ACHIEVEMENT OF VISUAL REALISM IN A VIDEO GAME USING FREE SOFTWARE

Author:
Alex Monzó Machirant
Supervisor:
Inmaculada Remolar Quintana
June, 2022

UNIVERSITAT JAUME I

ABSTRACT

Visual realism is necessary for most of the AAA (and some independent) games, as they are usually more focused in this aesthetic than a stylized one, but normally the software used to achieve that realism requires expensive licenses.

The purpose of this project is to both make an analysis on how to achieve visual realism in a game engine and to use that knowledge to create an interactive scene to show that it is possible to reach that level of realism using free software. Therefore it will consist in two parts: a theoretical one that will collect all the important aspects to consider when making a visually realistic game and a practical one that will show how taking care of those aspects can indeed get the desired result through the use of a free engine and free modeling and texturing programs.

KEYWORDS

3D Modeling, Visual Realism, Free Software, Texturing

CONTENTS

1 TECHNICAL PROPOSAL

1.1 INTRODUCTION

Visual realism is something that has been sought after for a long time, and nowadays there is finally the possibility to create something that can be indistinguishable from reality. The 3D industry is a market that is constantly growing due to industry 4.0, which is trying to fuse the real world with technology, creating an augmented reality like the metaverse, for example.

This project intends to explore the different technologies available to create realistic 3D models and environments within the framework of a videogame by creating an explorable scene in the style of Cyberpunk, making use of only free software to demonstrate their capabilities of obtaining highly realistic results in comparison to others that require a license. It tries to show the strengths of taking alternative paths to achieve similar results while also creating a community that makes it possible to do so for free.

If the industry changes following this path, it would greatly reduce the economic costs while still maintaining the quality of the product, and it is also an incentive for independent developers to have the opportunity to make great games that otherwise (because of the economic costs) would not be able to do it.

1.2 WORK MOTIVATION

There are a couple of reasons why this topic has been chosen. The first one is that there is an intention to work professionally in the 3D industry, so it will be interesting to learn about visual realism, as it is not the only field in 3D modeling, but usually the most demanded one. The second reason is that, as a user of the free 3D modeling software **Blender**, which is a really powerful tool, usually companies only look for experience in software with a really expensive license (although this is gradually changing) and this is an economic barrier for most people. Moreover, new alternatives are emerging for other paid softwares like **Substance Painter** (in this case **Quixel Mixer**), so it is worthwhile to try them so there is not a dependency on licenses to keep learning and working.

1.3 RELATED SUBJECTS

The subjects related to this work are mostly the ones that have to do with 3D modeling, such as *3D Design* and *Character design and animation*, and with graphics, like *Graphic computing* and *Game engines*. Although the latter is more focused in the **Unity** engine, what has been learnt in it can be applicable to **Unreal Engine** using different approaches.

VJ1216 - 3D DESIGN
VJ1221 - GRAPHIC COMPUTING
VJ1226 - CHARACTER DESIGN AND ANIMATION
VJ1227 - GAME ENGINES

1.4 OBJECTIVES

The main objective of this project is to have completed a Cyberpunk scene at the end of it, made solely with free software. To achieve this, the main objective can be divided in three more specific objectives necessary to obtain the expected results:

- Researching and collecting all the important information regarding visually realistic modeling.
- Modeling, texturing and arranging an interactive scene using everything that has been learned.
- Showing that free software is capable of the same results obtained from other software that require a license.

1.5 TOOLS AND SOFTWARE

In accordance to the topic of the project, all of the software used will be free:

Unreal Engine 5: A game engine property of Epic Games. It is free and usually recommended when looking for realistic graphics.

Quixel Mixer: A software for texturing models, similar to Substance Painter but still in early access.

Blender: Multi-disciplinary, free and open source software used for modeling, sculpting, animating and much more.

Krita: Software usually used by artists that allow image editing. Mainly used for the modification of textures in this project.

Meshroom: Open source software used to get a 3D textured model using the photogrammetry technology.

Instant Meshes: Free software used to reduce the number of vertices of a high poly mesh.

2 PLANNING AND RESOURCES EVALUATION

2.1 PLANNING

The final degree work has 12 credits assigned, which translates to 300 hours. As the objective is to present it on the first call, they will be spread over the course of three months. The Table 1 shows how the different tasks will be organized regarding the time:

TASK	HOURS
Research	40h
Writing the memory	40h
Preparing presentation	10h
Technical proposal	5h
Modeling the assets	120h
Creating materials and texturing	50h
Arranging the scene	25h
Testing the scene and interactivity	10h
TOTAL	300h

Table 1. Initial Planning

In the Table 2, a Gantt diagram can be seen to represent better the organization of tasks, although it is possible that some of the work (especially the modeling and texturing) overlap.

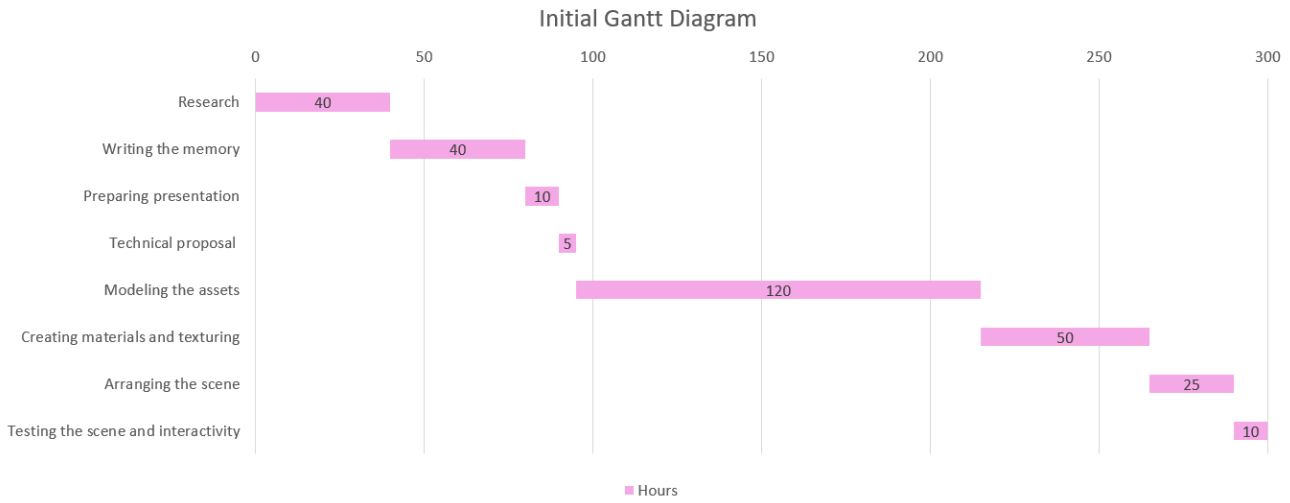

Table 2. Initial Gantt diagram

2.2 RESOURCE EVALUATION

In this section, the total cost of the project will be analyzed. This depends on different factors, as the salary of the worker or the price of the hardware. In this case, the advantage is that the cost of the software will be 0.

2.2.1 Hardware

The general and most important pieces of hardware for this kind of project are the **CPU**, the **graphics card** and the **RAM** memory. For this work, the following hardware has been used:

- 16Gb of RAM ≈ 70€ [1]
- NVIDIA GeForce GTX 1070 8Gb ≈ 379€ [2]
- Intel Core i7-6700 3.40Ghz ≈ 255€ [3]

Adding to this around 250 euros for the rest of the components like the motherboard or the storage, will result in **about 1000 euros**. For this price it will be just enough to be able to work, although it is recommended to have better specifications to be able to use multiple programs at the same time.

2.2.2 Salary

Based on different statistics about the salary of 3D artists in Spain, the average salary is about 12 euros per hour[4][5]. Taking this into account, 300 hours of work will result in **around 3.600 euros** for the whole project.

2.2.3 Conclusions

Adding both the salary and hardware would result in a cost of 4600 euros approximately. It can be interesting to compare this with what would cost to use the alternative versions that require a license:

- **Substance Suite:** 19.99€/month[6]
- **3ds Max:** 279€/month[7]
- **Zbrush:** $43.05 /month[8]
- **Agisoft Metashape (photogrammetry):** 179€, single payment[9]
- **3D Coat (retopology):** 379€, single payment[10]
- **Photoshop:** 36,29 €/month[11]

As stated before, the developement time planned for the project will be **three months**. Taking this into account, the total cost of the software would be: **Substance Suite** (19.90€ x 3) = 60€ + **3DS Max** (279€ x 3) = 837€ + **ZBrush** (43.05€ x 3) = 130€ + **Agisoft Metashape** = 179€ + **3D Coat** = 379€ + **Photoshop** (36,29€ x 3) = 109€ = **1.694 €**. It is a considerable difference.

3 SYSTEM ANALYSIS AND DESIGN

Here will be explained the general approach taken into account for the development of the project, based on the available resources and the objectives of it. In addition to that it will elaborate on the artistic and technical decisions.

3.1 SYSTEM ANALYSIS

Before starting with the development of the project, it will be necessary to organize how the work is going to take place (Fig. 1). First of all, a research of several topics regarding the software and techniques must be made. **Unreal** is an engine that has not been used in any of the subjects of the degree, so it will require some experience, being it using the program or learning about it. **Quixel Mixer**, on the other hand, is a new software, and as with all the new technologies, it needs to also be learned. It has an official YouTube channel regarding this information that will be useful[12]. Finally, photogrammetry is also a technology that has not been practiced on the degree, so it will need research like the other topics mentioned.

Once all the important information has been gathered, it will be the moment to collect all the references. As the topic is Cyberpunk, which will be explained later why, and it is quite popular at the moment, there will be no problem when looking for inspiration.

After having the references needed, a blockout will be made with Blender, not only to get a glimpse of the structure and placement of the objects, but also to see if the references have been enough or if it is needed to add more elements to the scene that require more images to take inspiration from[13].

When the blockout is finished, the modeling and texturing can begin. Depending on the type of model, different techniques will be used. For this step, **Blender, Quixel Mixer, Meshroom** and **Instant Meshes** will be used.

After getting all the models needed, they can be exported to use in the **Unreal Engine**. There, the meshes can be arranged as intended and the materials configured. Besides that, other aspects have to be taken care of, like the illumination, Post-Processing or the creation of FX[14]. While using **Unreal** in the development, it is expected the need to research again for specific topics.

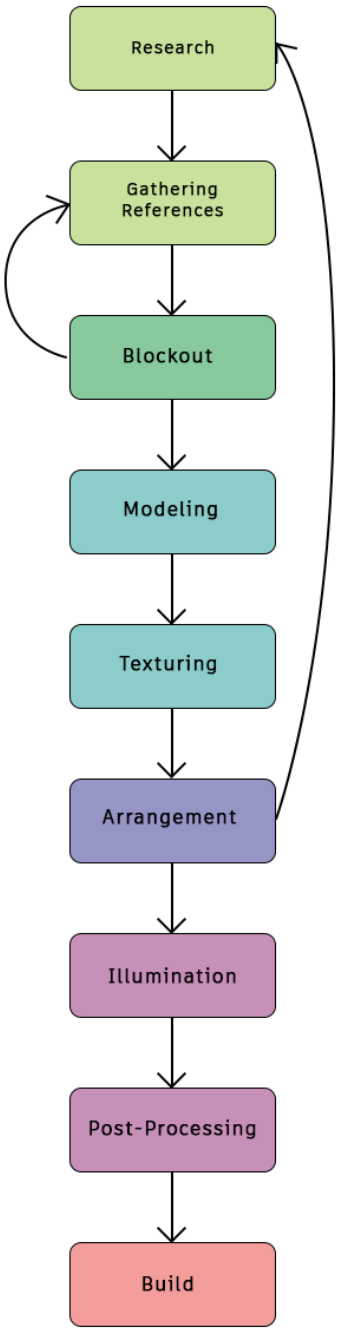

Fig 1. Analysis scheme

As this project will serve more like a technical demonstration than a videogame *per se*, the programming involved will be mostly regarding the movement of the player. On top of that, **Unreal** allows the use of what they call *Blueprints*, which greatly simplifies the implementation of interaction.

The last step is to build the level and check that everything works as expected.

3.2 DESIGN

The theme chosen for the project is the Cyberpunk aesthetics. This aesthetic has been chosen, on the one hand, because it allows to explore the new *Lumen* technology from **Unreal Engine**, which performs a dynamic global illumination with a low cost, so the illumination of all the neons work in real-time without baking and with amazing looks[15]. On the other hand, usually Cyberpunk is stripped of the “punk” part, meaning that its original political statements are despised only to be left with the visuals. In its origins, it is a critique and an hyperbolization of capitalism and its tendency to monopolize everything, which is really pertinent to this project, as the trigger for this idea was seeing how a lot of the software frequently used by artists were all being bought by large companies like **Adobe**. Ultimately, this project takes both aspects into account.

3.2.1 Game Atmosphere

Usually, when photorealism is used in video games it is to make the players more immersed in the atmosphere they are trying to create. In this case there is also an intention to do so. In this scene, the objective is to achieve the feeling of the Cyberpunk aesthetic, with a lonely, saturated atmosphere, based on the moodboard shown in Fig 2.

Fig 2. Atmosphere moodboard

To achieve this, different resources will be used, not only the visuals, but also the sound and other effects.

3.2.2 Game Concept

The game will consist in what is usually known as a “walking simulator”, as the point of the project is solely to show the graphics. The player will take control of a character in first person and they will be able to explore the scenes as they please, making it more similar to a technical demonstration.

3.2.3 Scene Design

The scene will be a Cyberpunk inspired street, as it is a perfect way of showing artificial lightning interacting with the materials with its neons and wet alleys. Some quick sketches with a top-down view and some of the views of the scenery can be seen in figures 3, 4 and 5.

Figs 3 (top left), 4 (top right) and 5 (bottom). Scene sketches

It will be a little street with a highway on top, some cars parked at the side of the street and a cul-de-sac at the corner. The street will be all surrounded by buildings of different types. Behind the shorter buildings the city will be able to be seen in the distance.

3.2.4 Blockout

The blockout will be useful to get an idea of how many buildings will be needed to fill the scene, their height (because they can't be taller than the highway if they are below) and some other objects like the garbage containers. It is also a good way to know approximately the measures of the street, etc. In short, it is the first step to organize the work that will take place later on[13]. The initial blockout that has been made can be seen in Fig 6:

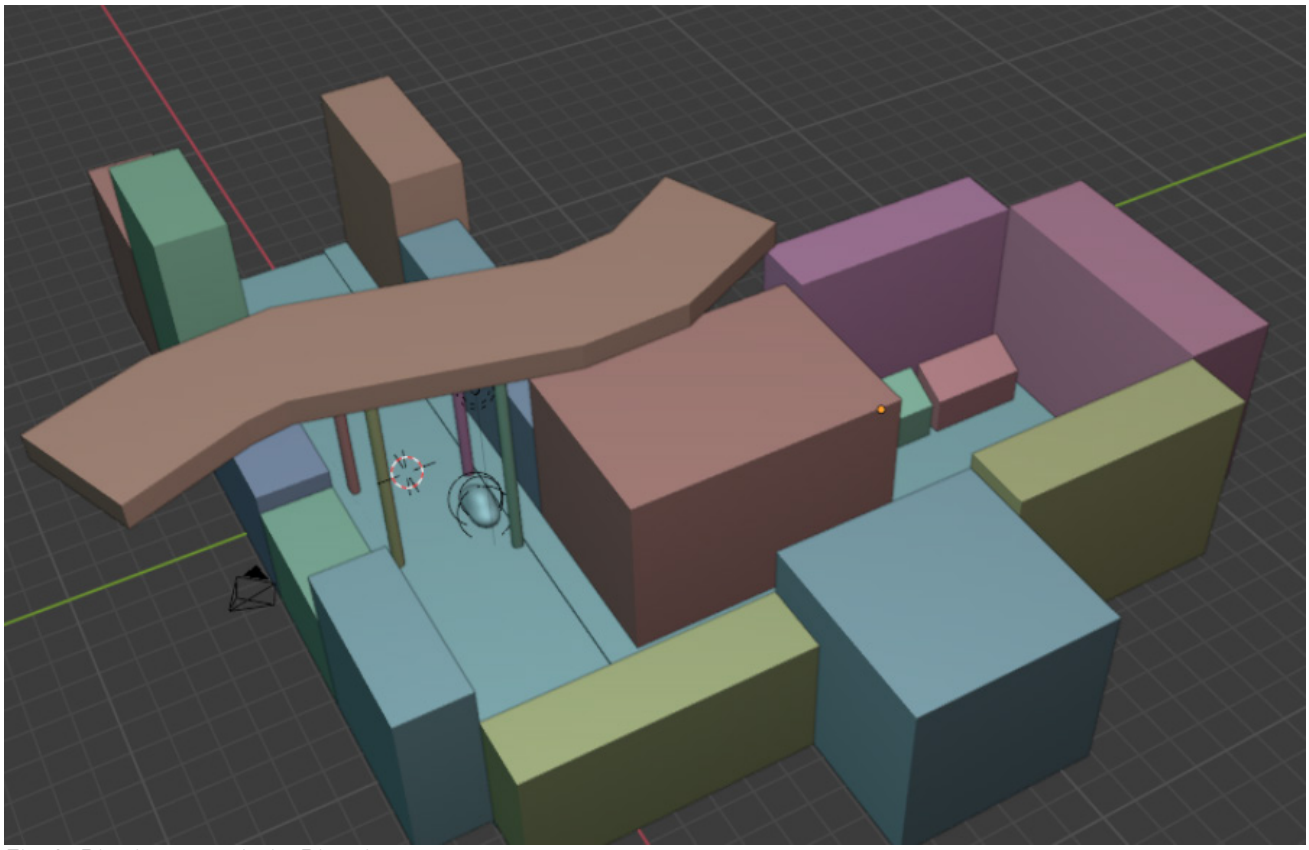

Fig 6. Blockout made in Blender

4 DEVELOPMENT

4.1 WORKFLOW

With all the different software available for free, it has been possible to use different techniques to achieve the desired result, be it a quick way to model something, a certain appearance, etc. For the most part, a general workflow has been followed to obtain the majority of the models, but in some cases, alternative ways of modeling have made the job faster and easier, and even with greater results.

A breakdown of each method is provided below:

4.1.1 General Workflow

As stated before, this has been the method used for most of the models for this project. Fig 7 shows the flow of the work starting from the image references.

