|  |
| --- |
| **Procedurally Generated,**  **Environmentally Responsive Trees**  Stylianos Zachariou  BSc (Hons) Computer Games Applications Development, 2023 |

|  |
| --- |
| School of Design and Informatics  Abertay University |

Table of Contents

[Table of Figures ii](#_Toc92987347)

[Table of Tables iii](#_Toc92987348)

[Acknowledgements iv](#_Toc92987349)

[Abstract v](#_Toc92987350)

[Abbreviations, Symbols and Notation vi](#_Toc92987351)

[Chapter 1 Introduction 1](#_Toc92987352)

[Chapter 2 Literature Review 2](#_Toc92987353)

[Chapter 3 Methodology 3](#_Toc92987354)

[Chapter 4 Results 4](#_Toc92987355)

[Chapter 5 Discussion 5](#_Toc92987356)

[Chapter 6 Conclusion and Future Work 6](#_Toc92987357)

[List of References 7](#_Toc92987358)

[Appendices 9](#_Toc92987360)

Table of Figures

Table of Tables

Acknowledgements

Abstract

Abbreviations, Symbols and Notation

UE5 – Unreal Engine 5

SCA – Space Colonization Algorithm

SPA – Shortest Path Algorithm

FPS – Frames Per Second

# Chapter 1 Introduction

# Chapter 2 Literature Review

For the effective implementation of procedurally generated, environmentally responsive trees, extensive research was required, to explore suitable branching algorithms. As well as efficient real-time generation, it’s necessary for the chosen algorithms to achieve realistic growth, therefore, further biological investigation was necessary for the recognition of key tree attributes and affecting environmental factors. In this chapter, three algorithms, commonly used for the procedural content-drafting of trees, will be analysed, with further exploration on biological factors affecting trees and effective evaluation methods.

SCAs are noticeably suitable for the procedural generation of trees, due to their often resulting organic branching structures (Fu, et al., 2023). As the name suggests, these algorithms, starting from a single root point, can successfully expand and cover a large amount of a given volume (Runions, et al., 2005). As described in (Runions, et al., 2007), the process starts with the initial node being placed where the tree is required to grow. A volume is chosen, close to the root node, and filled with randomly placed attraction points. The colonization begins with vectors being drawn from the original node to nearby attraction points. Those vectors are normalised and added up, with a new node placed in the attraction direction. Each attraction point also has its own decimation radius and will be deleted if a node is detected entering. With the recursion of the SCA and the addition of interpolation while new nodes are added, the tree can seem to be growing in real-time.

SPAs can also be employed for the successful procedural generation of realistic-looking trees in real-time. Similar to the SCA, a root node and a volume should be initialized at the required tree location. An irregular graph is created in the volume, with random edge weights and multiple random endpoints. Using the algorithm, the shortest path from each endpoint to the root is found and a mesh is created representing that path. The process is repeated, and a tree structure is constructed (Xu & Mould, 2012). In previous implementations, like XU & Mould, 2012, of the SPA, Dijkstra’s algorithm was used due to the performance flexibility content-drafting provides. However, because of the imperative attention to performance in this project, the A\* might provide a more suitable solution, due to its higher efficiency (Wayadhi, et al., 2021). Although satisfactory results can be achieved using this technique, guiding vectors can be utilized, adding more control to the tree’s shape (Xu & Mould, 2015). Guiding vectors will alter the edge weights, therefore, influencing the path between endpoints and root point. Guiding vectors are also a key to enforcing a controllable amount of naturalistic randomness, while ensuring branch dispersion.

Initially introduced by Lindenmayer in 1968, L-systems is a mathematical theory of simple multicellular organism development, which was later applied to plant growth. (Prusinkiewicz & Lindenmayer, 1990) The main concept of this theory is the recursive replacement of an object’s parts using a specific ruleset. For the rewriting to take place, the intended structure must be represented using a string genotype. Each rule defines the replacement of certain symbols with the general formula being: (symbols) -> (new symbols). The more iterations of a ruleset on a genotype, the more complex structures will be produced. Theoretically, this process can be repeated an infinite number of times, therefore creating fractals (Prusinkiewics, 1986). Because of their fractal-like, recursive structures, trees could be replicated with the employment of simple genotypes and rulesets.

Although widely used, L-systems greatly affect performance, because of their fractal-like behavioural pattern. Since this might threaten the real-time aim of the project, the SPA and SCA will be favoured over it, due to their prospect of better performance with minor modification.

