Moreover, as the VFX industry moves towards using real-time game engines in place of the traditional pipeline, having tools available in these engines is of utmost importance. In this paper, the process for implementing a plant visualisation tool, that forms the groundwork of a forest ecosystem generator, is demonstrated in the popular game engine Unreal Engine 4.

Contents

1 Introduction

The tasks associated with the generation of artificial terrain vary from project to project, with some projects requiring massive mountainscapes (e.g. The Mandalorian), and others luscious alien forests (e.g. Avatar), and for each of these requirements, there are an ever-increasing number of approaches and solutions. Some approaches take 3D photoscans of real-world objects and, using combinations of these, artists flesh-out entire environments procedurally and by hand, resulting in realistic virtual sets. Other approaches use nature-inspired algorithms, such as thermal erosion or plate tectonics, to generate entire terrains from a few input parameters.

As these terrains are important for setting the scene and providing immersion to the viewer, lots of time and effort is spent trying to get them picture-perfect. Often times, the tools available are complicated to use and therefore require the expertise of technical artists to configure and set up.

In recent years, there has been a seismic move toward using real-time rendering engines (traditionally used for video games) in the film industry. There are many areas where they are used such as virtual production, virtual reality (VR), augmented reality (AR), and previsualisation. Currently, the main engine being used is Epic Games' Unreal Engine 4 (UE4). UE4 allows artists to quickly develop and visualise intricate sets, all in real-time, vastly improving the efficiency of taking a concept all the way to final pixel, when compared to traditional VFX pipelines.

At present, the number of artist-friendly tools that can create realistic, real-time forests is scarce, therefore, this projects seeks to fill this space by providing a plugin within the powerful and intuitive UE4. However, due to the complexity of creating such a tool, a prototype version that visualises a single plant's growth has been created, providing foundations that could one day be used to create an extensive, real-time ecosystem generator.

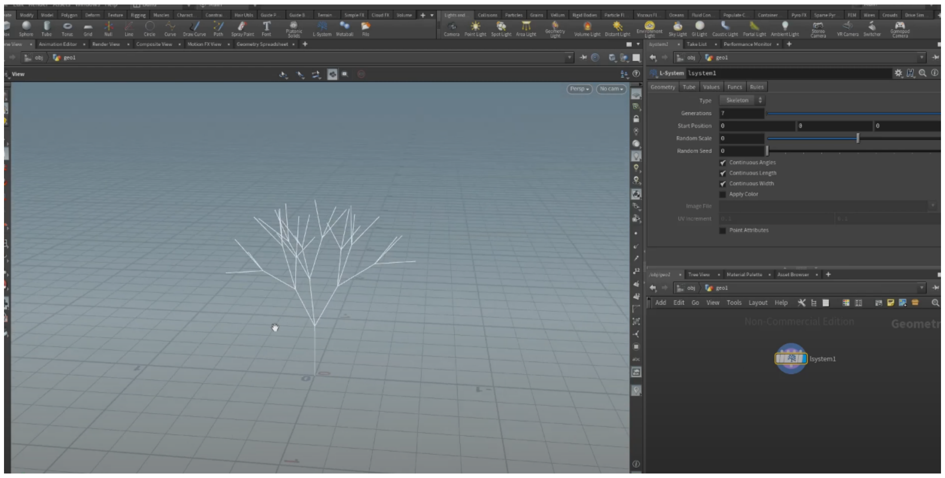
2 Previous Work

To generate realistic forest ecosystems, consideration must be taken of the many biological phenomena that are present in nature; in real forests, labyrinths emerge in the tree cover due to senescence (as a plant ages it will release fewer seeds but will block non-shade-tolerant plants due to its size, leading to gaps around old trees). When computer graphics researchers approach these phenomena, lots of research into the specific morphology of the organisms they wish to represent is required, as the complex branching structures of these organisms are hard to replicate manually.

This area of research is well explored and, even though some novel approaches are sometimes used, most researchers will initially start by analysing how effective rule-based languages can be for generating complex branching structures. One such language is the well-known Lindenmayer System or L-System, Lindenmyer (1968).

2.1 L-Systems

Named after Astrid Lindenmayer, L-systems are a rule-based mathematical language that defines complex branching structures using a few basic rules and iterations. In L-systems, strings of characters are matched against specific rules for how the branch should be developed, which are used to generate the underlying graph or geometry of the organism.



L-System lsystem1
Type: Skeleton
Generations: 7
Start Position: 0 0 0
Random Scale: 0
Random Seed: 0
✓ Continuous Angles
✓ Continuous Length
✓ Continuous Width
Apply Color
Image File
UV Increment: 0.1 0.1
Point Attributes

Figure 1: L-system implementation in SideFX's Houdini.

L-systems can be used for many applications, ranging from simulating lightning, to the telescoping of wheat stems, and the complex fractal patterns of unique snowflakes, SideFX (2020).

In artificial forest generation, L-systems can be used to represent the self-similarities within each organism, allowing for natural—but extremely complex—organic forms.

One issue that researchers face when using L-systems for plant modelling is that they do not allow for external factors such as collisions (both self and external) or any other inter- and intra-organism interactions. L-systems can also be quite complex to design, and when used in VFX, many artists may find the system hard to use as it is very different from traditional 3D modelling.

2.2 Advanced L-Systems and Beyond

As it is important to model inter-organism interactions in artificial ecosystems, L-systems have to be modified or alternative methods need to be used to facilitate this requirement. In Lane et al. (2002) for example, L-systems are used along with a "plant competition algorithm" that handles and resolves any collisions between plants, leading to complex but natural emergent behaviours and patterns.

One alternative to using L-systems is using a voxel-based system derived from cellular automata. In Greene (1989), a stochastic modelling algorithm is applied to voxel data, and using the competition for light and space as environmental parameters, organisms are developed from prototype geometric elements according to rules based on collisions and proximity.

As these systems get more complex, and as they get better at modelling the complex biological intricacies of different ecosystems, unfortunately, they also become far slower at generating the resulting ecosystem.

2.3 Faster Ecosystem Modelling

As indicated before, it is important to find a balance between a sufficient level of complexity and intricacy, and the efficiency of the algorithm(s) adopted.

One approach, discussed in Beneš et al. (2009), combines simple biology-based algorithms with stochastic plant colonisation algorithms, and using these, simulates the ecosystem at a sufficient level of detail (LOD) where a balance is found between the quality and speed of the simulation.

Another approach could be to use specialised rendering techniques to reduce the number of polygon and draw calls sent to the GPU.

In Argudo et al. (2016), billboard techniques are exploited by using an "image generation algorithm" that is capable of generating a realistic single-tree model from only one input photograph. This high-quality image is then projected over a simplistic geometry, meaning the entire forest is much faster to generate and render. However, this approach is not without issues: once the camera is too close to these models, the illusion fails, making this system only applicable to background assets.

2.4 Interactive Ecosystem Modelling

A balance has to be found between complexity and speed if an algorithm is to be successful. One way to do this could be if a system computes an offline render of the generated ecosystem. This can provide both speed and real-time feedback—by using basic geometry in the editor—and allows for the complex intricacies of real ecosystems.

One such approach that does this is Makowski et al. (2019). Here Makowski et al. provide both an interactive system that allows artists to modify parameters and specific organism building blocks, as well as an offline renderer that outputs the final ecosystem. As this provides artists with both real-time feedback and an ecosystem that contains many complex biological phenomena, this approach is the current state-of-the-art artificial ecosystem generator. Therefore, it provides the groundwork for the implementation of this project.

A more detailed description of the technical side of this approach is discussed in Section 3.1.

3 Technical Background

3.1 Synthetic Silviculture: Multi-scale Modelling of Plant Ecosystems

For the SIGGRAPH 2019, Makowski et al. developed a multiscale approach to realistic forest ecosystems, providing a framework that could model the nuances of complex plant architecture.

The main concept exploited is the self-similarity of branching structures in plants, wherein each plant is split into architectural units called 'branch modules' (BM). Each BM grows depending on external and internal parameters that results in complex and realistic geometries with uncanny resemblance to biological phenomena seen in nature.

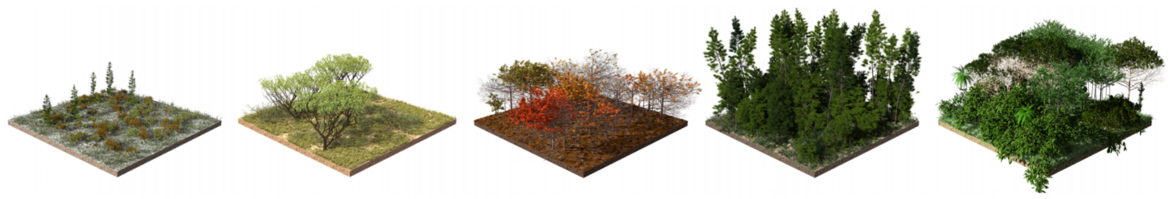


Figure 2: Five example outputs from the Synthetic Silviculture algorithm.