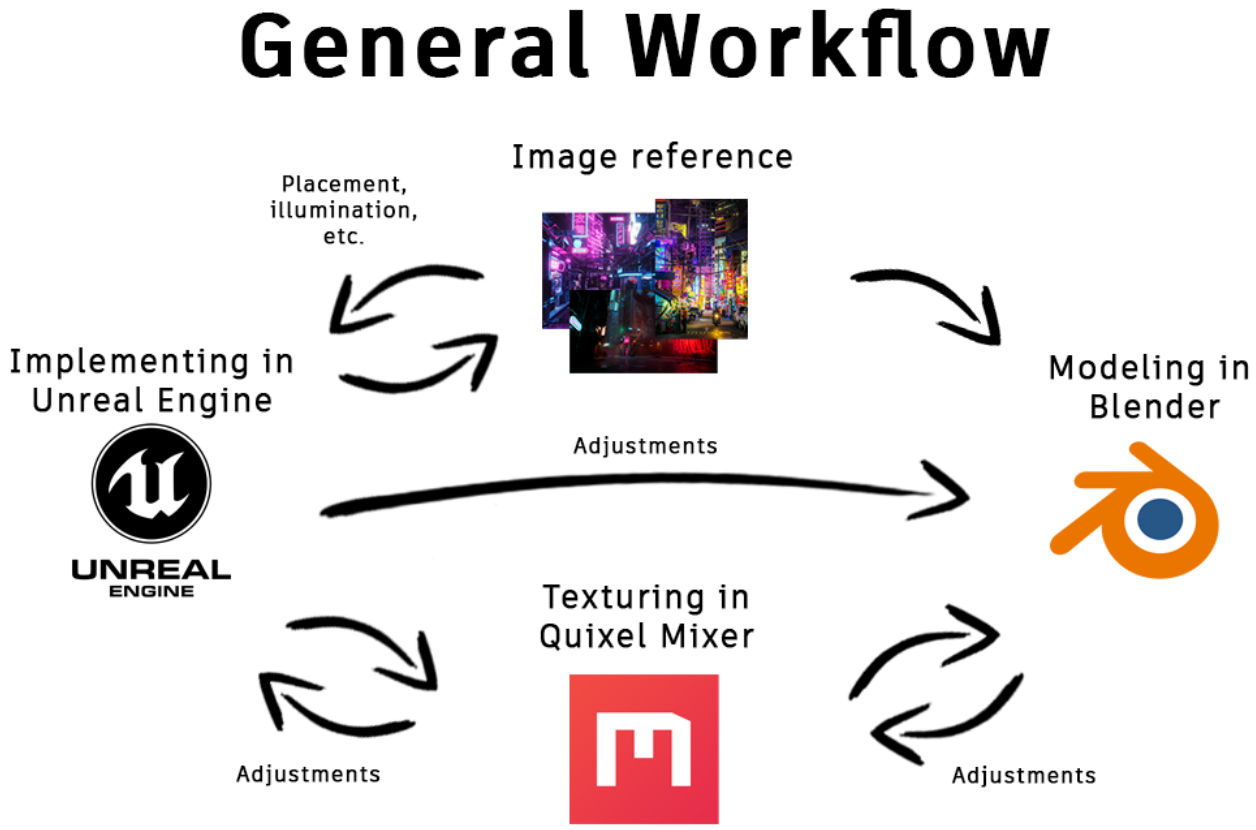

Fig 7. General workflow diagram

4.1.1.1 From references to modeling in Blender

The steps followed in this part can be quickly seen in the diagram shown in Fig 8:

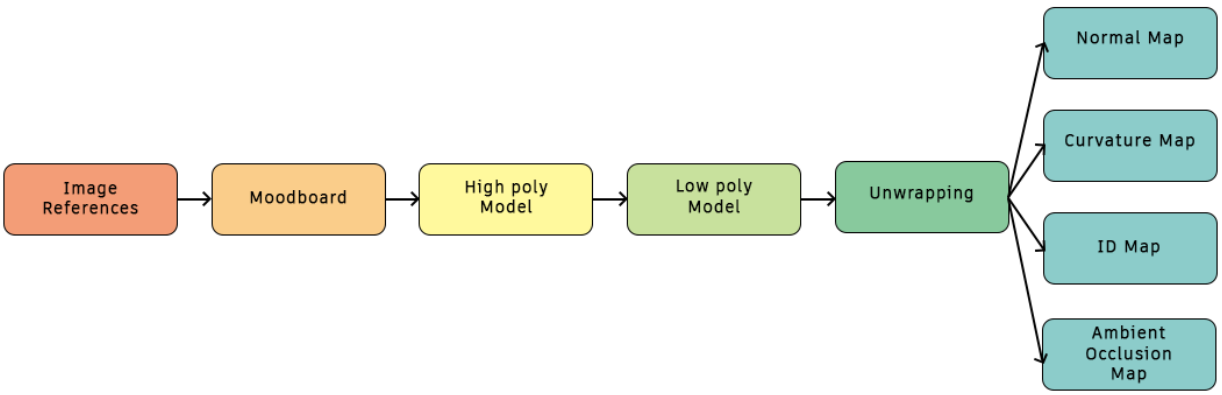

Fig 8. References to Blender diagram

First of all, a “moodboard” with all the image references for each object or building is made. After getting an idea of how the model is going to look, the modeling process in **Blender** can start. Everything has been kept low poly for the final result, even though the new technology from **Unreal**, *Nanite*, allows an enormous amount of polygons while keeping the performance stable[16].

In Fig 9, an example of one building can be seen. After looking at the references, the idea of making a garage came up, so a moodboard with all the images with similar concepts was made. Based on that, a model of the garage was created.

Fig 9. Model obtained based on the moodboard

Once the model has been created, it will be required to have the *UV*’s ready. This will be made using the orthodox method: creating seams on the edges of the mesh and then using the *Unwrap* option. This however will be for the general workflow, as later on alternative methods will be shown.

If the model is complex, extra steps are needed. First of all, the high poly mesh is made, and after that, based on that mesh, the low poly one (made by hand in this case). It is required to have both meshes in the same position, as the next step is to create the maps for the low poly mesh. An example can be seen in Fig 10 with a trash bin that had too many faces, so a low poly version was needed.

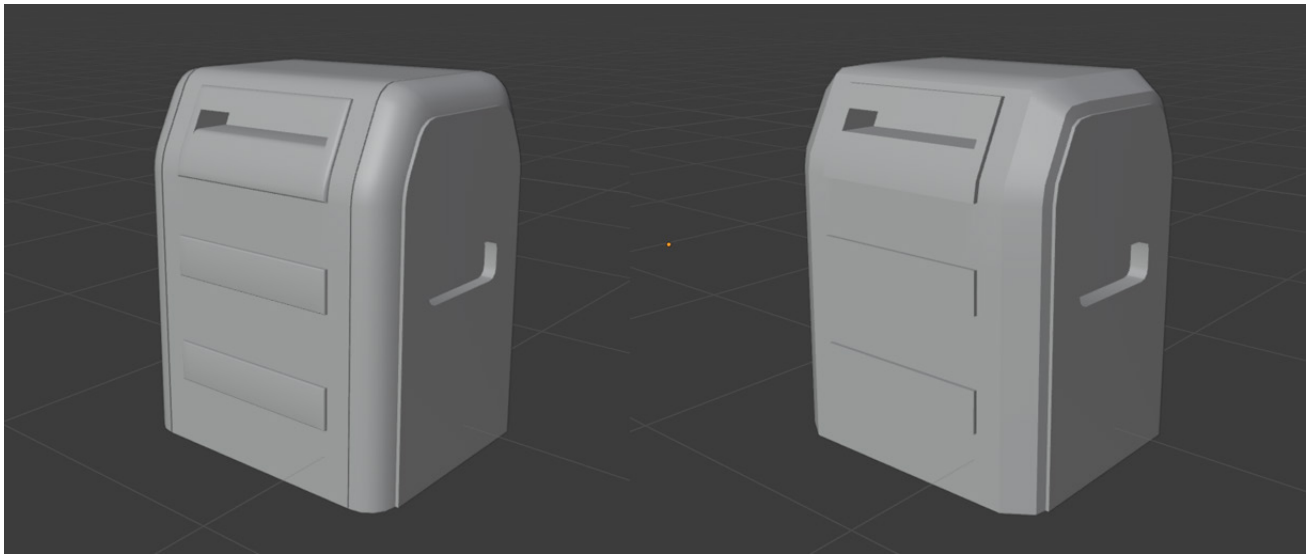
Fig 10. Low poly model at right (356 faces) obtained from the high poly one at left (1.696 faces)

To get the intended output, it will be necessary to have the *Normal map* projected from the high poly version to the low poly one to make it look like it has more detail than it really has. For this, an image will be created in the material of the latter to store the *Normal map*. This image has to be set to *Non-Color*.

Fig 11. Selected to Active menu

After the rendering of the *Normal map* has finished, It will be needed to check if everything has gone as expected, since occasionally some artifacts can appear if there is a lot of difference between one mesh and the other. If this is the case, it is possible to make some tweaks to the available options (Fig 11). These options are: changing the amount of extrusion applied to the model, the maximum ray distance or creating manually a cage, which works as a bounding box that limits the rays. This is usually made by using the low poly mesh and scaling it up along the normals. If a high poly mesh has not been used, all these steps can be disregarded.

Other maps needed before exporting the model to **Quixel Mixer** are the *Curvature, Ambient Occlusion* and *ID maps*[17]. The *Curvature map* highlights the edges of the models, and this can be really useful for materials like metals, as usually, in real life, metallic objects are more damaged along the edges. For the *Smart Materials* this map is also essential, as otherwise it would not work[18]. The *Ambient Occlusion map* adds shadows that are self generated, like the ones that can be seen in objects with cavities. Finally, the *ID map* creates a texture with the different colors assigned to each face of the model[19].

Getting the *Ambient Occlusion map* is simple: after creating a new image in the same material, **Blender** takes care of rendering the *Ambient Occlusion* in the new image after changing the option to the aforementioned. It is also possible to bake it from a high poly mesh as explained before, and change the number of samples if the quality of the render was not enough[17]. Fig 12 shows how the model of the garage looks with Ambient Occlusion and the resulting map.

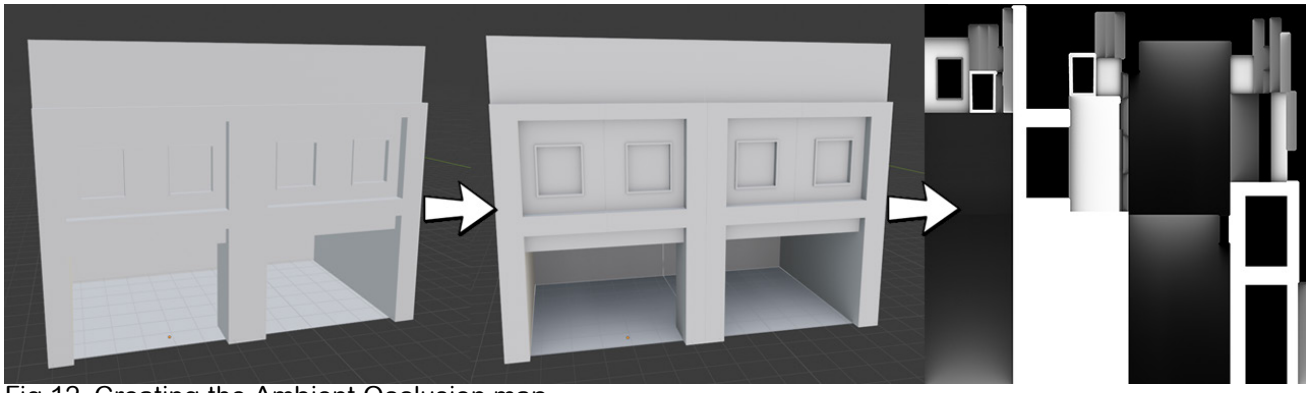

Fig 12. Creating the Ambient Occlusion map

For the *Curvature map*, some extra steps will be needed. First of all, as always, a new image will be created. After that, a *Geometry* and *ColorRamp node* will be added to the material. The *Pointiness* attribute will be connected to the *ColorRamp*, and this one to the *Emission* of the material, as it can be seen in Fig 13. The goal is to adjust the parameters of the *ColorRamp* to reach a point where the model (seen from a rendered preview) has its edges clearly defined. When this is achieved, it is the moment to render it to the image, this time using the *Emit* option from the rendering menu[17].

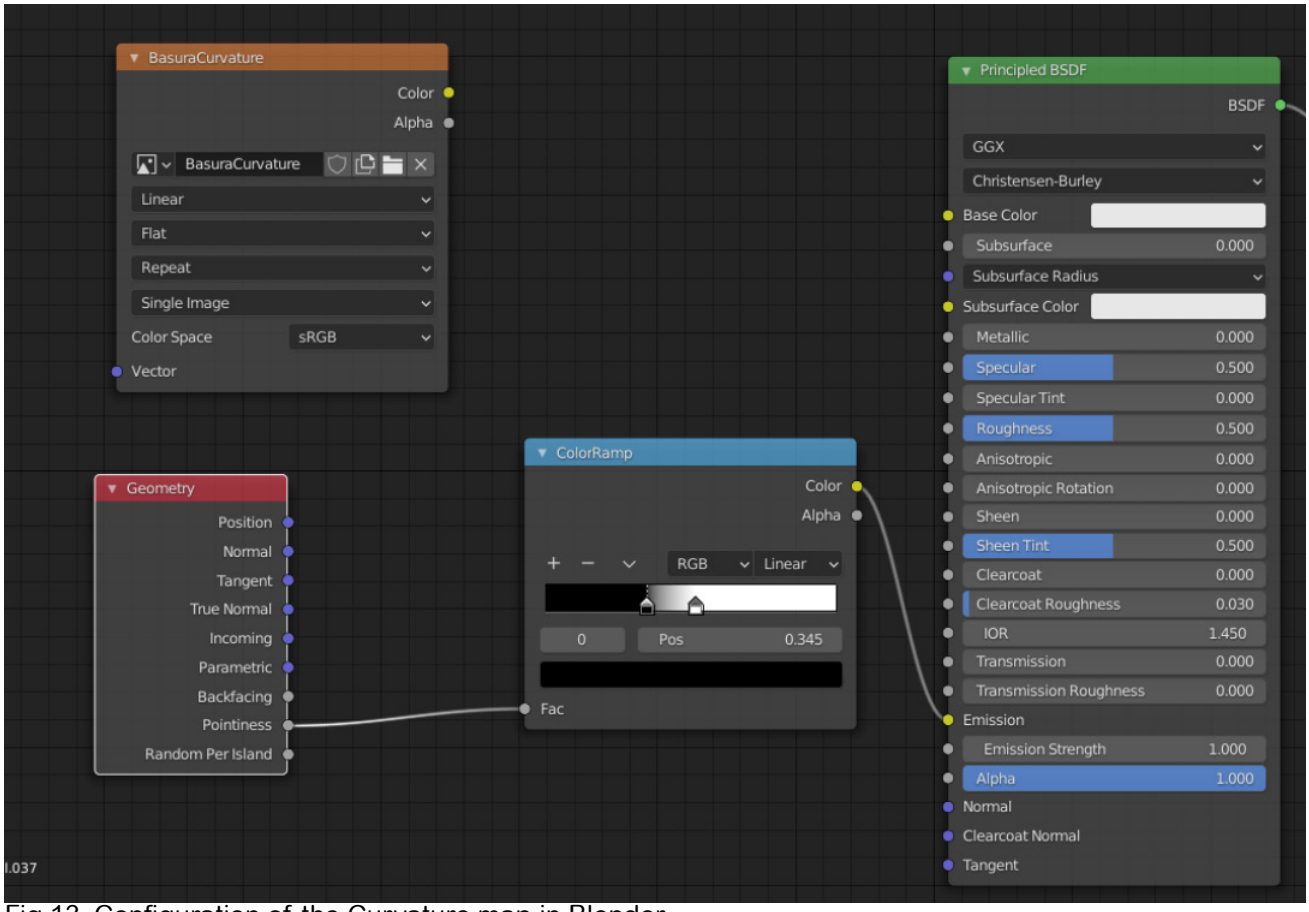

Fig 13. Configuration of the Curvature map in Blender

The last step is to create the *ID map*. This will be used to texture the model easier in **Quixel Mixer,** as explained later. It is desirable to have only one material for the model, therefore it is better to make a copy of the mesh, as several materials will be needed to render the map. To do so, one material will be necessary for each color (and thus, each texture), and then assign the corresponding materials to the faces of the mesh. After doing this, another image texture will be created to bake in it the diffuse color, excluding the direct and indirect lightning contribution[17]. In Fig. 14, all the different materials and how they have been used for the garage can be seen along with the resulting *ID map*.

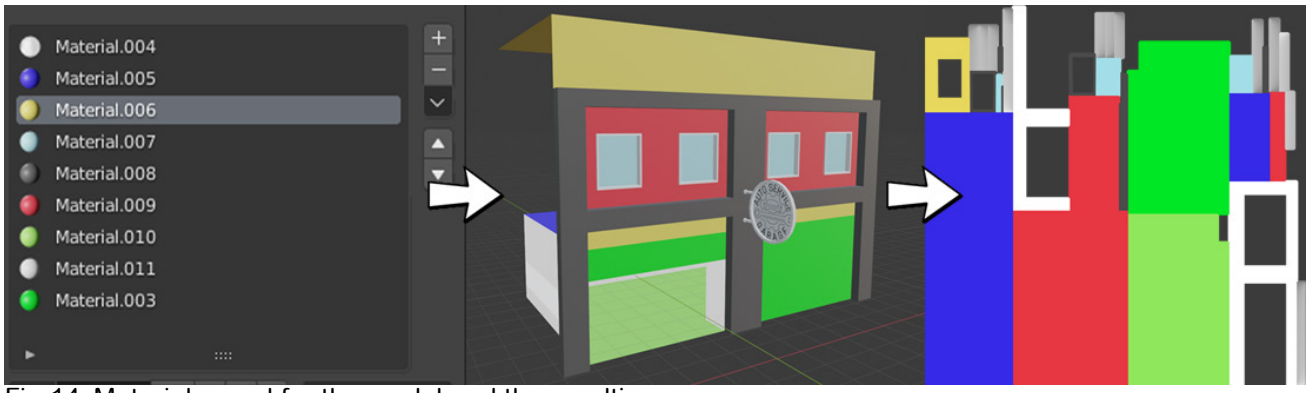

Fig 14. Materials used for the model and the resulting map

All this process has been repeated for most of the meshes. In Fig 15, some of the models made usign the general workflow can be seen. Most of them have been buildings, although some other objects have been made using this workflow as shown in the image.

