Procedural tree mesh generator plugin for Blender

Pablo Lorite Lozano

Final Degree Work

Bachelor's Degree in
Video Game Design and Development

Universitat Jaume I

July 1, 2021

Supervised by: José Vte. Martí Avilés

Abstract

The goal of this End-of-Degree project has been to create a useful tool that works on a procedural way in order to create unique vegetation tree models that adapts to the needs of each user.

In the last decade, procedural and modular generation techniques have seen their use increased notably in the video game industry. These techniques allow the creation of different instances of almost every element that compose a game in an automatic way, making it possible to achieve single unique gameplay experiences for the users with less time spent by developers.

Blender creation suite has been used for the development of the project. This work has two main parts: the procedural generation system of the optimized tree models and an integration of user input through an user interface added in *Blender*. On one side, the generation of the trees has been divided into four parts: generation of the roots, trunk, branches and leaves. On the other side, would be the interaction of the distinct user inputs parameters within the procedural scripted behavior allowing the user to control the final output. Finally, a fully working tree model generator plugin with a random behaviour driven by user inputs with the indicated characteristics in the Project Design Document has been acquired (see Chapter 2).

Contents

List of Figures

List of Tables

CHAPTER 1

Technical Proposal

Contents

This chapter constitutes the Technical Proposal of the End-of-Degree project at the Degree of Design and Development of Videogames of the Jaume I University. The work consists of the development of a procedural modelling plugin for *Blender* where the user can fully control the output of the final mesh and export it to a game engine once finished. To achieve the project will be necessary scripting the project and integrate it into *Blender* [1].

The differentiating element of this proposal among other works, consists of highlighting two methodologies, one is the design of procedural tree props, which includes a main scripting modeling system that creates the final mesh starting with simple primitives. The other prominent methodology would be the integration of the user inputs within the tree generation system to control the final mesh output.

1.1 Introduction and motivation

This project started motivated by the growing use of the procedural and modular techniques in video games nowadays.

With the basis of investigate the relationship between scripted modelling and the aesthetics of the vegetation in games, and in order to analyze the creation of infinite props that are consistent and good looking at the same time, this project was conceived.

This project aims to the creation of this vegetation props in the form of trees, giving rise to a tree generator tool to potential use in any game.

For that purpose, the first step is the system analysis and design, after that, the study of possible implementation techniques takes place, then the scripting of basics modelling operations going through a procedural generation system.

1.2 Subjects related

The main subjects related to this project are (sorted by University code):

- VJ1203 - Programming I
- VJ1205 - English
- VJ1212 - Graphic expression
- VJ1221 - Graphic computing
- VJ1224 - Software engineering
- VJ1226 - Character design and animation

1.3 Tools

The tools used in this project are:

- ***Blender3D with Python*** to script and model
- ***Adobe Photoshop*** to make sketches and figures.
- ***Overleaf*** to write end-of-grade work memory.
- ***Microsoft Excel*** to create tables and to analyze the user interface logical design.
- ***Lucidchart*** to create diagrams for end-of-grade work memory.
- ***Tablegenerator*** to create tables for end-of-grade work memory.

All the tools used for the development of the project are free and accessible to any user, except *Excel* and *Photoshop*, which only offer a free period of 30 days, but are easily interchangeable for any free alternative of free use.

1.4 Goals

- Carry out the proposal of a fully working plugin, and in addition eventually, it could be continued to improve aspects of it and adding new ones.
- To create an attractive visual style for an in game model and extend it along every output produced by the user.
- To obtain a logical coherence in the procedural generation of the distinct parts of the tree.
- To make every tree mesh optimized and consistent.
- To ensure the creation of a large mesh range.

1.5 Planning

Task Description	**Estimated Time (hours)**
Development of the Technical proposal and *PDD*	5
Research of a visual style and aesthetics	10
Tree model logical design	30
Implementation of the user inputs and plugin interface	55
Scripted procedural 3D modeling	110
Optimization of the generator result	30
Preparation of the end-of-degree project memory	50
Preparation of the end-of-degree project presentation	10
Total	300

Table 1.1: Early estimation of project tasks and hours dedicated

	Quantity	**Unit value**	**Total value**
Time cost	300	3,95 €	1185 €
Hardware cost	Basic pc with low specs: (see Subsection 4.2.2)	800 €	800 €
Software cost	Copy of *Blender* and *S.O.*	Free	Free

Table 1.2: Estimation of project economic cost

1.6 Expected results

The main goal of this project is the creation of a procedural modeling tool that generates unique and distinct tree meshes based on the interaction between user inputs in the form of parameters in the plugin interface and a main procedural process.

Therefore the final result requires the existence of at least one user-based plugin for the execution of the tree mesh generation in *Blender*.

It would also be necessary to expect the creation of consistent, coherent and stylized trees resulting in a wide range of assets can be exported and are ready to be used i.e., in a game engine.

CHAPTER 2

Project Design Document

Contents

This chapter is an overview of the analysis prior to the description of the project engineering and the structure followed at the time of the project was developed. In the next chapter, the methodologies and techniques used will be detailed more exhaustively.

2.1 Concept

The objectives that seek to capture the interest and give visibility to the project are: fluid creation of unlimited tree props without discontinuities in the mesh and with an stylized and organic look at the same time, for that will be necessary to develop a strong scripting modeling system that manipulates the user input data in order to be able to create different props each time and to achieve an useful and coherent implementation of the plugin *UI*.

2.2 Purpose and target

The purpose of the project is to consolidate automated modeling techniques that allow the creation of a tool capable of producing stylized, detailed and optimized tree models.

This project has been developed in order that the generator could be used by artists or developers without the need of previous experience, to being able to easily generate a large amount of resources in a very short time for later exporting and texturing in any modeling program or game engine.

2.3 Project work breakdown

This section describes the structure of the project, using a task breakdown diagram. This diagram constitutes an indicator of the tasks and sub tasks that will be analyzed in next chapter(see Chapter 3). These tasks are defined sequentially on a temporary basis, as they have been developed. Through this work breakdown diagram *(WBS)*, the tasks and sub tasks that make up the work are shown and, more specifically, it allows a clear view of the elements that make up the generator (see Figure 2.1).

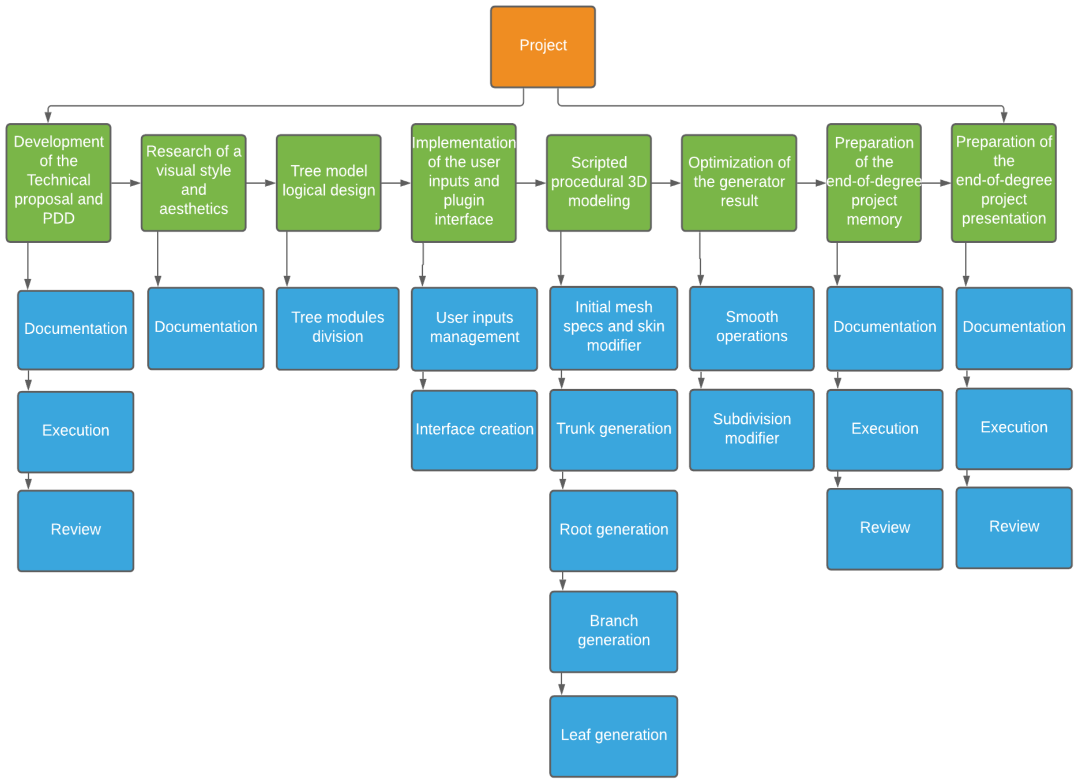

Project
Development of the Technical proposal and PDD
Research of a visual style and aesthetics
Tree model logical design
Implementation of the user inputs and plugin interface
Scripted procedural 3D modeling
Optimization of the generator result
Preparation of the end-of-degree project memory
Preparation of the end-of-degree project presentation
Documentation
Execution
Review
Documentation
Tree modules division
User inputs management
Interface creation
Initial mesh specs and skin modifier
Trunk generation
Root generation
Branch generation
Leaf generation
Smooth operations
Subdivision modifier
Documentation
Execution
Review
Documentation
Execution
Review

Figure 2.1: Work breakdown structure of the project (made with *LucidChart*)

In this way, a task sequence is defined that allows us to discern the structure of the work developed for this project.

2.4 Requirement analysis

The functional and non functional requirements of the project are described below to help understand development guidelines followed for the project. Only the main requirements are listed below, as they are enough to understand the design structure of the system and the main implementation intentions.

A functional requirement defines a specific functionality of the system or its subsystems, while a non functional requirement represent general restrictions on the design or implementation of the system (e.g., to meet performance, usability, or reliability constraints) [20].

2.4.1 Functional requirements

Functional requirements describes the capabilities or features that the solution must have to satisfy the requirements of the project objectives. They are expressed in terms of what the solution's behavior should be and what information it should handle, providing a sufficiently detailed description, to allow the development and implementation of the solution [18].

- System will produce an output for each user interaction with the interface.
- Each parameter in the user interface accepts only numeric values.
- Each output produced by the system will be different from the previous one as long as the user has modified a parameter in the generator interface.
- Every mesh produced by the system will be capable of being used and exported.
- In each user interaction with the graphical interface, the process will start from scratch, begin evaluating all the parameters and perform the calculations that will produce the mesh output.
- The calculations implemented by the scripting system are performed on the vertices and edges of the mesh. Or over a primitive in the case of leaves generation.

2.4.2 Non functional requirements

In short terms, non-functional requirements define how a system is supposed to be. Among the enumeration of non-functional requirements presented, are those referring to attributes such as efficiency, dependability and usability of the system.

- The project will be executed using free and open software on any platform.

- The interface design will allow any untrained user to be able to use the software developed in a short time of use.
- The system will be documented by a manual that will describe how to install and use it.
- The system will be developed as a single process using the native scripting language for *Blender*, *Python*.
- The interface will be developed under the *Blender Operator* class
- The system will produce outputs in time intervals less than 20 seconds as a worst case.

