UNIVERSITAT JAUME I

Access to Z-buffer information for the development of a video game with a depth effect as a mechanic.

CONTACT

Óscar Esteban Villalba

Degree Final Project's technical report

Bachelor's Degree in Video Game Design and Development

Universitat Jaume I

Supervised by: José Martínez Sotoca

INDEX

Summary

This document presents the whole work done for the final project of the Design and Development of Video games degree, based on the creation of an experience made in Unity3D and built for PC platform. The main mechanic of the game is a temporal jump between two different dimensions, achieved by doing a depth sweep with in depth with a Z-Buffer information. More specifically, in order to reach a better immersion in the game environment and make the effect more impressive, the camera will be in first person perspective. Furthermore, there will be a small story behind the gameplay to add a touch of mystery to the experience.

Keywords: Video game, Z-buffer, Unity3D, First Person, Dimensions

Figures Index

1. Introduction

1.1. Work motivation

The main motivation of this project is making a video game with realistic assets and lighting, taking into account as reference two different aesthetics. More specifically, a typical Spanish pub or cafeteria, and a dystopian neo noire futuristic pub.

Furthermore, I want to use this project to learn techniques that improve the screen space visual effects in video games, and one of those makes beautiful the technical mechanic of the project.

Also, I think that making a 3D video game is one of the most ambitious challenge that a student can face, and this is what motivates me to do it and learn a decent workflow for my future as a 3D artist.

1.2. Objectives

Nowadays there are a lot of techniques to make a video game looks better, sometimes applying shaders in objects, or making camera effects. In this project I have focused in this latter case, by accessing scene depth information (Figure 1) to create a visual effect that sweep the environment back and forth.

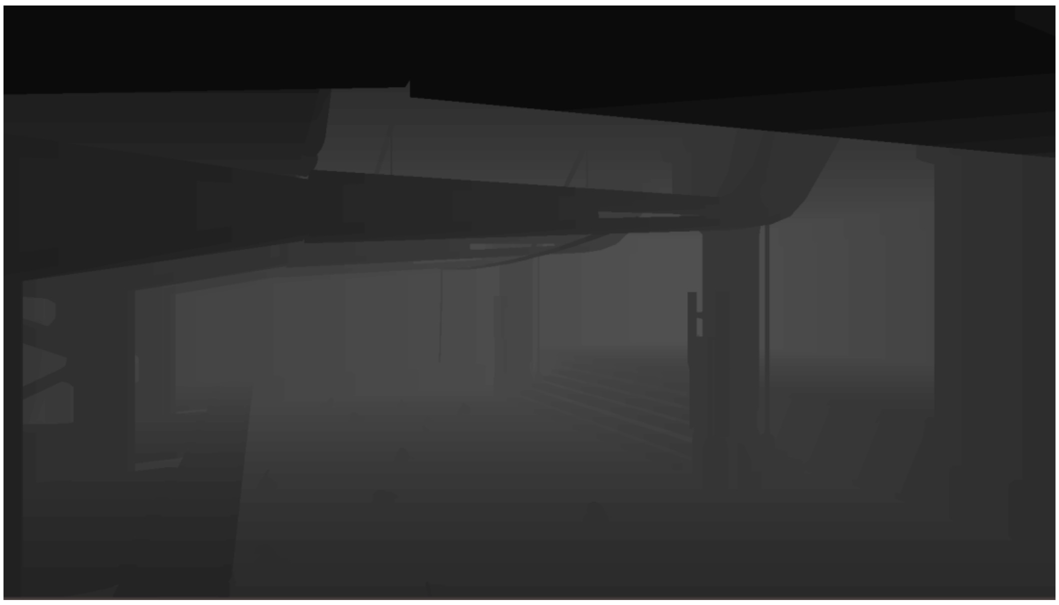


Figure 1: Depth information from one of my project scenes

This effect could be used to implement a bunch of different mechanics by introducing them into a story context, for example, radars, scene color filters or post processing effects, but in my case it was decided to apply the technique to jump between present and future, and this is the main reason why I have developed a simple demo of a story called *Contact.*

Specifically, the objectives presented in this project are:

- **Obj1:** Development of a good and clean menus system, including panel animations, loadless navigation between menus, and understandable information in every panel.
- **Obj2:** Make a project as a demo, without any professional narrative approach, where the story is a way to contextualize the effect that I want to develop.
- **Obj3:** The puzzles of the game must be discovered by the player, and each one will consist of different mechanics.
- **Obj4:** The environment is formed by realistic 3D assets, such as these assets must be historically believable, it is necessary to make an investigation of every model I will make.
- **Obj5:** One important aspect is the kinesthetic or group of sound effects and animations that give feedback to the player, making him feels like he is really doing something in the environment.
- **Obj6:** In a future, I wish I could scale the project up with new features, so it needs to be developed with that in mind.

1.3. Software

Several tools were used during the development of the project, helping me to accelerate the everyday workflow:

- **[Blender]:** A free open source modelling tool for every asset of the video game, including the architecture.
- **[Substance Painter]:** A texturing tool that allows me to apply realistic materials on the models I export from Blender.
- **[Photoshop]:** The tool that completes the 3D design workflow, allowing me to create texture atlas for optimizing materials. Also, it is used to design the User Interface (UI) and menus.
- **[Unity3D 2018.2.9f1]:** The game engine where all the project is made.
- **[Visual Studio]:** Every C# script of the game is written in this IDE.

Fortunately, there are a lot of online tools that helped me to inspire my work and keep it safe.

- **[Github]:** This tool permits me to keep my project safe in the cloud, and monitor every change is done in the video game.
- **[Artstation]:** An online repository where the people share 3D and 2D work that I used as reference.
- **[Substance Share]:** The official web of Allegorithmic, the Substance Painter company, where you can use a lot of materials for your 3D assets.
- **[FreeSound]:** Another repository where you can find and download free sound effects for your project.
- **[MP3 Cut]:** This one is very useful when you need to cut audio or convert it to .mp3 format.

Also, few Unity plugins were used to improve the final quality:

- **[TextMeshPro]:** plugin that replaces default Unity text and has an easy font creator.
- **[PostProcessingStack]:** A tool that has a lot of post processing effects which can increase video game visual quality.

2. Planning

The total work was planned for 300 hours, in this section is detailed every task divided by groups according to the development fields.

Every task is defined following these time intervals.

Documentation (40 hours):

- Technical Proposal (5 hours).
- Game Design Document (5 hours).
- Project Report (30 hours).

Design (50 hours):

- Main plot of the video game (10 hours).
- Layout the environment in paper (10 hours).
- Design the set of puzzles (10 hours).
- Design UI and Navigation (20 hours).

Modeling (100 hours):

- Collect a lot of references (15 hours).
- Block the environment (15 hours).
- Modeling a great bunch of props and realistic texturing (70 hours).

Programming (70 hours):

- Triggering all the story steps (15 hours).
- Simple saving data system (10 hours).
- Main Z-Buffer transition effect (15 hours).
- Robust scene changing system (5 hours).
- SFX and VFX to increment video game kinesthetic (5 hours).
- Menus and HUD (10 hours).
- Interaction with environment elements and puzzles (10 hours).

Music and Lighting (40 hours):

- Find ambient music and SFX (10 hours).
- Allocate lights and global illumination bakes (they could be done during no working hours) (30 hours).

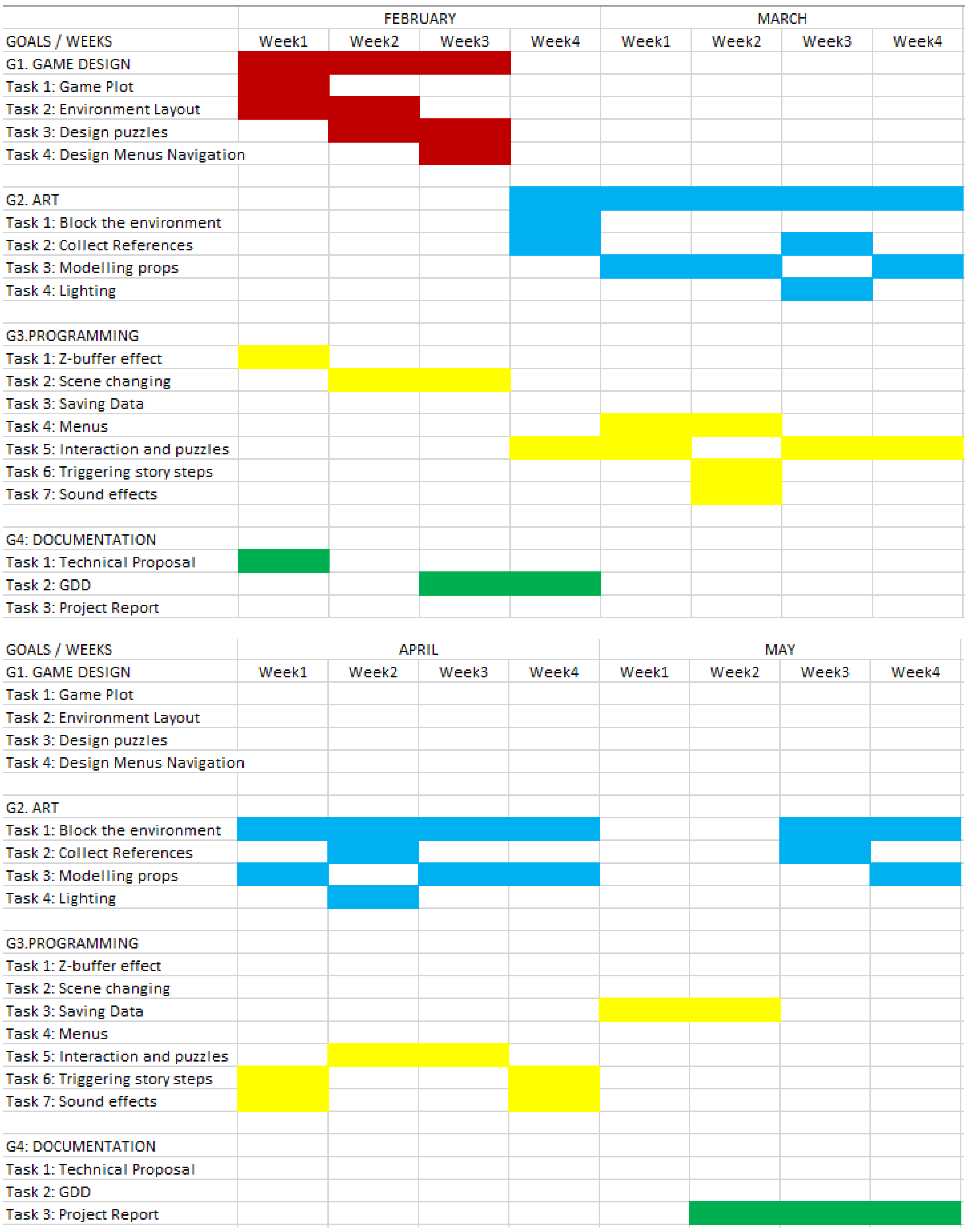
Final Presentation (10 hours).

The weekly plan that was stablished started in the beginning of February, doing work 4 hours every day of the week, from Monday to Sunday.

Due to my university internship, it was decided to do the work every afternoon, because I would be busy in the morning.

In addition to the hours planning, another weekly planning was made in order to stablish the goals deadline (Figure 2).

As it was mentioned at the beginning of this section, due to programming bugs and features that were very difficult to implement, these work plans have not been entirely followed actually, but they have been so useful to stablish good work patterns.



	FEBRUARY				MARCH			
GOALS / WEEKS	Week1	Week2	Week3	Week4	Week1	Week2	Week3	Week4
G1. GAME DESIGN	🟥	🟥	🟥					
Task 1: Game Plot	🟥							
Task 2: Environment Layout	🟥	🟥						
Task 3: Design puzzles		🟥	🟥					
Task 4: Design Menus Navigation			🟥					
G2. ART				🟦	🟦	🟦	🟦	🟦
Task 1: Block the environment				🟦				
Task 2: Collect References				🟦			🟦	
Task 3: Modelling props					🟦	🟦		🟦
Task 4: Lighting							🟦	
G3.PROGRAMMING								
Task 1: Z-buffer effect	🟨							
Task 2: Scene changing		🟨	🟨					
Task 3: Saving Data								
Task 4: Menus					🟨	🟨		
Task 5: Interaction and puzzles				🟨	🟨		🟨	🟨
Task 6: Triggering story steps						🟨		
Task 7: Sound effects						🟨		
G4: DOCUMENTATION								
Task 1: Technical Proposal	🟩							
Task 2: GDD			🟩	🟩				
Task 3: Project Report								
GOALS / WEEKS	APRIL				MAY			
G1. GAME DESIGN	Week1	Week2	Week3	Week4	Week1	Week2	Week3	Week4
Task 1: Game Plot								
Task 2: Environment Layout								
Task 3: Design puzzles								
Task 4: Design Menus Navigation								
G2. ART								
Task 1: Block the environment	🟦	🟦	🟦	🟦			🟦	🟦
Task 2: Collect References		🟦					🟦	
Task 3: Modelling props	🟦		🟦	🟦				🟦
Task 4: Lighting		🟦						
G3.PROGRAMMING								
Task 1: Z-buffer effect								
Task 2: Scene changing								
Task 3: Saving Data					🟨	🟨		
Task 4: Menus								
Task 5: Interaction and puzzles		🟨	🟨					
Task 6: Triggering story steps	🟨			🟨				
Task 7: Sound effects	🟨			🟨				
G4: DOCUMENTATION								
Task 1: Technical Proposal								
Task 2: GDD								
Task 3: Project Report						🟩	🟩	🟩

Figure 2: Weekly planning

Nonetheless, in practice, the total work has exceeded that mark.

The original planning included 300 hours starting in February, it was warned that the project needed to be planned to last less than 300 hours, keeping in mind the possible issues during the project development.

The final planning has been quite different from the initial one, despite working more than 4 hours a day, around 5 or 6, the project cannot be finished at 100%, and the total of hours sum up to around 500 – 550 hours, that could be few more because the work in the morning was impossible due to an internship period.

Also, as it was decided, the weekly work included Saturday and Sunday, there have been work days even in holidays.

Talking about complexity and difficulty during the development, there are several aspects that need to be mentioned.

The most difficult part of the planning is the one related with the main effect and the scene swap. Scripting shaders is a field completely different to the common programming, and despite of the basics learned in the degree, there are a lot of aspects that are not covered by the subjects; also, this effect includes a scene swap, and this means that it is necessary to combine usual programming and shader scripting, which is more difficult to keep under control.

Furthermore, one of the challenges inside this project is creating a big set of realistic 3D assets. The followed workflow includes several fields that need to be covered in every prop that is made, thus, the hours spent in each one increases significantly.

3. Game Design

Sometimes, there is a tendency to ignore and underrate the design of a video game in the first steps of the development, and certainly it is one of the most important stages of the project.

One of the typical mistakes in narrative video games is that usually, the player is trying to immerse himself into the story, and he suddenly encounters with something or someone that does not make sense, talking about the relation between gameplay and mechanics. For example, the game shows you a character that has never touched a gun, and you can fire everything you find without any problem at the same time. That kind of mistakes are grouped by a term called Ludonarrative Dissonance [Hocking C, 2007].

Another mistake is seen in games where the developers try to create a full immersive experience, but they also make a user interface full of information, or digits floating in the screen, and they are not justified by the story.

There are more problems like these, that bring to light bad design decisions. However, in this document I will focus on applying a technical effect, making it an important aspect in game mechanic.

From the previous reasoning, the first step was taking references of everything, game mechanics, visual style, stories, not only from other video games but also from films or TV series. Furthermore, a period of brainstorming was carried out.

In order to explain these design points, they are separated in different sections.

3.1. Story

The main plot of Contact is the first temporary anomaly located in Spain in the 80s. Rodrigo, a Spanish physicist who is engulfed in a depression get locked inside a cafeteria after one of his usual depression nights, and as soon as the main door closed strongly, he started to hear kind of strange synth sounds.

Suddenly, he gets himself into a quest which main goal is connecting that place between the present and the future.

The adventure will be the icing on the cake for his research about antimatter involvement in wormholes, the only thing that makes him wake up every morning.

The story begins with Rodrigo inside a cafeteria, where he cannot see anything, and his first mission is to turn on the power of the place.

At the moment he enters in the kitchen, he starts to hear weird sounds, something played with a synthesizer, but Rodrigo does not understand the meaning of that signals, and he thinks it is just a little bit of hangover.

Now that Rodrigo has turned on the power, he immediately sees a paper inside the electrical box, and decides to take it and read the content. But that paper only has four codes written, a combination of male and female symbols, however, this is so strange for him because it does not make sense, maybe it could be a simple numeric password translated to these codes, but he thinks it could be something paranormal, and keeps exploring the place.

It is near the toilets where Rodrigo hears the weird sound again, and he suddenly realizes the codes could be related to both toilets, the women one and the men one. For this reason, he decides to touch everything inside the toilets: walls, sinks, doors or even mirrors, but it is when he activates one of the toilets flushes where he feels a "click" sound emitted by the toilet. This cannot be a coincidence, and he immediately tries to follow the codes by activating men toilet flush or women one when a male or female code appears respectively.

Once Rodrigo introduces every code properly, the sound stops and then he understands that there is someone who is trying to communicate with him, he needs to follow the steps, but there is not any reason to keep going, and maybe it is dangerous or a simple paranoia.

Nevertheless, at the same time that he leaves the toilets corridor, Rodrigo sees the TV is emitting Morse translations, and he realizes that he was not introducing codes to acquire something, he was helping the mysterious person to access the antenna, and now this entity is trying to explain the next step.

Now the meaning of the Morse translations is not quite difficult for Rodrigo, after few minutes watching everything inside the cafeteria, the only one thing behaving strangely is a lamp over the bar, and certainly it is blinking with a pattern that repeats every time. It might not mean anything, but Rodrigo translates the pattern into numbers helping himself with the TV codes, but he does not know where to introduce the resulting code. What kind of device inside the cafeteria could be used to introduce numbers? Nothing in the kitchen, and the toilets just have dirt now. The only thing that have numbers to press in that place is the cash register, where effectively, the code needs to be introduced.

Doing this last step, and without knowing the impact of his actions, Rodrigo has created a temporal fracture, and suddenly, a strange device appears from that fracture, a device that can be used to travel between times, and allows him to go to the future.

3.2. Art

One of the main goals of this project is making the player feels inside a real world; thus, I have made a set of realistic 3D assets that contextualize the two-time dimensions, the 80's Spanish cafeteria and the futuristic dystopian bar.

In addition, the lighting has a very important role in the project because it is responsible for making those assets look properly.

In order to achieve a believable environment, I have searched a lot of references of old cafeterias in Spain, because today they are very different and tend to be more themed and decorated with modern things. In addition, I watched several films related to dystopian worlds just to see how they are built. The films I mean are recent titles like *Blade Runner 2049* (Denis Villeneuve, 2017) (Figure 3) or *Ready Player One* (Steven Spielberg, 2018) (Figure 4).

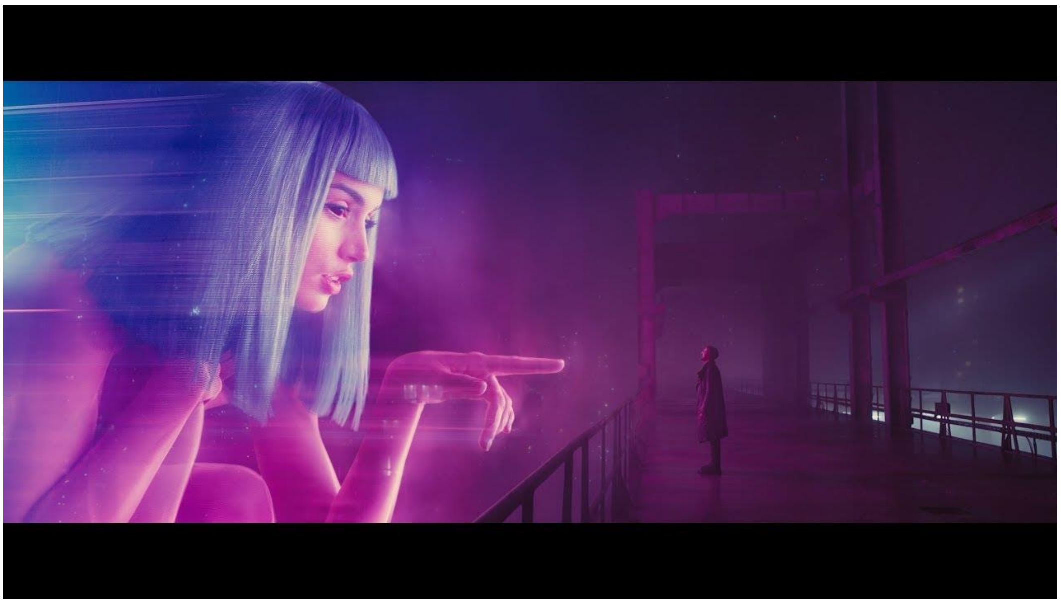


Figure 3: Blade Runner 2049 (2017) screenshot

Figure 4: Ready Player One (2018) screenshot

3.3. <u>Mechanics</u>

It pretends to be an immersive story where the player is not guided by an interface that tell him what to do. In Contact, you will be guided by the environment, and you will have to solve the time anomaly hearing sounds and looking for hints placed all over the cafeteria.

The most important mechanic of the game is to explore the game space that is offered to the player, and turn temporary space around, with the main goal of connecting the two dimensions.

To reach the perfect immersion, the game camera is in first-person perspective inside a 3D world; that is because technically, there are real time scenes swap made via Z-Buffer, and the result looks great if you see the world changing at your feet.

Because of being a PC video game, the player must have two simple peripherals, a keyboard and a mouse.

As it has been mentioned previously, the game is formed by several puzzles, but they are not explicitly shown to the player, and therefore the game is guided by the hints and inventory objects with the end to find out the way to pass the puzzle.

One of my favourite story plot is time travelling. Today there are a lot of video games that implement temporary jumps, one of the best feelings is what you feel when in a video game you visit the same place in two different times. This is seen in a famous title called *Dishonored 2* (Arkane Studios, 2016), where you can use a device to visit one place in different times (Figure 5).