A significant component of the project is the surrounding habitat and its effects on the generated trees. In Fowells & Means, 1990, some major environmental factors affecting tree shape or growth rates were identified and will be used as generation parameters in the application. Firstly, the average temperature, as well as the daily fluctuation in temperature, have an impact on the chemical reactions of trees which directly correlate to the rate of growth (Fowells & Means, 1990). Each type of tree has a range of temperatures in which it can survive and an optimal temperature in which it thrives. The average temperature will be an environmental factor affecting tree generation in the project, however, since the day-night cycle will not be simulated, the daily fluctuation in temperature will not be considered. Moreover, the book recognises light as an influential element. The light’s intensity and quality regulate photosynthesis, while the direction controls the tree’s shape, due to a process called “phototropism” (Kendrik & Kronenberg, 1994). The sun’s position will be used as a parameter for the procedurally generated trees, affecting the amount of tree branches created towards the direction of the light. Moisture and soil quality are also established as major environmental factors impacting a tree’s rate of growth (Fowells & Means, 1990). Soil quality is a broad term including texture, depth, acidity and structure. In the application, both moisture and soil quality will be implemented with the simplification of soil quality to the single parameter of soil acidity. Lastly, although minor, strong, continuous wind is mentioned to affect a tree’s shape with most of its branches growing towards the wind’s direction. Therefore, wind will be the last element added to the simulation, affecting the trees’ new branch growth.

The optimal values of these environmental factors greatly vary between different types of trees, thus, for this project the English Oak (“Quercus robur”) will be used as a template for the generated trees. In Gilman & Watson, 1994, common morphological and behavioural patterns of English Oaks are identified. This species of trees is found in temperate regions; hence, the cardinal temperature range is 4oC - 41oC, with the optimal being 25oC - 30oC (Gilman & Watson, 1994). To simulate this effect, the generated trees will grow the fastest when the temperature is optimal, with the rate being reduced while the surrounding temperature moves further from the 25oC - 30oC range. If the temperature is not in the cardinal range, the tree will not be able to develop. As most trees, the English Oak’s crown is affected by the light’s direction, with more branches growing in positions with increased exposure to the sun. This will be applied to the procedurally generated trees by increasing the probability of nodes being created towards the direction of the light. Moreover, even though the English Oak can adapt to various levels of soil acidities (4.5ph-8ph), a certain amount of moisture is required for it to thrive, specifically, occasionally wet to dry soil. In the project, if the soil conditions are not within the defined boundaries, the trees will grow much slower, with extreme cases completely stopping growth. The English Oak is known to be windfirm, meaning it can withstand high velocity winds without breaking, however, similarly to most trees, the crown can be affected by strong continuous winds in one direction (The Royal Horticultural Society, 2022). In most cases, there will be an increased number of branches growing in the direction of the wind. Depending on the wind direction and power, the generative algorithms will have a higher probability of spawning new nodes in the direction of the wind, therefore morphing the tree in the wind’s direction. Moreover, a range of impacting wind will be enacted of 4-10 Beaufort, rendering only high winds as effective (National Weather Service, 2007). According to the Royal Horticultural Society, the English Oak needs 20-50 years to fully develop, with a minimum height and spread of 12m and 8m respectively. Usually, a wide trunk and drooping branches with a reddish brown colour can be observed by a fully mature English Oak tree. In an attempt to mimic the size and shape, an area will be defined where the tree would be available to grow into. The area will slowly grow throughout the simulation to a maximum amount of 18m height and 15m spread. The trunk and branches will be correctly textured and slowly grow thicker in a similar rate to the procedurally generated tree’s, real-life counterpart.

Branch node creation based on real-life data is only a part of procedurally generating realistic trees. To increase realism, the generated trees have to be represented using a uniform, textured and correctly lit mesh. In the Unreal Engine’s documentation, the “UProceduralMeshComponent” was found as a suitable solution for the creation of the required mesh (Epic Games, 2022). The component takes in vertex positions (in triangles), UV coordinates and normal vectors to create a procedural mesh. Before the creation of the mesh, a circle will be calculated at a certain radius using each nodes position and rotation as the centre. Equidistant points will be created on the circumference of the circles and will be used as the vertex positions. Lastly, the UV coordinates will be calculated, between each pair of circles, using the cylinder formula. As time progresses, and the number of branches increases, the radius of circles surrounding older nodes will increase, therefore, rendering older branches thicker.

After completion, the project requires evaluation to determine whether the initial aim was accomplished. Realism and visual appeal are qualitive measures and opinions may vary between different people; thus, a survey will be constructed and distributed for the collection of ratings for the procedurally generated, environmentally responsive trees. In order to maximize the effectiveness of the survey, Hamed, 2016 was closely studied. The key advice stated in the article is: the questions should be unambiguous, simple, with the avoidance of technical language (Hamed, 2016). Therefore, the survey will consist of multiple images and videos of generated trees using all three algorithms and variations of the environmental factors’ values, paired with the simple questions: “How realistic do you find this tree?” and “How visually appealing do you find this tree?”. The user will be able to choose an answer between five numerical ratings, with number one signifying the least satisfaction, gradually increasing up to five, signifying the highest satisfaction. Moreover, the size of the survey has an impact on engagement, so the survey will have an estimated completion time of ten minutes maximum. The data collected will later be used to give each algorithm a realism rating, which will be used as part of the overall evaluation.

Even though three algorithms were analysed, only two will be implemented, due to their prospective low performance effect and high visual appeal, the SPA and SCA. The two algorithms will be modified to generate trees resembling the English Oak’s short trunk and thick crown, while also realistically dynamic to their surrounding environment. The trees will be displayed using the UE5’s procedural mesh component, with manual vertex and UV coordinate calculations. The realism and visual appeal of each algorithm will be later evaluated using a short survey consisting of various photographic material.