The algorithm in Makowski et al. (2019) is capable of modelling all biomes found on earth: desert, tundra, savanna, grassland, shrubland, boreal forest, temperate seasonal rainforest, and tropical seasonal rainforest, and, as shown in their examples, can be applied to massive landscapes with hundreds of plants.

Based on their research, Makowski et al. identified seven parameters that affect each plant's growth that they would model in their algorithm:

Competition between architectural units for light

Apical control where apical buds inhibit lateral buds

Gravitropism and phototropism how the plant is affected by gravity and the sun

Determinacy where buds develop into flowers preventing further growth

Climatic adaption how well the plant can adapt to different temperatures and precipitation

Shade tolerance how tolerant to reduced sunlight the plant is

Seeding strategy which method the plant employs for seed dispersal, e.g. wind, water, or animals

Each architectural unit or BM is represented as a directed acyclic graph (DAG) with each node representing a connection in the branch, and each edge representing a section of the branch between two connections. Each BM has one root node and at least one terminal node—which serves as a connecting point for subsequent BMs to be connected using their root node.

Each node has a set of values associated with it which describe how to generate the surface mesh of the plant: the position, physiological age, branch length, and thickening factor.

These BMs are then assembled themselves as another DAG, representing the entire plant. Each plant has a root BM and the BMs are either attached or detached based on their light exposure. In each simulation step, the total light exposure is calculated for each BM from tip-to-root and is summed to the root of the plant. This value is then used as the vigor of the plant, and a ratio of this growth potential is applied to each BM from root-to-tip.

The multiscale part of the algorithm is split into three parts: module scale, plant scale, and ecosystem scale.

At module scale, morphological parameters are interpolated to determine the position, diameter, and length of branches. The effects of flowering and branching structure are based on a determinacy value with high values producing monopodial (single trunk) forms. Conversely, low values will produce sympodial (multi-trunk) branching patterns.

At plant scale, BMs are attached and detached to express plant development according to the biological concepts of competition for light and apical control. Here, the vigor is calculated based on the light availability using an extended Borchert and Honda model (Borchert & Honda (1984)) and the BMs' orientation is represented by three Euler angles that are chosen using a collision-based optimisation algorithm.

Finally, at ecosystem scale, plasticity parameters (based on temperature and precipitation) are used to influence BM selection and growth. Growth is also influenced by how shade-tolerant the plant is. Plant offspring can be generated using an appropriate seeding strategy, allowing for biological phenomena such as labyrinths forming due to senescence.

The intricacies of the algorithm will be discussed thoroughly in Section 4.

3.2 Framework Integration

As previously mentioned, UE4 by Epic Games will be the engine used to implement the ecosystem generation algorithm described in Section 3.1. UE4 was chosen as it provides an open-source framework for quickly creating games and virtual sets, all in real-time.

Implemented in C++, and providing extensive tutorial courses for the many impressive features, UE4 allows developers to create plugins that can be used to modify specific parts of the engine or provide extended functionality.

To encourage artists to use UE4, the unique Blueprint Visual Scripting System (colloquially referred to as 'Blueprints'), which utilises a visual node-based framework, provides a high-level programming language that can be used to implement plugins and features, without any knowledge of C++. Epic also encourage programmers to use Blueprints to quickly prototype functionality.

Epic recommend that most developers use a mix of both Blueprints and C++ when approaching any task in UE4, Epic Games (2020). Template classes can be created in C++ then extended in Blueprints, allowing programmers to create functionality, and artists to modify this by implementing a Blueprint-class derived from C++.

For this project, the underlying algorithm will be implemented in C++, using JetBrains' Rider IDE—the recommended IDE to use with UE4 as it is widely agreed that it offers better C++ support and UE4 syntax when compared to Visual Studio.

Each class is first created using the built-in UE4 class tool, then the implementation is carried out in Rider, then built (again in Rider), and finally UE4 automatically reloads the solution.

Any part of the algorithm that requires modification by the user—for example, the branch module prototypes (BMPs)—will have a base class developed in C++, and then be implemented in Blueprints allowing artists to modify the visual aspects to their liking.

SideFX's Houdini Engine also provides integration with the UE4 ecosystem, and many artists will be familiar with Houdini already. Due to this, a potential future feature could be providing the ability to edit BMPs using Houdini Engine, all within UE4.

Finally, all C++ classes will have all functions and properties exposed to the Unreal Header Tool (using `UFUNCTION()` and `UPROPERTY()` macros). This gives the end-user more flexibility to extend the algorithm in either C++ or Blueprints, opening up the plugin to a wider audience.

Creating well-formatted, commented, and extendable code are important software engineering principles and will make it easier for future researchers to build upon this plugin. Therefore these principles will be upheld throughout this implementation.

4 Implementation

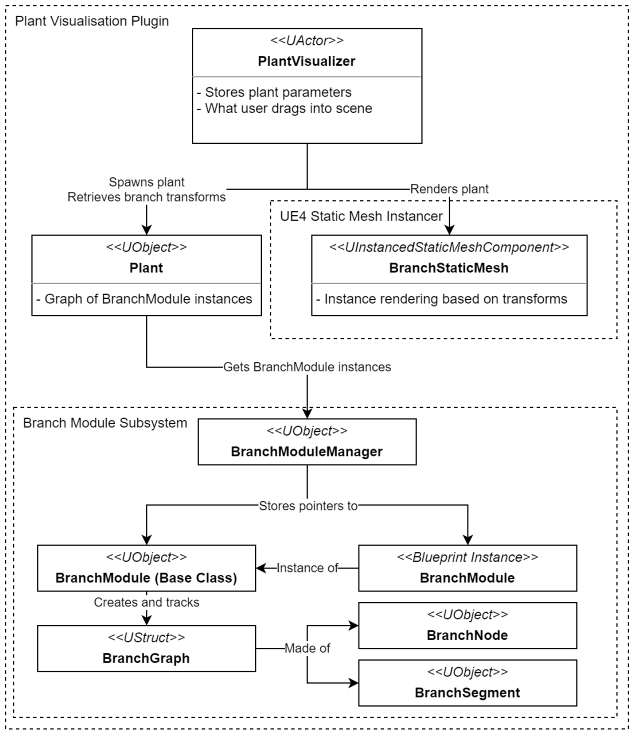


Figure 3: System architecture of this implementation.

In the following sections, a UE4 plugin is presented. This plugin implements a basic version of the method described in Makowski et al. (2019) (from now on referred to as "Synthetic Silviculture"), providing foundations that can be extended to provide an artificial ecosystem generator, all in real-time.

The plugin is mainly developed in C++, with Blueprints being used for the branch module prototypes (BMPs). All components are made with reusability and extensibility in mind, allowing further modifications and additions to be implemented with ease in the future.

Many new classes are provided by the algorithm, with the main entry point being the Plant Visualiser.

4.1 System Architecture

Initially, the entire algorithm described in Synthetic Silviculture was to be implemented, but due to time constraints, the steep learning curve associated with understanding UE4, and the complexities of the algorithm, a simplified Plant Visualiser was developed that allowed the user to visualise a single plant and experiment with different settings.

Before development, the plugin system architecture was modelled using a UML diagram (see Figure 3). For reference and future extensions to the project, the planned integrated architecture has still been included and can be seen in Appendix A.1.

The Plant Visualiser—which is responsible for creating the plant and initialising the helper components—is implemented as an actor in UE4, so that it can be dragged into the scene and setup accordingly.

Each plant—which is implemented as a `UObject`—has a reference to the BM manager that returns BMs based on certain parameters. This is also where user-defined BMPs can be created within the Blueprints system.

After each step of the simulation, the Plant Visualiser retrieves all branch structures from the plant and, using instanced static meshes, spawns instances of the branches, thus rendering the plant.

The system is intricate but can be simplified into two parts: the BM subsystem, which is the data structure that each plant is made from, and the plant, where the growth algorithm is implemented and subsequently rendered.

4.2 Branch Module Subsystem

In Synthetic Silviculture, plants are described as "an ordered tree graph of connected branch modules", where the BMs themselves are a DAG of nodes and segments. These BMs are essentially the building blocks for the entire growth algorithm and thus were implemented first. They also served as a good starting point for getting familiar and comfortable with UE4's Blueprints and C++ engine code.

The BM subsystem includes all classes and computation associated with the BM data structures. In this implementation, this includes classes to describe the nodes and segments in the graph, a structure used to store the graph, another for defining one, the BM class itself, and finally, the BM manager.