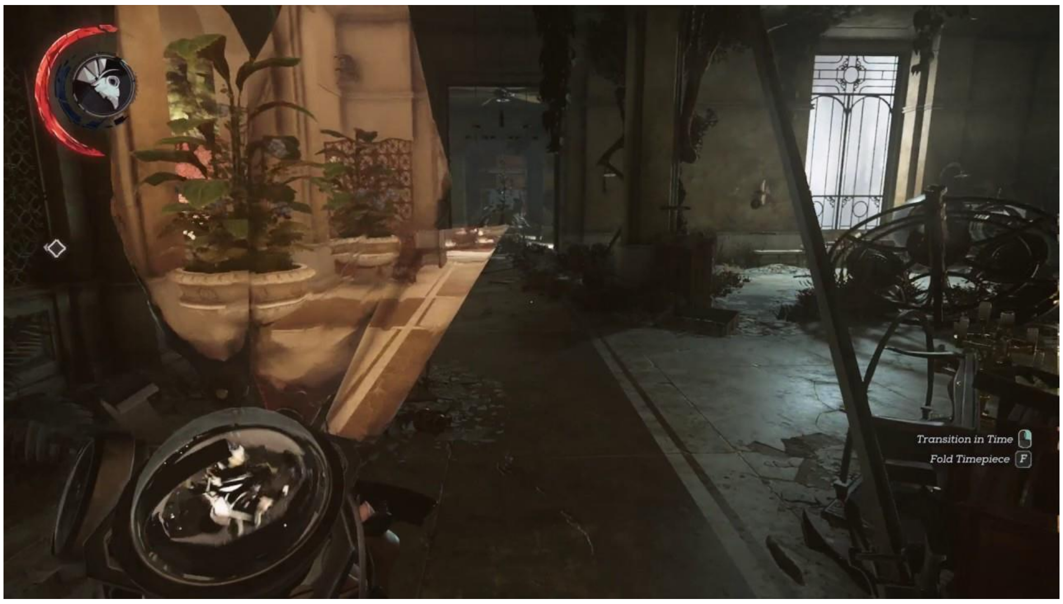


Figure 5: Time travelling mechanic in Dishonored 2

With this concept in mind, the project supervisor provided me a good starting point to make screen space effects based on scene depth information [Chyr W, 2013].

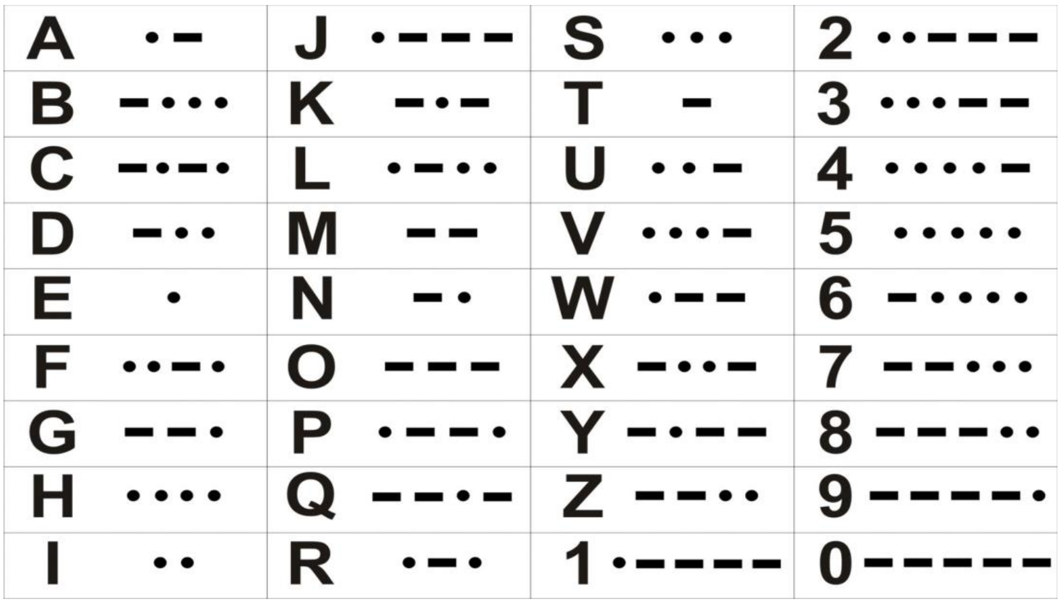
Finally, once I read it, it was decided to join the depth scanning effect with the time travelling feature, contextualizing it with the game as a mechanic.

Nevertheless, this mechanic is not available from the beginning of the game, because the player needs to acquire the object that permits time travelling by passing several puzzles.

These puzzles are not a mechanic by themselves, by contrast, the player needs to explore the environment searching evidences that carry him to discover the way to pass each puzzle.

The desired goal was making a demo where the player could investigate the whole place without being guided by a cinematic or a tutorial, just knowing the main controls of the game, but it does not mean that there are not elements in the screen in order to give feedback to the player.

After the brainstorming period determining which puzzles would be developed, it was decided to make them related with binary codes, Morse translations (Figure 6) and environment interaction, taking into account the references of typical scape room games.



A	•−	J	•−−−	S	•••	2	••−−−
B	−•••	K	−•−	T	−	3	•••−−
C	−•−•	L	•−••	U	••−	4	••••−
D	−••	M	−−	V	•••−	5	•••••
E	•	N	−•	W	•−−	6	−••••
F	••−•	O	−−−	X	−••−	7	−−•••
G	−−•	P	•−−•	Y	−•−−	8	−−−••
H	••••	Q	−−•−	Z	−−••	9	−−−−•
I	••	R	•−•	1	•−−−−	0	−−−−−

Figure 6: Morse alphabet

3.4. <u>Visual Style</u>

As it was mentioned previously in this document, one of the main goals is that the game has to be immersive at well as historically believable. This last premise applies to every prop in the 80's cafeteria, for obvious reasons; but it disappears in the cyberpunk environment, bringing me the opportunity to design invented assets from my own imagination.

Also, it is important to think and specify the aspect of the assets. In some cases, where you need to develop a fantasy video game, you can consider the possibility of making low poly assets with unrealistic forms and textures, which could be beautiful as it is demonstrated in a lot of games, but in my case, this could not fit the project objectives.

Thus, it was decided to make realistic 3D assets, increasing workflow time due to PBR materials creation. Additionally, in order to achieve that these assets generate a scene with high quality, the lighting of the environments has to be realistic. In video games, this is always a hard task, because the action must be smooth at the same time the materials behave properly. Here is where Light Baking (Figure 7) comes in, offering the possibility to store light influence in textures, thereby freeing GPU usage and contributing the environment to be more beautiful and realistic.

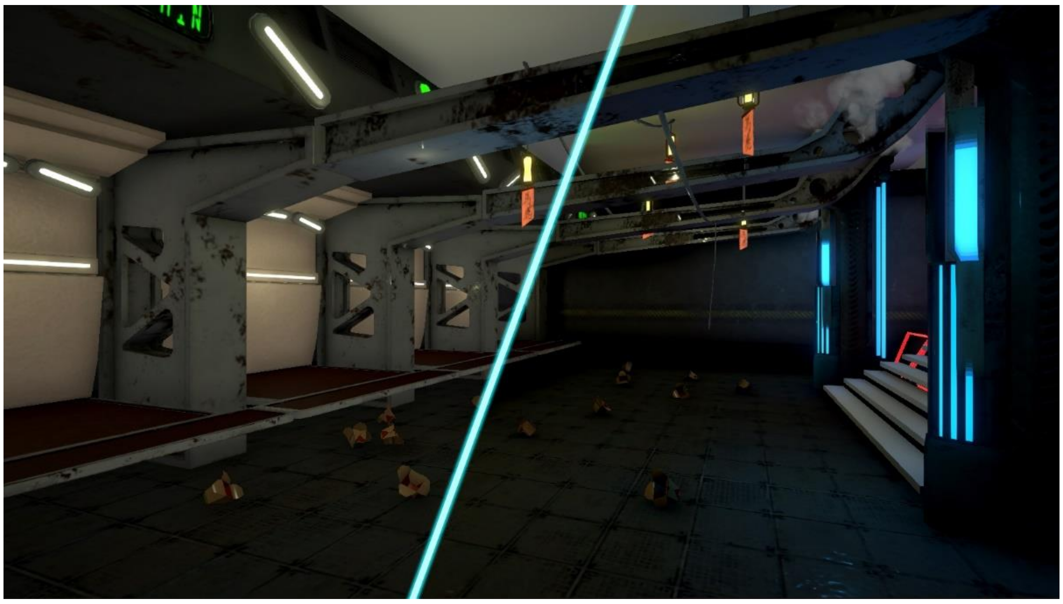


Figure 7: Non-baked lighting left / Baked lighting right

3.5. Environment

Talking about the environment, we need to distinguish between the cafeteria in the 80's, and the same place in a dystopian future.

The main place has a foundation that will be the same in both times, one big central room, one smaller room next to de big one, and two more rooms for the toilets.

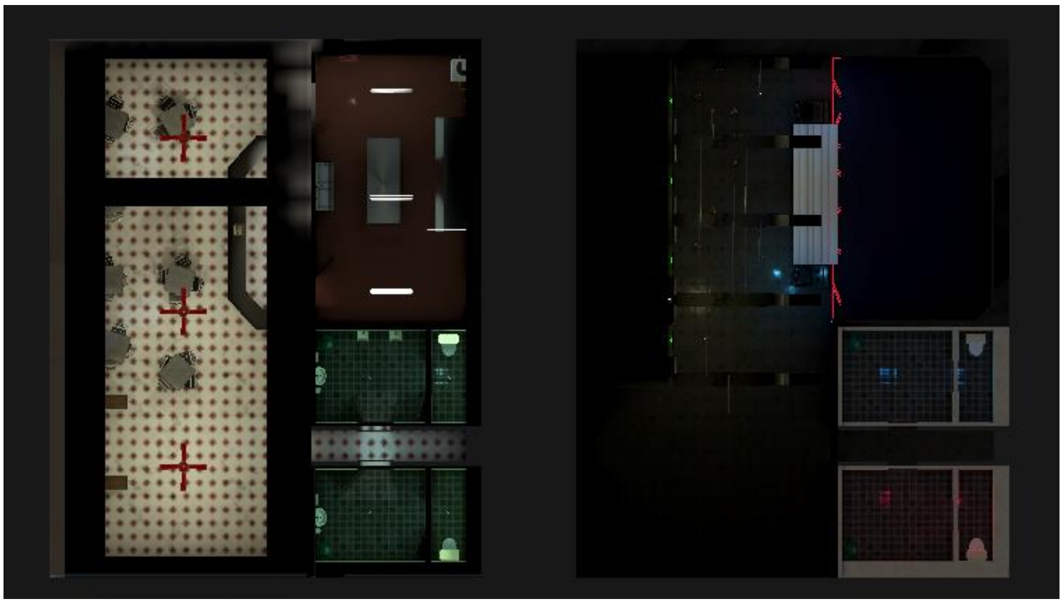


Figure 8: Environment structure - 80's cafeteria left / Dystopian future right

This structure has been designed so that I would be able to create more puzzles in order to expand the content of the game.

As we can see in Figure 8, the structure in both places is quite similar, and it has two reasons that involve design and programming decisions.

In one hand, if one of the goals of the game is that the player needs to feel inside a place but in different times, despite all construction changes and future decoration, the structure should be the same, then, it would be familiar to the player.

On the other hand, I realized when the main Z-buffer effect was developed, that if the structure of the environment in both scenes is a little bit different, it generates weird artifacts and the immersion is not maximized at all.

Once the structure was specified, an extended references research was done with the purpose of being as reliable as possible, I searched either in Google or different art webs like Artstation, looking for real places in the 80's, and good concepts done by professional artists, two good references of this can be seen in Figure 9.

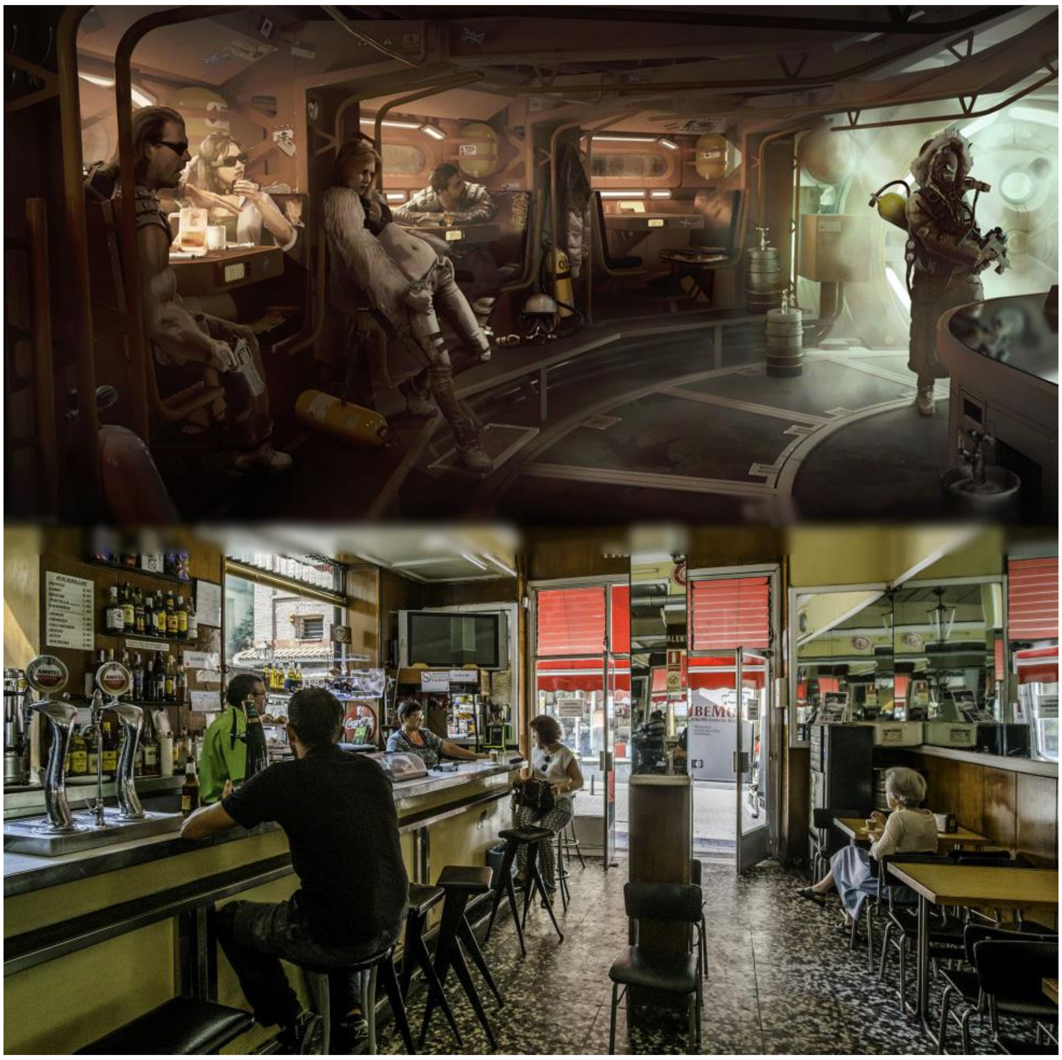


Figure 9: References taken for the project

One more thing that has to be mentioned is the light amount in both places, it was a difficult choice because the story places the character in a cafeteria at night, and the light that would enter through the window would be too low, making the environment looks dark. This was not a problem because few lights placed properly could fix it.

However, for the futuristic environment, the darkness could be a possibility, the smoke and neon lights will absorb the cities, making interior places look sad and synthetic; this is the main reason why the second environment looks darker.

4. Development

4.1. Tools in Unity

Before explaining the art and programming of the project, it is important to enumerate and describe every single tool that was decided to use before starting to develop the game, as well as the first changes in the project settings to improve the visuals.

Unity provides the user with basic tools like standard shaders for the materials, or a light mapper for baking the scene lighting, but if you want the project to look better, it is important to download few plugins that make the environment more beautiful.

4.1.1. Project settings

One of the first changes that was done for the project was changing the rendering color space from Gamma to Linear. This changes the light impact on the objects making the materials look smoother, with more defined shadows, as it can be seen in Figure 10, the assets material looks flatter in Gamma Space (right).

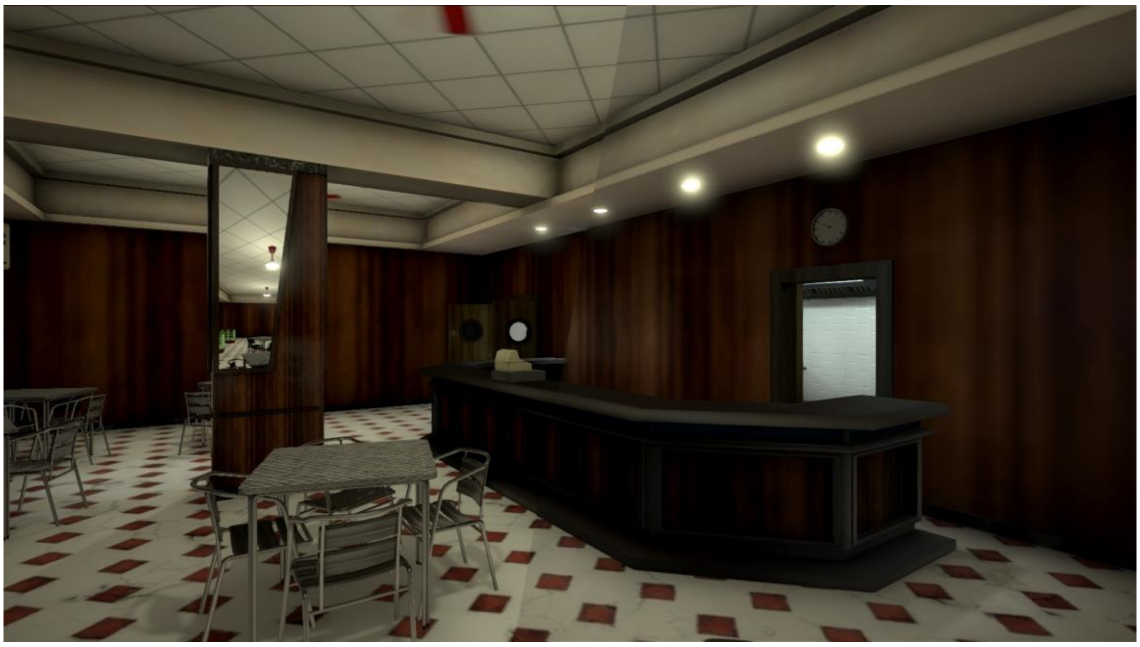


Figure 10: Difference between Linear Space (left) and Gamma Space (right)

The next thing it is important to change is disable the default antialiasing that comes with Unity, and this is because for this project, a post processing antialiasing is used just because it allows me to change the preset of the technique, letting me decide if it is better to use a high quality preset or the performance one.

Finally, before starting the project work, the last thing is needed to change is the rendering path of the cameras, from Forward, to Deferred.

If you use Forward rendering path, a few lights could affect the scene objects, appearing bad specular highlights and plane surfaces.

In contrast, if the cameras use Deferred rendering path, all the lights will affect the objects of the scene, making them more realistic.

We can see it in Figure 11, where the floor and walls with Forward are completely flat.

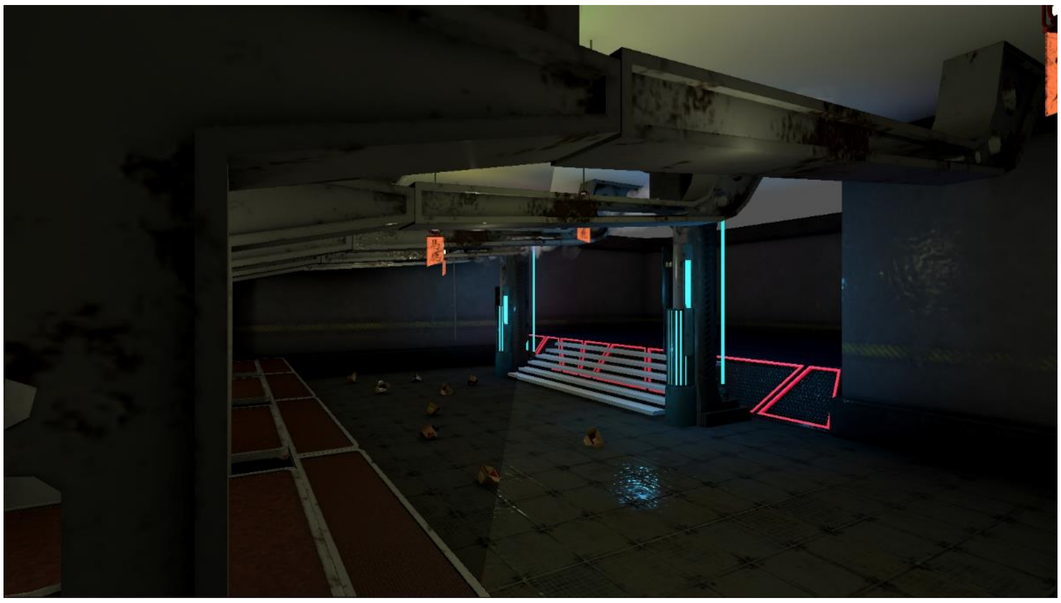


Figure 11: Difference between Forward rendering path (right) and Deferred

I also clarify that these changes decrease frame rate, and maybe during the project development they could be set as default again. In addition, there are few more changes that will be explained in different sections of this document.

4.1.2. <u>Standard Shader (Roughness setup)</u>

One of the most important improvements Unity included in previous versions was a standard shader that allows us to make Physically Based Rendering (PBR) materials. This shader uses realistic lighting models to accurately represent materials as we see them in real life.

In Unity, we can distinguish several textures that compose a single material:

- Albedo map: this map stores the color information of the object (Figure 12).

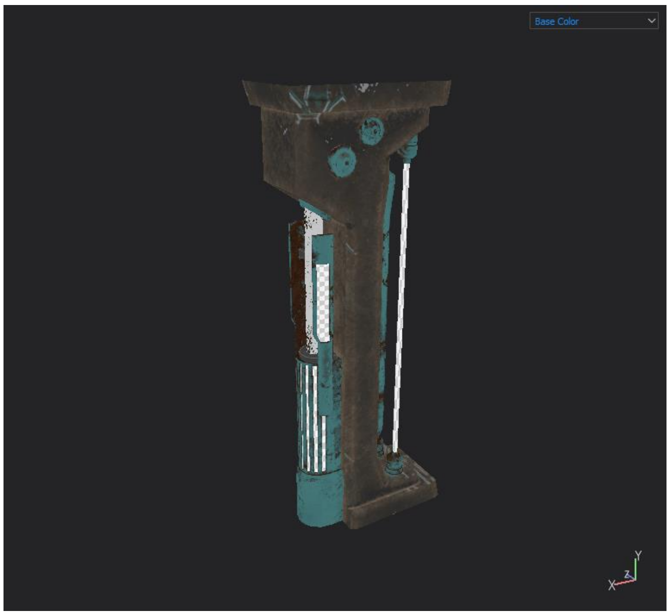


Figure 12: Albedo map of the column

- Metallic map: it manages the reflectivity of the surface (Figure 13).