2.5 Use cases

This section aims to document the behavior of the project system from the user's point of view. For this, the project's use cases have been analyzed. First, the actors that participate in the project's use cases are described:

- User: represents the people who use the generator. They have access to all the external management of the system and interact directly with the modeling suite and with the generator's graphical interface.
- Modeling system: represents the main process that is responsible for producing the output of the corresponding mesh with the values of the interface parameters.
- Plugin interface: this actor represents the visible part of the generator which houses the parameters through which the user will interact to make use of the system.
- Modeling suite: represents the modeling program on which the project developed as a plugin that works independently but using the technology provided by this modeling suite is added, in our case it is *Blender*.

The use cases are identified from the functionalities indicated as circles (see Figure 2.2).

Finally, the actors and use cases are connected by an association, which represents that an actor makes direct use of the case to which it is connected [8].

Below is the specification of the use cases (see Tables 2.1, 2.2, 2.3). In each table a use case is detailed, as well as the sequence steps needed to give rise to it.

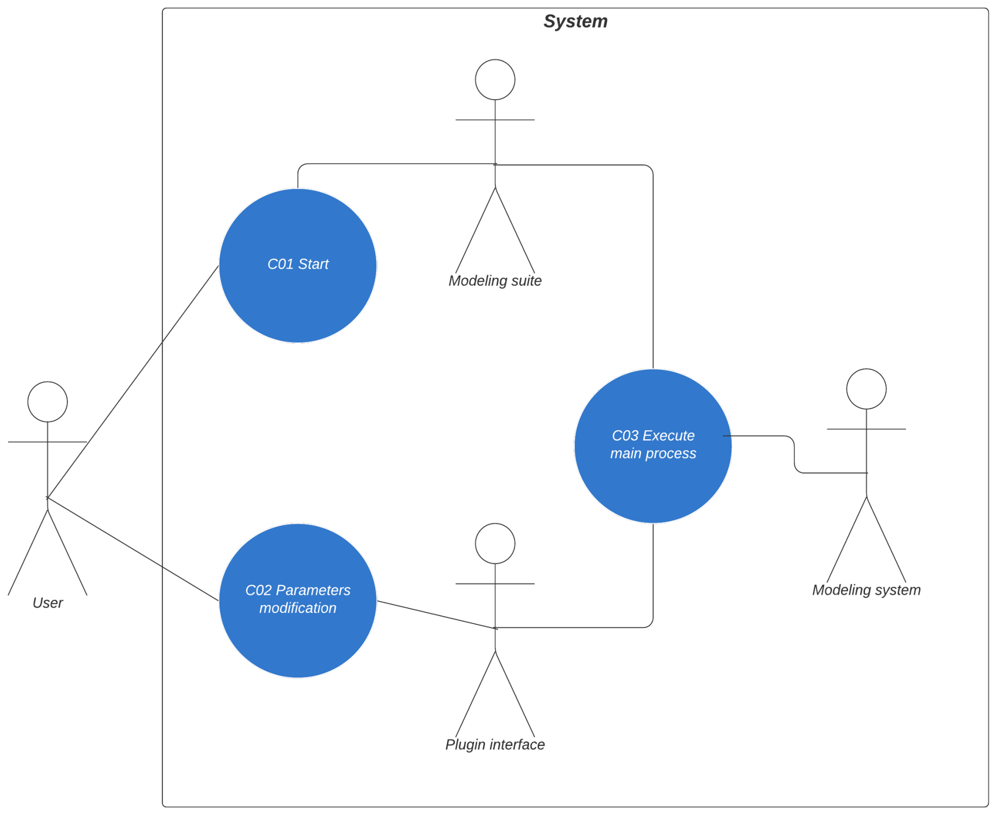


Figure 2.2: Use cases diagram (made with LucidChart)

2.6 System design

The project is broken into two main approaches: Parametric approach, and Procedural approach both are mixed together to make the final project implementation.

In summary the tree mesh generation has been implemented as a system divided into some linear and procedural subsystems that act over parameters modified by the user.

Calculations of the procedural subsystems correspond to the randomization parameters of each module that makes up the final mesh, which are detailed in the following subsections. Linear calculations adhere to the calculation of absolute dimensions, such as the height, length, and width of each module.

In the following subsections the system technologies developed are described in implementing order on the project. Since the project use case is only one because its behavior follows a linear sequence, the best way to describe the life line of the main process of the project is by including a sequence diagram (see Figure 2.3).

Case id: C01 **Name:** Start
Description: It allows starting the generator so that the user can view the graphical interface of the plugin with the parameters with their default values if it is the first time it is started, and if not, with the values of the previous execution.
Step sequence: User opens Blender. User open Add menu in Object Mode User select add Mesh User select Generate Tree System start main process Blender shows generator interface
Precondition: The plugin must have been previously installed. Previous execution of the generator must be closed by the user.

Table 2.1: Case C01

Case id: C02 **Name:** Parameters modification
Description: This case responds to the interaction of the user with the generator through the graphical interface, the modification of any value of the existing parameters in the interface executes the main process.
Step sequence: User change any parameter value in the interface System start main process
Precondition: An execution of the generator must be active.

Table 2.2: Case C02

Case id: C03 **Name:** Execute main process
Description: It allows the generation of a final result that is displayed on the screen, this case occurs every time the user modifies an interface parameter or when the generator is started.
Step sequence: User modify a parameter in the interface/ User starts generator System execute main process System show final output in screen
Precondition: An execution of the generator must be active.

Table 2.3: Case C03

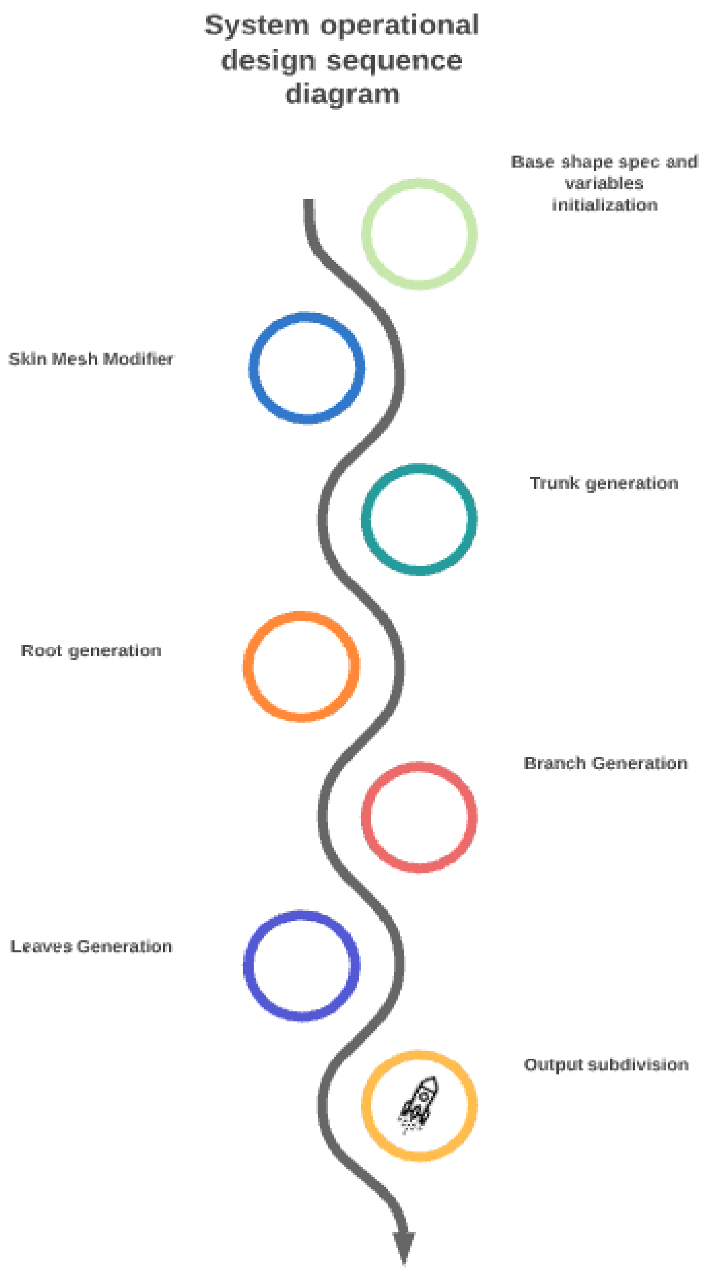


Figure 2.3: Behaviour of the unique main event process through an operational design sequence diagram (made with *LucidChart*)

2.6.1 Parametric approach

Weber and Penn introduced a botanical based but at the same time simple and comprehensive parametric description for tree models creations [17]. An overview of this methodology adapted to the project is given in (see Section 3.4) and all parameters are outlined in next sections. The parametric approach offers the user the possibility to interact with the system and control its output. Adding this approach to the project, allows to have all the good of both sides: manual an automated modeling, the precision when a modeler sculpts a model manually, without automated processes, paying attention to details and at the same time the possibility of producing large quantities of different models in a short time.

2.6.2 Procedural approach

This approach is much less canon than the previous one since it is a unique system created through several intermixed implemented technologies actually called procedural, but it is certainly a mix of several approaches under my own vision, that ultimately give the result of procedural. The different methodologies that it draws on are such as:

- Botanical, this is the less prominent method in the system and applied without a scientific vision and with a simpler one approach to attempt to accurately model the growth process of the tree when for example establishing growth behavior of any part of the tree in user interface

- Modular approach as the one that refers to modular division of space through the division of the mesh in fixed modules that are arranged and merged in a manner such that each join is coherent and consistent and also are independent from each other.

- Also pseudo random libraries have been used [14].

2.6.3 Initial mesh specs and variables

The tree basic mesh creation is built initially using a plane mesh. This plane is converted to the base mesh using the *Skin* modifier, then the initial trunk mesh is created adding a set of additional vertices to that base mesh using the default value for the parameters of the generator.

So initially the mesh is created from few vertices that creates the trunk and based in the input of the user, the system will add or remove any part of the tree. This is the next step on the behavior design of the system.

Distinct parts of the tree will be modified based on trigger variables that will be activated by the user interaction with the generator interface.

2.6.4 Trunk generation

Initially the trunk will appear in a vertical direction following the initial plane mesh that is the vertex marked as root in the mesh. The trunk mesh generation its affected by five subsystems:

- Trunk initial width.
- Trunk end width.
- Trunk body width.
- Trunk height.
- Trunk randomness.

The user will be able to modify the initial width of the trunk, width of the trunk body, width at the end of the trunk, height and randomness in the project interface. Randomness trunk parameter allows the user besides give the trunk a random appearance by modifying trunk vertices positions, to add up to four trunks, that will randomly appear as user changes parameter in the interface.

2.6.5 Root generation

The modelling of the roots follows the initial vertex of the trunk marked as root, so initially the roots will appear in positions calculated using random values that follow circle equation with center around the trunk.

The user will be able to modify the number of roots of the tree, width, length and randomness in the project *UI*.

- Root quantity.
- Root width.
- Root length.
- Root randomness.

Roots vertices initial position initially follows a random value within a circle path using values in a range from zero to one, with a fixed offset between each root to avoid overlapping meshes. Roots could be created in any position following this path with the offset given.

The maximum number of roots allowed to avoid discontinuities in the mesh are sixteen, four for each region of the plane x-y.

2.6.6 Branch generation

The user will be able to modify the quantity, width, length, randomness, and tropism through the project *UI*.

- Branch quantity.
- Branch width.
- Branch length.
- Branch randomness.
- Tropism.

A placeholder system for node branches around the trunk based on branches quantity parameter will be developed, this system will work with the data of the branch node vertex position around the trunk. Each trunk could have up to twelve nodes and each node could randomly generate a dead branch or a branch itself.