BMs are implemented as `UObject`s containing a graph (describing the branch structure, Section 4.2.1), a reference to the BM manager (required for attaching and shedding BMs, Sections 4.3.5 & 4.3.6), a bounding sphere (for calculating light exposure and vigor, Sections 4.3.1 & 4.3.2), and data describing the age of the BM.

4.2.1 Branch Module Graphs

As mentioned in the previous section, BMs are DAGs: $G = (N, E)$, made of nodes, N, and segments (traditionally called edges), E.

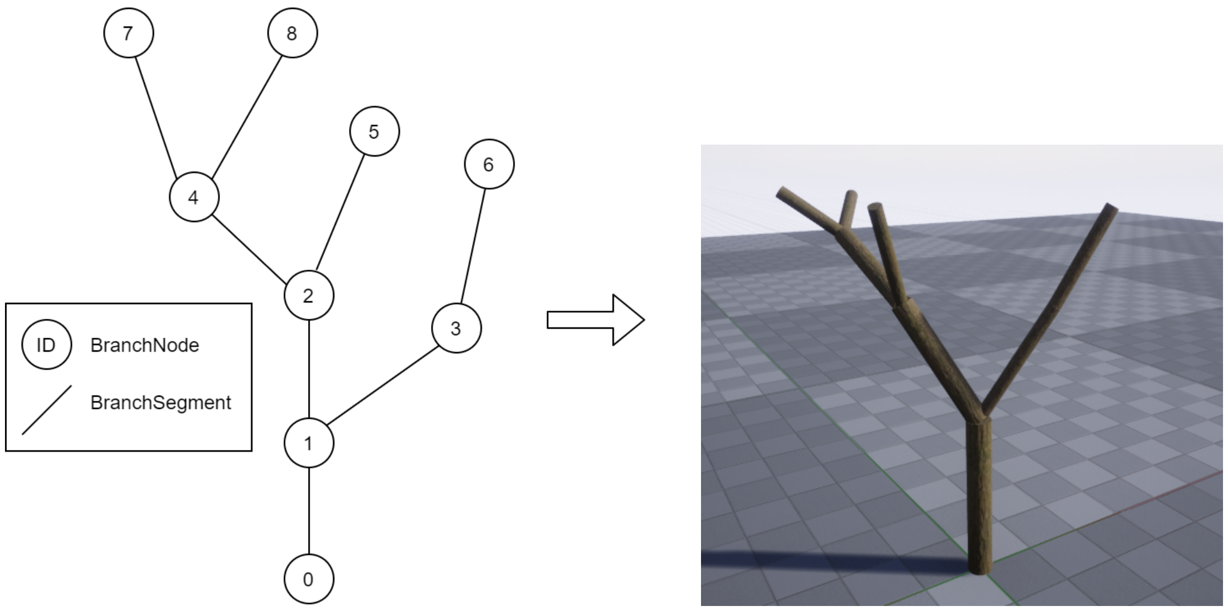


Figure 4: Visualisation of a branch module graph and how the resulting branch module is rendered.

The graph describes information about the BM branching structure (see Figure 4). The nodes store parameters such as position (using a UE4 `FVector`), physiological age, and parent and children segments. The segments describe the connections between nodes and the diameters of the branches.

The graphs are initialised with a graph definition and describe the branching structure of the fully grown BM. Parameters associated with the nodes and segments are used to represent the age of the BM, with segments being marked as either "available" or "not available" depending on the BM age (Section 4.3.3).

Most methods that can be applied to a node work in a recursive nature, wherein all the available child nodes also have the same method applied to them. For example, if $n_1, n_2 \in N$ and n_2 is a child of n_1, then calling $n_1.Translate(FVector)$ would also translate n_2.

Segments are also used to connect child BMs to the current BM. This is achieved by attaching the root node of the new BM to a terminal node (node with no children) using a connecting segment (Section 4.3.5).

4.2.2 Branch Modules in Blueprints

Due to the power and simplicity of Blueprints, the BMPs that were used as templates for the BM graphs, were implemented as Blueprint-classes based on the BM class.

These Blueprint-classes implement the method `GetGraphDefinition()` and return an `FGraphDefinition` that is used to describe how the graph is made. To make this function implementable in Blueprints, it was decorated with the `BlueprintNativeEvent` specifier.

Implementing the BMPs in this way meant that artists could easily create new ones without having to understand the code. This basic system worked but could have been improved further by providing a visualisation of the BM graph, or even providing an in-

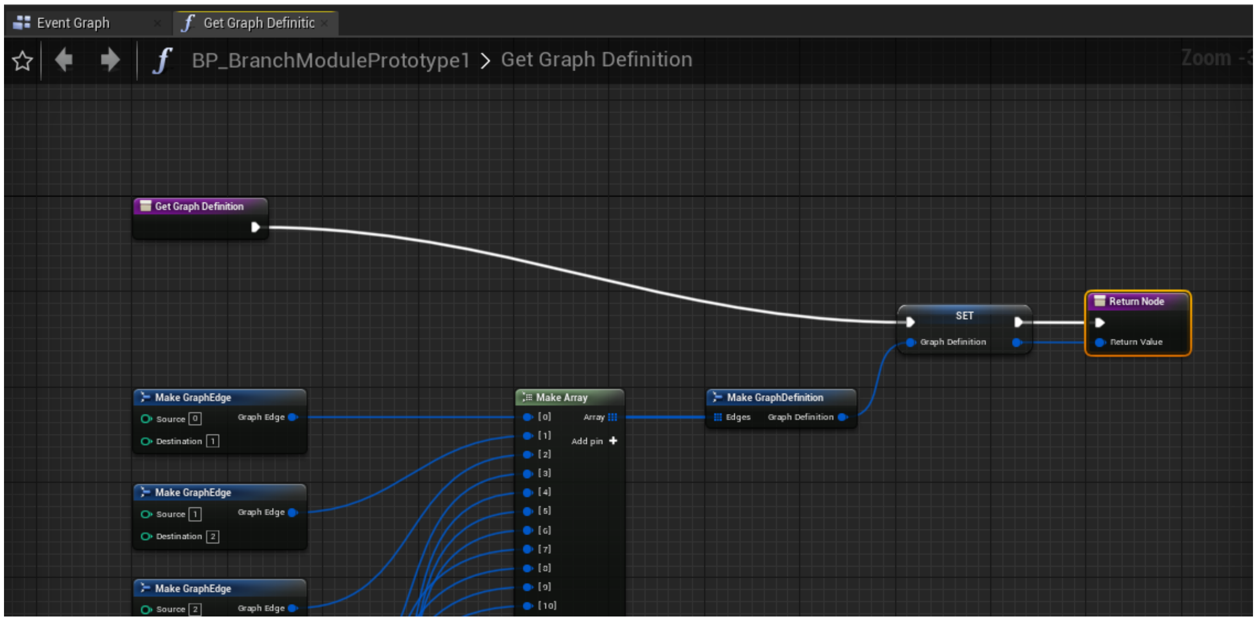


Figure 5: Branch module graph definition in a Blueprint-class.

editor graph tool.

Another future extension could be through exploiting Houdini Engine by creating a Houdini Digital Asset that allowed the artist to create a prototype using L-systems.

4.2.3 Branch Module Manager

As knowledge of other BMs is required to calculate the light exposure (Section 4.3.1), having an object that keeps track of all BMs currently in the system is vital to achieve this.

The manager is mainly responsible for storing the BMPs that are queried using the plant's apical control and determinacy values to create a new BM. For methods that needed executing on all BMs, such as calculating light exposure, the manager stored a pointer to all existing BMs and executed these methods on them.

Every plant and BM had a reference to the manager which allowed them to create and remove BMs when required during simulation. This class is vital to the plugin and provides a location for implementing functionality that is required in many different classes.

4.3 Main Plant-Growth System

The Plant Visualiser is an actor that can be placed into a scene and has settings associated with it for developing a single plant. These settings were a `UDataTable` used for storing different settings for different plant types, a `TArray` of BMPs, a static mesh that would be instanced for all branches in the plant, and finally the simulation time.

There are lots of settings associated with a single plant type. However, in Synthetic Silviculture there is no description of what some parameters are, how they are calculated, and what example values could be. A trial-and-error approach was therefore employed to select these parameters.

In further sections, these parameters will be used but not described, but the reader is encouraged to refer to Appendix A.2 for details about these parameters.

The Plant Visualiser is the main class in this implementation. On execution, it initialises the BM manager and the static mesh, it retrieves the first plant type from the data table, creates and initialises a plant object, and finally starts a timer to call the `Simulate()` method every 100 milliseconds. 100 milliseconds was chosen as the time step as it gave a pleasing animation of the plant growing when the program ran.

The `Simulate()` method performs one time step of simulation and then renders the result using the static mesh instancer. The result is rendered by firstly getting all branch definitions from the existing BMs, then calculating how to transform the input static mesh for each branch, and finally the meshes are instanced in the scene.