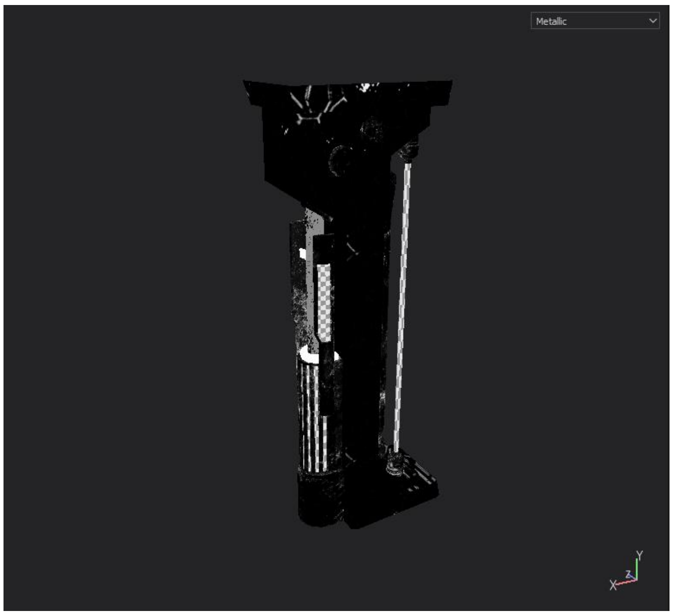


Figure 13: Metallic map of the column

- Roughness map: this one controls the smoothness of the surface (Figure 14).

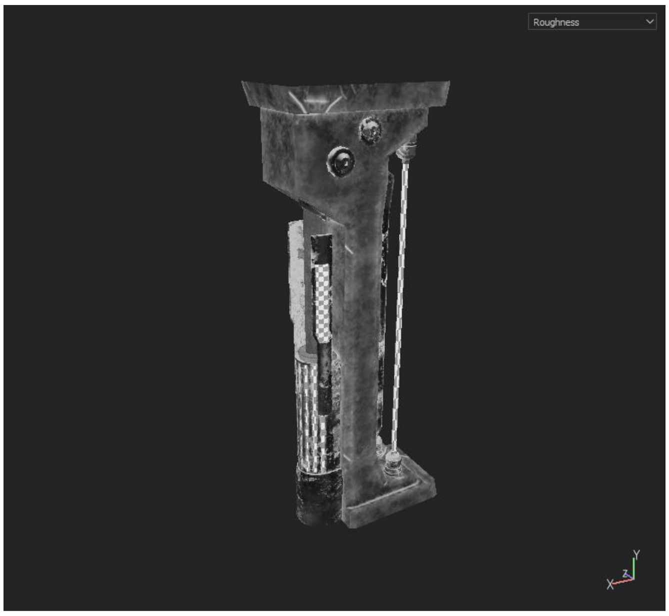


Figure 14: Roughness map of the column

- Normal map: this is a very important map because it adds a lot of detail to the mesh without increasing the polygons, just telling the light how it must behave (Figure 15).

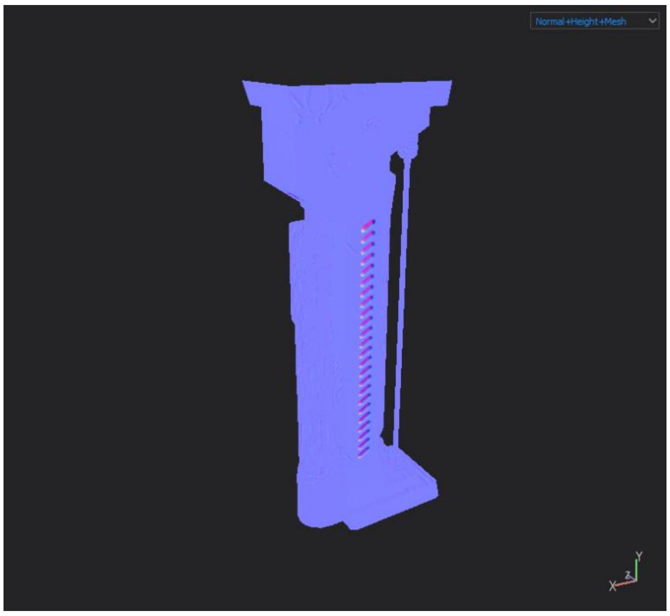


Figure 15: Normal map of the column

- Ambient Occlusion map: in order to avoid ambient occlusion real time calculations, it is possible to bake that information and apply it to the material (Figure 16).

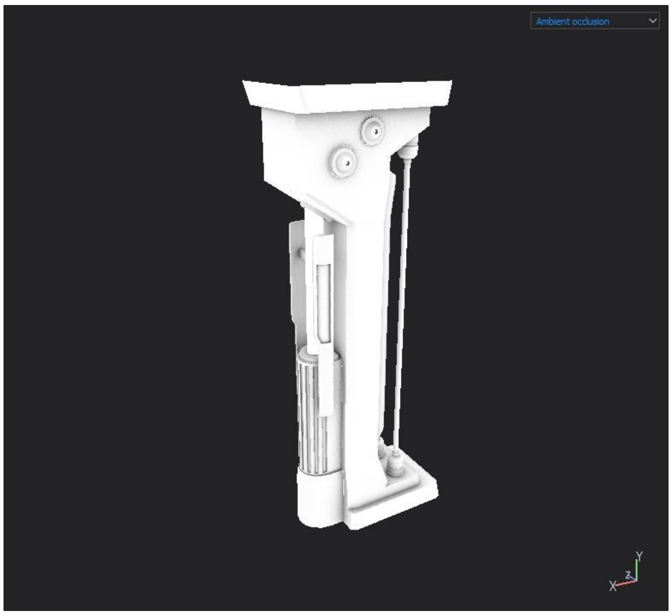


Figure 16: Baked ambient occlusion of the column

Finally, if an emission map is applied, we will be able to apply the material to the mesh and achieve quite good results, as we can see in Figure 17.

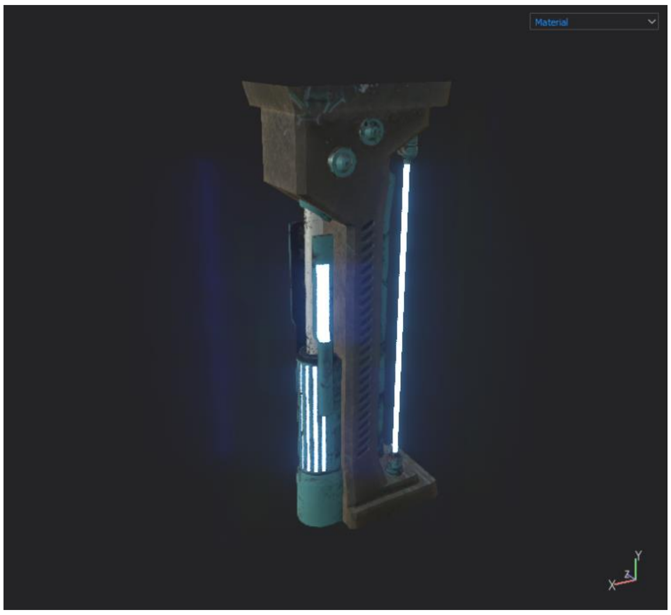


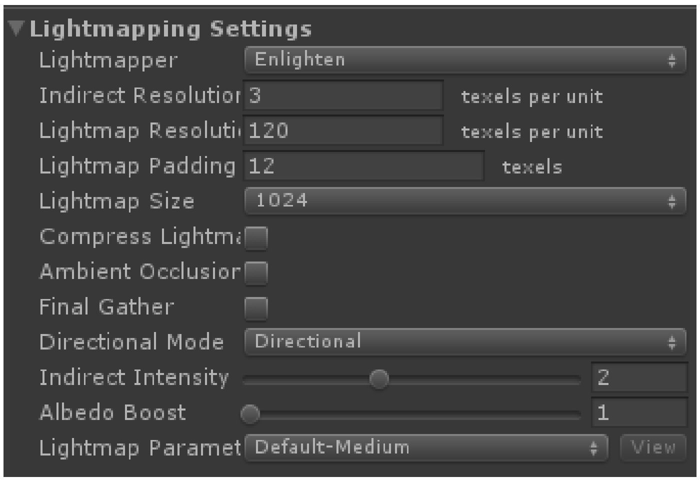
Figure 17: Final material applied to the column

4.1.3. Lighting

In Unity, it is possible to set up real time lighting, which is valid if the main goal of the project is not reaching high quality graphics, but in this case, the most important aspect of the game is creating an immersive world for the player. If just real time lights are placed inside the game, everything behave as simple lighting model says, if one fragment is accessible by a light source, then this fragment would be lit, but in real life this does not work like this, when a light source emits photons, they bounce in a lot of points lighting them.

With that objective in mind, there are few options to achieve it. The first solution is just baking the light produced by emissive materials into light maps. This method behaves similar to real life light sources by launching rays from light sources and draw the information in textures called light maps.

In Unity is possible to set up light mapper parameters (Enlighten light mapper in my case) such as light map resolution, compression or indirect rays' intensity (Figure 18).

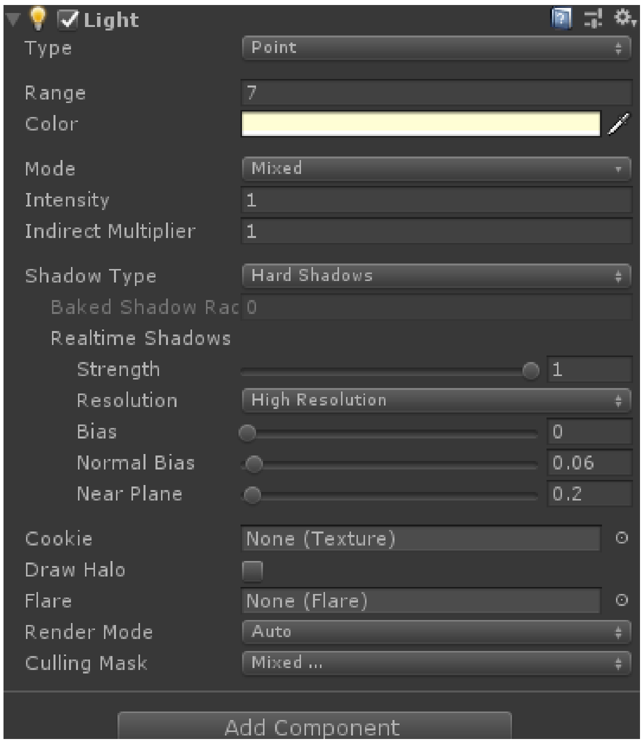


Lightmapping Settings

Lightmapper	Enlighten	
Indirect Resolutior	3	texels per unit
Lightmap Resoluti	120	texels per unit
Lightmap Padding	12	texels
Lightmap Size	1024	
Compress Lightm	☐	
Ambient Occlusior	☐	
Final Gather	☐	
Directional Mode	Directional	
Indirect Intensity	2	
Albedo Boost	1	
Lightmap Paramet	Default-Medium	View

Figure 18: Lightmapping Settings

But there is one more complete solution, this is by using mixed global illumination with real time lighting active. With this method, in addition to emissive materials, every single light in the scene set as mixed light will contribute to indirect lighting of the scene, creating more realistic shadows and also lighting inaccessible places for real time lights (Figure 19).



Light

Type	Point
Range	7
Color	
Mode	Mixed
Intensity	1
Indirect Multiplier	1
Shadow Type	Hard Shadows
Baked Shadow Rac	0
Realtime Shadows	
Strength	1
Resolution	High Resolution
Bias	0
Normal Bias	0.06
Near Plane	0.2
Cookie	None (Texture)
Draw Halo	☐
Flare	None (Flare)
Render Mode	Auto
Culling Mask	Mixed ...

Figure 19: Light source set as mixed

Also, there is a big problem when using just baked global illumination because it sometimes generates noise in light maps, and there are two more ways to fix that, increasing light map resolution and increasing indirect intensity, or simply using real time global illumination.

4.1.4. Post Processing Stack

One of the most important assets that must be in the project is Post Processing Stack, a bunch of camera effects that largely improve game graphics. However, it also reduces drastically the frame rate making the game unplayable in some PCs.

In this project, the effects that have been used are:

- Temporal Antialiasing: it is the expensive technique to smooth edges provided by the package, it accumulates frames in a buffer to achieve good results, as we can see in Figure 20.

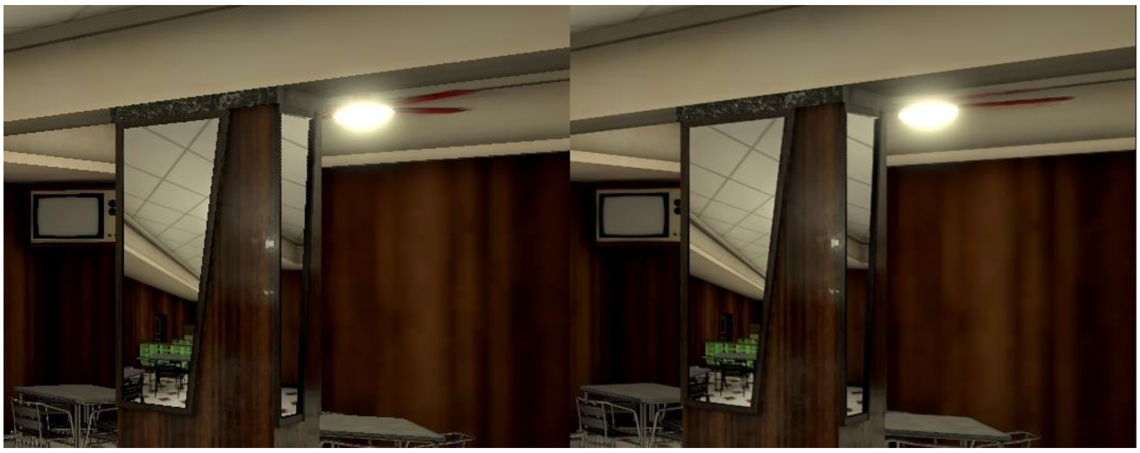


Figure 20: No Antialiasing (left) and Temporal Antialiasing (right)

- Ambient Occlusion: this effect achieves better results than baking the ambient occlusion, but it also requires a lot of calculations. The effect is used just as a supplement of baked data. In Figure 21 is remarkable the ambient occlusion effect in the right screenshot, it is so smooth because if the effect intensity is very high, it does not seem realistic.

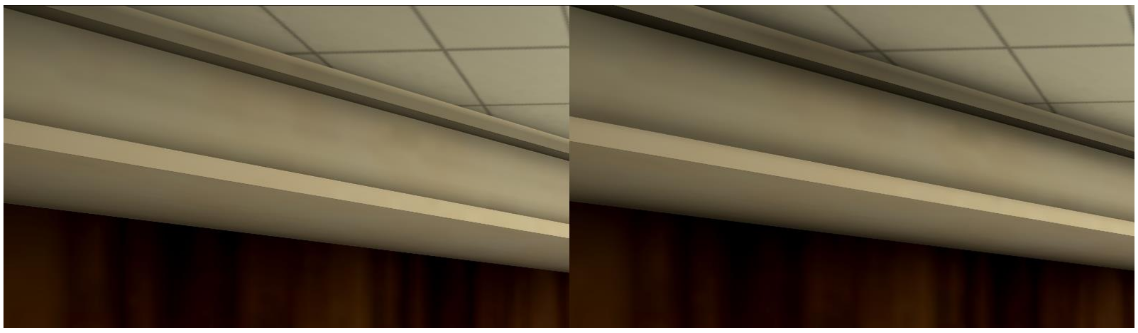


Figure 21: Ambient Occlusion disabled (left) and Ambient Occlusion enabled (right)

- Motion Blur: the motion blur effect tries to simulate what happens when you turn your eyes quickly, it blurs the middle frames during camera rotation (Figure 22).

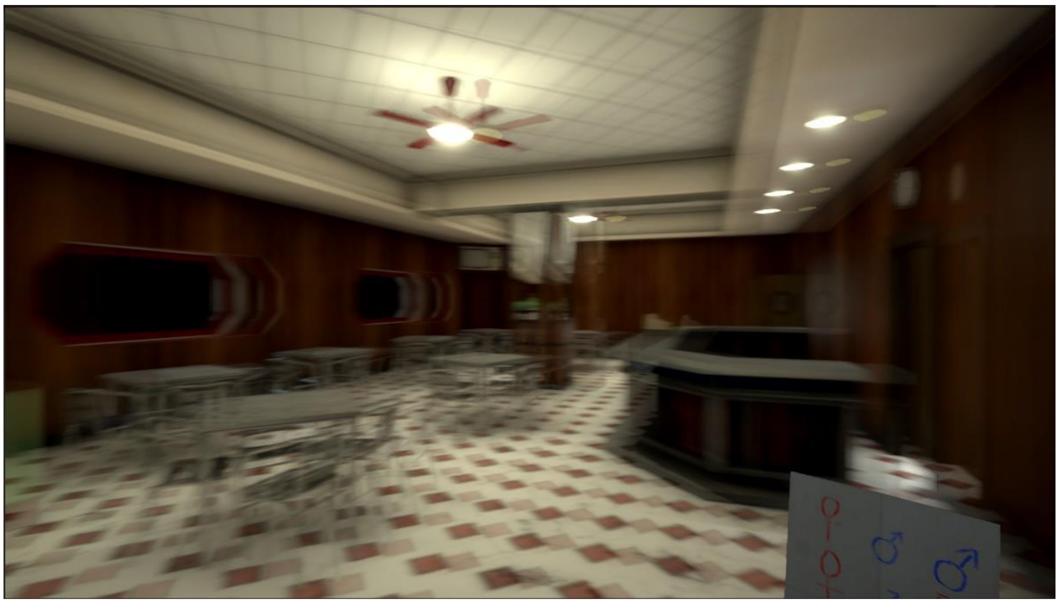


Figure 22: Motion Blur in action

- Bloom: if there are emissive materials, it is very important to highlight them using this bloom effect (Figure 23).

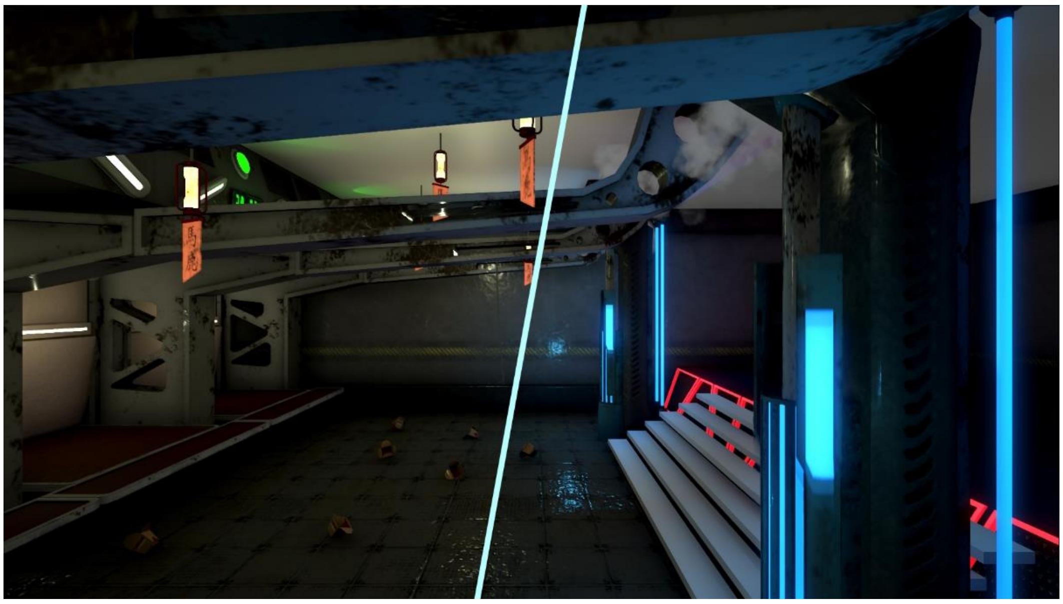


Figure 23: Bloom effect disabled (left) and Bloom effect enabled (right)

- Vignette: the last effect in this project is the vignette one, it places a smooth black border around the screen, increasing player focus feeling (Figure 24).

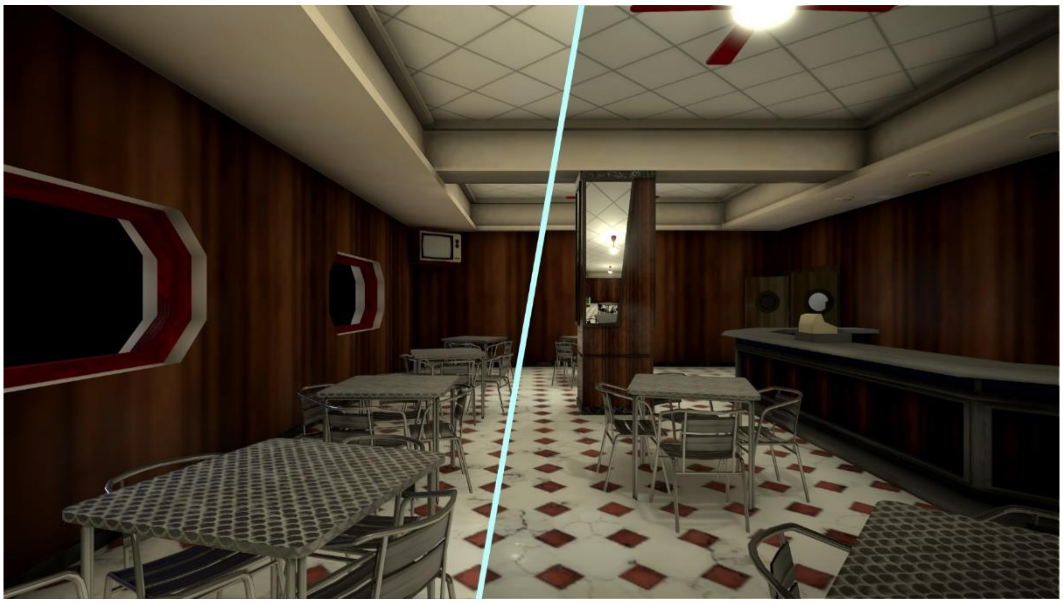


Figure 24: Vignette effect disabled (left) and Vignette effect enabled (right)

4.1.5. TextMeshPro

Sometimes, when a default Unity text is resized, it looks blurry because it has not a lot of information of what the text must look when resized. However, with TextMeshPro plugin, Unity uses a rendering technique called Signed Distance Field, that requires a high-resolution font atlas in order to avoid the improvisation of pixels by the engine. This quality improvement can be seen in Figure 25.

