3D mesh voxelization

by Adrián Ferrando Mazarro

Final's Work Memory

Degree in Video Game Design and Development

Universitat Jaume I

June 2019, Valencia

Acknowledgments

Thank my parents for investing their life in me. And to my friends, for enduring my total disappearance all these months. And, of course, to all those who devote part of their time to solve problems in the code of others through Internet forums, and all the teachers who have managed to make me a wiser person. Without all of you, this would not exist.

Abstract

This paper tries to make an approach to the use of voxels as a basic element of a videogame agent, both in terms of visualization and interaction through the use of a Unity game engine plug-in.

Table of contents

1. Introduction....06
1.1. Motivation of work....06
1.2. Objectives....07
1.3. Context and initial state....08
2. The Plug-in....09
2.1. Unity implementation....09
2.2. Object lesson....10
3. Planning and evaluation of resources....14
3.1. Planning....14
3.2. Evaluation of resources....15
4. System design and analysis....17
4.1. Requirements analysis....17
Functional requirements
Non-functional requirements
4.2. System design....18
4.3. System architecture....26
4.4. Interface Design....27
5. Development of work and results....29
5.1. Work Development.... 29
5.2. Testing.... 37
5.3. Results....40
6. Conclusions and future work....41
6.1. Conclusions....41
6.2. Future work....42
7. Bibliography....43

1. Introduction

This chapter will provide a compilation of the initial conditions of the realization of this project. The motivations that propitiated the present text will be exposed, the objectives set out in the beginning and the context of the execution, which I hope will contribute to better understand the evolution of the project.

1.1. Motivation of work

The main subject in the field of Computer Graphics studies how three-dimensional scenes are visualized in a 2D image. In short, and to simplify a lot, a mesh structure based on triangles is established and, through certain projections, it becomes an image from the point of view of a camera.

To represent different effects, certain calculations based on this mesh structure are applied, such as the reflection of the light photons of the environment and its visual result depending on the properties of the object. This is enough to recreate illuminated surfaces, both for its ability to reach a perfect visual realism thanks to the use of complex calculations and its relatively low computational cost.

So important is this type of representation that we have created dedicated hardware to deal with the massive calculations that are needed to be carried out: the graphics cards. However, they are not so good for representing volumes; after all, the triangle is a two-dimensional unit.

The destruction of objects, the thermodynamics of a volume, calculation of fluid mechanics or truly realistic calculations of physics are extremely complicated to do in meshes and, often, they are not perfect.

That is why the possibility of representing these meshes volumetrically could be a much closer approach to reality; create a kind of atoms, a minimum unit that allows computing it more or less efficiently. For this, it's possible to use cubes to divide the volume, making a simile with a raster image made of pixels. This, as I would discover later in my research, is called *voxel*.

The project will also help me to check, develop and consolidate my knowledge in fields as varied as data structures, graphic computing and hardware acceleration -in this case, through GPU-, the use of efficient algorithms in an applied way and so on.

1.2. Objectives

The objectives of this Final Degree Work report have been adapted as my knowledge about it was deeper. At first I thought directly about making my own engine, which would give me a lot of control over all the algorithms that were responsible for generating and rendering the structure, but it would also give me an too much work, as I found out as soon as I investigated in more detail the practice of implementing an engine from scratch.

After realizing this, I thought then about creating a plug-in for a game engine in which I already had enough experience as Unity is. In addition, its wide dissemination on the Internet would allow me to solve more easily the problems that might be encountered with the engine itself. This plugin would allow using meshes already made in standard modeling programs such as 3DS Max or Blender to generate voxel structures at different resolutions. It would also allow us to control this structure to some extent, to generate the in-game effects that the user wants.

Therefore, based on the above, we can establish the following objectives for the *plug-in* Unity:

-Creating a data structure that allows us to work with voxels efficiently in real time.

-*Baking*[1] meshes in this structure in a simple and automatic way so that the user of the plug-in does not have to do too much work to generate these structures -the existing tools to generate voxel structures are limited, complicated to use and require a very different workflow from the usual one to generate 3D meshes.

-To visualize this efficiently, using 3D meshes as an intermediate step between the structure of voxels and the image taken by the screen, so that we can take advantage of the already developed technology of 3D visualization to simplify our work and increase performance.

-To be able to manage in any of the ways previously proposed -transmission of temperatures, destruction/generation in real time- the structure of voxels in real time.

[1] Term in Computer Graphics defined as the act of pre-computing something in order to speed up some other process later down the line.

1.3. Context and initial state

Shortly before starting work, I moved from my student apartment in Castellón to Valencia, to the house of my parents. This was motivated by two main factors: the first, the realization of curricular internship in a small team whose mainly commission was to make a demo of a virtual museum for a medieval castle in the Aragon municipality of Mora de Rubielos. This took up most of my time during the first months: the whole morning was spent going to the office for working in the project. The hours that could devote daily to the development of the project that occupies us in the present text would be reduced, then, in this initial period.

In addition, I began to obtain positions of greater responsibility in the political organization in which I am a member and I started a separate project as study director with six students at Universitat Politècnica de València coursing the degrees in Computer Engineering and Industrial Design and Product Development Engineering. This left me with even less time available, so I had to have a very constant work discipline and to spend a lot of my free time on the project. However, I am a person who reaches his true potential after a good amount of time working, which did not allow me to work to the maximum of my abilities in this initial period.

In short, I started the project with a much tighter agenda than I would like, which, as we will see later, will give me some problems.

2. The Plug-in

This chapter will explain how the plug-in works, which tasks it has to do and we will also see an example of use. A deepest analysis of the coding and the system structure will be explained later in this report.

2.1. Unity implementation

There are several ways to make a plug-in for Unity. This is thank to his *Editor* classes, which are deeply customizable and allows to modify the interface as much as you want. In this case, with the purpose of making the tool very easy to use for every beginner in the environment of the Unity game engine, we choose to use the in-scene hierarchy of the engine itself. For this reason, we use a Unity concept called *Prefab.*

A Prefab (see in Figure 1) is a GameObject -an agent object in the scene- that can be saved in secondary memory for later be instantiated in the scene. For example, we can build an Enemy GameObject in the scene and set some parameters to his components -BoxCollider, Custom Scripts, Rigidbody and many more-. Then, we save this in a project asset folder. All the instantiated GameObjects using this prefabs will have the same configuration.

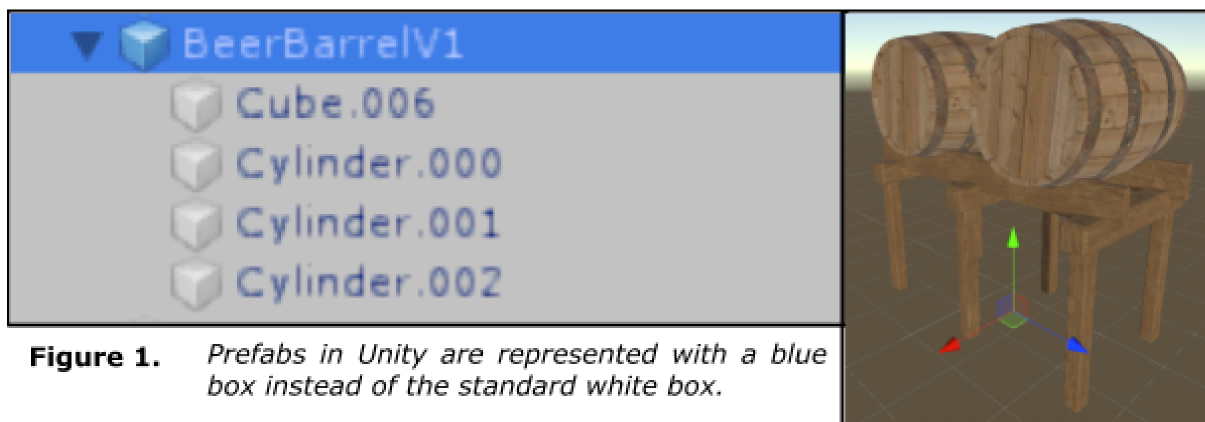


Figure 1. *Prefabs in Unity are represented with a blue box instead of the standard white box.*

And how this will work with our plug-in? We will use this for creating a *GameObject in-scene tool*, which could make all the spatial configurations we need, taking advantage of the built-in scene Gizmos of Unity. In the following section, we will see how to use the plug-in in a practical example and, in addition, it will explain us how to this tool works better.

2.2. Object lesson

In this lesson, we will see how the plug-in works with an example mesh. Here, Stanford Bunny will be used as example, modified for having no holes (see in Figure 2).

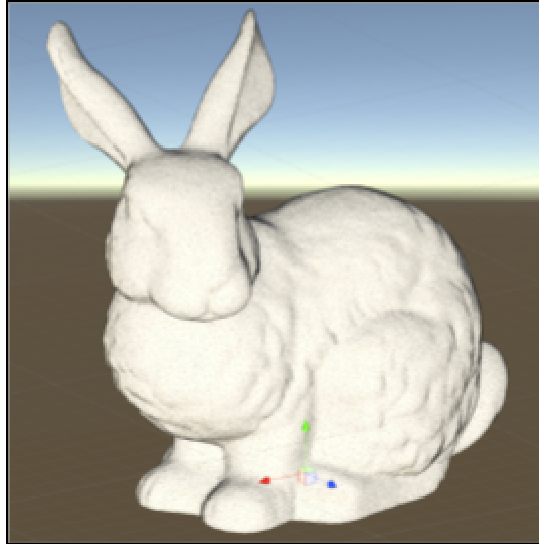


Figure 2. *High-poly Stanford Bunny.*

However, for the purpose of this plug-in, the 144046 faces shaping the mesh are too much and that will mean a huge slowdown in the performance. We should, then, decrease the number of polygons: we are not aiming for full detailed meshes, as we only need the shape in order to conform a clearly profile in the voxel system. The model was optimized to has 1% of the original faces, only 1436 faces (see in Figure 3).

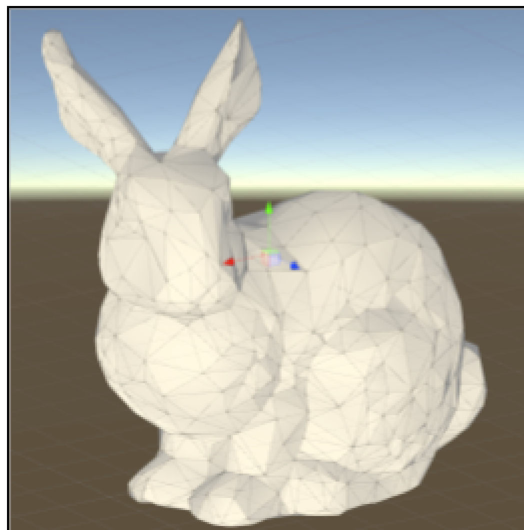


Figure 3. *Low-poly Stanford bunny.*

With that model imported into Unity, we go and select the *VoxelSystem GameObject* in our scene hierarchy (see in Figure 4). This will deploy in the Inspector the components of this agent. Here we can see the *VoxelSystem script component* (see in Figure 5), which let us to configure some variables for voxelizing a mesh.

Figure 4. *Our prefab in scene hierarchy.*

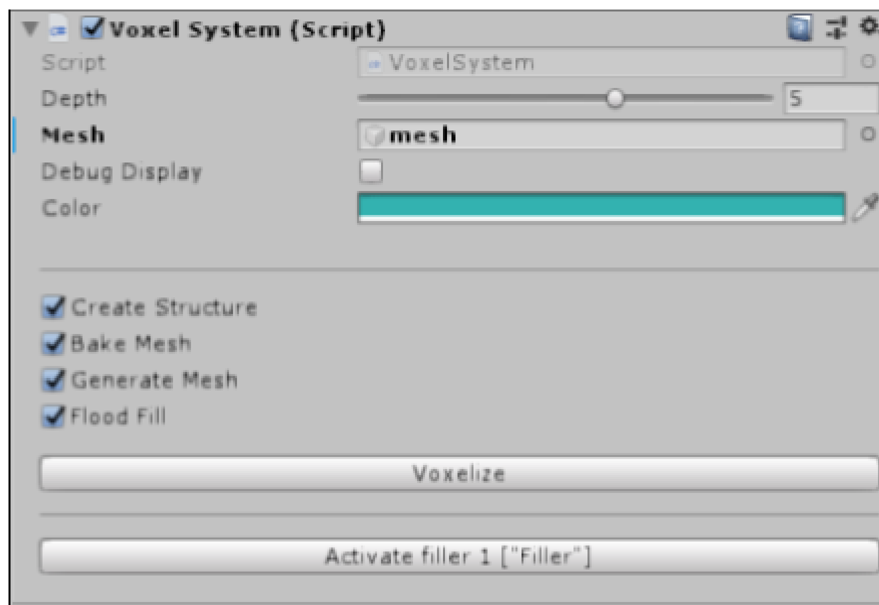


Figure 5. *VoxelSystem component.*

For correctly voxelize the bunny, we should adjust the voxelization volume box to fit the model as we see in Figure 6.