Fig 15. Some models made using the general workflow

After all these steps are completed and the mesh with a single material is exported as an .fbx or .obj file, everything is ready to begin with the texturing using **Quixel Mixer.**

4.1.1.2 Texturing in Quixel Mixer

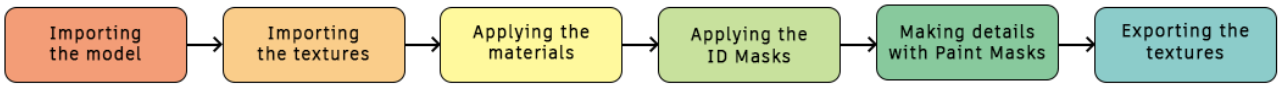

Fig 16. Texturing in Quixel Mixer diagram

Quixel Mixer is a program used for texturing that uses what they call *Megascans*, which is the result of a large group of their employees that travel the world and photograph different textures they encounter along with photogrammetry of certain objects. The textures available are what they have photographed converted to a seamless image, which gives it a realistic appearance, as it is taken directly from reality[20]. The downside of these materials is that, as they are photographs and not made procedurally, it is harder to modify them to adapt them to the models. There are also *Smart Materials*, which are customizable and work in the same way as in other programs like **Substance Designer.**

The steps that will be followed can be seen in Fig 16. The first step after opening the program, is to import the mesh. This is made by going to the *Setup menu* and choosing *Custom Model* in the *Type* section. After choosing the location of the model, the next step is to import all the maps created in **Blender** for this model by going into *Window* > *Texture Sets Editor* or pressing Control + Shift + T. It is possible to import extra maps if needed like the *Albedo, Metalness,* etc. When finished with this, the texturing can begin.

Before starting with the texturing it has to be clear that there isn't any error in the mesh, this being flipped normals for example, which makes the model in **Quixel** to look different, as the faces are only shown in the direction they are pointing to. If this or any other mistake has been made, it is easily fixable. It is simply modified in Blender and then exported as the same name as before. After that, just pressing the *Reimport Model* in the *Setup menu* fixes the problem.

The best practice is to organize everything in folders, one for every part that will be different in the model. Inside the folder can be stored the different materials that will compose that part. This is where the *ID map* takes part. Instead of having to manually paint and create masks by painting the texture itself, it is possible to just assign each folder or individual material to a color. This color corresponds to the color chosen when creating the *ID map*. Moreover, additional masks can be added on top of the color mask, to combine different materials or make them more interesting[18]. One of the buildings already textured can be seen in Fig 17, and its materials organized in folders along with the *ID masks*, in Fig 18.

Fig 17. Textured building

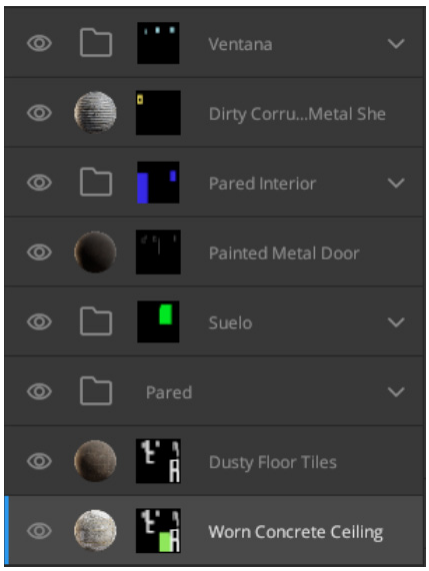

Fig 18. Materials with their masks

Each material can be modified to adapt to the desired results, changing the rotation, scale or type of projection. On top of that, each of the maps composing the material can be modified, changing its opacity, type of fusion, contrast, and even the option to adapt to the material below it, so that it can blend better with the look up until that point. For the *Albedo* it is possible to also change its "tint", although depending on the material, as it is based on photographs, it may result in an undesirable look, making it difficult to use some complex materials unless looking for that specific appearance.

After the texturing has been finished, all that is left to do is export the maps. It is possible to select which ones are needed to export and their resolution. In this case all the displacement maps have been discarded as the models are low poly and it would not make any difference.

4.1.1.3 Importing to Unreal Engine

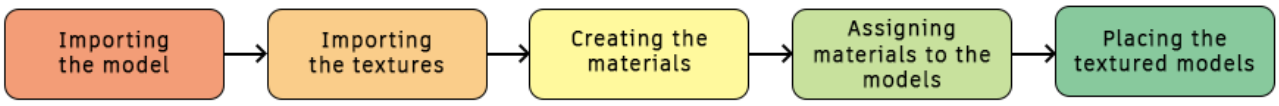

Fig 19. Importing to Unreal diagram

Now having the model as well as the texture it is the moment to import them to the game engine. Following the diagram in Fig 19, first it is desirable to import the model, as it will be imported along with the material created for it in **Blender.** After that, the textures will be imported to the engine so that the material can be modified to look as expected. Preparing the material is as easy as double-clicking it to open the material interface and dropping the textures to the *Material Graph.* Afterwards, it is necessary to link each map into their corresponding node in the material output (Fig 20). Finished this, the only thing left is applying and saving the material.

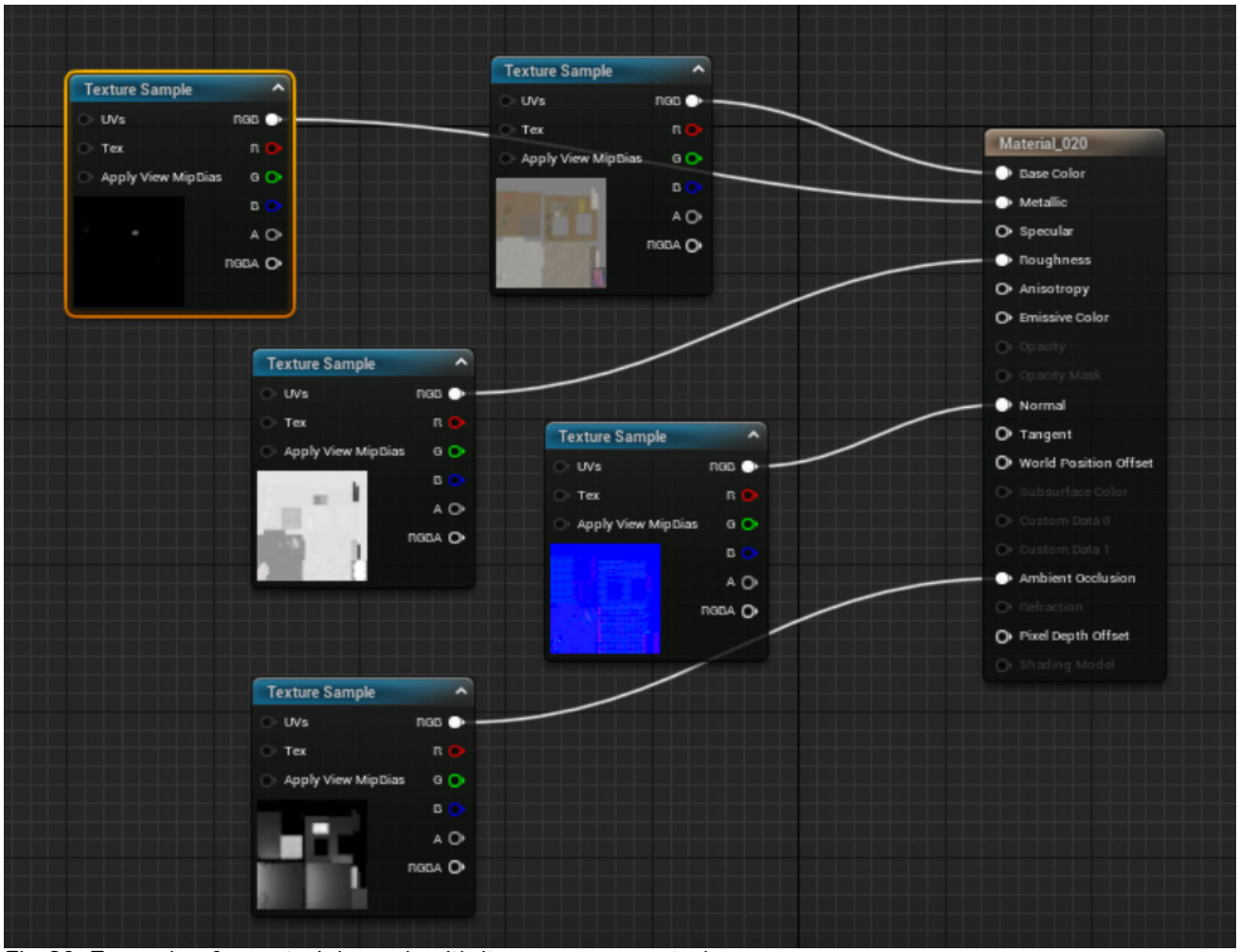

Fig 20. Example of a material graph with its maps connected

Now it is possible to drop the fully textured model into the scene. As the illumination along with other factors can make the model look different from what has been seen up until now, it is the right moment to see if the model inside the scene looks as expected, comparing it to the references. If not, it being by the mesh itself or its textures, the corresponding adjustments can be made using **Blender, Quixel** or even some other image editing software like **Krita** to make any changes necessary to the textures. As **Quixel** does not have an emission channel, every *Emission map* has been made in **Krita** while checking in **Unreal** that everything is correct.

After verifying that everything is as expected, the placement of the model can finally begin. Based on the references or the blocking made in **Blender**, it is just a matter of using the *Gizmo* to move the mesh to its corresponding position. This procedure is shown in Fig 21, where the design and positioning of the models are based on one photograph[14].

Fig 21. Reference compared to final result in Unreal

4.1.2 Modeling From Texture

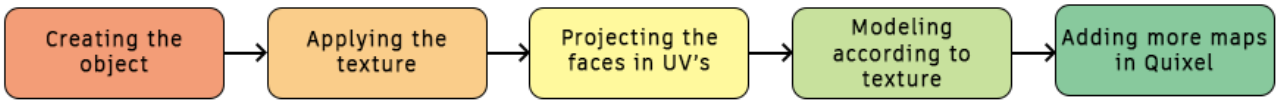

Fig 22. Modeling from texture diagram

An alternative form of modeling can be possible instead of the usual one, which commonly follows the next structure: *Modeling* > *Unwrapping* > *Texturing.* This one is inspired by the widely known artist Ian Hubert, which also has a series of quick tutorials in his YouTube channel[21]. The method he follows is diagrammed in Fig 22. He starts with a texture, usually a photograph, and then adapts the form of the mesh to the one in the texture[22]. This method makes it hard to make more complex models, but it allows to make simple ones much quicker and also make it look realistic, as it is taken directly from the real world.

The first step is to choose the texture from which the mesh will be modeled from. To illustrate it, the procedure for the modeling of one of the expending machines will be shown.

After having the texture stored in the computer, a basic shape along with a material have to be created in **Blender**, in this case a cube. Within the *Shading interface*, there must be created an *Image Texture node*, loaded with the texture chosen and linked to the *Base color output*.

Now, from the *UV editing tab*, the mesh must be modified to have a shape according to the texture. First, the object must be scaled to resemble its figure approximately. Then, each one of the faces must be unwrapped by selecting it and, from an orthogonal perspective, and looking directly at the camera, choosing the *Project From View* option. As there is only one face in the texture, the rest will be projected on the parts where there is nothing on it, like the gray part at the bottom. In this case those faces will not be seen because of the placement of the object, but it is advisable to have different textures for each face. This is easier if the textures had been photographed personally, but it also works with free images on the internet.

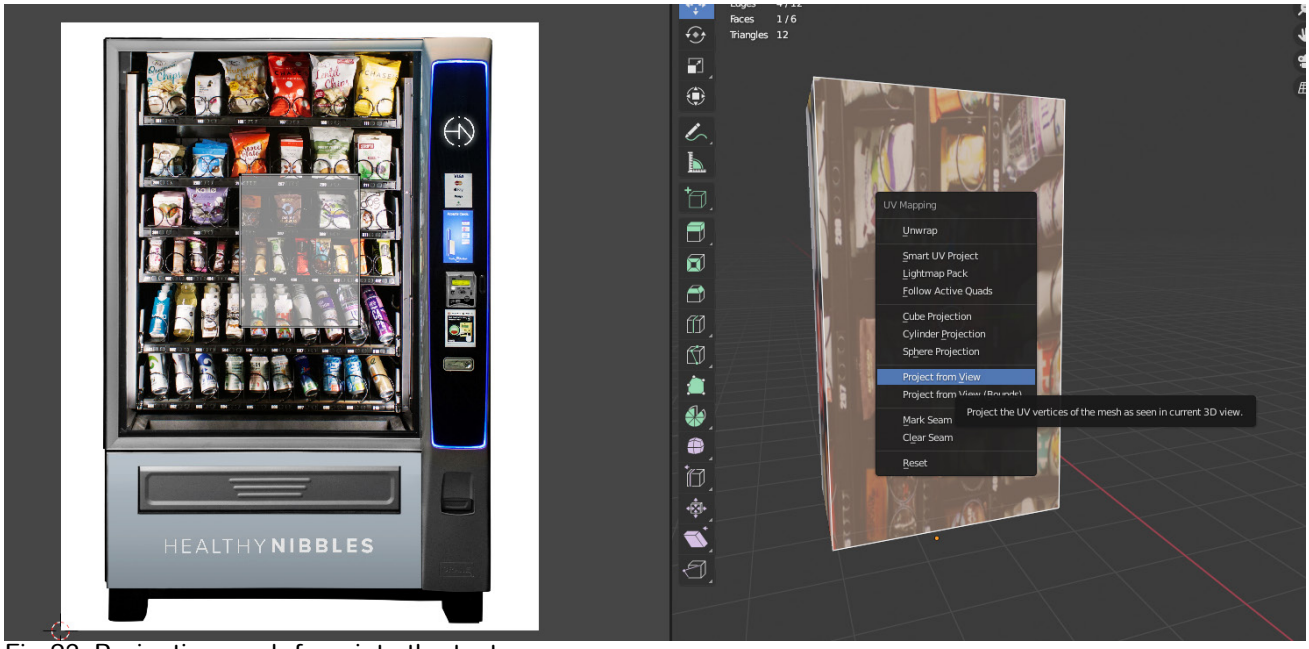

Fig 23. Projecting each face into the texture

After having every face projected (Fig 23), the dimensions of the shape can be adjusted to make sure that there is no stretching in the texture as seen in Fig 24.

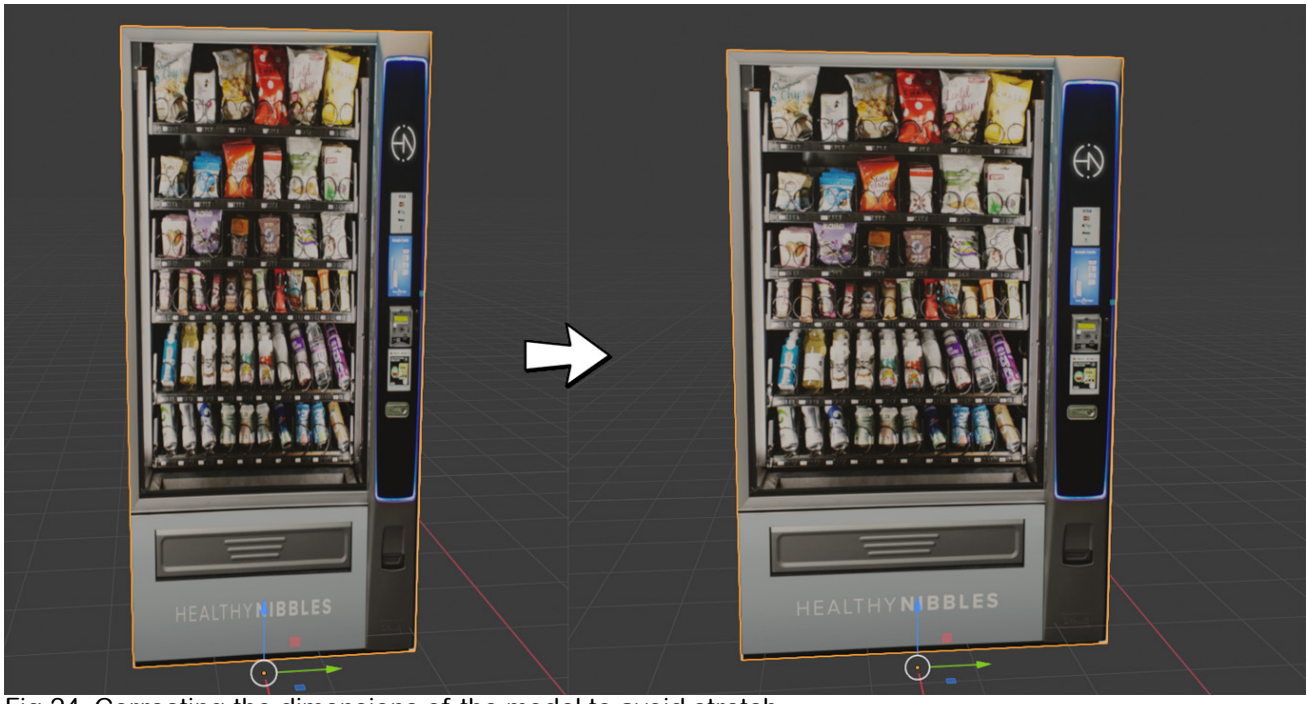

Fig 24. Correcting the dimensions of the model to avoid stretch

When everything is well proportioned, the modeling can begin. This is so the mesh is not just a simple plane with a texture projected, but a 3D model that resembles the original object. To do so, several loops and extrusions will be made along the cube, making sure that the texture is not distorted by moving the vertices along the edges (pressing G two times in **Blender**). In case that some part has a distorted image, either because of moving a vertex incorrectly or by doing an extrusion with a straight angle, it can be reprojected into the texture.

After having the desired mesh, there are still some steps that need to be done. So far there is just one map, the *Albedo*, whose only function is to show the color, but it is also important to have other maps like the *Metallic* or *Roughness map*. To do so, the same steps needed to render the *ID map* will be repeated, but this time with just the main material. This will have as a result an *Albedo map*, which will be imported to **Quixel Mixer** along with the model.

Having our model in **Quixel Mixer**, a basic material can be created to add the *Albedo map* and the rest of the maps can be disabled. To add the *Roughness, Metallic* and even *Normal maps,* the best idea is to choose one material that is similar to the one in the reference and, using a paint mask, paint over it, and then disable its color, so that only the original color is shown (Fig 25). Using this method the material has a more realistic look, as the light interacts with it in an accurate way.

Fig 25. Adding additional maps in Quixel to obtain more details

When the texturing is finished, the final results can be seen importing the mesh and textures to **Unreal** as usual. Fig 26 shows the process of the vending machine from the model to the textured model and finally how it is seen in the engine.