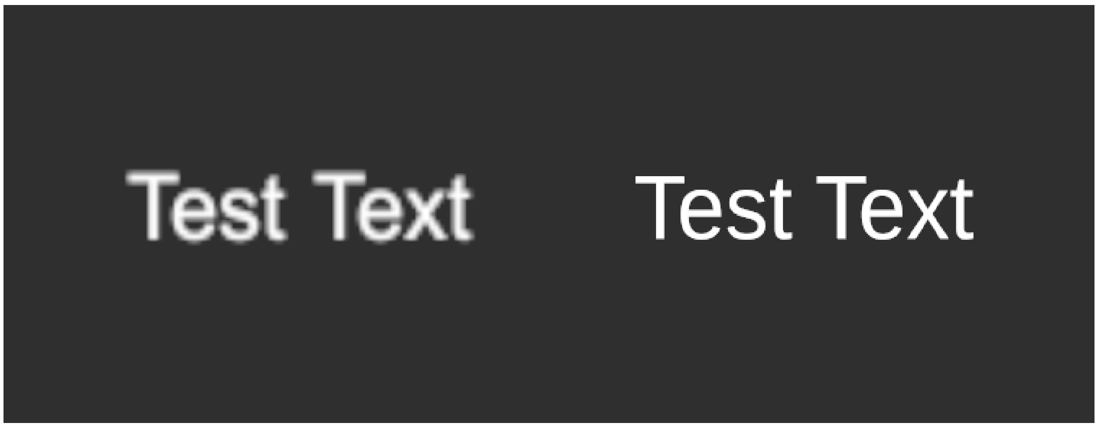


Figure 25: Default text (left) and TextMeshPro (right) when resized

Furthermore, if you need one specific font, it provides you with a font asset creator that makes it able to use in the project.

4.2. Art development

In this section, everything related with art in this project is going to be explained, since 3D models workflow to the sound effects included inside de game. Nonetheless, there are few technical optimizations that are going to be explained as well.

4.2.1. 3D Models

One premise has been taken in this project is that every single 3D asset would be designed and created by myself, and the final result is around 70 different props prepared to be placed in any project (Figure 26). This project is meant to be a personal assets repository where in a future I will be able to come again and take the assets I need.

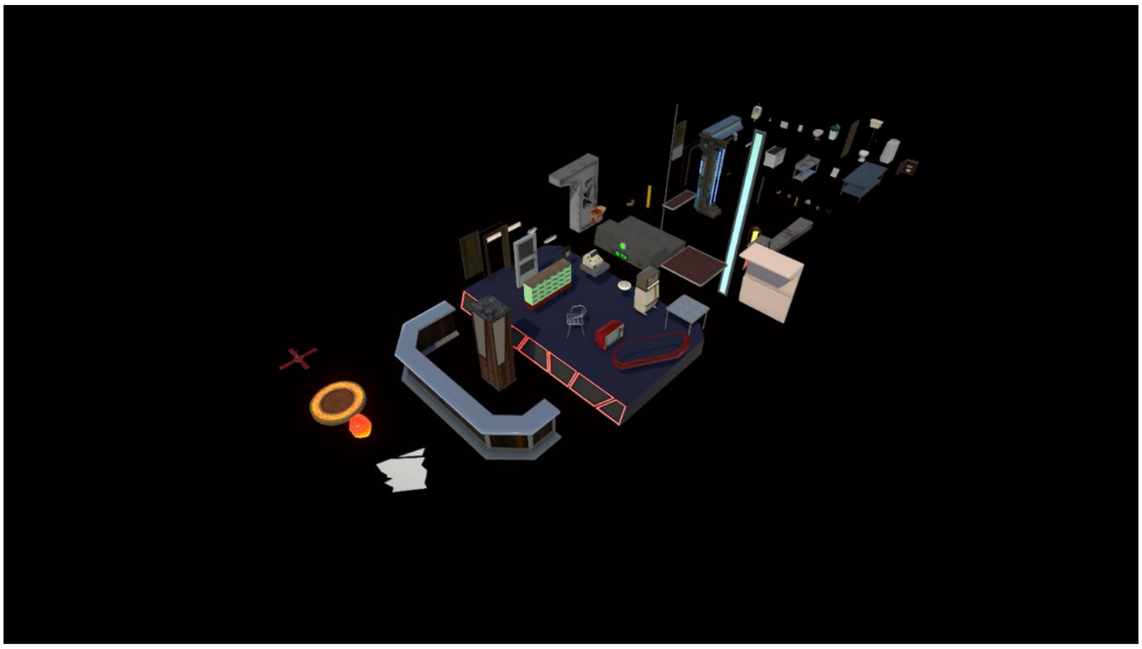


Figure 26: Assets used in the project

The chosen tool for creating all 3D models of this project is Blender, a free open source software. This decision was taken because my workflow does not include high poly models, and programs like Zbrush or Maya are not required.

As it has not been a need of making high poly models, workflow's time decreased a lot because props do not have to be retopologized.

The first step in this workflow is doing a block out of the scene inside Unity, just to have the skeleton of the environment and figure out how the final result should look. This step is sometimes forgotten because it is a little bit tedious and maybe you can think it has not sense if you are going to model the assets properly in order to place them in a block place.

When this is done, it is time to let the imagination run and start modelling all the assets. In Blender, once a medium poly asset is created, it is important to unwrap the UVs considering the different materials of the object; the objective is preparing an UV map in which every different part of the model is perfectly visible and distinguishable by its own material (Figure 27). This will help workflow's optimization in a future step.

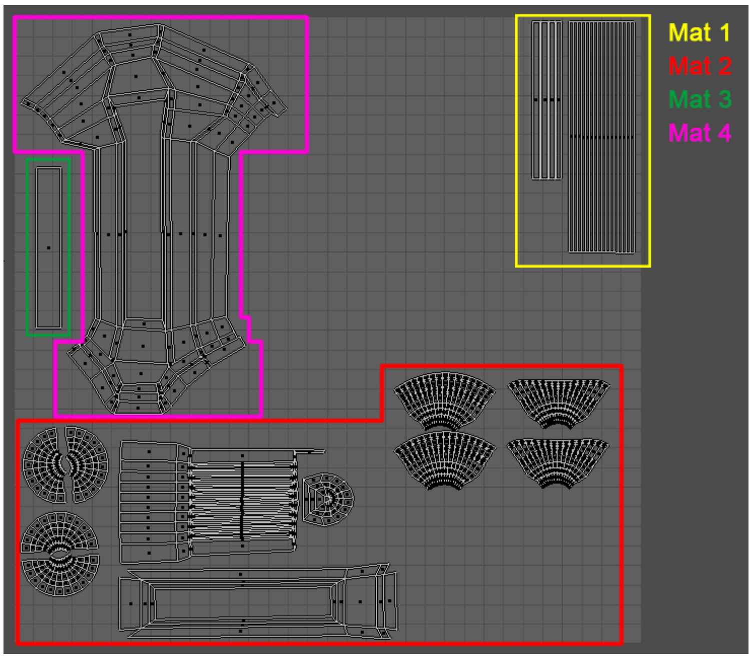


Figure 27: UV map of one of the project assets

Next step is placed in a different software, Substance Painter, where realistic materials can be applied to the models, it allows baking ambient occlusion maps as well. Once every material has been chosen, it is possible to export a maps package ready to be placed in Unity. But there is a problem that can affect game's performance due to Substance Painter exports five maps for every material that has been set in Blender. In this case five 1024x1024 textures for five different materials; if the texture size in memory is set in 100 KB on average, the whole texture package sums up to a total of 5 MB which is very large for just one asset.

Here is where Photoshop comes in; in this software it is possible to join different textures with the same information (base color, normal, metallic…) in order to make just five textures that hold all the materials, making a total of 1 MB for one entire asset (Figure 28).

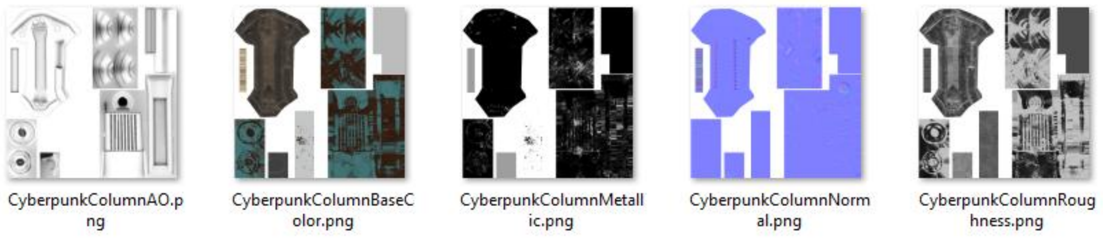
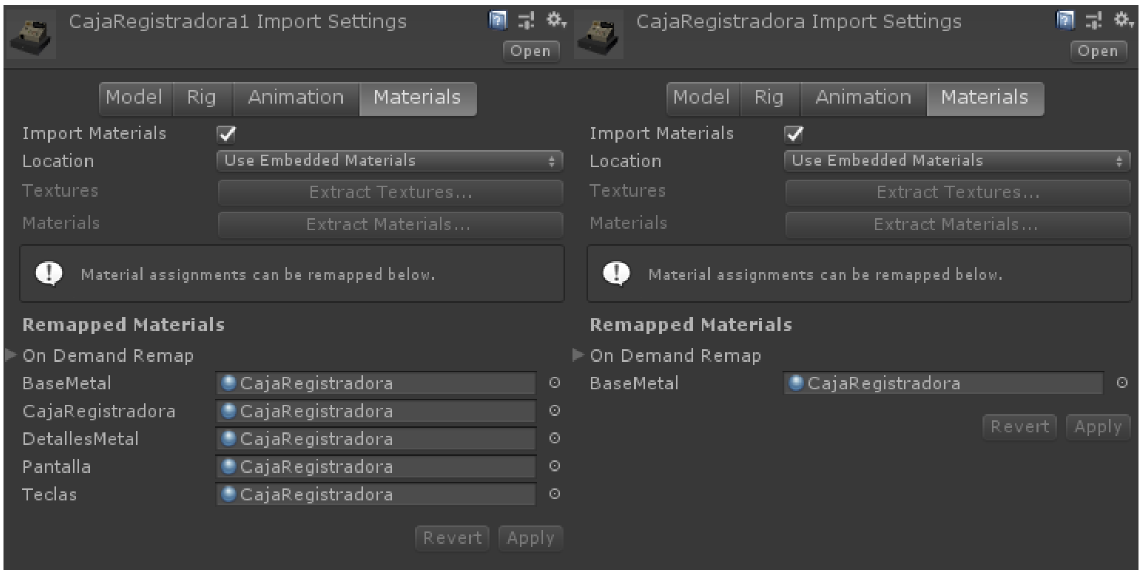

CyberpunkColumnAO.png
CyberpunkColumnBaseColor.png
CyberpunkColumnMetallic.png
CyberpunkColumnNormal.png
CyberpunkColumnRoughness.png

Figure 28: Asset final texture package

Last thing is important to know before passing to a different asset is that the .FBX file exported in first step has the number of materials applied in Blender, in this case five materials, and it can produce a significant performance issue later in Unity.

The Unity asset requires of five materials despite being the same ones, and this causes the GPU to have to process five instances with same information, which for one asset could be reasonable but not for a bunch of them.

In order to solve this problem, it is only needed to re-export the model just with one material attached, and Unity would only recognize that material (Figure 29).



CajaRegistradora1 Import Settings

Model | Rig | Animation | Materials

Import Materials ✓
Location: Use Embedded Materials
Textures: Extract Textures...
Materials: Extract Materials...

Material assignments can be remapped below.

Remapped Materials
On Demand Remap

BaseMetal	CajaRegistradora
CajaRegistradora	CajaRegistradora
DetallesMetal	CajaRegistradora
Pantalla	CajaRegistradora
Teclas	CajaRegistradora

CajaRegistradora Import Settings

Model | Rig | Animation | Materials

Import Materials ✓
Location: Use Embedded Materials
Textures: Extract Textures...
Materials: Extract Materials...

Material assignments can be remapped below.

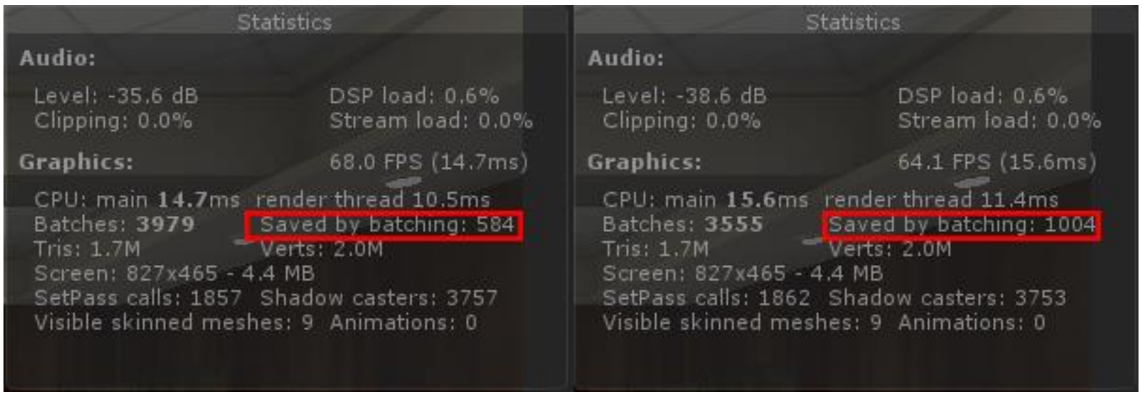
Remapped Materials
On Demand Remap

BaseMetal	CajaRegistradora

Figure 29: Not optimized materials (left) and Optimized materials (right)

Finally, there is one more step inside Unity, in case of the new asset would have been placed a lot of times in the same scene, standard material offers the opportunity to check a toggle that enables GPU instancing. This is so important because it allows the GPU to draw multiple object instances using few draw calls, and it increases game performance at great scale.

The difference is visible in Figure 30, showing the same scene from the same camera, but with chairs and tables materials set up with and without GPU instancing.



```
Statistics
Audio:
  Level: -35.6 dB              DSP load: 0.6%
  Clipping: 0.0%               Stream load: 0.0%
Graphics:                      68.0 FPS (14.7ms)
  CPU: main 14.7ms  render thread 10.5ms
  Batches: 3979     Saved by batching: 584
  Tris: 1.7M        Verts: 2.0M
  Screen: 827x465 - 4.4 MB
  SetPass calls: 1857  Shadow casters: 3757
  Visible skinned meshes: 9  Animations: 0
```

```
Statistics
Audio:
  Level: -38.6 dB              DSP load: 0.6%
  Clipping: 0.0%               Stream load: 0.0%
Graphics:                      64.1 FPS (15.6ms)
  CPU: main 15.6ms  render thread 11.4ms
  Batches: 3555     Saved by batching: 1004
  Tris: 1.7M        Verts: 2.0M
  Screen: 827x465 - 4.4 MB
  SetPass calls: 1862  Shadow casters: 3753
  Visible skinned meshes: 9  Animations: 0
```

Figure 30: GPU instancing disabled (left) and GPU instancing enabled (right)

The research of historical props combined with the explained workflow has its results in examples like in Figure 31, where props as 80’s tobacco machine or cash register are observable.

Figure 31: Examples of 80's assets

4.3. Technical development

Finally, a total of three puzzles have been developed for the demo, but the project is prepared perfectly for future additional puzzles.

Thus, there are three scenes in total, one holds 80’s cafeteria environment, one holds futuristic environment, and the last one is done for the menus.

In this section, every technical feature like menus, shaders, scripts are going to be explained in detail, using diagrams for a better understanding.

4.3.1. Shaders

The first point of this section is shader development, and this has not been chosen by chance, because here, the main goal of making a Z-buffer screen effect is explained, and it has been very relevant for the game good looking.

4.3.1.1. Ring Pass Effect

Final effect consists in one ring that passes back and forth changing what is seen by the camera. In Figure 32 the ring is active and it is changing from past to future time environment.

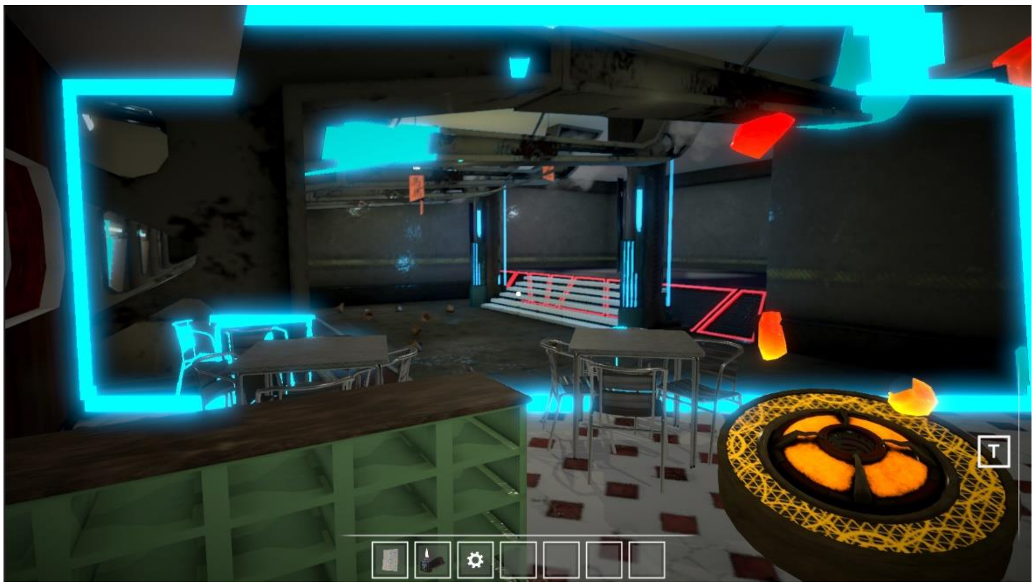


Figure 32: Ring Pass effect running

Firstly, there are a few properties (Figure 33) that need to be declared for a future use in fragment function:

- _MainTex: 2D texture that holds the current scene frames.
- _AnotherTex: 2D texture that holds the second scene frames.

These previous two textures will be used to paint zones whether the ring has passed or not.

- _RingWidth: a float that gives right size.
- _RingPassTimeLength: a float that stablishes the ring’s passing duration.
- _RingColor: this property stablishes ring’s color.
- _RingEmissionIntensity: in order to achieve a lighting effect, a value is declared to be applied on the final color and make it emissive.

Also, the shader needs one more variable that triggers the effect when an external script command it. This variable is _RunRingPass, and it is used as a Boolean value.

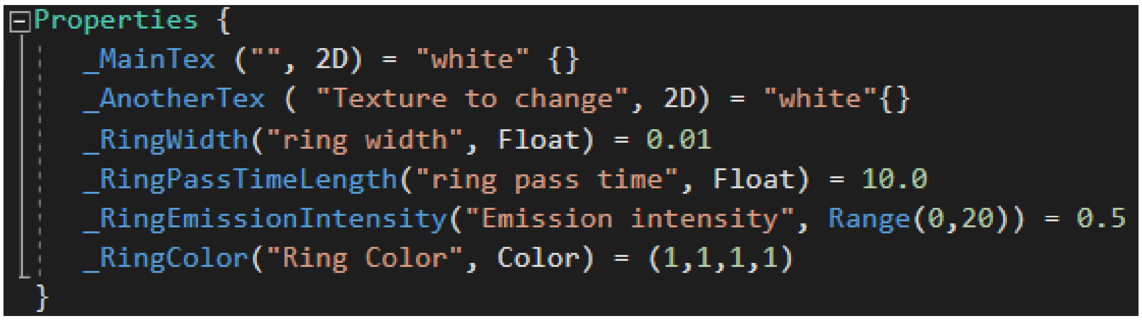
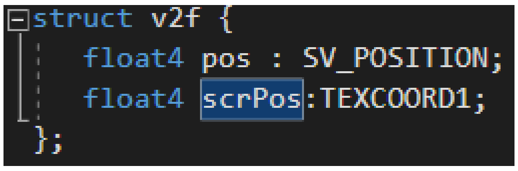

```
Properties {
    _MainTex ("", 2D) = "white" {}
    _AnotherTex ( "Texture to change", 2D) = "white"{}
    _RingWidth("ring width", Float) = 0.01
    _RingPassTimeLength("ring pass time", Float) = 10.0
    _RingEmissionIntensity("Emission intensity", Range(0,20)) = 0.5
    _RingColor("Ring Color", Color) = (1,1,1,1)
}
```

Figure 33: Ring Pass shader properties

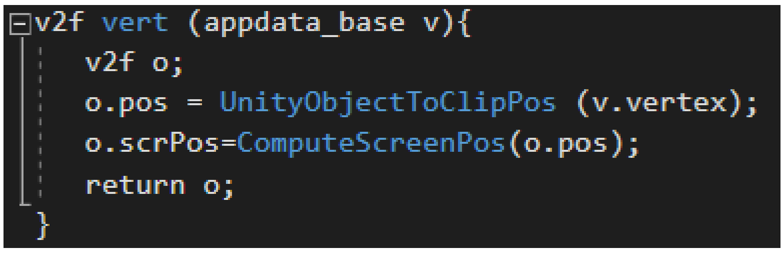
Now, before scripting fragment function, it is necessary to indicate the effect where it needs to be attached, and this is done in vertex function, specifically using a vertex data, a struct composed by basic information as the vertex position or a texture where the effect will affect (Figure 34), in any other cases the struct might have few more information.



```
struct v2f {
    float4 pos : SV_POSITION;
    float4 scrPos:TEXCOORD1;
};
```

Figure 34: Vertex data structure

In vertex function, a vertex data object is created specifying that the effect should take place in screen position, then the fragment function could know the location of the pixels that need to be changed. In Figure 35 is possible to see that screen position is attached to a vertex data object, but it usually does not look like this, for example, if the effect should affect a cube, then vertex data would hold that cube UVs.



```
v2f vert (appdata_base v){
    v2f o;
    o.pos = UnityObjectToClipPos (v.vertex);
    o.scrPos=ComputeScreenPos(o.pos);
    return o;
}
```

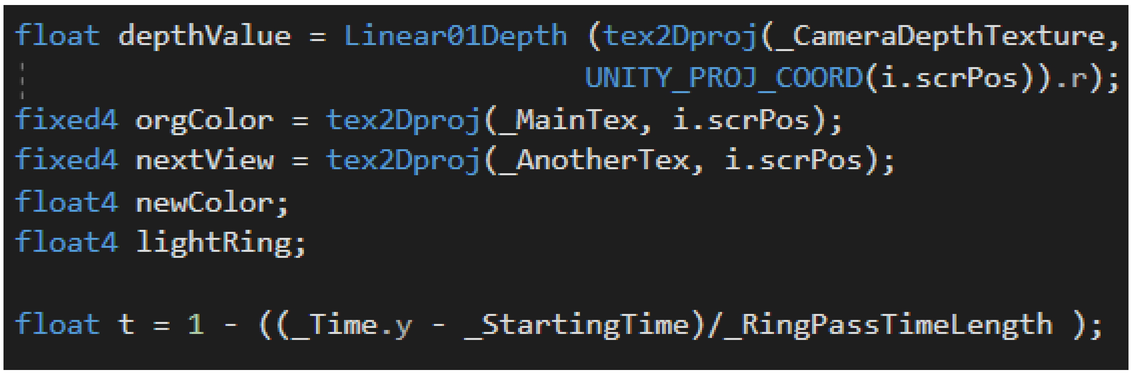
Figure 35: Vertex function of Ring Pass shader

Now, with the necessary properties and vertex information, it is possible to script the fragment shader.