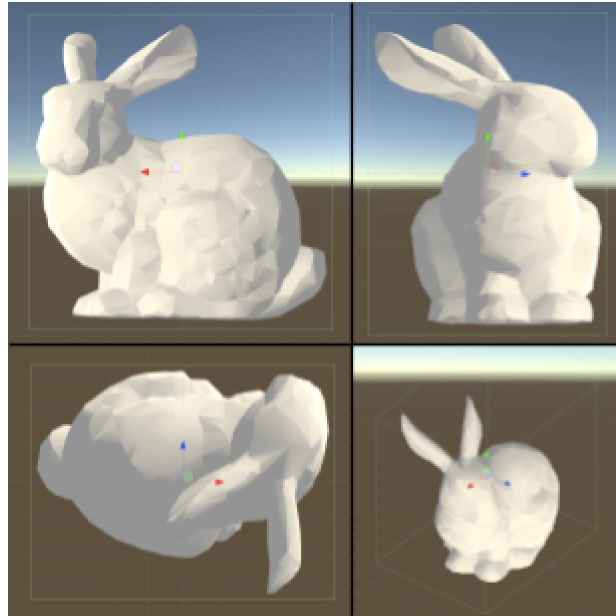


Figure 6. *Here we can see how the AABB of the VoxelSystem GameObject fits the model.*

Then, we proceed with voxelizing the model with a depth of 8 -depth, as we will explain later, is the *resolution* of our voxel system-. The result can take a while, so we wait until is finished and the result should be in Figure 7.

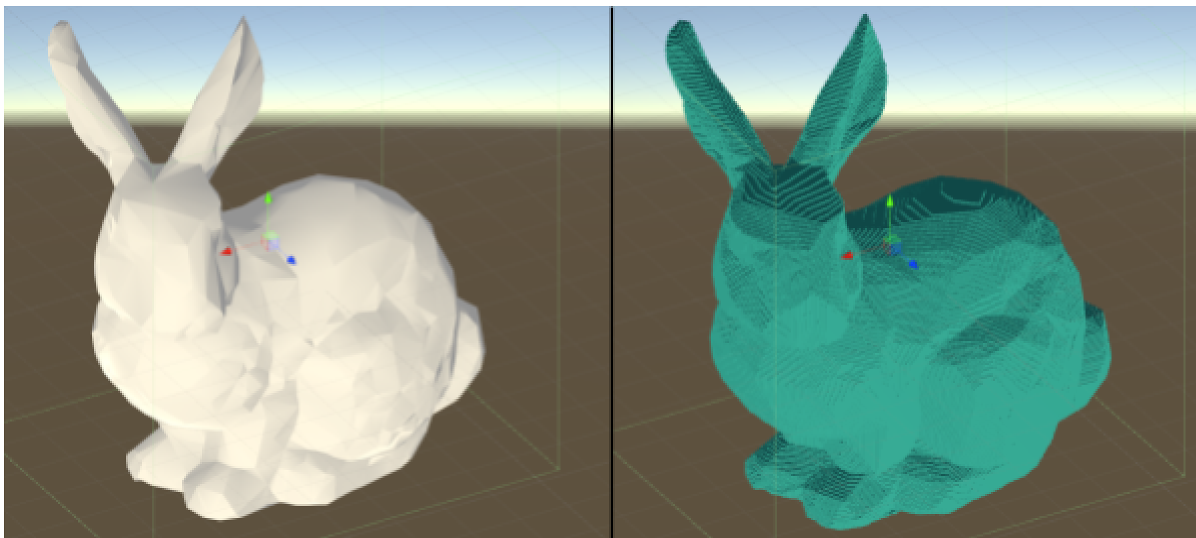


Figure 7. *The high resolution of the system shows the low-poly hard edges of the mesh, resulting in this flat-looking voxel array.*

As we see, the resulting voxel system keeps the flat faces of the low poly model. This can be avoided by reducing the resolution of our model. Let's see the result of that in the Figure 8.

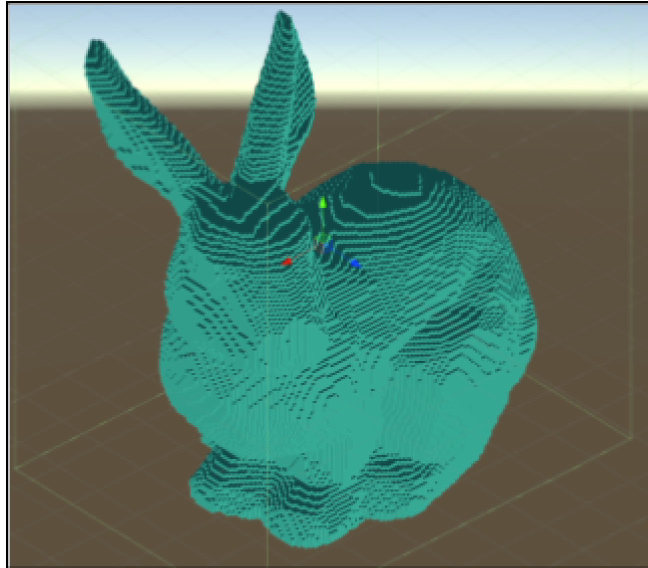


Figure 8. *VoxelSystem with 7 levels of depth.*

Finally, we can see the result using the *Debug* toggle in the inspector of the Figure 9. This will show us the surface voxels as purple wire cubes and the inner ones as blue cubes. This is useful for checking if all is setted correctly.

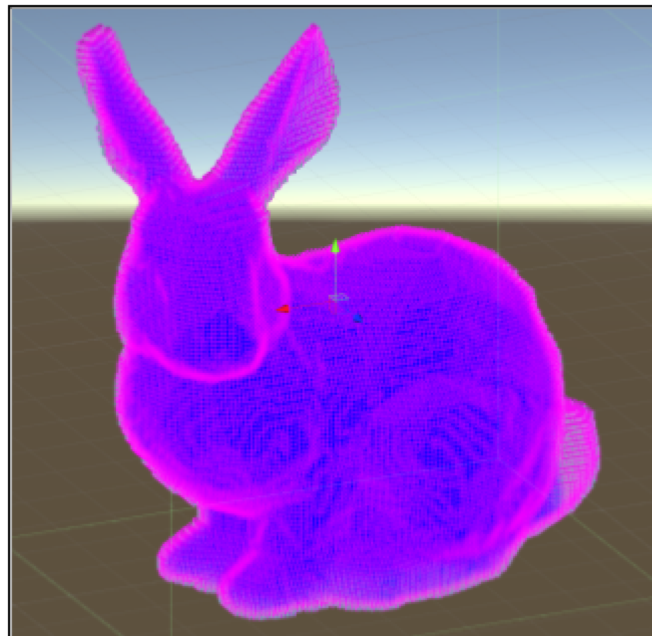


Figure 9. *Debug view of the VoxelSystem.*

3. Planning and evaluation of resources

In this section we will explain how the planning of the project times has been carried out. There will also be a brief reflection on the costs of the project in the event that it was a paid engineering project.

3.1. Planning

The initial conditions that I mentioned in the previous section in relation to the external workload that I suffered required a planning that was not only clear and concise, but highly flexible and with wide margins, so that I could coordinate it with the rest of matters of my daily life, sporadic and improvised by nature.

Let's set a target time of 270 hours. I assume that several tasks are likely to last longer than I expect, because experience in other projects leads me to extrapolate that I tend to assume that things will happen faster than they actually take to take place.

The calculation of times I made can be observed in the following table (see in Figure 10). In addition, you can also observe the time that was finally invested in each task.

Tarea	Subtarea	Estimación de tiempo	Tiempo real	Total (estimado-real)
Creación de la estructura	Investigación	10h	20h (+10h)	85h - 120h (+35h)
	Planificación	15h	15h	
	Implementación	50h	70h (+20h)	
	Ajuste	10h	15h (+5h)	
Conversión malla-vóxel	Investigación	10h	15h (+5h)	75h - 115h (+40h)
	Planificación	15h	15h	
	Implementación	50h	65h (+25h)	
	Ajuste	10h	20h (+10h)	
Visualización	Investigación	10h	7h (-3h)	45h - 37h (-8h)
	Implementación	25h	25h	
	Ajuste	10h	5h (-5h)	
Interacción con el sistema	Implementación	20h	17h (-3h)	30h - 23h (-7h)
	Ajuste	10h	6h (-4h)	
				235h - 295h (+60h)

Figura 10. *In this table we can see the time distribution of the project*

The method was according to the specific context of the project: due to my lack of initial knowledge on the subject, each section required an investment of time in *research* (except for the last one, because it is simply a construction on the systems that I have already created). In addition, I also include in the two most complex tasks a task for *planning*, which would help me establish the route

to follow for trying to avoid improvisation in the *implementation* as much as possible. After implementing the code, there is a small subtask -where some parts are optimized or some sections of the program are refined- that has been denominated here as *adjustment*.

In the end, the difference between the real and the estimated was 60 hours, which validates my hypothesis which establishes down that I tend to underestimate the temporal cost of the tasks, making the decision to estimate the project times with a generous upper room in a correct choice.

Below you can see a Gantt diagram that I have made with the purpose of visualizing the distribution of all the tasks over time (see in Figure 11).

As you can see, there are often overlapping tasks. This is due to two main reasons: some weeks I ended with one task and started with another, and the only way to represent it is by putting both active tasks in that week, and that, on some occasions, especially with the research stage, this overlap was due to the parallel performance of both tasks.

The subordination of some tasks to others follows a simple order: all tasks depend on the previous one. In the case of an individual execution of the project, this is true. However, in an execution with a team of people, the third stage -the *visualization*- could have been carried out in parallel with the second stage, and even with a certain more advanced part of the first stage.

3.2. Evaluation of resources

In summary, if it had been a paid project, the cost of the project would amount to 2950€ with a payment of 10€ per hour. Taking into account that the initial estimate was 2350€ (235h * 10€/h), the difference was 600€. That is to say, in a non-academic situation there would have been an extra cost of 25.5% with respect to the initial price. This type of things are common in software development, but should not occur: the customer could refuse to pay the extra cost and end up forcing the developer to take it on their own.

In addition, in the case of a company, this extra cost should be added to the payments of facilities and personnel, which can become a problem if the possibility of it happening is not taken into account.

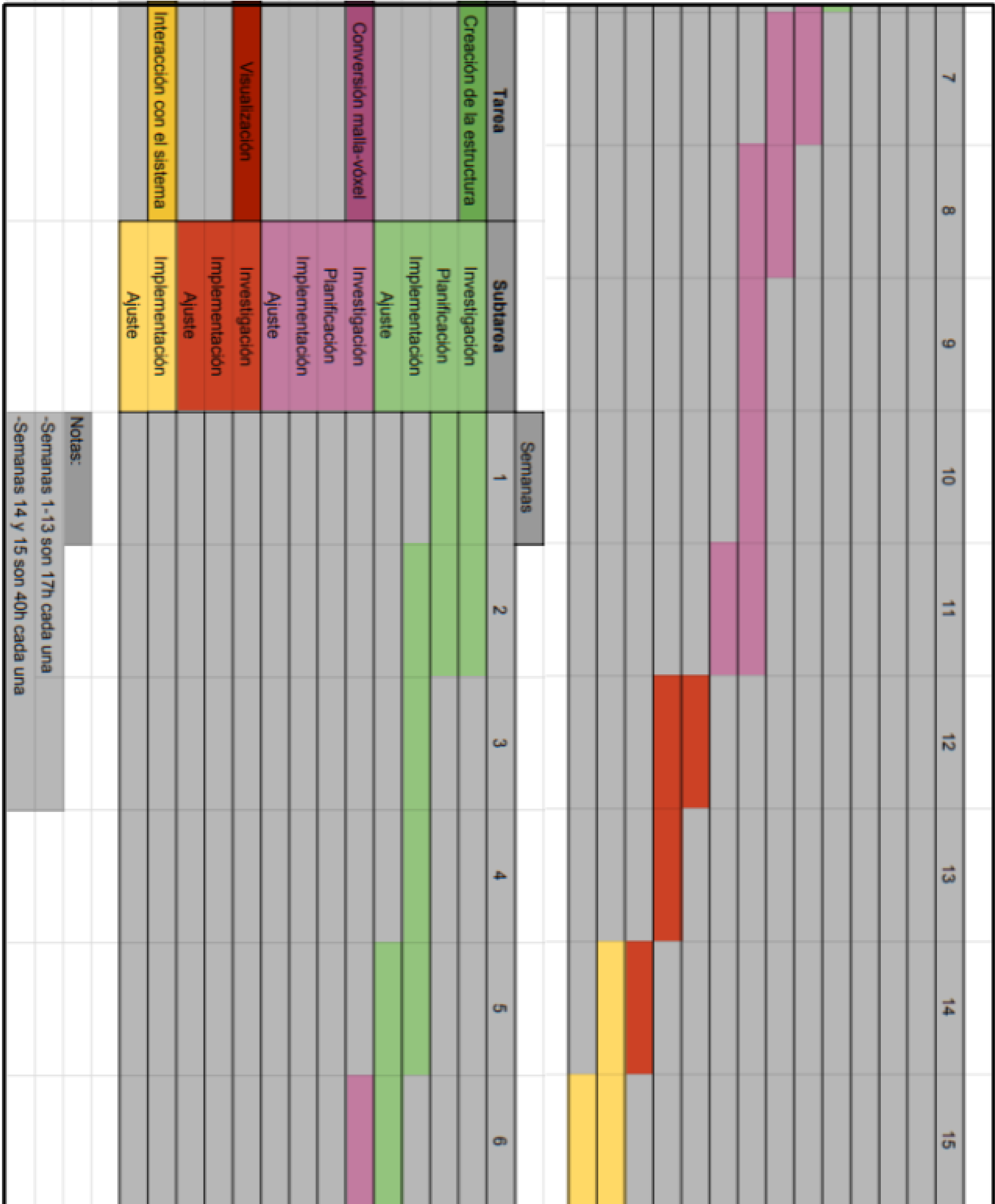


Figure 11. *Gantt diagram of the project.*

4. System design and analysis

This chapter will review all the topics related to the analysis, design and architecture of the system, as well as the design of the *plug-in* interface.