Tropism, its a parameter based on organic behaviour that will act on branches orientation over the z axis.

2.6.7 Leaves generation

The modelling of the leaves will be implemented using icospheres and *Boolean* functions that will randomly create meshes that envelope the tree following the branches positions and trunk. Also leaves will have an spherical irregular appearance due to the use of a random number of icospheres, the range of icospheres that could form each leaf is up to five.

The user will be able to modify the amount, size and randomness of the leaves.

- Leaves quantity.
- Leaves size.
- Leaves randomness.

Randomness of the leaves will add an offset to envelope points calculation for the leaves. The amount of leaves parameter behavior will be based on the amount of branches used for calculations of vertex positions where the branches will appear.

2.6.8 Output optimization

Before finalizing, the system must make sure that the mesh is consistent for later texturing or animation. In all stages of the mesh generation, the position of all the vertices corresponding to each part is stored. At this point where the smoothing process occurs, the discontinuities and exaggerated deformations of the mesh are lightened.

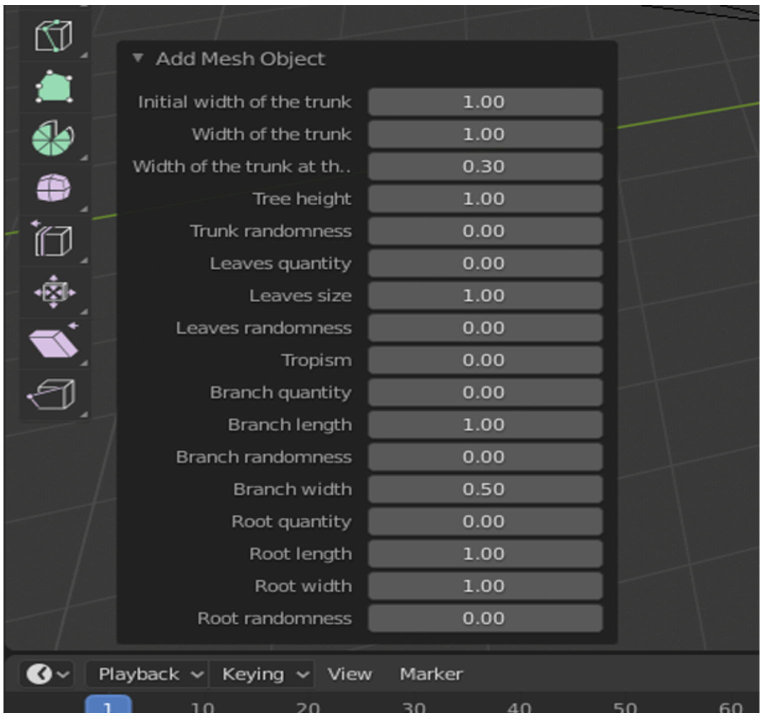
This is done at the end stage of the process so the calculation of the smoothing does not affect the previous calculations of the positions of the vertices. After that the subdivision surface modifier is applied so the mesh initially squared due to *Skin* modifier obtains a smoother appearance by recursive splitting of it geometry.

This process enables a rendering of the mesh with a smoother appearance while maintaining a low number of polygons. It also allows the user, by changing the subdivision surface modifier parameters, to obtain different levels of detail of the mesh.

2.7 Interface design

The interface is designed based on the attributes declaration structure provided by the *Blender Operator* class. The operator will appear at the bottom left of the work space. Within the operator, the interface menu consists of a list of parameters, which have been listed in the previous subsections. Interface parameters provide the input values for the automatic modeling process. All these values correspond to rational numbers within a set range and they have been implemented to work as sliders, although the numerical value can also be entered, typing the value in the value field.

Next, the generator interface is shown (see Figure 2.4).



▾ Add Mesh Object

Initial width of the trunk	1.00
Width of the trunk	1.00
Width of the trunk at th..	0.30
Tree height	1.00
Trunk randomness	0.00
Leaves quantity	0.00
Leaves size	1.00
Leaves randomness	0.00
Tropism	0.00
Branch quantity	0.00
Branch length	1.00
Branch randomness	0.00
Branch width	0.50
Root quantity	0.00
Root length	1.00
Root width	1.00
Root randomness	0.00

Figure 2.4: Detailed screenshot of the generator interface.

2.8 Concept arts and mockups

This section lists some graphic material corresponding to the initial logical design of the system through a kind of primary user interface and parameters values mockup and a concept art to exemplify and to get an idea about the variety of different shapes that the generator aimed to produce (see Figure 2.5).

Study of concepts and sketching was needed, going through the creation of digital 2D concepts and handmade, that way the general guidelines of logical design becomes more intuitive (see Figure 2.6).

In the Resources append (see Append A) there are more additional resources like screenshots of the outputs provided by the generator.

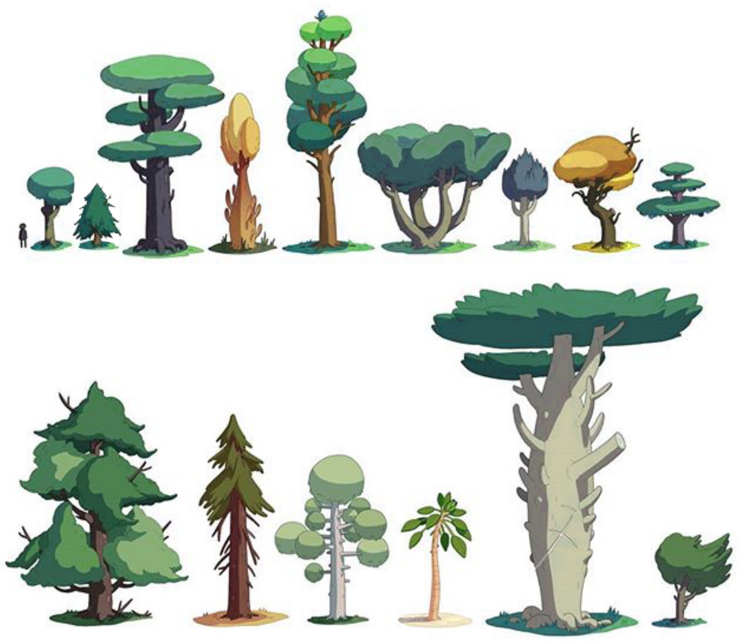
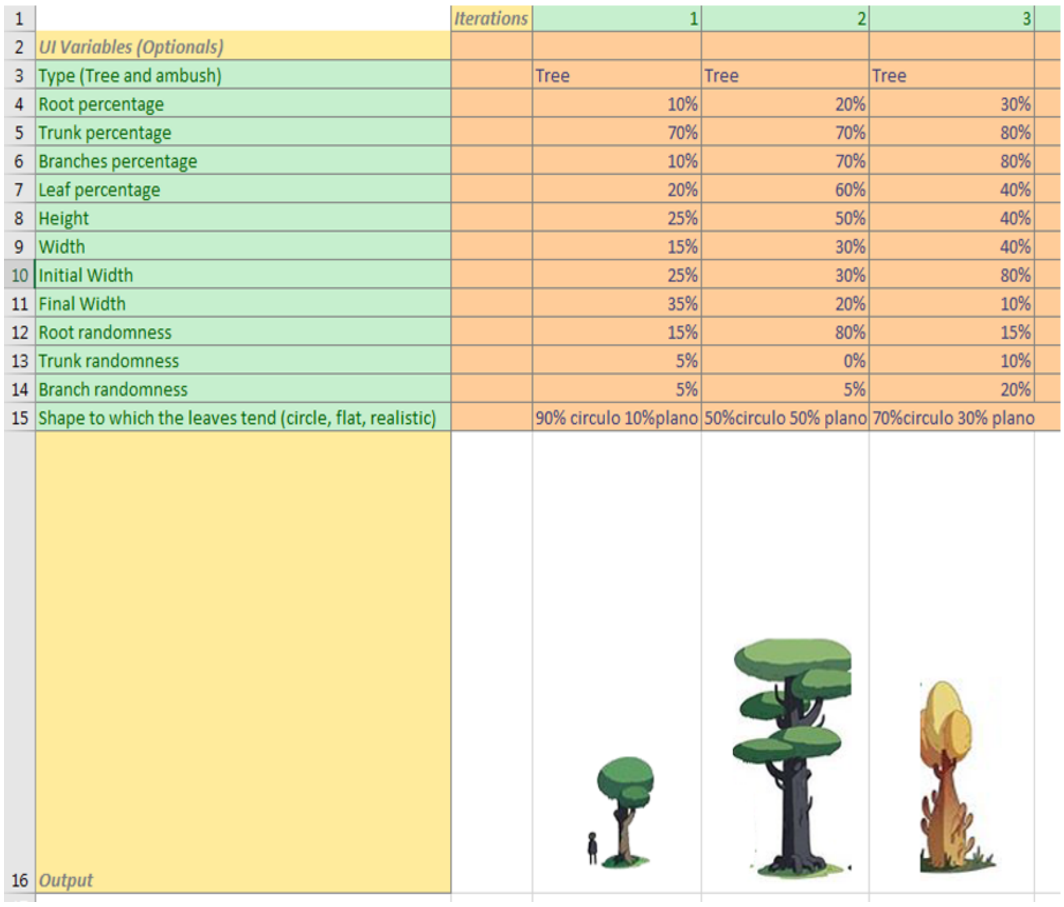


Figure 2.5: Concept art made using Jake Morrison tree design examples [11].



1		*Iterations*	1	2	3
2	*UI Variables (Optionals)*				
3	Type (Tree and ambush)		Tree	Tree	Tree
4	Root percentage		10%	20%	30%
5	Trunk percentage		70%	70%	80%
6	Branches percentage		10%	70%	80%
7	Leaf percentage		20%	60%	40%
8	Height		25%	50%	40%
9	Width		15%	30%	40%
10	Initial Width		25%	30%	80%
11	Final Width		35%	20%	10%
12	Root randomness		15%	80%	15%
13	Trunk randomness		5%	0%	10%
14	Branch randomness		5%	5%	20%
15	Shape to which the leaves tend (circle, flat, realistic)		90% circulo 10%plano	50%circulo 50% plano	70%circulo 30% plano
16	*Output*				

Figure 2.6: Initial user interface scheme used for logical design analysis.

Chapter 3

Project Development

Contents

Most relevant aspects of the developed work are explained in this chapter following a chronological order, according to the planning. The tasks listed in the project planning (see Section 1.5) will be detailed, as well as the sub tasks into which each one has been divided. Automated procedural modeling techniques will be emphasized as this is the core of the project.

All deviations from the initial planning are also detailed and justified. In this way, readers of the memory must be able to understand the main reasons for possible discrepancies between the objectives of the project, the initial system design, and the final results achieved.

3.1 Technical Proposal and Project Design Document

The first task faced in the development of the project was the development of both the technical proposal and the *PDD*, documents which have been adjusted to fit within the first two chapters of this report.

The task has consisted of the documentation and writing of both documents. The technical proposal sets out the main guidelines of the project, and the *PDD* details the initial design decisions made for the development of the project.

For the writing of these documents *Overleaf* [13] has been used, a free collaborative authoring set of tools that allows among other things to create and edit documents online using *LaTeX*, a powerful language for scientific and technical writing [12].

3.2 Representation and research of a visual style and aesthetics

Taking into account that the objective of this project is not the realization of a tree generator with a high number of polygons that results in high definition sample, since there are already numerous tools for this purpose available for use in *Blender* like *Speedtree* [15].