# Chapter 3 Methodology

**3.1 Overview**

The procedural generation of trees is often employed in game development, for the drafting of models, later to be polished and added to environments by artists. This chapter specifies how generative algorithms, commonly used for the creation of branching structures, are adapted and evaluated for the creation of tree models, responsive to their surrounding habitat, in real time.

After consideration of multiple engines and coding libraries, the UE5 was chosen, for the implementation of the two generative algorithms, due to its modifiability, high quality visuals and useful built-in functionality.

**3.2.1 Tree Seed**

The tree seed is the base class for the generative algorithms. It includes a scene component, for the placement of the trees in the scene, and variables, customisable from the editor, affecting the trees’ mesh, environmental sensitivity and feature attributes. Moreover, the class contains functionality, essential to both algorithms, such as mesh calculations and tree crown adaptations based on environmental factors.

**3.2.1.1 The Environment**

The environmental factors, identified in the literature review, were applied to the simulation with the creation of the “EnvironmentSettings” class, inheriting from the UE’s “WorldSettings”. The inheritance allows all trees in a level to be affected by the same environmental conditions, due to each level having its own, unique world setting values.

Since the algorithms are attempting to replicate the growth of the same tree species (English Oak), both should be identically affected by their habitat. Therefore, the “ApplyEnvironment” function was constructed, in the “TreeSeed” base class, for the adjustment of tree attributes in relation to the level’s environment settings.

Each tree’s crown position and rotation are affected by the wind’s direction, power, and light’s position. As mentioned in chapter 2, the English Oak is a relatively wind resistant tree species, thus, the wind’s direction only influences the crown if its power is more than 25mph (6 Beaufort). The maximum affecting wind power is set to 63mph, (10 Beaufort), since stronger winds normally uproot trees and would have caused improbable results. Using the wind’s values, an offset vector is calculated, with the maximum distance being equal to the crown’s radius. (Figure) The wind offset is stored and later used by each algorithm during the node spawning process.

Realistically, the light is required to act as an attractive force on the tree’s crown. Hence, a rotator is calculated in accordance with the direction vector between the tree and light’s position. (Figure) To prevent exaggeration, a pitch angle limit is applied to the rotator, in order to restrict the crown’s downward growth.

The remaining environmental factors, established in the previous chapter (soil PH, moisture and temperature), have an effect on the tree’s rate of growth instead of the crown. The rate of growth determines the speed with which a tree’s branches grow and is depended on the closeness of the environmental factors’ values to their cardinal ranges. Initially, the rate is set to its maximum value and gradually decremented by each factor with a non-optimal value. (Figure) If any of the factors has an extreme value which would normally prevent a tree’s development, the rate of growth is set to 0, preventing any new branches to grow.

ADD TABLE WITH ENV FAC AND EFFECT THEY HAVE

**3.2.1.2 Visual Representation**

Due to the complex nature of branching structures, the manual creation of the trees’ visual representation was required. Allowing the generation of separate mesh sections for the construction of a composite visual, the UE5’s procedural mesh component was added to the Tree Seed base class.

Both algorithms’ resulting trees consist of multiple nodes, each holding information of their transform, direction vector and location of their parent node (previous node on the branch). For each newly grown node, new vertices and UV coordinates have to be identified. The process begins with the definition of a circle around the required node. The circle’s radius is calculated based on the number of the node’s children and is rotated using its direction vector. The growth in radius is an additional factor enforcing naturalistic tree generation, with a logarithmic growth rate (Figure). Subsequently, equidistant points are determined on the circumference of the two dimensional shape, with the aid of the “CalculateNodeMeshVerticesAndUV” function, found in the Tree Seed class, with their three dimensional coordinates stored in the array of vertices (Figure). Moreover, the function calculates each location’s corresponding UV coordinates and stores them in the appropriate array. This method is also repeated for the node’s parent with the only exception being the circle’s rotation, which is rotated identically to the child’s circle. The similarity in rotation ensures the consistency of the required branch thickness and the depletion of visual bugs created from abrupt changes in direction. After all the vertices have been established, weaving instructions for the creation of mesh triangles are necessary. Therefore, the “CreateGridMeshTriangles” function, provided by the UE5’s kismet procedural mesh library, is purposed for the population of the triangles array containing vertex indices (Figure). The three arrays are then passed to the procedural mesh component’s “CreateMeshSection”, which generates and renders a new mesh section, representing the newly grown branch section.

Since branches grow randomly, often with sharp changes in direction, a spherical mesh section is also rendered for every new branch section. The sphere is created using the new node’s location and transform and adopts a radius equal to the node’s circle of vertices. The spheres assist in the formation of a seamless resulting visual component representing the generated trees (Figure).

The above described procedure, of composing a tree’s mesh, is encapsulated in the “CreateMesh” function. This function loops through the array of all spawned node’s, gradually each frame, to avoid performance bottlenecks, and develops each node’s mesh sections. Already rendered node’s have a stored mesh section index, which can be used to identify and update their corresponding vertices. To prevent unnecessary calculations, the function is only called after the creation of new nodes.