Using an instancer here means that drawing the entire plant is efficient and can be scaled to drawing entire forests with minimal work required from the GPU.

For each simulation step:

- the BM manager calculates the light exposure $Q(u)$ for all BMs,
- then the execution is handed to the plant where the vigor is calculated for each BM,
- each BM is aged and grown,
- new BMs are attached and orientated,
- finally, some BMs are shed.

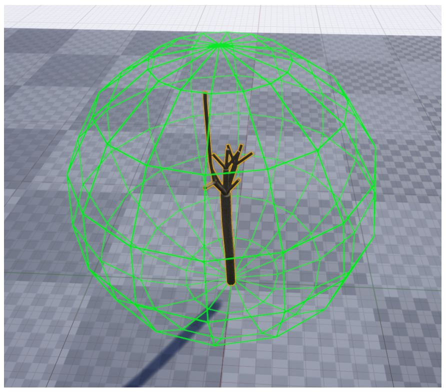
This process is repeated until either the plant dies or the simulation ends.

4.3.1 Calculating Light Exposure

To calculate the light exposure of each BM u, a spherical bounding volume B_u (see inset figure) is defined for each BM and intersected with all neighbouring BMs to find the number of collisions. This value is then used to calculate the light exposure,

$$Q(u) = exp(-f_{collisions}(u)),$$

described as an exponential decay. The method described in Synthetic Silviculture did not work correctly as the sum of intersections with B_u was high due to the large volumes of each sphere (meaning $Q(u)$ was regularly zero as $exp(-x)$ where $x > 10 \approx 0$). Instead, a ratio of how much of the BM was intersected was used as this provided better results.



Refer to Appendix A.3 for further details on how B_u and $f_{collisions}(u)$ are calculated.

Once the light exposure is calculated for each BM, the plant graph is sorted using a topological sort—as described in Muhammad (2010)—to perform a basipetal pass accumulating the light exposure at each intersection $Q(u) = Q(u_m) + Q(u_l)$ where $Q(u_m)$ is the main BM's light exposure and $Q(u_l)$ is the lateral BMs' (Figure 6). The main branch is decided based on the order of the topological sort. Finally, this value is accumulated in u_{root} where it is used to calculate the vigor.

4.3.2 Calculating Vigor

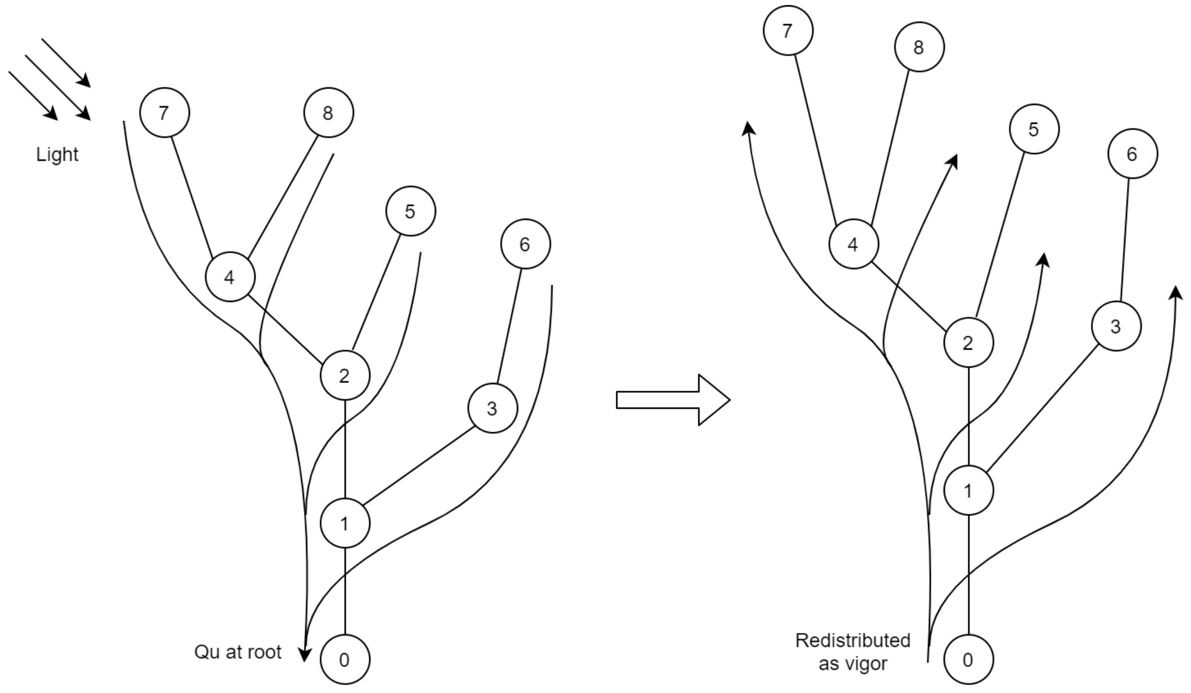


Figure 6: Light is accumulated in u_{root} in a basipetal pass and redistributed as vigor in an acropetal pass. *Note: here the graph nodes represent branch modules not branch nodes but the same method is applied in both cases.*

The vigor associated with each BM is used to drive its growth and is calculated using the value of $Q(u)$. This description in Synthetic Silviculture was vague so it was assumed that $\bar{v}(u) = min(Q(u), \bar{V}_{rootmax})$.

To redistribute vigor through the plant, the topological sorting of the BMs used in Section 4.3.1 was reversed so that vigor was distributed in an acropetal pass (Figure 6). Specifically, the root BM is first chosen and has the total vigor assigned to it. Then, for each of the children, the vigor is distributed using a similar equation used to sum the light exposure: $\bar{v}(u_l) = \bar{v}(u) - \bar{v}(u_m)$ where $\bar{v}(u_m)$ is calculated using the following weighted function:

$$\bar{v}(u_m) = \bar{v}(u) \cdot \frac{\lambda Q(u_m)}{\lambda Q(u_m) + (1-\lambda)Q(u_l)},$$

using $Q(u_m)$ and $Q(u_l)$ calculated in the previous section. All lateral BMs were assigned $\bar{v}(u_l)$ with the main BM receiving $\bar{v}(u_m)$, which was assumed to be the approach taken in Synthetic Silviculture as cases with more than two child BMs were not mentioned.

Once a vigor value had been assigned to all BMs, this value was used for calculating a growth rate that could be used for ageing and growth.

One final note was that once the plant's physiological age was greater than the max age ($p_t \geq p_{max}$), $\bar{v}_{rootmax}$ is linearly interpolated to zero. In this implementation, this was achieved by using `FMath::LerpStable(`$0, \bar{v}_{rootmax}, \alpha_{lerp}$`)`, where $\alpha_{lerp} = (p_t - p_{max})/p_{max}$ which was not described in Synthetic Silviculture but was a reasonable method to use.

4.3.3 Ageing Modules

After the vigor has been calculated, all BMs are iterated through in basipetal order, determining the growth rate of each and making structural changes based on this.

The growth rate of each BM is calculated using a smoothly interpolated sigmoid function $S(x) = 3x^2 - 2x^3$ that provides a slow growth rate to young and old BMs but a fast growth rate to the rest, taking influence from processes seen in nature. The value of x is the ratio of vigor to the minimum and maximum vigor and is multiplied by the growth potential:

$$\Upsilon(u) = S(\frac{\bar{v}(u) - \bar{v}_{min}}{\bar{v}_{max} - \bar{v}_{min}}) \cdot g_p.$$

It can be derived from this equation that plants with higher vigor will have a higher growth rate up to a maximum of g_p and that the vigor a BM gets is dependent on λ in the previous section, meaning if λ is large and the BM is a lateral branch it will not grow as quickly as the main branch. Before this calculation, $\bar{v}(u)$ is clamped to $\bar{v}_{max}$ to avoid excessive growth rates. The growth rate is then used to calculate the change in age of the BM over this time step:

$$da_u = \Upsilon(u) \cdot dt,$$

which is subsequently used in growth and spawning new nodes in the BM graph. Initially, $a_u = 0$ and is increased using this equation. Growing each BM is described further in Section 4.3.4. All current nodes are aged by this amount.

When it came to growing the BM graph to include new nodes, Synthetic Silviculture only showed a single diagram that was not useful. Moreover, the video presentation only mentioned it briefly with an animation, but again this did not give much information on the process they used. Therefore, this section of the program is entirely new compared to the original algorithm.

When initialising the BM graph, all nodes and segments are added and a depth is set to each segment specifying how far they are from the root node (see Figure 7).