It generally works by checking whether the ring has passed from the scene depth value, in which case the color would be the next scene pixel color; or not, in which case the resulting color would be the original scene. But then, all of this process is going to be explained properly.

First thing to do is initialize necessary variables (Figure 36):

- depthValue: it takes depth information from camera depth texture (camera setup will be explained in other section of this document).
- orgColor: it holds the color of the original scene render texture.
- nextView: it holds the color of the destination scene render texture.
- newColor: the color will be returned if the ring has passed a specific depth value.
- lightRing: it holds ring’s color.
- t: this variable works as an iterator through the depth value along time.



```
float depthValue = Linear01Depth (tex2Dproj(_CameraDepthTexture,
                                     UNITY_PROJ_COORD(i.scrPos)).r);
fixed4 orgColor = tex2Dproj(_MainTex, i.scrPos);
fixed4 nextView = tex2Dproj(_AnotherTex, i.scrPos);
float4 newColor;
float4 lightRing;

float t = 1 - ((_Time.y - _StartingTime)/_RingPassTimeLength );
```

Figure 36: Needed variables in Ring Pass fragment shader

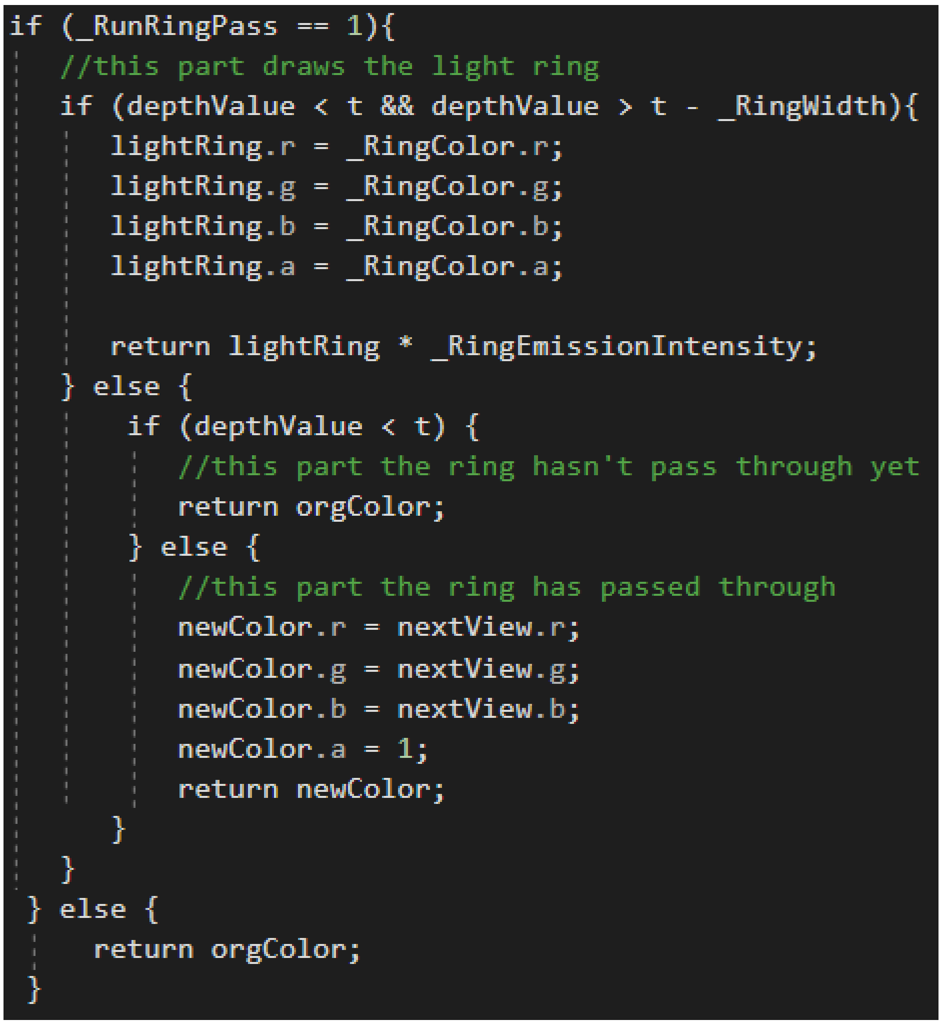
Now that everything has been declared, it is time to script the logic of the shader.

First check is evaluating if the trigger has been enabled, this would start t variable iterations.

Then, it is necessary to check if the t value is inside the ring region, which would draw ring color.

Finally, if the t value has passed through a specific depth value, the resulting color would be the next view texture color, and in counterpart, final color would be the original scene texture one.

This logic could be seen perfectly in Figure 37.



```
if (_RunRingPass == 1){
   //this part draws the light ring
   if (depthValue < t && depthValue > t - _RingWidth){
      lightRing.r = _RingColor.r;
      lightRing.g = _RingColor.g;
      lightRing.b = _RingColor.b;
      lightRing.a = _RingColor.a;

      return lightRing * _RingEmissionIntensity;
   } else {
       if (depthValue < t) {
          //this part the ring hasn't pass through yet
          return orgColor;
       } else {
          //this part the ring has passed through
          newColor.r = nextView.r;
          newColor.g = nextView.g;
          newColor.b = nextView.b;
          newColor.a = 1;
          return newColor;
      }
   }
 } else {
     return orgColor;
 }
```

Figure 37: Logic inside the Ring Pass

4.3.1.2. Time Travelling Portal

One of the latest features added to project was this effect, it allows the player to see through dimensions carrying a portal that holds what would be seen if character was in the other scene, as it is visible in Figure 38.

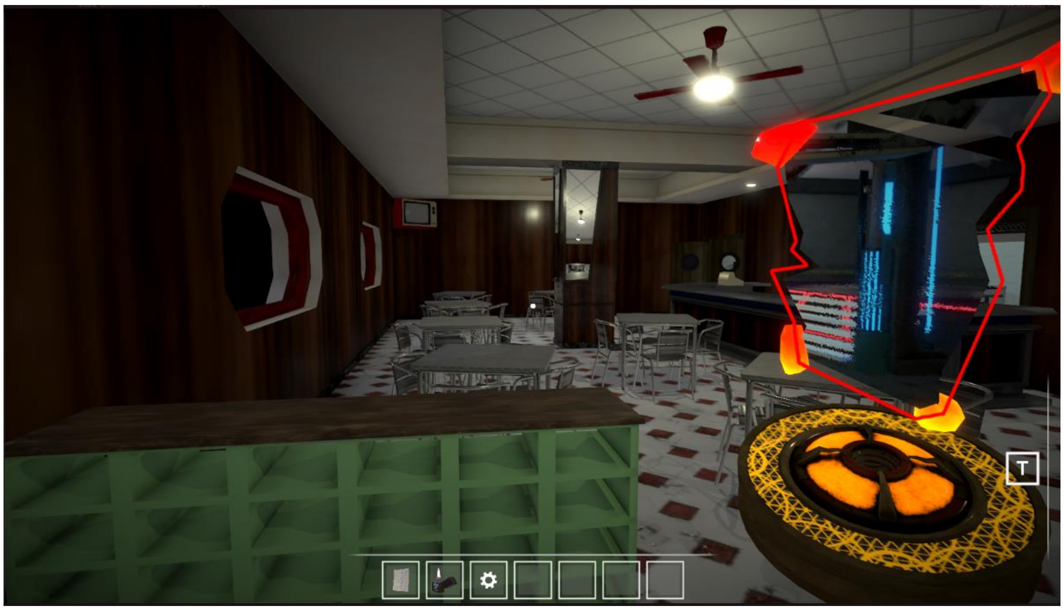
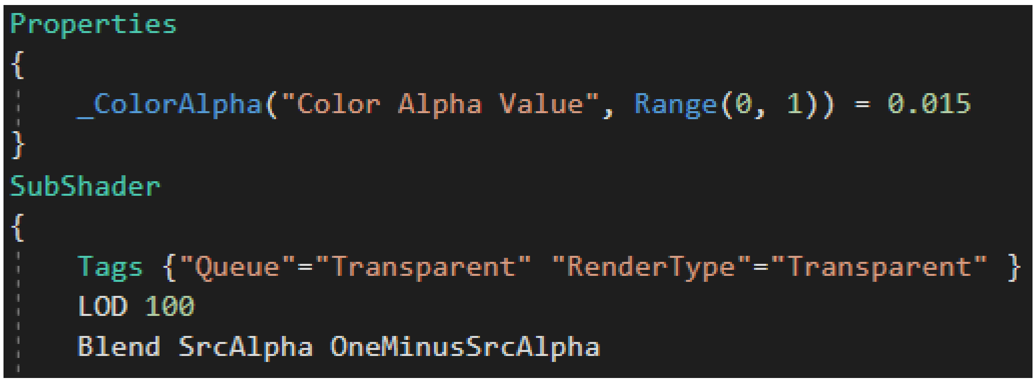


Figure 38: Time Travelling Portal effect

In this shader, there is only one property that needs to be declared in order to control the final color alpha value inside an animation. Also, in order to use a transparent material, it is necessary to specify the tags and blending mode (Figure 39).

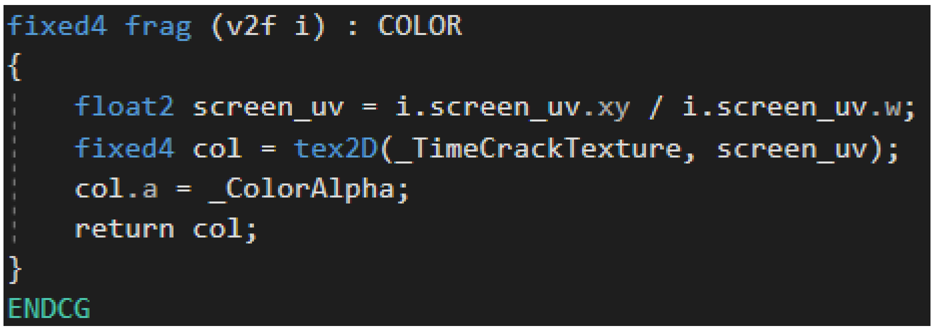


```
Properties
{
    _ColorAlpha("Color Alpha Value", Range(0, 1)) = 0.015
}
SubShader
{
    Tags {"Queue"="Transparent" "RenderType"="Transparent" }
    LOD 100
    Blend SrcAlpha OneMinusSrcAlpha
```

Figure 39: Time Travelling Portal shader transparent setup

Now, and a similar way of previous shader, vertex function creates a vertex data object and pass the screen coordinates for fragment function. But this time it is not the same process as in the previous one, and that is because the effect needs to be attached to a 3D model, and it can produce perspective problems inside the fragment function.

With this problem in mind, in fragment function is necessary to tweak the resulting UV values in order to fix the perspective problem, and this could be done by dividing the input UVs by the w component of that UVs, which stores world scaled view. After that, the resulting color would be the render texture of the camera, passed from another script, applied to calculated UVs (Figure 40).



```
fixed4 frag (v2f i) : COLOR
{
    float2 screen_uv = i.screen_uv.xy / i.screen_uv.w;
    fixed4 col = tex2D(_TimeCrackTexture, screen_uv);
    col.a = _ColorAlpha;
    return col;
}
ENDCG
```

Figure 40: Time Travelling Portal fragment shader

4.3.2. Cameras

There are three cameras in whole game, one of them only renders menus, and the other two are responsible of the different environments. In this section every camera setup is going to be explained.

The first camera and the simplest one is set as orthographic because it just needs to render the canvases of the menus. Also, the rendering path is set as forward since the orthographic projection do not allow deferred path.

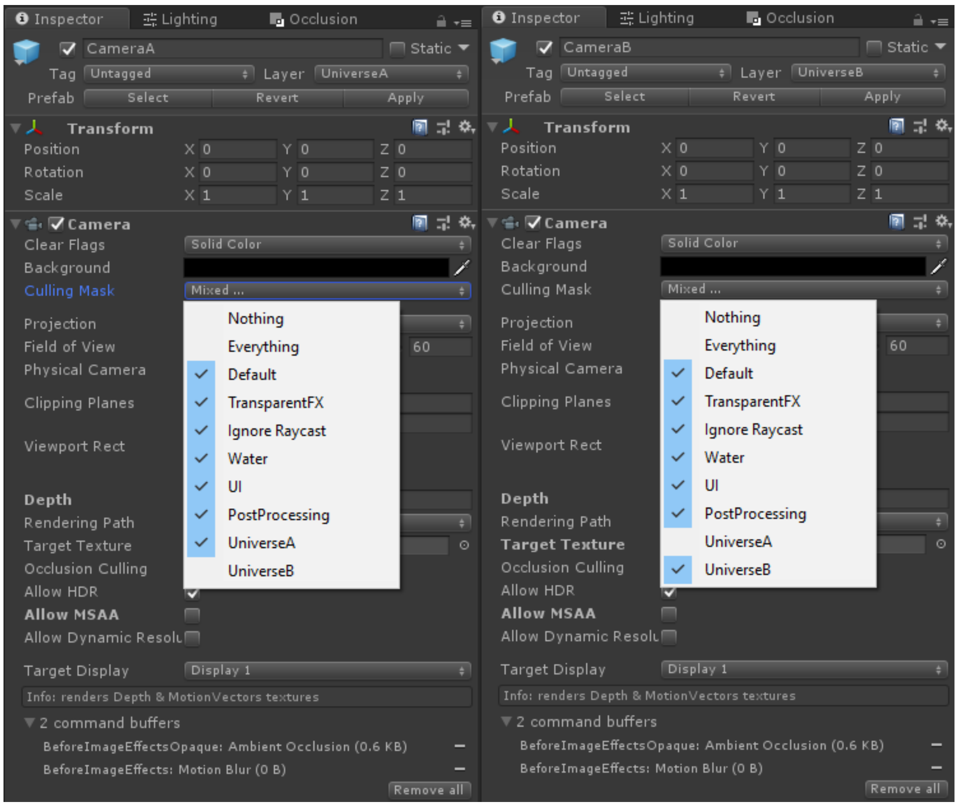
This camera also holds the ambient audio source and, in this case, it has not post processing effects.

Now it is time to explain game cameras, which are so important in the scene swap during the depth effect.

Both cameras have same setup, perspective projection with black color in the background, deferred rendering path just to achieve better graphics, and they both have the post processing profile explained in section 6.1.4 of this document.

However, there are few big differences between them, but first is especially important to underline that in order to distinguish scene A environment (80's cafeteria) and scene B (dystopic bar), each one is in different layers, so everything in scene A is place in layer UniverseA, and everything in scene B is place in layer UniverseB, and collisions between both layers are disabled to avoid the player colliding with invisible objects.

So now it is possible to indicate cameras differences. Basically, if one camera just needs to render what happens in one layer, Unity offers a culling mask that can be changed to ignore different layers that are not checked in the list. Said that, scene A camera has UniverseB layer unchecked in culling mask list, and scene B camera has UniverseA layer unchecked (Figure 41).



CameraA
Tag Untagged Layer UniverseA
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Clear Flags Solid Color
Background
Culling Mask Mixed ...
Nothing
Everything
✓ Default
✓ TransparentFX
✓ Ignore Raycast
✓ Water
✓ UI
✓ PostProcessing
✓ UniverseA
UniverseB
Projection
Field of View 60
Physical Camera
Clipping Planes
Viewport Rect
Depth
Rendering Path
Target Texture
Occlusion Culling
Allow HDR ✓
Allow MSAA
Allow Dynamic Resolu
Target Display Display 1
Info: renders Depth & MotionVectors textures
2 command buffers
BeforeImageEffectsOpaque: Ambient Occlusion (0.6 KB)
BeforeImageEffects: Motion Blur (0 B)

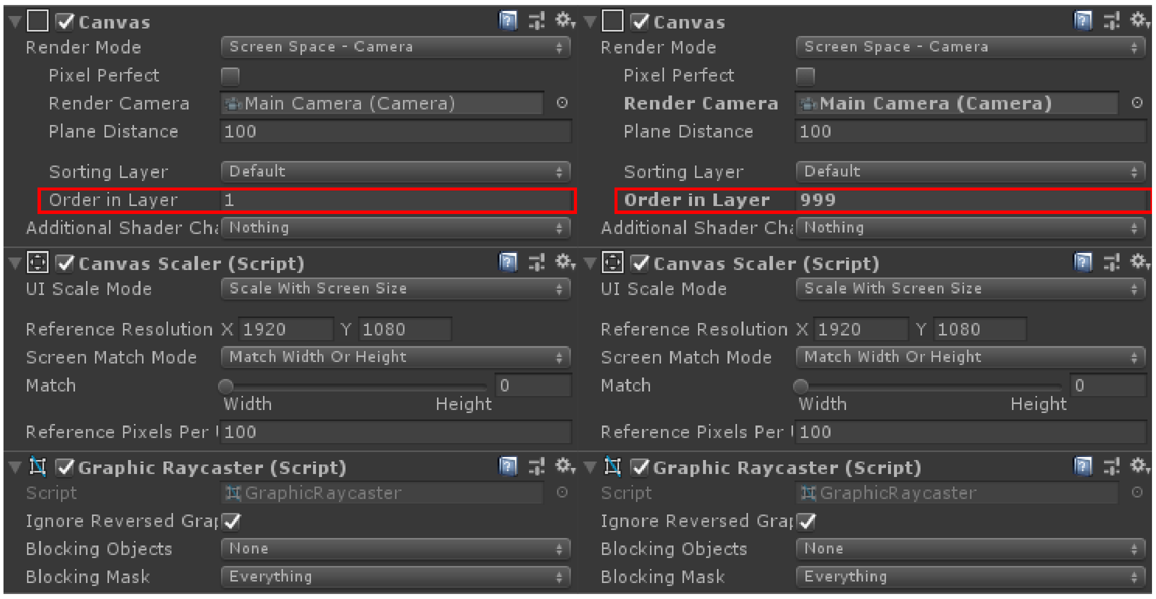
CameraB
Tag Untagged Layer UniverseB
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Clear Flags Solid Color
Background
Culling Mask Mixed ...
Nothing
Everything
✓ Default
✓ TransparentFX
✓ Ignore Raycast
✓ Water
✓ UI
✓ PostProcessing
UniverseA
✓ UniverseB
Projection
Field of View 60
Physical Camera
Clipping Planes
Viewport Rect
Depth
Rendering Path
Target Texture
Occlusion Culling
Allow HDR ✓
Allow MSAA
Allow Dynamic Resolu
Target Display Display 1
Info: renders Depth & MotionVectors textures
2 command buffers
BeforeImageEffectsOpaque: Ambient Occlusion (0.6 KB)
BeforeImageEffects: Motion Blur (0 B)

Figure 41: Project cameras setup

4.3.3. Menus and Navigation

Menus system has been a big task in this project because one of the goals were making an intuitive navigation system, without black transitions between menu panels and full of clean animations.

First of all, and in top of the hierarchy, menus system is formed by two canvases, one that contains all the panels that configure the navigation, and one canvas that makes the transition between menus and game. Nevertheless, both are configured similarly; render mode is set in screen space with the only one camera in the scene, but the second canvas is in a higher order in layer ensuring that always will overlap the other canvas (Figure 42).



```
Canvas                                        Canvas
Render Mode          Screen Space - Camera    Render Mode          Screen Space - Camera
  Pixel Perfect      ☐                          Pixel Perfect      ☐
  Render Camera      Main Camera (Camera)       Render Camera      Main Camera (Camera)
  Plane Distance     100                        Plane Distance     100

  Sorting Layer      Default                    Sorting Layer      Default
  Order in Layer     1                          Order in Layer     999
Additional Shader Ch Nothing                  Additional Shader Ch Nothing
Canvas Scaler (Script)                        Canvas Scaler (Script)
UI Scale Mode        Scale With Screen Size   UI Scale Mode        Scale With Screen Size

Reference Resolution X 1920   Y 1080          Reference Resolution X 1920   Y 1080
Screen Match Mode    Match Width Or Height    Screen Match Mode    Match Width Or Height
Match                                  0      Match                                  0
                     Width       Height                            Width       Height
Reference Pixels Per 100                      Reference Pixels Per 100
Graphic Raycaster (Script)                    Graphic Raycaster (Script)
Script               GraphicRaycaster         Script               GraphicRaycaster
Ignore Reversed Grap ✓                        Ignore Reversed Grap ✓
Blocking Objects     None                     Blocking Objects     None
Blocking Mask        Everything               Blocking Mask        Everything
```

Figure 42: Menus system canvas

First canvas has several panels whose navigation is managed by a single script; this is one of the most powerful features in the game because you can navigate between menus without needing to go to a main menu. This makes the flow so smooth and it is possible to access everywhere in just one click. Now, every single panel is going to be explained separately since there are not one unique flow between menus:

- Top Panel: this panel holds the buttons that change the content of the panel is seen (Figure 43). These buttons are Home button, Controls button, Puzzles button, Credits button, Settings button and Exit button, but they will be explained inside the scripts section.