While striving to increase realism, the importance of the slow evolution of the mesh, became apparent. Hence, the “GrowBranches” function was created for the progressive growth of new branches. This function loops through all newly generated nodes, initially added to the growing node array, and calculates a growth progress percentage based on time and the pre-determined “rate of growth” attribute (Figure). The mesh section creation process is then repeated, however, instead of the growing node’s translation used for the first circle of vertices, a location between the node and its parent is determined using the progress percentage (Figure). Once the growth progress is complete, the node is removed from the growing node array and added to the final node array which is later used in the “CreateMesh” function.

**3.2.2 SCA**

As discussed in the literature review, the SCA (SCA) was an obvious choice for the procedural generation of trees, due to its ability of producing branching structures by mimicking actual plant behaviour.

**3.2.2.1 The Nodes**

Firstly, some guidance is required to control the shape of the growing branching structure. Therefore, the “AttractionNode” class was created, inheriting from the UE5’s actor class, allowing it to be placeable in the world. This primitive class is compromised by a scene component and a collider, both important for the nodes’ placement and detection.

Moreover, the SCA’s growth pattern demands multiple simultaneous, actively developing branches, deeming the construction of a new “TreeNode” class necessary. Objects of this class are not only responsible for carrying out essential growing location calculations but also represent already grown branches. This class includes a scene component and a collider, similarly to the “AttractionNode” class, along with two distinctly sized, spherical, trigger colliders. In order to determine the position of the next node, all surrounding influences should be considered, thus, the larger collider is purposed as a detector. The detection sphere is set to generate overlap events, with the construction and assignation of new custom overlap begin and end functions. The “OnOverlapBegin” function is automatically executed when a begin overlap event is observed and is initially responsible for examining the colliding actor. If the actor is an object of the “AttractionNode” class, it is added to the attraction influences array, while if it’s an object of the “TreeNode” class, it is added to the detraction influences array. The orginal SCA only requires the use of attraction points, however, with additional experimentation, it was discovered that with the awareness of tree nodes as detraction points, the clamped spawning of tree nodes could be prevented.

To further prevent node clamping, the second, smaller sphere collider was purposed as an attraction node “kill” radius. Each tree node’s combined influences, commonly lead to a specific location, which when reached can cause a blockage. Hence, with the destruction of attraction nodes, close in proximity to tree nodes, continuous branch growth is ensured, with the altercation of external influences, affecting new node location calculations.

Being responsible for determining the next node’s location, the “TreeNode” class includes the “CalculateNextTreeNodePosition” function, which performs the necessary calculations and stores the resulting location in an accessible variable. Since the tree’s trunk and crown are generated with two different rulesets, two distinct node spawning calculation methods were implemented. Tree nodes comprising the trunk, need to be able to grow without the impact of any external influences, accordingly, the nodes’ direction vectors are used to devise new spawning locations. A node’s direction vector is determined directly after it was spawned with the help of the, “TreeNode” class’s, “CalculateCurrentDirection” function. Receiving the node’s parent’s location as a parameter, the function calculates the direction vector using a basic vector subtraction. Since this direction vector should symbolize the direction in which the next node must spawn in, a random vector is also added, providing some lifelike randomness to the simulation (Figure).

Without it being the sole affecting factor, the reserved direction vector is also used by growing crown branches. New crown nodes are generated by considering all external influences and combining them into a single average vector (Figure). Because of the two types of influences, an average attraction and detraction vector are initialized, with values based on the product of additions between all attraction and detraction influences’ directions, respectively. The node’s direction vector is also included in the average attraction value, with an equal significance as any other attraction node. The vectors are then normalized, the detraction value is subtracted from the attraction value, and the results are saved in a general average vector.

These growing techniques are distinguished by the “CalculateNextTreeNodePosition” function’s “useDirection” boolean parameter, provided by the SCA. As both approaches result in a single vector variable, the resulting position is found by multiplying the vector’s value to the tree’s branch length attribute, passed as a function parameter, and adding it to the current node’s position (Figure).

With the implementation of the above described functionality, the “TreeNode” class’s need for certain limiting factors, for the prevention of uncontrollable tree node growth, became clear. Subsequently, particular confining factors were added: the maximum number of children and surrounding tree nodes halting new node spawning, and an expiration time. The maximum number of children attribute restricts unstoppable node spawning from largely stimulated nodes, while the maximum number of surrounding nodes ensures the avoidance of crowded tree node areas. The time limit applied to tree nodes, optimises the algorithm by removing old, stale nodes from new branch location calculations.

**3.2.2.2 The Algorithm**

Even though the two node classes incorporate most of the required methods for the SCA, a new “SpaceColonizationTreeSeed” class, inheriting from “TreeSeed”, was created to combine all functionality and control the pacing of the algorithm.