All segments are then set to "not available" signifying they are not included in the graph yet. When a BM is spawned, all segments between the root node and its children are set to "available" and the age of the graph is set to zero. The depth is then used when ageing the BM by comparing the depth of each segment with the age of the BM, and if $depth \leq a_u$, the segment is set to "available", and the age of the child node a_n is initialised as $a_n = a_u - depth$, meaning the age of each node will always be correct even if the time step was large and essentially 'skipped' over the correct spawn time of this node.

After this, the child nodes need to be spawned—setting their initial position and direction compared to the parent node. To do this, the parent position p_p (`FVector`) and parent

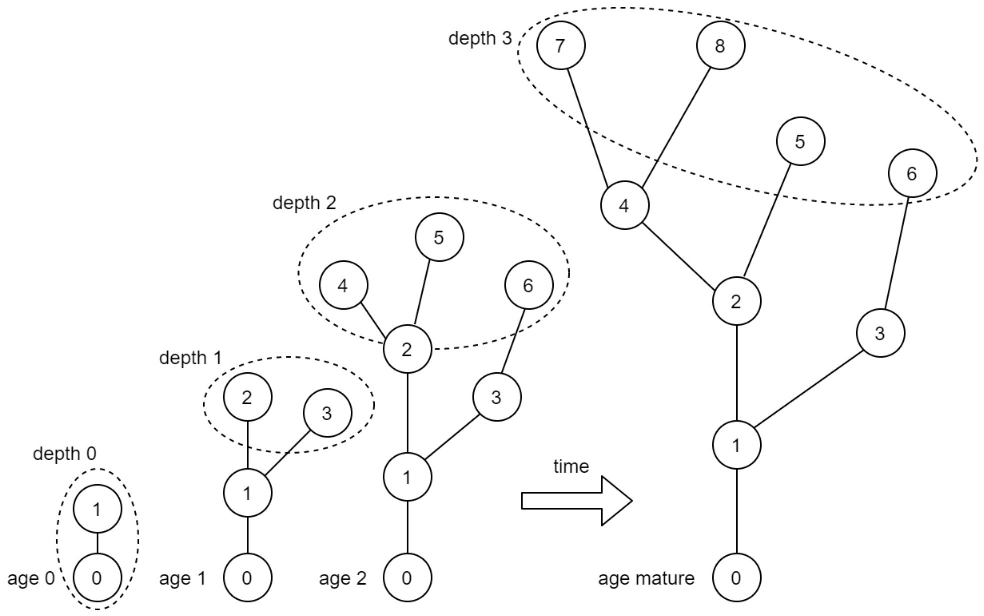


Figure 7: Different ages of a branch module are calculated by interpolating the graph using depth.

orientation r_p (`FRotator`) are calculated, then depending on the number of child nodes, each child is spawned with a certain direction from the parent.

The algorithm for positioning the children worked on any number of children from one to the maximum child allowance of five, and involved using `FMath::RandRange()` to introduce some randomness to the spawning.

Essentially, if there was an odd number of children, the first child was popped and spawned in the same direction as the parent, then if there were four children left, the next two were popped and spawned north and south of the parent, the final children were then spawned east and west of the parent. See Appendix A.4 for the code.

This part of the program worked very well and gave good visual fidelity and uniqueness to each BM that was spawned.

4.3.4 Growing Modules

Once the appropriate parts of a BM have been aged, the branch growth is calculated by assigning an age to each branch segment.

In Synthetic Silviculture, the age of each branch segment was defined as $a_b = max(0, a_u - a_n)$, where a_n is the age of the oldest node in the segment. However, when it came to implementing this, it did not make sense to do this as the first branch's (connecting the root node with its child) oldest node will always be the same age as the BM meaning $a_u - a_n \equiv 0$ so the branch segment would never age. Instead, a_b was set to the age of the destination node as the branch was made available at age zero of this node.

Once the correct age is calculated, branch growth is carried out: firstly, the change in diameter of each branch, then the change in length, and finally, a per-node tropism is applied.

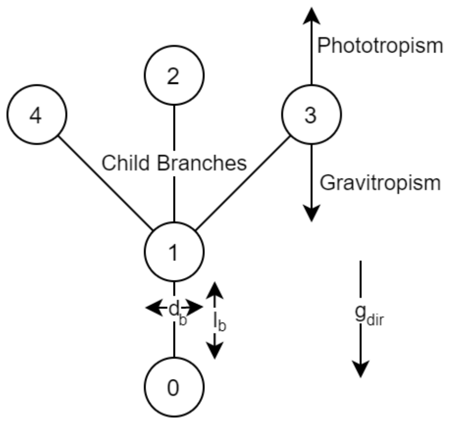
To calculate the diameter of each branch, each branch segment is iterated over and has a value assigned. This is based on the number of child branch segments attached to the segment's destination node. Again, this is carried out in basipetal order as the diameter requires the calculation of the child branch segments' diameters first.

The diameter of each branch is not associated with the physiological age of the branch but instead is calculated as a sum of all child branches. If a branch segment has no children, the diameter is set to ϕ, otherwise branch diameter is defined as

$$d_b = \sqrt{\sum_{c \in C_b} d_c^2},$$

where C_b is the set of child branches and d_c is the diameter of the child. To calculate the branch length, the following equation is used:

$$l_b = min(L_{max}, \beta \cdot a_b),$$



which dictates what the new branch length is. In Synthetic Silviculture, this is used for rendering branches however, for this implementation, it is used to move the nodes.

To do this, each node has a direction associated with it, which is a unit `FVector` from the parent node to this node, and a length from the parent node, providing the information required to translate the node so the new length is l_b. As previously mentioned, this will also translate all of this node's children as the `Translate()` method is recursive.

Finally, to calculate the per-node tropism offset, the following equation is used:

$$\vec{\tau}_{offset}(a_b) = \frac{g_1 \cdot \vec{g}_{dir} \cdot g_2}{a_b + g_1},$$

and applied to each node. One modification made in this implementation was to not translate the node by the tropism offset if $a_b < 0.5$ to avoid aggressive tropism on newly formed branch nodes which initially caused some bugs.

Once each BM had been grown, the bounding sphere was recalculated (as the nodes had moved or new ones had been added), so when the BM manager came to recalculating light exposure each BM had the correct bounding sphere.

4.3.5 Attaching and Orientating Modules

Once a BM had reached a_{mature} (the maximum depth of the graph), it could then produce offspring. To do this, a similar approach is taken as the method described in Sections 4.3.1 & 4.3.2 using the Borchert-Honda model, but this time at module scale instead of plant scale.

For each of the terminal nodes n_i in the BM without a connecting branch segment, a per-node light exposure q is calculated: $q(n_i) = Q(u)/k$ where k is the number of terminal nodes. This is then summed to the root node and redistributed as before to give a vigor value for each terminal node. To achieve this, a topological sort of the graph is required, using the same method for BMs.

Once each terminal node has a vigor value v, for each node where $v > \bar{v}_{min}$, a new BM is created at the terminal node's position and attached by the root node of the new BM using a connecting branch segment.

In Synthetic Silviculture, the BM was then orientated using an iterative gradient descent method, however, this method was vague and complicated and due to time constraints was not included. Instead, the new BMs were orientated using the connecting branch segments orientation so that they would grow in the correct direction for visual continuity.

This part of the system did not work well and, as it was one of the later sections to be implemented, it had less fine-tuning and resulted in many bugs. One issue encountered was plants growing excessive BMs, with every new BM requiring exponential computation for the plant, slowing the system down and sometimes resulting in UE4 crashing. A hard limit of 100 BMs for the plant was set to avoid this, but there were still plenty more minor issues here.

4.3.6 Shedding Modules

The final thing to be implemented was BM shedding and plant death. This was a very basic system and used the value of $\bar{v}_{min}$ as a shedding threshold. For each BM that had a vigor value less than this, it was removed from its parent, the BM manager, and finally deleted.

Due to many objects having references to each BM, there were initially a few issues here with BMs not getting removed or dead BMs (that were now `nullptr`s in memory) having methods executed on them causing program crashes. To alleviate this, a flag was added to each BM that could be used to determine if it was alive or not. Every method checked if the BM was alive (and not a `nullptr`) before executing.

The simulation ended when either the plant had lost all its BMs (due to $\bar{V}_{rootmax}$ being interpolated to zero once $p_t \geq p_{max}$) and was effectively "dead", or when the maximum simulation time had been reached.

4.4 Results

When it came to analysing the results achieved from this implementation, it was logical to compare them initially to the results obtained in the original algorithm. However, as there were many types of results presented in Synthetic Silviculture (such as flowering impacting further development), compared to the Plant Visualiser in this implementation, only the plant type library (seen in Appendix A.3 of Synthetic Silviculture) would be compared to. Synthetic Silviculture provided values for most of the plant type parameters, meaning the efficacy of both implementations can be compared easily.