4.1. Requirements analysis

We will divide the requirements into two classes: the *functional ones*, which refer to all those requirements with input and output of data and an established behavior, and the *non-functional ones*, which cover all the conditions characteristic of the design and implementation of the project.

Functional requirements

-Given a maximum resolution, a position and a scale in three dimensions, generate an octree with these characteristics capable of saving information.

-Given an octree and a mesh, transmit the information of the surface of the second to the first, filling this surface to obtain in the octree a solid structure equivalent to the surface of the mesh.

-Given a system of established voxels, to be able to visualize the system in a graphical way.

-Given a voxel system, to be able to **interact** with him in game time.

Non-functional requirements

The project requires the following characteristics:

-Efficiency in terms of the interaction and visualization of the system, so it can be used in real time at a minimum frame rate of 24 fps.

-Understanding and readability in the structure of the code over optimization, with the aim of being able to include in a relatively simple way new functional characteristics to the project, as well as to facilitate its understanding by third parties in pursuit of the scalability of the development team , either under contract of employment or for free on the Internet.

-Simple simplicity of use by the user, so that it does not require great knowledge about the system or the programming itself.

-Taking into account the previous section, it will be the *plug-in* which will explain itself.

4.2. System design

The system presented in this paper has a class structure defined in the diagram on the next page (see in Diagram 1). In it you can appreciate the relationships between different systems. We will proceed to review more carefully what job each one performs to meet the previously defined objectives.

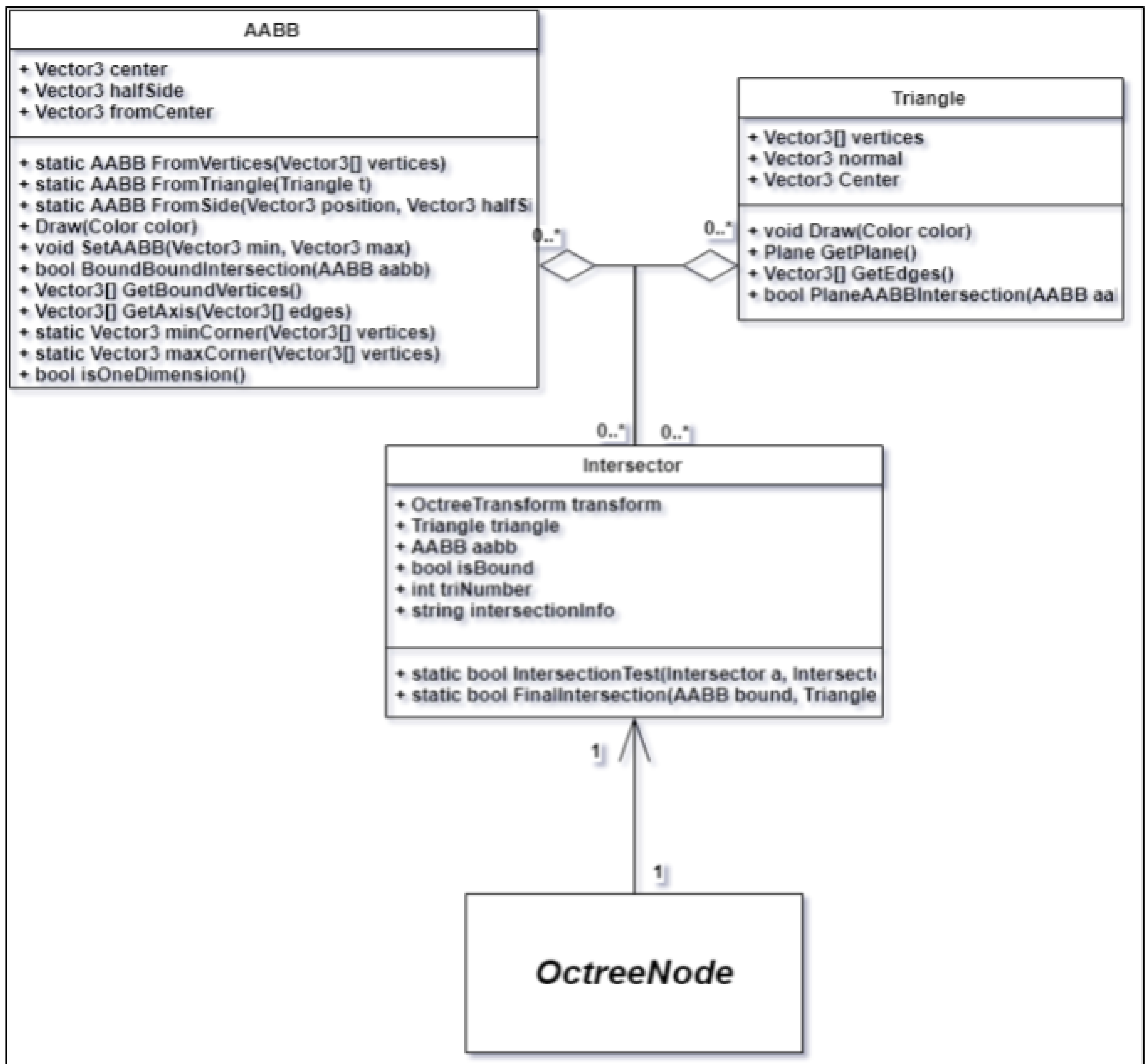


Diagram 1(i). *Intersector, AABB and Triangle handle with the geometric calculations of the system.*

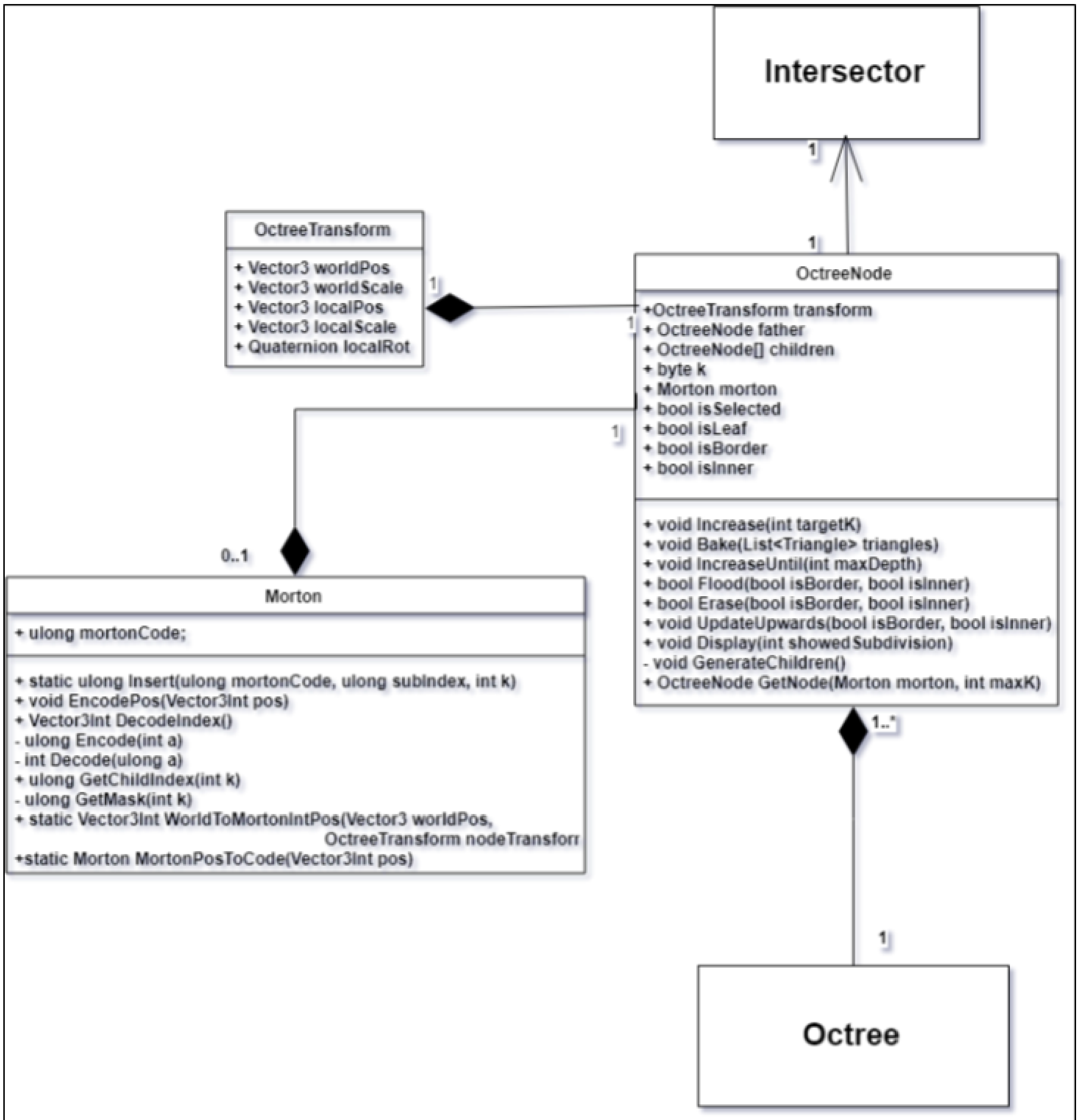


Diagram 1(ii). *The node structure is the core of the system.*

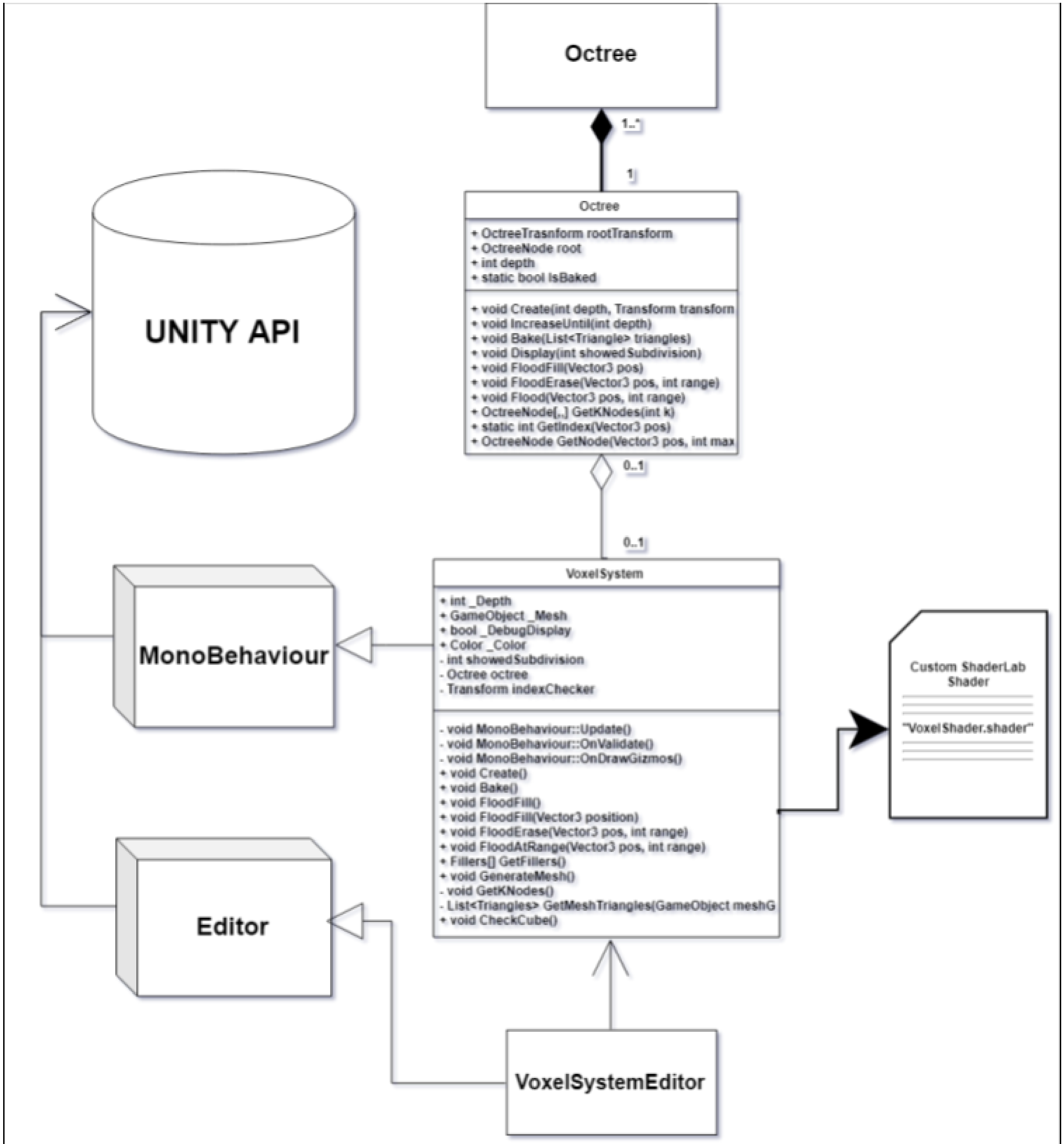


Diagram 1(iii). *The VoxelSystem links the octree with the whole engine.*

The most important part is the class *VoxelSystem*. This class composes the central axis of the *plug-in*, since it derives from the basic class of Unity scripting *Monobehavior*. This allows it to be the bridge between the internal system of the program and the Unity API; to derive from *Monobehavior* allows the script to belong to an agent entity in the game engine and, therefore, act within it.