Fig 26. Model from texture (left), textured model (center) and final result (right)

4.1.3 Photogrammetry

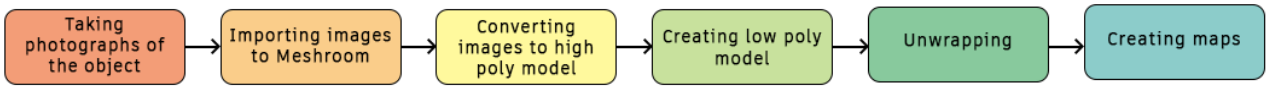

Fig 27. Photogrammetry diagram

Photogrammetry is the science of obtaining reliable information about the properties of surfaces and objects without physical contact with the objects, and of measuring and interpreting this information. The input is characterized by obtaining reliable information through processes of recording patterns of electromagnetic radiant energy, predominantly in the form of photographic images[23]. Comparing different points in the images and their distances, it is possible to extract a 3D mesh along with its texture. The process that will be followed can be seen in Fig 27.

The software used for this project is **Meshroom**, which is a completely free photogrammetry software, along with **Instant Meshes**, a software used to retopologize that is also free. As the model obtained from **Meshroom** has a really high amount of polygons, it will be necessary to decimate it.

For this example, 24 photographs have been taken with a high resolution camera. The object of the photograph was some kind of electrical or ventilation box seen in Valencia. Having a drone would have greatly increased the possibilities of scanning, but being limited to a hand-held camera, only small enough objects that allowed to make photographs from above have been possible to scan.

Once **Meshroom** is open, all the photographs must be imported into the *Images tab*. After that, in the graph editor below it can be seen that there are a lot of nodes. They can be modified, but usually all of them can be left as they are except the last three: *Meshing, MeshFiltering* and *Texturing (Fig 28)*.

Fig 28. The three most important nodes to modify

It is better to see the results with the default options first and then try to modify them if the result is not satisfactory. Once the images have been imported, the *Start button* can be pressed. The process can take a while, especially the *FeatureExtraction* one.

First, all the points are extracted and all the positions from where the photographs have been taken are shown. In Fig 29, all the points extracted along with the cameras from the different positions can be seen

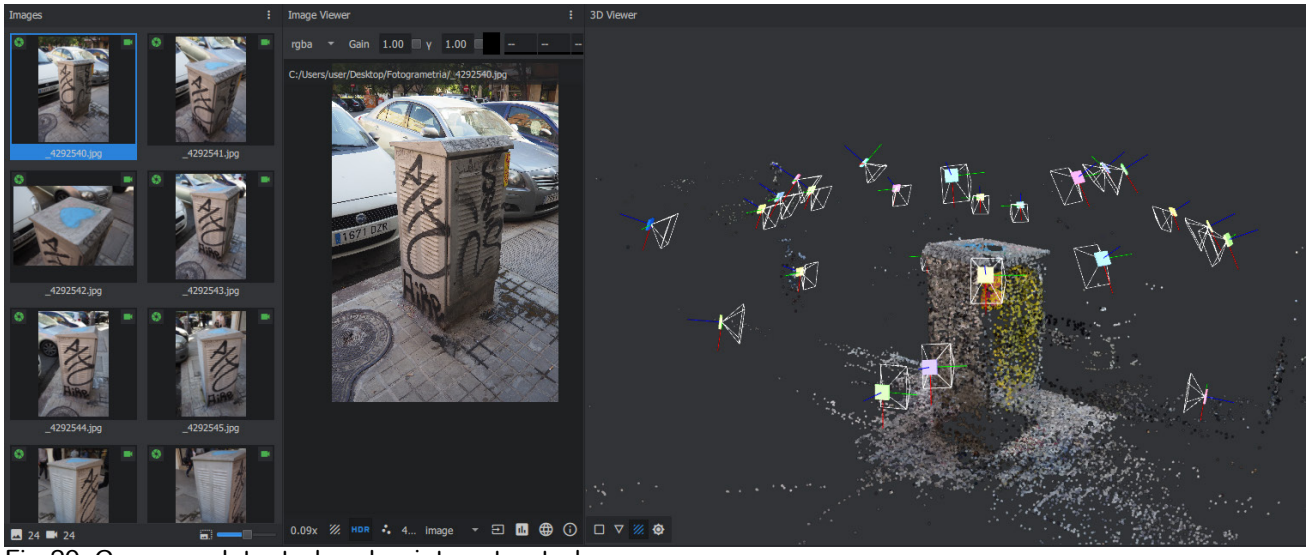

Fig 29. Cameras detected and points extracted

Based on those points, the mesh is generated. Some adjustments had to be made in the *Meshing* and *MeshFiltering nodes* to try to get the best result in this case. The final result can be seen in Fig 30.

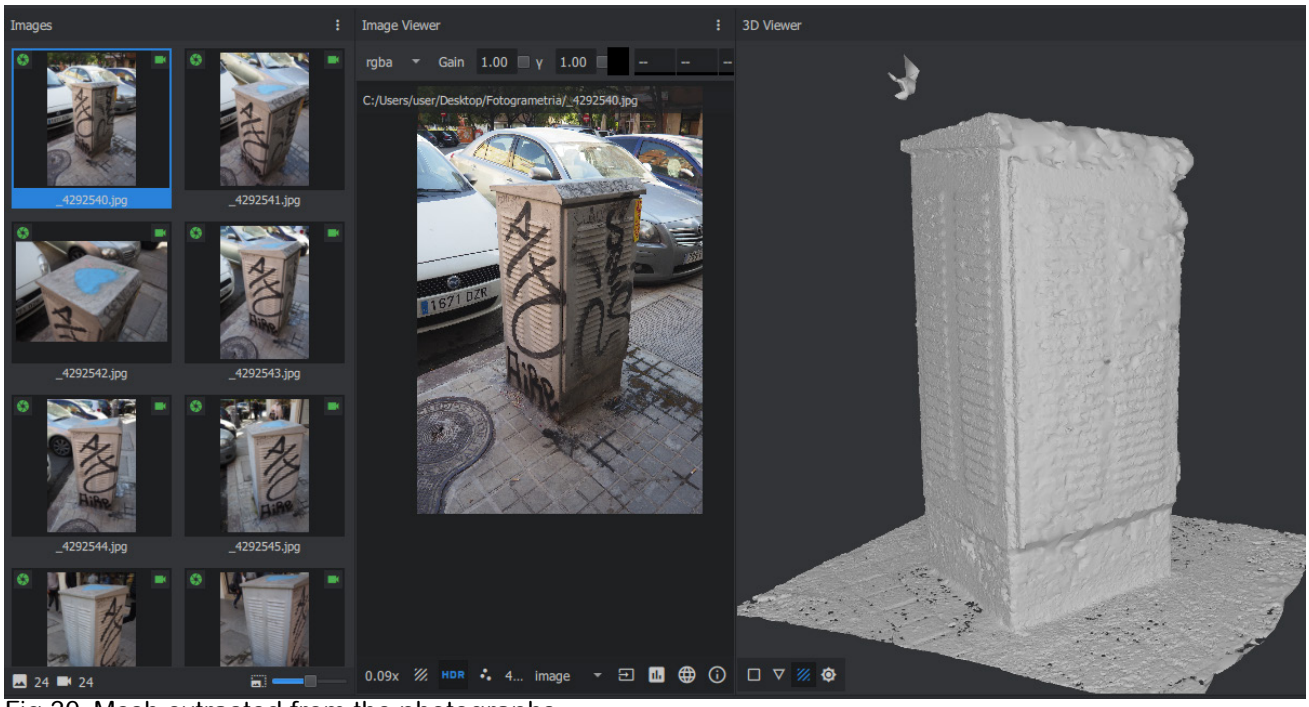

Fig 30. Mesh extracted from the photographs

And finally, the texturing process takes place. In Fig 31 the mesh can be seen already textured based on the photographs.

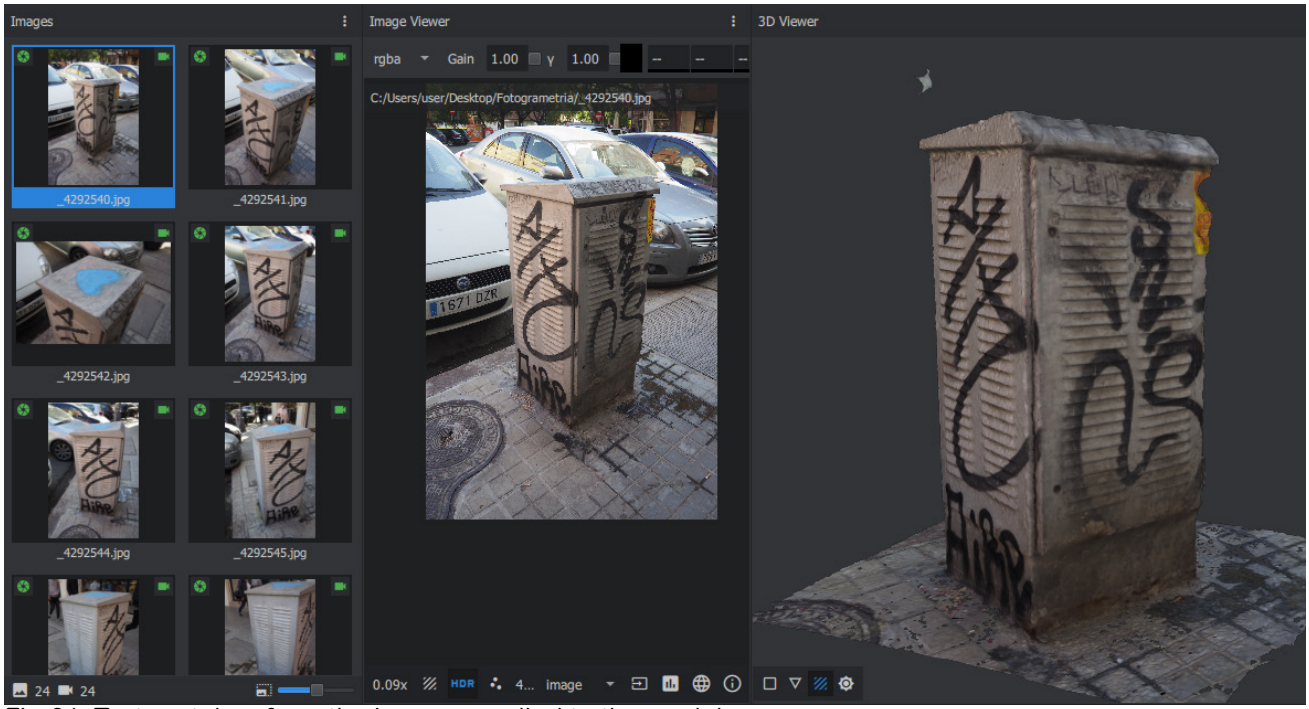

Fig 31. Texture taken from the images applied to the model

A lot of tweaking was made especially in the *Meshing node*, and a custom bounding box was made, but it can be seen that 24 photographs were not enough to get a satisfactory result. However, as the result was not excessively bad, and as the illumination is important in this case and it has to be the same, instead of taking more photographs, the decision made was to just fix the mesh in **Blender**.

After the process has been completed, double clicking the *Texturing node* opens the folder with the textured mesh. This mesh is then exported to **Blender** to give it the right orientation.

In **Blender,** the mesh must be rotated to give it the desired orientation and then it has to be exported again. This will be the high poly mesh. The reason as to why it is rotated first is because now a second mesh is needed to be on top of the first one, and it has to be in the same position and orientation[24], like explained in the general workflow.

Now, using **Instant Meshes**, the high poly mesh is imported into the program. It allows to choose a desired amount of vertices so, in this case, as it is for a videogame, around 2k is a good amount[25].

After that, the orientation field must be solved by pressing the *Solve button* (Fig 32). Normally it is recommended to use the *Orientation Comb* tool to make adjustments in the orientation of the field, but this being a fairly simple mesh, it is not necessary.

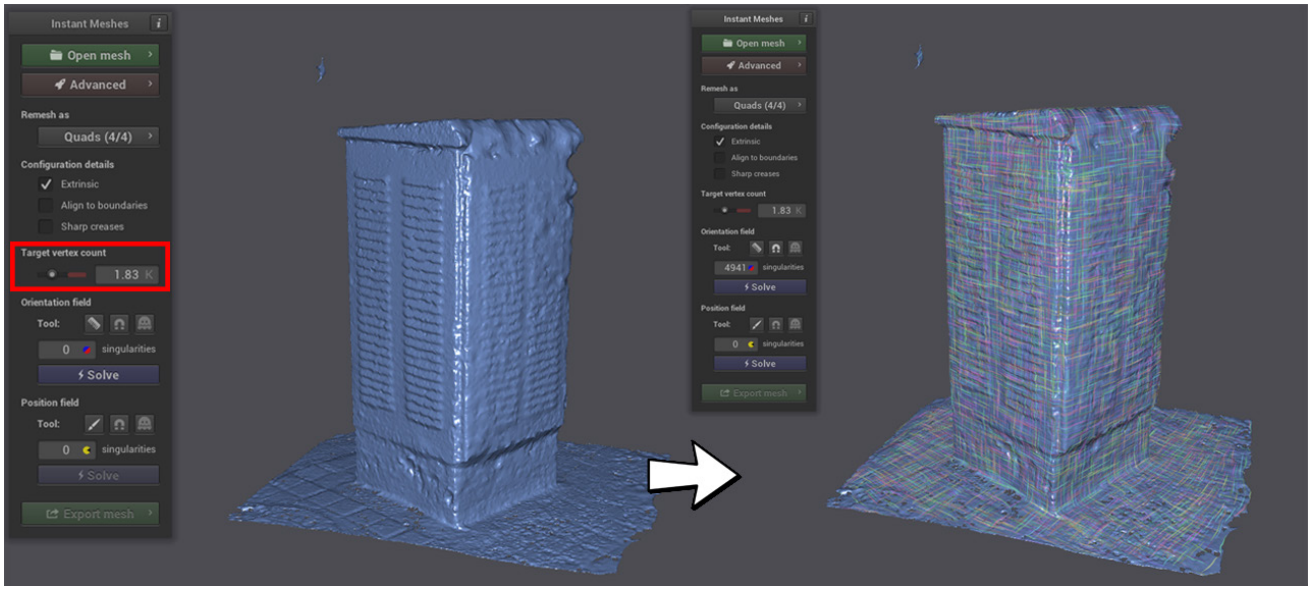

Fig 32. Orientation field solved

Finally, the position field is ready to be solved. After solving it, adjustments can be made again using the *Edge Brush tool*. If everything is correct, the mesh can be exported. For this model the option for *Pure Quad Mesh* has been checked and the *Smoothing Iterations* have been set to 1, as it can be seen in Fig 33.

Once both the high and low poly meshes are available, it is time to return to **Blender**. The low poly mesh must be imported while the high poly mesh is selected, this way both of them will be in the same place and orientation. Now some adjustments can be made. With this model it was necessary to use the sculpting tools with the high poly mesh and to move some vertices in the low poly one to fix some of the artifacts.

When everything is correct, the low poly mesh has to be *UV mapped*, and then the maps are rendered as usual, following the same steps as the general workflow, projecting from the high to the low poly. This is necessary because the maps generated by **Meshroom** are bits of polygons scattered throughout the texture, so it would be impossible to reuse those textures for the low poly one. After that, the mesh and textures can be imported in **Unreal,** and when the material is created, the model is ready to be used.

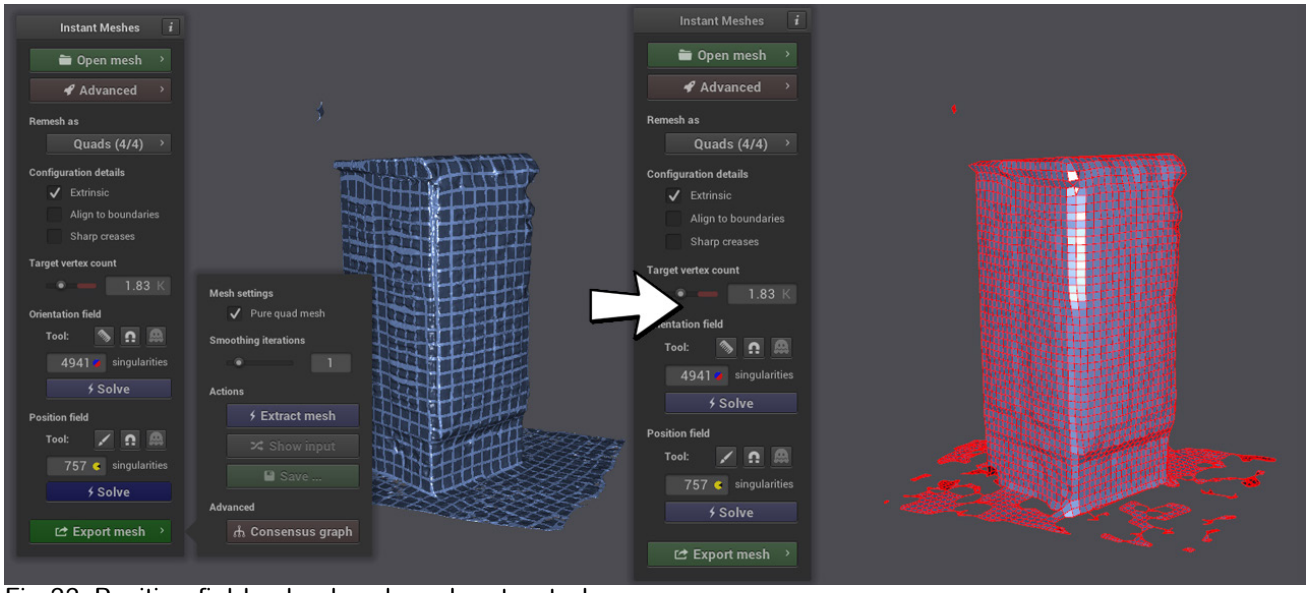

Fig 33. Position field solved and mesh extracted

Fig 34 shows the process and the final low poly result inside the **Unreal** engine.