After studying more organic approaches to the creation of trees, using different techniques such as recursion or fractals, or simply observing nature, and comparing with the style of the best-selling tree assets [16], it has been opted for an intermediate style, similar to cartoon but without being as realistic as a fractal or recursive approximation, and maintaining a quite decent topology of the mesh.

The project focuses on creating meshes with few polygons with a less complex approach but with higher precision than the usual for a low poly model, allowing to maintain high performance without sacrificing the detail of the mesh.

For this reason, the use of the *Blender Skin* modifier has been chosen since it gives a stylized cartoon appearance but with more detail than the usual low poly tree.

This, together with the use of the subdivision surface modifier, allows the user to choose the amount of polygons they want in their final mesh and therefore achieve an interactive level of detail, being able to choose a more or less softened appearance of the mesh.

3.3 Tree model logical design

After the study described in previous section where organic and low poly models were analysed and knowing the fact that the system developed wasn't following a recursively or fractal scheme due to its high calculations times and its excessive complexity of the mesh, totally avoidable in most of today's gaming assets. The choice was to make a modular division of space of the mesh where the user interact with the parameters that affect directly to mesh vertex and edges, being able to modify the behaviour of each module of the mesh allowing to each part affect to the others.

So a modular division of the tree was the technique focused, implemented using triggers and vertex groups, to achieve a customization of the mesh without many constraints. Dimensions such as height or width which are absolute and cover the tree's travel frame, called the tree line, are the dimensions that form the main constraints, as well as some minor smoothing constraints calculated in the final stage of the generation process (see Section 3.6).

By looking the following sketch (see Figure 3.1), the initial mesh division can be visually determined. The calculation of the generation of roots begins in v0, while the

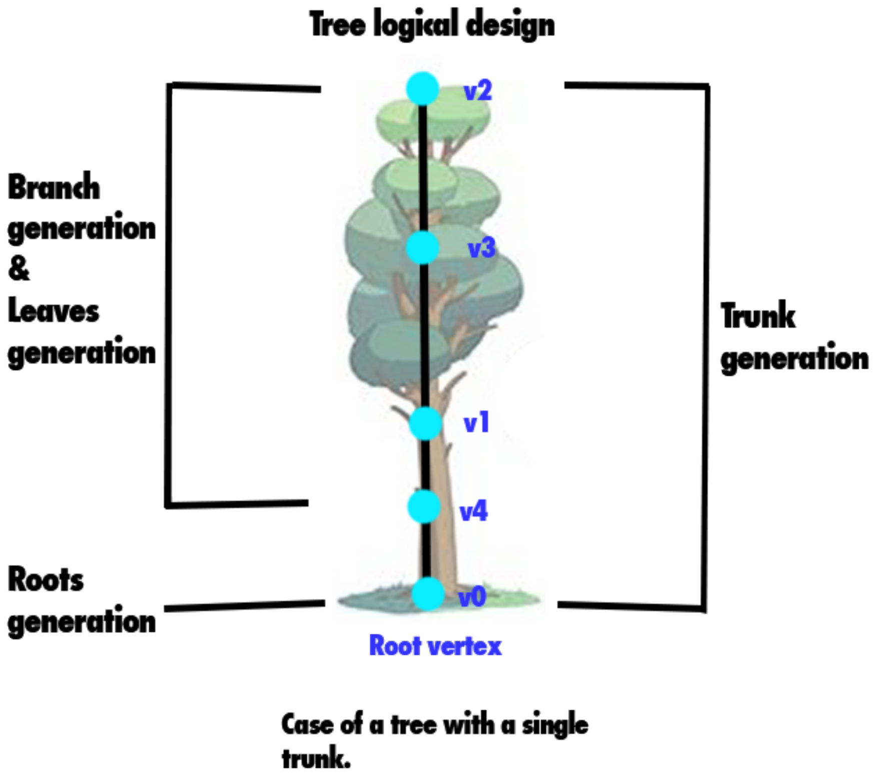

Tree logical design
v2
v3
v1
v4
v0
Branch generation & Leaves generation
Trunk generation
Roots generation
Root vertex
Case of a tree with a single trunk.

Figure 3.1: Tree logical design scheme and initial vertex distribution.

generation of branches occurs in the area that includes v2 and v4, the calculation of the positions of the leaves is carried out based on the position of the branches and covers the same area of influence, leaving the area of the tree closest to the ground without the possibility of generating branches or leaves. Finally, the generation of the trunk affects all the vertices that compose the tree line.

The zones of influence are calculated based on the four vertex groups which cover the different segments of the mesh that correspond to the modules of the tree detailed in previous figure, other groups of vertices have also been added for use in additional calculations.

From these vertices that make up the structure of the model, there are some that are permanent while others that are variable are added or subtracted depending on the variation of the different parameters of the generator, the permanent vertices correspond to v0, v1, v2 and v3.

The operative structure of the triggers is simple, the trunk module is permanent, only the positions and angles of its vertices are modified, but it is always active since every tree has a trunk, whether it is larger or smaller. The module of roots, branches and leaves are independent of each other, but dependent on the module of the trunk, which we could call the central module. Finally, the vertex groups correspond to the vertices that make up each of the four modules that make up the tree.

- Trunk vertices.
- Roots vertices.
- Branches vertices.
- Leaves vertices.

3.4 Implementation of the user inputs and plugin interface

I have chosen to implement a parametric system inspired on *Weber* and *Penn's* approach on parametric description but modified to use mainly iterative algorithms instead of recursion. This choice has been made due to the properties of the generator which are detailed in the next section(see Section 3.5).

The main contribution of the *Weber* and *Penn´s* model is the use of nodes and inter nodes to control the shape of each segment that makes up the mesh. The nodes of each element correspond to the permanent vertices of the mesh and the inter nodes are additional nodes that can be added or subtracted at any point in the execution, and that serve to add extra vertices, modify their topology and to be used in other calculations, such as the calculation of randomness.

Their model used recursive curves, however the model presented for the execution of this project uses vertices transformations, which together with the skin mesh modifier produce the same results as using curves but with a slightly smaller amount of calculations.

3.4.1 User inputs management

Because of this need of variable vertex and edges amount storage, user inputs system has been developed this way because in the iterative case, the system variables provide full access to any variable value change at any point of the execution, this is important because we always know the value of the final result for any variable even if we stopped calculation between steps. This does not happen with a recursive process because if we stop or change execution at any time we may not known the final value for a variable because we missed intermediate or assistant values used in recursion, so in a system that is constantly modified by the user, recursion doesn't look like the proper choice. It was necessary to create a structure that would allow for consistent data management.

In software engineering, singleton or single instance is a design pattern that allows you to restrict and fully manage the creation of objects belonging to a class and the value of any object parameter [23].

Singleton model ensures that a class has only one instance and one global access point to it. The *singleton* pattern provides a single global instance because the class itself is responsible for creating the only instance and it allows global access to that instance through an unique class method.

This pattern has been applied because the user manages a large number of variables which in turn become more variables at run-time during the modeling process written in the back end, and the values of the variables must remain with the same value between different executions of the generator, unless the user enter a new value in the interface, so these properties of the system need to be modified precisely, to adapt to the generation of a mesh in real time(see Append B).

This model of declaration for the properties of the system has been chosen since declaring the variables as global in the usual way within the program, caused problems since in each iteration of the generator the global variables were not stored with the values corresponding to the previous iteration. This occurs because in each user interaction with the generator interface, the global variables didn't store the previous values and deleted their content with each new user interaction.

The implementation made makes use of the *Blender Singleton* class in which all the system variables are declared within a single object, which is called slots, in this way the constructor of the class that gives values to the parameters only has to act on a single static object with multiple parameters, so this way system is able to store previous states value. A simpler alternative would be to store the properties or parameters directly in the *Blender* scene, but it's not reliable and it's a bit of a rough approach.

3.4.2 Plugin interface creation

To carry out the creation of the interface, the *Operator* class of *Blender* is used since it allows the creation of a floating menu, without being restricted to additional measures, as could be the case of having created a panel, or a toolbar. This alternatives were considered but later discarded. Inside the *Operator* class, the parameters have been declared as float property, with default values and minimum and maximum values within a set range(see Append B). Below is a table with the description of the parameters on the user interface(see Table 3.1).

3.5 Scripted procedural 3D modeling

Procedural modeling is usually implemented using algorithms that establish the creation of a set of logical rules for creating the resulting objects, and usually do not allow the user to modify this set of rules. However, the focus of this project does allow the ability of the user to modify this set of rules through the user input parameters.

Parameter title	**Parameter description**
Trunk initial width	Width of the trunk at the initial point
Trunk end width	Width at the end of the trunk
Trunk body width	General width of the trunk main body
Trunk height	Absolute height of the tree
Trunk randomness	Amount of randomness applied to the trunk
Leaves quantity	Amount of leaves generated
Leaves size	Magnitude of the leaves
Leaves randomness	Amount of randomness applied to the leaves
Tropism	Affect branch and leaves orientation and direction
Branch quantity	Amount of branches generated
Branch length	General length for the branches
Branches randomness	Amount of randomness applied to the branches
Branch width	General width of the branches
Root quantity	Amount of roots generated
Root length	General length for the roots
Root width	General width of the roots
Root randomness	Amount of randomness applied to the roots

Table 3.1: Generator user interface parameters description

Procedural modeling is often applied when it comes to objects that follow a modular structure, objects for which it would take too long to create a 3D model by hand. A modular structure refers to the fact that if an object is divided into different modules, each individual module would follow the same generation patterns as the complete module. Therefore the procedural generation is especially suitable for use in objects that follow modular structures such as plants, architecture or mechanical parts [21].

Objects that have this type of structure are more predisposed to their procedural generation because it allow to follow an understand some simple logical rules behind that allow distinguishing its more complex patterns and finally being able of parameterizing its generation.

Since the parameters interact constantly with the procedural modeling system and are subject to be modified at any time during the execution, the process should be programmed iteratively and not using recursion for reasons of consistency during execution.

Recursive loops use intermediate calls of the process to calculate the final result. Therefore in a system that uses a recursive main process, additional memory would be needed to store the intermediate values of each call while the rest of the recursive calls are made, in order to use them to calculate the final result.

In the case that the process was iterative, it can be seen that the result is always accessible even if the execution of the process is modified by the user at any point, in that case the calculation is interrupted and the value before the interruption is returned as a result. In other words, we will never depend on having a failure when trying to obtain an intermediate value after an interruption as in the recursive case since it is not necessary to use any intermediate value, because everything is stored in the arguments of the function.

In fact, this is basically why a main iterative process that uses procedural techniques has being developed. The behavior and implementation of this techniques that make up the project is detailed below.

3.5.1 *Skin* modifier

The mesh skeleton consists only of vertices and edges. These elements are the only ones elements to be modified under the procedural generation system. This is possible due to the use of the *Skin* modifier that allows the creation of a skin around this vertex and edges skeleton, using a radius calculated for each vertex as the distance to the skeleton.

This modifier is especially suitable for use in the creation of meshes with aleatory topology [5]. For this reason, it is the main modifier used and which gives the characteristic appearance to the mesh.

Prior to the use of vertices and edges together with this modifier, an approximation was made through the use of curves and vectors, but to achieve the same ability to modify precisely the mesh many more calculations were required.

