Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Ingeniería Informática

Proyecto Final

Simulación de Comportamiento Orgánico mediante un Sistema Procedural de Generación, Animación y Transformación.

Autor del proyecto: José Elipe Ibáñez

Director del proyecto: Daniel Blasco Latorre

Zaragoza, 23 de junio de 2020

Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Ingeniería Informática por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

Firma

Fecha

23 de junio de 2020

Dedicatoria y Agradecimiento

Quiero agradecer a mis compañero/as de la universidad por mostrar su capacidad para mejorar sin olvidar nunca a las personas que se encuentran a su alrededor. En especial, quiero agradecer a Cristian Qiu la ayuda prestada al comienzo de este proyecto.

Quiero agradecer a mis amigo/as por permanecer a mi lado incluso en los momentos mas complicados. No existen palabras para expresar lo agradecido que estoy de haberos conocido.

Quiero agradecer a mis familiares más cercanos por el cariño con el que han velado por mí todos estos años. Gracias por hacerme sentir en casa.

Y, en especial, quiero agradecer este trabajo a este maravilloso mundo, ya que este trabajo solo pretende mostrar la subyacente belleza que se esconde en cada palmo de tierra.

Table of Contents

Table of Figures

Table of Tables

Table of Abbreviations and Acronyms

PCG: Procedural Content Generation.

RNG: Random Number Generator.

Heightmap: 2D texture that represents the height of a surface in grayscale values.

CW: Clockwise-Winding.

CCW: Counter-Clockwise Winding.

Topology: The shape of the model, as only the arrange of the three-dimensional vertices.

Primitives: Basic geometric topologies resembling a regular polyhedron.

Mesh: Class that stores all the three-dimensional data of an object.

LOD: Level Of Detail.

Resumen

La creación de objecto orgánicos mediante computación es compleja por el componente técnico / matemático y por el gran número de métodos disponibles. La creación procedural de contenido (PCG) es un campo que involucra el uso de algoritmos para modelar estructuras de datos, como alternativa a la creación manual. PCG es de gran utilidad para crear de manera automática grandes cantidades de contenido en aplicaciones.

El objetivo de este trabajo es estudiar, comparar y realizar métodos para poder crear, animar y transformar diferentes objetos para crear un escenario en tres dimensiones. Esto se habrá logrado usando marcos de trabajo comerciales y comparando los métodos en escenas de prueba.

Los resultados de este trabajo incluyen estadísticas que nos muestran como cada uno de los métodos estudiados producen diferentes resultados. Estos resultados están clasificados según sus propiedades, diseño y usos posibles en otras áreas. El propósito de generar estos resultados es proveer una mejor perspectiva y un punto de entrada para futuras investigaciones y trabajos comerciales centrados en PCG.

Abstract

Creation of organic objects through computation is complex for its technical / mathematical component and the wide range of methods available. Procedural Generated Content (PCG) is a field that involves the use of algorithms to model data structures, as an alternative to manual creation. PCG is useful for automatically creating large amounts of content in applications.

The objective of this work is to study, compare and develop methods to generate, animate and transform various types of objects in order to create a three-dimensional environment. This is done by using commercial frameworks and comparing the methods in test scenes.

The results of this work include statistics that show how each of the methods studied produces different outcomes. Those results are classified according to the properties, design, and possible uses in other areas of such methods. The purpose of generating these results is to provide a better perspective and a starting point for future research and commercial works that are focused on PCG.

1. Introduction

Procedural Content Generation (PCG) has different definitions. Conceptually, the broadest definition is given by 'What is Procedural Content Generation? Mario on the borderline': "Procedural content generation in games refers to the creation of game content automatically using algorithms" [1].

We understand that every type of content made by an algorithm in a videogame can enter in this explanation. Therefore, Jonas Freiknecht and Wolfang Effelsberg make a more specific definition: "Procedural content generation is the automatic creation of digital assets for games, simulations or movies based on predefined algorithms and patterns that require a minimal user input" [2].

We will use this definition as our implementation will need some end-user input. Although it is true that PCG can work without any type of input by the end-user, in our implementation we give the possibility of changing parameters in order to create specific environments.

It is also implied in the definition that PCG does not belong only to the videogames field. We can find examples of different media using PCG to generate a large amount of content. In cinema, programs such as 'MASSIVE' [3] allows the creation of multiple agents without doing it manually. The first use of this application was with Peter Jackson's *The Lord of the Rings: The Return of the King* (2003), generating armies with thousands of soldiers.

Figure 1: Army made with MASSIVE (*The Lord of the Rings: The Return of the King*, 2003).

In videogames, the first use of PCG are games with great inspiration on tabletop role games, such as Dungeons & Dragons. The first game that used this approach was *Beneath Apple Manor* (1978), followed by *Rogue* (1980), which popularized the genre and stablished it as 'Roguelike'. These first games were procedural dungeons made with ASCII symbols, generating different dungeons every time you play.

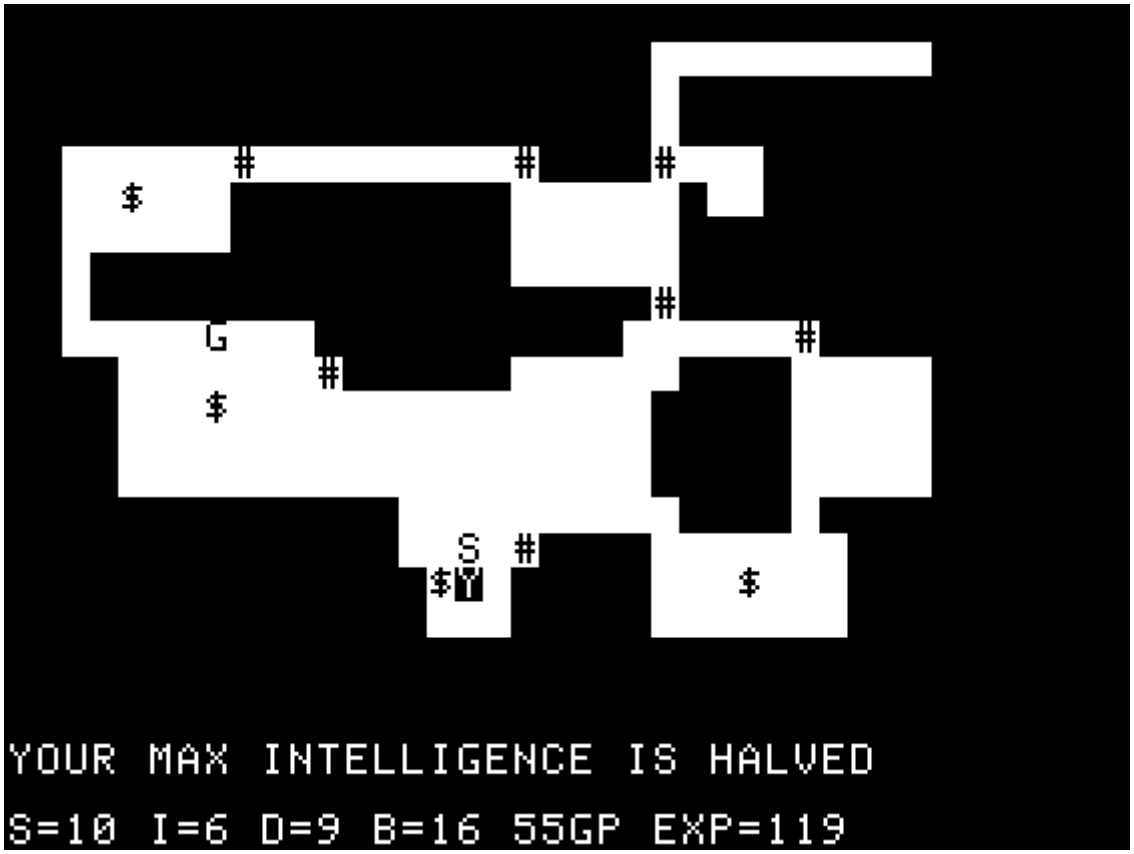

Figure 2: Procedural Dungeon with ASCII symbols (*Beneath Apple Minor*, 1978).

The same publication of Jonas Freiknecht and Wolfang Effelsberg show us that this technique could also be applied to modelling. *The Sims* (2000) showed a character editor where you could introduce parameters in order to change the appearance of a 3D model splitted into different parts, such as clothes, hair, body…

Later, a first-person shooter called *.Kkrieger* (2004) was uniquely created with procedural methods. The models of the game are primitives such as cubes and cylinders, which are generated procedurally in the execution time or after the start of the game. The total size of the videogame is 97,280 bytes, making them the winners of the 96k game competition at Breakpoint in April 2004.

Figure 3: Procedural models starting from basic primitives.

Then, PCG started expanding as a concept in videogames. *Spore* (2008) makes use of a system to create models of creatures in real time with their animation system. *Dwarf Fortress* (2006), *Minecraft* (2011) or *Proteus* (2013) makes use of procedural generation in order to create more organic environments. The most modern use of PCG is *No Man´s Sky* (2016) where the procedural algorithm allows to create over 1,8 x 10^{18} planets, with flora, fauna and alien species.

Figure 4: Planets and creatures procedurally (*No Man´s Sky,* 2016)

In this essay we will demonstrate the multiple methods to introduce procedural content. We focus on three types of PCG: modifying procedurally the geometry of the object, distributing the objects through the environment and making the environment interactive. Figure 5 shows visually these objectives.

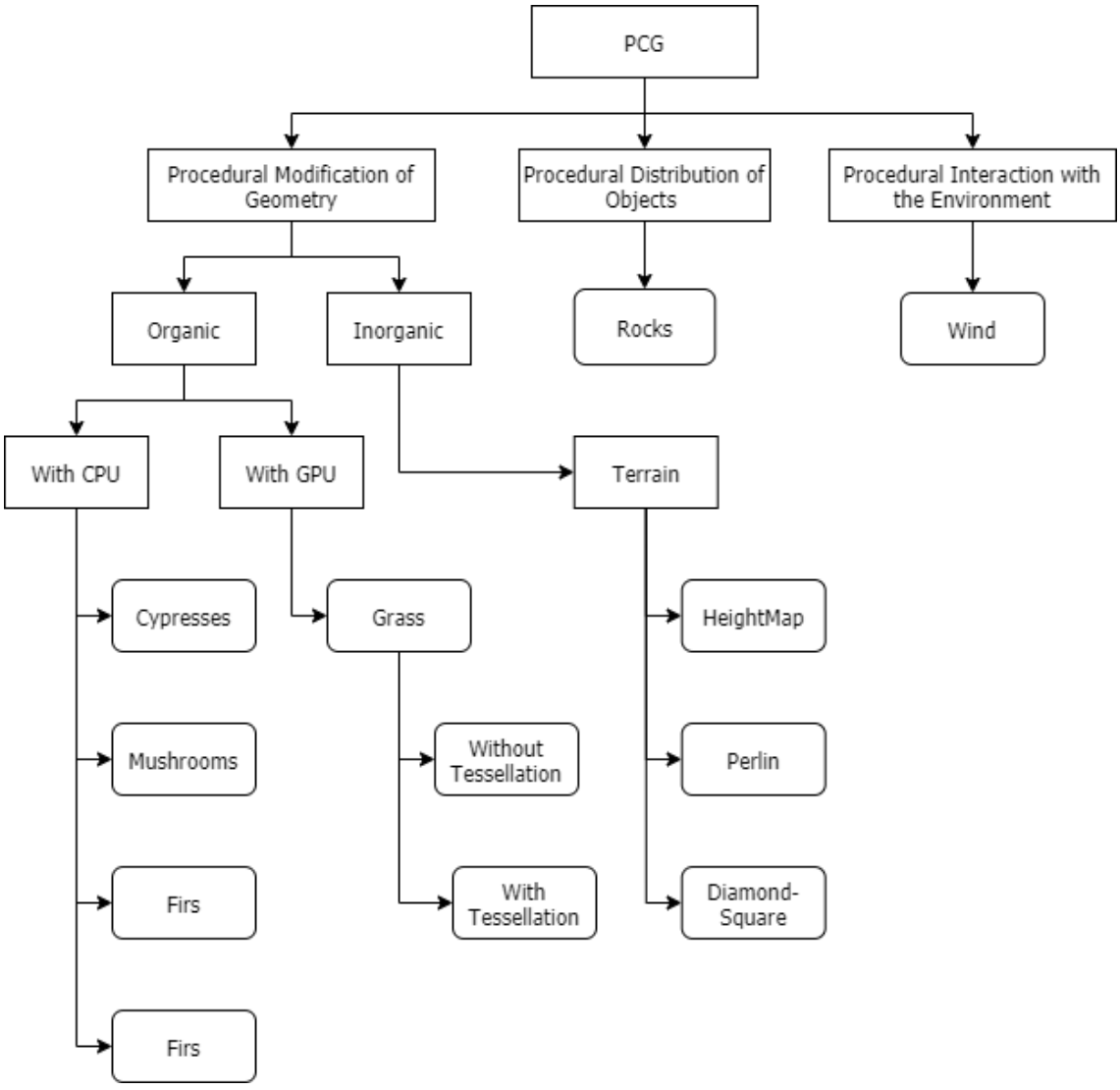

Figure 5: PCG objectives of this essay

We use Procedural Modification of Geometry to create organic and inorganic objects. We create one complex entity by CPU like vegetal life or we can create large amount of simple geometry by GPU like grass. We use Procedural Distribution to place objects in an already existing terrain, created through different methods. Finally, we add Procedural Interaction with the Environment to enable interaction between the end-user and the environment.

2. State of Art

In this section we are going to explain which algorithms we have used in order to complete the objectives of this essay. As seen in figure 5, these are: Procedural Modification of Organic Geometry, Procedural Modification of Inorganic Geometry, Procedural Distribution of Objects and Procedural Interaction with the Environment. This last one is not made by any PCG algorithm, so it will be explained in the development section.

2.1. Procedural Modification of Organic Geometry

2.1.1. *Geometry Mesh in Unity with CPU*

According to the definition given by Unity: "a mesh consists of triangles arranged in 3D space to create the impression of a solid object" [4]. Each triangle is defined by its three corners points or vertices. The vertices of the triangle are three-dimensional points with values in the XYZ coordinates, which are stored in the mesh as an array called "vertices" (VA).

Triangles are defined as integers and stored in the mesh as an array called "triangles" (TA). Each triangle is specified with three values that correspond to the VA. The size of the TA corresponds to the three times the number of triangles the mesh is going to have.

For example, in figure 6 we have defined the most basic mesh with three vertices and one triangle. Here, VA would be a Vector3 array of size three with positions $\mathbf{p_0}$ = (0.0f, 0.0f, 0.0f), $\mathbf{p_1}$ = (0.0f, 0.0f, 1.0f) and $\mathbf{p_2}$ = (1.0f, 0.0f, 1.0f). TA would be an integer array of size three with values 0, 1 and 2.

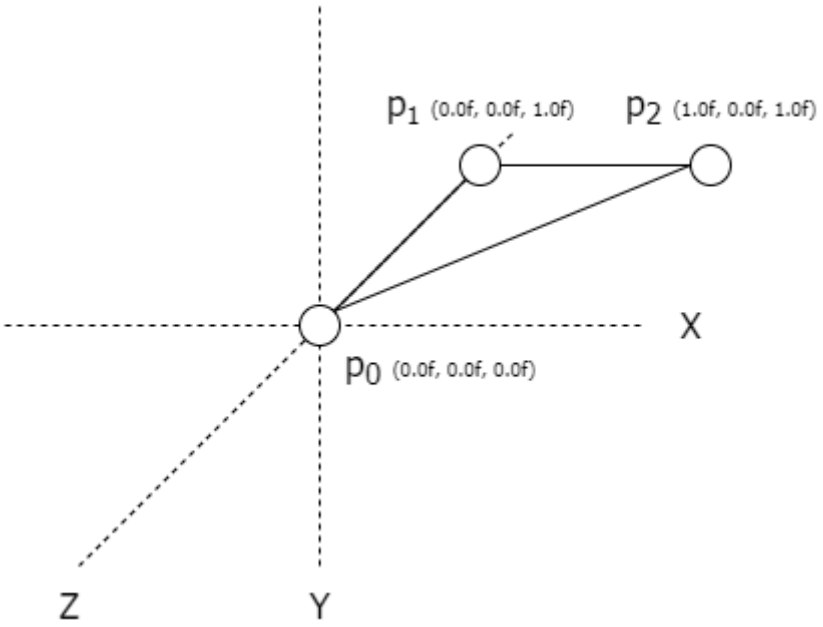

Figure 6: Example of a Mesh made by three vertices and one triangle

In order to create a triangle visible from outside, we must arrange the indexes of the triangle in a clockwise winding order (CW). The winding order can vary from one software to another. Unity uses CW for determining front-facing triangles. If we use CCW, we would be rendering the triangle making it visible from inside the mesh, as we can observe in figure 7.

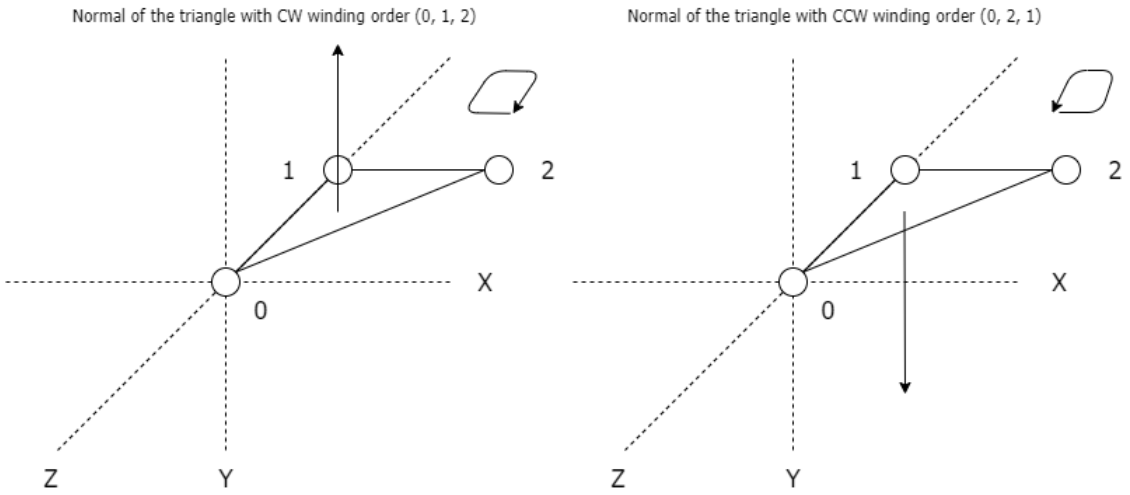

Figure 7: example of triangle using CW and CCW order

Although we have all the information to create a mesh, mostly we want to add more information to the mesh to render it better. If we want the mesh to be shaded, we must add normal vectors. Normal vectors are directional vectors that points outwards, perpendicular to a face of the mesh.

A face is an aggrupation of triangles with the same normal direction. Normal vectors are defined in the mesh as a Vector3 array called normals (NA).

Normals in Unity are assigned to the vertices and not to the triangles, making the size of the NA the same as VA, and not TA. Shading with normals per vertices is made with a calculation of three vertices of the NA to get a triangle's normal. This approach is not used by all three-dimensional software. For example, 'Blender' uses normals per triangles instead of per vertices.

Sebastian Lague made a visualization of these two approaches in his video *Procedural Landmass Generation (E12: normals)*. From 00:00 to 00:44, we can see in figure 8 normals being assigned to vertices as white vectors and normals being assigned to triangles as green vectors [5].

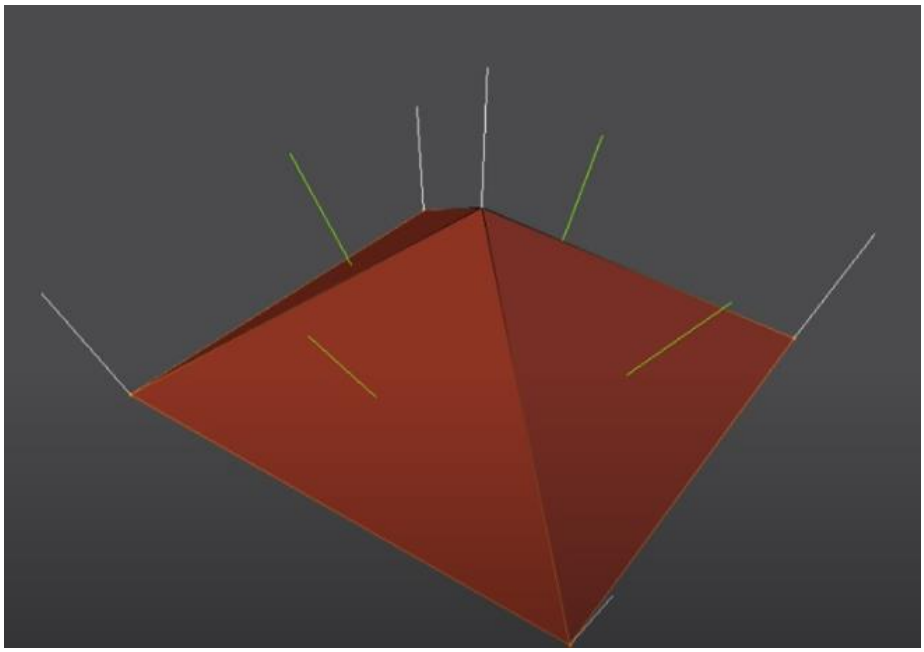

Figure 8: normals applied to vertices (White) VS normals applied to triangles (Green)

We can also texture the mesh procedurally. If we want to apply a texture to a mesh, we must define the texture coordinates into a Vector2 array called uv (UA). The UA has the same size as VA, because we are assigning the position of each vertex into a two-dimensional texture. The positions are limited between 0.0f and 1.0f in the X and Y axis. The origin position (0.0f, 0.0f) is at the bottom-left corner.

Vertices, triangles, normals and uv will be all the data that we will use in our essay. In figure 9 we are showing an example of a generated quad with four vertices (showed in blue), two triangles (separated by a black line), normals of each vertex (showed as yellow) and with a test texture.

