CREATION OF OPTIMIZED HYPERREALISM **FOR UNITY**

Author: *Valentín Gallego López*

Tutor: *Inmaculada Remolar Quintana*

Videogames Design and Development degree

UNIVERSITAT JAUME I

SUMMARY

The search for graphic quality in Unity is a complicated task to get the goal of achieving a hyperrealistic visual quality. For this, I have design an interactive demonstration called La Torre, where we can control a character in third person, demonstrating the potential of Unity to work with hyperrealistic graphics both in cutscenes sequences and during the gameplay.

It will be possible to cross the main stage and the player will have a certain freedom to explore and observe the breadth of detail of the scene, with an optimized configuration in Unity to support this graphic level during the whole demo.

This document intends to indicate the process to achieve this graphic quality showing the proper use and, in order, each one of the different resources presented by the engine such as shaders, lighting or FX and Post-effects; those allow to achieve these graphics maintaining a good performance throughout the demo.

And the creative process through which has been designed, modeled, sculpted, animated and textured every element that can be seen during the demo, such as the protagonist character Baltor, or the architecture and furniture that covers the scene.

KEYWORDS

Shader, hyperrealism, textures, performance, modeling and lighting.

INDEX

FIGURES INDEX

CHAPTER 1:
TECHNICAL PROPOSAL

1. INTRODUCTION

The Unity engine presents a notable difficulty to achieve a hyperrealistic graphic quality, unlike the cartoon aesthetics; which is simple, caricatured and easier to get because the engine default settings serve as a starting point to work with this aesthetic without having to adjust any internal Unity options. Given the opposing difficulty of both styles, Unity is not the preferred engine to work in games with a laborious graphic section.

Therefore the intention of this project is to develop an interactive demo and with it the manual, which will define the activities necessary to build the demo and optimize it, so one can play fully with a good performance having some hyperrealistic and detailed graphics. The whole process that will be explained to achieve this graphic quality, aims to show that Unity can be a rival to the competition, like the Unreal Engine, where much more facilities are given in the graphical adjustment than in Unity.

The main idea of the demo would be the view of a stage by controlling a character in third person. During the demo it will be possible to appreciate the environment, its visual aesthetics, and the quality of modeling and animations.

In the demo cutscenes sequences will be presented with no loading times between them and the game mode. During the game mode it's intended to give the player freedom to explore the environment naturally and to observe the variety of objects and models that are throughout the scene.

After the construction of the demo, this TFG is presented as the manual where, by sections it's explained each process carried out to achieve Unity support hyperrealistic graphics, showing in detail the results and processes used for the development of this interactive demo.

2. RELATED SUBJECTS

3D design	- VJ1216
Engines of videogames	- VJ1227
Art of videogames	- VJ1223

3. PROJECT OBJECTIVES

- Design an interactive demo which works optimally on Unity and presents the best quality possible in hyperrealistic graphics.
- Accomplish a cutscene without loading times and visual quality equal to or better than the gameplay of the demo.
- Accomplish a third-person action gameplay that shows professional quality.

4. PLANNING

The process to build the demos will go through several stages that can be divided into three main stages:

Stage 1: Realize a study of the features and tools offered by Unity, and find out how to optimize them to achieve the best efficiency.

- Find out how to achieve the best shaders configuration for realistic materials.
- Work options such as occlusion culling to improve performance.
- Study the forms of lighting and shading used by Unity and make the most of them.
- Find out how to use effects and post-effects optimally visually and in a good performance during a playable game.

The estimated time would be about 75 hours approx.

Stage 2: Design and model playable elements like scenarios, characters and objects.

- Design an attractive and high-quality gameplay.
- Work models, materials, textures and animations in order to achieve a professional end result.

The estimated time would be approximately 105 hours.

Stage 3: Program certain scripts to develop the movement and gameplay necessary to get a satisfying game experience both in the visual and the gameplay.

- Realize the scripts that create the game flow.
- Program the character movement and player actions.

The estimated time would be about 85 hours approx.

Stage 4: To finish final evaluation work.

- Make final technical proposal.
- Make the final memory.
- Perform the defense of the project.

The estimated time would be about 35 hours approx.

STAGE	TASK	TIME	TOTAL
1	Find out how to achieve the best viewing / shaders configuration	22 h	22 h
1	Work options to improve performance	18 h	40 h
1	Study the forms of lighting and shading used by Unity	20 h	60 h
1	Find out how to use effects and post-effects optimally visually	15 h	**75 h**
2	Design an attractive and quality gameplay	25 h	25 h
2	Work models, materials, textures and animations in order to achieve a professional finish	80 h	**105 h**
3	Realize the scripts that create the game flow	35 h	35 h
3	Program character movement and player actions	50 h	**85 h**
4	Make final technical proposal	7 h	7 h
4	Make the final memory	25 h	32 h
4	Perform the defense of the project.	3 h	**35 h**
			300 h

Figure 1: Initial planning.

5. TOOLS AND PROGRAMS

The programs that I will use for the development of the demo are divided depending on the purpose of its use:

For the work in the design, modeling, animation and texturing will be:

- **3Ds Max[3]:** base program for modeling in low-poly and from which the final models are obtained to use in the game engine.
- **TopoGun[9]:** program to work retopology to create a low-poly based on a high-poly of which will acquire its shape.
- **Unfold3D[10]:** program to unwrap the textures on the surface of low-poly models.
- **xNormal[11]:** program that allows to make the bake of low-polys with its high-poly to obtain normal maps and occlusion maps.
- **Maya[4]:** program to adjust the animations of the Autodesk family just like 3Ds Max.
- **Zbrush[5]:** program to sculpt high-poly models.
- **Photoshop[6]:** program to perform the texture of ingame models.
- **Quixel - nDo2[7]:** Photoshop plugin that allows transformations of certain textures to obtain different ones.

To program and build the demo it will be:

- **Unity[1]:** base game engine where the demo will be built.
- **Visual Studio[2]:** program to program the demo scripts.

CHAPTER 2: DEVELOPMENT

At the beginning of the project the main matter is thinking about the dimensions that could reach encompassing the demo. This allows us to foresee how much it will costs the future stages of the development and to avoid going blind in both the design and the programming of the demo.

The first step is to look for attractive ideas to show that can be displayed in the demo in real time and achieve on hyperrealistic 3D graphics inside the engine. This encompasses both the playable mechanics and the artistic elements of the game such as the stage and the character.

The idea that aroused the attempt to create an interactive demo with a photorealistic quality was the interest to replicate the same quality seen in the official demos released by Unity where it demonstrates the graphic power of its engine. The mentioned demos were *Adam* and *Blacksmith*.

Two initial ideas were initially contemplated:

- A demo with fantastic atmosphere suggested by the official demo of *Blacksmith*, with a focus on the use of action mechanics.
  - Possibility to focus on the design of characters with rudimentary clothes and with the possibility to experiment with armor.
  - Disadvantage of having to work with animations with greater care for humans.
  - Linear design of the stage, focused on the use of cutscenes.

- A demo with mystery context where the design is verisimilitude with the present, the mechanics mainly focus on the resolution of puzzles.
  - The focus of the demo would be to design a character that was inside a building that would function as a puzzle and the function of the character would be to escape from there solving puzzles and being surprised by strange phenomena that are happening to him along the demo.
  - Animations would be more basic and the rhythm of the game more paused.

Both demos would share the requirement to achieve the entire visual section to be recreated in photorealism, where the Unity engine were sized and showed its possibilities within the reach of the average user of this engine, to give a graphic quality at the height of the engines like Unreal Engine 4 or CryEngine competition.

The final idea ended up being an intermediate base between both conserving the aesthetics of the first one but contributing with the interior recreation of the second idea. Thus begins the design of the demo La Torre.

1. FIRST PHASE: CONCEPT ART

The first phase of this development begins with the design on paper of the scenarios and the protagonist of this demo. Several sketches were worked on the concept of the stage and exploration that would carry out the personage. The concept art technique is necessary to have a clear idea of the designs to model in the future, avoiding an unnecessary time spent imagining how each modeling could be better without a general cohesion than if it were obtained from the concept arts. The technique consists in making sketches on paper that later pass to Photoshop to draw the range of colors for each element on more polished sketches of each design.

1.1. ENVIRONMENT

The initial scenario was conceived as a fairly large space to go through the demo but the increase of interest in recreating the interiors of the game, made it be thought again three times the stage before finding the definitive.

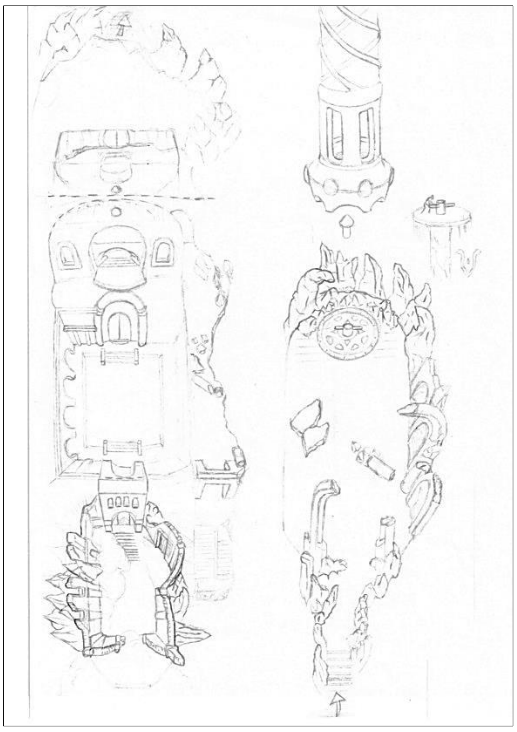
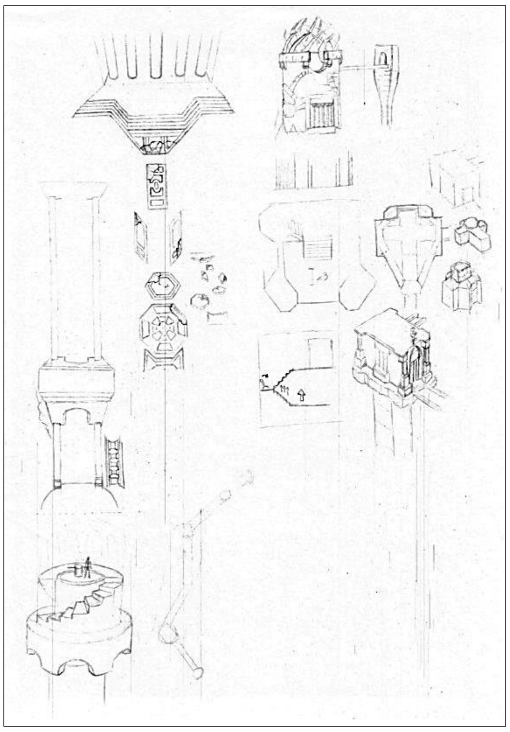



Figure 2: Left image is the initial zone of the first sketch of the scenario and right image is the next zone.

The first sketch of the stage had a large size and was divided in two areas, one by land and one crossing the sky. The zones were going to be very wide so they could see enemies. It was a very complicated idea.