Before comparing however, two differences should be highlighted: in this implementation foliage was not included due to time constraints, and Synthetic Silviculture's plants were rendered offline versus this implementation rendering in real-time.

The results of 16 different plant types from both implementations can be seen in Figure 8.

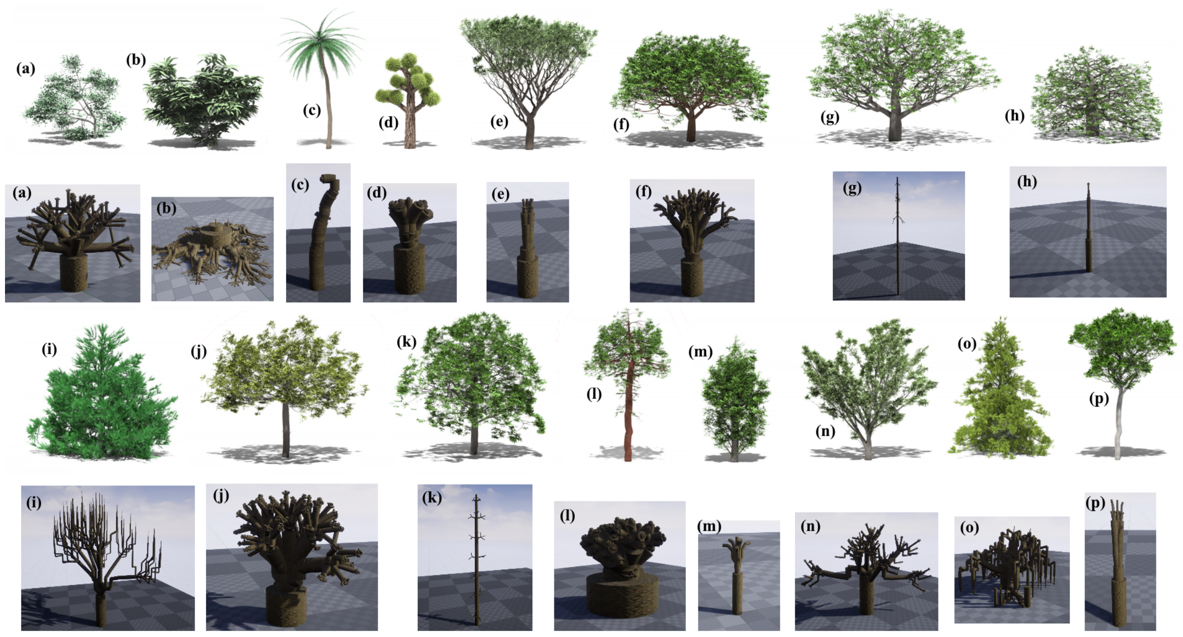


Figure 8: Comparison of results achieved from Synthetic Silviculture (rows 1 & 3) and this implementation (rows 2 & 4) using the values in Appendix A.3 (Table 4) of Synthetic Silviculture. *Plant types are (a) Bush 1, (b) Bush 2, (c) Palm, (d) Joshua, (e) Acacia, (f) Apple, (g) Ash, (h) Oak, (i) Pine, (j) Maple, (k) Beech, (l) Sequoia, (m) Poplar, (n) Cork, (o) Spruce & (p) Meranti.*

At a glance, the results are clearly not identical. Many of the plant settings when input into this implementation resulted in vastly different plants. Whilst interesting self-similar branching structures have been captured, most of the plants created by this implementation are not realistic and certainly do not look like plants found in nature. However, despite many of the plants not looking the same as in Synthetic Silviculture, some plants would not look out of place if used in dense jungles or on alien planets.

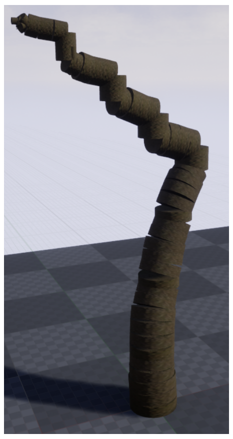
Focusing on only this implementation, some plants do deserve some merit; (b) while not looking like a bush does display intricate root patterns that could be used if combined with another plant, (k) looks like a large pine tree and with a few tweaked settings would provide better conifer results than (i) and (o), and (a), (f), (i), and (j) all look relatively natural—even if they do not look like the plants they are supposed to.

One of the important features that a realistic forest generator must have is being able to capture lots of the biological phenomena seen in nature. In this implementation, the effects of tropism, branch self-similarity, and little branch overlap—due to the plants self-organising—are all present.

The quality of this implementation's renders are realistic. Epic's "Quixel Megascans" was used for the branch texture and UE4 has realistic, ray-traced lighting built-in. This resulted in plants that had natural texturing and lighting, all in real-time—a vast improvement when compared to Synthetic Silviculture's less realistic, offline renders.

One issue with some of the larger plant types was no lateral branches being developed (seen in (g), (h), and (k)). This could have been due to the method for calculating collisions not working well for new BMs; when a BM is spawned, it would only have two nodes, resulting in its bounding sphere being small and often inside the parent's bounding sphere, meaning the BM would get no light exposure and subsequently did not receive any vigor. This results in the BM not growing (as $\bar{v}_u < \bar{v}_{min}$) and dying straight after spawning.

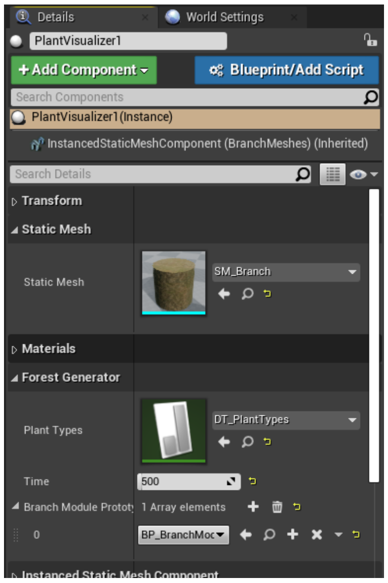
Another issue that plagued new BMs was tropism being too extreme for the small branches. When gravitropism was strong, newer branches inflected and grew either straight down, or along the floor (due to part of the algorithm preventing branches from spawning underground). Examples of these can be seen in (b), (n), and (o), with the reverse seen in (i).



Further to this, there was an issue when a certain depth of spawned BMs was reached where the same pattern would keep repeating, ignoring the tropisms and parent direction (see inset figure).

Despite these issues, the power of the algorithm is clear. Even though some parameters were not included in Synthetic Silviculture (such as L_{max}, $\bar{v}_{min}$, and $\bar{v}_{max}$), using the same value for these in all plants did result in very different results, implying that if the correct values were used the visual fidelity would be even greater (this can be seen in (c) where a small value of L_{max} was used to create the segmentation seen in real palms).

Each plant also used only one BMP—as the selection algorithm was never completed—yet it worked well for most of the plants. If more BMs were used (and chosen based on λ and D) then plants such as (o) may have looked more similar to Synthetic Silviculture's results as the correct BMs for each plant type would have been chosen.



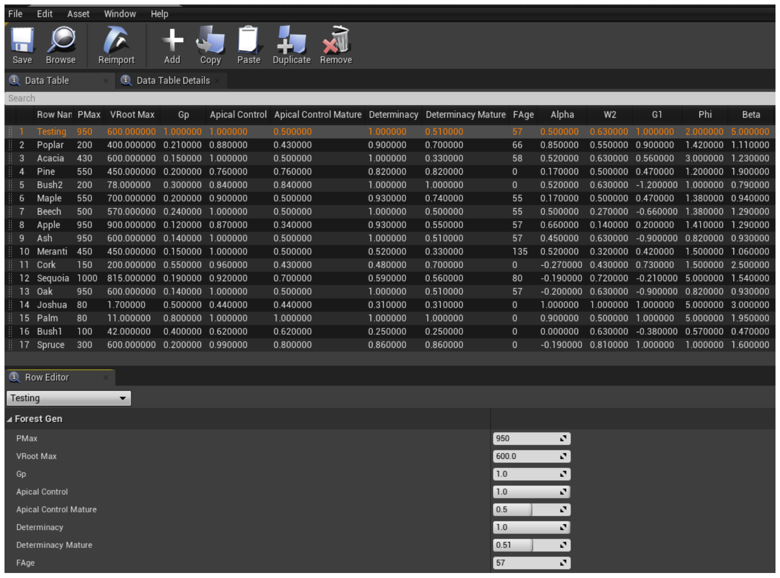
Details | World Settings

PlantVisualizer1

PlantVisualizer1 (Instance)

InstancedStaticMeshComponent (BranchMeshes) (Inherited)

Static Mesh	SM_Branch
Plant Types	DT_PlantTypes
Time	500
Branch Module Prototypes	1 Array elements
0	BP_BranchMod