3.5.2 Trunk generation

The base mesh corresponds to an initial trunk module, so the generator always starts first with the trunk generation, an initial height calculation is applied and depending on its value, auxiliary vertices will be added or subtracted, this initial calculation is later supported by a more organic growth calculation. At the initial stage is also where the vertex group corresponding to the trunk is created. Next, the state of the random trigger is checked and if the value of the randomness parameter is greater than zero, the value of three main variables are initialized:

- **Trunk random value**: a list of random numbers that is calculated based on the numerical value of the interface parameter applied is applied in each coordinate of the space on the position of each vertex of the trunk.
- **Number of trunks**: the number of trunks varies from 1 to 4 depending on the amount of randomness parameter that provides the user input. Four has been set as a limit due to the study and observation of different values and it was concluded that most trees have at most around four main trunks, in addition to the problems of mesh discontinuity that could appear if more trunks were added.
- **Trunk axis**: corresponds to the attraction weight of each axis in the event that more than one trunk is created. These values will be applied only to the end vertices of the trunk.

  The application of these variables takes place in the final function after initialization stage.

This function consists of a loop that will act according to the number of trunks, firstly the final vertex of the new trunk to be created is extruded from vertex v1, which we say is the root vertex of the trunk module when there is more than one trunk that needs to be created.

To this final vertex, the previously mentioned trunk random value is added to its position and the weight of each axis is applied also to determine the line of the final tree (see Append B). At this point it would be worth highlighting what it is considered as a tree line, and the reason for calculating the weight of each axis for each trunk. Generally the term tree line is referred to the general silhouette of the trunk formed by the main trunk or trunk. The calculation of the influence of each axis is carried out by modifying the final position of the vertex already randomized, aiming to avoid that two trunks grow in the same position.

Once the end vertex is extruded to the calculated position and the tree line is established, the different random values from the list are applied to the inter nodes that make up the trunk.

It is also at this point where, if there is more than one trunk, an additional vertex is added for each trunk to improve the control of the subsequent generation of the branches.

Finally an important technique that greatly improves the visualization of the model occurs, the smoothed growth system, it consists of the application of smoothing values that are dynamically calculated based on the position of the vertex within the trunk module, the value of the trunk randomness parameter, the height and the number of trunks.

In the first place, it is detected if there is a trunk or more than one. If there is more than one, an offset is applied to v1, the root vertex of the trunk module, this operation allows the root of the trunk to grow slower than the vertex at the end of the trunk, to resemble more the organic behavior that a tree would have. The values of the vertex positions closer to the end of the trunk receive a harsher smoothing. This smoothing is decreasing in intensity as the position of the vertices of the trunk decreases. Finally, the height and number of trunks affect smoothness calculation in the way that the higher the height value is, smoothing value will increase for all vertex and if the height is small, the smoothing will be minimal. The same happen with number of trunks, the more trunks the smoother it gets.

3.5.3 Root generation

The modeling of the roots begins with v0 which is the vertex marked as the root within the *Skin* modifier that acts on the entire mesh. The choice of this vertex as root, in addition to being obvious, allows the vertex to act as node with multiple connections to other nodes. This parameter on the modifier allows the skin to better adapt to the structure of these vertices resulting in a smoother representation of topology than it might be expected in a fan like vertex distribution (see Figure 3.2).

This are the main parameters that act over roots creation:

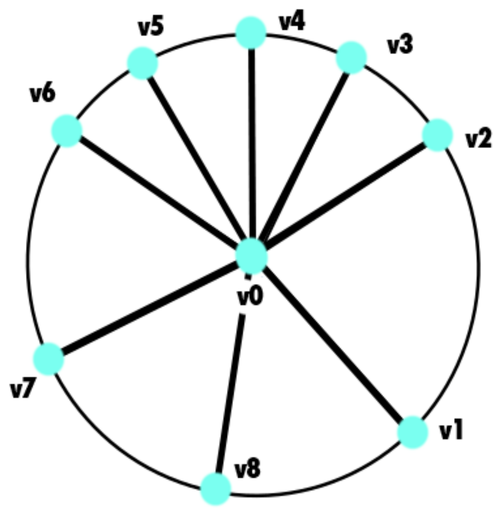


Figure 3.2: Fan vertex distribution.

- **Number of Roots**: First, the number of roots of the mesh is calculated based on the value of the interface parameter, this number varies randomly in a range from zero to a maximum of sixteen roots, always depending on the number of roots marked by the user(see Append B). Next step is the variation of the root positions based on the random parameter.

- **Root position**:the value of the position of each final vertex of the root is calculated and distributed randomly following a circular distribution. These values are fixed to maintain an offset that vary depending on the number of roots.

- **Root randomness**: Lists of random values that modify the vertex position, it has two stages depending on the value of root randomness parameter. In the lower half of the values of the input range, positions are calculated adding a random value for each axis, and in the higher half of the series of values available at the parameter interface, random values added to z axis are more abrupt, causing the final vertex of the root to lower its position further. In this part of the implementation each root also gets one additional vertex added for values that are in the lower half of the parameter range and two vertex added for values of the parameter in the higher half.

Finally, the calculation of the length and width obeys a modification of the position of the vertex and its scale respectively. It is a simple operation that responds to absolute dimensions.

3.5.4 Branches generation

The generation of the branches follows a process whose main calculations are the following:

- **Number of branches**: In a first stage, the total number of branches is calculated based on the number of branches parameter, the trunk module is divided into three segments (see Figure 3.1), each of them can randomly contain a number of vertices that goes in the range of none to four branches, giving rise to a maximum of twelve branches per trunk(see Append B).

  The algorithm for calculating the position of the node of each branch within each of the three segments that make up the tree, also ensures that the branches begin to grow at the end of the trunk, being able to have isolated nodes also in the lower part of the tree. This has been done to more closely resemble the organic appearance of a tree where the greatest number of branches are concentrated in the final trams of the trunk.

- **Branch position**:The position of the branch nodes within each segment is calculated randomly by maintaining a dynamic offset that varies depending on the height of the tree, to avoid two branch nodes generating too close from each other. Each branch node has an attribute referring to its birth that allows a branch to grow or not from each node.

  Also regarding a more botanical aspect, the branch growth distributions were studied, observing opposite, alternate and multi-radial distributions and in the end a mixture of all of them was chosen to allow a greater variation in the results.

  The value of the position of each final vertex of the branch is randomly placed in space and extruded based on length parameter where the branches follow a random distribution allowing each branch to grow differently.

  If there is more than one trunk, the generation sectors of the branch nodes change, because as explained in the section on the generation of trunks, if the tree has multiple main trunks, an additional vertex is added to prevent the growth of branches near the root node of the trunk module, which would be v1 (see Figure 3.3).

- **Branch randomness**: Lists of random values that modify branches vertex position. From the beginning, each branch is subdivided into two additional vertices that will see both its position and the position of the final vertex of the branch modified randomly.

- **Tropism**: This is a general parameter of plants that can be due to various factors such as light, the amount of water, or the force of gravity itself. This parameter affects the special growth that affects both branches and leaves where there is a change in direction and orientation in response to an external stimulus such as any of those mentioned above [24]. If the value of the parameter is positive, the vertices of the branch change their position in the same as the supposed stimulus, on the

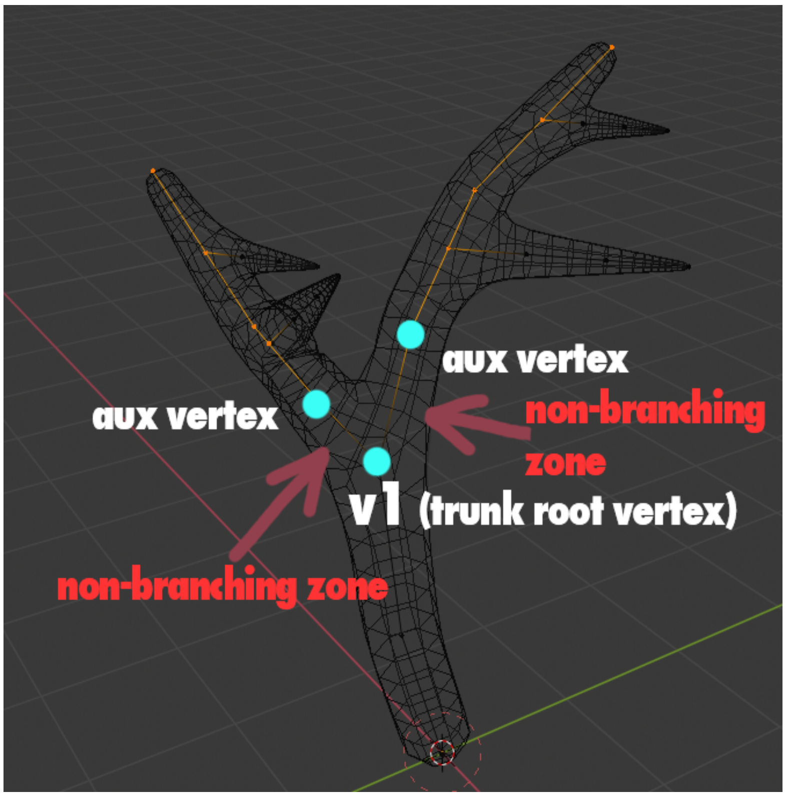


Figure 3.3: Non branching zone description.

contrary if the value of the parameter is negative, the vertices of the branch change their position towards the negative direction. Both behaviors can be interpreted as approaching or moving away from the stimulus

3.5.5 Leaves generation

For the generation of the leaves, icospheres have been used, which is a geometry primitive exclusive from *Blender*, but icospheres are more technically known as geodesic polyhedron, which is a convex polyhedron formed by triangles [19].

Knowing that the icosphere is an approximation to a sphere but with a much lower level of subdivision and therefore with less computational load than for example a sphere or a meta ball, this was the primitive chosen. Indeed the subsystem uses an icosahedron which is an icosphere with level one of subdivision. Besides this since leaves will had its topology modified due to the random attributes of the system, this type of primitive supports better an hypothetical subsequent mapping of the textures, since it offers a more regular surface since the polygons that make up their faces are triangles.

Due to the use of icospheres, the generation of the leaves is not affected by the *Skin* modifier since leaves are created from a primitive instead of vertices and edges.

The generation of the leaves has three main calculations, the position of leaves, number of leaves and their random appearance.

- **Leaves position**: Leaves position are calculated based on branch end vertex position, nevertheless leaves can be generated without the need for branches, in that case trunk vertex group will be used for calculations where a new vertex will be created with a random offset applied to make the leaves grow in random positions in each iteration.

- **Leaves quantity**: The number of leaves is calculated randomly based on the corresponding parameter of the interface, the lower the value, the smaller the number of leaves and vice versa.

  This value is also influenced by two factors, the first one is that the number of leaves in the mesh is a random number in the range of existing branches, and if there are no branches between the different sectors that make up the trunk module will be used but it is logical that if there are fewer branches or there are none, the generation of leaves is less than if there were branches.The second factor is explained in the next paragraph.

- **Leaves randomness**: Second factor has to do with the amount of icospheres used to create each module of leaves. If the parameter of the leaves randomness is modified, the value of a variable whose range varies dynamically randomly from one to five is calculated, the higher the value of the parameter the starting value of the range and the final value of the range of icospheres will be higher

  This calculation affects the creation of each leaf. Subsystem starts by creating one icosphere but can be varied to a different number. Each icosphere used to create a leaf will appear in a random position along a radius ranging from zero to a value slightly less than twice the radius of the first sphere, and also with a random scale and rotation applied. Scale values are in a fixed range that goes from the half of the radius of the first icosphere to a number slightly lower than the double of the radius.