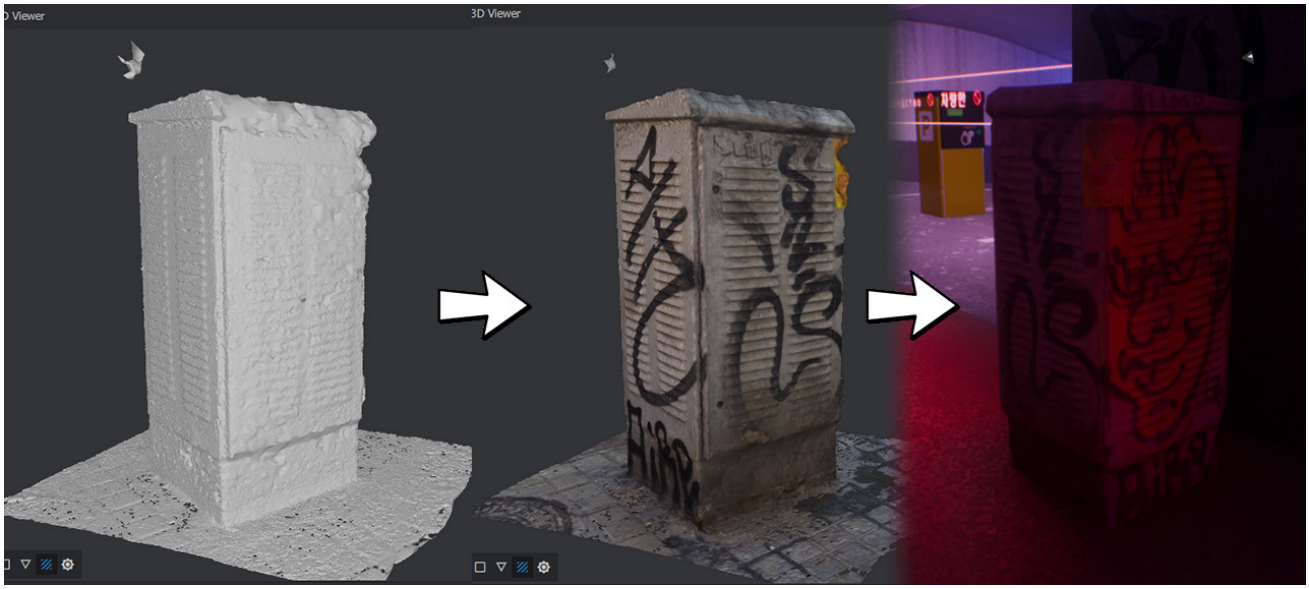

Fig 34. From extracted mesh (left) to textured mesh (center) to final low poly result (right)

4.1.4 Neon Workflow

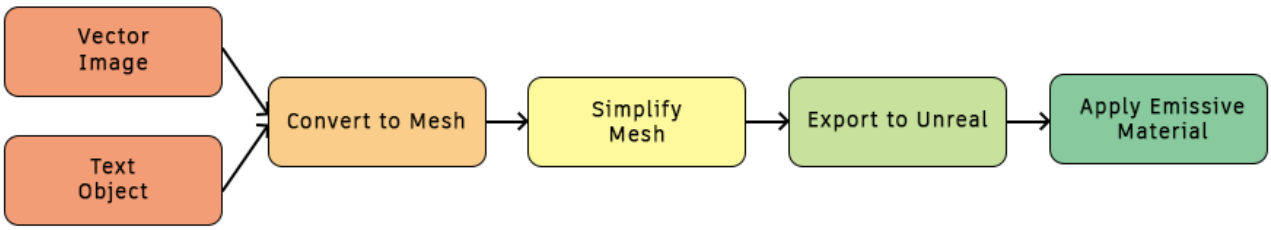

Fig 35. Neon workflow diagram

The Cyberpunk aesthetic is well known for having a lot of neons, so this project would not be different. To create the neons, a different approach has been given. There are also two types of neons, the ones that have text, and the ones that represent a figure, both being fairly similar, as it can be seen in the diagram in Fig 35.

For the ones with text, it is possible to start either with **Blender**[26] or a software that allows converting text into a .svg file and then importing it into **Blender**. As the first method is simpler, it will be the one used. To do so, a *Text object* will be created. It can then be edited to show whatever text is needed and to choose which *Font* will be used, selecting it from the fonts folder. Once everything is ready, some adjustments have to be made before exporting it.

First of all, the *Fill mode* must be changed to *None*, so that only the edges are visible. Then, a bevel is added with the *Depth* that better suits it, but always with a resolution of 0, as more polygons will not make any difference in the final result. In addition to that, the *Resolution U* has to be reduced to a point where the mesh still holds its shape but does not have that many polygons. Finally, the text can be converted to a *Mesh object*. Depending on the use of the model, the vertices in the back of the mesh can also be deleted if they are not going to be seen, thus reducing the number of faces. Fig 36 shows how the *Text object* is transformed into a *Mesh object* able to be used.

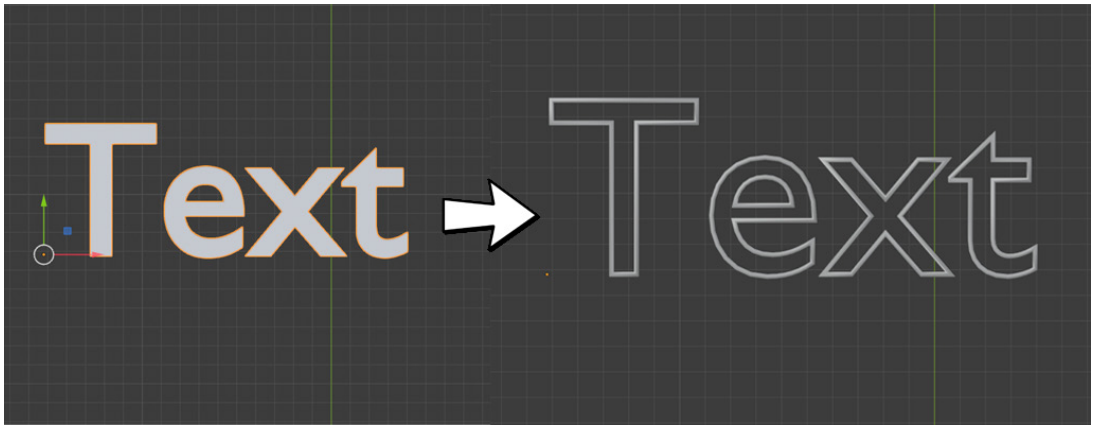

Fig 36. Converting a text object to a mesh object

As the neon is only going to have emission, it is not necessary to unwrap the mesh, so it can be exported as it is. Once in the engine, the mesh is imported into its corresponding folder and then a material is created. This material only needs one *Color node* with the desired color that is linked to the *Base Color* and then to a *Multiply node*. The latter is to increase the intensity of the light, meaning that the higher the value, the higher the brightness, and it will be linked to the *Emissive Color.* The final step is to save the material and everything will be ready. One of the materials and its resulting looks can be seen in Fig 37.

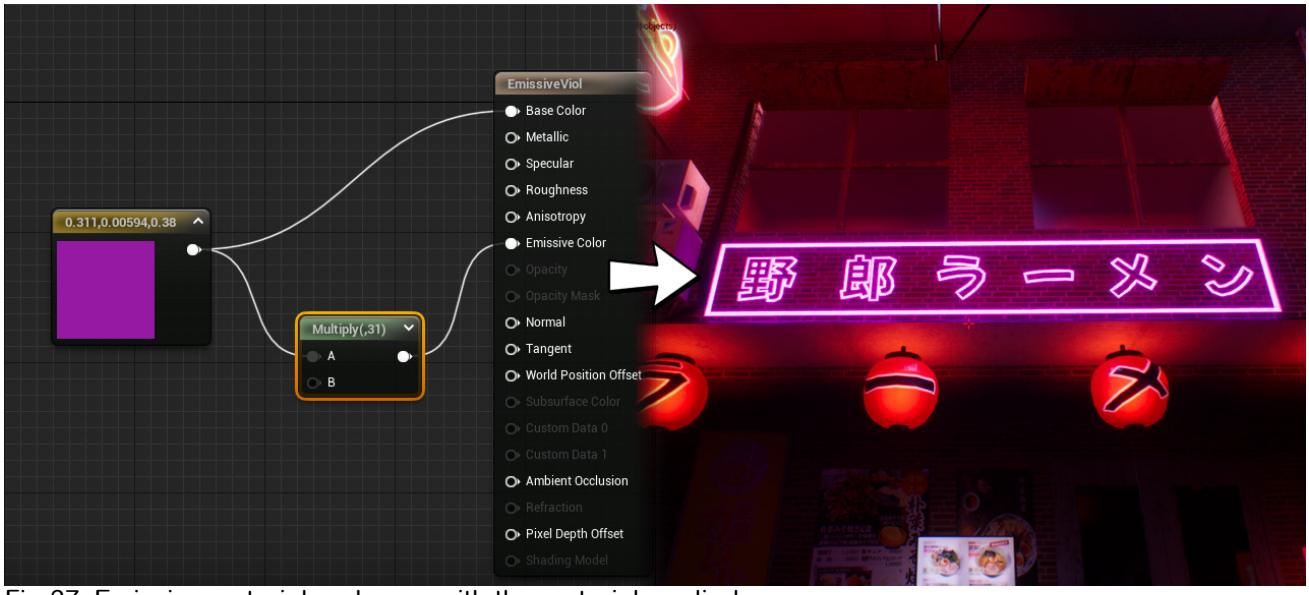

Fig 37. Emissive material and neon with the material applied

Fig 38. Neon made with vectors

If the neon is not a text, the process is very similar, but instead of using a *Text object* in **Blender**, it is possible to use a *Curve object*[26] or import an .svg file into the project. After that, in the curve options, the *3D* one in the *Shape section* has to be selected, and then the procedure is exactly the same as with the text.

With this, it is possible to create different signs with custom shapes, such as the neon sign made for the garage shown in Fig 38.

4.1.5 Background Buildings

To give the impression that the city is not empty, it will be necessary to fill the gaps shown above the shorter buildings, but making enough buildings with their own texture just to fill those gaps would take a lot of time that could be used in other aspects. Inspired by a YouTube video by Ian Hubert[27], a similar approach has been taken.

To get a clutter of buildings, all there will be needed is an image and a bunch of cubes. First, a cube is created along with a material and a *TextureImage node* attached to the *BaseColor*. Then, the *TextureImage node* is loaded with a high quality photograph of some city at night that can be found easily on the internet. Finally, after creating several cubes with different sizes and orientations, their *UV*'s are projected from view while in the *Orthogonal mode*. Following the shapes of the photograph, it is only necessary to move each *UV island* into the building that fits it better.

Then, the buildings are imported into **Unreal Engine** along with the photograph, and the material will only have the image connected to the *Albedo* and *Emission inputs*, as well as a parameter set to 0 for the roughness to simulate the windows of the buildings.

In Fig 39 the buildings can be seen in the distance on the left looking like they are real buildings whereas on the right, it can be seen how it really looks.

Fig 39. How the buildings are seen versus what they look like

4.2 UNREAL ENGINE

In this section all the different aspects that have taken place inside the engine, and that are beyond the modeling, will be described. This can be the editing of materials, creation of particles, among others.

4.2.1 Raining Effect

Fig 40. Niagara System configuration for the rain

To make the game look like it is raining, a *Niagara System object* with the *Fountain template* is created[28]. This works like a particle system: it spawns sprites randomly following certain conditions.

To get it to look like it is raining, it can not be left with the default options, so some adjustments will have to be made.

First, the *Add Velocity in Cone* and *Sphere Location properties* are deleted. Then, a *Box Location* will be added, which will be adjusted to cover all the visible parts of the map with its volume.

After that, the particles will need velocity, so the property *Add Velocity* will be added, changing its values to make it fall more rapidly.

To make it look more like rain, the particles' shape have to be changed into an elongated form, so it will be necessary to modify the *Sprite Size* in the *Initialize Particle* section and to change the form to *Non-Uniform*. The angle has to follow the direction of the particle too, and that option will be found in *Sprite Renderer.*

The only thing left is to change the *Opacity* in the *Color Mode* and the *Spawn Rate* to look like desired. Fig 40 shows the looks of the final *Niagara System* configured.

4.2.2 Creating Wet Materials

To create the wet materials, two materials have been combined with a parameter that determines the "strength" of one or the other, being this parameter the *Red channel* of the *Vertex Colors*, which will be modified using the *Vertex Paint* function[29].

First of all, the desired textures are added into the material graph. For this example, the texture for the asphalt has been taken from **Quixel Bridge**.

After that, a *Lerp node* has to be created for every parameter of the textures. For now, only 4, one for the *Base Color*, another one for the *Roughness*, one more for the *Specular* and a last one for the *Metallic output*. The textures will then be linked to the *A input*, whereas on the *B*, a parameter will be created. This parameter will represent the “wet” part, and will be a parameter so that it is easily customizable. Once everything is linked, each *Lerp node* is connected to its corresponding output in the material node.

Now, a *Vertex Color node* is created and its *Red value* is linked to every *Alpha input* of the *Lerp nodes*. This means that depending on the value of the *Red* for each vertex, between 0 and 1, it will determine how much of the second material (the wet one) is shown. In Fig 41, the full graph to obtain the wet asphalt can be seen.

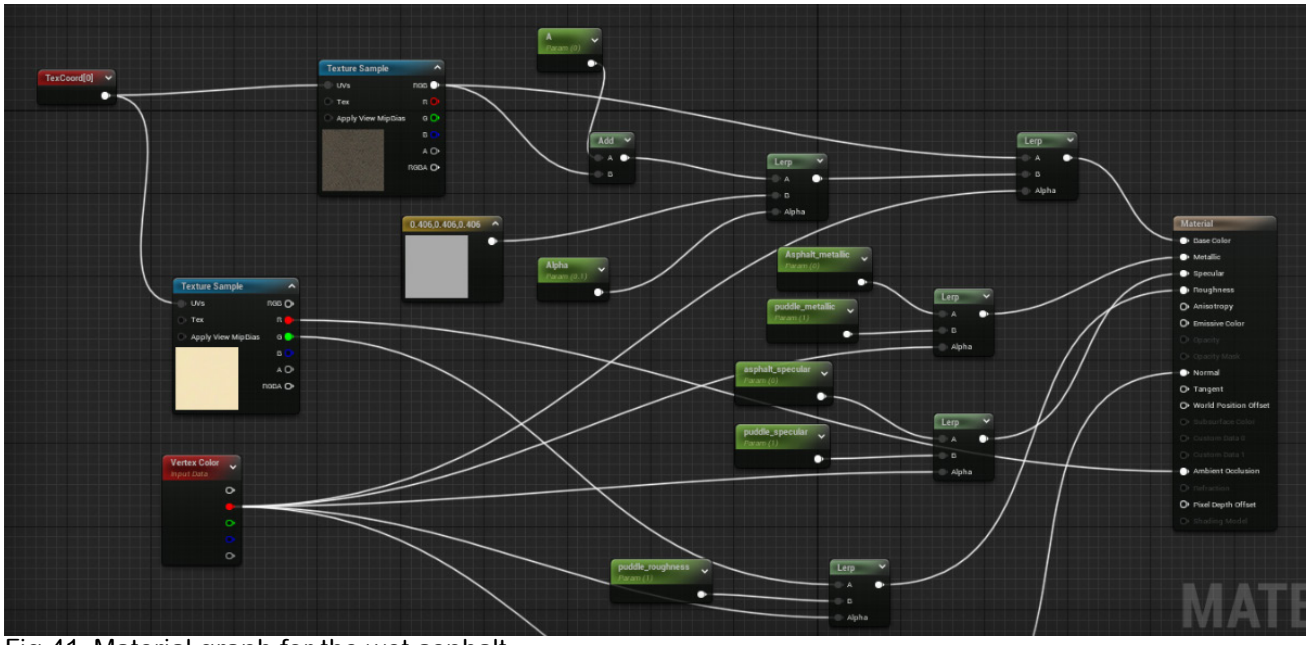

Fig 41. Material graph for the wet asphalt

After this, the material can be saved to test it. As it works with the color of the vertices, a simple plane can not be used, because it would only show any changes at its corners. Instead, a subdivided plane will be created in **Blender** to be exported into the program. After painting a zone using the *Mesh Paint Mode* with red in the plane with the material (Fig 42), the parameters can be adjusted to get the look intended.

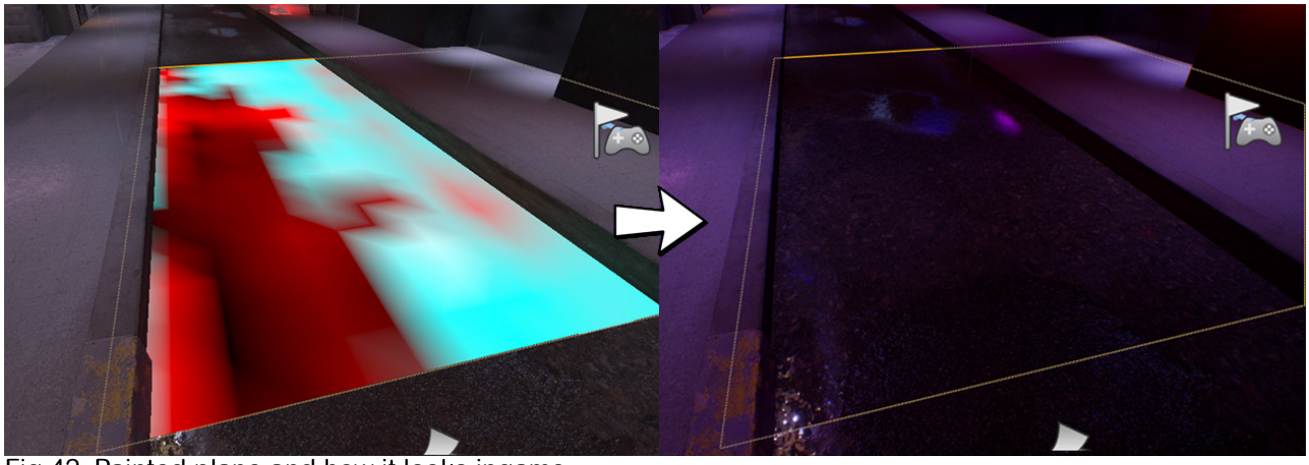
Fig 42. Painted plane and how it looks ingame

The material could be left like that, but as there is rain in the environment it would be a good option to add a bit of movement to the water. To do so, it will be necessary to return to the material graph.

To create the effect of rain hitting the surface it will only be necessary to use the *Normal maps*. First, the *Normal map* is copied, and linked to the *UV input* will be a *Panner*. The *Panner* will receive *Texture Coordinates* that are multiplied by a parameter to make it larger or smaller, depending on the desired effect, and a *Speed parameter*, that will determine how fast the texture is panned. Then, each *Normal map* will be linked to a *FlattenNormal node* with another parameter that will control how "deep" the normal is, and those nodes will be linked to another *Lerp* with the *Red* value linked into the *Alpha input*. Finally, the *Lerp node* is linked to the *Normal output.* Fig 43 shows all the elements used in the material graph to make the water look like it is moving.