Objects of this class can be placed in the scene, wherever needed, to generate a tree, optionally using the surrounding environmental factors. Firstly, the simulation starts in the “SpaceColonizationTreeSeed” class’s “BeginPlay” function, called automatically by the engine at the beginning of the application. This function sets the foundation for the algorithm by calling the “ApplyEnvironment” method, belonging to the parent “TreeSeed” class, detecting the surrounding habitat’s conditions, and accordingly regulating some of the tree’s attributes. These attributes are then used for the spawning of the crown’s attraction points, in the “CreateAttractionPoints” function. As the English Oak has an indisputable semi-spherical crown, all attraction points are aimed to spawn in random locations inside the required shape, with a pre-determined radius. This is done by initially choosing random positions in a cube, with edges equal to the sphere’s diameter, and saving the ones in the required semi-sphere, while the other positions are discarded and re-calculated (Figure). Depending on the tree attributes, adapted based on the environment, a transform is constructed to alter each attraction point’s position, affecting the final crown’s placement (Figure).

After the completion of the attraction point semi-sphere, the “BeginPlay” function spawns the first tree node using the “SpawnNewNode” function, indicating the beginning of simulation. The “SpawnNewNode” function, takes a parent “TreeNode” pointer as a parameter, however, since the first node has no parent, a null pointer is passed instead. Receiving the empty pointer, the function responds accordingly, creating a parentless node, saving it in the growing node queue.

Since only one node is initially created, a branch is not able to grow yet, therefore, the node is moved to the final nodes array. The lack of growing nodes prompts the class’s “Tick” function, which is called every frame, to call the “QueueNewTreeNodes” method. Aiming to queue new tree nodes for spawning, this function loops through the final nodes array and inquires the validity of each active node’s next tree node position pointer. If the pointer is not null, the parent tree node is added to the new node queue array. While the new node queue is not empty, the “Tick” function is required to call the “CreateNewNodes” function, responsible for spawning the queued nodes. The creation of the new nodes is carried out gradually to decrease the effect on performance. Therefore, the “SpawnNewNode” is called three time per-frame, until all queued nodes are spawned. Additionally, the “SpaceColonizationTreeSeed” class is responsible for checking whether the attraction point crown has been reached by the tree nodes, saving the information with boolean data structure. This boolean is passed as a parameter to each tree node, when spawned, to regulate the next tree node location calculation technique.

With the detection of newly spawned tree nodes, the “GrowBranches” function is called, by the “Tick” function, and slowly grows branches, starting from parent nodes to the new nodes. When all new branches are fully grown, the whole procedural mesh is updated by the “CreateMesh” function, offering a visual representation of the tree’s current state to the user.

The above described procedure, compromises one step of the SCA, which is repeated until the maximum time of growth is reached.

FIGURE OF COMPLETE SC TREE

**3.2.3 SPA**

The SPA (SPA) was the second algorithm chosen for the procedural generation of environmentally responsive trees, due to its high efficiency, aiding in real-time generation, and path branching ability.

**3.2.3.1 Guiding Vector**

In order to employ the SPA, a graph was required, for supplying an infrastructure where the algorithm would be allowed to operate. As discussed in the literature review, the graph should be consisting of guiding vectors for the realistic rendition of branching structures, therefore, the “GuidingVector” class was created, inheriting from the UE5’s actor class.

The class consists of a scene component, providing transformation abilities, with a sphere and capsule colliders, for detection and communication with adjacent guiding vectors. Moreover, the class includes essential functionality, such as the partial creation of the graph and algorithm specific score calculations.

While constructing the graph, connections should be established between neighbouring nodes. Thus, the “DetectConnections” function was introduced to the “GuidingVector” class along with an initially empty, connections array. When called, the function uses the object’s sphere component to detect any overlapping nodes, adding them to the connections array (Figure). Depending on the radius of the sphere collider, this function may occasionally be unable to locate any surrounding objects, therefore, leaving certain nodes stranded. This might cause further issues if the secluded node is later appointed as an endpoint, which would render the completion of the algorithm impossible. To prevent node isolations, a while loop was added to the function, which keeps increasing the radius of the sphere component until an overlap actor is detected. The incrementation in size is gradual as to not create excessive or unexpected connections.

The fundamental difference between a regular node graph and the guiding vector graph, used in this simulation, is that guiding vectors point towards a specific direction which provides a more beneficial score, when followed by the SPA. In this implementation, guiding vectors are crucial for influencing the growth of branches, hence, their direction was made to point outwards, aiding in the spreading of the crown, although still including some naturalistic randomness. The “SetThisAsConnectionsParent” function, was constructed in the “GuidingVector” class, and is a key component on determining each guiding vector’s direction. The function loops through all the node’s connection and sets it as their parent, converting the regular graph nodes into guiding vectors pointing away from the node. Furthermore, if required, it can be used by specifically chosen nodes two purposely manipulate the direction of guiding vectors.

The SPA used for this application is the A\* algorithm. This algorithm calculates and compares each node’s F score using the goal location. Since the calculated scores are node specific, a “CalculateFGScores” function was constructed in the “GuidingVector” class. The function takes the end location position as a parameter, as its essential for determining the score, and initially calculates the node’s G score. The G score represents the distance of the whole path completed to reach the current node and is later to be used during the F score calculation. The distance between this and the preceding guiding vector is calculated, with the amount halved if the previous guiding vector was pointing towards this direction, therefore applying the guiding bias. The current G score is then measured by adding the distance and the previous guiding vector’s G score (Figure). The F score is then calculated with a single equation, adding the G score with the node’s distance to the goal location. The F score is then saved to be later used by the algorithm.