  At the end icospheres positions are following a random variety of a structure known in sacred geometry [22] as the egg of the life (see Figure 3.4). Final step is managed by the application of the *Boolean* modifier that uses join operation where all the icospheres that make up each leaf are mixed in a single mesh to form the topology of each leave.

  After applying the *Boolean* modifier, the *Remesh* modifier is applied to simplify the mesh slightly and give it a more organic appearance [4].

  Remesh modifier is a *Blender* modifier that allows to recreate new topology for a given mesh choosing the level of simplification of the geometry. The application of this modifier allows to make the topology of the icospheres or icosphere used more uniform after the random transformations applied to each icosphere and at the

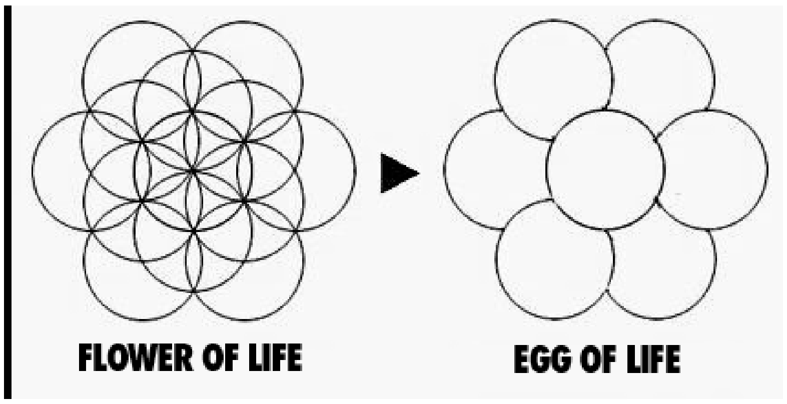


Figure 3.4: Sacred geometry distributions.

same time make the resulting mesh follow the curvature of the surface described by the different icospheres.

A final random transformation is applied to the result of previous step. Each leaf will be re-applied a random transformation that affects its rotation and scale, the range of variation of the scale this time will be smaller to allow the leaves of each branch not to appear of a very different size.

3.6 Optimization of the resulting mesh

Once the pertinent transformations have been applied to each module that makes up the mesh, it is necessary to smooth those parts whose value of the randomness parameter is very high.

To allow this calculation to be carried out at the end of the process, the groups of vertices that make up each module of the tree have been previously saved. This allows each part of the tree to be individually smoothed.

The smoothing process at each vertex is necessary due to the intrinsic operation of the *Skin* modifier, since it places the skin envelope around the vertices with a distance based on an inner radius that it calculates internally for each vertex, sometimes this internal radius between each vertex and the skin can be affected by another very close vertex or by sudden changes in the direction of the next vertex. This causes discontinuities and failures in the mesh and to avoid it is necessary to smooth the vertices positions and consequently the skin calculation will also be smoothed.

Finally the *Subdivison Surface* modifier is applied, it is used to increase the detail of the final mesh by splitting its faces into smaller ones, giving it a smooth appearance. It enables also to create complex smooth surfaces while modeling simple, low-poly meshes. It also avoids the need to save and maintain huge amounts of geometry data, and gives a smooth “organic” look to the mesh [6].

Subdivision surface is applied at the end of the system because *Blender* does not allow to edit the mesh geometry after applying the modifier, so if it had been applied i.e. from the beginning we could not have been able to modify the vertices and edge of the mesh.

3.7 Preparation of the End-of-degree project memory

This section is about how the End-of-degree project memory has been carried out. Like Technical Proposal and *PDD*, memory of the End-of-Degree Project has been written with *Overleaf.* For the creation of this final report, it has been followed the *"LaTeX template for writing the Final Degree Work"* for Video Game Design and Development Degree at Jaume I University developed and provided by *Sergio Barrachina Mir* and *José Vte. Martí Avilés.*

Another tool used for this memory writing has been *Tables Generator* [9] along with *Excel* for some initial work. This generator has helped to understand tables logic, but later in the writing most of tables that appear in the document are written directly in *LaTex.*

It was necessary to include the next info regarding the structure and content of the different chapters and sections, for their correct achievement:

- Planning and cost estimation
- Analysis and design
- Developing
- Results
- Conclusions and future work
- Bibliography

Having in mind that the present document has been divided into five chapters: Technical Proposal and Project Design Document (*PDD*), Project development, Results and Conclusions. Planning and estimating costs are included in the first chapter. Analysis and Design, is included in chapter two, and the rest of the sections correspond to those chapters of this report with the same name. Also bibliography has been added has a chapter at the end of the present document.

A slight test phase has also been carried out to obtain some data reflected in Chapter 4.

Additionally it is necessary to emphasize that all the content of this report is authored by *Pablo Lorite Lozano* and external contributions have been duly cited and reported. It should also be noted that all the images, tables and diagrams that appear in this memory have been created exclusively for this project, except Figure 2.5

The fact of not including excessive images or codes lines has also been taken into account to cite only those necessary for a better understanding of the present document. For additional images and code snippets, two final appendices have been added, Resources and Source Code so that can be used to go and consult the material.

3.8 Preparation of the End-of-degree project presentation

For the final presentation for the tribunal, a general review of the tree generator run-time process has been prepared. Beginning by describing the project functionalities that has been created and passing by all the stages that have been carried out for the realization of the work: objectives, modeling techniques, deviations and solutions are the main issues to address.

There is a predilection for showing the behavior of the generator in the presentation itself, but if this is not possible for technical reasons, a sample video has been prepared where the generator is shown running.

3.9 Problems and deviations

The initial problems and deviations and the solutions implemented are detailed below. It will be detailed according to the subsystem to which each problem correspond.

- Root generation deviations: In the initial stage of development, the project started from the generation of the roots, initially roots were going to be treated as parametric curves through the use of *Bezier* curves in *Blender*, but when the utilization of vertices and edges together with the *Skin* modifier was found, this idea was discarded since the calculation using the *Bezier* curves had two times more parameters to control than with vertex structures.

  After this, the main headache appeared with the generation of a high number of roots and the consequent continuous visualization errors in the mesh. Once it was understood that the failure came from the internal calculations of the envelope radius from the *Skin* modifier, the solution was to apply an smoothing operation over the vertices to relax the distribution.

- Trunk generation deviations: The main design modification applied in the middle of the project development process corresponding to the trunk module was related to the growth behavior of the tree. In first instance, the growth of the trunk and consequently of the tree only worked by increasing or decreasing the value of the position of all the vertices that make up the trunk on the z axis. It was observed that the growth for a trunk did not vary much, but when adding more trunks the growth seemed unreal and not very organic.

  To solve this, a dynamic smoothing system was added that acted differently depending on the position of each vertex in the trunk, not as initially where the height value was applied equally to all vertices of the trunk.

- Branches generation deviations: This is the subsystem that has suffered the least deviations from the original design. The only decision discarded from the initial design was the possibility that a branch node could be strongly connected, so that many branches would come out of it. This idea was discarded since the point in which the idea of the use of cardboard technology in the leaves was also discarded, this type of generation of branches would cause many visualization problems. Even without concerning about visualization problems when generating a leaf for each branch, already generating only high number of branches, this idea is not suitable. This is because *Skin* modifier don't allow that there is more than one vertex acting as root vertex in the mesh, which is the parameter that gives the option for a vertex to be strongly connected with many others keeping a smooth appearance. Most important problem with branch generation its the gap or deformations that occur when the branches are generated in some points of the mesh, it has been minimized with the smoothing of the position of the vertices but it can continue to occur with certain values of the system parameters at some points in the execution, this is due to internal calculations of the *Skin* modifier.

- Leaves generation deviations: For the creation of the leaves, tests were carried out with meta balls, which at first were used to achieve an interaction between nearby leaves, but this was ruled out due to the irregularity in their topology when mapping possible textures, and because of their high computational cost.

  Cardboard technique was also taken into account [10], even though it is a technique that acts in a complementary way on an already created volume to give the appearance that this volume is formed by smaller leaves and consequently obtain a more realistic appearance. However the use of this technique requires the texturing of the sheets that are going to be used in each cardboard, and since the generator only produces an untextured mesh as the final result, with the aim of providing a blank canvas for each user to texturing as desired.

  Instead, it was decided to use icospheres that, together with the *Re-mesh* modifier and the use of *Boolean* operations, were closer to the aesthetics sought.

  Finally, there were problems that affected at the time of programming, in part due to the way *Blender* works. *Blender* has two modes, the *Edit* mode and the *Object* mode [3], in the *Edit* mode it is possible to access the different components of the mesh such as vertices, edges and faces, and in the *Object* mode transformations are applied to the corresponding data block to the object. The problem arose at the time of handling the vertices, since there was no way to store them globally to refer to them at any point in the execution since when changing from one mode to another the memory address where the reference to said vertex was stored disappears. The edit mode is where most of the system calculations are carried out, however it is necessary to change to the object mode to apply the modifiers on the object or for example to modify vertices. The solution found was the use of the *Bmesh* API that allowed to create a reference to the geometry of the mesh of the object and this allowed to make use of a static structure throughout the development of the project [2].

CHAPTER 4

RESULTS

Contents

All procedural modeling techniques require algorithms to manage and store data, so in this project there are two main critical aspects of evaluation. Data storage corresponds to user inputs parameters system to control the procedural generation and the management part of the system data corresponds to procedural modeling techniques itself.

Most importantly is the visual output, arising after these two techniques converge in a final result, which will be detailed in this chapter. Another features analyzed are the performance of the system and the usability of the system. Finally, the result of the work carried out will be compared with the initial implementation objectives.

4.1 Results and outputs

The final result of the project is a tool for *Blender* that generates tree meshes using procedural modeling techniques such as those described in the previous chapter. The results produced are previously optimized for their potential use in any type of visual media, although originally these results are focused on their use in a game engine, their use is versatile since it can be used as a 3D model in numerous fields of application, or as an image if the render result is used.

Some outputs produced by the generator are provided in the Resources appendix (see Append A).

The following link allows to access to a *Google Drive* folder where the generator in the original written program in *Python* language can be seen and downloaded for it use.

4.2 Performance

It must be reasonable to think that the system is capable of being used as a tool that could be used in a workflow for someone producing content for games or art. For this reason, it is necessary that the performance of the generator is suitable for use in professional environments. This is followed by an analysis of performance main data.

4.2.1 Data collection

Aiming to analyze system performance requires collecting data referent to the time it takes for the generator to produce an output. For this purpose, multiple instances of the main process have been started and one by one the interface parameters have been modified. In the first place, it was found that after a certain number of active instances, the execution time increases exponentially.

This served unintentionally to detect an error in the *Blender* scene garbage collector, because although at the beginning of each iteration the program removes all the existing meshes in the scene, there is data corresponding to the geometry of the deleted mesh that is kept stored in cache memory. Since it is an intrinsic failure of *Blender* that only appears after doing a stress test on the system with numerous iterations, this calculation has simply been ignored and the data has been added in the data table until the slowdown of system happens [7].

4.2.2 Generation time

Tree generator is able to produce any mesh in order of seconds, with an upper limit of four seconds for the maximum amount of calculations at worst case scenario. As the project has been created allowing user modification, this makes it very difficult to analyze the performance stats in a uniform way because the system is modified by user inputs which are completely unpredictable.