This system can be controlled by the user to the extent that the class *VoxelEditor* allows, which is derived from the class *Editor* Unity. This allows you to create buttons, fields, slide bars and more input tools into the Unity Inspector (that is, the GUI of the engine that allows the user to control the agents of the scene).

Encapsulated in the class *VoxelSystem*, the class *Octree* is a tree-type data structure where each node, with the exception of the *leaves* -the deepest nodes-, has exactly 8 children, and where every node has one or no node as a parent. A deepest explanation of the octree concept is in the *Work Development* chapter. It is responsible for managing all the nodes of the Octree and contains only a relationship with the root node -the only node without a parent and the one with the least depth-, so that all the data is accessed recursively.

The class that defines the nodes, then, is *OctreeNode*. In this class there are several attributes that allow us to save information. The two most important are the transform, a struct called *OctreeTransform*, and the morton code, defined in the class *Morton*. The use of the latter will be explained later, but, to summarize, it is the coded position of the node with respect to the root.

In addition, each node generates during the *Bake* operation an object of the *Intersector* class, which stores information about its AABB (*Axis Alligned Bounding Box*), and serves as a controller for the intersection calculations to which the structure will have to submit in order to convert the mesh into the voxel system. It is not encapsulated in the node in order to decrease the total RAM used by the system.

Intersector, in turn, has as attributes the *AABB* of the object which it has to manage and, if needed, the *Triangle* it includes -this is false in the case of being a node *Intersector*. The AABB can be either from an *OctreeNode* or from a mesh triangle. The *Triangle*, as its name suggests, is the class that helps with certain geometric operations for the intersection algorithm used in the system.

Next we will see how these systems communicate with each other in the three main operations that are carried out in the system. For this, some sequence diagrams have been made with the aim of helping us to better understand how the systems relate to each other.

In the case of the first large operation, *Create()*, which generates the structure at a certain depth level, we will observe the diagram (see in Diagram 2). First, the *VoxelSystem* sends the command to the *Octree*, who in turn transmits the command to the root node. This node creates its children, in turn they create other 8 children each in a recursive loop until reaching the maximum depth of the structure defined by the user.

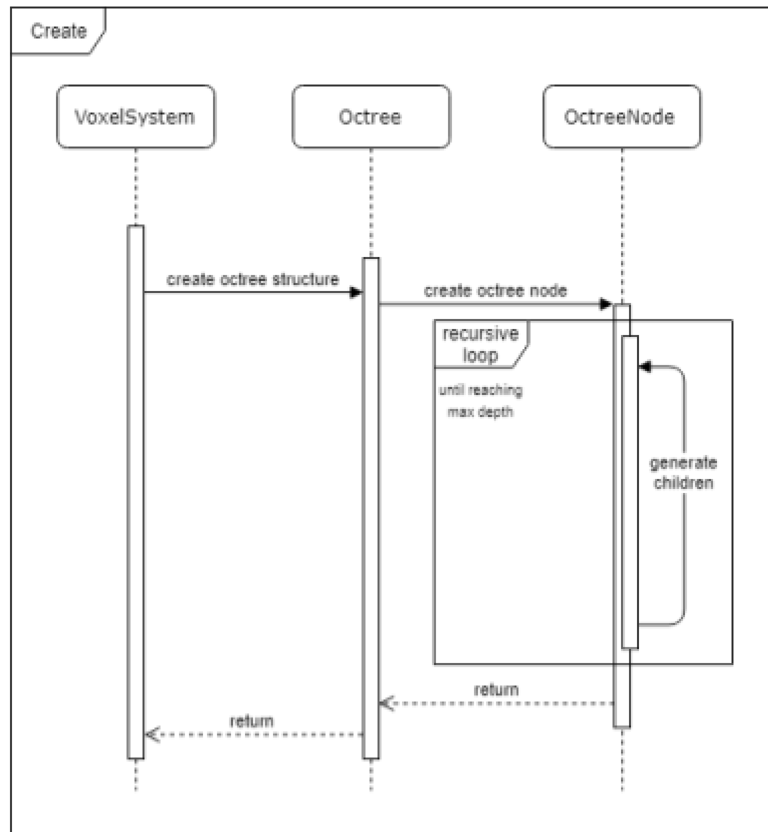


Diagram 2. *Octree is created with recursive function calls*

The following diagram (see in Diagram 3) is the one that represents the *baking* of the mesh in the voxel structure: *Bake()*. In this operation, a similar pattern to *Create()* is followed: the message being transmitted by the *VoxelSystem-Octree-OctreeNode* hierarchy to the root of the octree. Once at the nodes level, it is checked whether or not there is an intersection with any triangle of the mesh. If there is no intersection, that node does not propagate to its children, since by definition there will be no intersection either. In the case of having it, it spreads. In this way, we avoid calculating nodes that we already know will not intersect any triangle. To check each intersection, the node orders the test to be performed by its intersector, who executes the tests in the respective geometric classes.

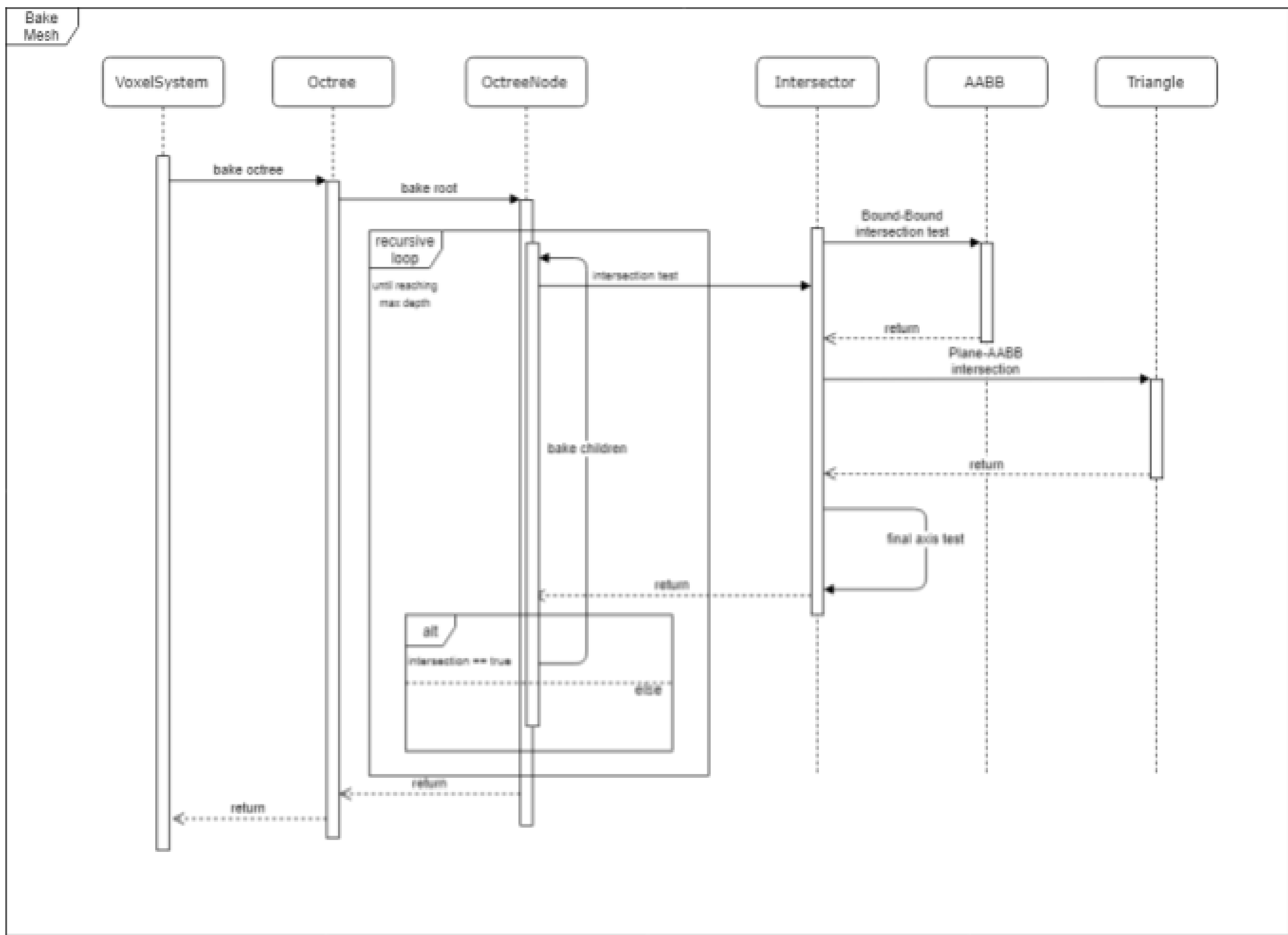


Diagram 3. *The baking process is the most complex one.*

The last sequence diagram (see in Diagram 4) corresponds to the operation generated by the structure display grid, *GenerateMesh()*. In this operation there is a noticeable change with respect to the others: instead of using a self-calling recursive function in the *OctreeNode* class, a while loop is used in the *Octree* class, iterating until all the nodes in some given k-level are obtained. This does not mean that the order to obtain a specific node is not propagated by the data structure. It has been done in this way for not complicating the diagram unnecessarily and thus demonstrate the importance of the operation: a loop that will collect in a data buffer all the positions of the nodes at a certain level. The peculiarities of the *GetNode()* command will be explained later. After this, the *VoxelSystem* sends the position buffer to a shader that will be responsible for drawing the geometry by using a geometric function and computing all the visual aspects of the system using a fragment function.

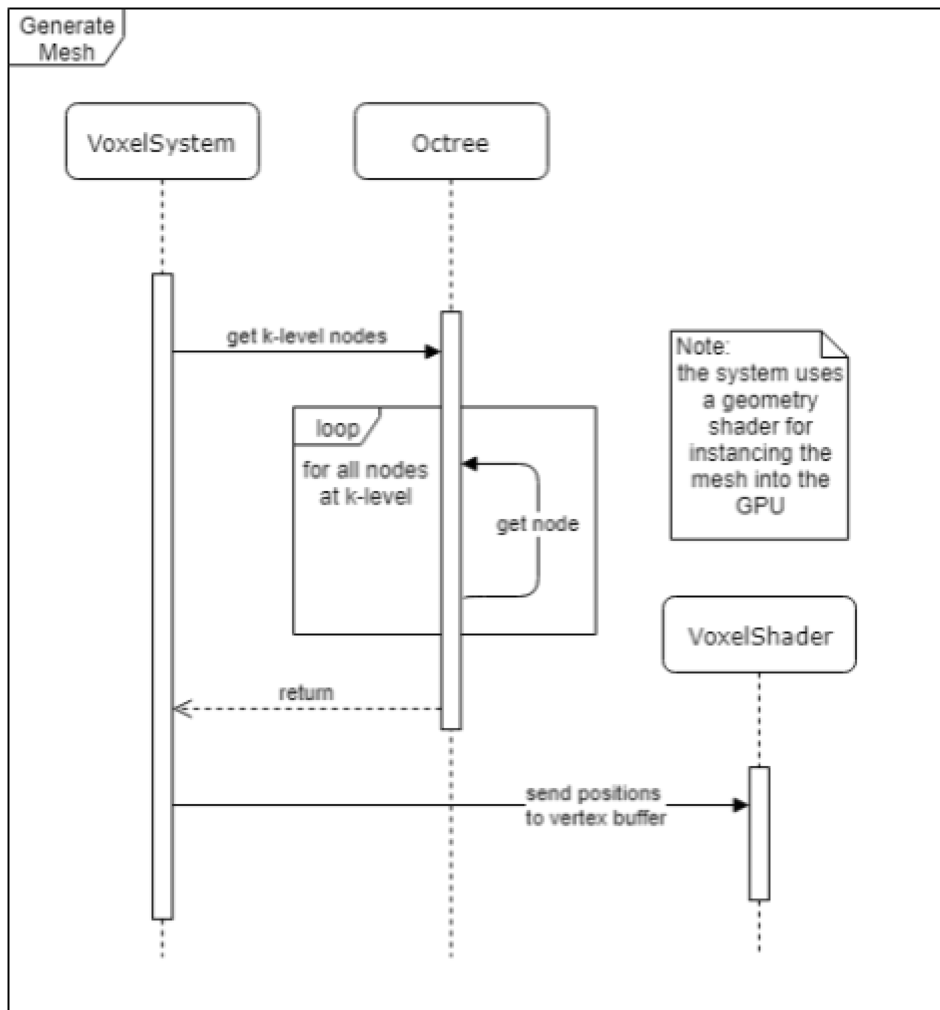


Diagram 4. *Here a loop traverses all nodes in Z-Order.*

Finally, I have made the following activity diagram (see in Diagram 5) that represents the Möller algorithm[2] to check the intersection between an AABB and a triangle. It was chosen because one main factors: it is above 2 times faster -this increased performance depends on the CPU you use- than the main previous intersection algorithm found in Gems V[4], which is also faster than the previous one described in Gems III[5]. The algorithm made by Thomas Akenine-Möller consists, in broad strokes, in making a series of very quick preliminary checks to execute that result in a false positive: if it turns out that there is no intersection, this statement is completely true; if it turns out that there is an intersection, the affirmation does not have to be true. In this way, the only test that ensures the intersection is the last test, which is based on the *separation axes theorem* -or *hyperplane separation theorem*- which is more expensive to compute. Due to the huge number of voxels that statistically do not intersect with a certain triangle, this small trick allows for much shorter computation times.