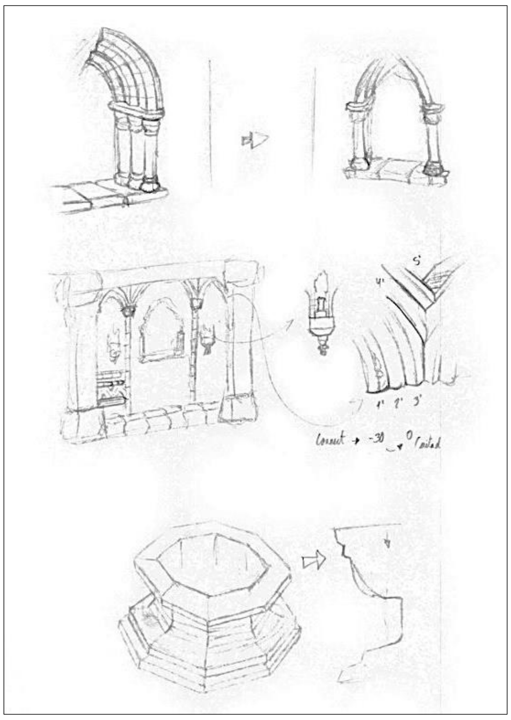
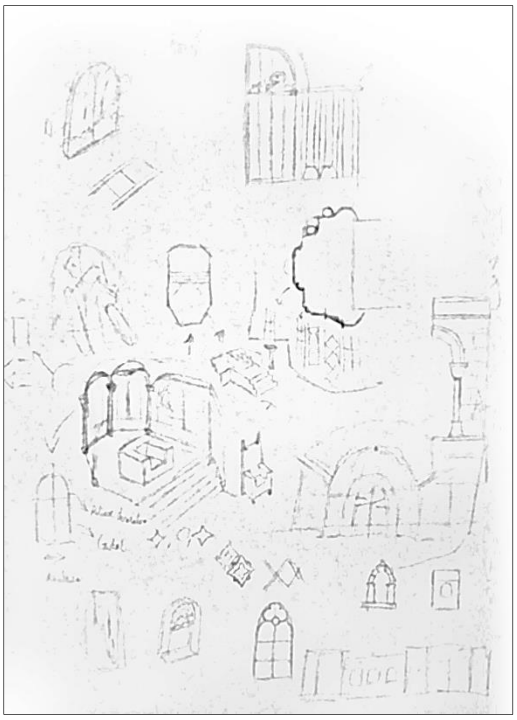



Figure 3: Stage interior designs.

The interiors were designed drawing several options that were giving more importance to the rooms of the stage and each time had less exterior areas.

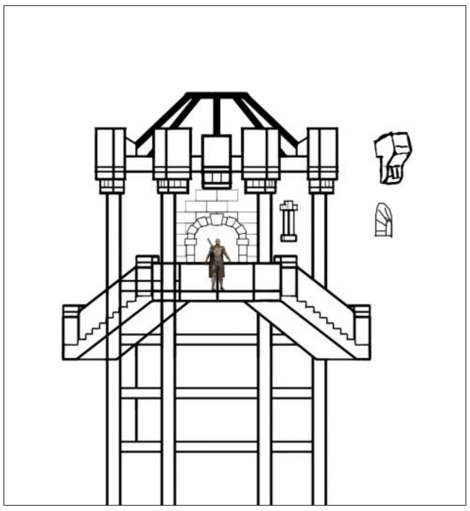
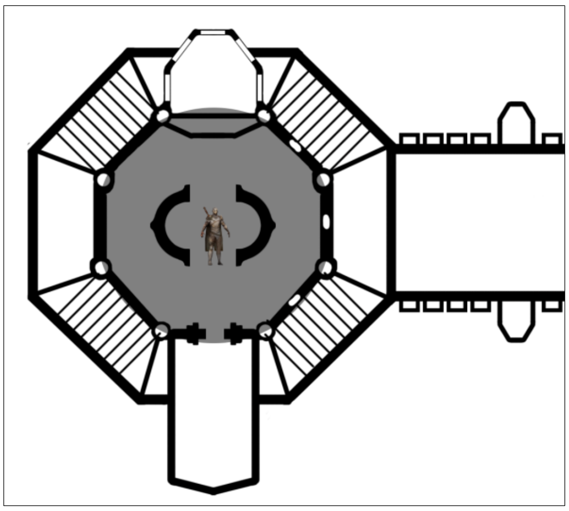



Figure 4: Left image is the Front view and right image is the Top view of the stage design like tower.

The last design was conceived like a very tall tower with a staircase that descends around it to the ground, where there would be a bridge to go to the next area and the demo would end.

The visual style is similar to the games of the Dark Souls saga. The appearance of the structures is inspired by the scenes of the saga, the rooms of the demo maintain that characteristic style that presents recharged Romanesque elements that approaches the gothic style.

1.2. CHARACTER

The character that appears in the demo was devised as a warrior that seemed imposing but had his human side; this idea is also inspired by the same saga of games, Dark Souls. The armor he wears was referenced in several armors of these games reaching a style of his own after several sketches.

Each armor element was individually worked to give the most attractive shape possible. Several versions were made until the most optimal and each one combined with the others so that the character was well complemented.

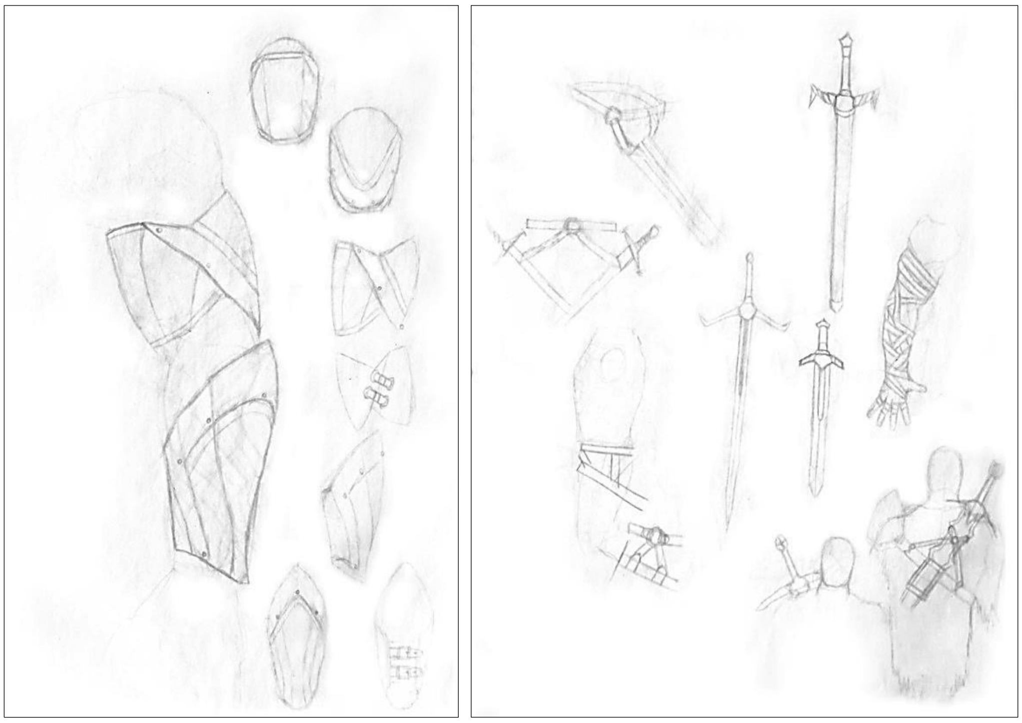


Figure 5: Left image is the final design of armor arm; right image is the sketch of weapons and belts.

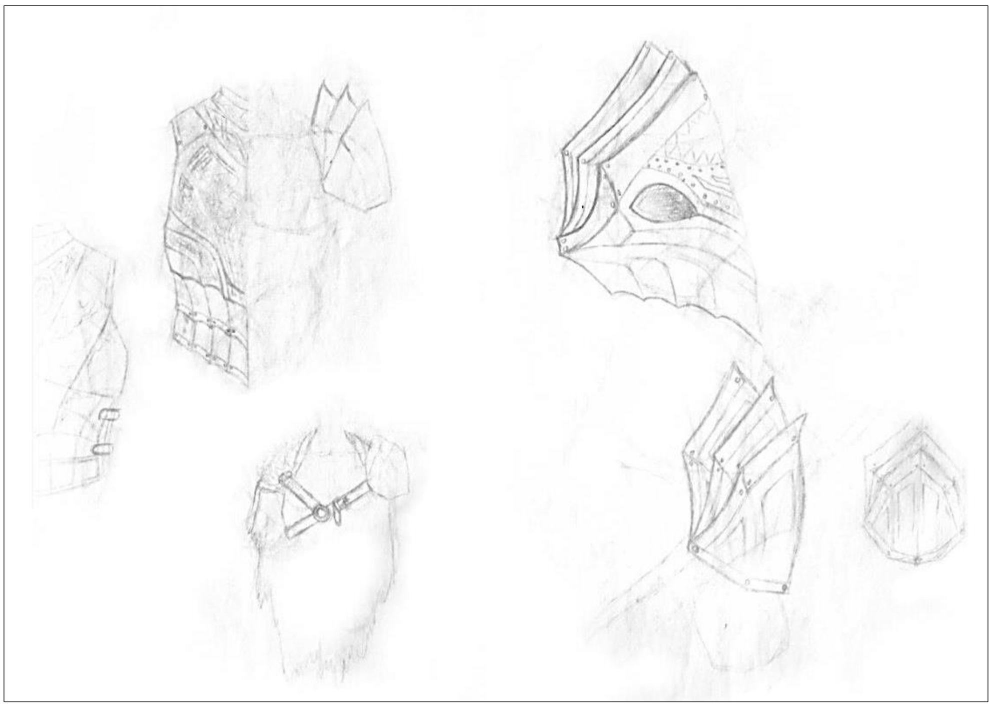


Figure 6: Chest armor and shoulder protector sketch.