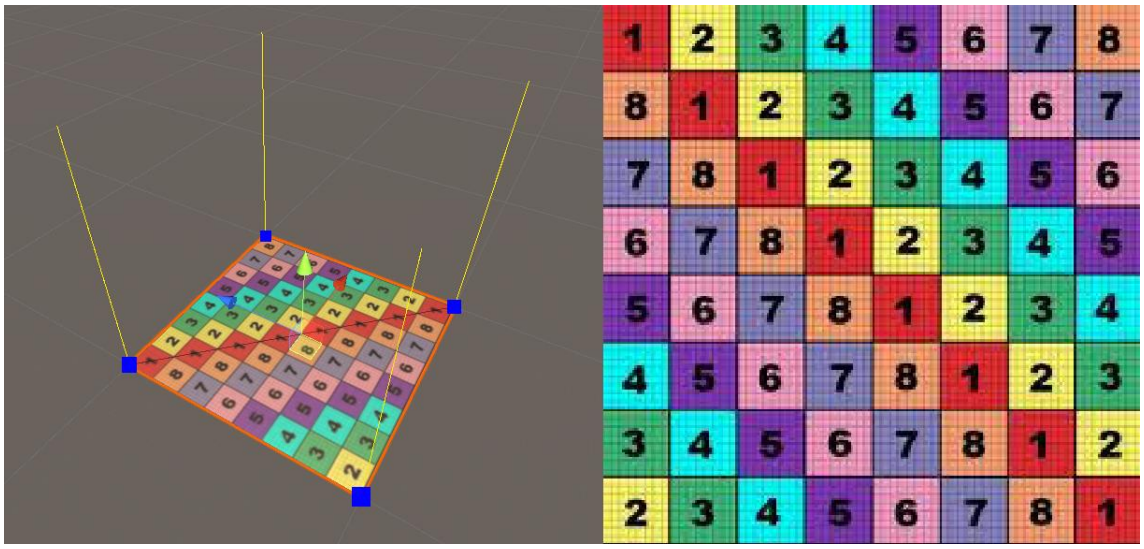

Figure 9: Basic mesh with vertices, triangles, normals and UVs (Left) VS Texture used (Right)

2.1.2. *Geometry Mesh in Unity with GPU*

We must describe first what is a Graphic Rendering Pipeline to understand the Graphic Processing Unit (GPU). We will use Chapter two of 'Real-Time Rendering: fourth edition' [6]. The Graphic Rendering Pipeline can be split into several stages, each of which performs a part into a larger task. The main function of the pipeline is to generate, or render, a two-dimensional image given a virtual camera, three-dimensional objects, light sources and more.

The architecture of the Graphic Rendering Pipeline consists of 4 parts: Application, Geometry Processing, Rasterization and Pixel Processing. Application is what it is executed in the CPU and the developer has full control of it. Geometry Processing stage is responsible for most of the per-triangle and per-vertex operations. Rasterization finds the pixels of the topology that corresponds to the previous transformed and projected vertices. Pixel processing is the stage where per-pixel or per-sample computations and operations are performed on pixels that are inside a primitive.

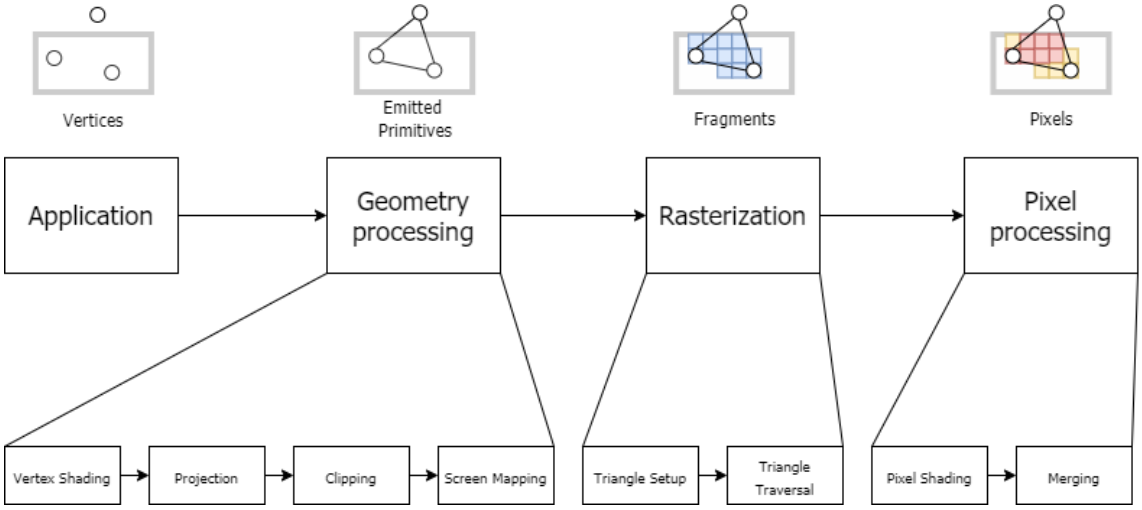

Figure 10: Architecture of a Graphic Rendering pipeline

The GPU implements the geometry processing, rasterization and pixel processing pipeline stages. These are divided into several hardware stages with varying degrees of programmability. In figure 11 we can observe which parts the user have control of it: green parts are fully programmable. Dashed lines are optional stages. Yellow stages are configurable but not fully programmable. Red stages are completely fixed in their functions.

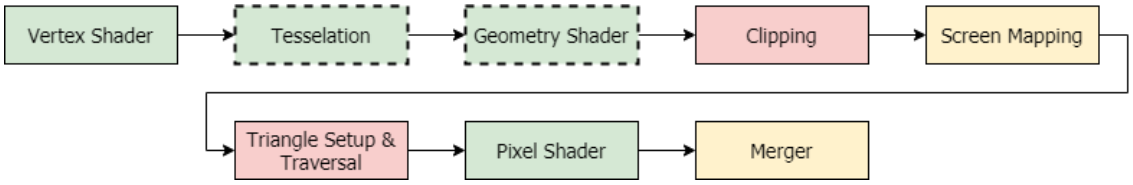

Figure 11: GPU implementation of the rendering pipeline

The vertex shader provides a way to modify, create or ignore values associated with each triangle's vertex, such as its colour, normal, texture coordinates and position. Then, it outputs several values that are interpolated across a triangle or line. In this essay we are focused on the steps that proceeds the output of the vertex shader: tessellation and Geometry shaders.

Tessellation stage allow us to subdivide triangles, creating more triangles for a given one. Tessellation stage always has three steps: hull shader, tessellator and domain shader. Hull shader has the function to tell the tessellator how many triangles should we generate and in what configuration, outputting a set of control points called patch.

Tessellator is a fixed-function stage only used by tessellation shaders. It has the task of adding several new vertices with the configuration sent by the hull shader. The domain shader will input the vertices of the tessellator and output them as triangles to the next step on the pipeline. These three steps can be ignored if the tessellation factor is equal to one, making tessellation completely optional.

Geometry shader can turn topologies into other topologies. The geometry shader inputs a single topology and its vertices, process this topology and then outputs zero or more vertices. Outputting zero vertices makes these steps optional.

We have used these two steps to create grass in a terrain. We have created two forms of generating grass: with tessellation (creating more vertices per triangle and using them as input) and without tessellation (using only the vertices of the input mesh as inputs for the geometry shader). In figure 12, we can appreciate the difference. Both terrains are made with the same Perlin noise parameters and a Level of Detail (LOD) of 125: the left one does not have tessellation and the right one has a tessellation factor of two and a half.

Figure 12: Grass without tessellation (Left) VS Grass with tessellation (Right)

2.2. Procedural Modification of Inorganic Geometry

2.2.1. *Perlin Noise*

Described by the documentation of Unity, Perlin noise is a pseudo-random pattern of float values generated across a 2D plane. The noise does not contain a completely random value at each point but rather consists of "waves" whose values gradually increase and decrease across the pattern. [7]

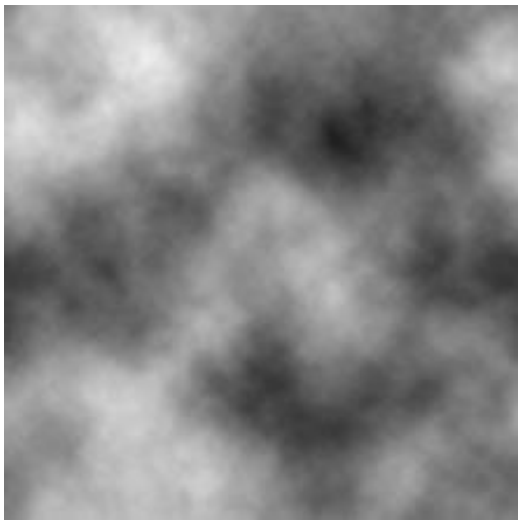

Figure 13: Example of Perlin Noise

It is different to other types of random because Perlin Noise give us interpolated random values in an easy way. This interpolation makes the method to create data-structures with gradual changes that makes possible to create structures that looks organic, such as fire, grass or water.

We can create determined results with Perlin noise by modifying its properties [8]:

- Amplitude: the maximum value that the function can output.
- Scale: distances between 2D points.
- Octaves: controls the amount of detail of Perlin Noise.
- Lacunarity: controls the frequency of each octave.
- Persistence: controls how quickly the amplitude of each octave decreases.
- Seed: Value that determines the random value of the random function.

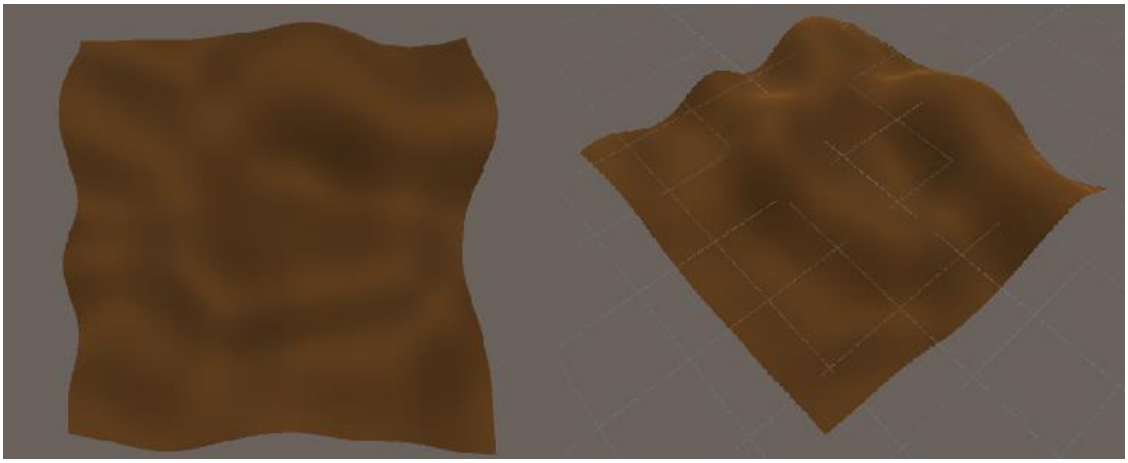

Figure 14: Example of a terrain with Perlin Noise

2.2.2. *Fractals*

The term 'Fractal' was named by the Mathematician Benoit Mandelbrot in 1975. In his seminary work '*The Fractal Geometry of Nature*', he defined it as "a rough or fragmented geometry shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole" [9]. This definition, although true, can be discussed in the context of where fractals are applied.

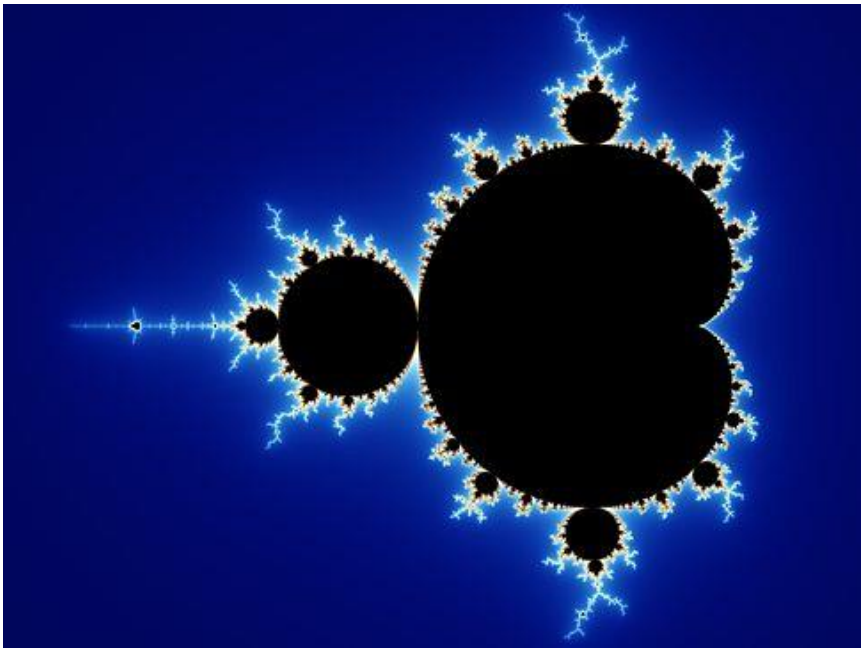

Figure 15: The Mandelbrot Set in a coloured environment

From a computer science perspective, we can find other definitions with a more helpful understanding of how to create and use them. For example, Fractal Foundation describes them as "Fractals are a never-ending pattern. Fractals are infinitely complex patterns that are self-similar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop" [10]. This approach is more helpful, although we will show in this essay applications of this algorithm with a finite use.

Fractals allow us to create objects with a geometry like some structures that we can find in the nature, such as flowers, rivers or trees. We can create primitives, apply them in a recursive way and control some variables (such as rotation or position) in order to create a geometry that resembles our objective.

Figure 16: Examples of Fractals in nature

Although fractals are largely used to create visual results, we can interpolate the meaning of fractal to other areas. For example, design in videogames. Caleb Compton, for example, discuss that given a main core mechanical idea, the rise of complexity and uses that the game does to that concept can be understood as a fractal game design. He uses *The Witness* (2016) as an example of it, justifying that the simple mechanic of the game allows it to create multiple layers of complexity based on a twist of the main idea, maintaining it the same [11].

In modern days we can see different uses for fractals that can be applied to computer science and videogames. In this essay we have used this approach to create terrain through the Diamond-Square algorithm, which we will explain with more details in the development chapter.

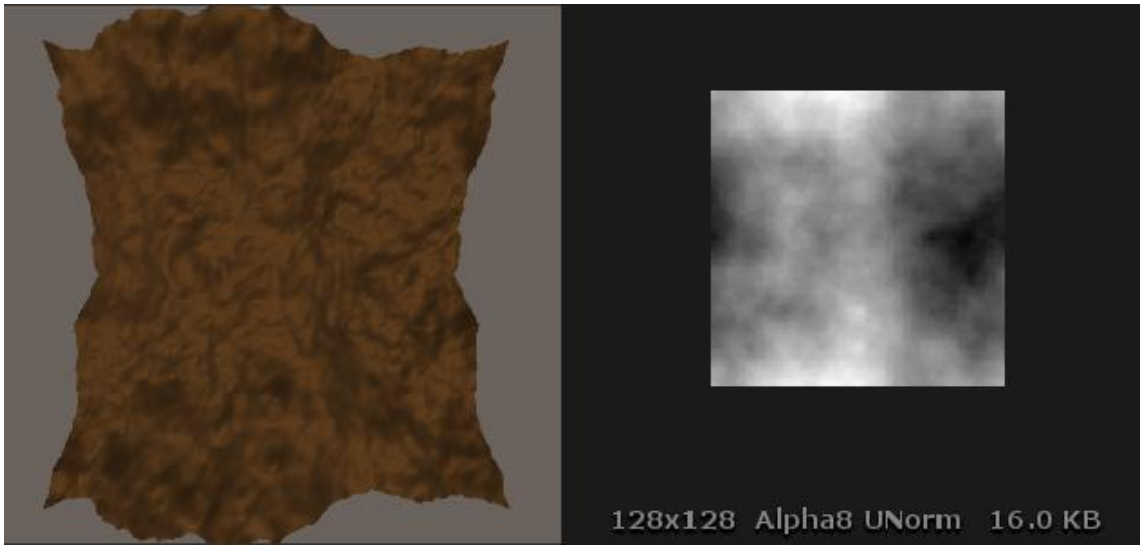

Figure 17: Terrain generated by Diamond-Square algorithm (Left) VS its output as a heightmap (Right)

2.3. Procedural Distribution of Objects

For distributing the objects through a terrain, we have used the results of a Random Number Generator (RNG) to change initial parameters such as position, rotation and / or scale. An RNG, defined by Microsoft Documentation, is an algorithm that produces a sequence of numbers that meet certain statistical requirement for randomness [12].

This might be the easiest way to create any object in a procedural way. We only need an RNG and a basic data-structure to modify. We consider that this method is different to other PCG algorithms as the structure to apply the randomness is not previously modified, while other algorithms use RNG to apply certain data while altering it.

There may be different ways to apply RNG to a data-structure. For example, we give a random value to each pixel of a texture, producing white noise. Other example may be to apply RNG to a structure like a prefab or a model to modify initial parameters. The values we create through RNG must be delimited within a minimum and maximum.

Although this process is very simple and visually effective, all the possible outcomes are limited. We can notice that we really are not generating new objects, but really creating new instances

of an existing object with different parameters. It is possible that a combination of similar values and the same initial structure may produce similar outputs.

One way to fix this problem is considering having a pool of initial data-structures and a method to choose one of them. In the case of inorganic topology, this approach may produce a great result.

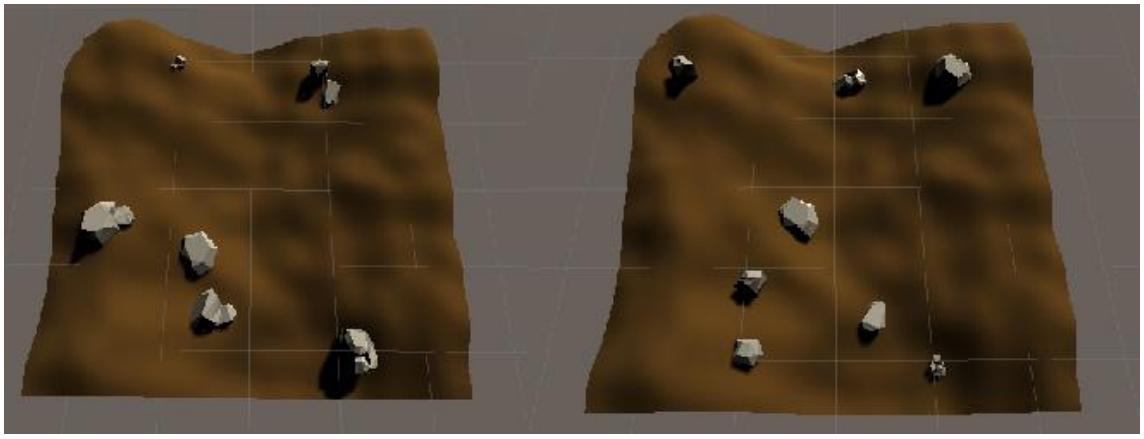

Figure 18: Pool of rocks being applied in a terrain twice

This use of data-structure pool can be applied also to the design in videogames. For example, *Dragon age II* (2011) reuses the same layout of some areas in order to create more zones for the player to complete. There are some random variations like some paths being blocked or not, the number of objects in the scene, the number of enemies or the number characters that you can talk. This approach on design makes production of the project faster, but some players may think that the new areas they discover may seem familiar to them [13].

3. Objectives

In this chapter we organize the objectives of this essay. Initially, the objectives were more focused on development of organic life having the environment as an input. This has been modified because, while researching about techniques and algorithms, it was clearer that the environment was more complex and could offer more to the investigation of this field than the initial objectives.

The objectives of this essay are:

1. Research the techniques of procedural generation, animation and transformation, and compare which would be more fitted in the creation of an organic environment.

2. Research and implement different methodologies to be able to create:

    - A terrain where to habit the objects we create.
    - Individual organic life using CPU computation algorithms.
    - Large amount of organic life using GPU geometric shaders and Tessellation.
    - Add interaction with the end-user using a system that belongs to the environment (e.g. wind).

3. Analyse the results of the system in terms of functionality and optimization, and compare the system obtained with different inputted parameters.

4. Methodology

In this chapter will explain the phases of the project, the tools used and the structure of the code that made possible the development.

4.1. Phases of the project

Three phases were stablished at the beginning of the project in order to keep a stable flow of work. These phases were research, study and development. Below we justify why we need three phases instead of two, which regularly are research and development. This methodology has been applied to all processes of the project.

This chronology shows the duration of each objective with its corresponding phases (see Figure 19).

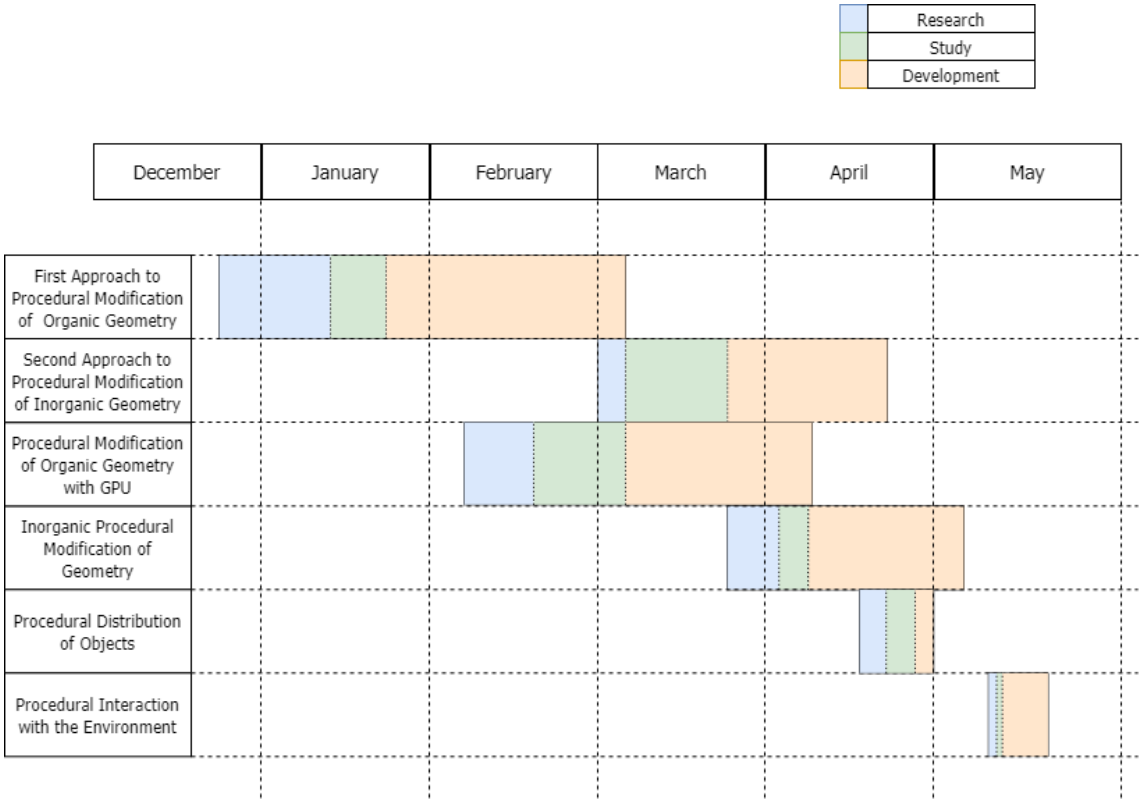

Figure 19: Chronology of the project

We can observe that in the first months Researching and Studying were the main activities, followed by an intense period of development phases, corresponded to a first and a second

approach on Procedural Modification on Organic Geometry. We will justify why there are two phases in the development chapter. Also, other objectives, such as Procedural Interaction with the Environment, are smaller compared to another tasks.