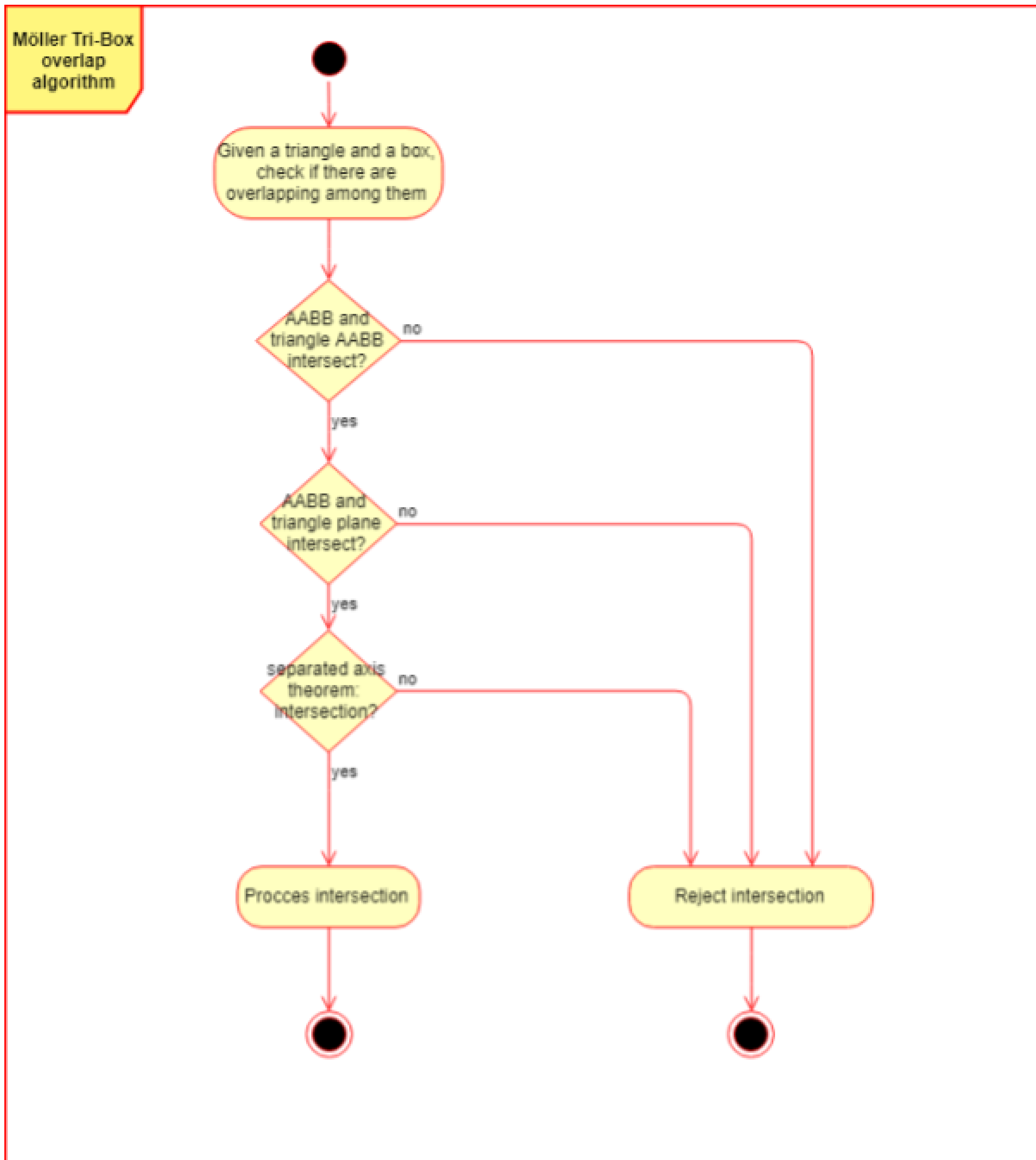


Diagram 5. *We can see the three tests are made in the Möller algorithm.*

4.3. System architecture

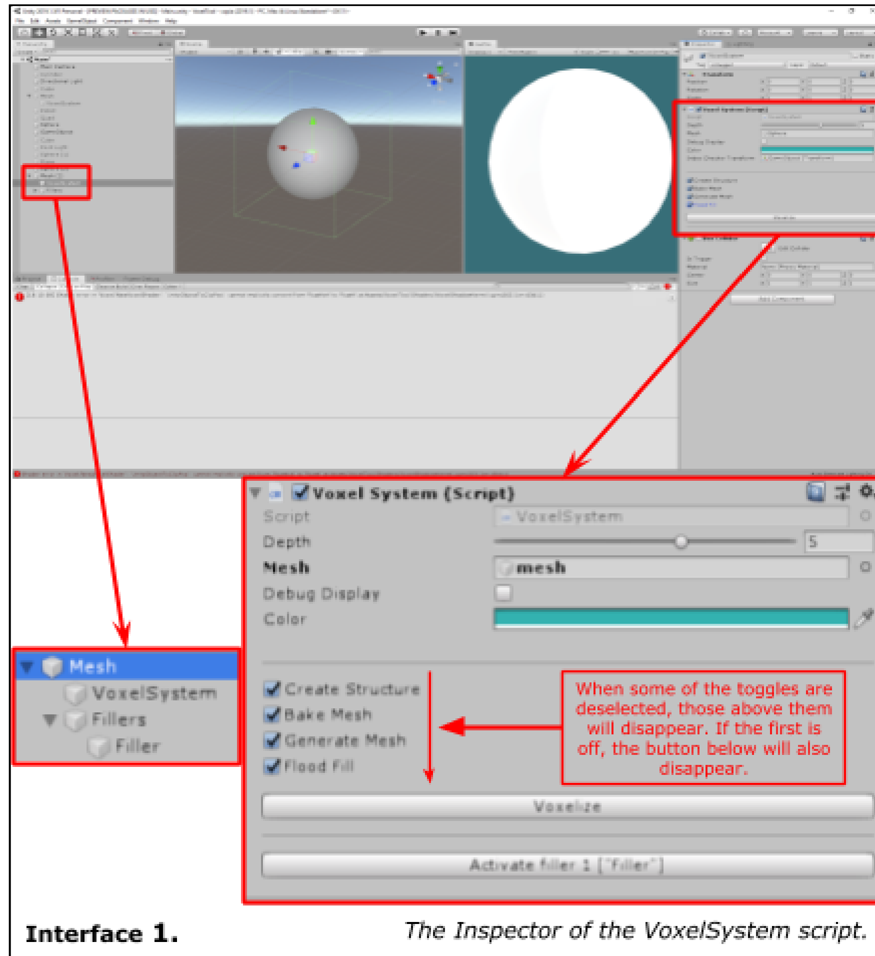
Due to the requirements mentioned above, a computer with the following basic characteristics is required, in addition to all the standard elements of a PC oriented to today's average games:

-Medium-high/high-end home graphics card range with a recommended VRAM amount of 4GB or more and support for DirectX 11 or higher or OpenGL 3.0 ES or higher. In the development an NVidia GTX 1070 of 8GB of VRAM was used.

-Quantity or greater than 4GB of system RAM, although it is recommended to have at least 8GB.

-Processor: by not using this version of the parallel computing project, the available physical or virtual cores of the processor does not really matter, but it is fundamental to be able to maintain a high clock frequency for extended times. There are, therefore, not discarded but poorly recommended, the laptop processors (due to their limited cooling capacity) or processors with low clock frequencies (less than 2GHz).

4.4. Interface Design



Interface 1. *The Inspector of the VoxelSystem script.*

For the interface Unity Inspector has been used. The class that manages it is the *VoxelSystemEditor*. In it you can enter a series of values: the depth of the octree that you want to generate, with a slider that includes from 0 to 8; a box where to specify the agent that has the mesh that we want to voxelize; a marker where we can activate a drawing with a series of colors that helps the user to see what is happening with his voxel system; the color we want the voxels to have. In addition, in a section separated by bars, we find a series of markers that work as specified in Interface 1. When the Voxelize button is pressed, all the tasks named in the labels of the mentioned markers will be executed in order from top to bottom. When the scene is loaded by entering the game mode, it is equivalent to pressing the Voxelize button.

The entire project, in the end, is in a Unity agent. As we can see in Interface 1, on the right we have the hierarchy of the scene with all the agents present in it. In our case, the *VoxelSystem* is located in the agent of the same name, which is the son of the mesh that it generates (and that will be applied,

therefore, in the parent object *Mesh*). As a brother agent of *VoxelSystem* we have the *Fillers* agent, which is a agents container that act as situational points in the space of the scene from the inspector, with the objective that the user places them in all the regions that they want to be filled with its voxel structure.

5. Development of work and results

This chapter will present a chronological journey through the development of the project, in order to observe both my academic growth during the course of the same as the reason for each of the fundamental decisions of the work done.

5.1. Work Development

I started the project by researching an appropriate way to represent three-dimensional space. This is how I easily found the data structure called octree (see in Figure 12). Each node represents a cube, and each cube can be divided into 8 cubes of equal size. Thus, we can recursively obtain a separation of the space with a variable precision for each one of the sections of it, according to the criteria that we follow when using these structures.

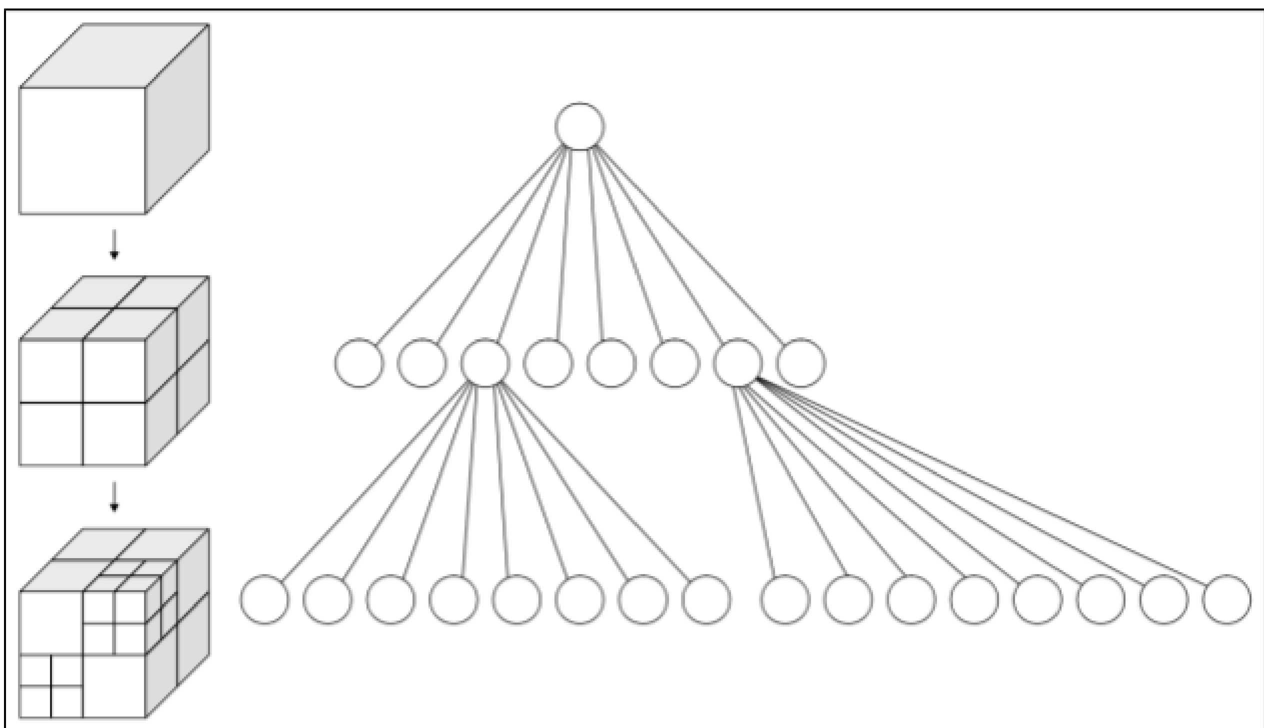


Figure 12. *Representation of an octree as cubes (left) and as a tree (right).*

Having seen this, I began to plan the structure and how each of the parties would communicate. With this I achieved a system capable of altering the scale

of each of the cubes. In the figure (See in Figure 13) you can see the result in Unity.