For this reason five tests have been created with five different generator configurations, ranging from a configuration with a fewest number of calculations to the maximum number of calculations(see Table 4.1).

Tests allows to see the failure in the garbage collector of the *Blender* scene mentioned previously, because in each iteration every step with the same computational complexity, costs more time to perform (see Table 4.2).

It is important to bear in mind that the main subsystems that cover the greatest number of calculations and therefore those that constitute the two most important bottlenecks are the generation of branches and the generation of leaves. Branch generation

Variables	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**
Number of roots	16	16	16	16	16
Number of trunks	1	2	4	4	4
Number of branches	1	8	20	48	48
Number of leaves	0	2	15	48	48
Randomness	No	No	No	No	Yes

Table 4.1: Test Configurations

Execution time(ms)	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**
Number of roots	20ms	40 ms	65ms	80ms	85ms
Number of trunks	5ms	70 ms	110ms	150ms	170ms
Number of branches	83ms	90 ms	400ms	800ms	1.000ms
Number of leaves	0	240 ms	2.300ms	5.500ms	7.800ms
Randomness	0	0	0	0	13.000ms
Total	108ms	440ms	2.875ms	6.530ms	22.055ms

Table 4.2: Performances times detailed

time grows exponentially along with number of branches of the mesh, just like leaves generation time also grows exponentially along with the number of leaves.

This amount of time is negligible compared to the amount of time used by a modeler to create a mesh with similar features by hand. At this point, it can be noted that the performance of the system meets more than the expected quality for this type of tree generators.

It is difficult to reach an absolute decision on the generation times and to find out which of the established systems is faster, since there are many different implementations. Studying the canonical implementations of each system to adjust to the established norm in terms of performance of each implementation, if we used fractal structures, the generation time would be the longest, however if we used L-systems or parametric systems the execution time is significantly reduced. These times are the most similar to those of the implementation followed in the project. The implementation described in this project has generation times almost 5 times lower, because it also uses a simpler logic, since it is not intended to emit a realistic model and generation time depends directly on the complexity of the mesh to be produced.

Tests in this section have been performed on a system with the following technical specifications.

- Operative System: Windows 7 Home Premium 64 bits
- Ram: 8 GB
- Processor: Intel(R) Core(TM) i5-3330 CPU @ 3.00Ghz
- Graphic card: Nvidia Geforce GTX 650 Ti BOOST

- DirectX Version: DirectX 11

4.2.3 Mesh complexity

Another aspect that affects the performance of the system is the complexity in the geometry of the resulting meshes. It is true that most existing tree generation models produce a tree that usually have more complex topologies, but this was not the objective of this project, since it seeks to produce a mesh as simple as possible so that a high number of calculations to modify the position, orientation or scale of its geometry data, does not have to be done and allow to decrease execution time.

Ideally, it is wanted to produce the least complex model possible to get a good visual result. Compared with other organic generation systems, such as fractals, the L system, or a modular generation system, all of them will produce less predictable results, thus offering a wider range of possible outputs, but these have a complexity in the mesh much larger than that produced by the generator described in the present document.

4.3 Usability

Finally the usability of the system is also analysed, it must be accessible to any inexperienced user who may have limited knowledge of programming concepts or 3D modeling.

To do this, an interface has been created that brings together seventeen parameters, a significantly lower number than most of the parameters found in most parametric tree-generation implementations. So it has been possible to increase simplicity and reduce the parameters with which the user interacts. These parameters are made up of numerical attributes within a given range, which are ordered by the subsystem they affect within the interface menu.

The user only has to know the structure of a tree to know for example that the trunk is the central element of it. This was designed precisely so that anyone without experience can test the generator and experience all the possible results since it is not necessary meet any additional requirements to generate any modules. Specifically, since the trunk is part of the base mesh and is always active, the user can place or remove the leaves without having to fulfill any prerequisite, or can add leaves and branches since the only element necessary for this is the trunk, and It is always present, in fact leaves can be generated even without having branches.

Also, the parameters are clear and explicitly describe their function. They are intuitive and allow you to clearly see the effect of changing a parameter in the interface and its reaction in the mesh. The generator allows you to design a new type of tree in just a few minutes. Currently, the input of all parameters values are defined by using numerical values and the user can interact through the integrated slide in the interface or by typing the parameter value directly with the keyboard.

4.4 Project numbers

Some important data regarding the development of the code and technical characteristics of the project implementation are indicated below (see Tables 4.3, 4.4).

Module	**Maximum Quantity**	**Number of vertices**
Roots	16	16*3=48
Trunks	4	4*3+3=15
Branches	12*4=48	8*4*4+4*4=144
Leaves	12*4=48	Use a primitive of approx 30*8*4+30*4=1080
Total	116	1287

Table 4.3: Maximum data numbers.

Class	**Number of methods**	**Number of variables**	**Lines of code**
Singleton (Main Process)	25	150	1458
Operator (Input system)	20	100	202
Total	45	250	1660

Table 4.4: Summary of the main classes implementation numbers.

4.5 Miles achieved

In this section the results of the work are compared against initial objectives.

- Goal 1 "Carry out the proposal of a fully working plugin, and in addition eventually, it could continue to improve aspects of it and adding new ones."

  A fully functional plugin for *Blender* has been achieved, which produces tree meshes in a procedural way, material added (see Append A) verifies the fulfillment of this objective. Of course there is much room to improve and expand its functionality in the future, for example procedural texturing could be added or a greater variety of parameters that add more variety to the distinct outputs such as the subdivision of branches, or the possibility of generate individual leaves.

  It is also important to note that the appearance of the mesh can be improved so that it is smoother and appears with fewer imperfections, although most of these problems are derived from the use of skin modifier, it is important to highlight it.

- Goal 2 " To create an attractive visual style for an in game model and extend it along every output produced by the user."

  The particular style of the trees is minimalist and has a smooth surface. After the first approach to generation using *Bezier* curves, a homogeneous appearance between all parts of the tree was not achieved. Once the *Skin* modifier was found, it was possible that the entire mesh had a uniform surface. To achieve this stylized appearance of the tree, the smoothing of the vertices was the chosen method. Finally application of the modifier *Subdivision surface* play a main role allowing the user to choose the level of detail they want for the final mesh. The visual style achieved is almost identical to that proposed in the initial concepts so it could be considered that this objective has been fulfilled.

- Goal 3 "Obtain a procedural generation system that follows a coherent and consistent logical design."

  It has been possible to create a procedural generation system through an initial division of space, which gives a modular structure to the mesh, then each module follows a series of procedural and random instructions that are modified at each intervention by the user, producing different outputs every time without errors. It is a fact that if at the time of producing the mesh there are no execution errors is due to the fact that the logical design followed is consistent within the system.

- Goal 4 "To make every tree mesh optimized and consistent."

  A coherent logical design has been achieved for the representation of the trees, it has also been possible to generate meshes of a very simple complexity and it has also been achieved the fact that the mesh is generated with the least visual errors possible, however generator still produce some outputs that presents any holes or discontinuities in the mesh when a high number of roots or branches is introduced, this is due to an intrinsic calculation of *Blender Skin* modifier and is usually fixed by minimally modifying any other parameter of the interface to allow Blender to recalculate the skin. Although it is true that the majority of outputs produced with this type of discontinuity have been eliminated, it has not been possible to completely reduce this type of failures.

- Goal 5 "To ensure the creation of a large mesh range."

  The results obtained are satisfactory knowing that the characteristics of the mesh are configurable by the user. This allows you to cover a wider range when designing a set of trees that are different but maintain the same overall appearance. Although it is true that the appearance could have been varied more if additional parameters had been added such as subdivision of branches or leaves created using different primitives.

  A comparison with the Figure 2.5 is detailed below to show the range of meshes obtained in Figure 4.1.

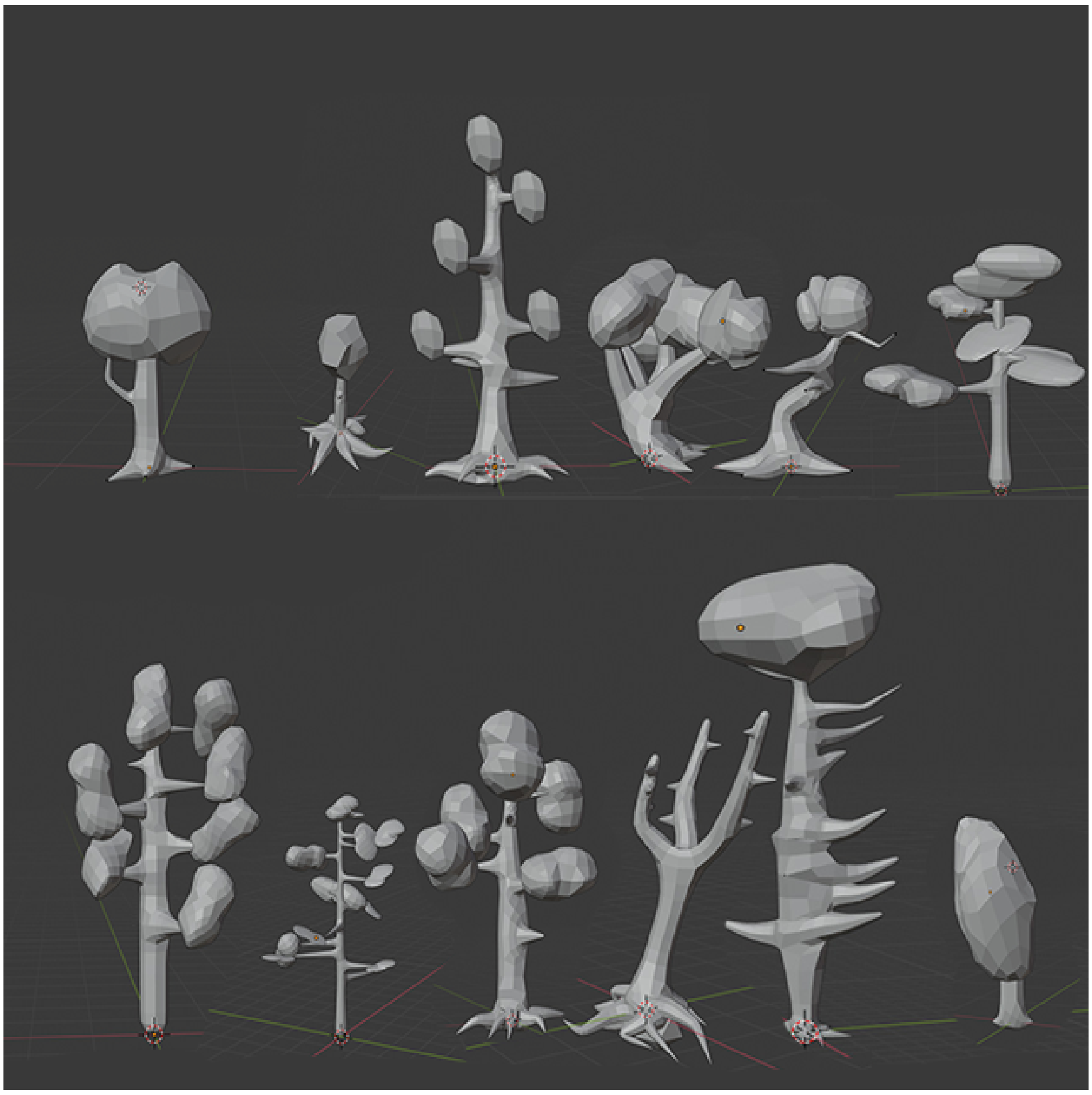


Figure 4.1: Comparison of results against initial tree concepts.