**3.2.3.2 The Algorithm**

For the execution and organisation of the SPA, a separate “ShortestPathTreeSeed” class was created, inheriting from the “TreeSeed” base class. The inheritance not only allows the placement of the class’s instances in the scene, but also provides essential environmental adaptation and visual representation rendering capabilities.

Starting from the “BeginPlay” function, immediately called at the beginning of the application, the surrounding environmental factors are identified and used to alter the appropriate growing attributes, with the exploitation of the parent class’s “ApplyEnvironment” function. The adjusted attributes will be used for the corresponding spawning of the guiding vectors, however, being the goal position of the SPA, the first guiding vector is placed without any external influences to ensure its connection to both, trunk and crown. Therefore, a transform is constructed, only using a z-axis translation equal to the trunk’s height, to calculate the initial guiding vector’s location. The position vector is later passed, as a parameter, to the “SpawnGuidingVector” function, which spawns a new object of the “GuidingVector” class at the required location.

There are two functions constituting the spawning of all the simulation’s guiding vectors, “CreateTrunk” and “SpawnAllGuidingVectors”. The “CreateTrunk” is firstly executed, in order to create a vertical line of nodes, below the initial guiding vector, that will, when connected, formulate the tree’s trunk. It starts by spawning a single node at the base of the trunk and establishing it as an end point, making use of the guiding vector’s “SetEndpoint” function. Thereafter, while iterating through positions between the trunk’s base and peak, new nodes are spawned at pre-determined intervals, with the addition of a minute random offset. Although minimal, the offset provides a sense of naturalistic growth to the trunk without the production of extreme or unrealistic results.

In order to complete the irregular graph’s foundation, the “SpawnAlllGuidingVectors” function is then called. As described for the previous algorithm’s attraction node crown construction, a semi-spherical shape should be instituted in an attempt to mimic the tree’s real-life counterpart’s (the English Oak) shape. Therefore, a specific number of random positions in a cubic volume, with edges equal to the required sphere’s diameter, are selected. The nodes located in the desired semi-spherical top half are spawned and saved, while the rests’ positions are re-calculated. During the creation of the nodes, a transform is also produced using all environmentally adapted attributes, causing the crown to realistically respond to its surrounding habitat.

After their initialization, the “BeginPlay” function loops through all nodes, calling each’s “DetectConnections” function, weaving them into a collective irregular graph. As discussed, in order to convert nodes to guiding vectors, parents need to be established. Therefore, the algorithm’s “SetParentsGuidingVectors” function calls all existing nodes’ “SetThisAsConnectionsParent” function, converting each node’s neighbours to guiding vectors. This process starts with the base crown node, dictating the crown’s middle guiding vector directions to ensure the outward spreading of branches, while guiding vectors’ directions further away from the centre are unpredictable.

FIGURE OF ALL GUIDING VECTORS

The final necessary preparation stage is the random election of endpoints. Endpoints mark the beginning of each path, or the end of each branch, needed to be traced to the initial crown base node by the SPA. The “ChooseEndpoints” function was created for the arbitrary picking of a pre-determined number of endpoints. The function includes the choosing of random indices, each representing a guiding vector, converting the corresponding node to an endpoint. The endpoint’s index is then saved in a separate array, marking the end of the algorithm’s preparation.

Prior to any further implementation of the SPA, two new essential arrays were initialized, the visited and unvisited node arrays. At the end of the “BeginPlay” function, the first end point is added to the unvisited nodes array, inciting the launch of the simulation. The “ShortestPathTreeSeed” class’s “Tick” function, then starts to automatically execute every frame by the engine. This function is responsible for progressing and, when required, reset the A\* algorithm, while also rendering and updating the tree’s visual component.

Initially, the “Tick” function confirms the existence of end points. If none are found, the algorithm is completed, however, if the endpoint indices array is populated, the “StepAStarAlgorithm” function is called. This function aims to move the SPA by one step each frame, as to not affect performance, and returns a boolean representing the state of completion of the algorithm (True: Path completed, False: Path not yet completed). The process begins with the iteration of the unvisited node array, with the guiding vector owning the lowest F score declared as the current node. The current node is then validated and diverted to the visited node array. If the current node is not the algorithm’s goal node, all its undiscovered connections are obtained and added to the unvisited node array, while also calculating their scores using the “CalculateFGScores” function. This allows each step of the algorithm to explore more guiding vectors, drawing a path to the target position. On the other hand, if the function has reached the required goal node, the path needs to be saved. With this implementation of the SPA, it is essential for branches to be saved in a separated data structure, in order for the mesh functions to slowly render them. A simple structure was created named “Branch”, consisting of a guiding vector array. When a path is completed by the “StepAStarAlgorithm” function, a new branch object is created with its “nodes” array being populated by the guiding vectors constituting the path. The order in which the nodes are placed in each branch’s array is reversed, therefore, starting with the goal node and ending with the endpoint node, the only exception being the trunk path. The first solved path is the trunk, which is required to visually grow in the opposite direction as all other branches; instead of starting from the crown, it should start from the endpoint. Thus, the stored branch’s array order is switched before adding it to the “growingTreeNodes” array.