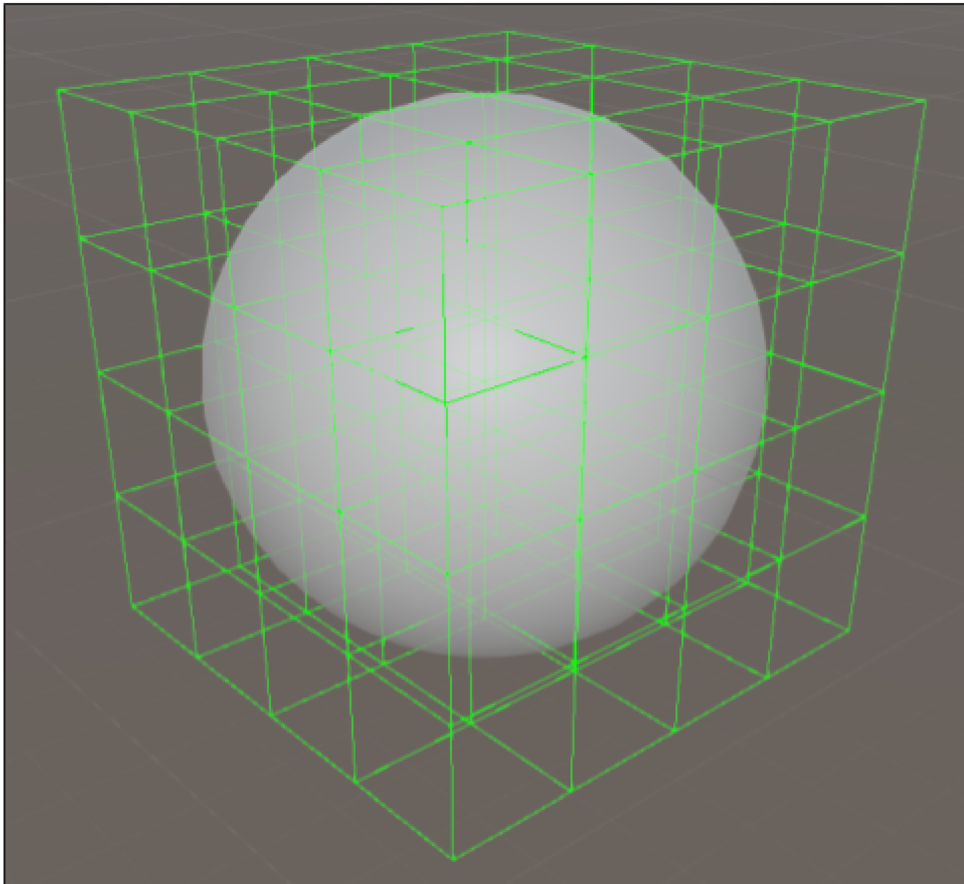


Figure 13. *The octree here is created but mesh information has not been baked yet.*

The previous octree has a depth of k = 2. The depth indicates the number of times the structure is subdivided. When k is zero, we have a single cube: the root node. The amount of memory consumed by the structure at each level of detail scale based on the following formula:

$$y = (2^{n-1})^{x-1} * m$$

Where ***n*** is the number of nodes per child, ***x*** the depth of the tree, ***m*** the memory consumed by each node and ***y*** the amount of memory consumed at the depth level ***x***. This gives us the following graph (See in Figure 14) for $n \in 2, 4, 8$, which corresponds to a bintree, quadtree and octree, respectively. In it we can see in the *x axis* the depth of the tree and in the *y axis* the spatial cost of the

depth level. The total cost of the structure is $\sum_{x=0}^{k}((2^{n-1})^{x-1} * m)$, where ***k*** is the maximum depth level of the structure.

Due to this excessive growth of the spatial cost of the structure, excessively high values of depth are discarded in their treatment in real time. It will also be necessary to minimize the number of nodes to increase the performance of the system as much as possible.

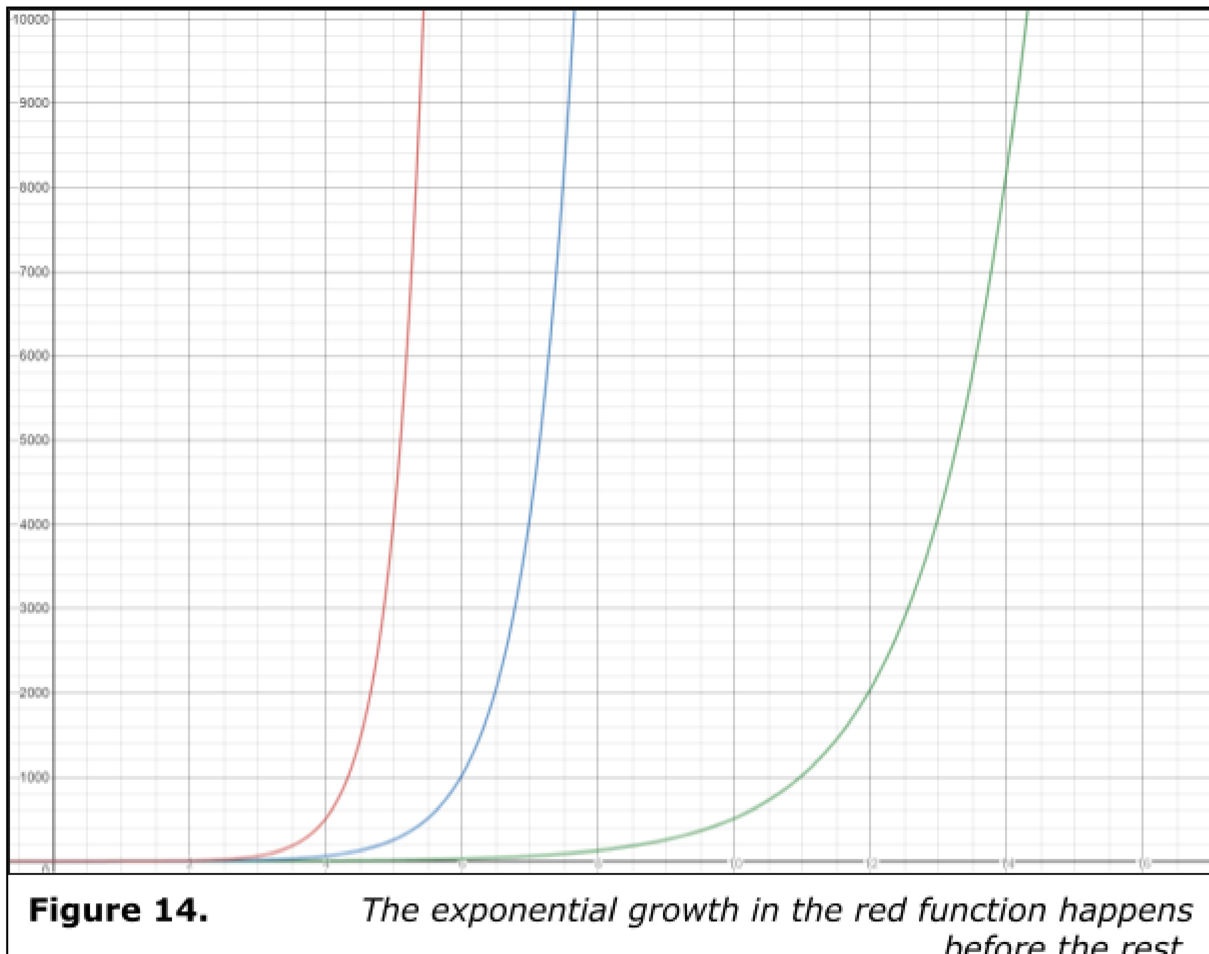


Figure 14. *The exponential growth in the red function happens before the rest.*

Then, during the development a fundamental problem of the structure was found: how could I search for a specific node without having to go through the entire tree node by node? This problem was going through my mind for a long time until, one day, in a conversation with a friend who was studying computer engineering, he told me that he was studying structures called hashtable. These structures allow finding elements within them in O (1) based on a parameter called key, which can be a word, a code, etc. Just what I needed.

Investigating this issue and its application I found something curious: the Z-order curve. This type of order when saving the elements in a two-dimensional

structure looks similar to the one in Figure 15. This is the way in which the quadtrees are ordered, and is mathematically applicable to three-dimensional spaces, that is, to octrees, presenting a similar shape (see in Figure 16).

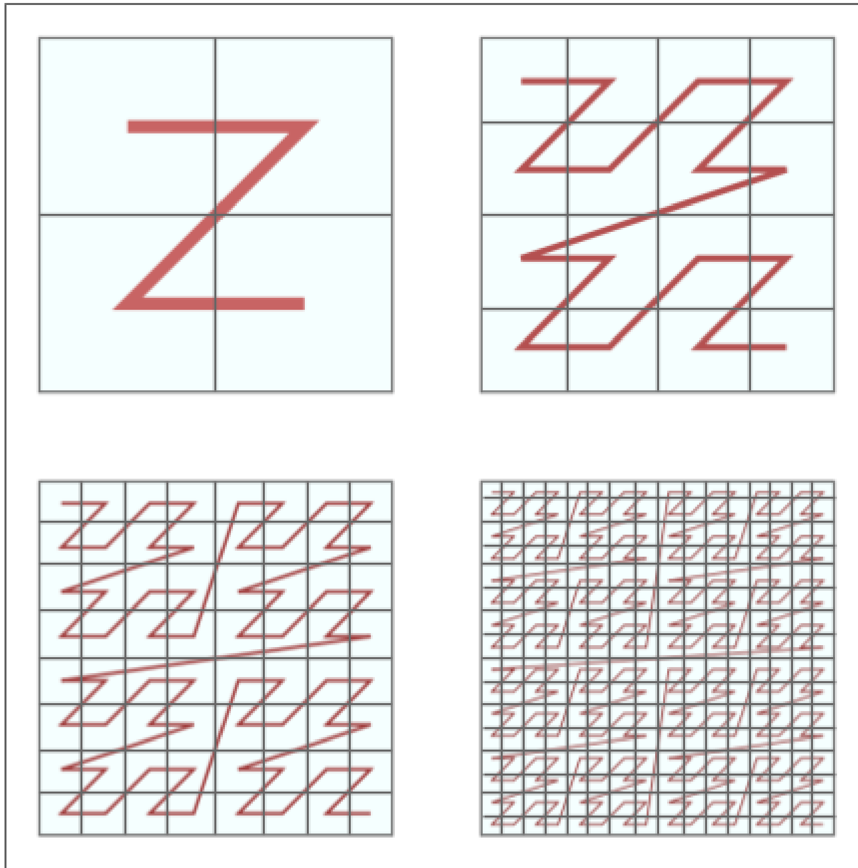


Figura 15. *The way quadtrees and octrees travels through the nodes has this type of Z pattern.*

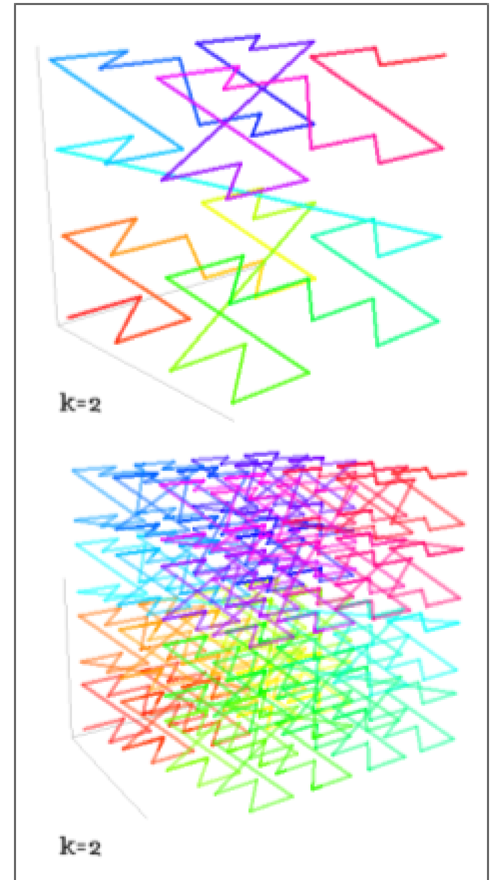


Figure 16. *Z-Order in 3D octree.*

This type of order fulfills a property: it is encodable by means of a code called code of Morton[1], that allows to convert into a single integer a coordinate of N dimensions encoded in its binary code. This allows to take a node from a position in O(1) using only bitwise operations. A summary of the operation can be seen in the two-dimensional case[3] in the figure (see in Figure 17). We will extrapolate this to a three-dimensional case.

	x: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y: 0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

Figure 17. *The Morton order in a quadtree is yxyxyx..., where -x-x-x... are the binary digits of x and y-y-y-... those of y.*

As a result of this, we have that, virtually, our system can reach a depth level up to 21. This is due to the fact that the three-dimensional coordinates of each node have to be encoded in a ulong-type variable, which comprises 64 bits of space. Therefore, each coordinate can be a maximum of 21 bits. This is not a major problem since a depth of 21 would be equivalent, with each node containing 117 bytes of information, to 154.2 exabytes (or $1.542 * 10^{11}$ GB), which obviously escapes from our technological capabilities, so we do not need more precision for making the Morton coordinates. In fact, given the architecture of the aforementioned system, the depth cannot be greater than 8, since it is equivalent to 2.243 gigabytes, and would not support 9 levels of depth, which equals 17.95 gigabytes, much more than our limit of 8GB of RAM.

With this clear and implemented matter, I went on to investigate how to detect which triangles intersected with which voxels to be able to reflect the mesh in the system. This task did not have to be very fast, since it is *baked* -precomputed- in the Unity editor before using it in the game. However, I did not have to greatly extend the development of the work of the designers of the game, so I needed a fairly fast algorithm. Investigating I found a huge amount of different algorithms for this purpose, but there was one that convinced me for its simplicity and elegance: the Möller triangle-cube overlap algorithm[2]. The algorithm is explained in more detail in the section on system design (see in Diagram 5).

The result can be seen with a depth value equal to 3 in Figure 18. We can also appreciate it better in an image captured with a orthographic camera from the side (see in Figure 19).