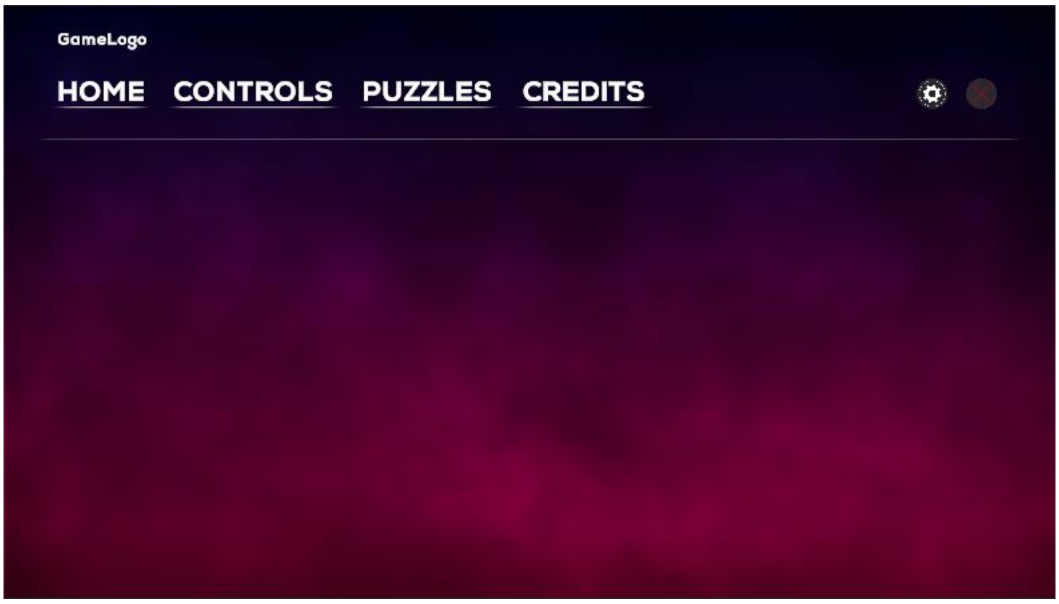


Figure 43: Top Panel of menus system

- Home Panel: this one is from you can accessing the game (Figure 44); it contains a New Game button that allows the player starting a new adventure, one Continue

Game button that leads the player to the same puzzle where he was, and a Puzzles button that leads you to Puzzles Panel.

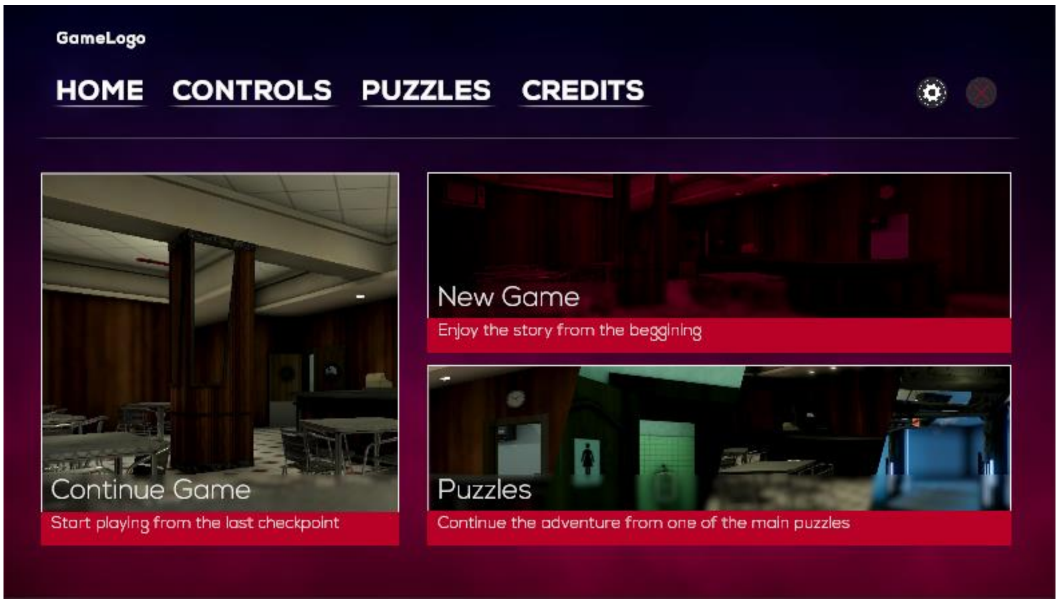
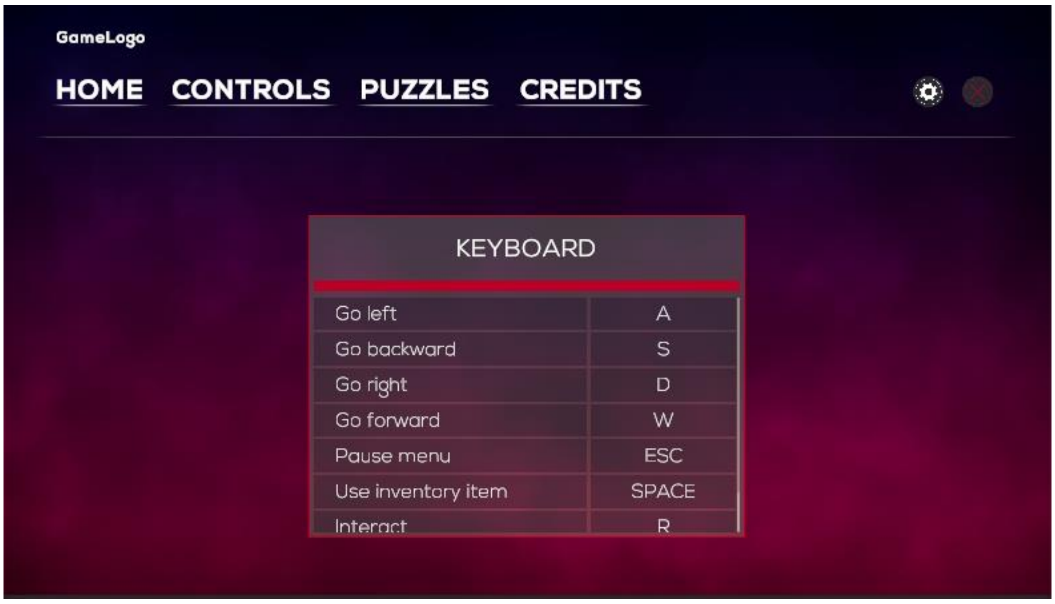

GameLogo
HOME
CONTROLS
PUZZLES
CREDITS
New Game
Enjoy the story from the beggining
Continue Game
Start playing from the last checkpoint
Puzzles
Continue the adventure from one of the main puzzles

Figure 44: Home Panel of menus system

- Controls Panel: here there is just a scroll where every key bind is explained (Figure 45).



KEYBOARD

Go left	A
Go backward	S
Go right	D
Go forward	W
Pause menu	ESC
Use inventory item	SPACE
Interact	R

Figure 45: Controls Panel of menus system

- Puzzles Panel: in this panel, it is possible to access any puzzle if the player has passed that puzzle (Puzzle 46).

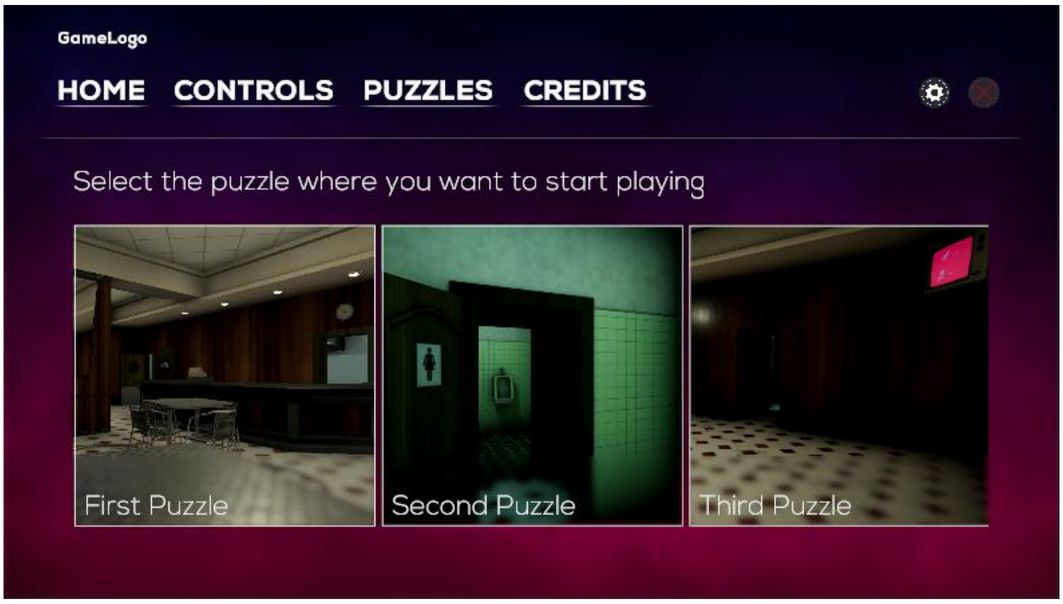
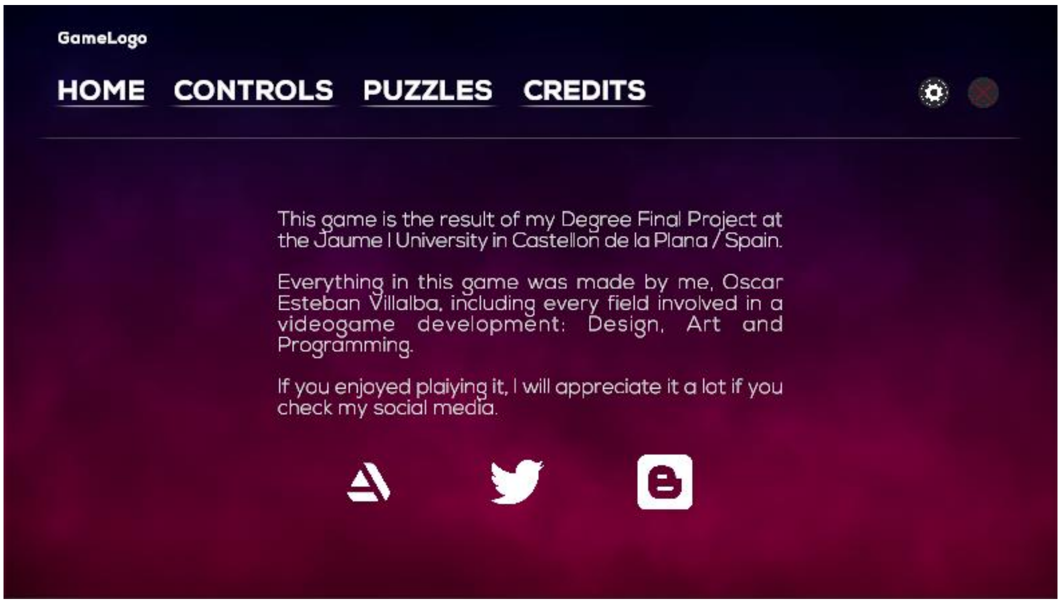


Figure 46: Puzzles Panel of menus system

- Credits Panel: in this panel, anyone who plays the game could see the project context and following me in my social media (Figure 47).



This game is the result of my Degree Final Project at the Jaume I University in Castellon de la Plana / Spain.

Everything in this game was made by me, Oscar Esteban Villalba, including every field involved in a videogame development: Design, Art and Programming.

If you enjoyed plaiying it, I will appreciate it a lot if you check my social media.

Figure 47: Credits Panel of menus system

- Settings Panel: this last panel holds a scroll with technical options that can be changed if the framerate of the game is low, it includes settings like general volume, quality, resolution, Fullscreen mode or post processing effects management (Figure 48).

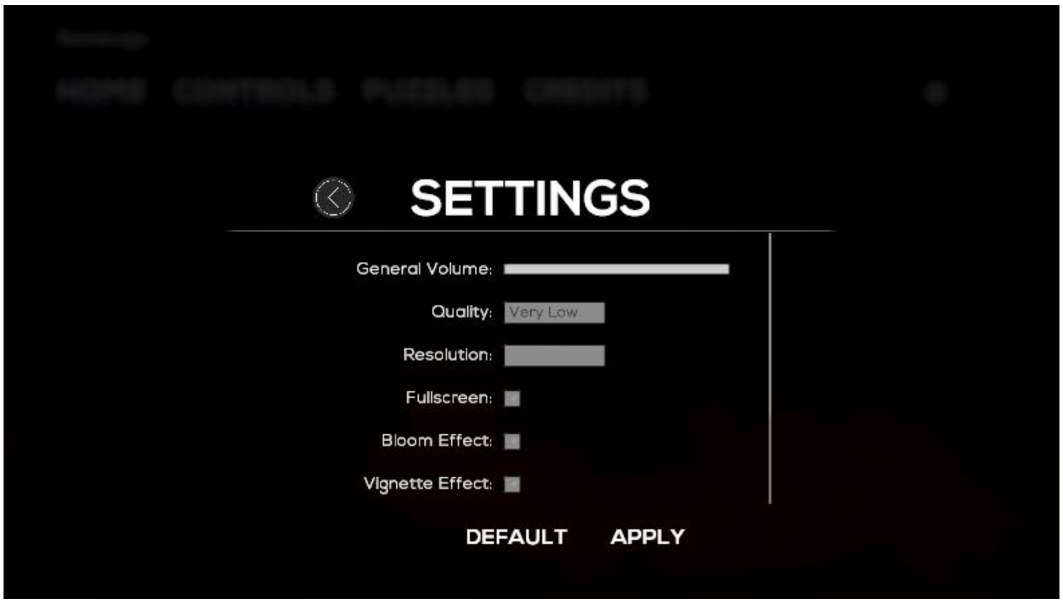


Figure 48: Settings Panel of menus system

The second canvas just have a black image controlled by an animation, and this animation is enabled by the buttons that lead to the game, but they will be explained in the following section.

4.3.4. Scripts

In this section the most important scripts of the project are going to be explained. However, the project source code link is in the bibliography for more information about the project [Esteban O, 2019].

Before the explanation, I believe it is very important to say that since the project main goal is more oriented to 3D art and graphics programming, features like Dialogue system and Inventory have been adapted from already created scripts by a youtuber named Brackeys who has been a great help during the development. However, the most important scripts that compose the game such as progression system, menus and puzzles have been developed by myself.

4.3.4.1. Menu Scripts

The following scripts manage menus navigation and trigger panels and buttons animations:

- **Top Panel Manager:**

For a good explanation of this script, it is good to clarify that every panel in the menu has two animations, one animation fade in the panel and the other fade out the panel; they are called MM Panel In and MM Panel Out respectively.

In Top Panel Manager class, there are two lists that store all panels and buttons for accessing their animations in any moment.

Methods:

PanelAnim(int): it receives the list index of the objective panel and it is called when a button of the top menu is pressed; in that moment two animations are played, MM Panel Out of the current panel, and MM Panel In of the objective panel.

The script acts as a panel manager and it is responsible of the navigation between menus.

- **On New Game Click:**

One way to enter game is by pressing New Game button. This On New Game Click script is attached to the second canvas, and contains the method that starts scene changing.

Methods:

LoadNewGame(int): it loads the first scene initializing the game from the first puzzle.

- **On Continue Game Click:**

This class is also attached to the second canvas, and it is quite similar to the New Game one.

Methods:

LoadGame(int): it starts the game from the player last gaming session.

- **Puzzles Selector:**

This script is attached to the puzzles panel; it stores the buttons of every puzzle.

Methods:

SetCurrentPuzzle(int): it receives the number of the puzzle, and it is called by pressing the button of the puzzle the player wants to go.

- **Settings Menu:**

In Settings Menu it is possible to change a bunch of game settings in order to improve the performance.

Every listed option has two global variables stored in Unity PlayerPrefs, one stores current configuration of the setting, and the other one stores the previous configuration of the same setting; this last variable is useful if the player needs to revert changes. Moreover, the listed options have a Set method that changes the first global variable with the value received by parameter.

Methods:

ApplyChanges(): it changes the value of every previous setting configuration global variable, and access the corresponding variables that store the corresponding setting, changing it with the changed variable. These variables are:

  - AudioListener.volume;
  - QualitySettings.SetQualityLevel(int index);
  - Screen.fullScreen;
  - Screen.SetResolution(int width, int height, bool fullscreen)
  - PostProcessingProfile.bloom.enabled;

- PostProcessingProfile.vignette.enabled;
- PostProcessingProfile.motionBlur.enabled;
- PostProcessingProfile.antialiasing.enabled;

RevertChanges(): this function set these variables listed in the previous paragraph with the value stored in the previous setting configuration global variables.

SetDefaultSettings(): it simply calls every Set method of each setting and apply the changes immediately.

4.3.4.2. Interaction Scripts

Here the basics of interaction of the game are going to be explained, specifically there is only one major script, but there are also a few scripts that inherit the class which can be seen in Figure 49, but they will be explained in their respective sections.

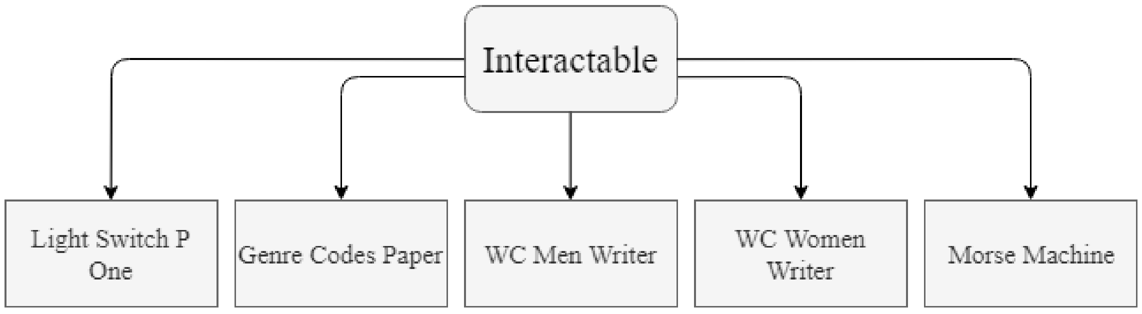


Figure 49: Interactables diagram

- **Interactable:**

This script is the foundation of every single object the player is able to interact.

Methods:

Interact(): this method will be override by every interactable variations inside the game, and each one will do different actions.

OnFocused() and OnDefocused(): they store the information about the character focus on a interactable object.

Update(): method that prevent the Interact() method to be spammed when looking to an object.

4.3.4.3. Dialogue Scripts

One of the limitations of the project was that there would not be voices, because the hardware and software that allow to produce that kind of features is quite difficult to acquire. The solution was developing a dialogue system that would print the story quotes in the screen, and it is an adaptation of the system made by the aforementioned youtuber [Brackeys, 2017].

The system is formed by three different scripts that follow the flowchart in Figure 50.

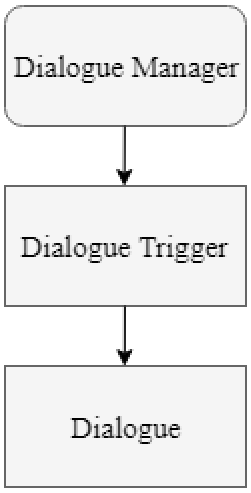


Figure 50: Dialogue system diagram

- **Dialogue:**

With this class is possible to create objects that contain an array that holds the strings will be printed in the screen.

- **Dialogue Manager:**

This is the first manager explained in the document; therefore, it is important to explain that every manager of the project follows a design pattern called Singleton. This pattern allows the access to the content of the object anywhere, and it prevents the object being multi-instantiated.

In this case, Dialogue Manager needs to be accessed wherever a dialogue object is placed, and this is the reason of being a Singleton object.

Methods:

StartDialogue(Dialogue): it receives a Dialogue object explained previously, and inserts the sentences inside a queue before displaying them.

DisplayNextSentence(): this function types the next sentence letter by letter invoking a coroutine, and it is called from two different code lines; in StartDialogue(Dialogue) method after enqueuing all the sentences, and everytime the player press key T.

EndDialogue(): it just enables player movement and triggers the dialogue container close animation.

- **Dialogue Trigger:**

This script acts as dialogue container and activator; every time a dialogue is needed in the project, it is necessary to attach this script to an object, write the lines that will be displayed in runtime, and invoke the only one method this script has.

Methods:

TriggerDialogue(): it calls Dialogue Manager StartDialogue(Dialogue) method passing him passing the Dialogue object created.

4.3.4.4. <u>Inventory Scripts</u>

In this inventory system there are two branches, one oriented to the UI inventory, and the other one oriented to player objects holder and these objects switching (Figure 51).

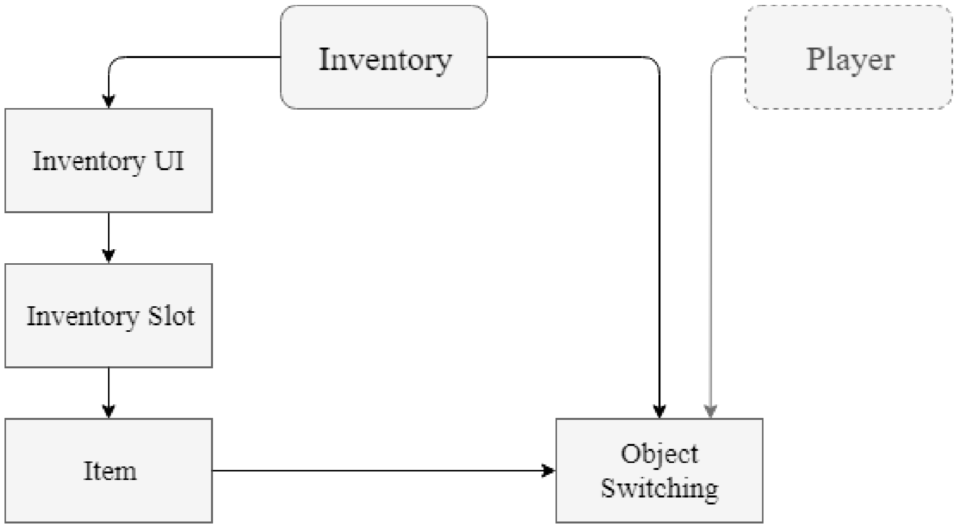


Figure 51: Inventory System diagram

- **Item:**

The basic particle in this system is the Item, a scriptable object that stores the icon of the object that will be placed in the UI, and also the prefab that will be used by the player. This second element will be allocated inside the player objects holder object.