4.1.1. *Research phase:*

This phase was essential for all objectives as at the beginning I was unfamiliar with the topic. Although I started as a vague idea how 3D meshes works in Unity thanks to its documentation, soon began clear that I needed more information as it was getting more complex. It's interesting to see that a lot of people has their own way to stablish their algorithms, so a lot of information was available.

My research was focused on 3D meshes, types of algorithms, transforms with matrices, geometric shaders, tessellation and creation of terrain.

I researched which types of algorithm could we apply in the creation of a mesh. I decided to use a two different types of fan algorithm, to compare later which one would be better. This topic used the explanation given by Linden Reid about triangulation [14]. In my first approach to geometry on organic meshes, I needed to transform a set of vertices using transformation, rotation and scale matrices. Using Chapter 4 of "Real-Time Rendering" [7] gave me a full understanding of these operations.

Jayelinda Suridge's approach [15] was completely different to mine, and some of the problems I was trying to figure out were already solved, so I decided to use her implementation to get more immediate results. Her approach was not difficult to understand as my first approach introduced me in these topics.

For tessellation I used the implementation of GPU by Jasper Flick [16]. For an approach on implementing grass with geometric shaders I used the implementation of Erik Roystan [17]. For terrain, I used the implementation of Jayelinda Suridge [18], although Sebastian Lague has a series of videos which helped me to understand better Perlin noise and how to assign materials depending on the height [19].

4.1.2. *Study phase:*

For me, some objectives were not about founding the information, but rather applying it in a context that could benefit the project. This process of understanding the information was as hard as finding the information. As everyone have their own approach, a lot of information were available but completely different between each other, so I needed time to understand and think how to apply the data found.

An example of this was my first approach on modifying in CPU organic geometry. As I found more information that could give me results in less time, I abandoned this approach in order to use a second approach, although the first one was viable and correct to the proposed objectives.

I chose heightmap, Perlin and Diamond-Square algorithms to explain the different methods that we can create procedurally. With heightmap, anyone that creates a white and black texture can create a terrain with this approach. Perlin noise is an example of terrain that does not require user input (the input for the noise can be assigned by RNG). Diamond-Square is an example of fractal algorithm.

4.1.3. *Development phase:*

Here's where the implementation started. I worked with an agile methodology, iterating over time for each objective. In some cases, iterating over a same process changed radically the development, as we can appreciate in the chronology with my take on Procedural Modification of Organic Geometry with CPU. I choose working with an agile methodology because I need to iterate over time in order to research more and make faster decisions in the development process. Agile methodology allowed me to have faster results without modifying the entire schedule.

The development phase is the one that I applied most of the time. Although researching and studying were crucial, the technical component of the project was very complex, and I needed a lot of time to implement correctly all methods. These were moments of trial and error which were not so harmful to the project thanks to an agile methodology.

4.2. Tools:

The project was developed in the engine Unity. As a code editor I used Visual Studio 2017. Modelling the rocks was made by the three-dimensional modelling Software 'Blender'.

The images used in this essay were made with two-dimensional image editors. Some test textures used in scene were made with 'paint'. The diagrams of this document were made with 'Draw.io'. Some figures of this document were made with 'Photoshop CS6'.

I used a repository using Unity Collaboration System. At the beginning I was saving the project in a portable hard drive, but it showed inefficient in the long term. Changing to a control system was not only better for saving the project itself, but also to control the status of the project, the modifications done, the implemented features…

4.3. Code style:

The code was mostly written in C# and HLSL. I followed a guide of rules to keep a coherence for all written code. These are:

- Functions, Classes, Constants, Structs and Enumerators start with capital letter.
- All code must have a correct indentation.
- Use of regions to delimitate and structure large regions of code.
- Use of descriptive names for variables, with their access modifiers specified. While naming variables, the use of underscores must be avoided.
- Brackets should start or end in new line. If the content inside of the brackets is a get, set or a length equivalent of a single line, we can avoid the use of brackets, although we would have to set it on a new line with an indentation.
- Use of summaries that describes the purpose functions, classes, structs and Enumerators. If they have parameters, they must be specified to.

5. Development

In this chapter we will explain the development process of the project. The tables below show the mathematical notation that we will use in this essay.

Type	Notation	Examples
Angle	Lowercase Greek	α , β_a , ϕ_1
Vector or Point	Lowecase bold	$\mathbf{v}$, $\mathbf{p_a}$, $\mathbf{t_1}$
Array	Capital bold and Italic with brackets	$\boldsymbol{T_o[n]}$, $\boldsymbol{T_n[1]}$
Matrix	Capital bold	$\mathbf{T}(\mathbf{p})$, $\mathbf{S}$, $\mathbf{R_z}(\mathbf{d})$
Plane	Capital Italic	M , U_a , F_1 , $P(\mathbf{d_1}, \mathbf{d_2})$
Triangles	$\triangle$ and three points	$\triangle\mathbf{t_0t_1t_2}$

Table 1: Mathematical Notation

Operator	Descriptor
$\|\| \mathbf{v} \|\|$	Magnitude of a vector
$\mathbf{u} \perp \mathbf{v}$	Two perpendicular vectors
$\mathbf{u} \cdot \mathbf{v}$	Dot product of two vectors
$\mathbf{u} \times \mathbf{v}$	Cross product of two vectors

Table 2: Vector operators

Functions	Notation	Examples
Functions I implemented myself	Tahoma regular	BuildSphere()
Functions from other classes in Unity	`Courier New regular`	`Mathf.Lerp()`

Table 3: Function notation

5.1. Procedural Modification of Organic Geometry with CPU

5.1.1. *First Approach with Extrusion*

My first objective was to modify the geometry of the mesh by CPU. I had no experience with meshes at the beginning, so researching and studying this field was crucial. My first objective was to create vegetal life that grew in real-time through a direction.

Therefore, I decided that I needed to create an extrusion class that takes a polygonal base that could grow over time. Blender defines extrusion as "A procedure to duplicate vertices while keeping the new geometry connected with the original vertices" [20]. Because we are doing more than just replicate vertices, we are going to give a different definition in this essay: "Extrusion can be defined as generating a new surface or face from an existing one along a direction".

The first class I implemented was an approach based on Linden Reid's on procedural geometry and triangulation, which helped me to stablish some basis for a Mesh class [15]. I also implemented an editor based on Sean Duffy's 'Runtime Mesh manipulation with Unity' [21], stablishing a basis for debugging the mesh in the editor. These two classes were the first one in which I focused.

The editor script allowed me to see in the scene view of Unity the position of each vertex and its normal. It takes as parameter two meshes: the original one and the procedural generated. With the original one we can reset the geometry of the mesh to the initial mesh. With the second one, we can show in the screen where is the position of each vertex by drawing a 3D visual Gizmo in each position. We also have a method for displaying the direction of the normal of each vertex. This editor is really helpful as I had a visual help of everything I was creating and where it was located.

The extrusion class starts by creating a n-sides regular polygon as a base and iterating several times extruding the previous face, calculating new triangles and normals. Each face varies on position, rotation and scale. At the end, we stellate the final face. Linden Reid defines Stellation as "the process of extruding a face and converge it into a point" [22].

For generating any type of primitive, we have used two triangulation algorithms: open fan algorithm and closed fan algorithm. These two triangulations are defined as fan methods in 'Surface Reconstruction for Three-Dimensional Rockfall Volumetric analysis' [23]. Although the two algorithms belong to fan algorithms, to avoid confusion, we are going to call them fan and centroid algorithm respectively.

Fan algorithm generates only one triangle per three vertices, sharing one common vertex called 'central vertex'. It's the most basic way of triangulating a mesh. Centroid algorithm generates one more vertex and two more triangles in order to modify the central vertex, allocating it in the centre of the regular polygon. Figure 20 shows us a representation of these algorithms given a regular polygon of five sides. We can observe how the central vertex (in red) is modified depending of the algorithm.

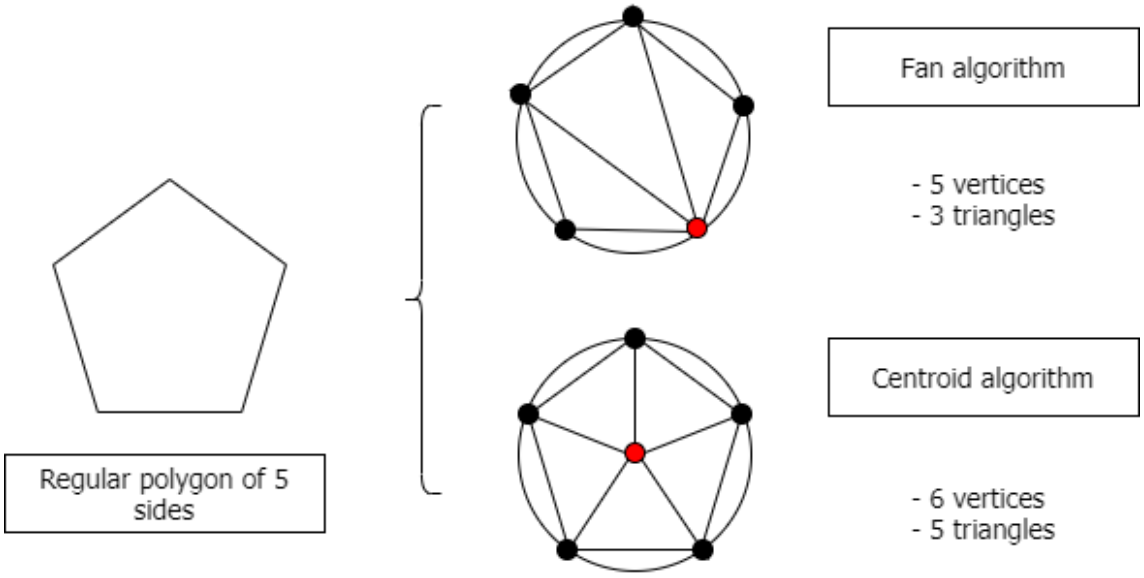

Figure 20: Fan algorithm VS Centroid algorithm.

Choosing which one is better is complex. Centroid algorithm requires more data to save because we are storing more vertices and triangles per face (exactly one vertex and two triangles). If we are generating a very complex mesh, using this technique might consume more resources. But having a vertex in the central position of a polygon can help us later with some techniques such as extrusion or stellation.

Fan algorithm, in general, is the best algorithm as it stores the smaller number of vertices and triangles for a mesh. Unless we want to apply some specific techniques on a smaller mesh, we would use Fan algorithm.

So, we initialize the mesh cloning the meshFilter (the Unity component where the mesh is stored) and working with a copy of the mesh. It is better to work with a copy rather than the original mesh. The original mesh will be used for the editor script to reset the mesh in case we want to. Then, depending on the algorithm type, we initialize the base.

The technique to create the vertices is similar in both as they create regular polygons. We divide the length of a circumference in radians by the number of sides of the polygon. Having that proportion, we can calculate the position of each vertex in the XZ plane with the cosine and sine of the proportion value multiplied by the iteration number. Multiplying those values with the radius of the circumference, we create a regular polygon with n-sides. In the case of Centroid algorithm, the number of sides is incremented in one and the central vertex **o** will be placed in the origin as **o** = (0.0f, 0.0f, 0.0f).

In this essay we have taken a design decision of storing always the last vertex as the central vertex, the vertex that all triangles will share. It could be the first vertex, or it could be in any position, but we have taken that decision because we can get its position easily and it will be helpful for later techniques.

For triangulating the vertices, we calculate the number of triangles of each face because each algorithm follows a linear progression given the number of vertices. Defining **n** as the number of sides of the polygon, in fan algorithm the number of triangles of each polygon is equal to **n** - 2, while with Centroid algorithm is equal to **n**. We can observe that progression fits also with our previous figure.

In fan algorithm we initialize the triangle array of the mesh by assigning indexes to a pointer $\mathbf{t_0}$ = **n**-1, a pointer $\mathbf{t_1}$ = 1 and a second pointer $\mathbf{t_2}$ = 0. We create an iteration where each triangle is made is $\triangle \mathbf{t_0 t_1 t_2}$. At the end of each iteration we increment $\mathbf{t_1}$ and $\mathbf{t_2}$ by one.

With Centroid algorithm we follow a similar scheme: $\mathbf{t_0}$ = **n**, $\mathbf{t_1}$ = 1 and $\mathbf{t_2}$ = 0. But with Centroid algorithm has a condition in each iteration: if $\mathbf{t_1}$ is greater or equal to **n**, $\mathbf{t_1}$ equals 0. This is made

to create correctly the last triangle. Figure 21 shows a visualization of the triangulation for both algorithms, with the central vertex in red.

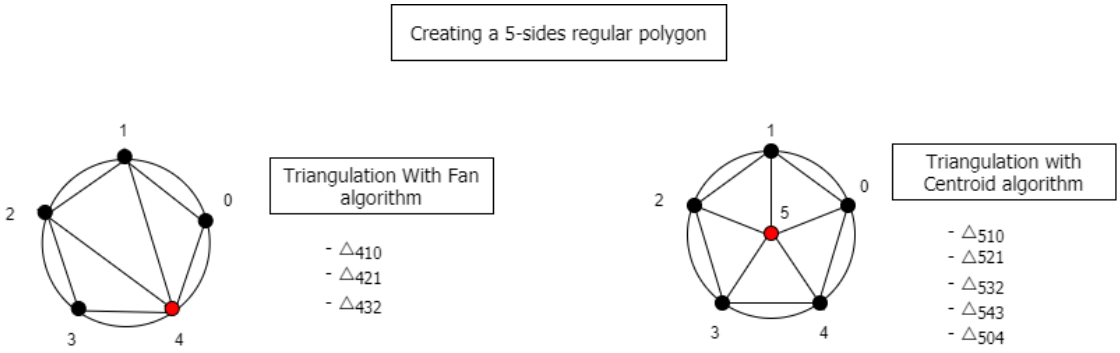

Figure 21: Triangulation with Fan Algorithm VS Centroid Algorithm

Normals are assigned with a directional vector **v** = (0.0f, 1.0f, 0.0f). We are not going to apply uvs to the mesh as we have considered not relevant to the final result. Lastly, we add an optional method that closes the mesh in case we want to observe the mesh from the perspective of the back-faces.

We cannot share the vertices for closing the polygon with a CCW algorithm because the normals are assigned to the vertices, and both faces of the mesh would be wrongly illuminated as they were facing the same position.

To fix this we have implemented a new method to duplicate vertices, assign the indexes with CCW order and assign the normal correctly. The result is a polygon of n-sides but the size of each array (Vertices, Triangles and Normals) is equal to 2**n**. We save the back-face data at the beginning of the arrays because future methods use the last position of the arrays to retrieve the data of the front-face, which it is always the face to be extruded.

Once we initialize the mesh, we save the indexes of the face we have created. In our class, as we want to extrude specific vertices, we have to save which indexes we have modified in the last operation. We call this triangle array ***$T_o[n]$***.

First we create the vertices of the new face. We must define some variables: **n** is the number of vertices per face, which is the same for both; F_1 is the face that is going to be extruded and F_2 is the extruded face. Vertices on F_1 and F_2 share same values at the beginning as we animate the

vertices of F_2 with a TRS matrix and interpolation. The transformed face by the TRS matrix will be defined as F_2'.

TRS stands for "Translation, Rotation and Scale". It is a concatenation of matrices in order to apply a complete transformation, in this case to a set of vertices. If we apply it to a set of three-dimensional points we can move, rotate and scale them in the three axes. As matrices are not commutative, the sequence of concatenation is important.

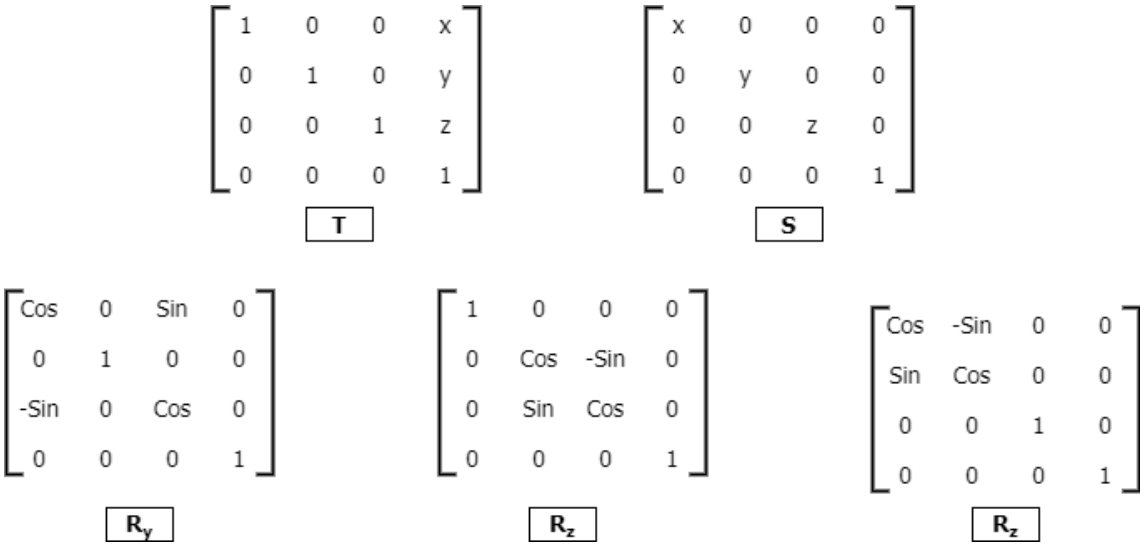

Figure 22: Representation of matrices.

Figure 22 shows the different types of matrices we apply in TRS: **T** is a translation matrix, **S** is a scale matrix, $\mathbf{R_y}$ is a rotation matrix in Y-axis, $\mathbf{R_z}$ is a rotation matrix in Z-axis and $\mathbf{R_x}$ is a rotation matrix in X-axis. The sequence of concatenation in order to create a TRS matrix **C** is equal to **C** = (**T**(**R**$_y$(**R**$_z$(**R**$_x$(**S**(**p**)))))), where **p** is the vertex to be transformed. We can group the rotation matrix into **R** = (**R**$_y$(**R**$_z$(**R**$_x$))), making **C** = (**T**(**R**(**S**(**p**)))).

For **T** and **R**, we need a directional vector **d** which is the direction of the extrusion. This directional vector **d** is calculated in each step of the process with a positive random value, in order to extrude with different values.

Also, as we need to establish the centre of $F2$ as the basis of the transformation. We will call this point **o** and we will have to calculate a Matrix **M** = **C**(**T**(**o**)). With Centroid algorithm is easy to calculate **o** as we have saved it as the last vertex of F_2 . With Fan algorithm, as we are operating with regular polygons, we have to calculate the average of the **n** vertices to calculate **o**. **T**(**p**) and

R(**p**) will use the directional value **p** = **d**, although for **R** we will have to convert it into radians. As we want to scale it uniformly, the values for **S** will be the same in the three axes.

One of the objectives of the extrusion was to animate the extruding face, so we will use these values in the interpolation of F_2. However, as we are not interested in recalculating the normals of F_2 in each iteration, we use the values of F_2' to calculate the normals. Therefore, we establish F_2 vertices as the same of F_1.

Once we create the vertices of F_2 we must triangulate F_1 with F_2. Depending on the algorithm we are using, triangulation will be different. We will define $\mathbf{t_n}$ as the number of triangles in the extrusion process. In fan algorithm, the number of triangles we are going to create is equal to $\mathbf{t_n} = \mathbf{n} * 9 - 6$. In Centroid algorithm is equal to $\mathbf{t_n} = \mathbf{n} * 9$.

These formulas are the result of calculating that, per each side-face, we want to create a quad with two triangles that will join four vertices, which will be added the triangles of the top-face of the mesh, which is the same formula that we have used for initialization.

We call the new array of indexes $\boldsymbol{T_n\,[n]}$, which corresponds to the triangle indexes of F_2. The triangle indexes of F_1 corresponds to the triangles array that we store after each extrusion, $\boldsymbol{T_o\,[n]}$.

We will need four pointers in each algorithm: $\mathbf{t_a} = \boldsymbol{T_o\,[0]}$, $\mathbf{t_b} = \boldsymbol{T_n\,[0]}$, $\mathbf{t_c} = \boldsymbol{T_n\,[1]}$ and $\mathbf{t_d} = \boldsymbol{T_o\,[1]}$. We iterate **n** times and generate two triangles: $\triangle \mathbf{t_a t_b t_c}$ and $\triangle \mathbf{t_a t_b t_d}$.

In Fan algorithm, we check in each iteration that if $\mathbf{t_c}$ is greater or equal to $2\mathbf{n}$ it changes to $\boldsymbol{T_n\,[0]}$ and if $\mathbf{t_d}$ is greater or equal to **n** it changes to $\boldsymbol{T_o\,[0]}$. In centroid algorithm we check that if $\mathbf{t_c}$ is greater or equal to $2\mathbf{n} - 1$ it changes to $\boldsymbol{T_n\,[0]}$ and if $\mathbf{t_d}$ is greater or equal to $\mathbf{n} - 1$ it changes to $\boldsymbol{T_o\,[0]}$. For the top face we do the same procedure that in the initialization but using $\boldsymbol{T_n\,[n]}$. Figure 23 shows is a visualization of the process.