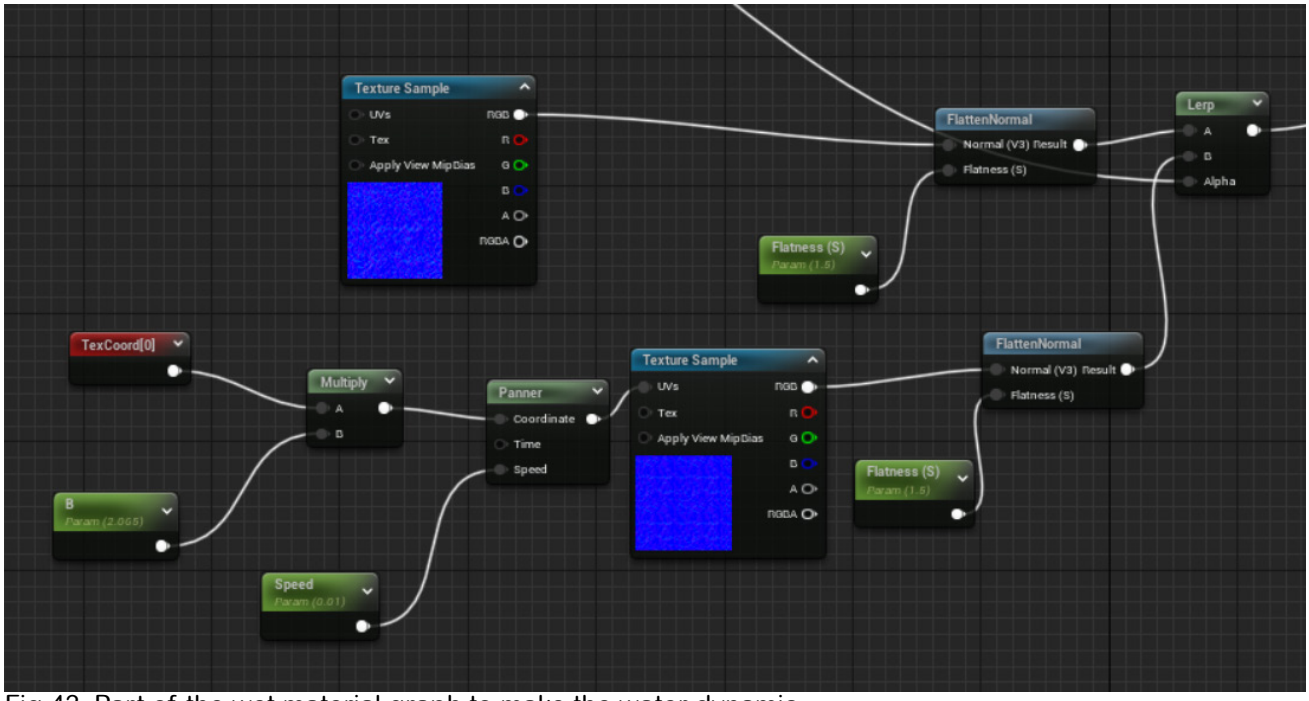

Fig 43. Part of the wet material graph to make the water dynamic

Finally, After editing the parameters, the wet areas will look like the rain is hitting them.

4.2.3 Milky Glass

At some point during the development, there appeared the necessity to have blurred or "milky" glass. This type of glass became useful in conjunction with emissive objects or lights that otherwise would look unappealing.

To obtain this, the *SpiralBlur node* was used in the material graph[30]. The *Base Color* is set to black, the output of *Result* from the *SpiralBlur* node is connected to *Emissive* and the *SceneColor clamp to 0* to the *Opacity*. What this node does is stated in its name: it creates a blur effect in a spiral motion, which is what it is looked for. After creating a window, it will be necessary to make some adjustments depending on the preferred appearance. In this case, modifying the *Distance* and *Radial Steps* attributes was enough, and two different materials were created to use depending on the case. Fig 44 shows the material graph of one of the materials used to obtain a milky glass and how it looks at its right.

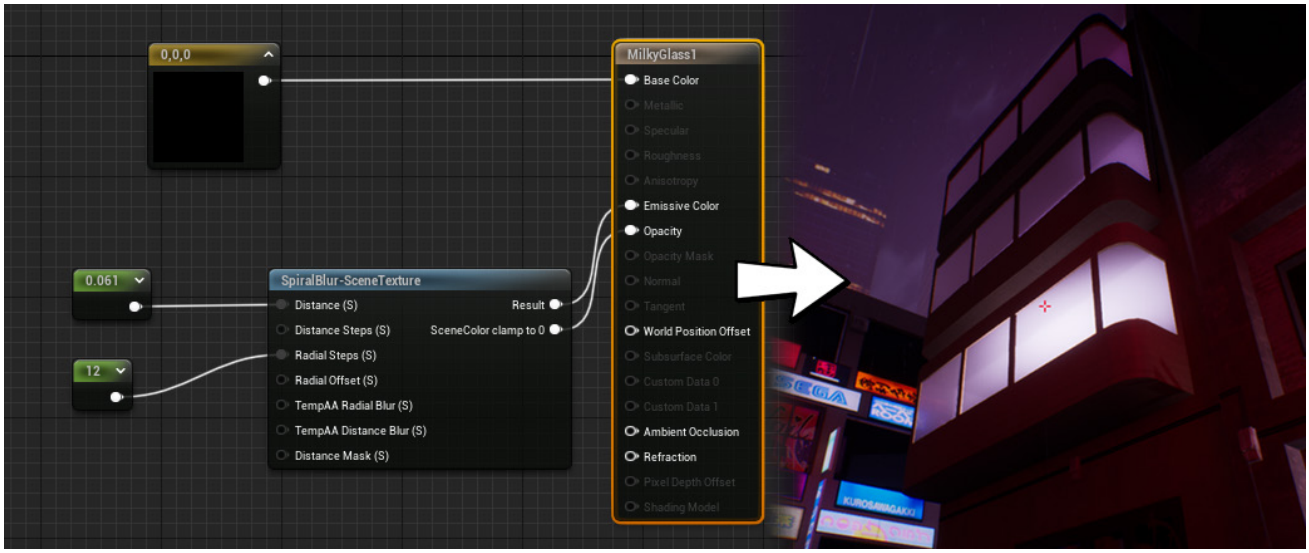

Fig 44. Material graph for the milky glass and results

4.2.4 Neon Flickering

To create the effect of a flickering light, a really simple method was used. Instead of programming externally a function that changes the emission value of some material, it is made inside said material[31]. For this, a *Time node* is multiplied by a parameter that can be modified depending on how fast the flickering is desired. Then, the result is linked to a *Sine node* to create a continuous wave, and what it is obtained from that is only left with its fraction, or in other words, a value between 0 and 1, thanks to a *Frac node*. The emission is then multiplied by this, so in the final result, it will appear as a flicker. The resulting material can be seen in Fig 45.

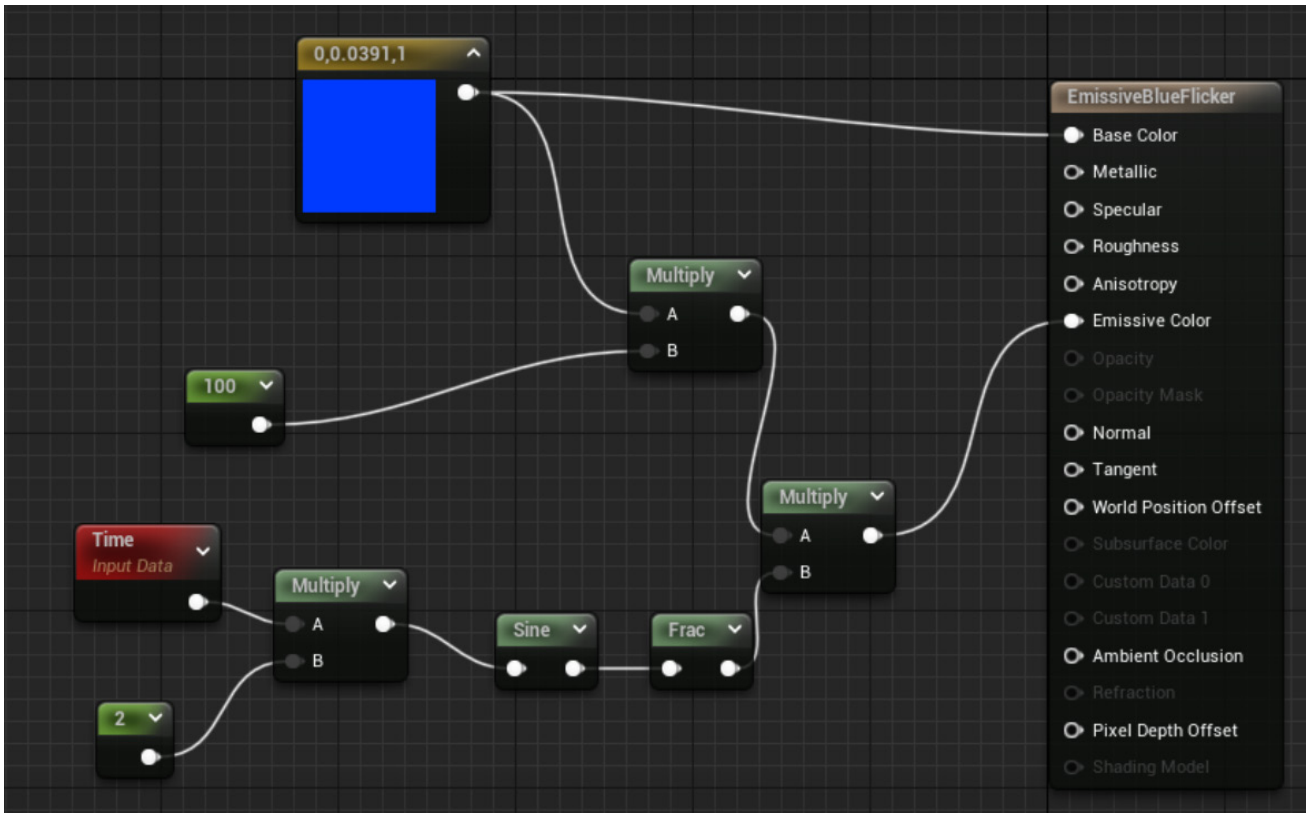

Fig 45. Material graph to make the light flicker

4.2.5 Interior Cubemap

Some of the buildings' windows are just a plane with its *UV* projected into a photograph with windows on it and with emission. This makes it on some occasions to look too flat. Another way of filling the windows without making them directly black, is to use *Interior Cubemaps*.

There are some developers that have prepared cubemaps made by hand able to be used, making a cube with a texture for each of the faces except the front one and also some planes in between the cube that use an opacity mask (Fig 46). With this, when looking through the window, it seems like a 3D room creating a parallax effect with the planes in the middle. Two different rooms have been used in the project.

Fig 46. Planes that compose the interior cubemap and how it looks from the exterior

There is another way of doing so, as seen in the technical demo that **Unreal Engine** has recently released: *Matrix Awakens*. In that demonstration, it can be seen that the rooms seen from the windows of the buildings have a 3D effect. To recreate it, an *HDRI* of an interior has been used along with the *InteriorCubemap* node in the material graph.

First of all, a parameter of value 1 has to be attached to the *Tiling value* of the node, as it allows to create a large cube with several rooms, but only one is needed to make a single room. Then, when trying to join directly the output of the *InteriorCubemap* to the texture, it returns an error, as the coordinates trying to be sent are a *Vector3*, when the texture can only take a *Vector2* value. Because of this, it has to first be transformed into an allowed value, so a *LongLatToUV* has to be used first. After being converted, the texture can be used, in this case in the *Emissive Color input* (Fig 47).

When trying this texture with a cube, however, the results are not really satisfactory, as there is a lot of stretching in the texture (Fig 48). Unfortunately there is not a lot of documentation for this aspect. Many things have been tried, and there is not a lot of understanding of how this node works, so at the end, the flat windows have been left as they were. Probably in the future more information is shared, but for now, being this version of the program certainly new, there is not a lot to investigate. Maybe the problem has more to do with the textures used, but in any case making custom ones would result in having to render (and model and texture) several rooms and making after that the cubemap textures, so in this case, having limited time, it was not possible to.

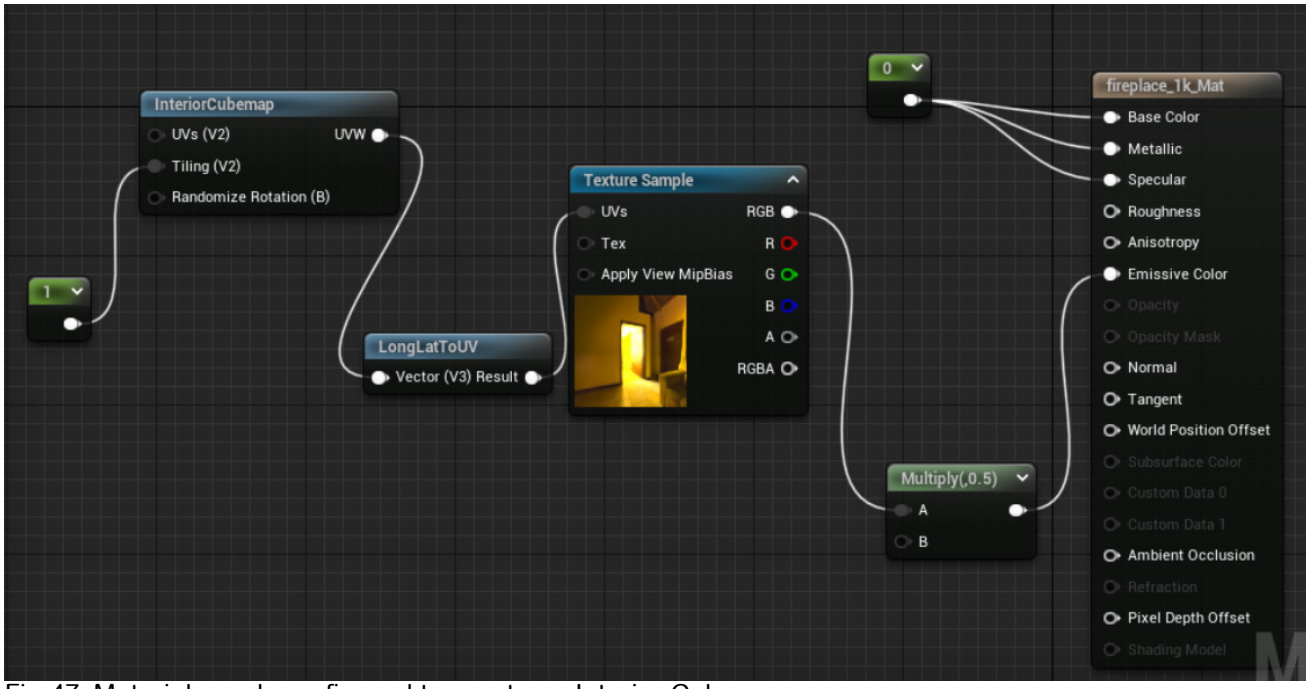

Fig 47. Material graph configured to create an Interior Cubemap

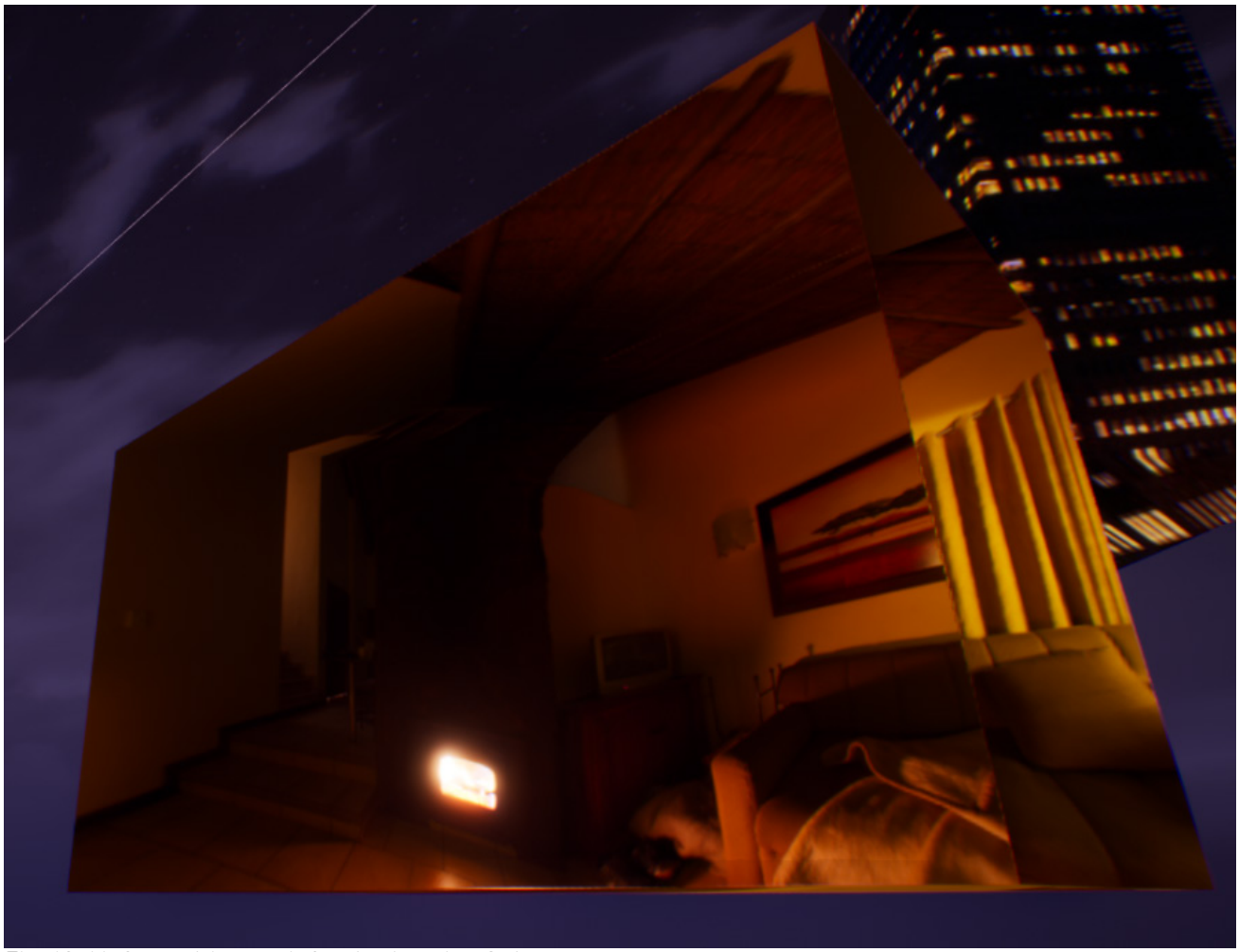
Fig 48. Unfavorable result for the Interior Cubemap

4.2.6 Decals

Unreal Engine has an add-on called **Quixel Bridge**, that has a collection of all the megascans from **Quixel**, this being models, textures and decals. It allows the user to download them in the chosen quality and use them directly in the program after importing them. A couple of models and materials have been added to the project, but what has been the most used are the decals.