Items could be created thanks to the header in the script that enables a new button inside the asset creation menu.

- **Inventory Slot:**

The UI dedicated to the inventory is formed by several objects with this script attached.

Methods:

<u>AddItem(Item)</u>: it enables the UI icon and set its sprite to the one the item received by parameter has.

<u>ClearSlot()</u>: it disables the UI icon and set the sprite to null.

- **Inventory UI:**

This script is the one who updates what is shown in the inventory UI.

Methods:

<u>UpdateUI()</u>: it iterates through the inventory slots calling AddItem(Item) or ClearSlot() methods.

Every time an item is added or deleted from the inventory, the method UpdateUI() would be called.

- **Inventory:**

In this script, a Delegate object is used to call the Inventory UI UpdateUI() method.

Methods:

Add(Item): a method that in addition to update the UI, before that, it inserts one item in the items general list and instantiate the prefab in player's object holder.

Remove(Item): it deletes the item from the items general list and it also update the UI.

As well as the dialogue class can be accessed from anywhere, this script is also set as Singleton, because it could be necessary to add items when the player interacts with them in the environment.

- **Object Switching:**

This is the last class related to the inventory system; it only has one variable that stores the selected object index inside the player objects holder hierarchy and its main goal is waiting for the player to scroll down or up the mouse wheel.

Methods:

SelectObject(): it iterates through the player objects hierarchy enabling the corresponding object with the selected object index value, and disabling the other objects.

4.3.4.5. Player Scripts

In this section, the player controller is going to be explained, specifically, in this project, it has been divided in two scripts, one that manages the movement and one that rotate the camera (Figure 52).

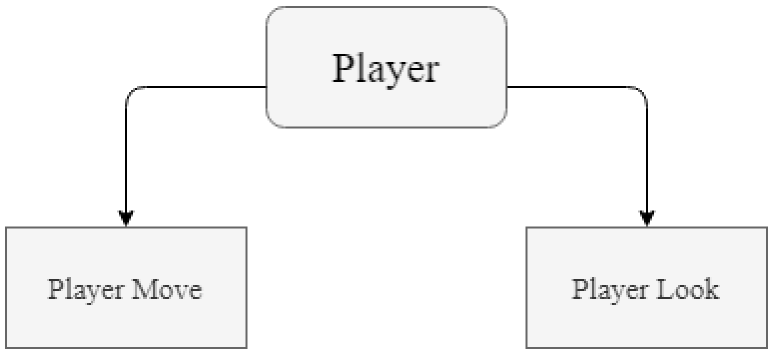


Figure 52: Player System diagram

- **Player Move:**

This script has two methods executed inside a Update().

Methods:

PlayerMovement(): it moves the Character Controller if the player inputs horizontal or vertical axes.

PlaySoundStep(): it is responsible for the step sounds during the movement of the character.

- **Player Look:**

Player Look class has two important features, one that rotate the camera in first person perspective, and manages the player focusing in interactable objects. Both are executed inside a Update() method.

Firstly, camera rotation is controlled by mouse movement.

Also, this script controls when the player has interacted with an interactable object. This is done by throwing rays and casting the hit.

- **Player:**

This last script of Player System manages the player interaction with the dialogues explained in section 6.3.4.3 by waiting for a T key input in order to call DisplayNextSentence() or TriggerDialogue() methods whether a dialogue is already opened or not.

Furthermore, this class controls the player save and load system, but this is explained in the following sections.

4.3.4.6. Sounds Manager

When developing this project, it was considered very important to have several managers to avoid repeating code and being able to access generic methods from anywhere. In order to achieve that, every single manager was created as a Singleton object.

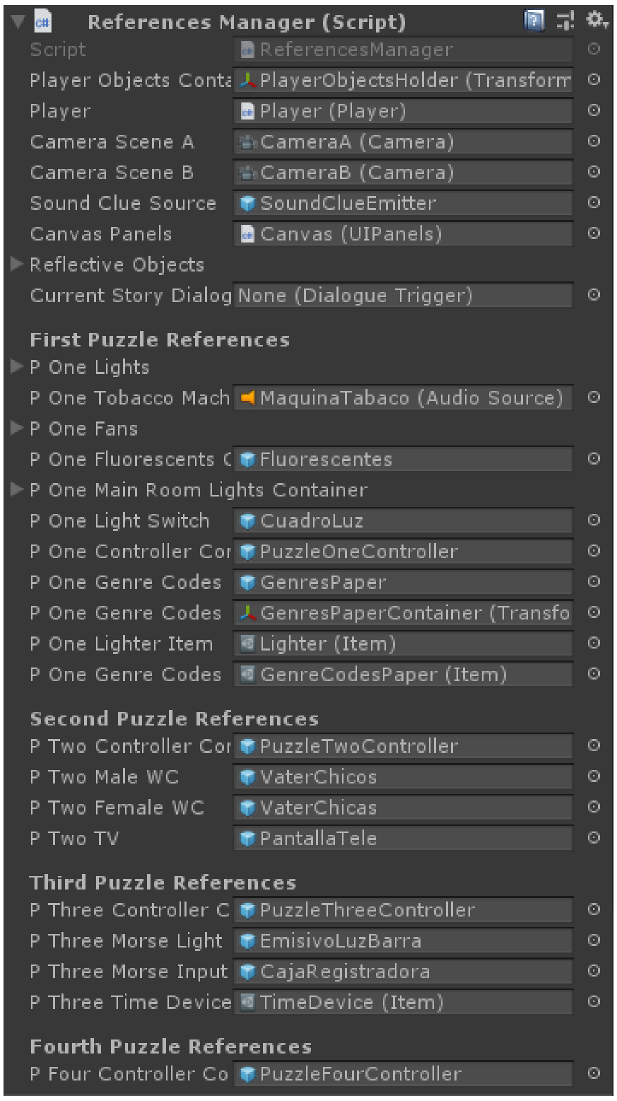
Sound Manager is the first one, it contains all the sound effects that are used in the game.

Methods:

PlaySFX(AudioSource, AudioClip): it receives the AudioSource that will emit the sound, and the AudioClip that will be played by the previous parameter.

4.3.4.7. References Manager

This class works as a repository for every single reference needed in the game (Figure 53); the main goal of this script is avoiding Find code lines that overload the CPU, and because of this, every reference is dragged and dropped in Unity editor. This could produce big null references problems, but before adding a new variable in this script is quite important to study if the variable could be destroyed in any moment during the game.



References Manager (Script)

Script	ReferencesManager
Player Objects Conta	PlayerObjectsHolder (Transform
Player	Player (Player)
Camera Scene A	CameraA (Camera)
Camera Scene B	CameraB (Camera)
Sound Clue Source	SoundClueEmitter
Canvas Panels	Canvas (UIPanels)
Reflective Objects	
Current Story Dialog	None (Dialogue Trigger)
First Puzzle References	
P One Lights	
P One Tobacco Mach	MaquinaTabaco (Audio Source)
P One Fans	
P One Fluorescents C	Fluorescentes
P One Main Room Lights Container	
P One Light Switch	CuadroLuz
P One Controller Cor	PuzzleOneController
P One Genre Codes	GenresPaper
P One Genre Codes	GenresPaperContainer (Transfo
P One Lighter Item	Lighter (Item)
P One Genre Codes	GenreCodesPaper (Item)
Second Puzzle References	
P Two Controller Cor	PuzzleTwoController
P Two Male WC	VaterChicos
P Two Female WC	VaterChicas
P Two TV	PantallaTele
Third Puzzle References	
P Three Controller C	PuzzleThreeController
P Three Morse Light	EmisivoLuzBarra
P Three Morse Input	CajaRegistradora
P Three Time Device	TimeDevice (Item)
Fourth Puzzle References	
P Four Controller Co	PuzzleFourController

Figure 53: References Manager in Unity Editor

Also, this script is a Singleton that can be accessed from anywhere.

4.3.4.8. Game Manager

Now it is time to explain the script that controls the puzzles passing, it has two different branches inside the Start() method, one of them used when there is nothing to load, and the other one every time the player continues his adventure.

Methods:

SetPuzzleEnvironment(int): it receives the number of the current puzzle and add the correspondent puzzle controller to a hierarchy object; this will set up the scene with the essential elements.

SetEnvironmentVestiges(int): it receives the puzzle number, and configures the scene as it would be if every puzzle before the puzzle number received was passed. Also, SetPuzzleEnvironment(int) is called receiving the current puzzle number.

LoadSceneAdditive(int): which will be useful for loading the scene B at the end of the third puzzle; it simply receives the scene build index, and load the correspondent scene as an additive one, everything via Unity SceneManager.

So now, it is time to detail how each puzzle is managed, but before, the diagram is defined as it is visible in Figure 54.

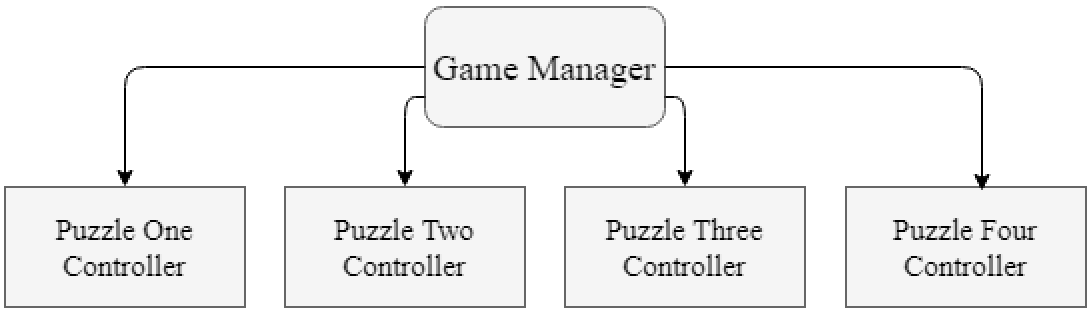


Figure 54: Puzzles management diagram

4.3.4.9. Puzzle One

In this first puzzle, the player wakes up inside a dark place, only with a lighter in the inventory, and its mission is turning on the light of the cafeteria.

The involved scripts in it are:

- **Puzzle One Controller:**

This script acts as the manager of the puzzle one.

Methods:

ConfigureLevelOne(): it does several things:

- Set the lightmap to a darker one (this feature is explained in the next script).
- Turn off all the lights referenced in References Manager.
- Inserts the lighter into the player's inventory.
- Instantiate the interactable that allows passing to the next puzzle, in this case is a paper with several symbols in it.

Once the player interacts with the light switch in the kitchen, this controller detects it with a Boolean variable, and it does the following:

- Set the lightmap to a brighter one.
- Turn on all the lights.
- Turn on every electric device inside the environment, as the fans, the TV or the tobacco machine.

Finally, if the player interacts with the paper inside the light switch, another Boolean variable is changed telling this controller the puzzle has been finished; then, a sound effect that indicates the puzzle solving is played, the Game Manager SetPuzzleEnvironment(int) method is called passing it the next puzzle number, and finally, in order to reduce CPU work, the controller is destroyed by itself.

SetPuzzleOneVestiges(): this function set everything as it needs to be if the puzzle would be passed:

- It turns on the power of the cafeteria.
- It changes the lightmap to the brighter one.
- It inserts the lighter and the mentioned paper with the codes, and finally it destroys the controller as well.

It is very important to keep in mind that this last method is used for the save system in order to update the scene until the current puzzle.

- **Lightmap Switch:**

This feature is one of the most interesting at the same time it has been one of the most difficult to implement.

It basically creates two public lightmaps given the correspondent texture sets. But this feature has to be done because Unity is not prepared to do that kind of lightmap changes; it is also known that the scripted solution is very hardcoded, but in the explanation all of this will be clarified.

There are three texture sets needed for making one lightmap data, one set for the lightmap directional information, one for the lightmap light information, and one last for the shadowmask information; the problem is that this textures, that could be more than 20 for each set, need to be placed in the editor by hand, because Unity does not divides them in different folders, and this can be quite tedious.

One more bug caused by the lightmap switching is that once the game is built, it is not placed properly if the batching is disabled; batching is a technique that Unity uses in order to free CPU usage by grouping meshes and rendering them together.

Finally, if everything is correctly set up, it would be possible to bake different lighting in two different folders and place the texture sets in the Lightmap Switch via Unity editor.

4.3.4.10. Puzzle Two

Now, the player has one new item, a paper that has several codes written, specifically, these are codes based on genre symbols combinations, and the only place they can be introduced is in interacting with the two toilets.

- **Puzzle Two Controller:**

All the controllers are very similar because they have the same structure for initializing the puzzle and detect when it has been solved.

Methods:

ConfigureLevelTwo(): in this case, the actions done by the function are:

- Enable the screen of one of the TVs.
- Add the script that manages the WCs puzzle.

Once the puzzle is solved, the script plays a sound effect and the method SetPuzzleEnvironment(int) is called again. In addition, the WCs puzzle manager, the interactable script of the toilets and the controller are destroyed.

SetPuzzleTwoVestiges(): it simply enables the TV screen, and makes this screen emitting the Morse translations used in puzzle three.

The script that manages the WCs puzzle need to be accessed by the two correspondent objects of the class Interactable; this diagram is defined in Figure 55.

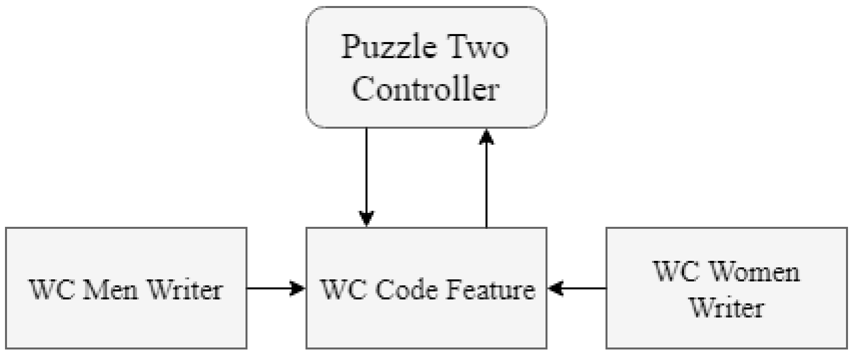


Figure 55: WC Code Puzzle diagram

- **WC Code Feature:**

This script contains the total of the four codes stored in different arrays, following the translation Female = 1 and Male = 0 and one list that stores the input the player introduces by interacting with the toilets.

Methods:

CheckCorrectCode(): it compares the current code with the player input one and returns a Boolean value. Finally, when the last code is introduced correctly, the script tells Puzzle Two Controller that the puzzle has been solved by changing a public Boolean value.

At the same time one code is introduced properly, a Morse translation appears on the TV screen, but this will be used in the next puzzle.

- **WC Writer:**

This script extends the class Interactable because both toilets need to be interacting with the player, but the main difference between them is that the one attached to the women toilet introduce the number 1 inside the input list, and the men toilet introduce the number 0. This is done by accessing the public input list from the override Interact() method every time the player press R while pointing to the toilet asset.

- **TV Screen Manager:**

As it has been mentioned, every time the player introduces a code correctly, one Morse translation is placed on the screen of the TV (Figure 56).

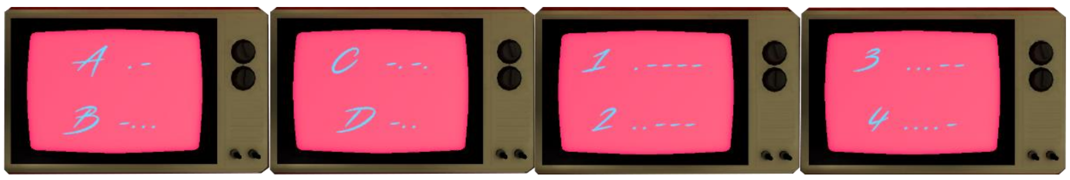


Figure 56: TVs Morse translations

This script has a coroutine that iterate through the available Morse translations, and change the texture of the screen every 3 seconds. This coroutine is a little bit tricky because it is impossible to add new textures while the coroutine is running, and it was necessary to make another list in order to store the added textures during the coroutine execution and insert them at the end of the loop.

4.3.4.11. Puzzle Three

The puzzle number 3 is the last that which has an enigma to solve, and it is a Morse codes system emitted by one of the cafeteria lamps. The player needs to watch that codes, and with the TV translation's help, the resulting code translation needs to be introduced in the register cash of the cafeteria.

The flowchart is defined by the following diagram in Figure 57, and it will be explained step by step.

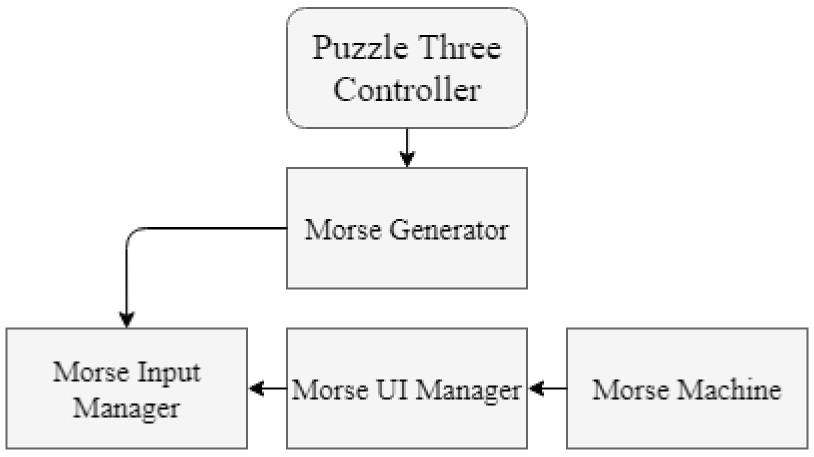


Figure 57: Morse puzzle diagram

- **Puzzle Three Controller:**

This script also has a method dedicated to configure the puzzle.

Methods:

<u>ConfigureLevelThree()</u>: it adds a Morse Generator script to one of the lights, and also add a Morse Machine and Morse Input Manager components to the cash register asset of the cafeteria.

As well as the other puzzles, it is waiting for another script that change a Boolean value that indicates the puzzle has been solved, and then it would do the following actions:

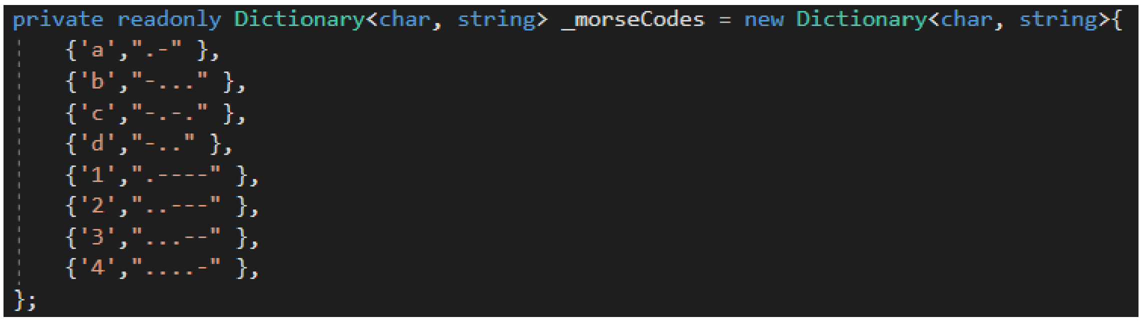
- Call Game Manager LoadSceneAdditive(int) method because the scene B will be needed in the next puzzle.

- Play a sound effect to indicate the puzzle is already done.
- Add the time travelling device to the inventory.
- Call SetPuzzleEnvironment(int) with the number of the last puzzle.
- Destroy every component of the Morse system and itself.

SetPuzzleThreeVestiges(): now, the function loads the scene B in the same way of the puzzle ending, it adds the time travelling device as well, and it finally destroy the mentioned elements.

- **Morse Generator:**

The main component of the generator is a Dictionary whose key is a character, which can be a letter or a digit, and its value is the Morse translation of the digit (Figure 58).



```
private readonly Dictionary<char, string> _morseCodes = new Dictionary<char, string>{
    {'a',".-" },
    {'b',"-..." },
    {'c',"-.-." },
    {'d',"-.." },
    {'1',".----" },
    {'2',"..---" },
    {'3',"...--" },
    {'4',"....-" },
};
```

Figure 58: Morse Dictionary

Methods:

FiveDigitsGenerator(): this method builds a string by adding 4 digits of the list randomly in a String Builder.

After that, a coroutine is invoked showing the code until the player passes the puzzle, by turning on and off the referenced light during different time intervals whether the Morse translation is a dot or a dash.

- **Morse Machine:**

This is the simplest script of the Morse system, it extends from Interactable, and it is attached to the cash register allowing the player interact with it.

Once it is attached, if the player interacts, a UI dedicated with several digits and letters will open (Figure 59).

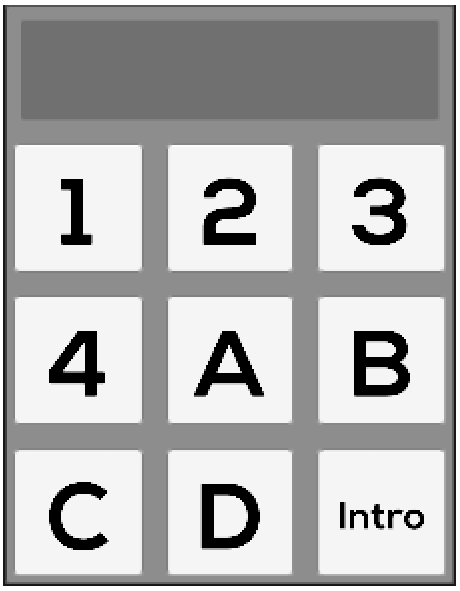


Figure 59: Morse input UI

- **Morse Input Manager:**

This manager acts as the system core, it contains the list that holds the player input, and it also validate the code when the player decides his input is the correct translation.

It waits until the UI script tells him a validation is needed.