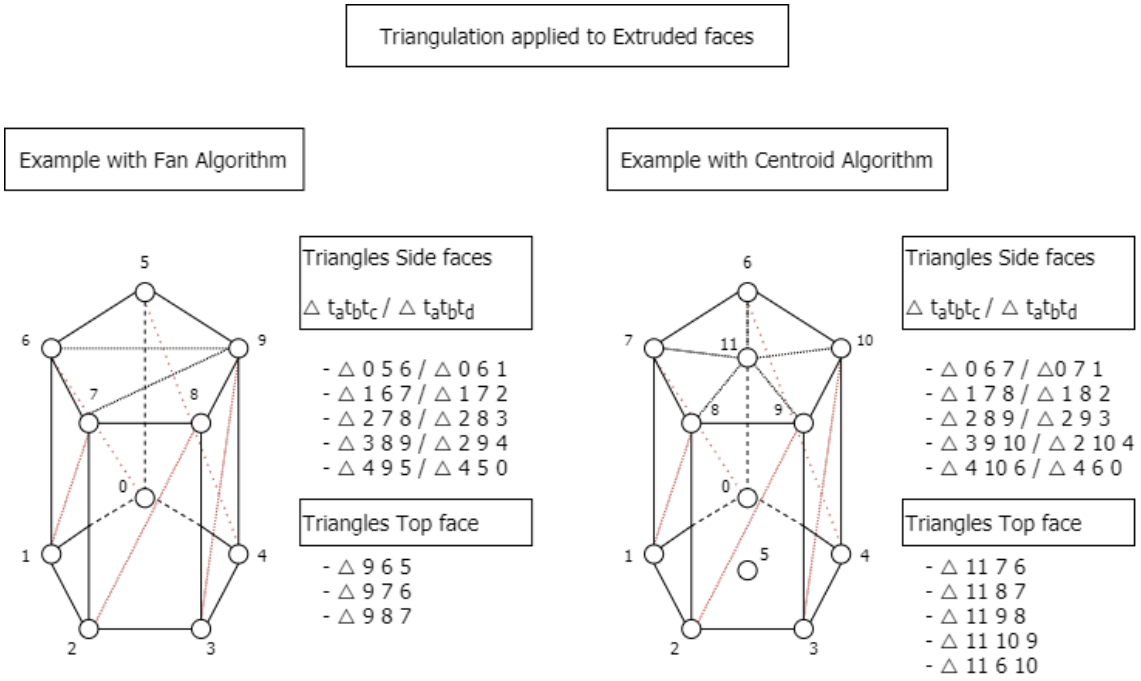

Figure 23: Triangulation applied to extruded faces

We have decided in this essay to not apply the centroid algorithm to the side faces. We wanted to join the vertices of F_1 and F_2 with the least number of possible triangles, that means, not creating central vertices per each side face. We could have done this if we would be interested on extruding side faces.

Once we have created the triangles, we assign the normals. It is the same method for both algorithms. We start iterating **n** times and assigning the normals to F_1. We use a method to calculate a surface normal given three points, which must be selected in a CW order to calculate correctly the direction, as we want them to point them outwards. If not, the final output lacks off light.

We use $\mathbf{p_1} = \boldsymbol{T_o\,[0]}$, $\mathbf{p_2} = \boldsymbol{T_n\,[0]}$ and $\mathbf{p_3} = \boldsymbol{T_n\,[1]}$, where $\boldsymbol{T_n\,[n]}$ belongs to F_2' instead of F_2. This is done because we want to calculate the normals from the transformed face. If we calculate it from F_2, we would produce incorrect lightning. With the points we can create vectors $\mathbf{v_1} = \mathbf{p_2} - \mathbf{p_1}$ and $\mathbf{v_2} = \mathbf{p_3} - \mathbf{p_1}$ and then the normalized cross product $\mathbf{v} = \mathbf{v_1} \times \mathbf{v_2}$, which will be the final normal.

For each iteration we increment $\mathbf{p_1}\ \mathbf{p_2}\ \mathbf{p_3}$, checking that if $\mathbf{p_3}$ is greater or equal to **n** its value changes to $\mathbf{p_3} = \boldsymbol{T_n\,[0]}$. The vertices of the top faces share the same direction as they are part

of the same face. We can calculate the surface normal for three CW points and assign it to all vertices of the face.

However, this whole process of calculating manually the normals can be skipped as the Mesh class has a function to recalculate the normals. As we wanted to introduce manually the normals of the mesh, we have given the possibility of choosing between calculating manually the normals or calculating through Unity's procedure. Lastly, we would save $T_n[n]$ data in $T_o[n]$ as the indexes of the last operation is the new face we have created.

Once we have finished the extrusion itself, we can start the process of animating it through interpolation. Interpolation is the method to calculate a percentage between two values. Although there are different types, we will use linear interpolation, which is given in Unity with the 'Lerp' function. As we want to interpolate a set of positions, we will use '`Vector3.Lerp`'.

We want the interpolate the vertices in **t** seconds, which will use to calculate the frequency **f** = 1 / **t**. We start with **c** = 0.0f and each iteration we calculate the factor $\mathbf{d_t}$ = Time.deltaTime * **f**. Time.deltaTime is defined by Unity as completion time in seconds since the last frame [24], and we can use it as the internal clock of the computer. We will calculate **c** += $\mathbf{d_t}$ in each iteration because **c** is greater or equal to 1.0f means that the animation has finished.

Each iteration we interpolate the values of T_2 and T_2' and we save it into a new array V_i. The rate of interpolation is calculated with easing functions. We have used the easing functions from the collaboration project started by Soledad Penadés [25], that enabling different types of acceleration. We can change the values from V_i to T_2 in each iteration, transforming it each time until T_2 is equal to T_2'. The extrusion is finished at this moment.

In our implementation we have a limit to the extrusion process. We establish the number of iterations and, once we reach it, we proceed to the stellation. This process is very similar to extrusion but instead of extruding a new set of vertices from a face F_1, we only have to control one vertex defined as **q**.

Although, this is not the case for Centroid algorithm because the centre vertex can be used as **q** and we would only have to calculate the normals for F_1. This is the main advantage of using Centroid Algorithm.

For fan algorithm, we calculate the average of the vertices of F_1, calculating the transformation of the vertex (which will be done by normalizing the directional vector and adding it to the position of **q** multiplied by the extruding distance) and calculating the triangles the same way that we do with Centroid algorithm. Then we calculate the normals with the transformed **q** and then using the interpolation method with **q** only.

Figure 24 shows us all the process of extrusion with four iterations, concluding with a stellation process.

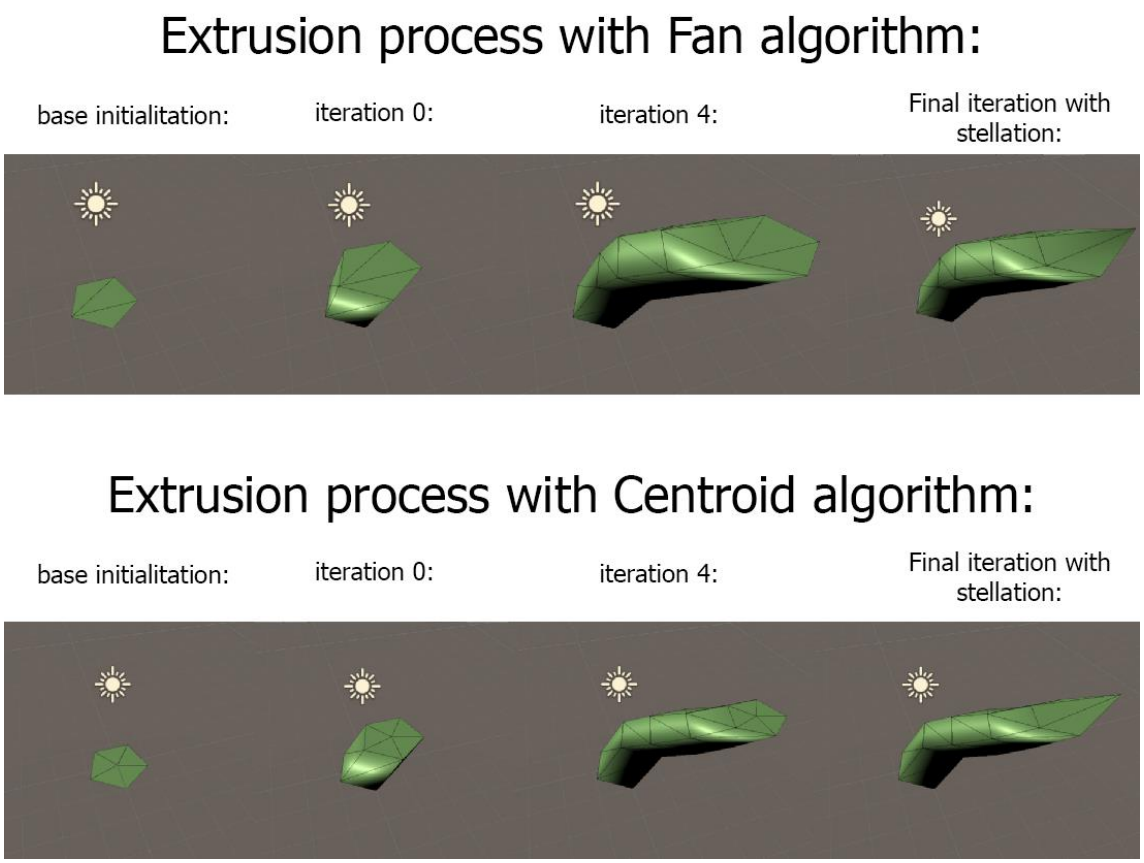

Figure 24: Process of extrusion with four iterations and stellation

5.1.2. *Second approach with geometry class*

Although we can extrude correctly the mesh like a growing vegetable life in real-time, the resulting mesh shows a problem. As we can perceive in figure 24, the illumination of the mesh seems odd. This is because, as we are sharing vertices per each face, the calculation of the illumination on each face is not correct. To have correct lightning, the normal of each triangle has

to be perpendicular to the triangle itself, but because we are sharing vertices, the sum of the normals of each triangle it is not perpendicular.

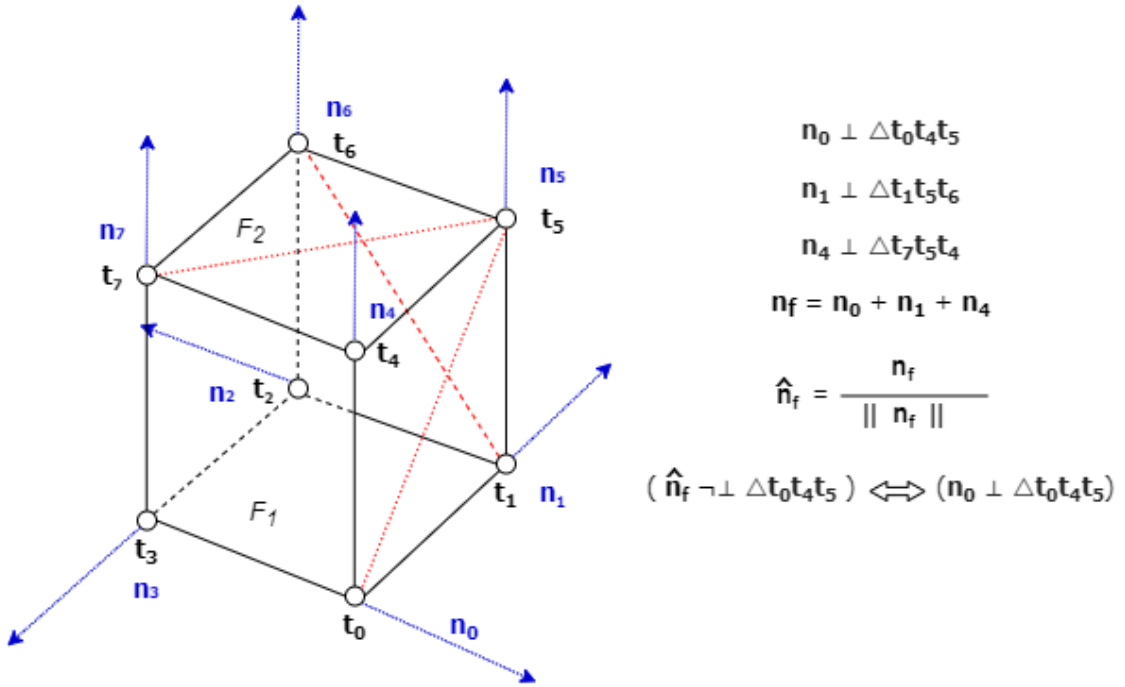

Figure 25: Problem with shared normals assigned to vertices

Figure 25 shows us the representation of this problem. We have chosen an extrusion process with Fan algorithm of 4 sides. The normal $\mathbf{n_0}$ corresponds to the vertex $\mathbf{t_0}$, being $\mathbf{n_0}$ perpendicular to the triangle $\triangle$ $\mathbf{t_0t_4t_5}$ and $\triangle$ $\mathbf{t_0t_5t_1}$. Normal $\mathbf{n_1}$ corresponds to the vertex $\mathbf{t_1}$, being $\mathbf{n_1}$ perpendicular to the next side-face triangles $\triangle$ $\mathbf{t_1t_5t_6}$ and $\triangle$ $\mathbf{t_1t_6t_2}$. Normal $\mathbf{n_4}$ corresponds to vertex $\mathbf{t_4}$, being $\mathbf{n_4}$ perpendicular to face *F2* and its triangles $\triangle$ $\mathbf{t_7t_5t_4}$ and $\triangle$ $\mathbf{t_7t_6t_5}$.

The final sum of normals is $\mathbf{n_f} = \mathbf{n_0} + \mathbf{n_1} + \mathbf{n_4}$, and its normalized value results in a directional vector that it is not perpendicular to $\triangle$ $\mathbf{t_0t_4t_5}$, making the approach of procedural geometry with sharing vertices nearly impossible with a framework that uses normals per vertices. In fact, we cannot use the method of Unity to recalculate automatically the normals as we have the same error that doing it manually.

While trying to resolve this problem, I researched for more information and other approaches that could help me. Finally, I found Jayelinda Suridge's approach [15]. In order to create triangles

with correct lightning, we would have to generate three unique vertices with same normal direction that would not be shared by any triangle. This way we can create correct lightning.

Her approach also shows different advantages. First of all, we are do not save the indexes of our previous operations as it does not interest us anymore. We can create a general class to build a Mesh in which we store all the data (vertices, triangles indexes, normals and uvs) in different lists, which at the moment of creating the mesh we transform it into an array.

For example, if we want to create a triangle, we will add to the list the data of this triangle in order. The output is the same that doing the same method but resizing the array each time we want to add more data.

Other advantage is that it is focused on modularity. Every figure can be discomposed into different sections, each section being a different application of the same base. For example, If we want to create a flower, we will start by the stem. The stem is similar to a cylinder, which is created by different quads in a specific rotation and position.

The method of creating a quad is simple and can be called if we want to create another primitive, for example while creating cubes. This leads us to a process of using simple methods in order to build a primitive instead of one specific method to build a specific primitive.

Therefore, I decided to change my approach on Procedural Geometry of Organic Objects. Figure 26 shows an UML Diagram of implementation [26] example of the essential classes required to build a cube: an abstract class called 'GeometryClass', a class that stores mesh data that can initialize the mesh called 'MeshBuilder' and a class that calculates the data mesh, in this case 'BuildCube'. We can store each position and rotation in every step with a class called 'PassData'.

Another thing that we are interested into creating is meshes with different materials. For this, we have to use subMeshes. SubMeshes are one or more topologies inside of a mesh. The mesh stores those topologies in an array, which we can modify to set a specific number of subMeshes.

In order to have a mesh with more than one subMeshes, we create a vertex, triangles, normals and uvs array per each subMesh. We create the primitives in a specific subMesh and, at the end, we build them together with its respective material. This way, we can generate parts of a mesh with different materials.

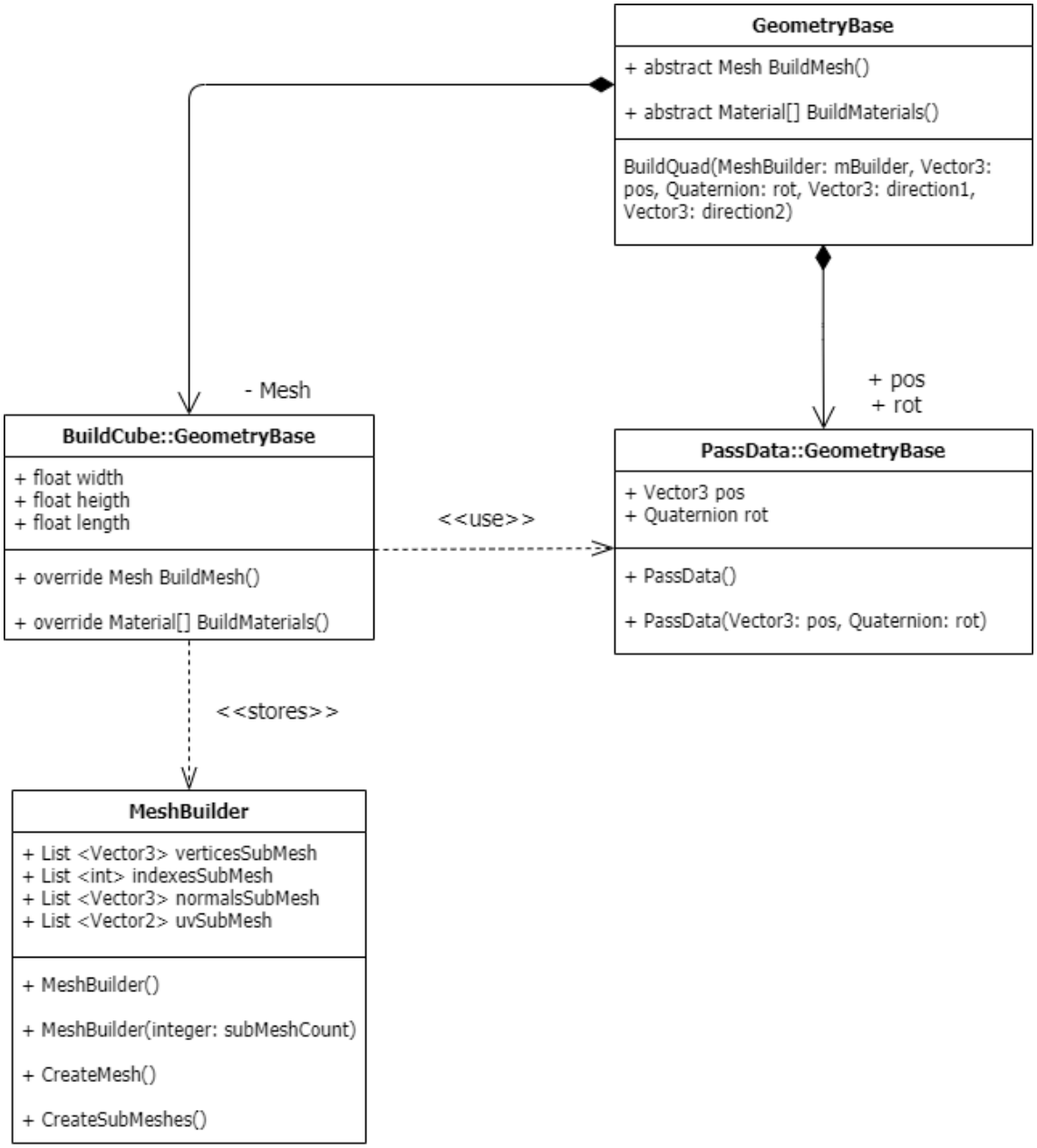

Figure 26: Class implementation diagram for building a cube

We start by explain the different methods to create topologies, being the simplest the cube. A cube only needs six quads, one per each face. We create the position for the left-bottom-nearest vertex called $\mathbf{v_0}$ and the position for the right-up-furthest vertex called $\mathbf{v_1}$.

We generate the quad with a method that takes as parameters the position it wants to be created, the rotation, a directional vector $\mathbf{n_1}$ and a directional vector $\mathbf{n_2}$. The magnitude and direction of n1 and n2 change depending on the quad.

We multiply the position with the rotation in order to have a position pivot called $\mathbf{p_0}$. Through $\mathbf{p_0}$, we can calculate the position of the rest of the vertices by adding $\mathbf{n_1}$, $\mathbf{n_2}$ or both of them. As we have added already the vertices, we took as a triangle index pivot the length of the vertex list and subtracting four as we have added four vertices.

We add the triangle indexes with a CW winding order. We can calculate the normal of the quad as the cross product between $\mathbf{n_1}$ and $\mathbf{n_2}$, normalizing the result.

We use this method creating three directional vectors, $\mathbf{d_1}$ = (width, 0.0f, 0.0f), $\mathbf{d_2}$ = (0.0f, height, 0.0f) and $\mathbf{d_3}$ = (0.0f, 0.0f, length). For generating quads related to $\mathbf{v_0}$, we create $F_0(\mathbf{v_0}, \mathbf{d_3}, \mathbf{d_1})$, $F_1(\mathbf{v_0}, \mathbf{d_1}, \mathbf{d_2})$ and $F_2(\mathbf{v_0}, \mathbf{d_2}, \mathbf{d_3})$. For the vertex $\mathbf{v_1}$, we create the faces $F_3(\mathbf{v_1}, -\mathbf{d_1}, -\mathbf{d_3})$, $F_4(\mathbf{v_1}, -\mathbf{d_2}, -\mathbf{d_1})$ and $F_5(\mathbf{v_1}, -\mathbf{d_3}, -\mathbf{d_2})$. Figure 26 shows the visualization of this process.

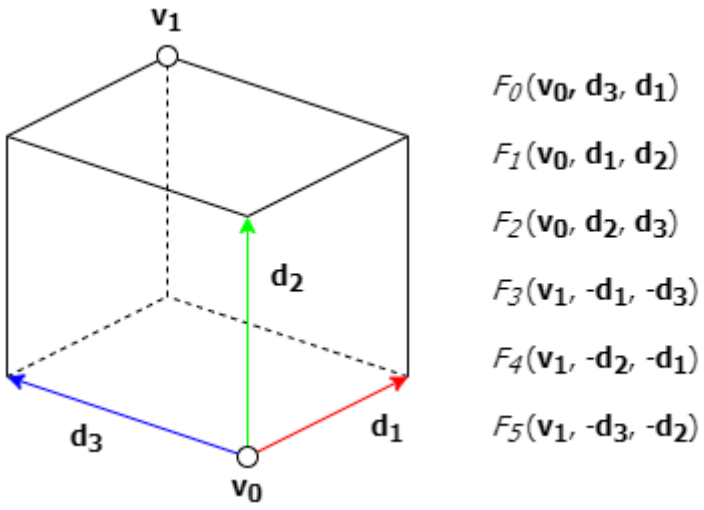

Figure 27: Creation of a cube through geometry class method

Besides cubes, we can create more complicated primitives based on curved surfaces, such as cylinders, cones, frustums and spheres. These primitives are similar to each other with some variances. The most basic one is the cylinder, which we use as an example.

Cylinders can be created as a sum of circumferences stocked in a direction. Our method starts by establishing an initial position and rotation. These two parameters are stores in a 'PassData' class as we use them to store the position and rotation at the end of each operation. We also have to

define the radius of the cylinder and its height. We define them **r** and **h** respectively. Then, we establish the number of subdivisions that we want.

There are two types of subdivision: radial subdivisions and height subdivisions. Radial subdivisions are the number of **r**. The greater this value, the more resemblance would have to a cylinder instead of a polygonal prism. We define this parameter as $\mathbf{s_r}$. Height subdivisions are the number of divisions of **h**. This parameter is helpful if we want to create a cylinder with a bending angle as the slope would be smoother. We define this parameter as $\mathbf{s_h}$.

Our method divides the height of the cylinder with the number of subdivisions, calculating the number of circumferences that the cylinder has. The number of circumferences is equal to $\mathbf{s_h}$. We iterate $\mathbf{s_h}$ times to calculate the centre of each circumference and call a method to build a set of quads. This centre is defined as $\mathbf{o_c}$.