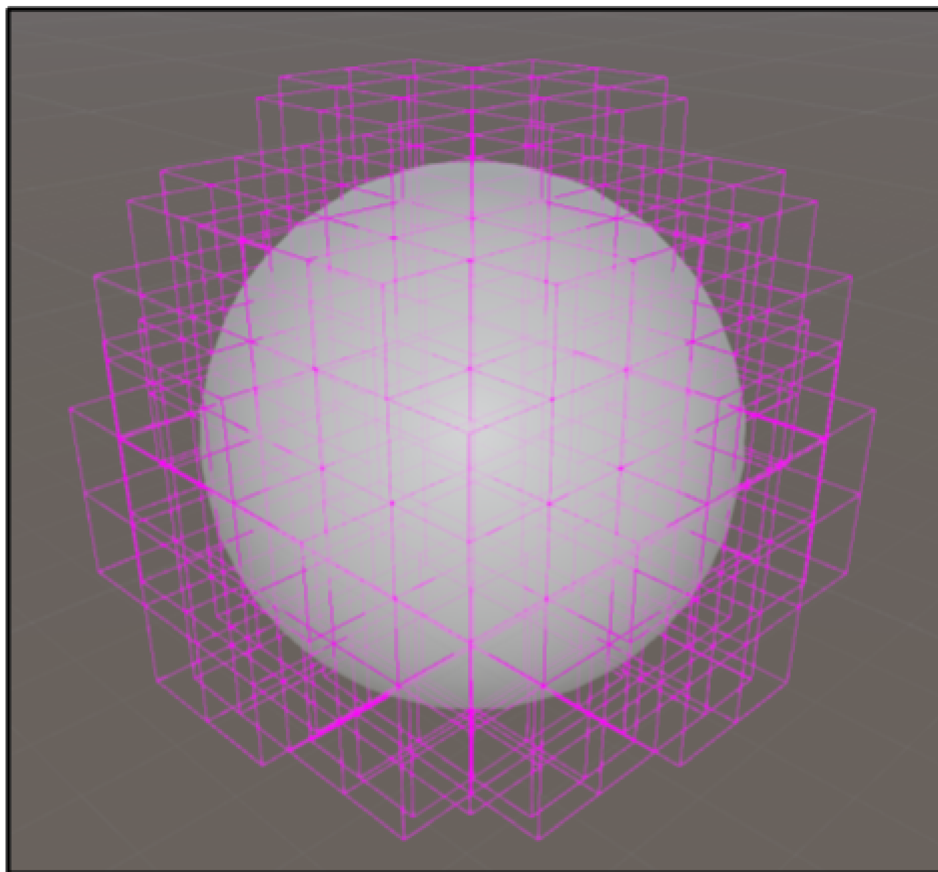


Figure 18. *Showed in purple wire cubes there are the border voxels.*

After this task, the structure is filled. This is done simply by using a standard *flood fill* algorithm[6] -the *cube tool* in many 2D image edition programs- in three dimensions that does not include diagonals. This will have to be done by the user himself, so that he decides which parts are really empty and which ones are empty.

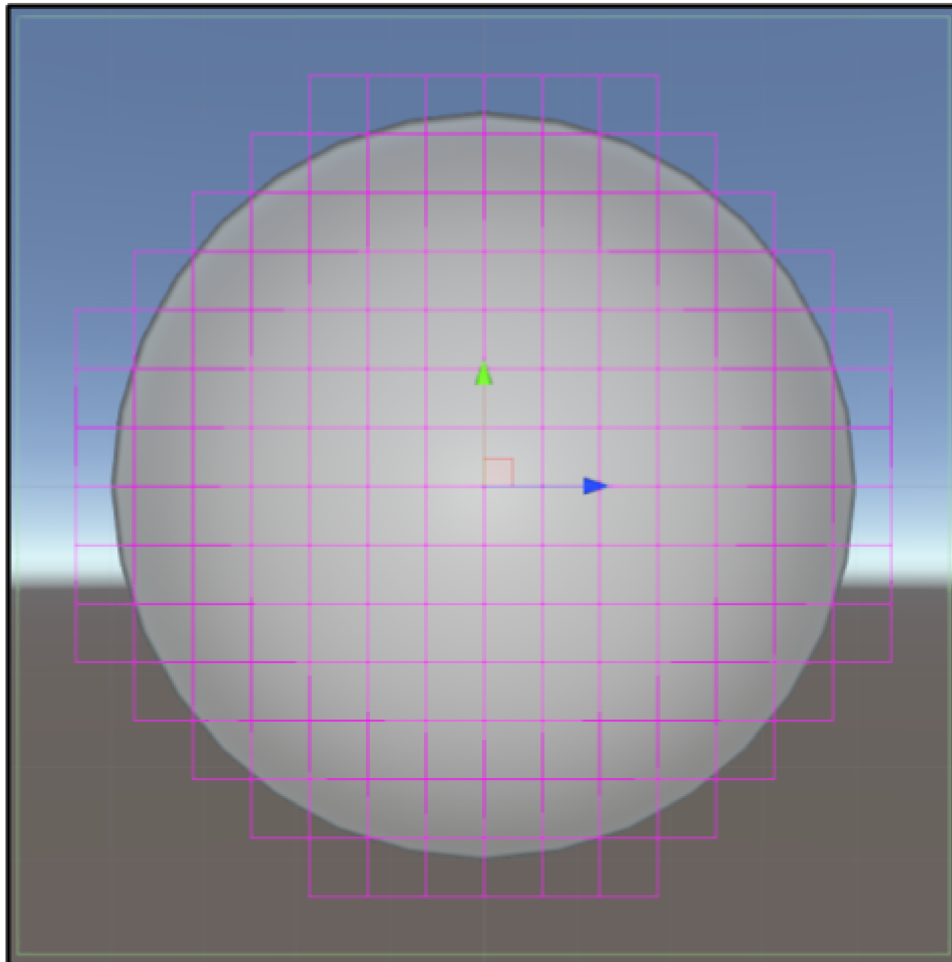


Figure 19. *Ortographic view from side. All the cubes contains at least a small portion of the target mesh.*

With all this, it is time to start visualizing the voxel structure efficiently, since the cubes that you have been seeing so far were drawn by a single thread of the processor, which was enormously expensive. To perform this task we will use the GPU, so that the cost is much lower.

To use the graphic card we use programs called shaders: ours uses a specific one called *geometry shader*[7], which allows adding vertices to the mesh that receives from the *vertex shader* (see in Figure 20).

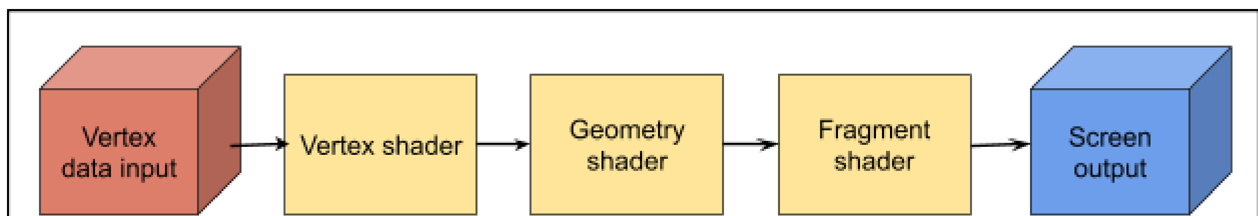


Figura 20. *The vertex data is processed in the vertex shader. The geometry shader creates new geometry from the vertex shader output and transfers to the fragment, which draws the pixels it will be showed on screen.*

This allows us to generate a mesh of points, where each point is a position of a node. It should be noted that only those nodes that make up the surface are transmitted to the GPU, since it is absurd to draw those parts that are not visible. As the geometry shader is executed for each vertex, we are executing it once for each node, which allows us to create the cube that visually represents the voxel. In the fragment shader we perform the necessary color and light calculations, so that we obtain an optimal visualization (see in Figure 21).

The shader, for reasons of time and complexity of combining shadows with geometry shaders in the Unity ShaderLab API, neither casts shadows nor is projected by them.

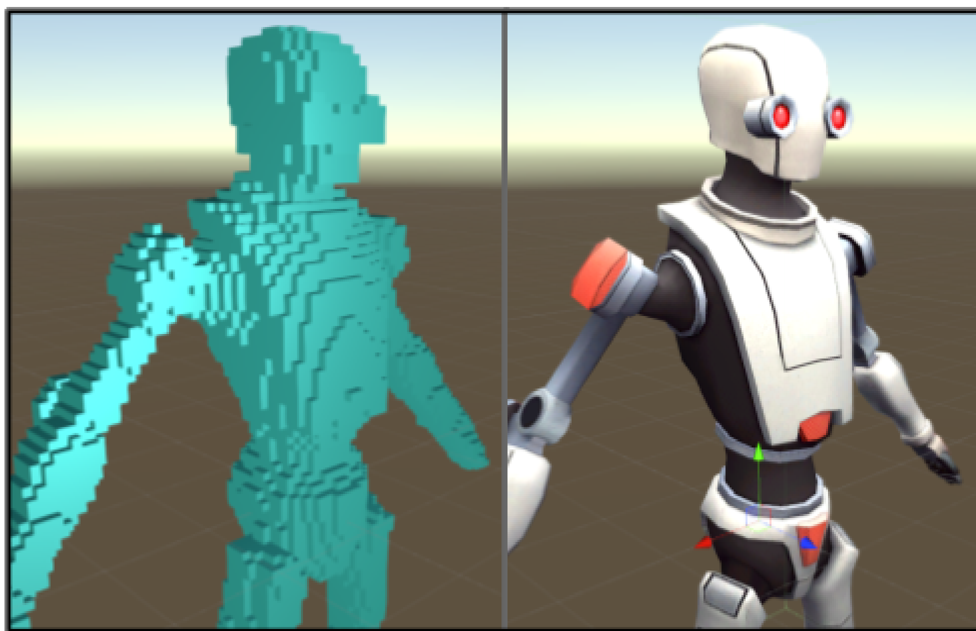


Figure 21. *In the left, the voxel structure. In the right, the target mesh of that structure.*

For the interaction I was thinking exactly what I should implement. The objective in the original Technical Proposal was to design a system of realistic destruction. However, in the final planning presented in this document, this required a quantity of time that, according to my research, greatly exceeds the time of dedication to the subject. However, leaving the project without any interaction with him could leave him lame, lacking of any practical employment.

For all this, I decided to implement two functions that allow to contemplate the potential of this type of structure, and to show its intrinsic qualities that differentiate it from the classic mesh system. One allows you to generate voxels with a certain radius without the need to use meshes; the other makes it possible to erase the voxels in a given radius, respecting the original concept of the volume given by the voxels.

In Figure 22 we can see a voxelized sphere to which voxels have been erased from three positions with a radius of 21 voxels.

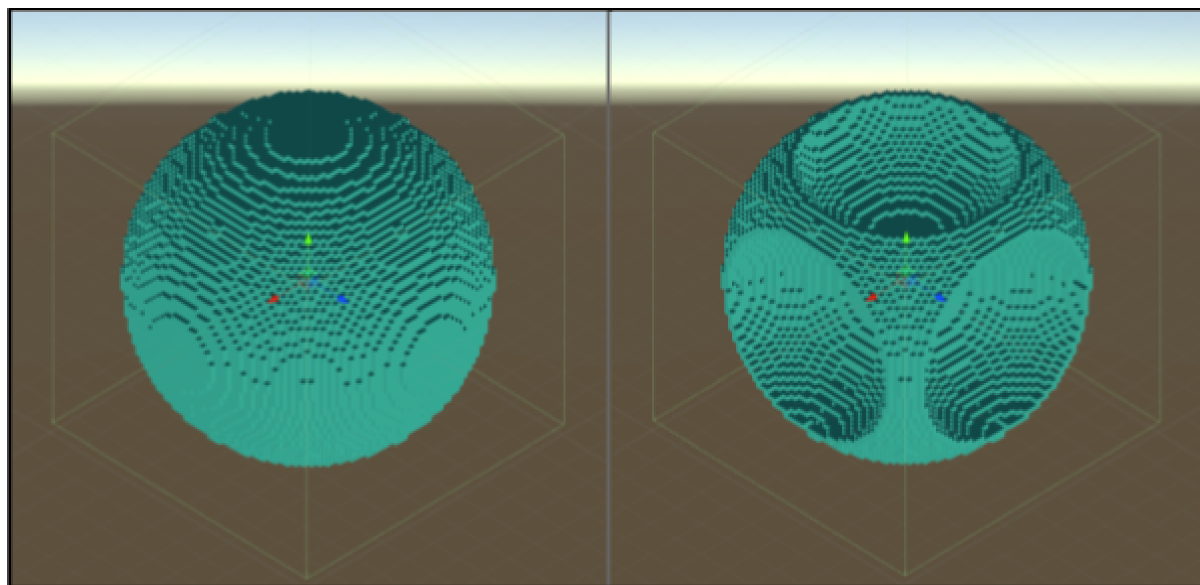


Figure 22. *Left, a voxelized sphere. Right, the same sphere erasing the voxels from three points in each axis.*

5.2. Testing

A great way to debug in Unity is to employ the built-in gizmos it provides. With them we can see in the 3D space where and what is there by using the color, the position and the scale of standard cubes, wire cubes, lines, spheres and so on. Here we made use of wired and non-wired cubes -because we are working with cubic voxels-, and their colors says us what type of voxel it is.

The way to see this debug is with the *Debug* toggle in the script inspector. With this, as we explain previously in *The Plug-in* chapter, the system will be showed with the shape voxels in wired purple cubes and inner ones with blue filled cubes. This mode, depending on how high is the depth level chosen, the performance can come down and the inspector will run at 15-30 fps, although the objective of this debugging tool is not running in the game.

Also numerous of computational cost test were carried out. Those helped me choosing what approach to take for implementing some algorithms, like reducing the amount of triangles to compute for the children nodes of the current node in the *Bake()* operation by discarding the triangles which already has been established as not intersecting the prevailing AABB node.

In the Figure 23 we can see the difference between triangle-discarding approach and the old one. The last measurement of the Old Bake was not taken

because of the large amount of time it would took to take enough results -it was more than 10 minutes computing when I aborted the test-.