Methods:

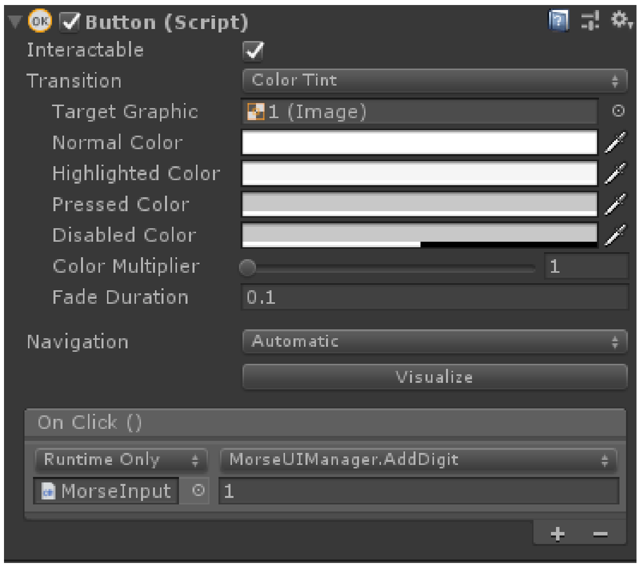
<u>ValidMorseCode()</u>: it iterates and compares the objective code and the player’s code, and tells the Puzzle Three Controller that the puzzle is solved, or in counterpart it plays a sound effect indicating the code is wrong.

- **Morse UI Manager:**

Once the player enables input UI, he is able to press the buttons in order to introduce the digit printed in the button.

Methods:

<u>AddDigit(string):</u> it accesses Morse Input Manager input list, adding the parameter string as a character (Figure 60).



Button (Script)

Interactable	✓
Transition	Color Tint
Target Graphic	1 (Image)
Normal Color	
Highlighted Color	
Pressed Color	
Disabled Color	
Color Multiplier	1
Fade Duration	0.1
Navigation	Automatic

Visualize

On Click ()

Runtime Only | MorseUIManager.AddDigit
MorseInput | 1

Figure 60: Morse UI button configuration

<u>ValidateCode()</u>: it tells Morse Input Manager that it has to check if the introduced code is the correct one.

Now every puzzle has been passed, the scene B is loaded, the player is able to travel to the future changing the scene which is rendered and triggering the main effect of the project, but this is managed by another script.

4.3.4.12. Scene Changing System

This system is the responsible of the main effect of the project, and basically it is formed by two important scripts.

- **Twin Camera Controller:**

As it was explained in Ring Pass Effect section (6.3.1.1), the shader requires of two main textures, _MainTex that contains what the active camera is watching, and _AnotherTex that holds what the hidden camera is watching. Well, in this script first instance, the active camera is the one that renders the scene A, and the hidden camera is the one that renders the scene B. But there is a problem with the hidden camera, and that is because in order to have a visualization of what that camera is watching, it is necessary to create a RenderTexture that keeps an image of what is happening in that scene at any moment.

Now the basic elements are correctly referenced, the script awaits the player's Space key input, and then few methods are invoked.

Methods:

<u>ChangeSceneMat()</u>: it is the method that triggers the Ring Pass Effect shader; it sets _AnotherTex with the target texture declared, it changes the trigger _RunRingPass property value with a 1, and it also establish the ring width changing the shader property _RingWidth.

Once the effect finishes, it is necessary to tweak a few variables; the active camera now must be the hidden one and the hidden camera is now the active one, and in consequence, the render texture needs to be attached to the new hidden camera.

ChangePlayerLayer(): it accesses to player's transform and change the layer recursively from UniverseA to Universe B and vice versa.

- **Camera RT Capture:**

This second script only has the goal of updating the content of the render texture explained in the previous point.

Methods:

OnRenderImage(RenderTexture source, RenderTexture destination): it is commonly used to make post processing effects, because it takes a source image taken after everything has been rendered, and after applying it a material, it puts the result in a destination RenderTexture.

Graphics.Blit(RenderTexture source, RenderTexture destination, Material mat): it applies the material received to the source, and draws a quad according to the resulting texture.

In this case, the applied effect is Ring Pass Effect, and the destination is the _MainTex property of the shader.

4.3.4.13. Save and Load System

In order to achieve a functional save and load system, Unity PlayerPrefs object has been used to store global variables that can be used by other scripts to obtain information between scenes and sessions. Now, every variable stored is going to be explained according to the script where it appears.

- **Game Manager:**

(Int) SomethingToLoad: used to load just the first puzzle or set the environment with all the puzzles vestiges until the last session puzzle. Its value is changed only when after enabling Pause Menu, the player decides to go to the main menu.

- **Player:**

In this script, player position is stored in global variables, and it is used to change the player's transform position once he continues the adventure.

(Float) PlayerX: stores the x position of the player transform.

(Float) PlayerY: stores the y position of the player transform.

(Float) PlayerZ: stores the z position of the player transform.

(Int) CurrentPuzzle: stores the current puzzle where the player is.

- **Settings Menu:**

As it is explained in section (6.3.4.1), every settings option has two global variables, one that stores the current value of the setting, and one that holds the previous value of that setting. For example, for the general volume setting, the variables are (Float) General Volume, and (Float) PrevGeneralVolume.

It is visible that the save and load system is not only dedicated to storing player's progress but also his computer settings, a feature that could be very useful in future projects.

5. Results

In this section, the project will be evaluated according to the degree of completion.

The project objective has been achieved, a small game demo that contextualizes a mechanic based in a screen effect that takes scene depth information, and a bunch of realistic 3D assets. However, both scenes are partially empty because the time required to fulfill two entire environments with detailed assets would be insane.

Now, few screenshots showing the game are going to be exposed and explained in order to see the final result in detail.

In Figure 61, it is shown the title screen that opens the menu, after designing a lot of failed titles, it was decided to make a simplest one, with a clear font, because after watching a lot of new logos of different brands or famous video games, there is a trend to design simple titles.

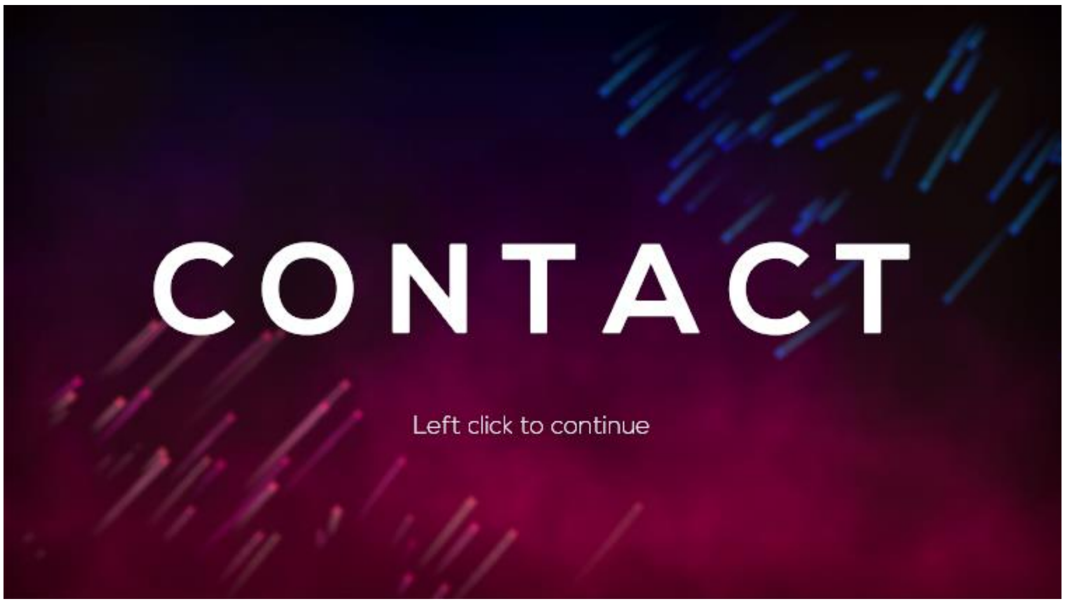


Figure 61: Contact screen title

The next screenshot (Figure 62) show all the player can see when the main character wakes up inside the closed cafeteria, this was made to increase the feeling of being alone inside the environment, and encourage the player to explore with the objective of turning on the power.

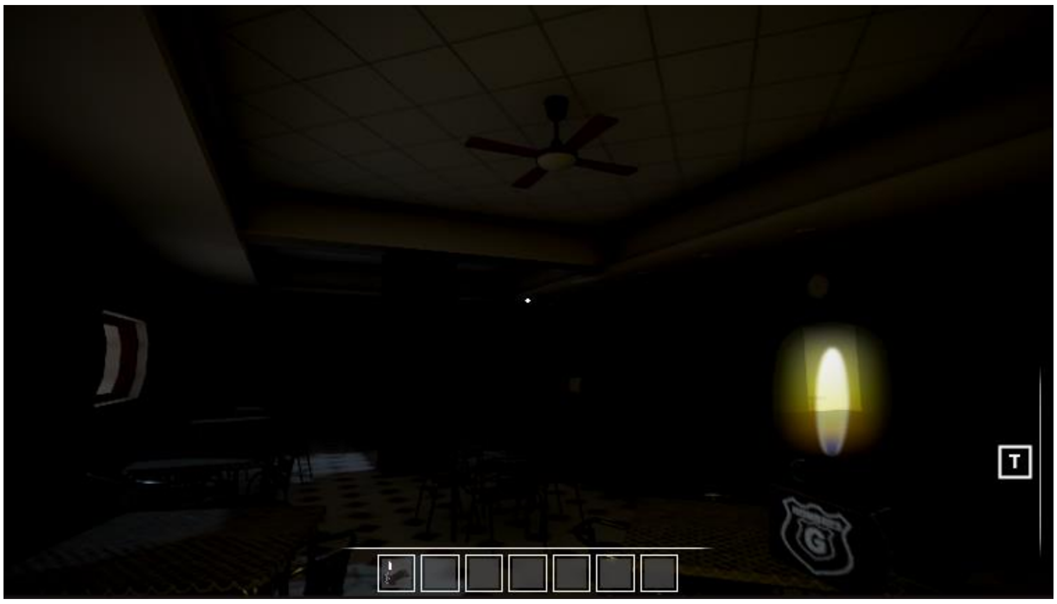


Figure 62: Game beginning

After turning on the power, the player can see everything clearly, in Figure 63 specifically the character is inside the kitchen, it is visible the lack of few more assets that fulfill the room, but there are details like the pipes and cables system that reveals the real work behind the models.

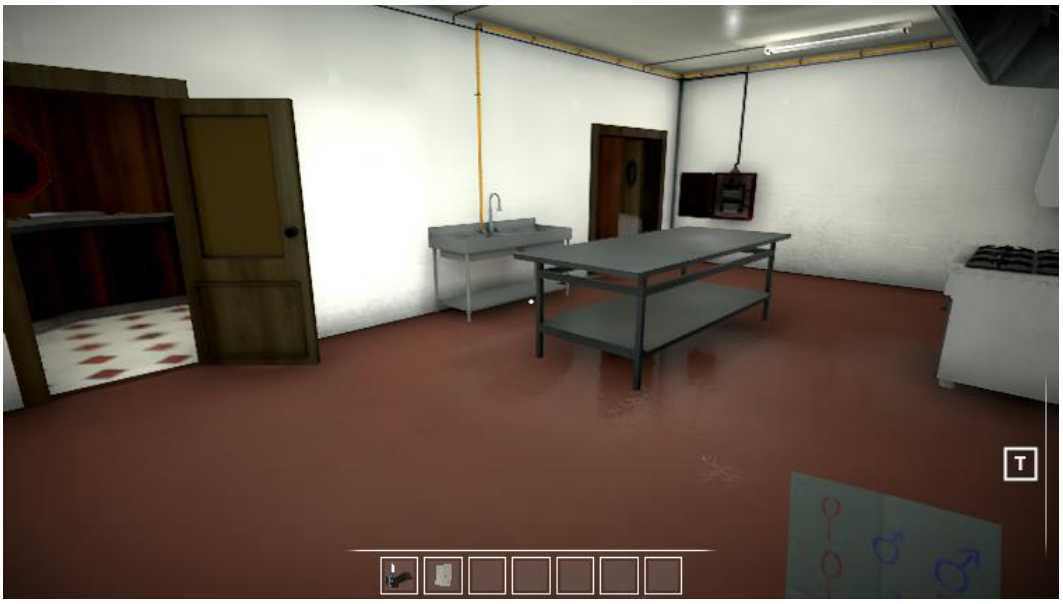


Figure 63: Starting second puzzle

The next rooms the player needs to go are the toilets, these rooms required a lot of references searching, because nowadays, the public toilets tent to be more modern than in the 80's, and square tiles and concrete were the chosen option (Figure 64).

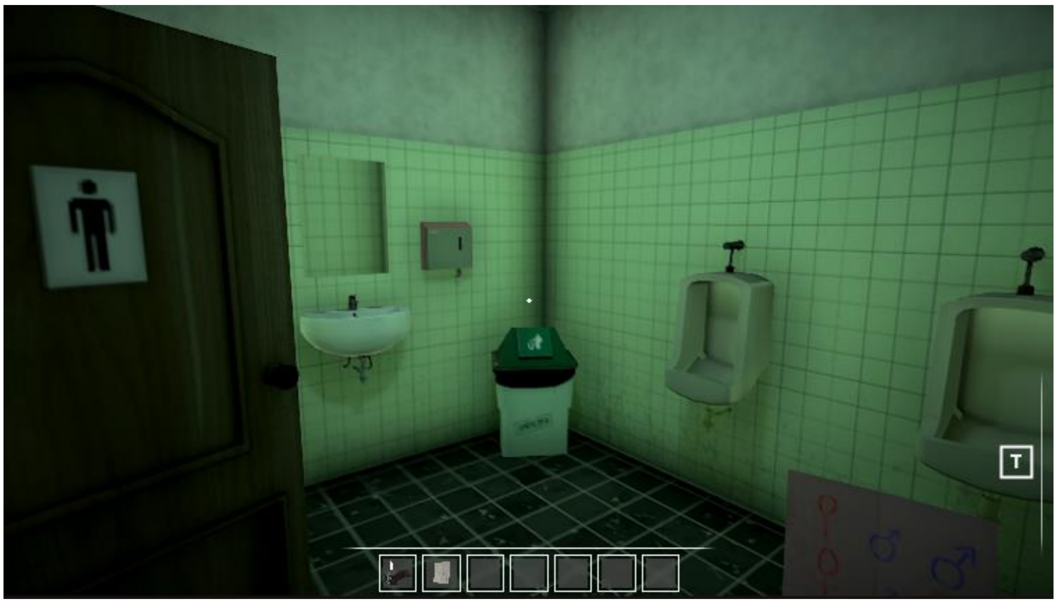


Figure 64: Cafeteria toilets

After passing the second puzzle, Figure 65 represents how the environment looks in the Morse codes puzzle, again, it is visible the lack of few more assets, but in general, the cafeteria looks like it is in the 80’s in Spain, and this was one of the main goals.

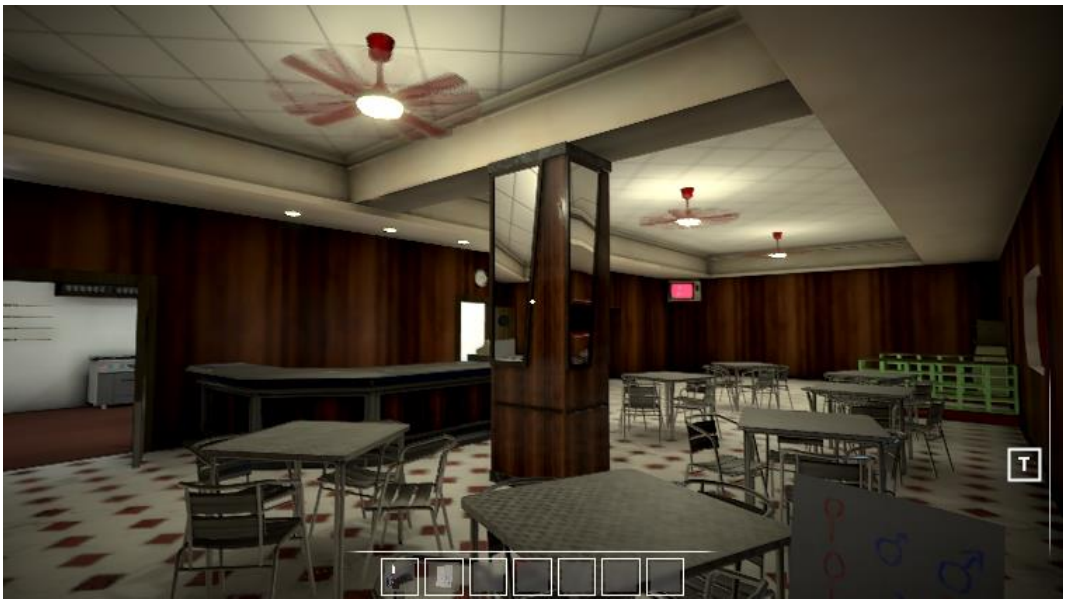


Figure 65: 80's cafeteria

Now in the second environment, in Figure 66 it is visible the difference between times in everything inside the scene; for this environment, it was decided to divide the scene in two zones, one for the poor people that can only stay 20 minutes inside the pub, and the lamps discriminate them in Japanese; the other zone is upstairs, full of neon lights and luxury. This division is made to visualize a future where the discrimination is excessively high, the middle-class has disappeared, and the cultures are completely merged.

This time, the environment required a vast amount of assets, and it was the last created, so a few more time would be very helpful, but this issue will be addressed in the next section.

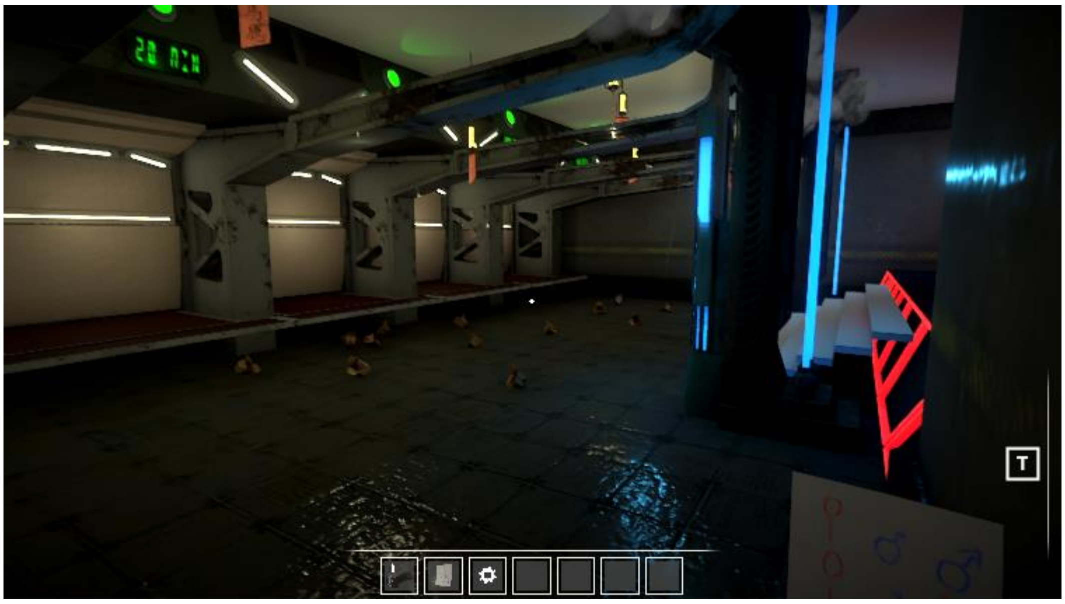


Figure 66: Dystopian future in Contact

6. Conclusions

Basically, this section includes an assessment of the project completion, and it is made keeping in mind every goal defined in the beginning of the report.

The final game is just a demo which has a very short and easy story that can be played in less than an hour; this simple story plot made the project work more diverse, complementing the art and programming. Nonetheless, despite of being a simple story, the gameplay requires the player to think a little bit because the puzzles could be difficult for someone.

One of the aspects usually forgotten is the navigation between menus, this was a challenge during the development, but finally there is a smooth system that allows the player feels comfortable before starting the adventure.

The final aspect of the project in terms of art is very satisfactory, there are around 80 realistic assets that obviously would have been had better quality if the workflow had included high poly meshes.

In the last weeks of development, it was decided to add more feedback elements, because it did not feel that the character was taking something when interacting, and in order to fix that, an animation and a sound effect were applied to the interact action.

As it has been mentioned, the demo is formed by 3 main puzzles, but all the programming structure has been done to be scaled with more puzzles.

Finally, and outside the goals, the project is valid for showing it in a personal portfolio, the effect can be reusable and reconfigured to do different effects, and there are a lot of assets that could be very useful in a future.

7. Bibliography

7.1. References

[Brackeys, 2017]: Last Access May 17, 2019,

https://www.youtube.com/watch?v=_nRzoTzeyxU

[Chyr W, 2013]: Last Access May 8, 2019,

http://williamchyr.com/2013/11/unity-shaders-depth-and-normal-textures/

[Esteban O, 2019]: Last Access May 19, 2019,

https://github.com/oskarter9/DegreeFinalProject

[Hocking C, 2007]: Last Access May 8, 2019,

https://clicknothing.typepad.com/click_nothing/2007/10/ludonarrative-d.html

7.2. Tools

[Blender]: Last access May 19, 2019,

https://www.blender.org/

[Photoshop]: Last access May 19, 2019,

https://www.adobe.com/es/products/photoshop/free-trial-download.html

[Substance Painter]: Last access May 19, 2019,

https://www.substance3d.com/products/substance-painter

[Unity3D 2018.2.9f1]: Last access May 19, 2019,

https://unity3d.com/es/get-unity/download/archive

[Visual Studio]: Last access May 19, 2019,

https://visualstudio.microsoft.com/es/

7.3. Webs

[Artstation]: Last access May 19, 2019,

https://www.artstation.com/

[FreeSound]: Last access May 19, 2019,

https://freesound.org/

[Github]: Last access May 19, 2019,

https://github.com/

[MP3 Cut]: Last access May 19, 2019,

https://mp3cut.net/es/

[Substance Share]: Last access May 19, 2019,

https://share.substance3d.com/

7.4. Assets

[PostProcessingStack]: Last access May 19, 2019,

https://assetstore.unity.com/packages/essentials/post-processing-stack-83912

[TextMeshPro]: Last access May 19, 2019,

https://assetstore.unity.com/packages/essentials/beta-projects/textmesh-pro-84126