Our method to calculate $\mathbf{o_c}$ varies if our cylinder has a bending angle or not. The bending angle is the slope of the cylinder in the Z-axis and will be defined as α. If the value of α is zero, we can calculate $\mathbf{o_c}$ = (0.0f, **t**, 0.0f), being **t** the actual iteration.

If α has a different value means that the cylinder has a slope. For calculating $\mathbf{o_c}$ in each iteration, we have to transform α in radians and divide it by $\mathbf{s_h}$. We define this increment in radians of each rotation as $\mathbf{i_z}$. We can calculate $\mathbf{o_c}$ = (`Mathf.Cos`($\mathbf{i_z}$ * **t**), `Mathf.Sin`($\mathbf{i_z}$ * **t**), 0.0f), being **t** the actual iteration.

This way we calculate the centre of the circumference, although we have not calculated $\mathbf{o_c}$ rotated in the Z-axis. If we want to rotate the circumference in each rotation, we have to calculate a quaternion **q**.

This quaternion **q** is the result of calculating the rotation in three-dimensions and then transformed into a quaternion. The value of **q** in three-dimensions is **q** = (0.0f, 0.0f, $\mathbf{i_z}$ * **t** * `Mathf.Rad2Deg`). But, for rotating only one point, we can calculate how the radius bends. This value is $\mathbf{b_r}$ = **h** / α * `Mathf.Deg2Rad`. The rotated centre is $\mathbf{o_c}$ *= $\mathbf{b_r}$.

As we want to build our cylinder in the local origin and not with a distance, we calculate the offset that the rotation produces from the origin as **d** = (**b**$_r$, 0.0f, 0.0f). Finally, **o**$_c$ with a slope of α will be equal to **o**$_c$ -= **d**.

Our method to build a ring of quads uses α as a parameter. If α is equal to zero, it would be a quaternion identity. If it is not, it is **q**. First of all, we calculate the increment of the length of a circumference divided by **s**$_r$, meaning that this increment is defined as **i**$_r$ = (2 * π) / **s**$_r$. Then, we iterate **s**$_r$ times. The position of each point is defined as **p** = (`Mathf.Cos`(**i**$_r$ * **t**), 0.0f, `Mathf.Sin`(**i**$_r$ * **t**)), being **t** the actual iteration.

Then, we have to apply a set of transformations: we rotate **p** as **p**$_r$ = **p** * **q** and translates **p**$_r$ as **p**$_f$ = **o**$_c$ + **p**$_r$ * **r**. **p**$_f$ is added to the vertices list and **p**$_r$ to the normals list. For building triangles, we have to check that we are not at the beginning of each method, because we cannot build triangles if we only have one row of vertices. The procedure is the same that we did in the extrusion but calculating previous indexes from the last index added.

The result of all these methods creates a cylinder, but it is not closed because we have built the exterior quads, ignoring the top and bottom base. We will control if we want to build the bases of the cylinders, as sometimes we are not interested into building them. We use a method to create to circumferences: B_0 and B_1. B_0 is created at the starting position with the starting rotation, and B_1 is created at the final position with the final rotation in the Z-axis.

The only difference between a cylinder and a cone or frustrum is that B_0 and B_1 do not have the same radius **r**. In a frustrum, **r**$_s$ and **r**$_f$ are not the same. In a cone, the final radius **r**$_f$ is equal to zero. Therefore, we can create frustums and cones with the same method.

To calculate the radius **r**$_t$ of each circumference, we calculate the interpolation between the starting radius and the final radius with the actual iteration divided by the number of vertical divisions, resulting in **r**$_t$ = `Mathf.Lerp`(**r**$_s$, **r**$_f$, (float) **t** / **s**$_r$). Figure 28 shows a visualization of these methods.

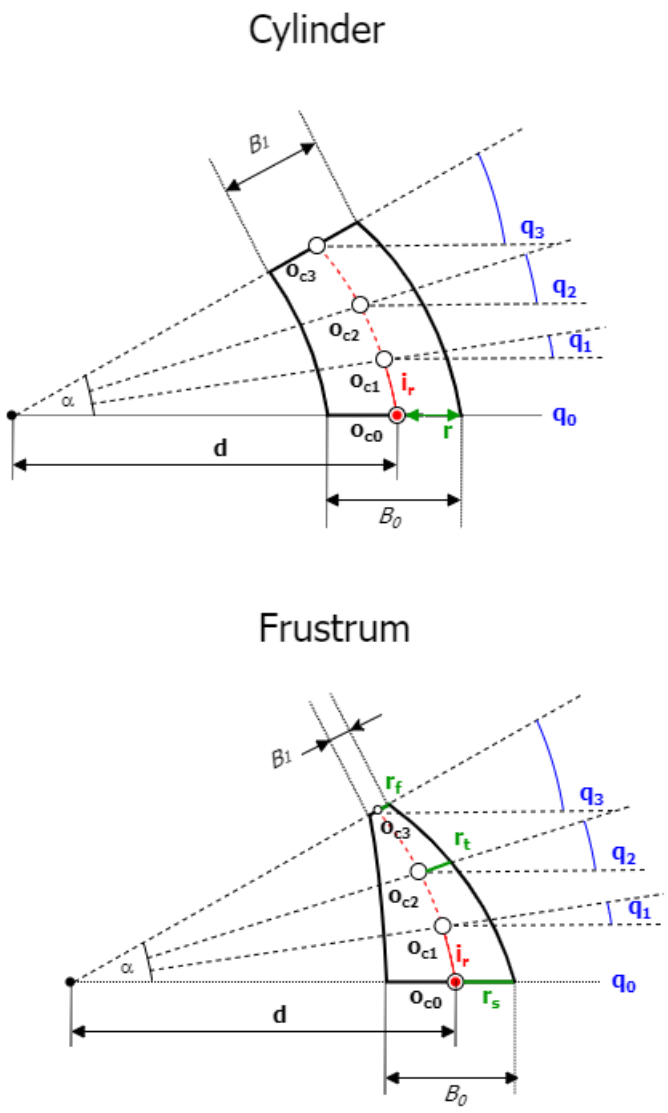

Figure 28: Creation of curved primitives through geometry class

The method for creating a sphere is similar to the frustrum. We have to consider a cylinder that starts with a radius $\mathbf{r_{c0}}$ = 0, reaches its maximum value at the middle as $\mathbf{r_{cm}}$ = $\mathbf{r_h}$ and at the end it reaches zero again $\mathbf{r_{cf}}$ = $\mathbf{r_{c0}}$ = 0. Its method is the same as following the arc of a semi-circumference of radius **r**. We can calculate the position of each circumference as the increment in radians dividing π with the number of height subdivisions $\mathbf{s_h}$, being the increment $\mathbf{i_c}$ = π / $\mathbf{s_h}$.

With this proportion we can calculate the centre of each circumference as $\mathbf{o_c}$ = (0.0f, -`Mathf.Cos`($\mathbf{i_c}$), 0.0f). For later purposes, we are interested in having different radiuses in X-Y-axis, so the radius of the circumference is calculated as $\mathbf{r_c}$ = `Mathf.Sin`($\mathbf{i_c}$) * $\mathbf{r_h}$, being $\mathbf{r_h}$ the horizontal radius and $\mathbf{r_v}$ the vertical radius. We have to define the proportion in the Y-axis and

calculate the vertical radius as $\mathbf{r_v} = \mathbf{r_h} * \mathbf{v_s}$, being $\mathbf{v_s}$ the vertical proportion. Finally, we can calculate correctly the proportions of a sphere by calculating the height of $\mathbf{o_c} = \mathbf{o_c.y} * \mathbf{r_v} + \mathbf{r_v}$.

Nevertheless, there is a problem related to the normals, as we have to calculate correctly the normal direction of the quads. We have to calculate the amount of slope in each iteration and create a new method to calculate the normal direction $\mathbf{n_c}$. The slope will be a vector 2 $\mathbf{v} = (-\mathbf{o_c.y} / \mathbf{v_s}, \mathbf{r_c})$. The value of $\mathbf{n_c}$ is $\mathbf{n_c} = (\mathbf{o_c.x} * \mathbf{v.y}, -\mathbf{v.x}, \mathbf{o_c.z} * \mathbf{v.y})$. Figure 29 shows a visualization of this method.

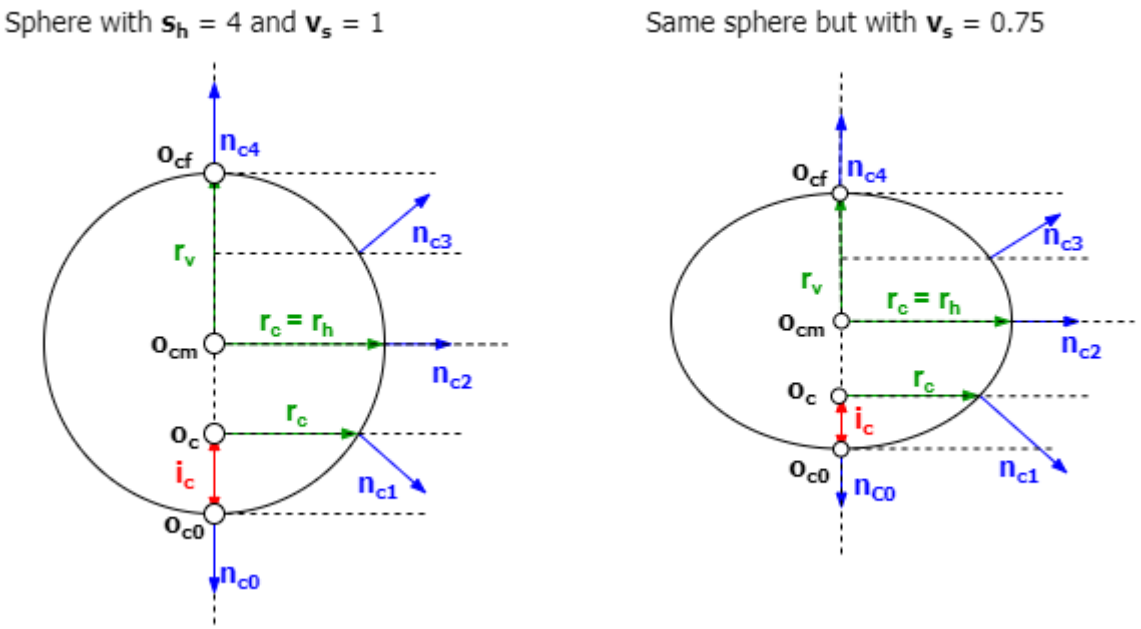

Figure 29: Creation of Sphere primitive through geometry class

The last method in our geometry class is to create basic surfaces with Bezier curves. A Bezier curve can be defined as "A resulting function by linear interpolations in a parametric form" [27].

Our method for creating Bezier curves is similar to the circumference, but instead of using an increment over an iteration, we calculate two points with their directional vectors and calculate a cubic polynomial between the two points, which is used to calculate the radius between the two points.

The starting point is defined as $\mathbf{p_s}$ and the final point will be defined as $\mathbf{p_f}$. Their directional vectors will be $\mathbf{v_s}$ and $\mathbf{v_f}$. We will iterate $\mathbf{t}$ times, defined as $\mathbf{t} = \mathbf{h} / \mathbf{s_h}$. We will calculate our curve $\mathbf{b_z}$ with the next polynomial $\mathbf{b_z}(t) = \mathbf{p_s} * (1 - \mathbf{t})^3 + \mathbf{v_s} * (1 - \mathbf{t})^2 * \mathbf{t} * 3.0f + \mathbf{v_f} * \mathbf{m_t} * \mathbf{t}^2 * 3.0f +$

$\mathbf{p_f} * \mathbf{t}^3$. The centre of each circumference $\mathbf{o_c}$ is $\mathbf{o_c}$ = (0.0f, $\mathbf{b_z.y}$, 0.0f), being translated and rotated to match a correct transform value.

For calculating the normal of the surface, we will calculate its tangent. Tangents are really to calculate as they are the first derivative of our curve. We calculate tangents $\mathbf{t_n}$ following the polynomial $\mathbf{t_n}(t) = \mathbf{p_s} * -(1 - \mathbf{t})^2 + \mathbf{v_s} * ((1 - \mathbf{t})^2 – (2 * \mathbf{t} * (1 - \mathbf{t}))) + \mathbf{v_f} * (-\mathbf{t}^2 + (2 * \mathbf{t} * (1 - \mathbf{t}))) + \mathbf{p_f} * \mathbf{t}^2$. We normalize this vector and calculate the slope of the surface as $\mathbf{n_c}$ = ($\mathbf{t_n.x}$, $\mathbf{t_n.y}$) and use the same method to generate quads that we use with the sphere. In figure 30 we have used an application of "A Primer on Bézier Curves" [27] to create a curve, which we have edited to show the variables we have used.

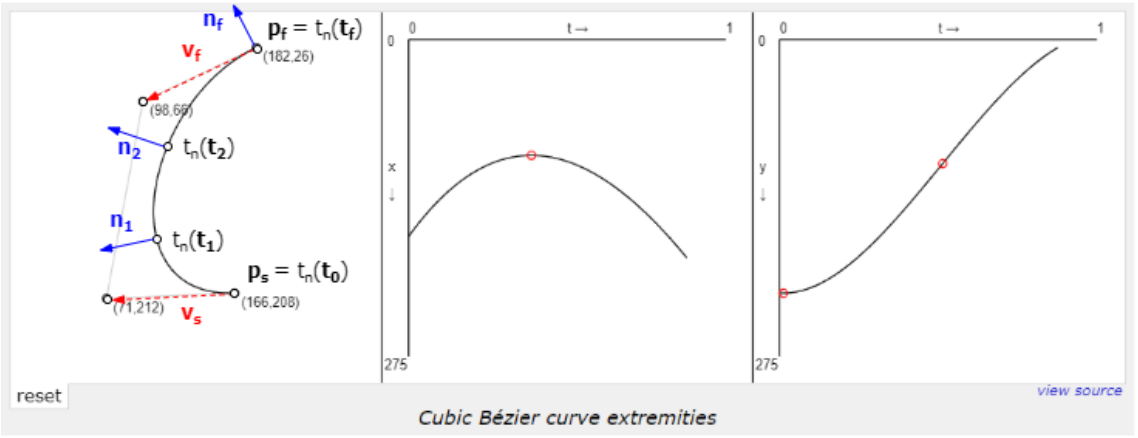

Figure 30: Cubic Bezier curve through geometry class

With all these methods we can procedurally create any mesh as a sum of primitives and/or curves. In our essay we have established four different geometric figures: a cypress, a mushroom, a fir tree and a flower. We start by explaining the cypress as it is the most basic one.

We declare 4 classes for building a cypress: 'DataObject' stores common parameters for the rest of the classes, 'LogData' stores the parameters for the log, 'LeavesData' stores the parameters for the leaves and 'BezierData' saves the parameters for the Bezier curve. 'LogData', 'LeavesData' and 'BezierData' inherit from 'DataObject'. This structure of having common parameters is shared by all the geometric figures.

Sometimes we are interested into not building some parts of the geometric figure because we would generate more variated instances with this. For example, building only a log in a tree.

Therefore, after building something in a geometric figure, we check if we want to build it by looking a Boolean variable called 'build'. This is why we want to have a class where everyone inherits from, because every class share three common variables: 'radial subdivisions', 'height subdivisions' and 'build'.

For creating the log, we build a frustrum with the function 'BuildFrustrum()'. The parameters belong to the public 'LogData' class, except 'isClosed' that it is true if we are not going to build leaves. If we are not going to build leaves, we leave the log closed. Once we have created the log, we store its position and rotation in a 'PassData' class. We use this class to build the leaves in the last position and rotation of the log, so leaves and log are aligned.

We have two possible outcomes for leaves. We can either create a sphere with a vertical scale greater than the horizontal radius or we can use a Bezier curve. To create a sphere, we use the function 'BuildSphere()'. Although effective, this method is not ideal as creating the shape of the leaves as a sphere is not very appealing, looking very regular and dull.

To use a Bezier curve, we will set 'BezierData.build' parameter as true. Then, we calculate the starting and ending position and their directional values. Finally, we use the function 'BuildBezierCurve()'. With this, we can create a surface that resembles more to an actual cypress.

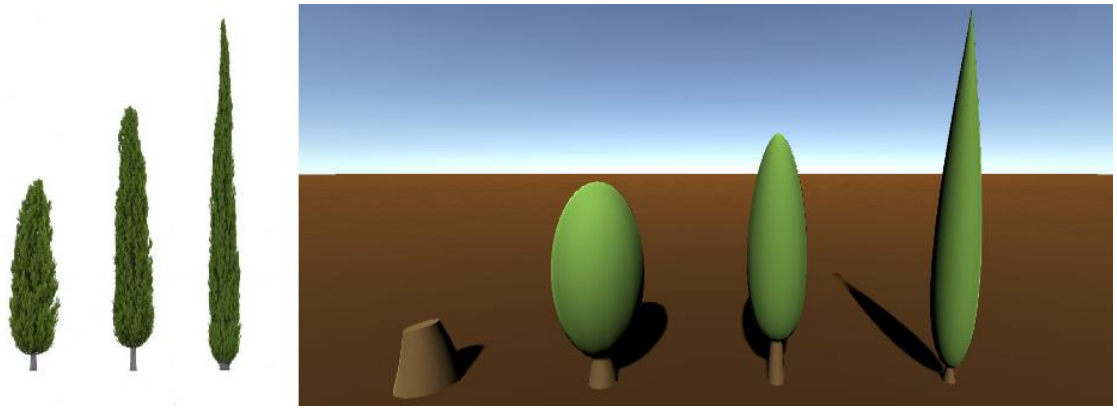

Figure 31: Creation of cypress with visual reference

In figure 31 we can observe different instances of a cypress. On the left, the visual reference that I have used. On the right, we can observe a cypress with only the log, the cypress created with a sphere and vertical scale and two cypresses built with different Bezier curves.

Mushrooms follows a similar structure like cypresses. We create the stem with a cylinder and the cap as a Bezier curve. The only differences are a second Bezier curve to close the inner part of the cap and a thickness variable defined as $\mathbf{t_h}$ to control how thick the cap should be. For closing the inner part of the mushroom, we calculate $\mathbf{p_f'}$ as $\mathbf{p_f'.y} = \mathbf{pf.y} - \mathbf{t_h}$ and the directional vector $\mathbf{v_s'} = (-\mathbf{v_s.y}, \mathbf{v_s.x}, 0.0f)$ being $\mathbf{v_s'} \perp \mathbf{v_s}$.

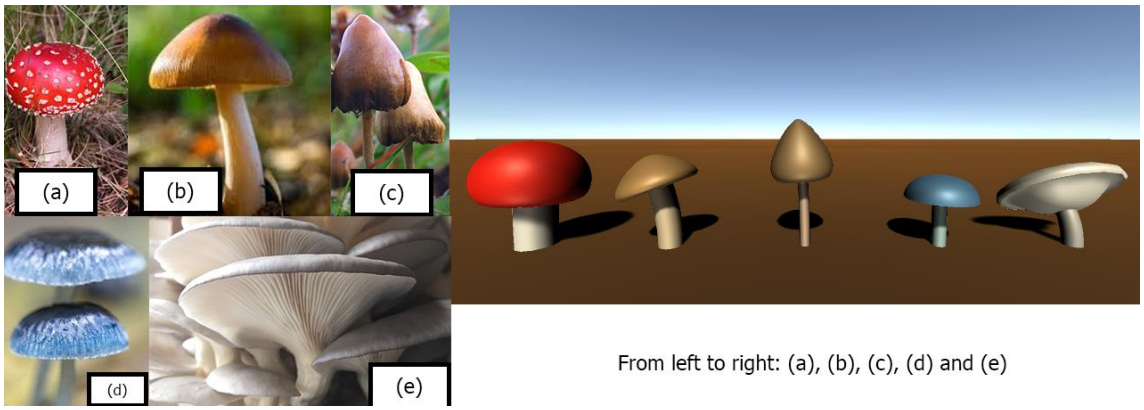

Figure 32: Creation of mushrooms with visual references

The classification of mushrooms in figure 32 are: (a) Amanita Muscaria, (b) Psilocybe Azurescens, (c) Pscilocybe Semilanceata, (d) Lactarius Indigo and (e) Pleurotus Ostreatus.

Firs have a similar structure to mushrooms and cypresses. We create the log with a frustrum with radius $\mathbf{r_l}$ and then we will iterate per each ring of the fir defined as $\mathbf{n_t}$. Like the cypress, we can build a geometric primitive or use a Bezier curve, although both methods share the same structure.

We create a frustrum with $\mathbf{r_f} = \mathbf{r_l}$ and $\mathbf{r_t} =$ `Mathf.Lerp(`$\mathbf{r_s}$, $\mathbf{r_f}$, (float)$\mathbf{t}$ / $\mathbf{n_t}$`)`, being $\mathbf{t}$ the actual iteration. If $\mathbf{t} = 0$, then $\mathbf{r_f} = 0$. We will do the same process with Bezier curves by changing 'BuildFrustrum' function to 'BuildTreeSectionBezier' that calculates both external and inner Bezier curves of the fir.

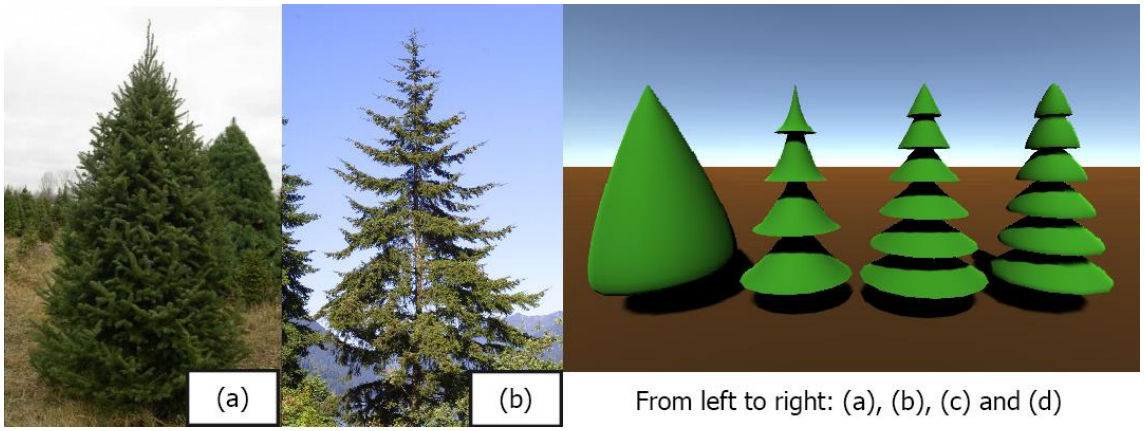

Figure 33: Creation of fir trees with visual references

The classification of fir trees in figure 33 are: (a) Abies Balsamea, (b) Pseudotsuga Menziessii, (c) with frustrum's primitives and (d) with Bezier curves.