They work like a volume that can be moved, scaled and rotated, and everything that enters that volume will have the decal projected into it in the direction it is facing (shown by an arrow). If there is the need for one object to not receive decals, that option is available in the object properties. The strength of the decals is that they can be placed in any area and allows to create some variation to the textures. Some models, for example, are symmetric, so if a graffiti is added in the texture, it would not look natural, as there would be one graffiti and the same one but inverted next to it. Using decals, a symmetric model can be made and there could still be added details that otherwise would make the symmetry noticeable. It is also possible to create custom decals as sometimes the **Bridge** catalog can be limited.

In Fig 49, a couple of graffiti decals are applied to the wall in the back. As it can be seen, one of the decals should go over the traffic cone, as it is in the same zone of the wall, but to avoid that, the option *Can recieve decals* of the cone has been disabled.

Fig 49. Graffiti decals projected into a wall

4.2.7 Video Textures

In order to add video adverts like the ones seen in Blade Runner for example, it was necessary to create *Video Textures*[32]. To do so, first of all a *File Media Source* had to be created, with its *File Path* with the direction of the desired mp4, which will be old advertisements in this case. After that, a *Media Player* will be created, along with a *Media Texture*. The *Media Player* will be loaded with the desired *File Media Source* and the option *Loop* checked, as the objective is to have the video always playing. When that is done, the *Media Texture* can be dropped directly into the target object, and a new material with that texture will be automatically created.

Now, the material needs some changes. First of all, the *Video Texture* will also be used as an *Emissive Color*, as it represents an LED screen. Because of this reason, a mask is also needed, so the parts that are shown are composed of little dots to make it look more realistic. For this to be possible, the material *Blend Mode* will have to be changed to *Masked*.

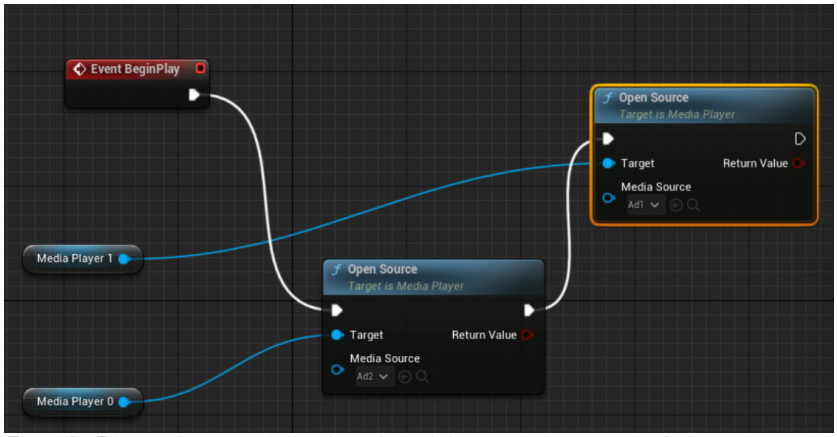

Fig 50. Blueprint used to play the videos at the start of the game

When all this is finished, the material will be ready, but if the level is played it will be noticeable that the videos are not playing and all there is is a static image. To fix this, it will be necessary to open the *Level Blueprint*. Once there, a couple of nodes have to be added. The first one is the *Event BeginPlay*, which will play an actor once the game starts.

That node will be linked to an *Open Source node*, that will have connected to it a *Media Player object reference node*. The *Media Player object reference* will have in its *Default Value* the *Media Player* required to load. In case more than one video is going to be played, different *Open Source nodes* can be linked, as the *Event BeginPlay* one can only be linked to one single node. Fig 50 shows the *Blueprint* used to load both of the videos used.

Once the *Blueprint* has been configured, the videos will be played in loop when the game starts. Fig 51 shows one frame of each of the videos used for the advertising signs.

Fig 51. Finished luminous adverts

4.2.8 Subsurface Scattering

After finishing the texturing of some japanese paper lanterns, the intention was to make them look like the real ones when they have a lightbulb inside. Using an *Emission map* would only make them look like that on one side. For this reason, an *Opacity mask* was created in **Krita** making darker spots where the wires would be, meaning that less light will pass through it, the same way it happens in real life. When placing the lantern and a light inside it in **Unreal Engine**, several options were tried: changing the *Blend Mode* to *Masked*, to *Translucent*, trying different *Lighting modes*, making it *Two Faced*, and even changing the *Shading Mode* to *Subsurface* and *Subsurface Profile.*

After some research, other people from the **Unreal** forums[33] said that they faced the same problems. It seemed like it was a bug in the Early Access version of the engine, and that the correct mode was indeed the *Subsurface shading mode*. The way to fix it was to disable the option *Cast Shadows* in any object that used *Subsurface Scattering*. In the later versions of the program this error has been fixed.

The final result can be seen in Fig 52. The japanese paper lanterns behave as expected: there is a white light placed inside each one of them that changes its color to red when passing through the “paper”. It also shows a glowing sphere inside that changes when changing perspective like it would in real life with a lightbulb and it shows the effect of the “wires” inside thanks to the *Opacity mask.*

Fig 52. Effect of the subsurface scattering on the paper lanterns

4.3 ILLUMINATION

Unreal Engine 5 uses a new dynamic global illumination and reflections system technology called *Lumen*. It is a combination of the *Voxel lighting system* and *Raytracing* but much cheaper in terms of computational load than the last one, and allows rendering through infinite bounces and indirect specular reflections in big and heavily detailed areas[15].

Lumen solves what is usually known as the color bleeding effect. If, for example, a white light hits a red object that is next to a white wall, the light picks the color of the object and reflects it into the white wall, coloring it red (Fig 53). This along with the indirect shadows create a realistic behavior of the lighting.

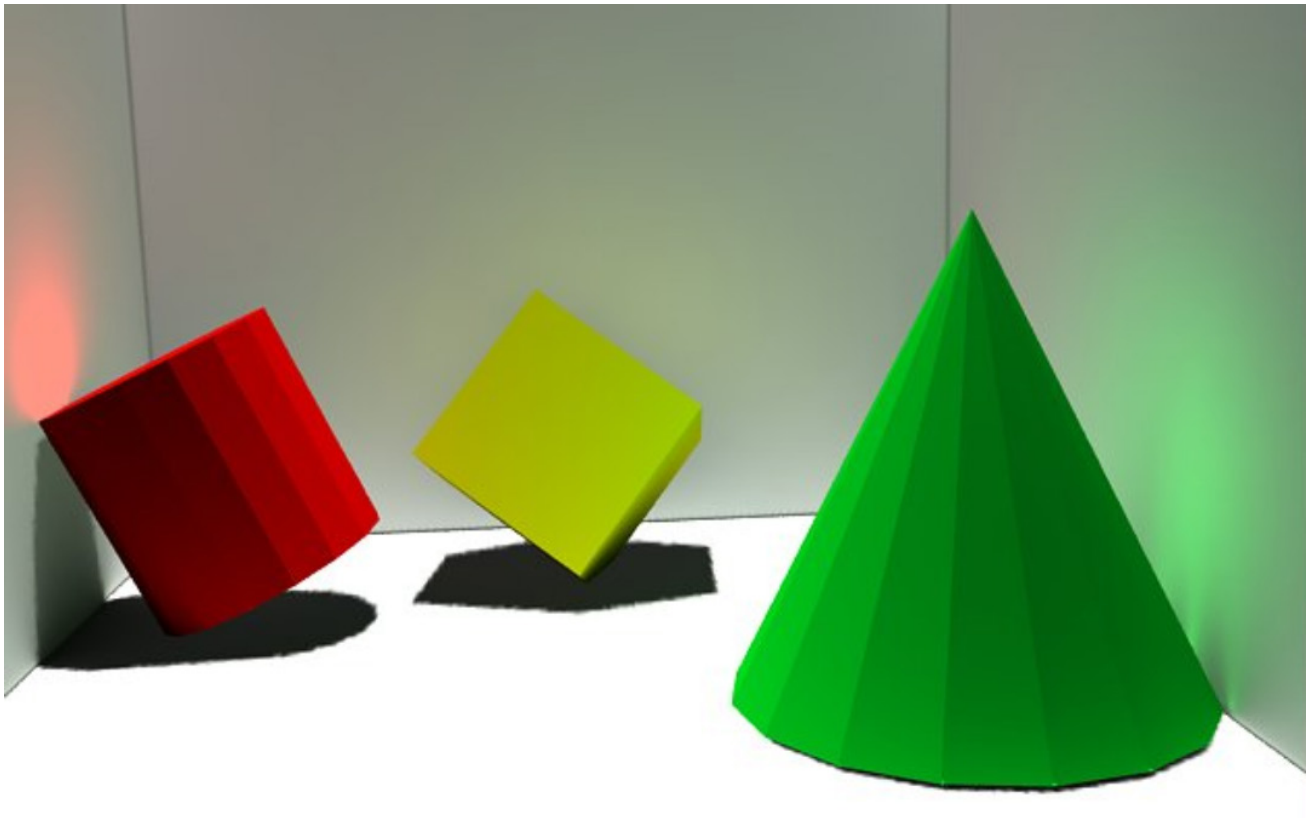

Fig 53. Color bleeding effect can be seen on the white walls

As stated before, a lot of neons have been used for this project. *Lumen* allows the objects to emit light in a dynamic way without having to make any kind of baking, and also at a low performance cost, so this technology has been useful especially for the models that have overlapping *UV*s.

To give the lights a volumetric effect, an *ExponentialHeightFog object* has also been used. This effect is the same that occurs, for example, in a concert when smoke is used so the light rays can be seen. The fog has a considerable customizability and has been adjusted to be at a certain distance from the player along with its *Density, Color* and *Opacity*, as well as the *Volumetric Fog* option to make it work as intended.

In Fig 54, the difference between having the *ExponentialHeightFog* off (left) and on (right) is noticeable. Apart from the effect of the fog itself, the light rays can be appreciated.

Fig 54. Comparison between the fog deactivated and activated

Other lights have been added in certain locations on top of the emissive objects, as it can be noticeable that when those objects can not be seen, neither the light they emit. If it made a great difference when the objects were occluded, a light imitating those properties has been added, otherwise it has been left like they were.

4.4 POST-PROCESSING

Post-Processing is a technique that allows to modify the render in numerous ways before drawing it on screen. This means that all the process configured to obtain the desirable result will be repeated each frame, in this case 60 times every second. To modify the scene in a certain way, it will be necessary to place a PostProcessVolume and adjust it to get the intended result. Multiple volumes can be placed in the scene, meaning that depending on where the player is standing, different effects will be applied. There are different kinds of “filters” that produce diverse results, and some of them have been used in this project to enhance the looks of the environment[34]:

- **Exposure**: The exposure has been modified to make the level look darker as well as the *Min Brightness* to create better contrasts of the light and get the light signs to look more realistic (Fig 55).

Fig 55. Scene before and after the exposure has been modified.

- **Bloom**: The bloom effect is used to give the light a mellow look and to expand it over the edges of the objects. This recreates a real artifact that happens with real cameras when pointing to a really bright area (Fig 56).

Fig 56. Comparison of lighting with and without bloom

- **Chromatic Aberration**: This is also an artifact that can happen to real cameras when failing to focus all colors to the same point. When this happens, it makes the colors to be on top of each other but displaced, creating an effect that is usually associated with a futuristic or “glitch” look (Fig 57), which is adequate for this scene.

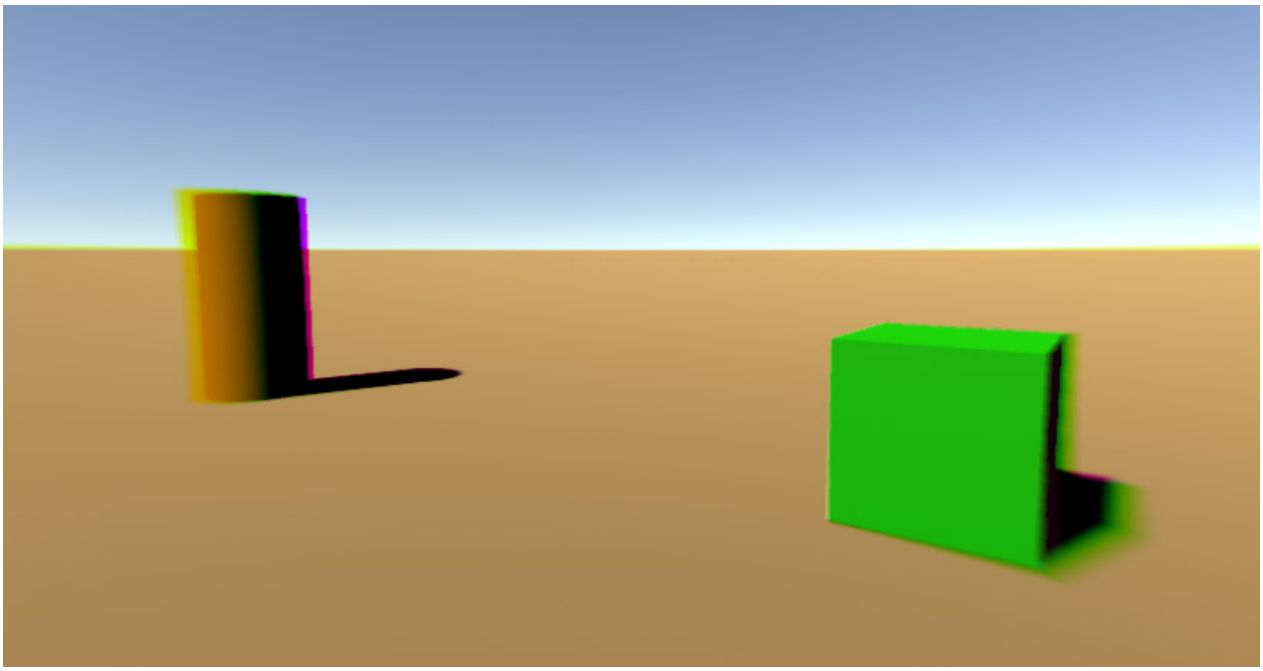
Fig 57. Chromatic aberration effect on two basic shapes

- **Color Grading**: There are several options to modify the colors of the image. *Tint* and *Temperature* have been used to make the scene more “warm” and then some small modifications in the *Global, Shadows, Midtones* and *Highlights*, based on what is usually used in the modification of photographs with programs like **Lightroom**. The objective was to achieve a more “pinkish” look, as it can be seen in Fig 58.

Fig 58. Street before (left) and after (right) color grading

- ***Lumen* Scene Detail**: The detail has been increased to have greater detail when the *Lumen* technology calculates the lighting.

- **Motion Blur**: To simulate the human eye when it sees movement, motion blur has been added to recreate it. When the camera moves around, a blur effect in the direction of said movement is applied (Fig 59).


Fig 59. Motion blur effect can be seen on the cars passing by

4.5 MOVEMENT

To move the player, **Unreal's** *Blueprints* have been used. The player is simply composed of a camera and a *CapsuleCollider* that will detect the collisions with the environment.

In order to look around with the camera, there has been used the *InputAxis Turn* and *InputAxis LookUp nodes* to get the amount of movement of the mouse, in the horizontal and vertical axis respectively. Those inputs are then connected to the *Add Controller YawInput* and *Add Controller Pitch Input nodes*, that will rotate the target (in this case the player) around itself in said axes according to the value of the input.

For the movement it works in a similar way, in this case using as inputs the *InputAxis MoveForward node* to move forward and backwards, and the *InputAxis MoveRight* to move right and left. Those inputs are both connected to an *Add Movement Input node*, but the difference is that one takes as a reference for the direction the *Forward Vector*, meaning that the movement will apply in that direction, and the other one a *Right Vector*, so it will move to its right and left depending on the value of the input.

To make the character jump, an *InputAction Jump node* is linked to a *Jump* one through the *Pressed attribute*. A *Stop Jumping node* can be added and linked to the *Released boolean,* but it is only necessary if the height of the jump is determined by how much time the jump key has been pressed. All the different aspects implemented in the *Blueprints* are shown in Fig 60.

As it can be seen, the *Blueprints* from **Unreal** make the job of adding mechanics to the game much easier, but it can be limited in some occasions if the mechanics needed are really complex.

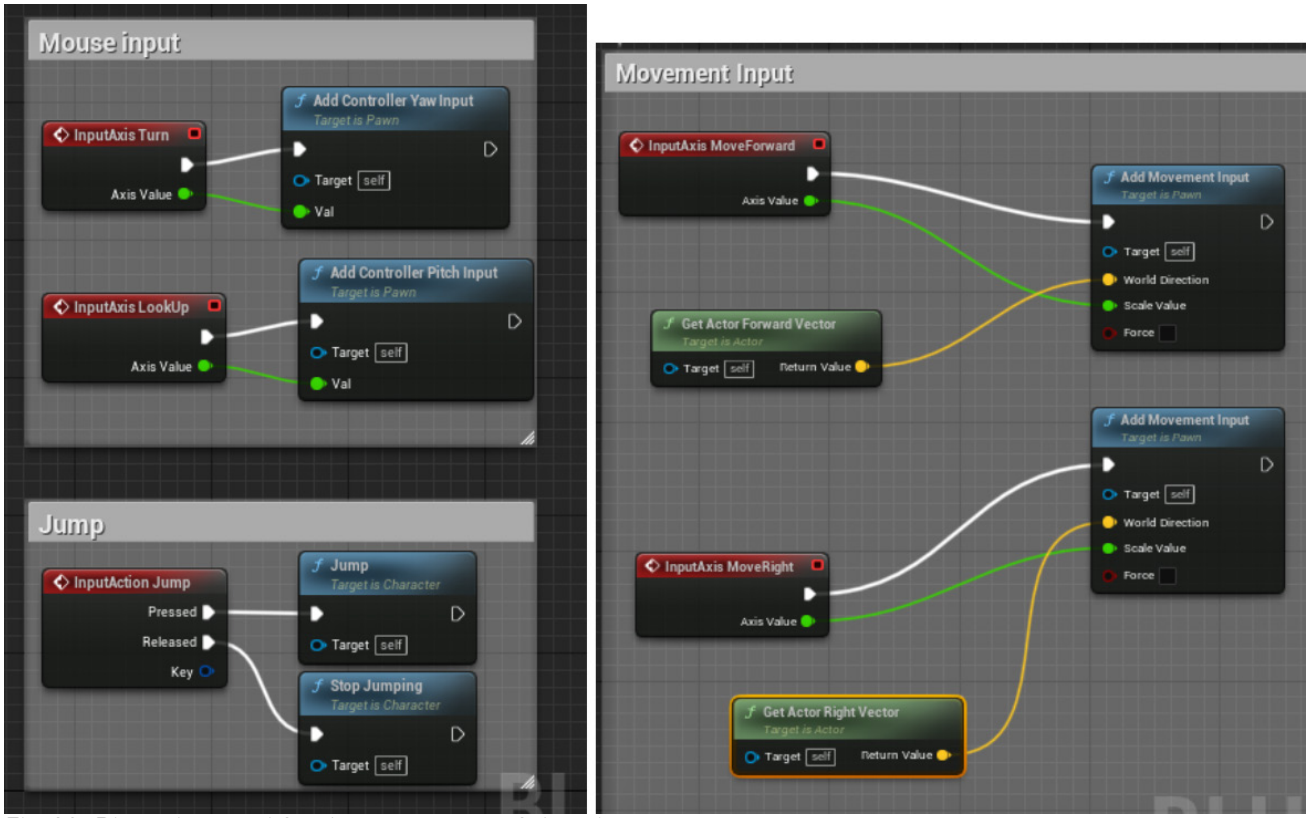

Fig 60. Blueprint used for the movement of the player

5 RESULTS

At the end of this project, a Cyberpunk scene has been successfully created using only free software. Excluding the props inside the garage, which have been downloaded form the **Bridge** addon, every mesh in the scene has been manually created, modeled and textured, using all the different techniques explained through this document. Shown below in the different figures from 61 to 65, can be seen the results obtained over the course of this work.