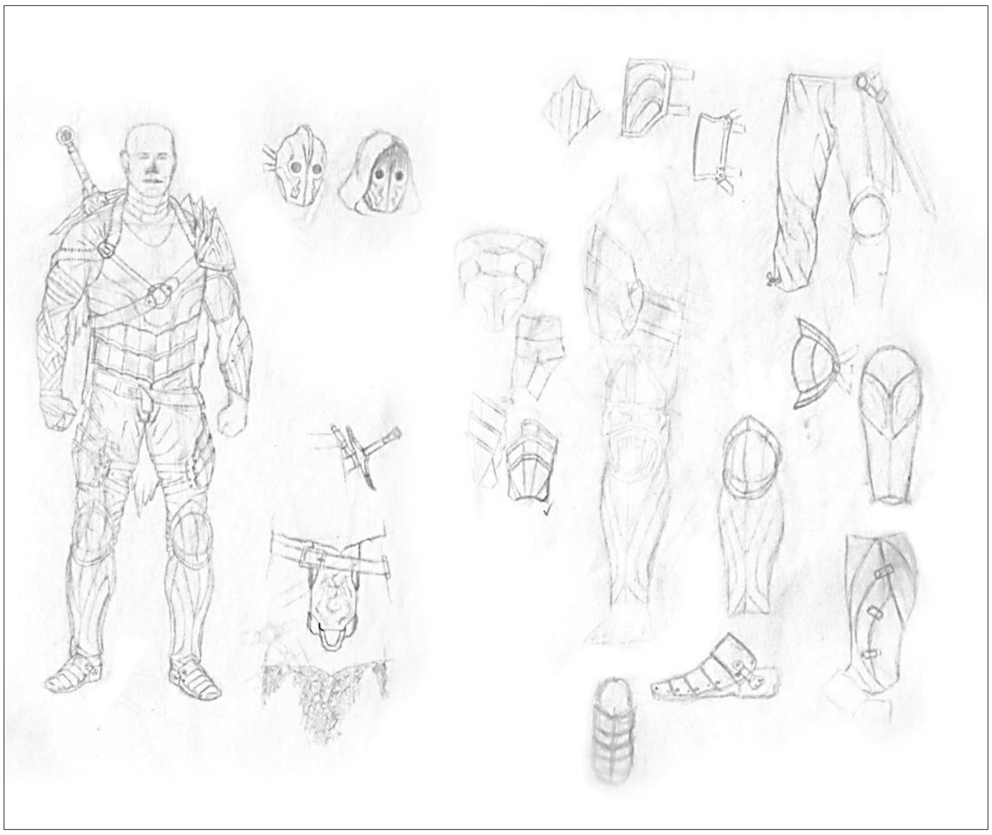


Figure 7: Baltor final sketch with alternative pants designs.

2. SECOND PHASE: MODELING & TEXTURE BAKING

2.1. CHARACTER MODELING - BALTOR

In this phase begins the 3D design of the character. Once the character concepts are finished, the 3Ds Max modeling with the design of a basic figure called *base mesh*, this object must define the most basic forms of the human being with very few polygons and in which each polygon must occupy an area similar to the rest.

In order to start from a basic cube, a division of the progressive mesh begins, while several sections of the cube are extruded forming the torso and extruding the other body parts, including the head.

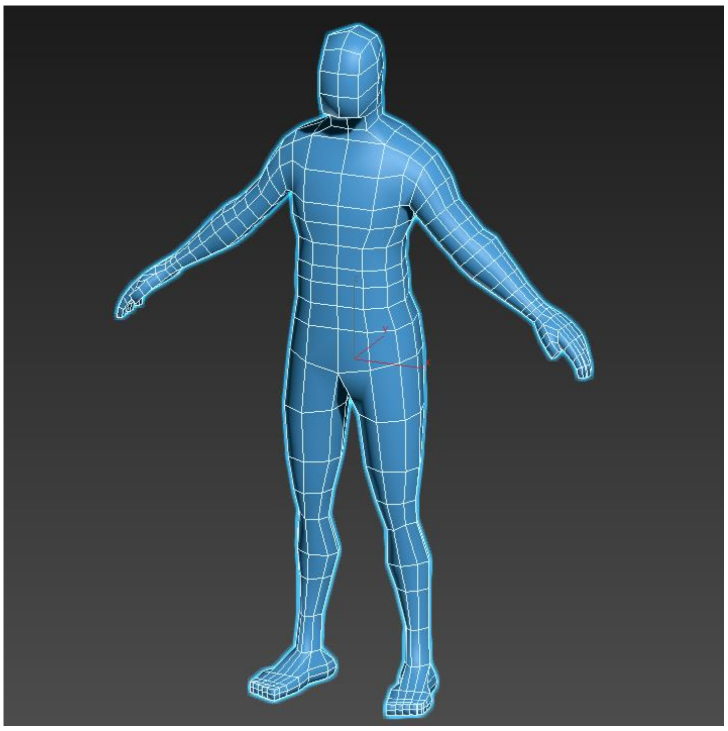


Figure 8: Base mesh of Baltor character.

Once finished the *base mesh* phase, it will be used later to be able to sculpt a high-poly in Zbrush. To take it to Zbrush it is necessary to export the mesh in .obj format, which is the only format that Zbrush reads when importing.

To begin this stage it requires a lot of knowledge about human anatomy to achieve a realistic sculpting, both in musculature and in factions and structural elements of the human body. The key is to get as many references as possible to define a realistic body.

The sculpture begins by defining the muscles of the torso abruptly, giving shape to where the rib cage should be placed and where the stomach and abdominals would be. Then the clavicle, shoulders and neck are defined in the same way and later sculpted arms and legs to finish working hands and feet with greater detail for later retopology.

For the head a method of sculpture is worked that requires defining at the beginning the skull of the character in factions, such as the hollow of the eyes, the nose, cheekbones and jaw; from there, several muscles that define the characteristic features of a human face will be sculpted. To adjust the eyelids, the volume of these is created using two spheres in each hole that simulate the white eyeball with a size proportional to the eyes with respect to the face, thereby part of the mesh is sculpted out to fit over Of each sphere and simulate the eyelids, when the basic shape of the face is done is missing add the ears by stretching the mesh out with the Zbrush move tool to then adjust that part the mesh and give it the necessary ear shape.

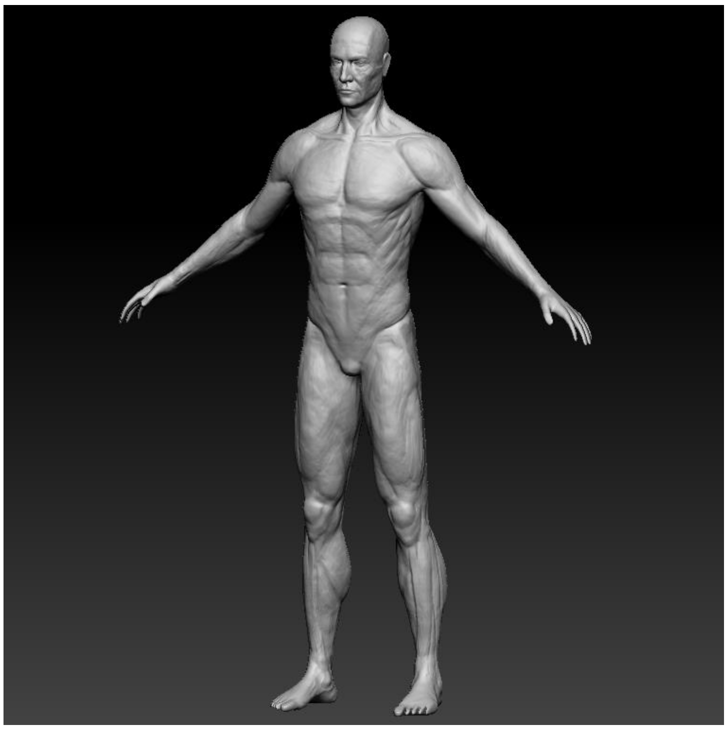


Figure 9: First high-poly of Baltor character.

Once the first sculpture in Zbrush is finished, the mesh is passed to the TogoGun program. There with the high-poly of the character, the technique called retopology is realized. With it, a new low-poly mesh from the high-poly version is obtained. The process consists in using the high-poly model as a reference, this is represented as a base object on which one builds the new mesh. The design of the new mesh is started by creating squares on the surface of the high-poly that are intertwined to form the new low-poly mesh. Depending on the area of the body the squares are more closely united and forming arcs of strips of polygons that fit better the shape of the mouth or nose of the character.

This new mesh, unlike the initial *base mesh* of the character, has more polygons better distributed along the mesh to better define the forms of the high-poly. Therefore areas such as the hands or the face have a greater amount of polygons Allowing to preserve with more fidelity the contour of the high-poly being a mesh with less polygons, also when the character is going to be animated the meshes will adapt properly when being stretched or contracted in zones like the mouth, the fingers or the eyes.

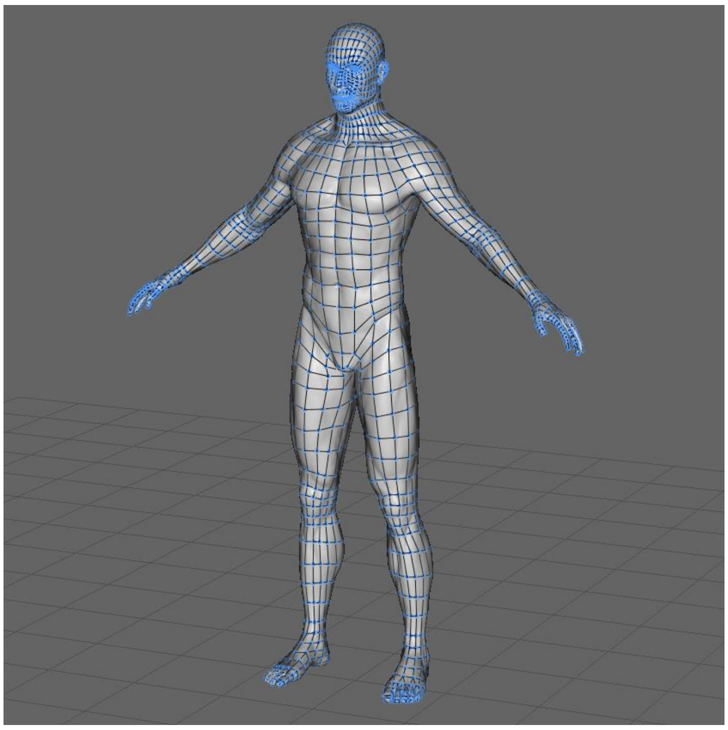


Figure 10: New low-poly made in TopoGun.

Once the mesh is finished, it's taken back to Zbrush. But this time it's something different, instead of carrying alone the low-poly .obj file, TopoGun can get a new high-poly using the new low-poly mesh and maintain the original high-poly appearance. The low-poly is obtained by saving the file in .obj format and the new high-poly is calculated through a program tool that uses the new low-poly as a base and subdivides it to the desired level, thus obtaining the new mesh in high in .obj.

In Zbrush, the merging of both meshes is done using the low-poly subdivided up to the specified level in TopoGun, at that time the high-poly is imported by unifying both meshes and passing the high-poly to subdivision of the low-poly.

Now the musculature of the body will be refined and detail like wrinkles or skin folds can be added to define a more realistic high-poly mesh of the character.

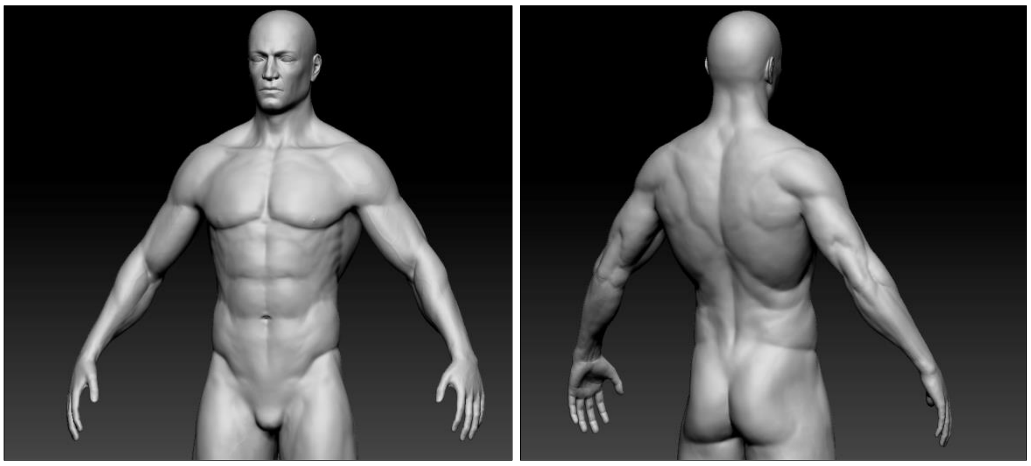


Figure 11: Final Baltor's body high-poly made in Zbrush.