Flowers are more complex. We will start by creating a cylinder and a sphere with a small vertical scale to resemble a stem and a disk of a flower. Then, we use a use the 'BuildPetalsRing' to create a ring of petals. This function iterates $\mathbf{n_p}$ times calculating the position and rotation of each petal, calculating random start angle defined by β and random bending angle defined as ϕ.

The total bending of a petal is defined as $\chi = \alpha + \phi$, being α a fixed bending angle we will always apply. With these two angles we can generate a single petal with a random starting rotation and a random bending angle.

The process of generating a petal is to create several quads into a direction. We calculate the increment of each petal as $\mathbf{i_p} = \alpha$ * Mathf.deg2Rad / $\mathbf{n_p}$. Then, the centre of each quad will be equal to $\mathbf{o_c}$ = (0.0f, `Mathf.Cos`($\mathbf{i_p}$ * **t** + β * Mathf.Deg2Rad), `Mathf.Sin`($\mathbf{i_p}$ * **t** + β * Mathf.Deg2Rad)), being **t** the actual iteration.

With this, we create curved planes towards up. As we want to generate petals and not planes, we create an offset **p** that behaves like a sine through a semi-circumference: at the beginning and at the end of the plane, the width is zero, while at the middle the width is at is maximum. Lastly, we generate the same petals but pointing outwards, so it is visible for both sides.

We generate a second ring of petals in order to create the sepals of a flower. With this, we can create flowers that heavily resembles to the Asteraceae family of flowers, which petals are grouped in chapters [28].

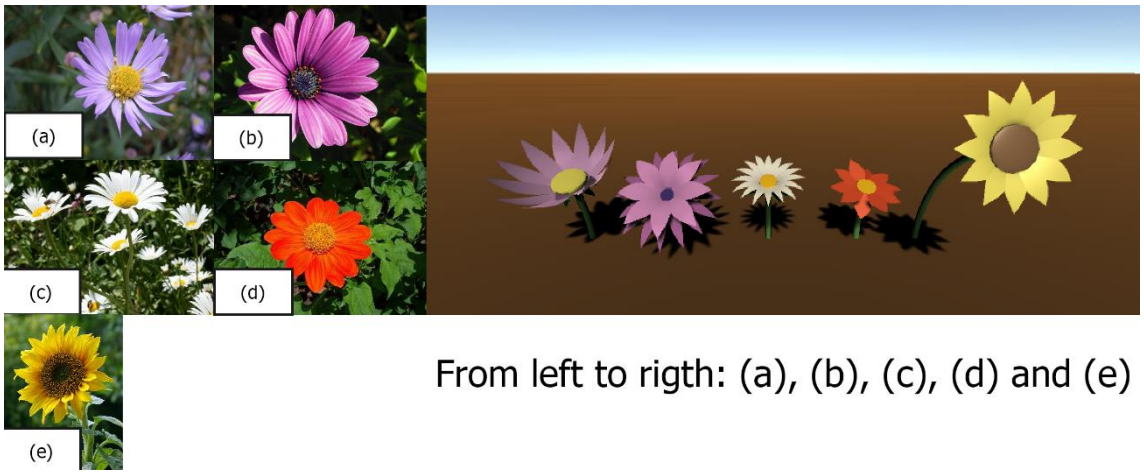

Figure 34: Creation of flowers with visual reference

The classification of flowers in figure 34 are: (a) Symphyotrichum novi-belgii, (b) Dimorphotheca ecklonis, (c) Leucanthemum vulgare, (d) Helianthus annuus and (e) Tithonia rotundifolia.

5.2. Procedural Modification of Organic Geometry with GPU

5.2.1. *Grass without Tessellation*

One of the objectives we set to accomplish in the development was to create grass. Although at the beginning we thought that it wouldn't be so difficult to implement through CPU, in reality turned out to be very resource-consuming: a lot of processes were executed, and the performing was unstable.

While searching for more information, we found Roystan's approach [17] to build grass. It was based on geometry shaders, which is a part of the rendering pipeline. It takes a topology as input and the outputs zero or more vertices. This output is a 'TriangleStream', which represents a triangle strip (a series of joined triangles sharing vertices). It takes as a parameter the vertices array $\mathbf{V_a[3]}$, which will have size three as we take as input a triangle.

To create more vertices, we append the position of our vertex in clip space (which is the space of the object normalized in with the w component of the camera). This way we assure to not create triangles in world position.

As the topology of the terrain is not always a terrain, we need to build the blades of grass in tangent space. Figure 35 shows a representation for what a tangent space is for a vertex **v**, being **n** the normal, **t** the tangent and **b** the binormal.

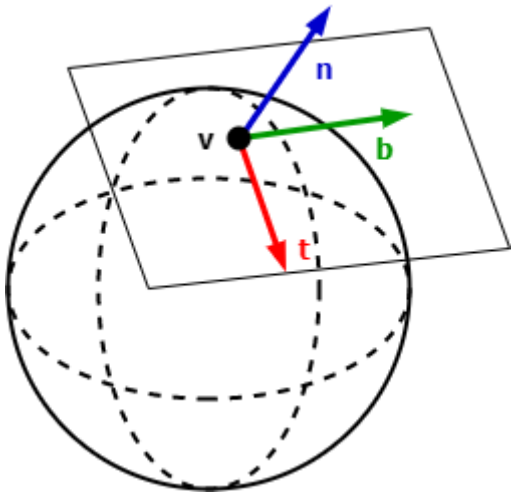

Figure 35: Visualization of a tangent space for a vertex v

We need two structs to define the tangent space: 'VertexInput' and 'VertexOutput'. 'VertexInput' takes the position, normal and tangent of the vertex as input and 'VertexOutput' is the struct data that outputs the vertex shader. In this process, we say that the values of 'VertexOutput' are the ones from 'VertexInput'. Having the normal and tangent, we can calculate the binormal as **b** = **n** x **t**.

We create a float3x3 matrix $\mathbf{T_s}$ with the value of each row the component x, y and z. Each column corresponds to **t**, **b** and **n**. We take as offset the position of the first vertex as $\mathbf{p_c} = \mathbf{V_a[0]}$. We can create basic triangles by creating three points ($\mathbf{p_0}$ = -0.5f, $\mathbf{p_1}$ = 0.0f, $\mathbf{p_2}$ = 0.5f) and calculating the final position by multiplying $\mathbf{p_n}$ to $\mathbf{T_s}$, adding $\mathbf{p_c}$ and transform the result into clip space.

Now that we generate a triangle per each vertex, we need parameters to make each of them different. We transform each blade of grass so we can customize its width, height, a bending angle in the X-Axis and rotation in the Y-axis. Therefore, we need to define two functions: one to

give us random values and a second to which we can generate any float3x3 matrix in a given axis. With these two we can generate any transformation with random values.

We start by createing a 'RandomGenerator.cginc' file in which we will create functions that return us random values for each axis. This implementation is the same that Ronja Böhringer does for creating white noise [29].

We take a value **v**, calculate the sine of **v** in order to produce a small value, produce a scalar value from a random direction, multiply the result by a big number and return the result. We can do this method per each direction. For creating a float3x3 matrix, we have used the implementation of Keijiro Takahashi [30].

For creating blades with random rotation in the Y-axis, we generate a random value from $\mathbf{p_c}$ and multiply it by 2 times π in the Y-axis. The returned value is a float3x3 matrix defined as $\mathbf{R_y}(\mathbf{p_c})$. To create a rotation through the X-axis, we generate a random value from $\mathbf{p_c}$ through the X-Axis and multiply it to half π as we are interested into rotating it 90 degrees. The result will be a float3x3 matrix $\mathbf{R_z}(\mathbf{p_c})$.

For creating blades of grass with a certain width and height, instead of defining $\mathbf{p_0}$ = -0.5f, $\mathbf{p_1}$ = 0.0f, $\mathbf{p_2}$ = 0.5f with those values, we will define them as $\mathbf{p_0}$ = -width * 0.5f, $\mathbf{p_1}$ = height, $\mathbf{p_2}$ = width * 0.5f. Finally, we will need to combine all these processes. As matrices are not commutative, the final transformation matrix will be $\mathbf{T_f} = \mathbf{R_z}(\mathbf{R_y}(\mathbf{T_s}(\mathbf{p_c})))$.

To make the grass more organic, we can subdivide each blade and add a rotation from the base. Instead of hardcoding the position of each vertex, we can change the length of the triangle strip as $\mathbf{V_a} = \mathbf{n_b} * 2 + 1$, where $\mathbf{n_b}$ is the number of triangles per each blade.

We add the position of each vertex with a loop into 'TriangleStream' and, once we finish the loop, we add the final vertex as the tip of the blade. We calculate the width of each blade as $\mathbf{w_n}$ = width * (1 – (**t** / (float)$\mathbf{n_b}$)) and $\mathbf{h_n}$ = height * (t / (float)$\mathbf{n_b}$).

For describing the curvature of the blade, we calculate the amount of rotation with a random value calculated with the position multiplied by a scalar that determines how forward it should be. Then, we create as parameter a float that determines the slope of the curve.

We calculate the power of that slope with **t**, the actual iteration, and multiply it by the amount of rotation. By calculating the angle of the triangles with a power, the rotation offset is not be linear, creating a curve.

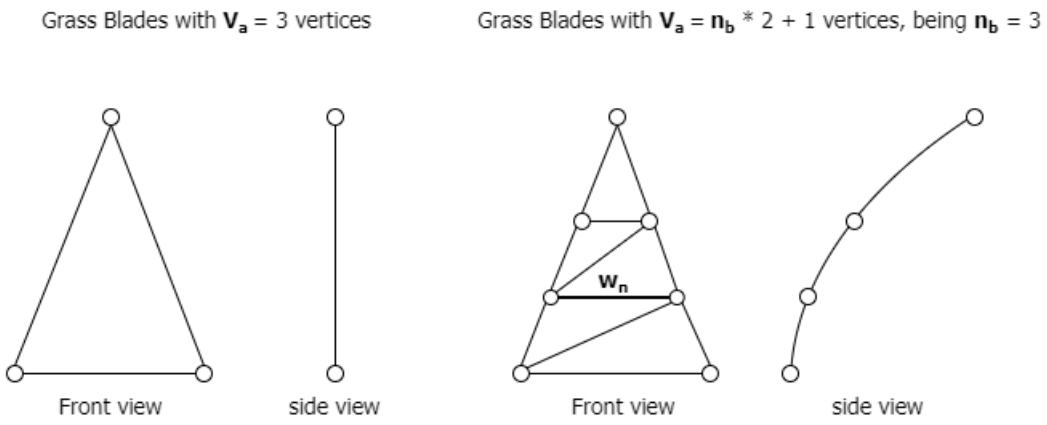

Figure 36: Blades with no subdivision (Left) VS Blades with subdivisions (Right)

We are generating blades, but as we have not assigned any colour, the output are white triangles. We can define one colour and applying it, but the Roytan's solution allow us more customization: instead of applying one colour, we can define two colours (one for the base of the triangle and a second to the top) and assign an interpolated colour depending on the Two-dimensional position of the vertex. To use this, we need a fragment shader.

Fragment shaders can be defined as "a program that runs on each and every pixel that occupies on-screen, used to calculate and output the colour of each pixel" [31]. We assign the colour to each pixel as the interpolation of the vertex's uv in the Y-axis between the bottom colour and the top colour.

Both colours are multiplied by the light intensity. The intensity of the light $\mathbf{l_i}$ can be calculated as the dot product between the normal of the surface $\mathbf{n_s}$ and the direction $\mathbf{l_d}$ of the directional light, $\mathbf{l_i} = \mathbf{n_s} \cdot \mathbf{l_d}$.

We allow the shader to receive and cast shadows by using a subshader with different passes: one for casting shadows with a fragment shader that uses the function

`SHADOW_CASTER_FRAGMENT` and `SHADOW_ATTENUATION` to cast shadows. Also, we need to activate the necessary pragmas, which are the compilation directives. Some of these are `#pragma fragment`, `#pragma geometry`, `#pragma domain` or `#pragma hull`. These allow us to compile the different functions that we have created.

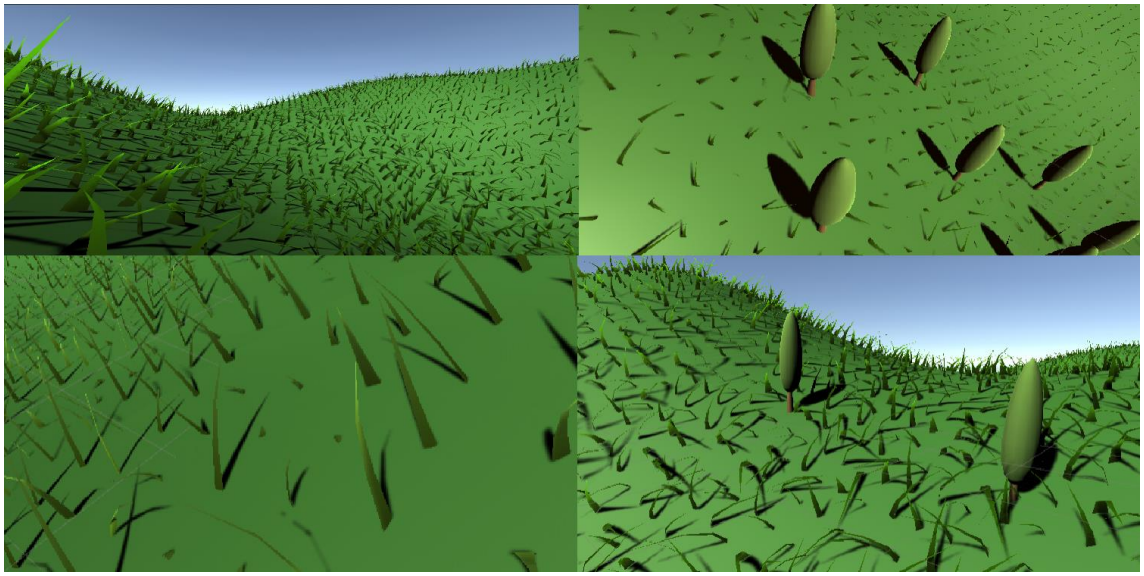

Figure 37: Visualization of grass without tessellation

The result of all these processes is shown in figure37. We can observe how the blades of grass are created per each vertex of the terrain, how each blade of grass is made with the interpolation of two colours, how the shadows affect the grass and also how the grass produces shadows.

Although it is not a bad result, we do not want to generate grass depending on the vertices of the terrain, as a terrain with a small Level of Detail (LOD) would not produce a satisfactory result. We will use tessellation in order to produce more vertices, producing a more organic result.

5.2.2. *Grass with Tessellation*

Tessellation allow us to generate more triangles by subdividing one into multiple triangles. This solution allows us to generate blades of grass without being placed like a grid, giving a more organic appealing. Although tessellation through CPU can be done as well, tessellation is mostly associated with the GPU as it is one of the processes right after the vertex shader.

Roystan's implementation into grass did also included a .cginc file, which is an implementation by Jasper Flick [16]. Adding this .cingc file to a shader (including it into the CGINCLUDE section) and declaring a variable as '_TessellationUniform' that allows to control the tessellation factor. We also have to include the pragmas `#pragma hull` and `#pragma domain`.

Tessellation stage works internally with a Hull program and a Domain program. The hull program determines on which surface are we working on (triangles, quads or isolines) and feeds it with necessary data. Once that it is determined how the surface should be divided, the domain shader evaluates and generates the vertices for the final triangle.

Hull shaders need as input a patch, which is a collection of mesh vertices. Each patch contains three vertices as we are operating with triangles. The hull function is called once per vertex in the patch with additional arguments, that specifies which control points should work with, and outputs a single vertex.

We have to set a few parameters to the GPU. We will have to set as 'tri' the `UNITY_domain` to clarify we are working with triangles, '3' to `UNITY_outputcontrolpoints` to tell we output three vertices per patch, 'triangle_cw' to `UNITY_outputtopology` to work with a CW winding order, 'integer' to `UNITY_partitioning` to set the partitioning method and 'ConstantFunction()' to `UNITY_patchconstantfunc` which is a method that returns the tessellation factor which tells the GPU into how much parts the patch should be cut.

The next part in tessellation corresponds to the tessellator, which we cannot modify. The tessellator returns us the barycentric coordinates of the vertex, which we use them in the domain shader to interpolate them and calculate the final position of the vertex. Not only position, we have to interpolate normal, tangent and uvs. Applying it to all vertices, we can now subdivide patches creating uniformed triangles.

Applied to the grass geometry shader, we control the creation of more vertices and thus, more grass. We control the tessellation factor with a range between 1 and 10, allowing us to produce no tessellation if we want.

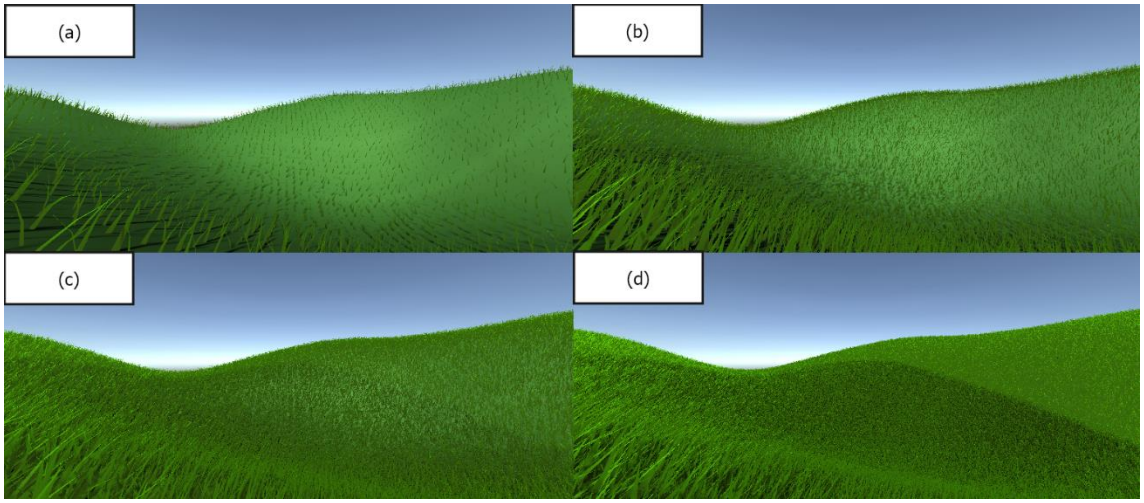

Figure 38: Visualization of grass with multiple tessellation factors

Figure 38 is a visualization of the same field with different tessellation factors: (a) has a factor of 1 (tessellation is not applied), (b) has a factor of two and half, (c) has a factor of five and (d) has a factor of ten.

5.3. Procedural Modification of Inorganic Geometry

We can create multiple inorganic objects the same way we can do it for organic. For our essay, we have delimitated that the inorganic approach would be focused on terrain. We show three ways of creating terrain: with a heightmap as input (producing always the same result), with a random class (we can generate noise that resembles to a terrain) and with a fractal algorithm (to show an example of the applications of fractals in PCG).

In practice, all these ways share a similar structure to the procedural modification of organic geometry of the second approach. Figure 39 shows the class implementation diagram, using as an example 'TerrainHeightMap', which is the class responsible of building a terrain given a heightmap texture.

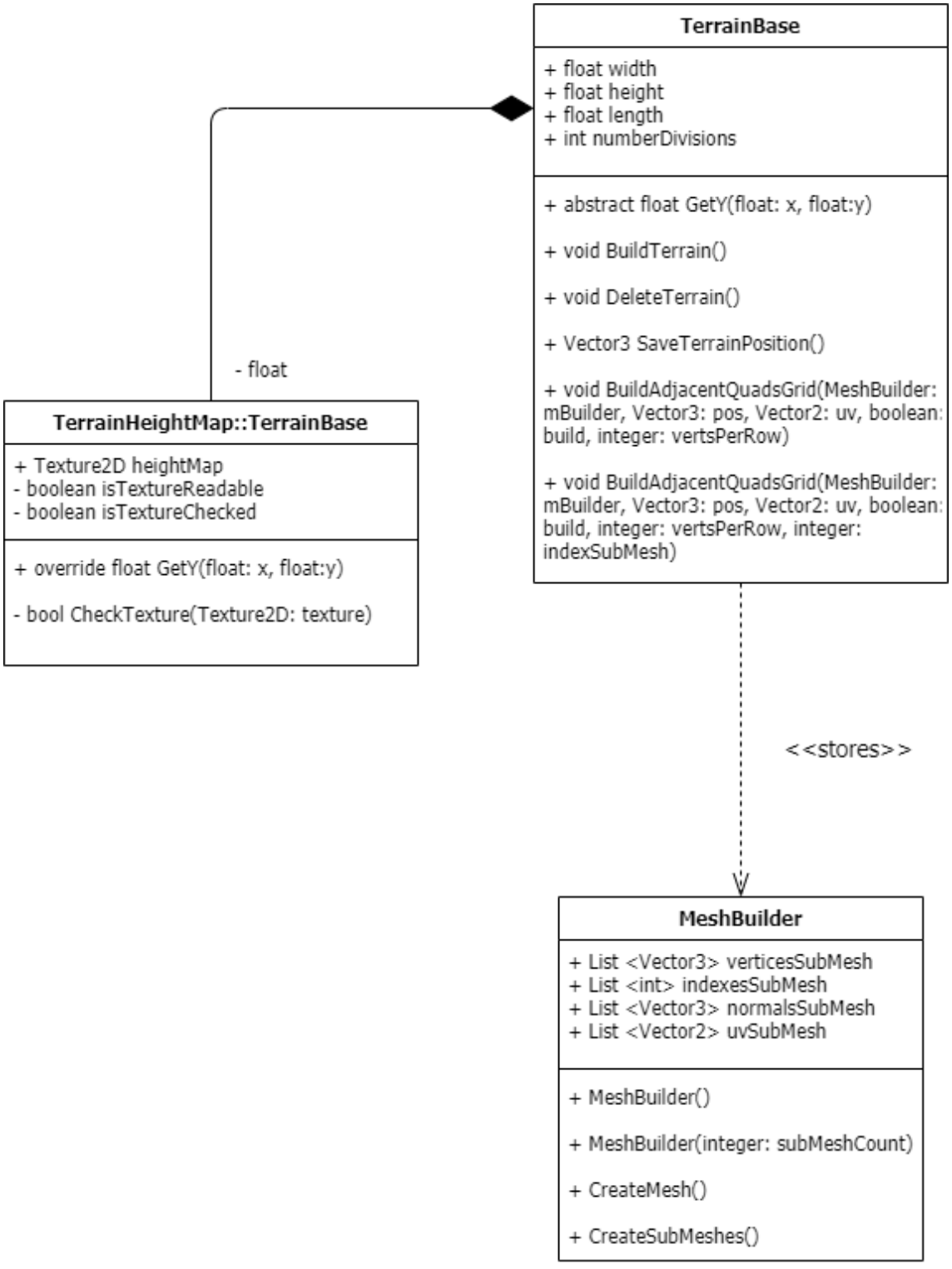

Figure 39: Class implementation diagram for building a terrain using a heightmap

The three types of terrain share some common structure. First of all, we normalize the position of the terrain, so it is placed correctly. This process is done by subtracting half of the width,

length and height to the actual position, so no matter the size of the terrain it is always created in the same placed.