Depth	New *Bake()* time (s)	Old *Bake()* time (s)
3	1.4	2
4	2.82	4.2
5	6.15	11.3
6	17	37.7
7	57	145.4
8	294	*[undefined]*

Figure 23. *Time table of each version of the Bake() operation. The time is the average between three time measures.*

In order to check the performance, we used a built-in tool of Unity called Profiler (see in Figure 24), which allows the user to see when and how much resources a process is taking. With this tool you can study the CPU usage, the RAM usage, the rendering process values -vertices, pass calls, triangles and so on-, the network usage and physics of your scene. You can even code your own profiler if you need it.

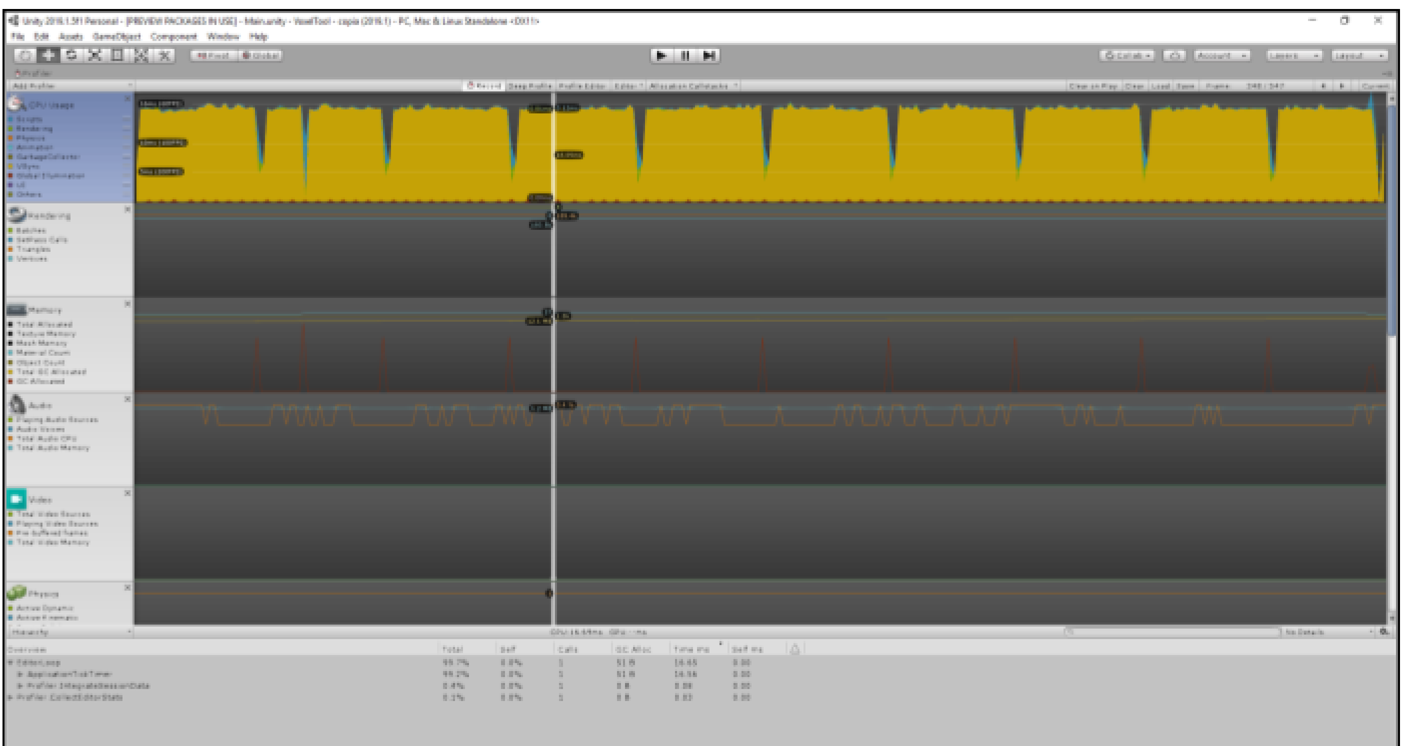


Figure 24. *Here we see the profiler. We can choose some point in the timeline to see how resources are distributed.*

With this profiler we saw some low performance *spike* on the CPU usage, dropping the frame duration to 250ms. Researching a bit deeper in the tool provided info, there was a process using the 99.8% of the Unity CPU usage. This was the Unity self Garbage Collector that, for some reason, was consuming a lot of resources erasing the heap memory. This seemed a dead-end for the solution, as Unity GC is built-in and you cannot do a lot with it.

Some research about that question lead me to how the garbage collector works. The method it follows is the Boehms-Demers-Weiser garbage collector[8], which is a GC that stops all the process running until it finishes cleaning the unused memory. It means that it can trigger in real time applications a huge drop in the frame rate. There is no real solution for avoiding this when you work with a lot of memory usage in Unity.

Nevertheless, in the research I made I found something interesting: the Unity team is working on an incremental version of this garbage collector. Incremental means that the collection of unused memory is made step by step reallocated in all the frames. This does not reduce the amount of time it takes to do the process, but by redistributing it allows the user to not see those 250ms frames that kills the global performance of the game or the application.

This was implemented in the 2019.1 version of Unity, so I had to move the whole system to this newer version and to activate it from the project settings -it is not allowed in the default configuration for his really early version.

Finally, this solved the problem, but it was one more thing affecting the performance according to the profiler: some process in the editor windows was causing some noticeable decrease of the performance. All the research I made wasn’t enough to solve this, but I found out that maximizing the Scene/Game window makes the problem disappear, so it should belong to some internal process of the Unity Editor. However, this will not affect the performance in the game build, so I decided to spend the time in other more important matters.

Having seen this, I will talk about probably the most important debugging tool I used in the project: the Visual Studio *step-by-step* debugger. With it we can go through our code line by line and see how the variables are changing (see in Figure 25). In other words, we can trace our code automatically in order to see what is happening with a certain problem. This was very useful in the making of the *Bake()* function and, along with the gizmos, was decisive in the completion of the project.

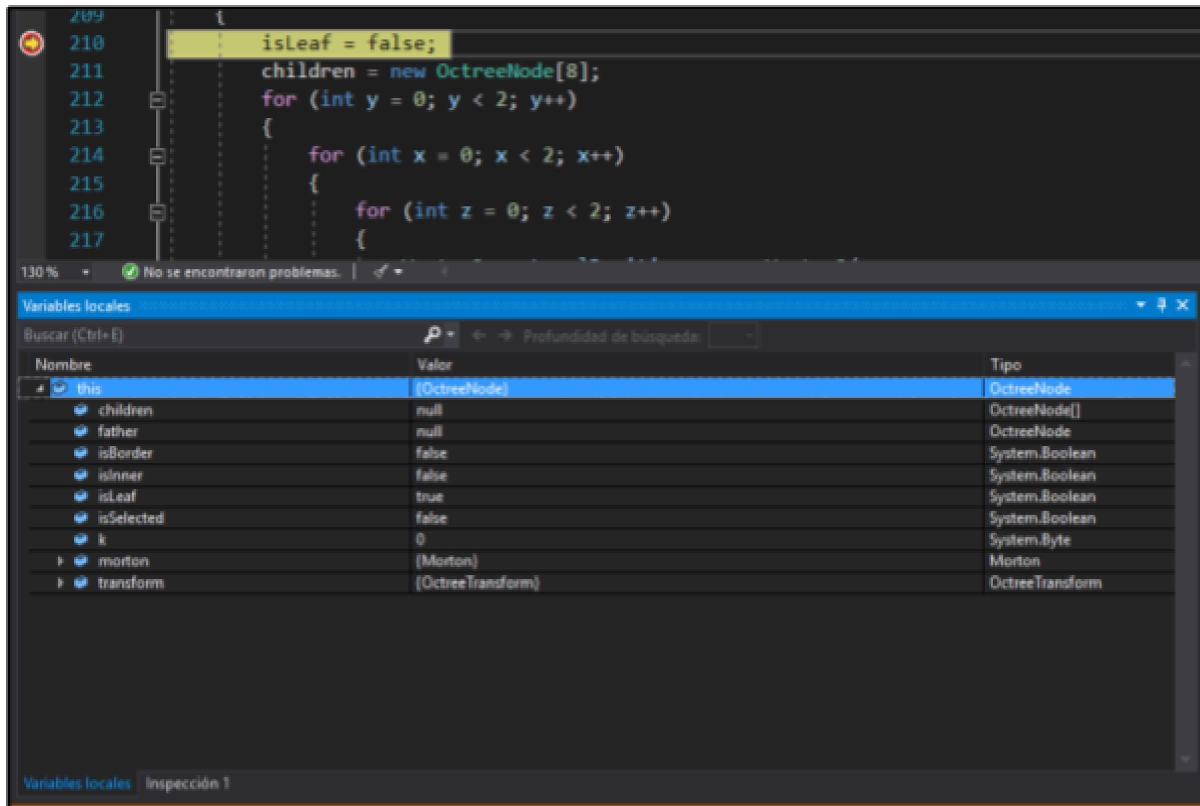


Figure 25. *With the variable inspector we can trace the variables of the process*

5.3. Results

The final state of the project is really satisfactory. In the beginning it was a job that I considered as a challenge to what I had learned throughout my career. I have put all my knowledge into this project, even those that I did not have in the beginning. The following results have been obtained:

-I have made a solid and optimized data structure.

-The structure allows to generate automatically, except for a simple step that has been left to the user's criteria, as is the selection of which areas should be filled.

-The structure is visualized making profit of all the hardware acceleration technologies the employment of GPU instanced meshes gives us, although it has no shadows.

-The interaction system, although it can be simple, allows its use in destructible environments. Its simplicity of implementation by the user allows multiple possibilities of utilizations by the person who desires to employ this tool.

6. Conclusions and future work

6.1. Conclusions

This project has been the most complex I have done throughout my academic development. I have applied in it everything I have learned throughout my university education. Even for what I did not know and I had to investigate on my own, this knowledge has served as a basis to better understand what I read. Without the knowledge of mathematical language that I received in the first year of the degree, I could hardly have understood correctly the Möller paper where the mathematical procedure I follow to check box-triangle intersections is explained, and much less to be able to compare it with other procedures in order to find the fastest of them. Without the base that graphic computing gave me, creating a shader that allows visualizing such a quantity of information would have taken too much time to be able to integrate it into the project. Without the ability of abstraction that gave me the subject of software engineering, working among so many connected systems would have been a problem of a greater order than it was. And so for many more cases that occurred throughout the project, probably too many to name in this section.

Although, there have also been many other knowledge that were not acquired in the college. These I got in my free time; hours and hours absorbed in front of the computer, learning from what other people did. It is incredible the community of programmers that there is in the network sharing content, solutions to problems and detailed guides to create all type of systems. All these programmers, although not at the same level as in the faculty, have contributed in a decisive way in my training and, above all, in my creativity as a software programmer for video games.

In addition, I have also learned something in the development of this project. Theoretical knowledge, organization, but mainly work under pressure, and learning to be flexible with the distribution of time if necessary -which, throughout the development, has had to be quite frequent due to the rest of my responsibilities-. There is an old war saying that no plan survives contact with the enemy, and although this is an illustrative hyperbole, there is a certain truth locked in it: improvisation, like planning, are two vital qualities in the development of a project with these characteristics.

6.2. Future work

Regarding the project, it is obvious that it is enormously expandable. What is presented in this text is the solid basis on which an enormous amount of functionalities can be based: simulation of liquids, optimized physics using Morton, calculation of structures tensions, and many more capabilities. I would have loved to include all this in the project, but the subject has a certain time and to introduce all this in that time is a nearly impossible task.

However, I am clear that I do not want to limit myself to what is presented here. Increase the optimization of the algorithms through the parallelization of processes, calculation of physical forces and improve the visualization of the structure are a series of objectives that I have in the short-medium term in collaboration with some computer engineers with whom I maintain a friendly relationship. And, who knows, maybe if the project moves forward enough, I will contact the Unity development team for its native integration in future versions of the engine.

7. Bibliography

[1] Barten, Jeroen. Morton encoding/decoding through bit interleaving: Implementations. https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/ Accessed: 06/26/2019

[2] Akenine-Möller, Tomas. Fast 3D Triangle-Box Overlap Testing. http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/tribox_tam.pdf Accessed: 06/26/2019

[3] Gargantini, Irene. An Effective Way to Represent Quadtrees. http://www.csee.usf.edu/~tuy/Literature/QTree-Represent-CACM82.pdf Accessed: 06/26/2019

[4] Green, Daniel & Hatch, Don. *Fast Polygon-Cube Intersection Testing, Gems V p.375-379.* http://index-of.co.uk/Game-Development/Programming/Graphics%20Gems%205.pdf Accessed: 06/26/2019

[5] Voorhies, Douglas. *Triangle-Cube Intersection, Gems III p.236-239.* https://doc.lagout.org/Others/Game%20Development/Programming/Graphics%20Gems%203.pdf Accessed: 06/26/2019

[6] Wikipedia. *Flood fill algorithm*. https://en.wikipedia.org/wiki/Flood_fill Accessed: 06/26/2019

[7] Liu, Jingyu. [Unity3D] Intro to geometry shader. https://jayjingyuliu.wordpress.com/2018/01/24/unity3d-intro-to-geometry-shader/ Accessed: 06/26/2019

[8] A garbage collector for C and C++. https://www.hboehm.info/gc/ Accessed: 06/28/2019