When the model of the character's body is finished, by means of basic meshes like the Zbrush spheres, the modeling of the armor and the character's dress is started. Each element is modeled like an independent mesh and in the case of the parts that should be two symmetrical elements in the armor, only one is sculpted at the beginning and with the final mesh a symmetry mesh is done to complete the model. The process is similar to that seen with the body of the character. But it is recommended to pass two times by TopoGun, one to give a mesh more similar to what would be the *base mesh* of the character to adapt the high-poly then subdividing the mesh it can give detail of equal way to the whole object and finally a new mesh will have to be done and this is the final one. It will be possible to make greater use of the amount of polygons to work with parts of the armor curves or with pretension to suffer twisting and to stretch their polygons.

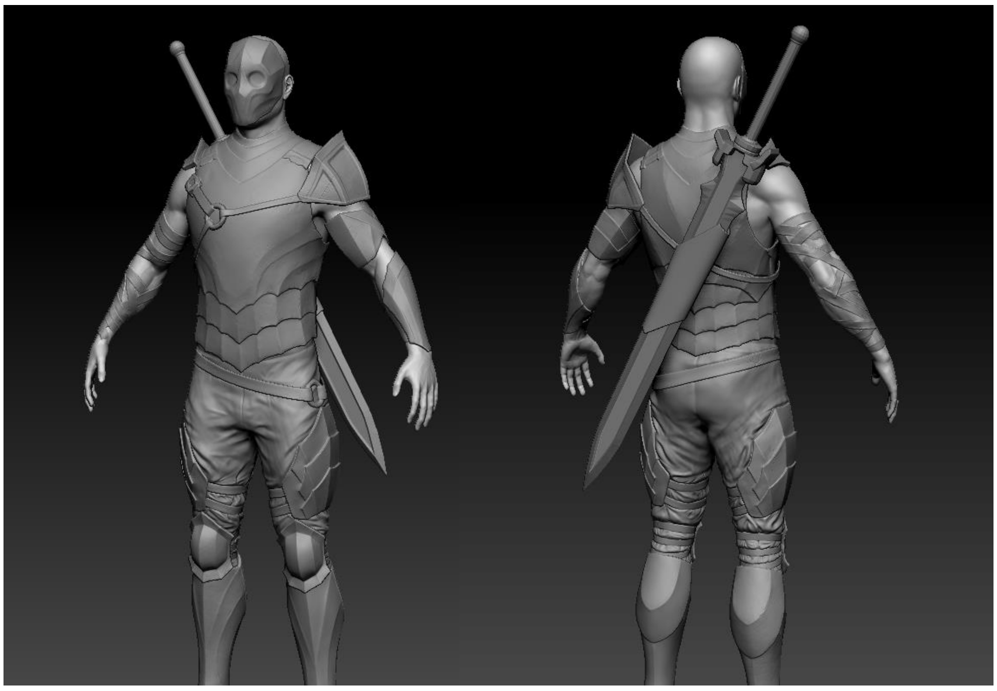


Figure 12: Final Baltor's body high-poly made in Zbrush.

When it's sculpted the whole character, each low-poly part is exported to 3ds Max. There, the extra modeling will be done like final eyes modeling. The eyes must occupy the same space like the prototype spheres created in Zbrush, but this time the spheres have an amount of 32 segments and each eye has a supplementary sphere above that unlike the first this works the form of the cornea and the anterior iris and the pupil. Then a 3d line is added that covers through the edge of the lower eyelid to the tear and then make an envelope and convert it to polygons, thus producing the brightness of the eye inside the Unity engine.

Then 3d plans are created to design the hair of the character for it is designed a small amount of planes with 4 to 6 polygons, the unwrap is made for them together and then the texture is painted in Photoshop. When it is done, only it's necessary to adjust the planes to manually making several clones of these and repeating them by the scalp, the area of the beard or already for the eyebrows and eyelashes, for it can be made use of the tools of 3Ds Max or return to use Zbrush and the move tool, which is quite useful for this case since it allows a greater precision for the artist in the placement of the planes.

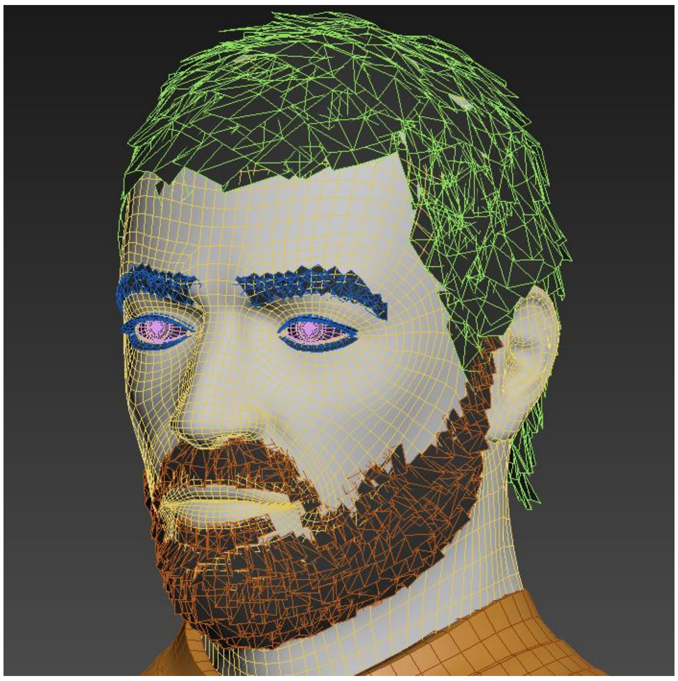


Figure 13: Baltor's new eyes and hair made in 3Ds Max.

Finally there are elements such as the sword and the knife which are designed aside in 3Ds Max and they are taken to Zbrush to work his high-poly and later TopoGun is used again to make the new low-poly of these extra elements of the character.

Having all the low-polys finished, they are taken one by one to another program called Unfold3D, this program allows unwraps easily on organic surfaces and curves like in the case of the mesh of the character. To open the mesh uvs only the program asks to signal the edge line, which must be used later to section the mesh and the program automatically adjusts its dimensions in 2d.

The resulting mesh is brought back to 3Ds Max and here, with the modifier Unwrap the shapes of each part of the uvs are adjusted to optimize the space for the textures. In this case several groups have been made with the different meshes of the character's armor to facilitate the process of texturization that will proceed later. The character ends up needing only the mesh of the head and the arms separately, discarding the rest of the body of the character, to optimize the model when animating it and taking it to Unity. Beside this object are the meshes of the armor divided into two parts, the part that is all the armor that is above the waist of the model and the other mesh is all that is below.

2.2. SCENARIO MODELING

The modeling of the scene was done in 3Ds Max by using blueprints previously made in Photoshop from the final scenario sketches.

Initially the stage was to be a series of interlaced corridors that would take the character by several instances or connected rooms but with clear differences between each one. Later the idea ended up receiving a reduction in its size and also added a more vertical approach to the scenario.

The blueprints made show a main room on the top of a tower of about three floors, this room opened onto a balcony that continued down a spiral staircase to the ground floor.

This idea evolved during the development of the demo, seeing that it was more feasible to work the room inside and to show more detail to the player than to work an exterior only visible for an instant. For this reason the tower was reimagined like a construction on top of a mountain and consisted of two rooms, a corridor and the balcony.

The largest room was thought octagonal to facilitate a 360-view of it during the gameplay of the demo. To model it was worked with a base cylinder with ten faces, from one of them began to work a section of the room that would later be repeated for its equivalent nine remaining sides, including both the corresponding roof portion and the part of the of the same sector.

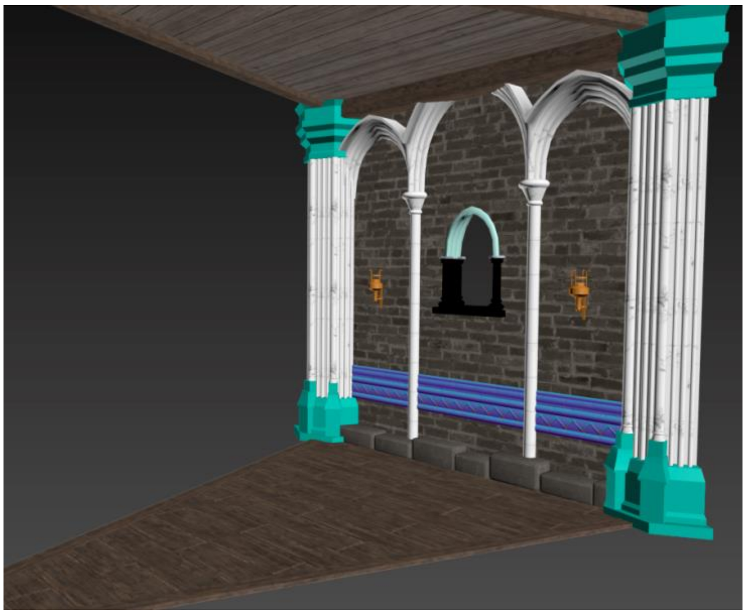


Figure 14: Sector model made in 3Ds Max.

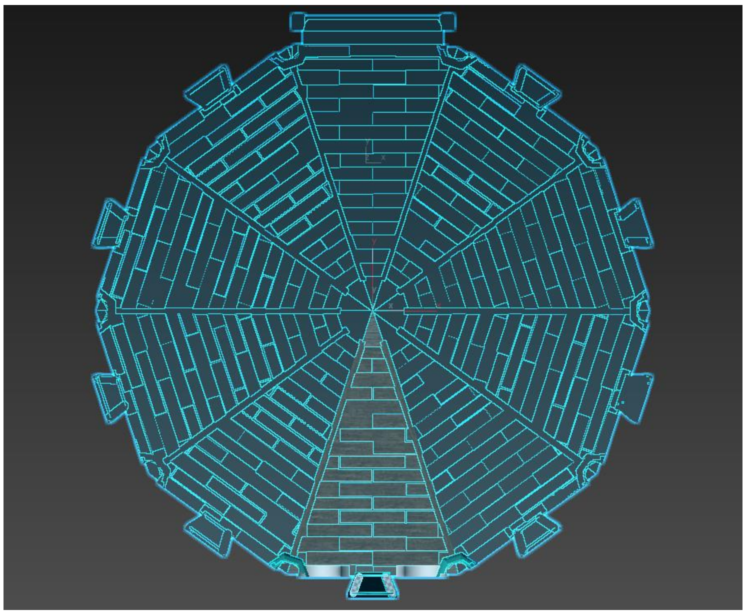


Figure 15: Top view of sector's circle.