Then, we create the terrain by dividing the width and length by the number of divisions, which controls the LOD.The greater this value, the more subdivisions the terrain will have, being more complex and detailed.

We loop two times, creating first the rows and then the columns. This led us to the creation of a terrain with the same distance between rows and columns, where in each vertex we have to calculate the height of the vertex. Once we have calculated the height, we add the vertex to a MeshBuilder. If it is not in the edge of the terrain, we add a triangle following a CW winding order. Once we have finished the two loops, we finish by creating the mesh.

Therefore, the way we calculate the height of each vertex is determined by each method. In a terrain created by a heightmap texture, we use as parameter the width and the length. If we can read/write the texture, we normalize those positions and calculate the pixel in the texture with those X-Y coordinates. That value will have a value between zero and one, so if we multiply it by the total height, we have the height for our vertex.

This method is fast as it only checks a position in a texture, useful if we want always to have the same output and it allows artists to create a terrain with a specific or complex shape. The problems are that we can only generate one result per texture and slower in terms of production time, as an artist has to draw a texture per terrain.

Perlin noise is a common solution in terrain generation. It allows us to calculate patches of random heights which can be modified. We create a 'PerlinNoiseValues' class with scale, octaves, persistence, lacunarity, offset and seed as parameters. We start with a function that calculates the seed, so this way we can generate always the same terrain as long as we save the seed.

The method for calculating the seed returns a Vector2 per each octave, which contains offsets for the X and Z axis. Those offsets are calculated with a random class created with the seed and with the offset introduced as a parameter to the 'PerlinNoiseValues' class. This way we have random values but at the same time we can control them via parameters.

To calculate the height of the vertex, we iterate with the number of octaves, generating a random Perlin noise value. This value is calculated with the function `Mathf.PerlinNoise`(perlinX, perlinZ). Those parameters, perlinX and perlinZ, are calculated with the same method but using their correspondent values: perlinX with the values of the X-axis and perlinY with the values of the Z-axis.

We take as the position of the vertex in the X/Z-axis, divide it by the scale and frequency of the 'PerlinNoiseValues' class and add their correspondent offset. The final noise value is added to a variable which be the final output and stores the previous iterations multiplied by the amplitude. In each iteration, the amplitude and frequency are multiplied by the persistence and frequency respectively. Finally, we normalize the height, so it is between zero and the maximum height.

Perlin noise is often used as it is a method that allows a great control over the terrain we want to build, so we will use it in this essay as the default creation of terrain method. It allows us to control the scale without changing the width and length, how rugged is the final output and the resource management.

The final method that we have implemented in our essay is the Diamond-Square algorithm, which is a fractal algorithm. Basically, we generate five new vertices per each quad: one in the centre and the other four in the middle of the quad's edges. Once that we have generated these new vertices, as we have new quads in our surface, we repeat the previous step the necessary number of times.

To do this, we create first a 2DTexture which we fill with our data and, once it is created, we use the same method that we do with the heightmap texture. The process of generating the texture is where the Diamond-Square logic is implemented.

We generate an array which is filled with random values from minus one to one. We calculate the length of each step of the algorithm based on the LOD of the terrain divided by the noise level. The bigger the noise parameter, the less we iterate and thus the height of each vertex gets more random.

In each iteration we call two functions: 'DiamondStep()' and 'SquareStep()'. 'DiamondStep()' calculates the centre of the middle vertex with the average of the four vertex corners and add a small random value. 'SquareStep()' does the same thing but except averaging the four diagonal

vertices. We call 'SquareStep()' two times instead of one, so it generates two vertices per row and column, having each quad at the end four new vertices.

We normalize those values and create a new texture, which we use to check the position of each vertex of the terrain. Figure 40 shows a visualization of 'DiamondStep()' and 'SquareStep()' functions.

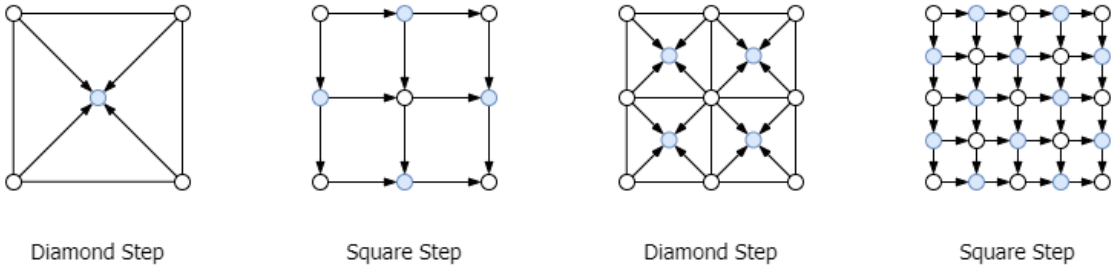

Figure 40: Visualization of Diamond-Square algorithm

As we are not interested into creating just a mesh of one colour / material, we will follow the same procedure that we do with the geometry figures: we separate the mesh into different subMeshes, each one with a different material. The only problem is that the grass only outputs grass, not showing the surface of it is created. We will fix this by creating a second 'MeshBuilder' and building a mesh only for the grass, generating a triangle only when we are on a 'grass' area.

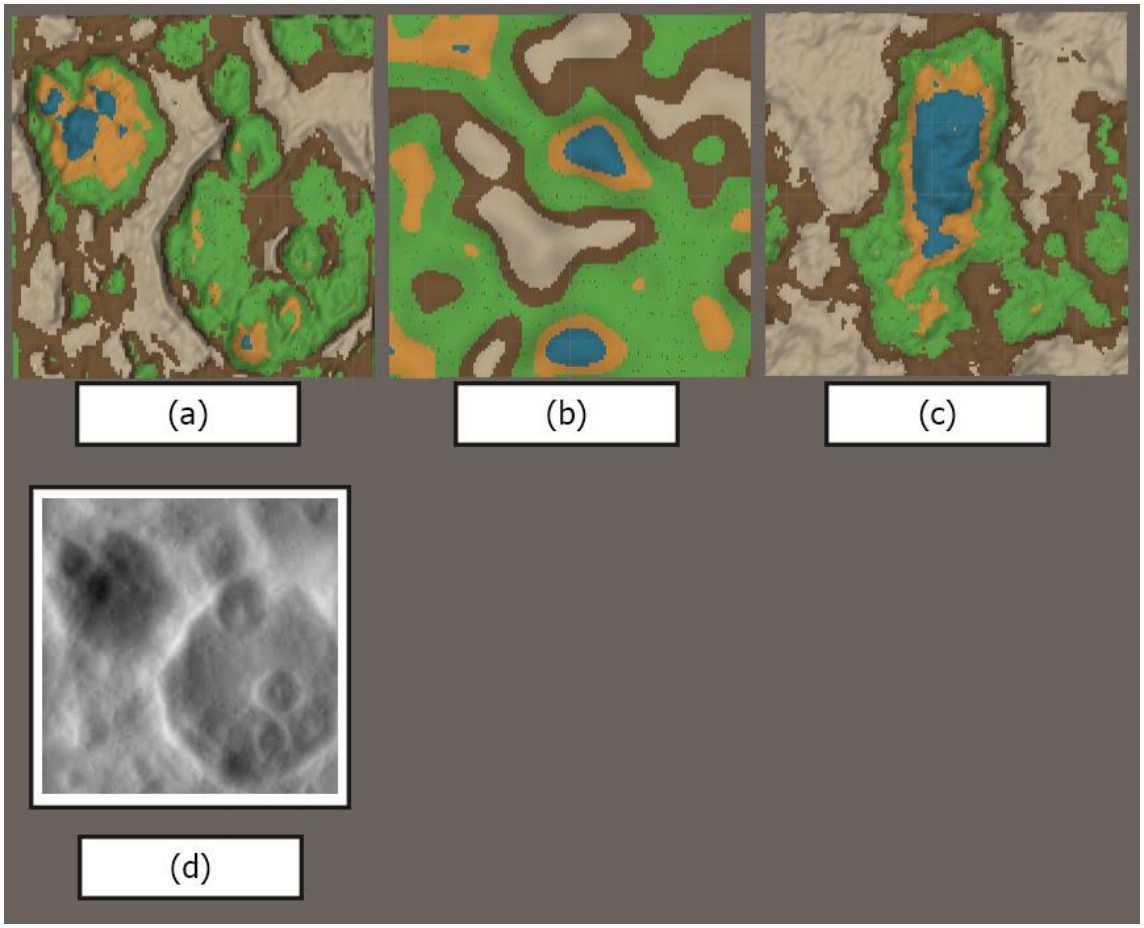

Figure 41: Visualization of different terrain methods

Figure 41 shows the visualization of the three types of terrain with different areas: water, beach, grass, mountain and snow. The three terrains are created with the same common parameters (those are the areas parameters and a LOD of 125).

The different terrains correspond to: (a) by heightmap texture, (b) by Perlin noise and (c) by Diamond-Square algorithm. (d) Corresponds to the heightmap texture applied to terrain (a), so we can check that it works as it should be.

5.4. Procedural Distribution of Objects

We distribute the objects with a simple random process. We assign to which areas do we want to build our objects and place them in the terrain with their correspondent height. To do this, we have to change a few things on the 'TerrainBase' class.

We have changed the areas to include a Vector3 list. This list stores the position of each vertex if the condition of building a triangle is true. With this, we can store the position of each vertex. As we have included it in the areas, we know the position of each vertex in each area, allowing us to choose in which areas do we only want to build.

Only with these lists we can start placing objects in the scenario. We started by rocks. Although we could investigate or tried to implement a method to generate rocks through a procedural algorithm, we would be out of the scope of the project.

Instead, we have modelled five different models in Blender. With these five models, we could generate more instances as prefabs, which are game objects of unity saved as invocable instances. We created five prefabs mixing the original five models, making a total of ten different models.

The script that distributes the rocks through the terrain needs as parameters the terrain where to distribute the objects and some parameters for us to randomize more the rocks. For example, we have created some transforms parameters to scale the rocks with some variances: to scale them with different values, scale them with different values in the three axes, to scale them randomly or the scalar value to which we should scale them.

To select in which area should we build our objects, we have introduced a string that checks that, if its value is equal to the name of the area, we have access to the list position. This list is changed into an array so we can point a random position. We also have a parameter that selects how many objects we instance.

We instance an object in a random position, with a random rotation in the Y-axis (although rocks could be instantiated with a random rotation in the three axes, we have chosen to only rotate in the Y-axis) and with a scale set by the parameters. We have used this script as well to instantiate the water, which is a translucid plane normalized with the terrain's position with the height of the water's area.

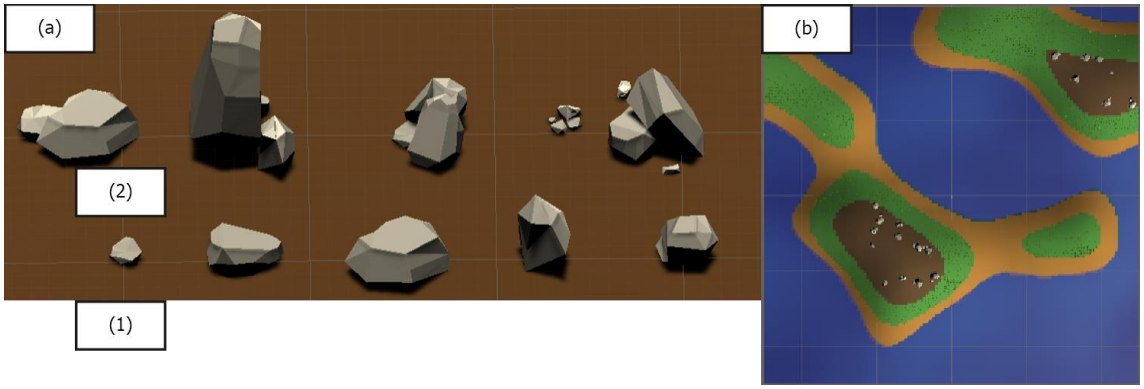

Figure 42: (Left) Rocks created manually and (Right) placed on the terrain

Figure 42 shows how twenty rocks are distributed in a "Mountain" area with the water. In image (a), we can observe two rows: row (1) are the models made in blender while in row (2) are the prefabs made with a combination of models of row (1). Image (b) shows a top view of a terrain where the rocks have been distributed along the mountain area (with brown colour in the image).

To apply the geometric figures in a terrain, we have used a similar procedure where we have a set of prefabs with the code corresponding to a geometric figure and placed them while we initialize them in the terrain. The parameters of those geometric figures are fixed to resemble the visual reference that we have used in this essay.

5.5. Procedural Interaction with the environment

We have decided that the interaction between the end-user and the PCG would be through the wind. The end-user is able to select the rotation of the wind and its magnitude, making the objects able to rotate in that direction with certain frequency. To this frequency we have added perlin noise, so it has a more organic resemble.

We have modelled an arrow in Blender, so we have a visual feedback to where the wind is pointing to. We create two scripts: 'WindControl' to control the wind and 'WindForce' to apply the wind to a geometric figure. Thanks to the script attached to each geometric figure, we are able to control the maximum angle of rotation, the intensity of Perlin noise and the seed of the random class.

In 'WindControl', we start by generating a random rotation between 0 and 360, although we can input that direction manually. We transform that value into a quaternion thanks to `Quaternion.Euler()` and only in the Y-axis as we only want the arrow to rotate in the Y-axis.

Also, we have to subtract 90º to the rotation as the angle of the arrow's model is not pointing correctly at the origin value (when the rotation is zero, it points downwards). We can change the rotation input as the scripts detects when it has a new rotation and rotate it at the moment.

'WindForce' takes as input a 'WindControl' class which takes the direction in Y and transform it into radians, defined as $\mathbf{v_w}$ = WindControl.directionY * π * 2 / 360.0f. In this case, we want to rotate the objects in X-Z axes and not into Y-axis, as objects do not rotate in the Y-axis when the wind is applied to them.

With the Y-rotation value, we generate a quaternion as $\mathbf{q_w}$ = `Quaternion.Euler`($\mathbf{d_w}$), being $\mathbf{d_w}$ a directional vector as $\mathbf{d_w}$ = (`Mathf.Sin`($\mathbf{v_w}$), 0.0f, -`Mathf.Cos`($\mathbf{v_w}$)). We apply $\mathbf{q_w}$ to the rotation of the geometric figure because the geometric figure is built in the origin of the object, so the rotation is applied in the correct position.

However, with this we only set the rotation of the object but not making it move with the wind. What we need is to create a sine value that changes over time. To achieve this, we have created a value $\mathbf{m_w}$ equals to $\mathbf{m_w}$ = `Mathf.Abs`(`Mathf.Sin`($\mathbf{d_t}$ * WindControl.Magnitude)), being $\mathbf{d_t}$ = Time.deltaTime. mf generates a sine wave value over time with frequency the magnitude of the wind, so if the wind is stronger, objects rotate with more frequency.

We will create a Perlin noise value $\mathbf{p_n}$ so each frame the wind has a random value, making it more organic. This Perlin noise needs two X-Y parameters, which will be $\mathbf{p_n}$ = ((seed.x + $\mathbf{m_w}$) * $\mathbf{i_w}$, (seed.y + $\mathbf{m_w}$) * $\mathbf{i_w}$), being $\mathbf{i_w}$ the intensity of the wind for this particular object. The final value for the quaternion $\mathbf{q_f}$ is $\mathbf{q_f}$ = $\mathbf{q_w}$ * $\mathbf{m_w}$ * $\mathbf{a_r}$ * $\mathbf{p_n}$, being $\mathbf{a_r}$ the maximum degree the object can rotate.

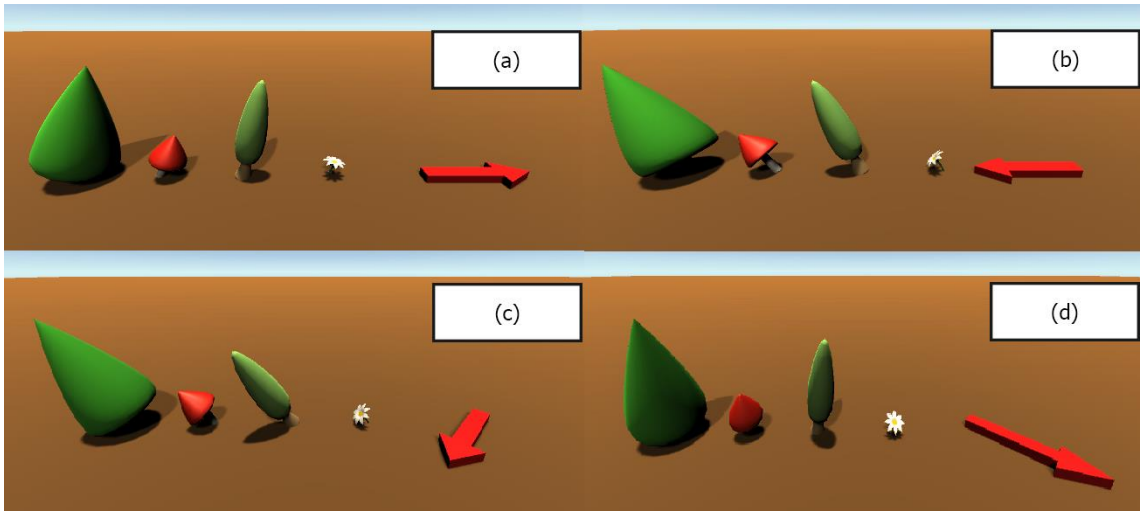

Figure 43: Geometric Figures moved with wind

We can see the application of this method in figure 43, although not in motion: (a) has a Y-rotation value of 0, (b) has this value with 180, (c) with 225 and (d) with 315.

With this we can simulate the wind for geometric figures, but not for the wind. Roytan's implementation has a way of simulating wind through a 2D texture, but we are not interested as we cannot control the wind's direction. Instead, we have made a different approach.

We have created a new script called 'WindTerrain', which is used to calculate $\mathbf{d_w}$, control the amplitude and the frequency of the wind. We have this script because 'WindForce' rotates the object to which it is attached and because we use it exclusively to pass information between 'WindControl' and the grass shader.

We need to use four new parameters for the grass shader: the wind's direction $\mathbf{d_w}$, the wind's magnitude raw value $\mathbf{m_w}$, the wind's frequency waves $\mathbf{f_w}$ and the wind's amplitude $\mathbf{a_w}$. We want not only to rotate the blades of grass, but also to make waves of wind to simulate large fields of grass.

To do this, we have to calculate the float2 vector **p** which is equal to **p** = **pos.xz** · `normalize(float2(-`$\mathbf{d_w}$`.z, -`$\mathbf{d_w}$`.x))`, where **pos** is the position of each vertex. This relationship allows us to calculate the direction of the wind's wave in the terrain. With this, we can create a float2 vector **w** that **w** = -**p** * $\mathbf{f_w}$ + $\mathbf{m_w}$ * Time.y (Time.y is equal to Time.deltaTime for shaders). By multiply **p** by $\mathbf{f_w}$ we have related the wave of wind to a controllable frequency, and by adding

the magnitude with the time, the greater the wind's magnitude is, the more it moves each frame time.

We sample this result with a float2 vector $\mathbf{v_w}$ that $\mathbf{v_w}$ = `float2`(`cos`(**w.x**), `sin`(**w.y**)) * $\mathbf{a_w}$ * π, which produces a rotation in a circumference. We control the amplitude's value to not be greater than 0.5f as we want the blades to rotate with 180º at maximum. With this we produce a sine wave which amplitude is controlled by $\mathbf{a_w}$.

Now that we have the sine wave, we have to select of which axes is going to rotate. We want to rotate with the logic we have followed in 'WindForce', so we create a float3 $\mathbf{w_{xyz}}$ = `normalize`(`float3`(**d_w.x**, -**d_w.z**, 0.0f)), which is our rotation axis. Our matrix **W($\mathbf{v_w}$, $\mathbf{w_{xyz}}$)** has to be concatenated with the final transform matrix to rotate the blades in the wind's direction.

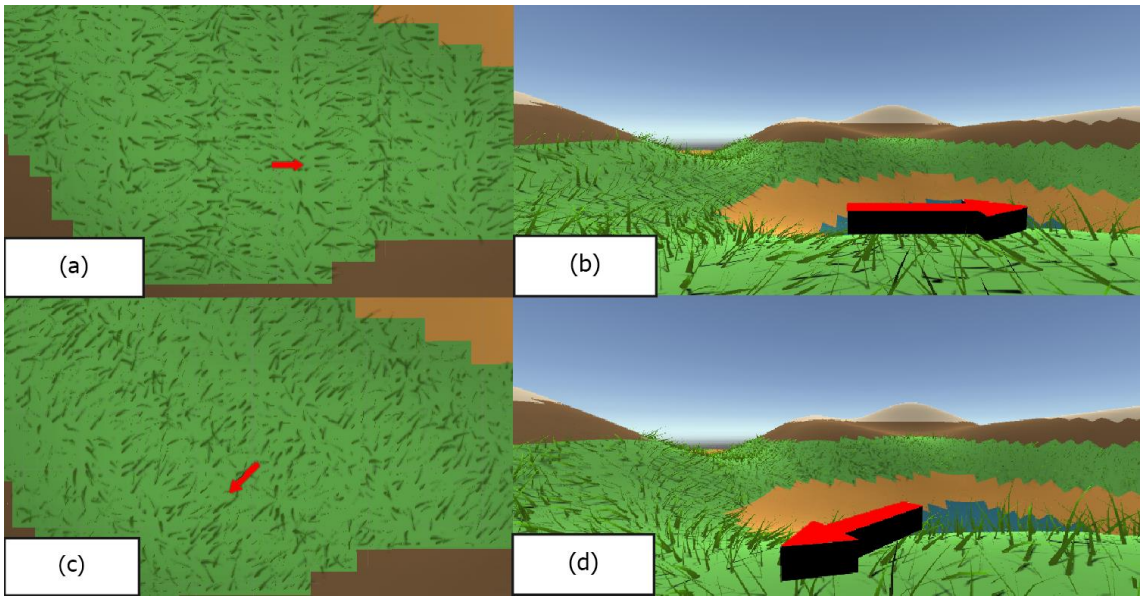

Figure 44: Wind applied to grass into a direction

Although it is difficult to demonstrate as we cannot see it in motion, we can observe in figure 44 how the grass is moved into the direction of the red arrow and how the motion of the blades generates a resemblance with wind's waves. (a) - (b) corresponds to the same rotation (Y-rotation value equals to zero) and (c) - (d) as well (Y-rotation value equals to 225).