Fig 61. Screenshot of the street from one end.

Fig 62. Screenshot taken from the exit of the vending zone

Fig 63. Screenshot of the building with the vending machines

Fig 64. Screenshot from the other end of the street

Fig 65. Screenshot of the dead end

5.1 TIME BALANCE AND DEVIATIONS

At first, when making the technical proposal, the idea was to make two scenes, one that is the scene made in this project, and another one that would have been a room taken directly from reality and made completely with photogrammetry. As time advanced it became clear that it would not be possible to make the other scene, so the decision taken was to add at least photogrammetry to the first one.

In general, time has been miscalculated, as the development of the scene has taken much more time than expected. Part of the problem is the lack of experience making big projects where a lot of modeling and texturing is involved, and it became clear that big scenes in video games take a lot of time to make if none of the usual techniques are used like procedural modeling or the reusing of textures and models. Even though the scene has a lot of elements, it is not big enough to repeat models without it being noticeable.

Another part of the problem is that there has not been enough effort put into planning the different aspects of the work. This has been taken into account for future projects. “Modeling the assets” or “Creating materials and texturing” was too general to make conclusions, and probably breaking it down into more steps would have helped to calculate better the amount of work.

To make the work more dynamic, the method of working has been, for each building, getting the references > modeling the building > texturing it > placing it in unreal. At first each building was finished with all the props related to it, and later on in the project just the buildings, and then, at the end, the props. It may have been better to have first an idea of all the buildings that would be needed instead of making them one by one completely, and after modeling them all, start to texture them. This way, some textures could have been reused and in general it could have been more efficient.

In the initial idea of the project, the intention was to also include a futuristic car in the scene. Even though it was modeled, the results were not satisfactory, so at the end, the idea was discarded, as it probably needed much more time to design a car from scratch that looked good. This resulted in wasted time.

Something positive about the managing of time is that the project followed a constant pace of more or less the same hours every day, although maybe not completely efficient, as those are the disadvantages of working one by themself if there is not a good organization.

Tables 3 and 4 show the real hours spent on the project and the actual organization that has been followed.

TASK	ESTIMATED	REAL
Research	40h	20h
Writing the memory	40h	30h
Preparing presentation	10h	10h
Technical proposal	5h	5h
Modeling the assets	120h	200h
Creating materials and texturing	50h	70h
Arranging the scene	25h	45h
Testing the scene and interactivity	10h	10h
TOTAL	300h	390h

Table 3. Planning compared to real time spent

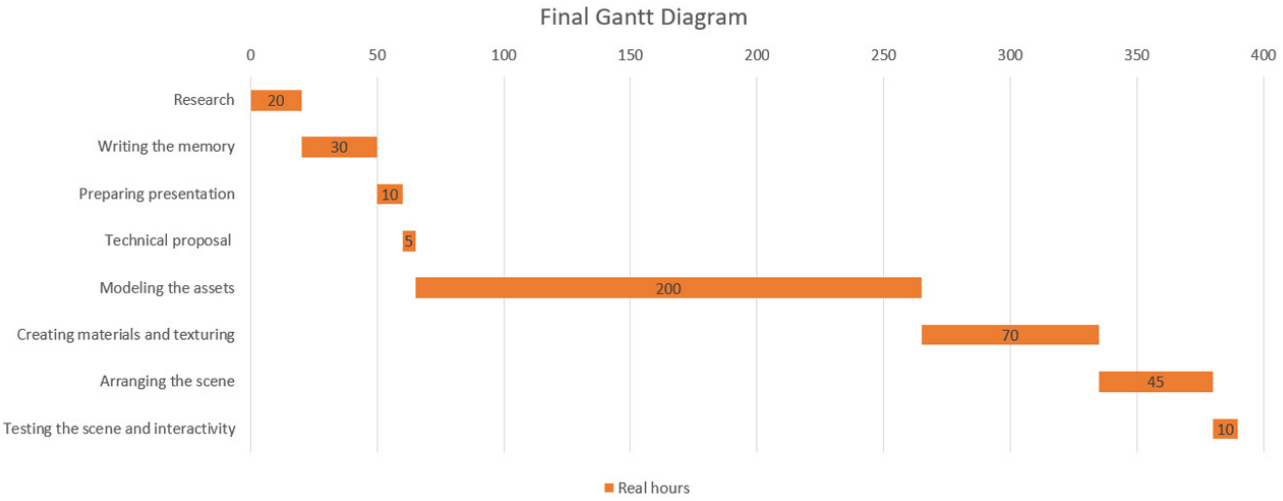

Table 4. Gantt diagram with the real organization

5.2 ACHIEVING THE OBJECTIVES

Taking into account the objectives mentioned in the technical proposal, most of the objectives have been completed:

- A lot of important aspects regarding visual realism have been learned over the course of this project, like the use of *Lumen, Nanite*, Photogrammetry, etc.

- All of said aspects have been put into practice in the scene and have been understood throughout the use of them, but the scene has not been made interactive.

- Having achieved a desirable result, that is comparable to one that could have been obtained using expensive software, it could be said that the objective has been completed.

5.3 DOWNSIDES OF THE FREE SOFTWARES

The point of this project is to show that using only free software, it is possible to obtain good results, thus proving that it is not necessary to spend a lot of money buying expensive software to be able to work or practice one by themself. But nonetheless, not everything has been perfect, and comparing it to their substitutes show that there is still a long way to go, especially with **Quixel Mixer**, which is still in its beta version.

With **Blender** there has been no problem, and as an open source it has shown to be up to the task when compared to other well known programs. The proof of it is that now a lot of companies are starting to request **Blender** experience.

Meshroom and **Instant Meshes** can not be compared to other paid software due to the lack of experience in any of those. **Krita**, on the other hand, has not shown any problem compared to other programs when doing all the tasks required for this project.

Here below will be enumerated the difficulties faced with **Quixel Mixer** compared to other programs like **Substance Painter**:

1. **Quixel** does not have *auto-unwrap*, which is the reason why every map had to be rendered in **Blender**. It does not take a lot of time for each model, but when doing several objects, it is much more comfortable to just use that option that at the end saves a lot of time.

2. There are no polygon masks (which is why *ID maps* are needed). In **Substance Painter**, it is possible to just select the faces from the program to create the mask, whereas in **Quixel** it is needed to import an *ID map* rendered beforehand.

3. The *Paint masks* are extremely slow when painting, and usually requires lowering the resolution to be able to paint. This is due to the way it works, as it runs real-time curvature calculations every time something is modified, so if it has a lot of layers, it becomes slow.

4. Sometimes the Control + Z command has not worked properly when painting a mask, meaning that if an error was made, it had to be fixed by hand again.

5. Any kind of mask can not be copied and pasted, so if the intention is to make a layering of several materials in a concrete zone, it would have to be painted every single time.

6. The selection of brushes is much more limited.

7. Sometimes the *Smart Materials* do not work as they should, and it just shows a repetition of what seems like a tiny texture.

8. Sometimes adding a *Smart Material* to a custom model made the program completely crash, and even restarting the program would not fix it, so that model would be permanently excluded from using a *Smart Material*.

9. It has less customization compared to the programs that use procedurally generated materials

The process with **Unreal Engine** started when it was still in the Early Access of the 5.0 version, but the official version released in the middle of the project, so most of them have been fixed:

1. When enabling the *Nanite* option to the meshes, sometimes the models would swap their materials, but would return to normal once selected. This has been fixed in the last version.

Fig 66. Objects with Nanite activated changing their materials randomly

2. When using the **Bridge** interface inside **Unreal**, it would make everything go extremely slow. This has been fixed, but it still makes it rather slow.

3. *Subsurface Scattering* did not work correctly, so after trying different options, the solution was to disable the *Cast Shadow* option. This has been fixed.

6 CONCLUSIONS

After the project being finished, and taking into account that it has not been by a professional 3D artist with years of experience, the final result has been satisfactory. For this reason, the point of all this work has been proven: it is possible to create a photorealistic scene for a videogame using only free software.

On the other hand, **Quixel Mixer** for example is free, but only if it is used with **Unreal Engine**, so at the end the motivation behind it is merely economic, as, although the software is free, publishing a game made with **Unreal** has a cost for the developer depending on how much money they make, so the company is interested in having a lot of games made with **Unreal**. At the end companies like these can not be trusted, as one strategy that is usually used is to have something free or at a low price to eradicate the competitors and then raise the prices again.

A good example of the way to go is **Blender**. It is completely free, without conditions, and open source, and its community is bigger each day and is constantly improving aspects of the program. It has new updates constantly, and has reached a point where it is at the level of other programs like **3DSMax**. A lot of developers and artists are changing from those softwares to **Blender**, even professional ones that have worked in important projects, as they are too punishing for people with less resources.

The bigger the community the better, and free software help to democratize the work. Starting to model from scratch is now as easy as downloading **Blender** for free and watching the millions of tutorials available at no cost made by the enormous community at YouTube or other platforms. Programs like **Substance Designer** or **Substance Painter** make the job much easier, so not being capable of paying for them is a huge disadvantage for a small developer.

On top of this, it is not only positive for small developers, but also for big companies and its workers. When taking into consideration how much money it is spend on licenses for each worker and every program they use, having the option to spend it somewhere else is a huge change. It could make it possible to hire more workers, thus reducing the amount of work for the employees and the infamous “clutches”, and also reducing the time needed for the developement.

Techniques used in this project like the photogrammetry also show a huge potential, which is already being exploited by **Quixel**. It creates totally realistic models that can also have its texture changed, so a single mesh could be transformed into other objects or with different styles while maintaining its shape. Along with the *Nanite* technology, it could be the future of the video game and 3D industry as a whole.

6.1 FUTURE WORK

The work presented in this document is one that does not have an end. Technologies are constantly evolving, and so is the video game industry, as both things are linked together. It is essential for the workers in this field to constantly experiment and research to keep up with the emerging technologies.

Video games are not immutable, and the concept itself has been in constant change thanks to the possibilities that the new technologies offered, making it possible to create new mechanics, visual representations, etc. For this reason, it is also necessary to make those technologies available to everyone, as collectively the knowledge and development of them are much greater and allows them to evolve much faster. This is why part of the future work is to also keep supporting free and open source software.

As much as it may seem, 300 hours is not enough time to really comprehend all the technologies used in this project, so further research will be needed to achieve better results and more efficient ones, taking into account the experience acquired during the development.

7 BIBLIOGRAPHY

1. PC Componentes. (n.d.). *RAM Memory Search. PC Componentes.* Retrieved May 12, 2022, from https://www.pccomponentes.com/buscar/?query=ram&

2. Benchmarks. (n.d.). *NVIDIA GeForce GTX 1070 Review.* UL Benchmarks. Retrieved May 12, 2022, from https://benchmarks.ul.com/hardware/gpu/NVIDIA+GeForce+GTX+1070+review

3. PassMark Software. (n.d.). P*assMark - Intel Core i7-6700 @ 3.40GHz - Price performance comparison. CPU Benchmarks.* Retrieved May 12, 2022, from https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i7-6700+%40+3.40GHz&id=2598

4. Talent. (n.d.). *Salario para Artista en España - Salario Medio. Talent.com.* Retrieved May 12, 2022, from https://es.talent.com/salary?job=artista

5. Glassdoor. (2022, May 9). *Sueldo: 3D Artist (Mayo, 2022).* Glassdoor. Retrieved May 12, 2022, from https://www.glassdoor.es/Sueldos/3d-artist-sueldo-SRCH_KO0,9.htm

6. Adobe. (n.d.). *Plans and pricing for Creative Cloud apps and more.* Adobe. Retrieved May 12, 2022, from https://www.adobe.com/creativecloud/plans.html?filter=3dar&plan=individual

7. Autodesk. (n.d.). *Comprar 3ds Max.* Autodesk. Retrieved May 12, 2022, from https://www.autodesk.es/products/3ds-max/overview?term=1-MONTH&tab=subscription

8. Maxon. (n.d.). *Planes y precios*. Maxon. Retrieved May 12, 2022, from https://www.maxon.net/es/buy#monthly

9. Agisoft Metashape. (n.d.). *Online Store. Agisoft Metashape.* Retrieved May 12, 2022, from https://www.agisoft.com/buy/online-store/

10. 3DCoat. (n.d.). *Licencia permanente, Alquiler con opción de compra, Opciones de suscripción.* 3DCoat. Retrieved May 12, 2022, from https://3dcoat.com/es/buy/

11. Adobe. (n.d.). *Plans and pricing for Creative Cloud apps and more.* Adobe. Retrieved May 12, 2022, from https://www.adobe.com/es/creativecloud/plans.html

12. Youtube. (n.d.). *Quixel*. YouTube. Retrieved May 12, 2022, from https://www.youtube.com/c/quixeltools

13. Ahearn, L. (2017). *3D Game Environments: Create Professional 3D Game Worlds*. CRC Press, Taylor & Francis Group.

14. Evans, D. (Ed.). (2013). *Digital Mayhem 3D Landscape Techniques: Where Inspiration, Techniques and Digital Art Meet.* Focal Press.

15. Fabián, A. (2021, June 1). *Conoce todo sobre Lumen en Unreal Engine 5.* UT-HUB. Retrieved May 12, 2022, from https://www.ut-hub.com/lumen-unreal-engine-5/

16. Fabián, A. (2021, June 1). *Descubre la importancia de Nanite.* UT-HUB. Retrieved May 12, 2022, from https://www.ut-hub.com/nanite-unreal-engine-5/

17. Quixel. (2020, August 21). *Baking and Exporting Custom Assets in Blender.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=dmoy1p9z8vY

18. Quixel. (2020, August 5). *Mixer Fundamentals: Smart Materials.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=9pFjAlU2gQw

19. Flipped Normals. (2019, February 20). *Texture Maps Explained - Essential for All Texture Artists.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=ZOHNRlrd1Ak

20. Quixel. (n.d.). *Megascans.* Quixel. Retrieved May 12, 2022, from https://quixel.com/megascans

21. Youtube. (n.d.). *Ian Hubert*. YouTube. Retrieved May 12, 2022, from https://www.youtube.com/c/mrdodobird

22. Hubert, I. (2019, August 16). *Making Pipes in Blender - Lazy Tutorials.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=vTADG1omjVY

23. Schenk, T. (2005). Introduction to photogrammetry. The Ohio State University, Columbus, 106.

24. Aunmar, M. (2022, January 16). *How to Improve Your 3D Modeling with Photogrammetry.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=0ohnayBu8SE

25. Prusa, J. (2021, July 1). *Meshroom to Blender Low-Poly - Clean up photogrammetry Tutorial.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=-dc4KN2bdrw

26. Hubert, I. (2020, January 5). *Make Neon Signs in Blender - Lazy Tutorials.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=8eNN2Ep3Rqs

27. Hubert, I. (2019, October 11). *Make Cities with Blender - Lazy Tutorials.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=JjnyapZ_P-g

28. Code Like Me. (2021, November 1). *Unreal Engine Rain Particle with Splashes using Niagara FX System.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=qmZCW7eQ6rc

29. Quixel. (2020, October 13). *Megascans Plugin for Unreal Engine: Vertex Blend Material.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=j_3_IYyydgA

30. Marvel Master. (2018, January 14). *UE4 Tutorial: Milky Glass - fastest and easiest way - aka frosted glass [Unreal Engine 4.18].* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=527ZcxYnf_s

31. 3D Asset Library. (2021, October 4). *How To Create A Simple Neon Light Flicker In Unreal Engine.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=XCycpz0V-Rg

32. Jayanam. (2017, June 5). *UE4 Media Player to Render a Media Texture.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=7OEbO353_GM

33. Unreal Engine. (n.d.) *Subsurface Scattering is broken in UE5*. Unreal Engine. Retrieved May 12, 2022, from https://forums.unrealengine.com/t/subsurface-scattering-is-broken-in-ue5/241244

34. Gonçalves, S. D. G. (2021, Nov 25). *Post-Processing para videojogos*. uBibliorum. https://ubibliorum.ubi.pt/handle/10400.6/12016

35. Quixel. (2018, October 18). *Leveraging Decals with Mixer.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=wq8dr5FNIhs

36. Quixel. (2020, July 8). *Mixer Fundamentals: Painting.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=Yt78oTprWjY

37. Faucher, W. (2022, February 22). *Lighting a NIGHT-TIME exterior in Unreal.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=1LfiYtKDsac

38. Unreal Engine. (2019, September 27). *Materials - Tinted Glass Part 2 | Tips & Tricks | Unreal Engine.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=XRwFh6s5wqE

39. Unreal Sensei. (2021, June 3). *Unreal Engine 5 Beginner Tutorial - UE5 Starter Course!* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=gQmiqmxJMtA

40. Urschel, J. (2020, April 8). *Make a funky Japanese vending machine in Blender.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=4C0g64nQIwU

41. Wicked Engine. (2017, August 30). *Voxel-based Global Illumination – Wicked Engine Net. Wicked Engine Net.* Retrieved May 12, 2022, from https://wickedengine.net/2017/08/30/voxel-based-global-illumination/

42. Coronado, C. (2021, June 2). *Tutorial Unreal Engine 5 para principiantes en Español* (Básicos, Lumen, Nanite y Quixel Megascans). YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=OxzkkbpuxLA

43. The CRYER. (2021, January 9). *Ue4 Niagara Steam Smoke tutorial.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=cuQyeQxO0Kk

44. Lucas, J. (2019, August 30). *Cyberpunk Street in UE4 - Extended Breakdown.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=mVyXGmgb9lk

45. Martín, A. (2019, June 26). *Las Claves del Fotorrealismo - Conceptos básicos de iluminación.* YouTube. Retrieved May 12, 2022, from https://www.youtube.com/watch?v=w_IIiGTI6cY

8 LIST OF FIGURES

9 LIST OF TABLES