When the SPA is completed the “ResetAStar” function is called, in order to reset all essential arrays and guiding vectors to their original state, allowing the commencing of the SPA’s next iteration.

The final essential element of the simulation is the mesh generation. Hence, with the detection of newly generated branches, the “GrowBranches” function is called, to slowly progress the visual growth of the already developed branches. When all new branches are fully grown, the “CreateMesh” function is executed, updating the visual component and offering a complex mesh, representing the final procedurally generated tree.

FIGURE OF COMPLETED SPA TREE

MAYBE ADD UML DIAGRAM SECTIONS TO BOTH ALGOS

Add figures to SPA

**3.3 Evaluation**

The aim of this project was the adaptation of branching algorithms for the in-game procedural generation of environmentally responsive trees. To therefore, accurately compare each algorithm’s response to the envisioned functionality, both qualitative and quantitative evaluation methods were employed.

Procedurally generated content is usually manually edited by professionals before being added to a game. These algorithms, however, are implemented as to automatically produce content, without any human interference, hence requiring immediate naturalistic and aesthetically pleasing results. Since realism and visual appeal are qualitative variables, an anonymous survey was conducted for the gathering of appeal and realism ratings. This survey consisted of videos, which the participants were asked to rate by answering the following questions:

*“How visually appealing do you find the above tree?”*

*“How realistic do you find the above tree?”*

There are five number rating choices for each of the questions, number one signifying the worst (dissatisfaction) and five signifying the best (satisfaction). Because of the diverse growth conditions and results the algorithms can produce, eight videos of distinctive trees were chosen, grown by both algorithms in the following habitats:

* Negligent environmental conditions, having no effect on the trees. (Figure)
* Strong continuous wind in a specific direction. (Figure)
* Light only reaching the trees from a certain side. (Figure)
* Light only reaching the trees from a certain side and strong continuous wind in a specific direction. (Figure)

With the additional goal of simulating procedural tree generation in real-time, quantitative performance calculations were essential for the fair evaluation of the two algorithms. Therefore, using the UE5’s timing capabilities, each algorithm’s delta time will be recorded, measuring the amount of time spent per frame. Moreover, the scalability of the procedural generation algorithms will also be tested by recording the framerate of the application while generating various amounts of trees simultaneously.

Finally, the two algorithms will be compared by combining the results of both evaluation methods, quantitative and qualitative, aiming to determine the most balanced procedural generation method.

**3.4 Summary**

Although very distinct, common factors were identified between the SCA and SPA simulations, which enabled an object oriented approach while implementing the project. By inheriting from a common base class, consisting of environmental and mesh calculation methods, each algorithm specific class was able to further expand functionality and successfully procedurally generating trees. Through various optimization techniques, the two algorithms were able to simulate tree growth in real-time, confirming the possibility of in game adaptations. Due to the complexity of this projects aim, both qualitative and quantitative evaluation methods will be used, to accurately assess each algorithms performance effect and visual appeal, with the goal of identifying the most well-balanced method for procedurally generating environmentally responsive trees.

# Chapter 4 Results

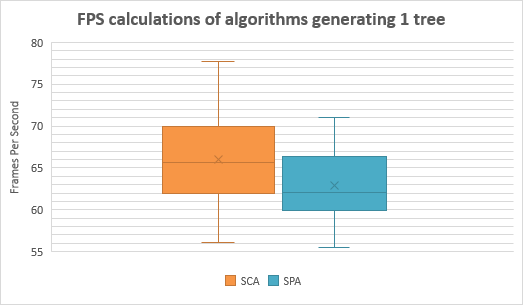
As mentioned in the previous chapter, due to the complexity of this project’s aim, both qualitative and quantitative evaluation methods were required for the accurate comparison of the implemented algorithms. Therefore, using extensive performance calculations and an anonymous survey, all necessary data were acquired.

**4.1 Quantitative Data**

In order to assess the effect on performance, the possibility of in-game adaptability and scalability of each algorithm, time measurements were essential.

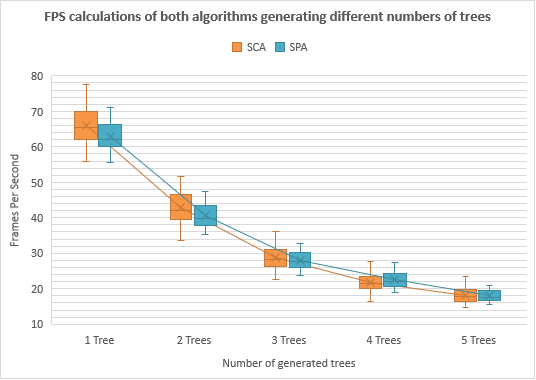
The two algorithms were tasked to generate one tree, without any affecting environmental factors. The delta times of the first two thousand frames, consisting of both trunk and crown generation, were then measured and exported to a text file. This method was repeated five times for each algorithm, with repetition and total averages being calculated. The delta seconds were then converted to frames per second for the easier understanding of values. For the clearer perception of data, a line graph was constructed, since the raw data consisted of twenty thousand numbers (Figure).