This method allows to made more detail for each object belonging to a sector. Example the pillar of each corner carries a work similar to that shown in the modeling of the character. The abutment consists of several elements separated into two sets, the base of the pillar and the trunk part of this. Both sets are worked differently:

The bases are carried to Zbrush to sculpt the stone of the column in each element that forms it, adding details of crack, disintegrations and wear in the material of these, each part is worked individually and for the symmetrical elements is realized one symmetry with the finished model.

The trunk part is worked by the technique of tileable texture. This is understood by placing the uvs of this part of the model so that when the texture in tileable mode is repeated continuously filling the mesh with the texture vertically or horizontally, in this case, this is the first and facilitates the detail using a size of texture of 4096x4096 instead of being three times greater.

For the ground only five planks were worked and just like the bases of the pillars, which were sculpted one by one in Zbrush to then work the detail of old wood with their respective normal maps. The floor trick works with the orderly distribution of the tables but including them in a random order which implies that there are many more.

The wall was worked by dividing it into several elements: the brick wall, the window and the moldings (elements adjacent to the wall such as reliefs, arches and attached columns). All these elements used tileable textures, giving an excellent result and allowing the rapid use of these textures.

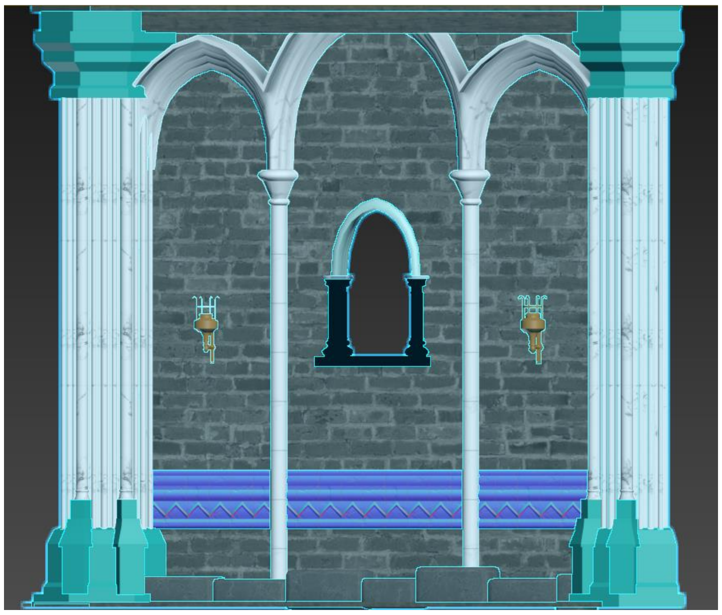


Figure 16: Moldings of sector's wall.

For the room it was used a different sector when designing an access door for the character. This door had a frame that used a tileable texture too, and a prototype door that forms part of a double door that would be obtained with the symmetry of the same.

In this room, a central fireplace was modeled inside the room formed by two pieces, and a series of elements like candelabras, tables, barrels and shields that decorated the scene.

Once the large room is finished, there is the hallway which was built in the form of pieces with an idea similar to that of the sectors of the large room but with the particularity, this time the repetition is done in line and not in circles. The idea is to design an extensive floor in one direction and on one of its sides is designed the base prototype wall that is repeated along this.

In this case, it is two types of wall that is at both ends of the row with a single half arch and which has two to get the symmetry in the repetition along the row of walls. For the wall, on the opposite side only a complete wall symmetry is required.

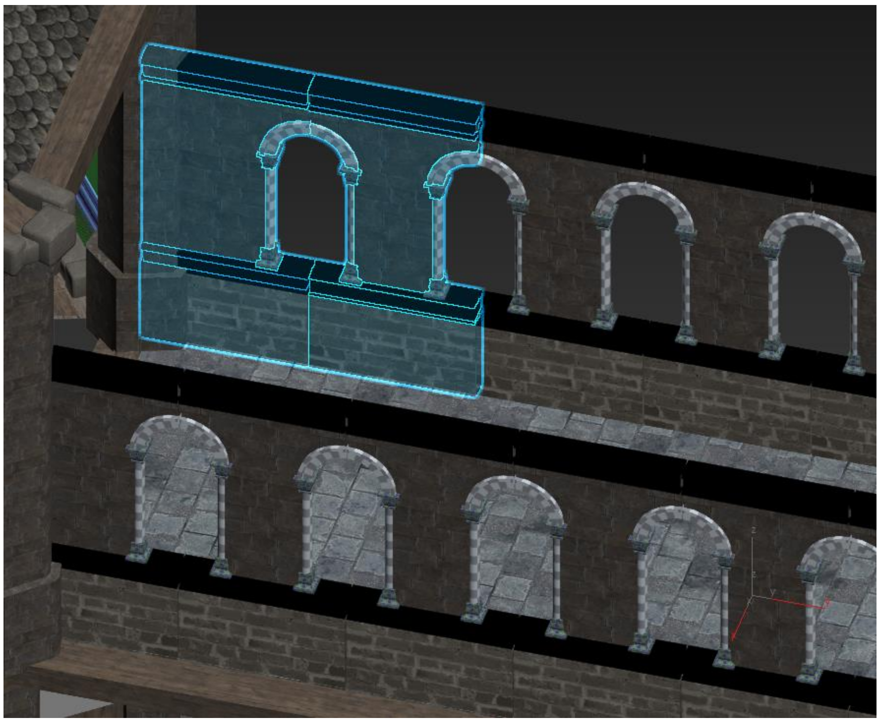


Figure 17: Hallway's walls.

And it remains to mention the small room, this presents a rectangular base, the main floor is a plane, with a cube deformed to show a staircase to a seat at the end of these. The walls repeat the format of the walls of the corridor where a base is designed from one side and is repeated for the following consecutive walls. And the texturizing method for these is also using tileable textures.

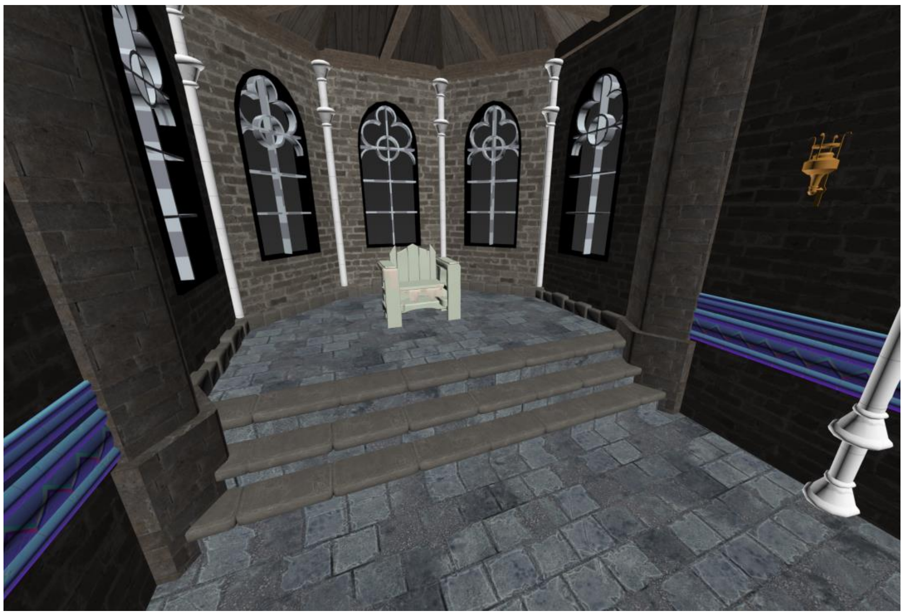


Figure 18: Small room's interior.

El exterior de la construcción está formado por una serie de pilares en la base, un par de travesaños y lejas en el tejado como toque final mostrando la salida de la chimenea en el tejado de la sala grande.

The exterior of the building consists of a series of pillars at the base, a pair of crossbeams and layouts on the roof complete the stage model, showing the chimney output on the roof of the large room.

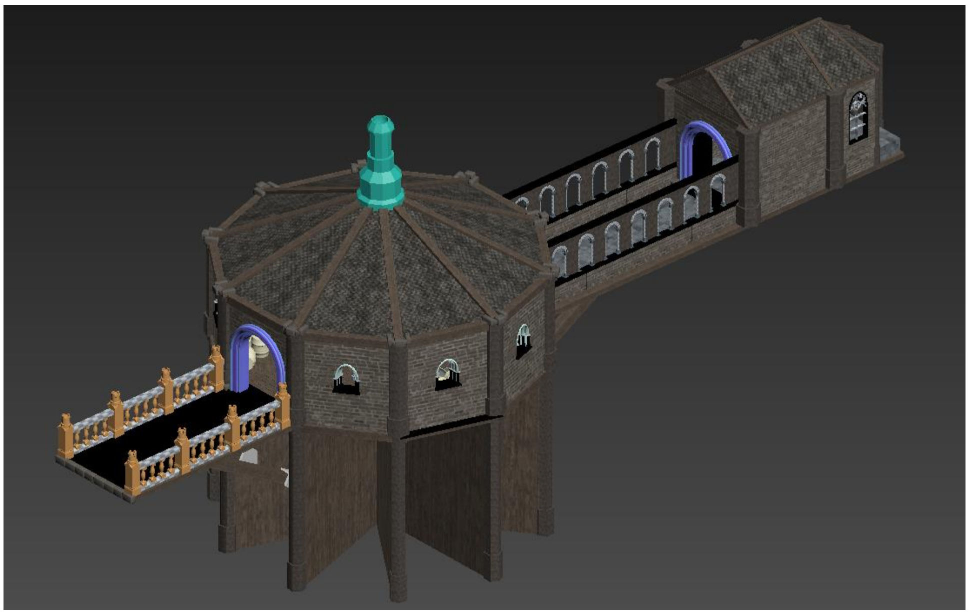


Figure 19: La Torre's exterior.

2.3. TEXTURE BAKING

Achieved everything up here begins the moment of making the bake for all models of both the character and the stage that have high-poly. This process allows to obtain the base textures to begin with the texturing phase, the first is to take the meshes to the program called xNormal that allows to obtain maps of normal and of ambient occlusion, these maps are obtained using the high-poly mesh like a reference for then imitate its surface with the low-poly and cage, which is the same low mesh but transformed with the 3Ds Max's Push modifier to overlay the same polygons of the low high-poly shape, can be simulated high-poly surface in real time using only the low mesh in the game engine.

With the new textures for each mesh the process is continued in Photoshop, to start texturizing the character. With the help of the Quixel nDo2 plugin is possible to draw more maps from the two basic ones obtained in xNormal, e.g. the normal map can be transformed in a cavity map, which is a basic texture to emphasize hard edges of the original surface of the high-poly in textures like albedo or specular alpha.

3. THIRD PHASE: TEXTURES

The textures needed by the game are those of PBR (Physically Based Rendering) materials, which are albedo, specular, normal, height, occlusion and emissive.

The *albedo texture* determines the color of the surface which would be equivalent to the diffuse map of the traditional materials before the PBR, but unlike this the albedo does not keep any value of brightness or illumination.