CHAPTER 5

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

It has been hard work, especially at the beginning when I was only clear that I wanted to carry out a project that used procedural techniques, but I didn't even know what to do. The beginning of the implementation in *Blender* was difficult because the variables system had to be created and it was necessary to make sure that its storage and management were correct.

The next headache was choosing the type of geometric representation used for the mesh. Once it was decided to use vertices and edges together with the *Skin* modifier, the implementation began to advance a little more. The next turning point was the understanding of the behavior of the *Blender* scripting system, which I learned from mistakes that it only works correctly if the exact sequence of actions is followed. Once it have been understood *Blender's* restrictions such as the importance of the order of events when writing in *Python.* development began to be more fruitful.

During the development of the project I have been with a part-time job, and working as a freelance since the beginning of the course, however I believe that I have been able to carry out an implementation more than worthy of the initial objectives I had in mind.

I realized that I had to choose a topic that I was really passionate about in order to be able to want to finish it, I also realized that the more effort I put into the project, the more likely it would be to have a great job to add to the curriculum, there was also the possibility of using it as a tool in the future and even marketing it.

5.2 Future work

Work should continue to achieve greater variation in the results produced. The main lines of improvement detected are the following:

- Procedural texturing: a color selection parameter could be added for the modules corresponding to the trunk roots and branches and another color selector for the leaves, in addition to this it would be necessary to implement a noise function to give a more organic surface appearance. And based on these noise and color parameters, generate procedural textures.
- Greater variety of leaves: different primitives could be used to create other types of leaves such as conical, similar to the structure of a Christmas tree or simply surfaces to which a noise texture can also be applied to make it appear more like realistic leaves. The possibility of generating many multi-node leaves from multiples branches similar to the structure of a palm tree could also be implemented.
- Adaptation to possible bushes or other types of plants: it would be possible to adapt the system with relative ease to also produce shrubs and plants that follow a clear modular structure.
- Flowering parameters: this subsystem could be added and the system could generate shapes that could be used as leaves and petals.
- Improve actual outputs: outputs generated by the system can be improved to increase their randomness and variety and to produce outputs with the least possible number of visual errors in the mesh.

5.3 Personal reflection

At the end of the project, the feeling has been of a quite enriching as well as stressful experience. However the overall feeling is that of a quite positive experience that has allowed me to establish my knowledge about geometry, the behavior of different techniques of automatic and procedural modeling, and the way *Blender* works. Also now seeing the amount of excellent level tools in terms of plant generation, it makes me continue to want to improve the project, to see how far could it go.

Bibliography

[1] Blender. About blender. https://www.blender.org/about/. Accessed: 2021-02-10.

[2] Blender. Bmesh module. https://docs.blender.org/api/current/bmesh.html. Accessed: 2019-02-28.

[3] Blender. Object modes. https://docs.blender.org/manual/en/dev/editors/3dview/modes.html. Accessed: 2019-02-28.

[4] Blender. Remesh modifier. https://docs.blender.org/manual/en/latest/modeling/modifiers /generate/remesh.html. Accessed: 2021-02-10.

[5] Blender. Skin modifier-blender. https://docs.blender.org/manual/en/latest/modeling /modifiers/generate/skin.html. Accessed: 2019-02-28.

[6] Blender. Subdivision surface modifier. https://docs.blender.org/manual/en/latest/modeling /modifiers/generate/subdivision$_surface.html. Accessed : 2021 - 02 - 10$.

[7] Blenderartist.org. Force garbage collection? https://blenderartists.org/t/force-garbage-collection/674618. Accessed: 2021-02-10.

[8] Reyes; Nebot Romero Victoria Campos Sancho, Cristina; Grangel Seguer. Fonaments d'enginyeria del programari. http://repositori.uji.es/xmlui/handle/10234/167532. Accessed: 2021-02-10.

[9] Tables Generator. Tables generator. https://www.tablesgenerator.com/. Accessed: 2021-02-10.

[10] Markom3d. How to create a leaf texture in blender - eevee. https://www.youtube.com/watch?v=TrfIRDw88bM. Accessed: 2021-02-10.

[11] Jake Morrison. Jake morrison instagram profile. https://www.instagram.com/jakemorrisonart/. Accessed: 2021-02-10.

[12] Overleaf. About overleaf an latex. https://www.overleaf.com/about. Accessed: 2021-02-10.

[13] Overleaf. Overleaf: Real-time collaborative writing and publishing tools with integrated pdf preview. https://www.overleaf.com/. Accessed: 2021-02-10.

[14] Python. Generate pseudo-random numbers. https://docs.python.org/3/library/random.html. Accessed: 2019-02-28.

[15] SpeedTree. The standard for vegetation modeling and middleware. https://store.speedtree.com/. Accessed: 2021-02-10.

[16] Unity. “unity asset store”. https://assetstore.unity.com/3d. Accessed: 2021-02-10.

[17] Jason Weber and Joseph Penn. “creation and rendering of realistic trees”. in: Siggraph ’95. acm inc., sept. 1995, pp. 119–128. http://www. cs . duke . edu / courses / cps124 / spring08 / assign / 07 $_papers/p119 - weber.pdf. Accessed : 2021 - 02 - 10$.

[18] Wikipedia. Functional requirements. http://en.wikipedia.org/wiki/Functional_requirements. Accessed: 2019-02-28.

[19] Wikipedia. Geodesic polyhedron. https://en.wikipedia.org/wiki/Geodesic$_polyhedron.Accessed : 2019 - 02 - 28$.

[20] Wikipedia. Non-functional requirements. http://en.wikipedia.org/wiki/Non-functional_requirement. Accessed: 2019-02-28.

[21] Wikipedia. Procedural modeling. https://en.wikipedia.org/wiki/Procedural$_modeling.Accessed : 2019 - 02 - 28$.

[22] Wikipedia. Sacred geometry. https://es.wikipedia.org/wiki/Geometría$_sagrada.Accessed : 2019 - 02 - 28$.

[23] Wikipedia. Singleton. https://en.wikipedia.org/wiki/Singleton. Accessed: 2019-02-28.

[24] Wikipedia. Tropism. https://en.wikipedia.org/wiki/Tropism. Accessed: 2019-02-28.

Appendix A

Resources

Graphical information, source code of the project and an installation manual of the generator is presented in this append.

This information can be accessed through this link:

https://drive.google.com/drive/folders/1xZHFwckDikWdBaASed2MEy-utgmKgXWj?usp=sharing.

Appendix B
Source code

This appendix shows the main code snippets cited in the document. The complete code of the project can be consulted in the previous appendix.

Singleton

```
#Slot declaration, Singleton constructor, instance creation and parameter value modification.
class Singleton:
        __slots__ = (
            "init_width",
            "initwidthvalue",
                 . . .
        )

        def __init__(self):
            self.init_width=False
            self.initwidthvalue=1
                  . . .

state = Singleton()
del Singleton
state.init_width=True
```

Operator parameter

```
#Parameter initialization.
initwidth = FloatProperty(
       name="Initial_width_of_the_trunk",
       default=1.0,
       min=0.1,
       max=10.0,
       description="Initial_width_of_the_trunk",
       update=initwidth_func
       )
```

Trunk generation axis selection

```
#Extract of the trunk axis selection system.
if(len(state.trunkrandomfinalvec)>0.0 and state.trunkheightvalue>0.65):
        j=0

        for i in range (state.numberoftrunks):
            decreasefactor=1

            if(state.trunkheightvalue>1.99):
               decreasefactor=1

            if(state.trunkheightvalue>6):
               decreasefactor=1.2

            trunkheightvar=state.trunkheightvalue/decreasefactor
            selectVertex(bpy.context, 1)
            trunkaux.vertex_groups.active = trunkaux.vertex_groups[0]

            if(state.trunkrandomlist[i]=="-x"):
                bpy.ops.mesh.extrude_region_move(MESH_OT_extrude_region={"use_normal_flip":False, "mirror":False},
                TRANSFORM_OT_translate={"value":(-state.trandomvalue*trunkheightvar,
                (state.trunkrandomfinalvec[j+1]*trunkheightvar/2)/2, trunkheightvar))
            elif(state.trunkrandomlist[i]=="x"):
                bpy.ops.mesh.extrude_region_move(MESH_OT_extrude_region={"use_normal_flip":False, "mirror":False},
                TRANSFORM_OT_translate={"value":(state.trandomvalue*trunkheightvar,
                (state.trunkrandomfinalvec[j+1]*trunkheightvar/2)/2, trunkheightvar))
            elif(state.trunkrandomlist[i]=="y"):
                bpy.ops.mesh.extrude_region_move(MESH_OT_extrude_region={"use_normal_flip":False, "mirror":False},
                TRANSFORM_OT_translate={"value":((state.trunkrandomfinalvec[j]*trunkheightvar/2)/2,
                state.trandomvalue*trunkheightvar, trunkheightvar))
            elif(state.trunkrandomlist[i]=="-y"):
                bpy.ops.mesh.extrude_region_move(MESH_OT_extrude_region={"use_normal_flip":False, "mirror":False},
                TRANSFORM_OT_translate={"value":((state.trunkrandomfinalvec[j]*trunkheightvar/2)/2,
                -state.trandomvalue*trunkheightvar, trunkheightvar))

            bpy.ops.transform.translate(value=(state.trunkrandomfinalvec[j],state.trunkrandomfinalvec[j+1],
            state.trunkrandomfinalvec[j+2]))
            bpy.ops.transform.skin_resize(value=(state.finalwidthvalue, state.finalwidthvalue, state.finalwidthvalue))

            j=j+3
```

Number of roots calculation

```
#Root quantity initialization.
 if (state.root_quantity):

        state.rootqvaluex=[]
        state.rootqvaluey=[]
        state.rootquantity=self.rquantity
        state.randomvaluevec=[]
        state.randomvaluefinalvec=[]

        if state.rootquantity==0.0:
            state.rootcount=0
        elif state.rootquantity<=10.0:
            state.rootcount=random.randint(2, 3)
        elif state.rootquantity<=20:
            state.rootcount=random.randint(4, 5)
        elif state.rootquantity<=30:
            state.rootcount=random.randint(5, 6)
        elif state.rootquantity<=40:
            state.rootcount=random.randint(6, 7)
        elif state.rootquantity<=50:
            state.rootcount=random.randint(7, 8)
        elif state.rootquantity<=60:
            state.rootcount=random.randint(8, 9)
        elif state.rootquantity<=70:
            state.rootcount=random.randint(9, 10)
        elif state.rootquantity<=80:
            state.rootcount=random.randint(10, 12)
        elif state.rootquantity<=90:
            state.rootcount=random.randint(12, 14)
        elif state.rootquantity<=100:
            state.rootcount=random.randint(14, 16)
```

Branch quantity calculation

```
#Branch number calculation.
        bp = state.branchcount
        i=3
        j=2
        iter=0

        nsubdvlist= [1,2,3,4]
        factor=4
        itercount=state.numberoftrunks
        minus=itercount

        if (itercount-minus==0):
            minus=minus-1
            while (i>0 and state.branchquantityvalue>0):
                n_subd=random.choice(nsubdvlist)
                if n_subd<=bp:
                    if bp-n_subd<=factor*j:
                        state.branchlist.append(n_subd)
                        bp-=n_subd
                        i-=1
                        j-=1
                        iter+=1
                        if bp==0:
                            break
```