Figure darada displays the two algorithms’ performance in the first two thousand frames of the tree generation. As can be seen, the SCA had a higher overall FPS fluctuation, while the SPA had less value variation. The SCA can be identified as the best performing algorithm with a higher range of FPS than the SPA. Moreover, there is a clear negative correlation between the two algorithms’ performance and the simulation’s progression, which is stabilised during the final five hundred measurement frames. A box and whiskers graph can also be plotted to better visualise and compare the general performance of both algorithms (Figure).



The box and whiskers graph clearly shows that the SCA’s and SPA’s results were in the ranges of 56.26-77.68 FPS and 55.54-71.09 FPS respectively. Furthermore, the SCA had an average of 65.99 FPS and median of 65.62 FPS, which is higher than the SPA’s average, 62.96 FPS, and median, 62.07 FPS.

For evaluating scalability and the possibility of multiple of these trees being adapted to a game’s background environment, the same timing process was repeated for both algorithms while attempting to generate multiple trees simultaneously. As before, for each number of generated trees the simulation was repeated five times, with the first two thousand frames’ delta seconds being stored and exported. Each algorithm’s separate results were plotted on a line graph to observe the difference in performance with the addition of more trees (Figure). The collective results were also plotted on a box and whiskers graph for the simplification of the immense amount of data (Figure).



In FIGUREs TADADA and tadaera the negative correlation between the applications’ FPS and the progression of the simulation can still be seen, although fewer fluctuating values were recorded as the number of generated trees increase. Also, from all three graphs, an exponential decay pattern can be seen concerning the FPS, since the values decrease with a decelerating pattern, with a higher effect on performance between one and two trees, than four and five trees. Moreover, a from figures LALALA , the algorithms had a sudden increase in FPS at the beginning of the applications, which was caused by the initialization functions of trees, stacking up and greatly affecting performance.

In comparison, the SPA has mostly higher maximum, mean and median FPS values, in higher ranges than the SCA. Additionally, as each algorithm is tasked to generate more trees, the difference of average and median FPS between the two algorithms declines because of the more abrupt drop in FPS by the SCA. This causes the algorithms to perform almost identically when procedurally generating more than 3 trees.

**4.2 Qualitative Data**

SURVEYY

# Chapter 5 Discussion

# Chapter 6 Conclusion and Future Work

# List of References

Epic Games, 2022. *UProceduralMeshComponent.* [Online]   
Available at: https://docs.unrealengine.com/5.0/en-US/API/Plugins/ProceduralMeshComponent/UProceduralMeshComponent/  
[Accessed 25 November 2022].

Fowells, H. A. & Means, J. E., 1990. The Tree and its Environment. In: *Silvics of North America.* Washington DC: Department of Agriculture, Forest Service, pp. 1-11.

Fu, Y. et al., 2023. *Skeleton extraction and pruning point identification of jujube tree for dormant pruning using space colonization algorithm.* s.l.:s.n.

Gilman, E. F. & Watson, D. G., 1994. *Quercus Robur, English Oak.* s.l.:University of Florida.

Hamed, T., 2016. How to Design and Create an Effective. *International Journal of Academic Research in Management (IJARM),* Volume 5, pp. 37-41.

Kendrik, E. R. & Kronenberg, M. H. G., 1994. *Photomorphogenesis in plants.* 2nd ed. s.l.:Kluer Academics.

National Weather Service, 2007. *Estimating Wind Speed.* [Online]   
Available at: https://www.weather.gov/pqr/wind  
[Accessed 6 May 2023].

Prusinkiewics, P., 1986. Graphical Applications of L-Systems. *Proceedings of Graphics Interface*, pp. 247-253.

Runions, A., Lane, B. & Prusinkiewics, P., 2007. *Modeling Trees with a Space Colonization Algorithm.* Prague, Proceedings of the Eurographics Workshop on Natural Phenomena.

Runions, A., Lane, B. & Prusinkiewicz, P., 2005. Modeling and visualization of leaf venation patterns. *ACM SIGGRAPH 2005 Papers,* pp. 702-711.

The Royal Horticultural Society, 2022. *Quercus robur.* [Online]   
Available at: https://www.rhs.org.uk/plants/14294/quercus-robur-common-oak-aac/details  
[Accessed 25 November 2022].

Wayadhi, R. M., Ginting, S. H. N. & Syahputra, D., 2021. Greedy, A-Star and Dijkstra's Algorithms in Finding the Shortest Path. *International Journal of Advances in Data and Information Systems,* 2(1), pp. 45-52.

Xu, L. & Mould, D., 2012. A procedural method for irregular tree models. *Computer & Graphics,* Volume 36, pp. 1036-1047.

Xu, L. & Mould, D., 2015. Procedural Tree Modeling with Guiding Vectors. *Computer Graphics Forum,* 34(7), pp. 47-56.

# Appendices