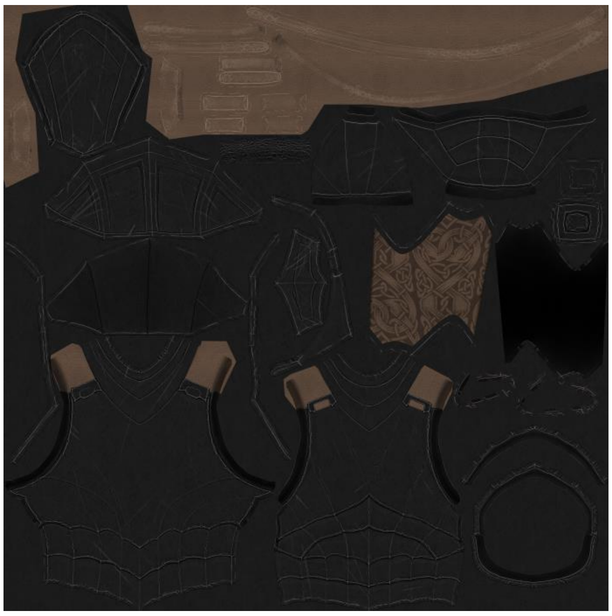


Figure 20: Albedo map of Baltor's superior armor part.

The *Unity specular texture* works to define the type of brightness and reflection exerted by light and surroundings on the material. The values of the specular allow to define the type of brightness in intensity and color as well as the reflection of the surroundings. The reflection is defined, using an alpha in the image that specifies the smoothness value of the specular and that defines like polished the surface, i.e., the whiter is the specular alpha easier to reflect the environment on the surface of the object that carries that texture in the material.

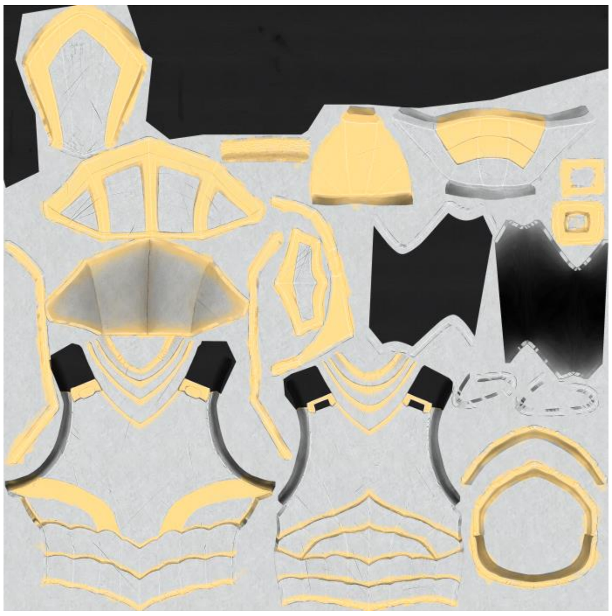


Figure 21: Specular map of Baltor's superior armor part.

The *normal map* texture allows to simulate the volume and roughness of the surface that would have the object's high-poly over the low-poly model. To obtain this texture it is necessary to work the bake of the high-poly with a good low-poly and a cage that covers the whole model and allows to extract all the information on the original volume of the mesh.

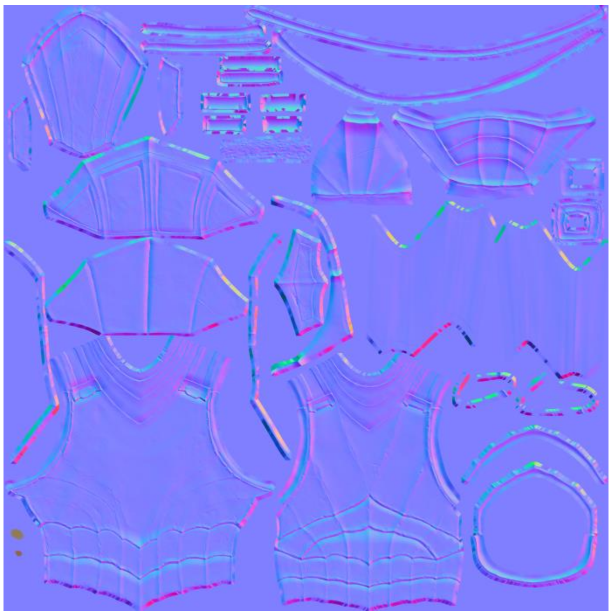


Figure 22: Normal map of Baltor's superior amor part.

The *occlusion map* or *ambient occlusion* is a supplementary map that allows to define the shadow that is between two very close surfaces. This map is also an essential part of the bake that can be used like a multiplied layer that helps define the texture of the albedo along with others such as the texture cavity.

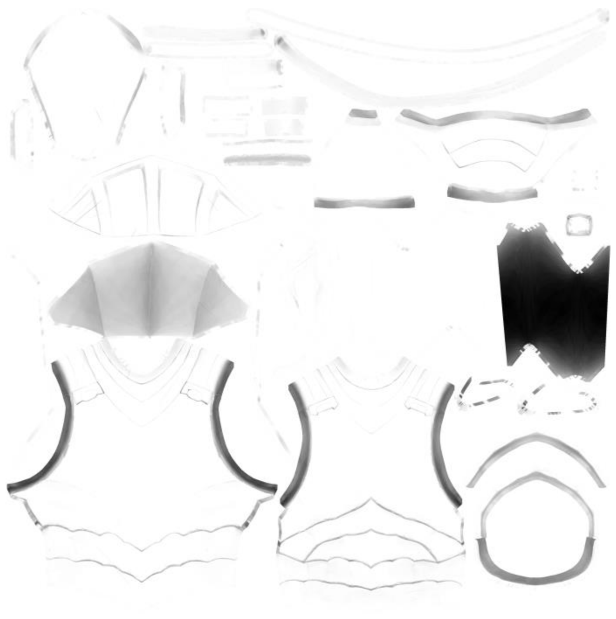


Figure 23: Occlusion map of Baltor's superior armor part.

The texture *height map* (also known as parallax mapping) is a concept similar to the normal map, however this technique is more complex and therefore also more expensive. Height maps are generally used along with normal maps to give additional definition to surfaces where these textures are responsible for producing large protrusions.

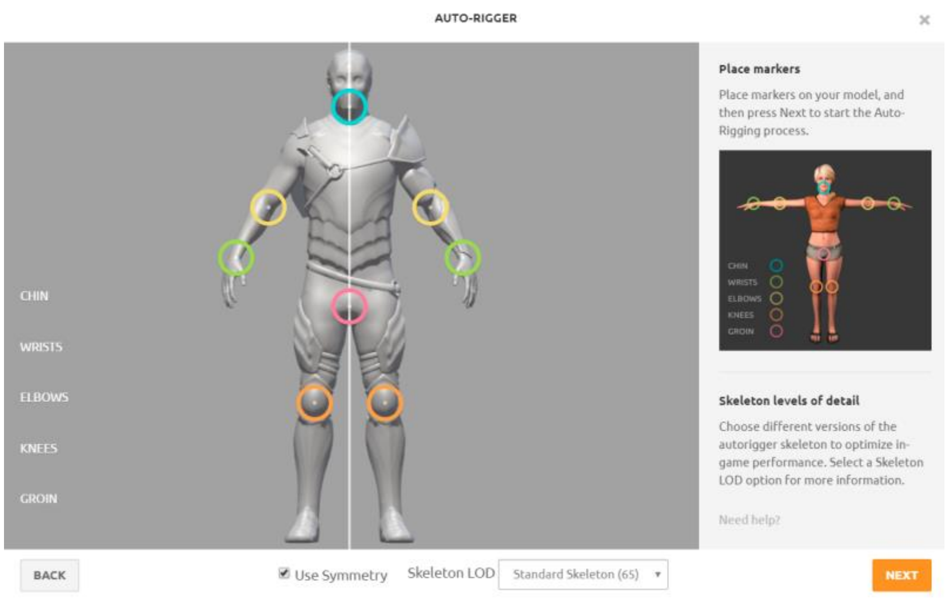
The *emissive map* works the self-illumination of the material, it controls the color and the intensity of the light emitted from the surface of the object with that material. The values are moved in a color scale where the total black indicates zero self-illumination and the white is the maximum self-illumination of the object.

For making these textures is necessary to follow a priority order when working through the traditional mode using Photoshop. The first thing is to use the normal map and the ambient occlusion in the same .psd, from these they work separately the albedo imitating the basic colors of the surface adding the multiplied effect of ambient occlusion and the cavity superposed on the same base texture or the color used to mimic the surface. Then it is necessary to calibrate the degree of darkness of the texture of the specular to know how reflective the surface of the material is, it is convenient to respect the official tables that gives Unity to take them like a base to work, and after this the alpha is measured by eye using the smoothness bar of the material to find the final brightness and then use that intensity in Photoshop to paint the alpha that will be coupled to the specular texture. The emissive only requires the use of a black base and then adds a brighter color so that the material can emit light in that color, how much brighter is the color, more powerful is the emitted light.

4. FOURTH PHASE: ANIMATION AND RIGGING

The animation of the character has been made by the automatic rigging of the page Mixamo 2.0[12].

This page provides the user the possibility by importing a character to fit a skeleton onto a humanoid body by assigning 5 key points across the entire structure, chin, elbows (symmetrical, automatically adjusts the opposite elbow), wrists, knees and pelvis / root of the skeleton.



Place markers

Place markers on your model, and then press Next to start the Auto-Rigging process.

Skeleton levels of detail

Choose different versions of the autorigger skeleton to optimize in-game performance. Select a Skeleton LOD option for more information.

☑ Use Symmetry

Skeleton LOD: Standard Skeleton (65)

Figure 24: Screenshot of Mixamo rigging window.

Another advantage that the animation by Mixamo presents is that the page has a wide assortment of animations that the new skeleton of the character can be easily chosen and adapted. Even for some animations it shows customization options to separate the arms more or less depending of the interest for a different pose, and variables that subtly change the aspect or speed animation.

Once made the selection of animation these are exported in .fbx along with the model with the skeleton in another .fbx separately, this format, unlike the format .obj ,allows to store data to part of the mesh like in this case the frames of the animations. To finish the animations that have not gone to taste are passed through 3Ds Max and are reset depending on what the author wants. In this case, the animation of sitting was adjusted so that the pose better match the anatomy of the character and synchronized with that of getting up.

5. FIFTH PHASE: UNITY

The assembly of the scene in Unity, is the stage of import and placement of the models, the animations, the effects, the materials and the textures, configuring each object in the scene for an optimal use and to continue with the code programming of the demo and finally the use of tools offered by the engine like lighting or post-effects to achieve photorealism.

5.1. UNITY SCENE

When starting a new scene in Unity, the engine prepares has by default a series of adjustments with basic-minimum quality, which must be changed if the scene is intended to achieve photorealism. So the first part is to select new Unity options that unlock and adjust the graphic quality to achieve.

5.1.1. Linear vs gamma

The Unity Editor offers both linear and gamma workflows. The linear workflow has a color space crossover where textures that were authored in gamma color space can be correctly and precisely rendered in linear color space.

Textures tend to be saved in gamma color space, while shaders expect linear color space. As such, when textures are sampled in shaders, the gamma-based values lead to inaccurate results. To overcome this, Unity can be set to use an RGB sampler to cross over from gamma to linear sampling. This ensures a linear workflow with all inputs and outputs of a shader in the correct color space, resulting in a correct outcome.