	Row Name	PMax	VRoot Max	Gp	Apical Control	Apical Control Mature	Determinacy	Determinacy Mature	FAge	Alpha	W2	G1	Phi	Beta
1	Testing	950	600.000000	1.000000	1.000000	0.500000	1.000000	0.510000	57	0.500000	0.630000	1.000000	2.000000	5.000000
2	Poplar	200	400.000000	0.210000	0.880000	0.430000	0.900000	0.700000	66	0.850000	0.550000	0.900000	1.420000	1.110000
3	Acacia	430	600.000000	0.150000	1.000000	0.500000	1.000000	0.330000	58	0.520000	0.630000	0.560000	3.000000	1.230000
4	Pine	550	450.000000	0.200000	0.760000	0.760000	0.820000	0.820000	0	0.170000	0.500000	0.470000	1.200000	1.900000
5	Bush2	200	78.000000	0.300000	0.840000	0.840000	1.000000	1.000000	0	0.520000	0.630000	-1.200000	1.000000	0.790000
6	Maple	550	700.000000	0.200000	0.900000	0.500000	0.930000	0.740000	55	0.170000	0.500000	0.470000	1.380000	0.940000
7	Beech	500	570.000000	0.240000	1.000000	0.500000	1.000000	0.500000	55	0.500000	0.270000	-0.660000	1.380000	1.290000
8	Apple	950	900.000000	0.120000	0.870000	0.340000	0.930000	0.550000	57	0.660000	0.140000	0.200000	1.410000	1.290000
9	Ash	950	600.000000	0.140000	1.000000	0.500000	1.000000	0.510000	57	0.450000	0.630000	-0.900000	0.820000	0.930000
10	Meranti	450	450.000000	0.150000	1.000000	0.500000	0.520000	0.330000	135	0.520000	0.320000	0.420000	1.500000	1.060000
11	Cork	150	200.000000	0.550000	0.960000	0.430000	0.480000	0.700000	0	-0.270000	0.430000	0.730000	1.500000	2.500000
12	Sequoia	1000	815.000000	0.190000	0.920000	0.700000	0.590000	0.560000	80	-0.190000	0.720000	-0.210000	5.000000	1.540000
13	Oak	950	600.000000	0.140000	1.000000	0.500000	1.000000	0.510000	57	-0.200000	0.630000	-0.900000	0.820000	0.930000
14	Joshua	80	1.700000	0.500000	0.440000	0.440000	0.310000	0.310000	0	1.000000	1.000000	1.000000	5.000000	3.000000
15	Palm	80	11.000000	0.800000	1.000000	1.000000	1.000000	1.000000	0	0.900000	0.500000	1.000000	5.000000	1.950000
16	Bush1	100	42.000000	0.400000	0.620000	0.620000	0.250000	0.250000	0	0.000000	0.630000	-0.380000	0.570000	0.470000
17	Spruce	300	600.000000	0.200000	0.990000	0.800000	0.860000	0.860000	0	-0.190000	0.810000	1.000000	1.000000	1.600000

Testing

Forest Gen

PMax	950
VRoot Max	600.0
Gp	1.0
Apical Control	1.0
Apical Control Mature	0.5
Determinacy	1.0
Determinacy Mature	0.51
FAge	57

Figure 9: The Plant Visualiser settings (*left*) and the Plant Type settings (*right*) inside UE4.

Finally, the tool is easy to use and experiment with different settings. UE4 makes it straightforward to modify the static mesh used and the other Plant Visualiser settings; using a `UDataTable` for the Plant Type settings makes it intuitive to create new plants and modify existing values (see Figure 9); having the BMPs inside of Blueprints makes it easy to create different versions and test whether each plant setting worked as intended.

5 Closing Remarks

The original objective of this project was to implement an entire artificial forest generator based on the Synthetic Silviculture algorithm. However, this was too much work for the time frame, especially considering the need to become familiar with UE4 first. Subsequently, only a single plant visualiser was attempted, with many features such as flowering, inter-plant interaction, temperature and precipitation, and BM selection not included.

There were other parts of the code that were meant to be placeholders but never got implemented, such as proper plant type selection, and towards the end of the project the monolithic codebase became increasingly more "spaghetti"-like with important bug fixes inserted at random points in the code.

While the plugin was easy to use it did not perform well; every other simulation would crash UE4 or freeze the system, in part due to the lack of experience with UE4. It was also important to use a suitable IDE. Initially Visual Studio was used—along with the Visual Assist plugin that helps with UE4 syntax—as it is the default IDE for UE4, however, this proved to be very difficult and slow to use, especially with the large code base. Switching to Rider was much better but it still regularly crashed or lost connection to UE4 due to it still being a beta version.

UE4 is certainly powerful, but with complexity comes steep learning curves and many of the impressive features were still not fully understood throughout this implementation. It took considerable time to implement working code and the structure of the system continuously had to be rethought as new UE4 features were learned. One issue that regularly happened was knowing what needed to be accomplished but not knowing if, and how it was possible in UE4. The Blueprints system is certainly quick to learn and can be used for a variety of tasks but was not suitable for prototyping early versions of this implementation.

Becoming familiar enough with Synthetic Silviculture to start implementing this project was expectantly time consuming, with plenty of background research required. However, once the implementation was started, it was mostly straightforward to implement each part of the algorithm.

If this project was to be continued, starting anew and using this implementation simply as a reference would be the suggested method. Many of the classes and responsibilities of this implementation could have been distributed better, and if the initial data structures were made in C++, Blueprints could be used to rapidly prototype the entire system.

On the other hand, if this project was to be attempted again, tackling it as part of a group—with developers experienced in UE4 and C++—would be much better, as a great understanding of the algorithm has already been achieved, meaning the combined expertise of the team would result in a more efficient system.

The UE4 and C++ knowledge gained from this project will be invaluable in the future. Especially as almost all VFX houses are moving towards adopting this method in place of tradition pipelines.

It has been shown that while Synthetic Silviculture provides excellent results, one of the main reasons for SIGGRAPH papers is to present a complex algorithm so that someone with a reasonable level of computer graphics knowledge could implement the system, which this paper did not achieve.

Therefore, one of the main learning points of this project is the importance of describing every part of the algorithm and the approach in sufficient detail. Lots of assumptions had to be made when implementing Synthetic Silviculture's algorithm, meaning many parts did not work as expected.

Finally, the ordering of Synthetic Silviculture was quite cluttered and disorderly, requiring the reader to know information from a future section before they could understand the current one. Despite this, this paper has described the process taken to try to implement Synthetic Silviculture, the assumptions taken, and extensions employed, so that a future researcher could follow the same process and arrive at the same end results.

References

Argudo, O., Chica, A. & Andujar, C. (2016), 'Single-picture reconstruction and rendering of trees for plausible vegetation synthesis', *Computers & Graphics* **57**, 55–67.

Beneš, B., Andrysco, N. & Št'ava, O. (2009), Interactive modeling of virtual ecosystems, *in* 'Proceedings of the Fifth Eurographics conference on Natural Phenomena', pp. 9–16.

Borchert, R. & Honda, H. (1984), 'Control of development in the bifurcating branch system of tabebuia rosea: a computer simulation', *Botanical Gazette* **145**(2), 184–195.

Epic Games (2020), 'Balancing blueprint and c++'.
URL: *https://docs.unrealengine.com/4.26/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/*

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

Lane, B., Prusinkiewicz, P. et al. (2002), Generating spatial distributions for multilevel models of plant communities, *in* 'Graphics interface', Vol. 2002, Citeseer, pp. 69–87.

Lindenmyer, A. (1968), 'Mathematical models for cellular interaction in development, i. filaments with one-sidedinputs, ii. simple and branching filaments with two-sided inputs', *Journal of Theoretical Biology* **18**, 280–315.

Makowski, M., Hädrich, T., Scheffczyk, J., Michels, D. L., Pirk, S. & Pałubicki, W. (2019), 'Synthetic silviculture: multi-scale modeling of plant ecosystems', *ACM Transactions on Graphics (TOG)* **38**(4), 1–14.

Muhammad, R. B. (2010), 'Topological ordering'.
URL: *https://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm*

SideFX (2020), 'L-system geometry node'.
URL: *https://www.sidefx.com/docs/houdini/nodes/sop/lsystem.html*

Weisstein, E. (2021), 'Sphere-sphere intersection - MathWorld, A Wolfram Web Resource'.
URL: *https://mathworld.wolfram.com/Sphere-SphereIntersection.html*

A Appendix

A.1 Initial Planned System

The initial plan for the project was to implement the entire system described in Synthetic Silviculture but due to time constraints this was an unrealistic goal. The system architecture for this plan can be seen in Figure 10.