Finally, to give more visual feedback, we scale the model of the arrow in the Z-axis with value one plus the magnitude of the wind multiplied by its sine value per time, so we can visually

understand that increasing the magnitude of the wind makes the objects move with more frequency.

6. Economic Study

This chapter covers the economic study of the project. We include in this study the human resource, the software licenses and the development time that has taken for the demo and this document.

Using 'Glassdoor' to search for an average junior salary as technical artist [32], we have found that the salary oscillates between 20.000 and 30.000 euros, although searching for an average salary for a game developer the salary reaches 33.000 euros as an average.

As our activity has been closer to the work of a technical artist rather than a unspecify game developer (as a game developer could also integrate other activities that we have not covered) we consider the salary closer to a technical artist.

We set it at 25.000 euros, to which we have to add the 30% of Spanish's social security, making a total of 32.500 euros per year. If we assume 1.780 hours of work per year, the result is 18,258 euros per hour and 2.708 euros per month.

The work of this essay is estimated to be around six hundred hours work. Although the development of this essay started in December 2019, we have not worked in this project 8 hours per day in a weekday. Therefore, those six hundred hours corresponds more likely to four months of full work and 150 hours of work per month.

From that time, almost 45% of that time has gone to the implementation of the demo and the development of the work, 25% to the elaboration of this essay, 20% to the research of different techniques and information and 10% to the study and elaboration of the internal logic.

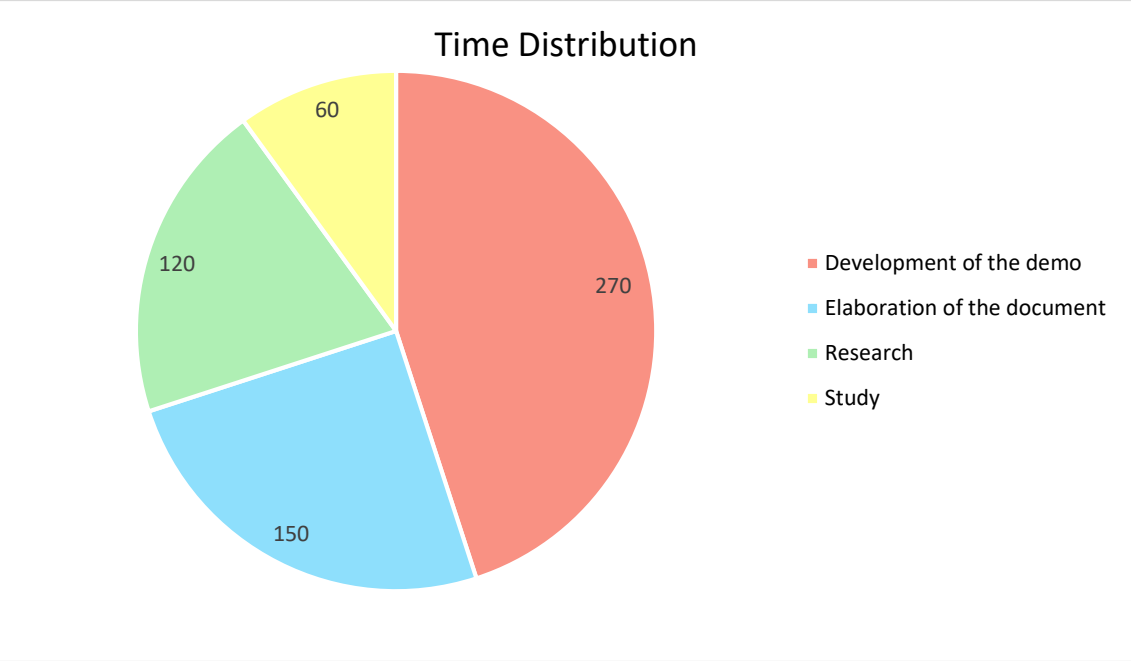

Figure 45: Distribution of hours along the project

Task	Time (hours)	Cost (per hour)
Development of the demo	270	4.929 €
Elaboration of the document	150	2.738 €
Research	120	2.191 €
Study	60	1.095 €
Total	**600 hours**	**10.953 €**

Table 4: Human Resource cost of this project in hours

Table 4 is the estimation of the total work with its correspondent cost. We can observe that the total cost of this work is 10.953€. We have also splitter the total cost into the cost of each task, observing how much all the tasks have cost.

We can observe that the total cost is 10.953€ and the cost of four months with 2.708€ per month is 10.832€, leaving us with a difference of 121€ that corresponds to 6 hours of work. Therefore, our estimation of time, although similar, is not correct as the total working time has been of 594 hours of work instead of 600.

7. Results

This chapter compiles the different test we have made to elaborate the relationship between quality-optimization and the specifications of the device in which we have produced those tests.

7.1. System Specifications

As we have large number of processes involved (geometric shaders, geometric figures, terrain creation and wind), the hardware is bound to the limit of the hardware. Therefore, we must specify which requirement our device has in order to provide a framework to understand our test. The requirements of our device are:

- CPU: Intel Core i5-6200U with 2 cores / 4 threads @2.4 GHz.
- GPU: Intel(R) HD Graphics 520.
- RAM: 6.00 GB (5.84 GB usable).
- Storage: 594 GB available.

Also, the project started with Unity's version 2019.2.5f, but later updated to Unity 2019.3.15f.

7.2. Tests

We have used the implementation of Jasper Flick to calculate the average of the framerate having a pool of framerates [33]. We have considered that this implementation would show a more realistic view of the performance of the project. We have run the tests on a build, so the inspector of Unity does not interfere with the performance.

We have noticed that, the closer the camera is to a field of grass, the more it consumes as it has to render the shadows and more. Therefore, when measuring frames per second, we have given two results: one as the lowest we have encountered and another as the average of the performace.

We will use three types of tables for testing. The first one will compare the terrain, and how much the tessellation and the LOD affects the frames per second. These tables are titled as 'Terrain' tables.

The second one will observe the geometric instances and how much do they consume when they are generated (in seconds) and when a lot of instances are affected by the wind. In this second one, the terrain will have low quality in order to focus on the geometric figures and used a Perlin Noise terrain as default. These tables are titled as 'Geometric Figures' tables.

The last one will summarize both of them to show the framerate in general. These tables are titled as 'General test' tables and show how the parameters would be applied into commercial purposes.

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	64	1	10	75 – 99
Perlin	64	1	10	75 – 99
Diamond-Square	64	1	10	75 - 99

Table 5: Terrain test at low quality in general

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	125	2.5	100	60 – 70
Perlin	125	2.5	100	60 – 70
Diamond-Square	125	2.5	100	60 - 70

Table 6: Terrain test at medium quality in general

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	125	2.5	1000	55 - 65
Perlin	125	2.5	1000	60 - 75
Diamond-Square	125	2.5	1000	60 - 75

Table 7: Terrain test at medium quality and a large amount of rocks

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	255	2.5	100	30 – 50
Perlin	255	2.5	100	30 – 50
Diamond-Square	255	2.5	100	30 - 60

Table 8: Terrain test with a large Level of Detail

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	125	5	100	40 - 60
Perlin	125	5	100	35 - 60
Diamond-Square	125	5	100	40 - 60

Table 9: Terrain test with medium / large tessellation factor

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	125	10	100	15 – 45
Perlin	125	10	100	15 – 45
Diamond-Square	125	10	100	15 – 45

Table 10: Terrain test with maximum tessellation factor

Terrain	Level of Detail	Tessellation factor	Number of rocks	Frames per Second
Heightmap	255	5	1000	20 – 30
Perlin	255	5	1000	20 – 30
Diamond-Square	255	5	1000	20 – 30

Table 11: Terrain test at high quality in general

Geometric Figures	Cypress Instances	Mushroom Instances	Fir Instances	Flower Instances	Frames per Second	Seconds loading
Test 1	5	5	5	5	80 - 90	0.0
Test 2	10	10	10	10	75 - 85	0.0 – 0.5
Test 3	50	50	50	50	80 - 70	0.7
Test 4	250	250	250	250	55 - 45	1.5
Test 5	2000	1	1	1	55 - 65	0.8 – 1.0

Test 6	1	2000	1	1	45 - 60	1.1
Test 7	1	1	2000	1	20 - 35	3.15
Test 8	1	1	1	2000	15 - 20	6.5 – 7.5
Test 9	500	500	500	500	30 - 35	3.0
Test 10	1000	1000	1000	1000	15 - 20	5.0

Table 12: Geometric Figures test

General test	Level of Detail	Tessellation factor	Number of rocks	Geometric Figures Instances	Frames per Second
Heightmap	64	2.5	40	40	75 – 90
Perlin	64	2.5	40	40	75 – 90
Diamond-Square	64	2.5	40	40	75 – 85

Table 13: General Test at low quality

General test	Level of Detail	Tessellation factor	Number of rocks	Geometric Figures Instances	Frames per Second
Heightmap	125	3	120	120	55 - 80
Perlin	125	3	120	120	55 – 80
Diamond-Square	125	3	120	120	53 – 80

Table 14: General Test at Medium quality

General test	Level of Detail	Tessellation factor	Number of rocks	Geometric Figures Instances	Frames per Second
Heightmap	255	5	200	400	15 – 35
Perlin	255	5	200	400	15 – 35
Diamond-Square	255	5	200	400	15 – 35

Table 15: General Test at High quality

We can conclude different results. In terrain, for example, we can see that the LOD and the Tessellation Factor are the agents that determines the performance. Specially the Tessellation factor, because at its maximum value it drops its value drastically.

For geometric figures, we can observe how much their presence affect the performance due to the wind. Also, more complex figures such as flowers require more time to load than simpler

figures such as the cypresses. We have considered that beyond one hundred of each instance is considered a test case, because we wouldn't use such larger number of instances.

In a general case, we can observe how they are stable (although in medium quality sometimes drops below sixty frames per second to fifty-five) except for high quality. A large number of tessellation factor with a large number of LOD makes the project to not be usable at all.

8. Conclusion and Future Work

This essay has shown the great possibilities that PCG algorithms have for creating procedural objects and terrain without previously creating them manually. We can create large complex meshes with simpler meshes and adjust them by setting parameters instead of having to create them manually. This approach is better if we want to generate large instances of objects with similar resemble and shapes without being the same object.

Also, we have shown the great possibilities that tessellation and geometric shaders have in order to change the topology of a mesh. We can generate simple stream outputs consisting on defined vertices and triangles in order to create effects such as grass.

This approach is helpful to designers and artist who, by only changing parameters via inspector, can radically change the appearance of terrains and objects. The production in a videogame can increase drastically considering these types of techniques, because their manual labour is decreased by only introduce the data they want to create and observe if the final result matches their expectation.

PCG algorithms consider end-users as agents of change as their actions can change procedurally the environment. These changes are not only limited to the creation of environment, but also to the modification of an existing world. With this approach, we let the end-user experiment and change the world with a new perspective by not letting be only an observer.

Nevertheless, we must not forget that these changes are related to optimization and computation. Although we have not covered optimization with great precision (as the content of the work was exhausting per se), PCG algorithms are always bound to the framework and the code's optimization. The range of produced geometry figures is delimitated by the technical capacity of our implementation.

Future work might show a different way to create geometric figures, as we have explained that there are different approaches in this field. We have to resolve which would be the most efficient ways to generate simple primitives and how to transform them without consuming too much resources. We have seen in the first approach how extrusion could be solved if we would have a way of calculating correctly the normals, for example.

Animation of geometric figures might be also a field to investigate, as we have created them without any visual transformation. Future work might find a solution to create a procedural animation in real-time for geometric figures.

PCG is known from a few decades, and between the videogames who have used it, there are several who are recognized for being very impactful in the sector. By giving our approach, this essay hopes to clarify and unite some of the techniques and methodologies used to generate procedural simulation.

Anexo I – Propuesta de proyecto:

TÍTULO DEL PROYECTO

Simulación de Comportamiento Orgánico mediante un Sistema Procedural de Generación, Animación y Transformación.

DESCRIPCIÓN Y JUSTIFICACIÓN DEL TEMA A TRATAR

Este proyecto está centrado en el diseño e implementación de un sistema que permita simular comportamientos orgánicos como los que muestran en la naturaleza seres vivos que crecen, se desarrollan y reaccionan al entorno que les rodea, como los organismos vegetales, mediante un modelo alejado de comportamientos predefinidos y más adecuado a experiencias interactivas como los sandboxes. El sistema desarrollado se basará en la aplicación de técnicas relativas a áreas como la generación procedural de objetos 3D y la cinemática inversa.

OBJETIVOS DEL PROYECTO

Los objetivos de este proyecto son:

1. Estudiar las técnicas de generación, animación y transformación procedural utilizadas en la actualidad.
2. Estudiar los requisitos en cuanto a entornos que puedan influir sobre los objetos controlados, para poder incorporar dichos entornos a la evaluación del sistema.
3. Diseñar e implementar el sistema de generación y control de entidades 3D que se desarrollen proceduralmente en función del entorno en el que se encuentren.
4. Analizar los resultados de la utilización del sistema a nivel de funcionalidad y rendimiento.

METODOLOGÍA

La metodología se establecerá en las primeras fases del proyecto.

PLANIFICACIÓN DE TAREAS

Las tareas quedan predefinidas de manera global en los objetivos. Serán fijadas de forma concreta durante el desarrollo del proyecto.

OBSERVACIONES ADICIONALES

Anexo II - Reuniones:

1. **Fecha**: miércoles 12 de diciembre de 2019. **Modo:** Presencial. **Lugar:** Universidad San Jorge, Zaragoza.

   Contenido: Delimitamos los primeros objetivos y el como iba a enfocar el proyecto. Enseñé las primeras versiones de la primera aproximación de Modificación de la Geometría a través de la CPU. Ambos coincidimos en que era una buena aproximación y que debía seguir por esa línea de investigación.

   Establecimos unas fechas iniciales para marcar un rumbo en el proyecto y saber si la investigación seguía un ritmo correcto.

2. **Fecha**: viernes 7 de febrero de 2020. **Modo:** Presencial. **Lugar:** Universidad San Jorge, Zaragoza.

   Contenido: Enseñé el primer prototipo de la extrusión. Coincidimos en que estaba en los límites establecidos. Expliqué alguno de los problemas que tenía, explicando el método que utilizaba para modificar la geometría.

3. **Fecha**: viernes 13 de marzo de 2020. **Modo:** Online. **Programa utilizado:** Skype

   Contenido: Comuniqué que el error que tenía en esta primera aproximación me impedía continuar con la primera aproximación y enseñé documentación que me aseguraba continuar con la investigación con otra aproximación.

   A ambos nos pareció bien y concluimos hacer otra reunión al final de este mes para observar si proseguía una aproximación adecuada con los objetivos y con las fechas establecidas.

4. **Fecha**: jueves 26 de marzo de 2020. **Modo:** Online. **Programa utilizado:** Skype.

   Contenido: Enseñé la capacidad que tenía la segunda aproximación y ambos concluimos que era factible con los objetivos propuestos. Modificamos ligeramente las fechas para establecer cuando acabaríamos definitivamente el desarrollo de la demo técnica.

5. **Fecha**: jueves 30 de abril de 2020. **Modo:** Online. **Programa utilizado:** Discord.

Contenido: A falta de añadir la interacción con el viento, corregir fallos y acabar definitivamente la escena final en la que el usuario podría moverse, el grueso del proyecto estaba acabado en los límites establecidos. Comuniqué que, mientras solucionaba todas esas tareas, escribiría también el documento del proyecto. Ambos estuvimos de acuerdo y conforme a esa decisión.

6. **Fecha**: miércoles 3 de junio de 2020. **Modo:** Online. **Programa utilizado:** Discord.

   Contenido: Entregué finalmente el documento, aunque todavía quedaban detalles por pulir. Lo único que tenía que realizar era la escena de la demo técnica y corregir cualquier error del documento. Ambos estuvimos de acuerdo y satisfechos, ya que entregué el documento en las fechas establecidas.

Bibliography

[1] TOGELIUS, J.; KASTBJERG, E.; SCHEDL, D.; YANNAKAKIS GEORGIOS, N. What is Procedural Content Generation? Mario on the borderline [online]. *PCGames '11*. June 2011, p. 1-6. Available from: https://dl.acm.org/doi/10.1145/2000919.2000922

[2] FREIKNECHT, J.; EFFELSBERG, W. A Survey on the Procedural Generation of Virtual Worlds [online]. *Multimodal Technologies Interact*. October 2017, 1(4), 27. Available from: https://www.mdpi.com/2414-4088/1/4/27/html

[3] MASSIVE SOFTWARE. *Massive*. [online]. 2017. Available from: http://www.massivesoftware.com

[4] UNITY TECHNOLOGIES. Anatomy of a Mesh [online]. *Unity User Manual (2019.3)*. April 2020. Available from: https://docs.unity3d.com/Manual/AnatomyofaMesh.html

[5] LAGUE, S. Procedural Terrain Generation [online]. *Procedural Landmass Generation (E12: normals)*. [Sound Recorded]. June 2016. Available from: https://www.youtube.com/watch?v=NpeYTcS7n-M

[6] AKENINE-MÖLLER, T.; HAINES, E.; HOFFMAIN, N.; PESCE, A.; IWANICKI, M.; HILLAIRE. S. *Real-Time Rendering*. Fourth Edition. Boca Raton: Taylor & Francis, CRC Press, 2018. ISBN: 9781138627000.

[7] UNITY TECHNOLOGIES. Mathf.Perlin Noise [online]. *Unity User Manual (2019.3).* April 2020. Available from: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

[8] BEVINS, J. Libnoise [online]. *Perlin noise*. 2003 – 2005. Available from: http://libnoise.sourceforge.net/glossary/#perlinnoise

[9] SHIFFMAN, D. Chapter 8: Fractals. In: SHANNON, F (ed). *The Nature of Code* [online]. 2012. ISBN-13: 978-0985930806. ISBN-10: 0985930802. Available from: https://natureofcode.com/book/chapter-8-fractals/

[10] FRACTAL FOUNDATION. What are fractals? [online]. 2009. Available from: https://fractalfoundation.org/resources/what-are-fractals/

[11] COMPTON, C. Link's Awakening and Fractal Game Design [online]. *Gamasutra*. February, 2020. Available from: https://www.gamasutra.com/blogs/CalebCompton/20200204/357530/Links_Awakening_and_Fractal_Game_Design.php

[12] MICROSOFT. Random Class [online]. *Microsoft Docs*. 2020. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.random?view=netcore-3.1

[13] STEIMER, K. Dragon Age II Review [online]. *IGN*. May 2012. Available from: https://www.ign.com/articles/2011/03/08/dragon-age-ii-review

[14] REID, L. Simple mesh tessellation & triangulation tutorial [online]. *Linden Reid*. December, 2017. Available from: https://lindenreid.wordpress.com/2017/12/03/simple-mesh-tessellation-triangulation-tutorial/

[15] SURIDGE, J. Modelling by numbers: Part one [online]. *Gamasutra*. September, 2013. Available from: https://www.gamasutra.com/blogs/JayelindaSuridge/20130903/199457/Modelling_by_numbers_Part_One_A.php

[16] FLICK, J. Tessellation [online]. *Catlike coding*. November, 2017. Available from: https://catlikecoding.com/unity/tutorials/advanced-rendering/tessellation/

[17] ROYSTAN, E. Grass shader [online]. *Roystan*. March, 2019. Available from: https://roystan.net/articles/grass-shader.html

[18] SURIDGE, J. Modelling by numbers: Supplementary (Terrain) [online]. *Gamasutra*. November, 2014. Available from: https://www.gamasutra.com/blogs/JayelindaSuridge/20141113/230153/Modelling_by_numbers_Supplementary_Terrain.php

[19] LAGUE, S. Procedural Terrain Generation [online]. *Procedural Landmass Generation (E04: colours)*. [Sound Recorded]. February 2016. Available from: https://www.youtube.com/watch?v=RDQK1_SWFuc

[20] BLENDER. Extrude [online]. *Blender 2.82 Manual.* February 2020. Available from: https://docs.blender.org/manual/en/latest/modeling/meshes/editing/duplicating/extrude.html

[21] DUFFY, S. Runtime Mesh Manipulation with Unity [online]. *Raywenderlich.com.* October, 2019. Available from: https://www.raywenderlich.com/3169311-runtime-mesh-manipulation-with-unity

[22] REID, L. Procedural Stellation Tutorial [online]. *Linden Reid*. November, 2017. Available from: https://lindenreid.wordpress.com/2017/11/04/procedural-stellation-tutorial/

[23] BONNEAU, D.; DIFRANCESCO, P.; HUTCHINSON, D. J. Surface Reconstruction for Three-Dimensional Rockfall Volumetric Analysis [online]. *Multimodal Technologies Interact.* Figure 1. November 2019, 8 (12), 548. Available from: https://www.mdpi.com/2414-4088/1/4/27/html

[24] UNITY TECHNOLOGIES. Time.deltaTime [online]. *Unity User Manual (2019.3).* April 2020. Available from: https://docs.unity3d.com/ScriptReference/Time-deltaTime.html

[25] PENADÉS, S. es6-tween [online]. *Github*. May 2020. Available from: https://github.com/tweenjs/es6-tween

[26] FAKHROUTDINOV, K. The Unified Modeling Language [online]. *UML Class and Object Diagrams Overview*. 2007 – 2016. Available from: https://www.uml-diagrams.org/class-diagrams-overview.html#implementation-classes

[27] POMAX. A Primer on Bézier Curves [online]. *Github*. January 2020. Available from: https://pomax.github.io/bezierinfo/en-GB/

[28] PERALTA, J.; ROYUELA, M. Familia compositae (Asteraceae) [online]. *Herbario – Departamento de ciencias Universidad Pública de Navarra*. September 2019. Pamplona, Navarra. Available from: https://www.unavarra.es/herbario/htm/Compositae.htm

[29] BÖHRINGER, R. White noise [online]. *Ronja´s shader tutorials*. September 2018. Available from: https://www.ronja-tutorials.com/2018/09/02/white-noise.html

[30] TAKAHASHI, K. 3x3 Rotation matrix with an angle and an arbitrary vector [online]. *Githbub*. December 2017. Available from: https://gist.github.com/keijiro/ee439d5e7388f3aafc5296005c8c3f33

[31] UNITY TECHNOLOGIES. Vertex and fragment shader examples [online]. *Unity User Manual (2019.3).* April 2020. Available from: https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html

[32] GLASSDOOR. Sueldos para Junior Technical Artist [online]. *Glassdoor*. April 2020. Available from: https://www.glassdoor.es/Sueldos/junior-technical-artist-sueldo-SRCH_KO0,23.htm

[33] FLICK, J. Frames per Second [online]. *Catlike coding*. November, 2017. Available from: https://catlikecoding.com/unity/tutorials/frames-per-second/