To specify a gamma or linear workflow, go to *Edit > Project Settings > Player and open Player Settings*. Go to *Other Settings > Rendering* and change the Color Space to Linear.

Gamma Workflow

While a linear workflow ensures more precise rendering, gamma workflow is faster than it.

Linear Workflow

Working in linear color space gives more accurate rendering than working in gamma color space.

Linear color space can work if the textures were created in linear or gamma color space. Gamma color space texture inputs to the linear color space shader program are supplied to the shader with gamma correction removed from them.

For colors, this conversion is applied implicitly, because the Unity Editor already converts the values to floating point before passing them to the GPU as constants. When sampling textures, the GPU automatically removes the gamma correction, converting the result to linear space.

These inputs are then passed to the shader, with lighting calculations taking place in linear space as they normally do.

Differences between linear and gamma color space

When using linear rendering, input values to the lighting equations are different to those in gamma space. This means differing results depending on the color space. For example, light striking surfaces has differing response curves, and image effects behave differently.

Light fall-off

The fall-off from distance and normal-based lighting differs in two ways:

When rendering in linear mode, the additional gamma correction that is performed makes a light's radius appear larger.

Lighting edges also appear more clearly. This more correctly models lighting intensity fall-off on surfaces.

Linear intensity response

When gamma rendering is used, the colors and textures that are supplied to a Shader already have gamma correction applied to them. When they are used in a Shader, the colors of high luminance are actually brighter than they should be compared to linear lighting. This means that as light intensity increases, the surface gets brighter in a nonlinear way. This leads to lighting that can be too bright in many places. It can also give models and scenes a washed-out feel. When linear rendering is used, the response from the surface remains linear as the light intensity increases. This leads to much more realistic surface shading and a much nicer color response from the surface.

Linear and gamma blending

When blending into a framebuffer, the blending occurs in the color space of the framebuffer.

When gamma space rendering used, nonlinear colors get blended together. This is not the mathematically correct way to blend colors, and can give unexpected results, but it is the only way to do a blend on some graphics hardware.

When linear space rendering is used, blending occurs in linear color space: This is mathematically correct and gives precise results.

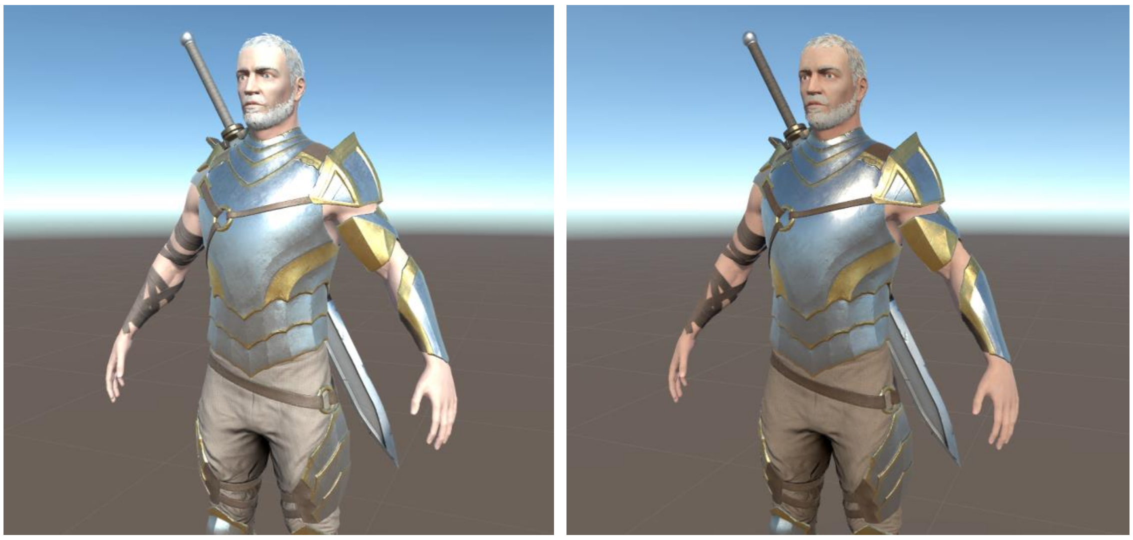


Figure 25: Left image is with Gamma workflow and right image is with Linear workflow.

5.1.2. Frame Debug

To work with the scene it’s necessary to know how and what Unity is rendering at each moment. For this the engine has an information window called Frame Debugger.

The Frame Debugger lets freeze playback for a running game on a particular frame and view the individual draw calls that are used to render that frame. As well as listing the drawcalls, the debugger also lets step through them one by one to see in great detail how the Scene is constructed from its graphical elements.

The Frame Debugger window shows the drawcall information and lets control the “playback” of the frame under construction.

The main list shows the sequence of drawcalls (and other events like framebuffer clear) in the form of a hierarchy that identifies where they originated from. The panel to the right of the list gives further information about the drawcall such as the geometry details and the shader used for rendering.

At the top of the information panel is a toolbar which lets isolate the red, green, blue and alpha channels for the current state of the Game view. Similarly, it’s possible isolate areas of the view according to brightness levels using the Levels slider to the right of these channel buttons. These are only enabled when rendering into a RenderTexture.

Only enabled when the camera is in Deferred mode.

5.1.3. *The Rendering Path - Deferred Mode*

The Rendering Path es la propiedad de las cámaras de Unity que determina el modo en el que se ejecutan las diferentes propiedades de los elementos de la escena como las luces, sombreado y shaders.

Unity supports different Rendering Paths and the one used by the project is chosen in each *Camera* in the scene:

Forward Rendering

Forward is the traditional rendering path. It supports all the typical Unity graphics features (normal maps, per-pixel lights, shadows etc.). However under default settings, only a small number of the brightest lights are rendered in per-pixel lighting mode. The rest of the lights are calculated at object vertices or per-object. This option does not interest to achieve photorealism.

Legacy Deferred

Legacy Deferred is similar to Deferred Shading, just using a different technique with different trade-offs. But it does not support the Unity 5 PBR shaders (physically based standard). This option cannot achieve photorealism.

Legacy Vertex Lit

Legacy Vertex Lit is the rendering path with the lowest lighting fidelity and no support for realtime shadows. It is a subset of Forward rendering path. This option does not interest to achieve photorealism.

Deferred Shading

Deferred Shading is the best rendering path with the most lighting and shadow fidelity, and is best suited if there are many realtime lights and get a hyperrealistic quality. It requires a certain level of hardware support. But this only works if the camera is in *Perspective projection*, deferred rendering is not supported when using *Orthographic projection*. If the camera's projection mode is set to *Orthographic*, these values are overridden, and the camera will always use **Forward** rendering.

When using deferred shading, there is no limit on the number of lights that can affect a GameObject. All lights are evaluated per-pixel, which means that they all interact correctly with normal maps, etc. Additionally, all lights can have *cookies* (textures that simulate shadows from the origin of light, e.g. clouds) and *shadows*.

Deferred shading has the advantage that the processing overhead of lighting is proportional to the number of pixels the light shines on. This is determined by the size of the light volume in the Scene regardless of how many GameObjects it illuminates. Therefore, performance can be improved by keeping lights small. Deferred shading also has highly consistent and predictable behaviour. The effect of each light is computed per-pixel, so there are no lighting computations that break down on large triangles.

On the downside, deferred shading has no real support for anti-aliasing and can't handle semi-transparent GameObjects (these are rendered using **forward** rendering).

The Deferred mode also allows to execute *SSR* (Space Screen Reflections) and *SSAO* through the tool of Post-processing.

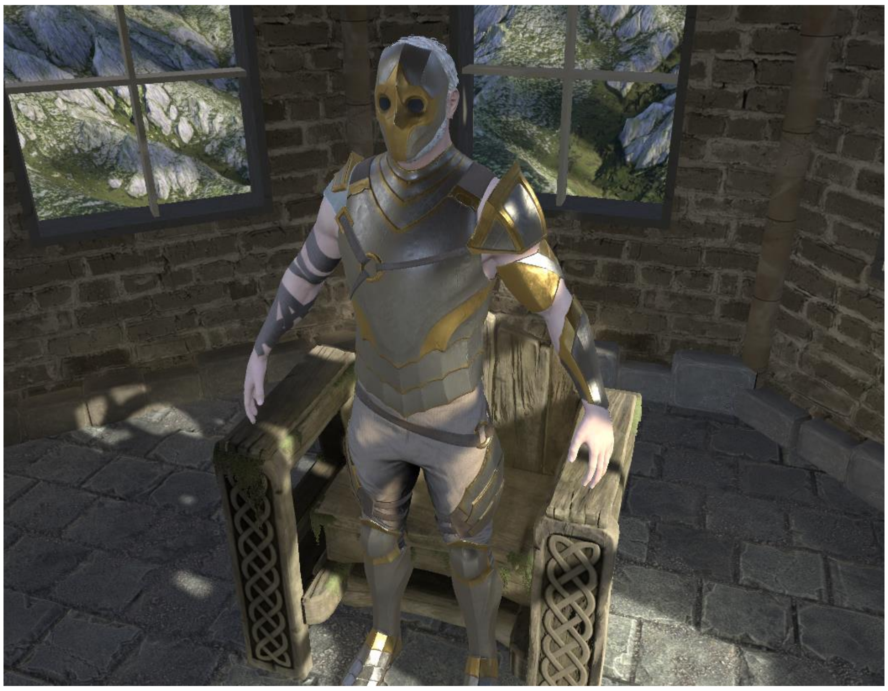


Figure 26: Screenshot with Forward rendering.

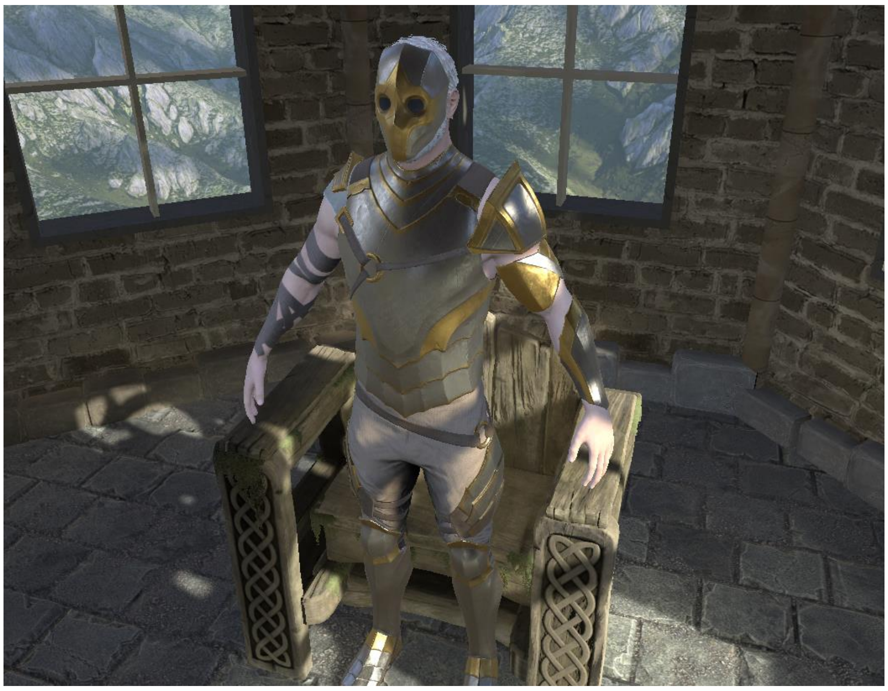


Figure 27: Screenshot with Deferred rendering.