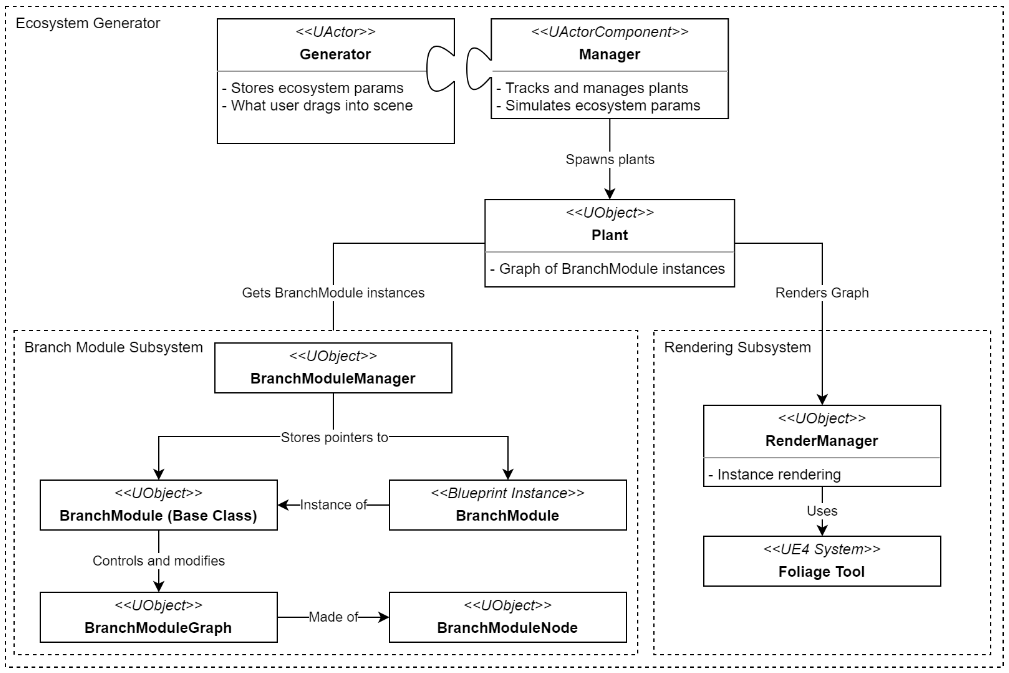


Figure 10: Initial system architecture diagram before project was simplified.

A.2 Plant Type Settings

Throughout Synthetic Silviculture, many variables and parameters are used in a myriad of ways. However, a description of each variable is not always necessarily included. To help aid the reader in understanding these better, all plant type parameters will be described here in detail to be referred to whilst reading this paper.

A.2.1 $\bar{V}_{min}, \bar{V}_{max}, \bar{V}_{rootmax}$

These are used as the minimum and maximum clamps of vigor at different stages of the algorithm.

$\bar{V}_{min}$ is used for shedding BMs (vigor values below this will cause a BM to be shed) and for deciding if new BMs should be attached (if a terminal node has vigor above this value a new BM is attached).

$\bar{V}_{max}$ is used for ensuring that BMs do not get too much vigor causing uncontrollable growth in a small time step.

$\bar{V}_{rootmax}$ is the maximum vigor that can be accumulated in the plant's root BM during one step of the simulation, limiting the overall growth potential of the plant.

A.2.2 P_{max}

The maximum age the plant can reach before it starts to deteriorate. Once $P_t \geq P_{max}$, where P_t is the current physiological age of the plant, $\bar{V}_{rootmax}$ is linearly interpolated to zero, hence the vigor allocated to the plant is reduced until all BMs are shed and the plant dies.

A.2.3 G_p

The growth potential of the plant separate to $\bar{V}_{rootmax}$. Used to calculate the growth rate of the plant with respect to dt (time step).

A.2.4 $\lambda, \lambda_{mature}$

These controlled the apical control of the plant by limiting the ratio of lateral buds, leading to the plant developing a trunk, with λ_{mature} being used once the plant had reached maturity.

A.2.5 D, D_{mature}

These controlled the determinacy of the plant where buds would develop into flowers, preventing further growth. Again, D_{mature} is used once the plant has reached maturity.

A.2.6 F_{age}

The flowering age of the plant, once the plant reaches this age, it will start producing flowers and offspring.

A.2.7 g_1

Controls how fast the tropism effect decreases with time. A high value of g_1 would cause the plant to be effected by tropism for longer as it ages, with a low value trending toward zero at a younger age.

A.2.8 α

This was used to determine the direction of tropism, with negative values indicating gravitropism and positive values phototropism. Used when orientating new BMs for the optimisation procedure and for BM adaptation, moving each branch node by a small amount. Is used to determine the value of g_2, where $g_2 = -(\alpha \cdot \tau_{strength})$.

$\tau_{strength}$ was a new parameter added in this implementation and served as a separate way of controlling the strength of the tropism affecting the plant during adaptation.

A.2.9 w_2

A weighting (< 1) that controls how much tropism effects the orientation angle of new BMs. w_1 could be calculated from this, $w_1 = 1 - w_2$.

A.2.10 ϕ

The thickening factor of the branches, used as the default branch thickness for branch segments with no child branches.

A.2.11 L_{max}

The maximum length that a branch can grow to.

A.2.12 β

The scaling coefficient of branch length, multiplied by the age of the branch to get the new length up to a maximum of L_{max}.

A.2.13 γ

The straightness of plant growth. A value between zero and one, with one being straight, controlling the random angle that each branch would grow compared to its parent. Only included in this implementation to improve the visual fidelity of plants.

A.3 Calculating Bounding Spheres and Intersecting Volumes

In Synthetic Silviculture, B_u is calculated using the centre point of each BM's geometry. In this implementation, a very basic algorithm to calculate B_u that gives "good enough" results is used.

The algorithm works by getting an average of the current node positions and uses this as the midpoint (also known as the centre of mass), then gets the furthest node from the midpoint and uses the distance of the vector between these as the radius. Some bugs initially occurred here as when a module initially spawns, it can have a close-to-zero radius, so a minimum radius of one was chosen for B_u.

To calculate the intersecting volume, the sphere-sphere intersection algorithm seen in Weisstein (2021) was implemented,

$$V_{intersect} = \frac{\pi(R + r - d)^2(d^2 + 2dr - 3r^2 + 2dR + 6rR - 3R^2)}{12d}$$

where R is the radius of B_u, r is the radius of the neighbour B_w, and d is the distance from the centre points.

A.4 Spawning Child Nodes

```
const FVector ParentPosition = Parent->GetPosition();
TArray<UBranchNode*> ChildrenNodes = Parent->GetChildren();
Algo::Reverse(ChildrenNodes);
UBranchNode* Child;

const FRotator ParentRotation = Parent->GetDirection().ToOrientationRotator
   () -
   FVector::UpVector.ToOrientationRotator();
FVector Position = FVector::UpVector;

// Children.Num() can be max of 5

// If there is an odd number of children one child is always straight up
if ((ChildrenNodes.Num() + 1) % 2 == 0)
```

```
{
   const FRotator Rotator = FRotator{
      FMath::RandRange(-10.f, 10.f),
      0.f,
      FMath::RandRange(-10.f, 10.f)
   } * (1.f - Straightness);
   Child = ChildrenNodes.Pop();
   Position = Rotator.RotateVector(Position);
   Position = ParentRotation.RotateVector(Position);
   Child->Translate(ParentPosition + Position);
   Child->RecalculateDirection();
}

if (ChildrenNodes.Num() == 0)
{
   return;
}

// This is used to make sure not all the branches come out the same
   direction for each SpawnChildren call
const FRotator Rotator = FRotator{0.f, FMath::RandRange(0.f, 360.f), 0.f};


if (ChildrenNodes.Num() == 4)
{
   Child = ChildrenNodes.Pop();
   Position = FVector{1.f, 0.f, 1.f};
   Position = Rotator.RotateVector(Position);
   Position = ParentRotation.RotateVector(Position);
   Child->Translate(ParentPosition + Position);
   Child->RecalculateDirection();

   Child = ChildrenNodes.Pop();
   Position = FVector{-1.f, 0.f, 1.f};
   Position = Rotator.RotateVector(Position);
   Position = ParentRotation.RotateVector(Position);
   Child->Translate(ParentPosition + Position);
   Child->RecalculateDirection();
}

Child = ChildrenNodes.Pop();
Position = FVector{0.f, 1.f, 1.f};
Position = Rotator.RotateVector(Position);
Position = ParentRotation.RotateVector(Position);
Child->Translate(ParentPosition + Position);
```

```
Child->RecalculateDirection();

Child = ChildrenNodes.Pop();
Position = FVector{0.f, -1.f, 1.f};
Position = Rotator.RotateVector(Position);
Position = ParentRotation.RotateVector(Position);
Child->Translate(ParentPosition + Position);
Child->RecalculateDirection();
```