5.1.4. Camera Setup

The camera is going to be the most important element to get hyperrealistic graphics, in it will be added a series of scripts developed for the Adam demo outdoors to improve directional lighting like Atmospheric Scattering, activates HDR option and component of Post-Processing Behavior, developed in beta for version 5.5.0f3 and already set by default in the package for the version of Unity 5.6.

HDR (High Dynamic Range Rendering)

In standard rendering, the red, green and blue values for a pixel are each represented by a fraction in the range [0, 1], where 0 represents zero intensity and 1 represents the maximum intensity for the display device.

More convincing visual effects can be achieved if the rendering is adapted to let the ranges of pixel values more accurately reflect the light levels that would be present in a real scene. Although these values will ultimately need to be mapped back to the range available on the display device, any intermediate calculations (such as Unity's image effects) will give more authentic results. Allowing the internal representation of the graphics to use values outside the [0, 1] range is the essence of High Dynamic Range (HDR) rendering.

When HDR is active in the camera, the scene is rendered into an HDR image buffer which can accommodate pixel values outside the [0, 1] range. This buffer is then postprocessed using image effects such as HDR bloom. The tonemapping image effect is what converts the HDR image into the standard low dynamic range (LDR) image to be sent for display. The conversion to LDR must be applied at some point in the image effect pipeline but it need not be the final step if LDR-only image effects are to be applied afterwards. For convenience, some image effects can automatically convert to LDR after applying an HDR effect.

> **Tonemapping** is the process of mapping HDR values back into the LDR range.
>
> **HDR Bloom and Glow:** using HDR allows for much more control in post processing. LDR bloom has an unfortunate side effect of blurring many areas of a scene even if their pixel intensity is less than 1.0. By using HDR it is possible to only bloom areas where the intensity is greater than one. This leads to a much more desiarable outcome with only super bright elements of a scene bleeding into neighboring pixels.

In the tool of *Postprocessing* appears the option of *Eye Adaptation* that allows the HDR to execute a balanced light transition from one zone with a lower light intensity to another with a higher level to level the darkness that appears on the screen, simulating the effect that exerts the eye to adapt to the conditions of light in certain closed spaces to change to open spaces.

Advantages of HDR:

- Colors not being lost in high intensity areas
- Better bloom and glow support
- Reduction of banding in low frequency lighting areas

But the HDR demands higher performance to the VRAM and is not supported by all hardware.

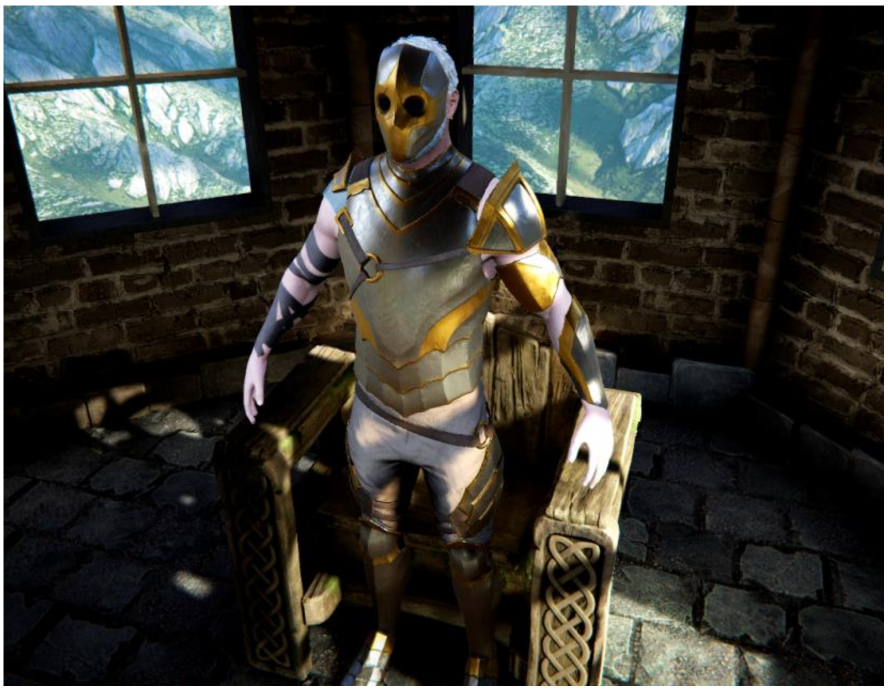


Figure 28: Screenshot without HDR.

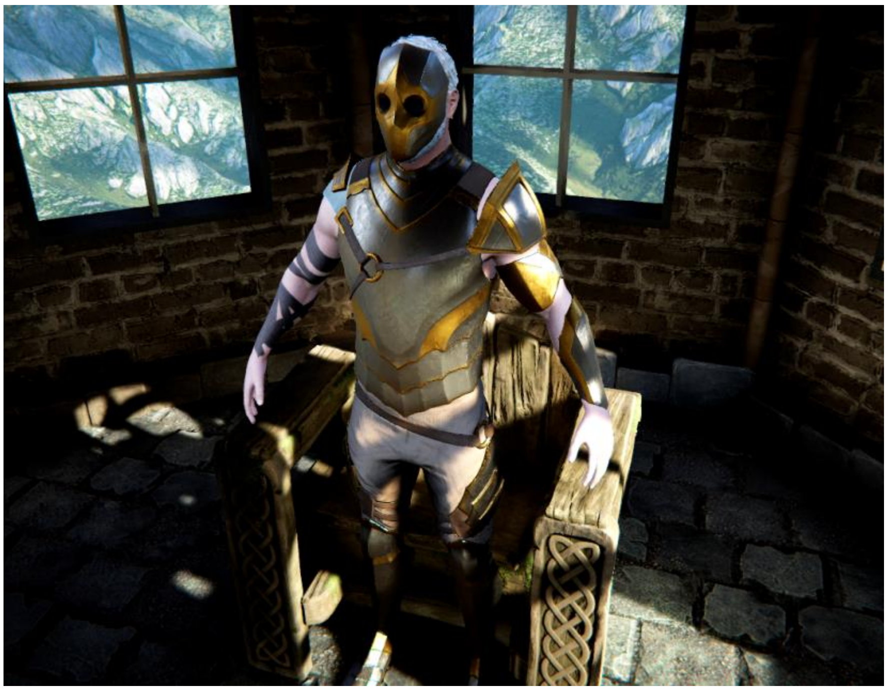


Figure 29: Screenshot with HDR.

5.1.5. FX

The Particle System differs from the rest of objects in scene that are formed by 3d meshes, which are the ideal way to represent “solid” objects with a well-defined figure. FX is used to represent other entities, fluid and intangible in nature and therefore difficult to represent using 3d meshes. Examples are different effects of liquids, smoke, flames or explosions.

For customizing the particles, Unity offers a wide range of options including initial values when emitting particles, such as the initial size or color of particles and other options such as the amount of particles emitted, the way in which they spawn, speed or color in function of the life time.

In the demo the following effects have been worked with the particle system:

- **Fire:** is composed of two types of particles, the first are the particles that simulate the flames and the second the embers that come off from the center of the flame. They are cyclic and are generated in the form of a circle around the emission center to mimic the realistic movement of fire.
- **Smoke:** It consists of two layers one that is emitted in cone to create a vertical row of smoke and another in the form of hemisphere to make the effect of the smoke that is dispersed the farther it is of the center of emission
- **Dust:** dust particles appear in the shape of a box to generate the random motion of this in different directions in a reduced space.

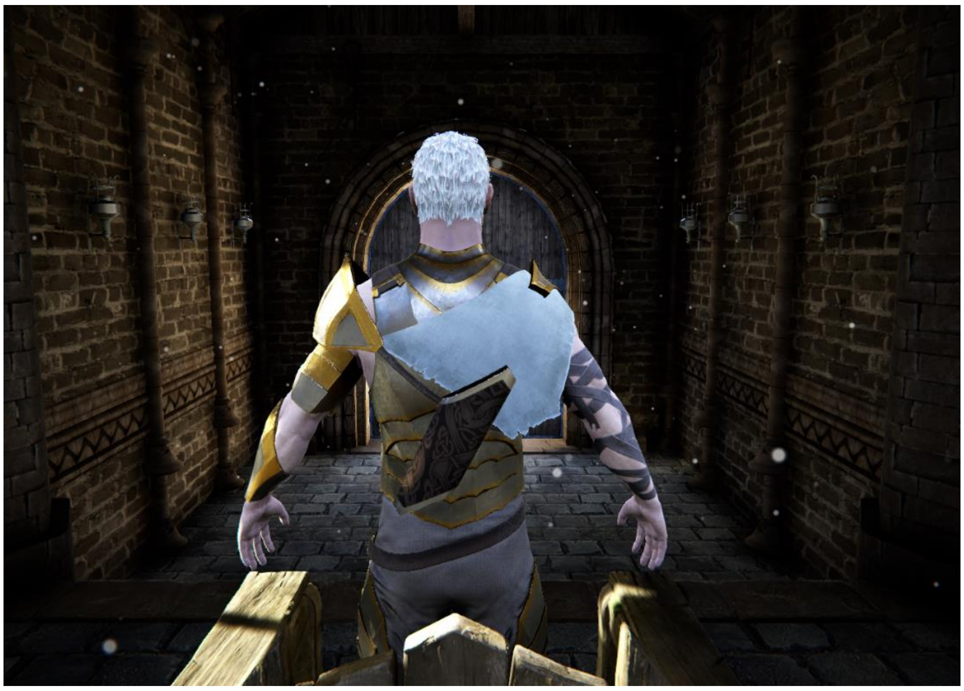


Figure 30: Screenshot with Dust Particles.

5.1.6. Post-Effects

Post-Effects allow to add the final layer, so the demo will look hyperrealistic. Unity allows thanks to the Post-processing package, which is added like a component to the camera that allows to add a series of features that at the beginning do not appear on the screen, such as Ambient Occlusion, SSR, Color Grading or Bloom, between Others.

For the demo the following post-processing features have been used:

- **Antialiasing:** This option allows to clean the image of the saw or aliasing edges that are seen on curved and inclined surfaces.
- **Ambient Occlusion:** allows to generate for the currently focused image a shading between the edges closer together of the different elements of the scene to favor the realism of the image
- **SSR (Screen Space Reflection):** for screen-focused elements, the option to reflect objects around the reflective surface shown in the image is activated.
- **Eye Adaptation:** this option adjusts the level of light intensity depending on whether the camera is in a closed or an open space, simulating how the eye would adapt to receive the same amount of light in both spaces.
- **Bloom:** effect that produces light stripes that extend beyond the edges of bright areas shown in the image.
- **Chromatic Aberration:** effect that simulates the result of the dispersion in which there is a failure of the lens to focus all the colors at the same point of convergence, producing the individual detection of several of them.
- **Vignette:** frames the image by a degree of darkening applied to the edges and corners of the screen.

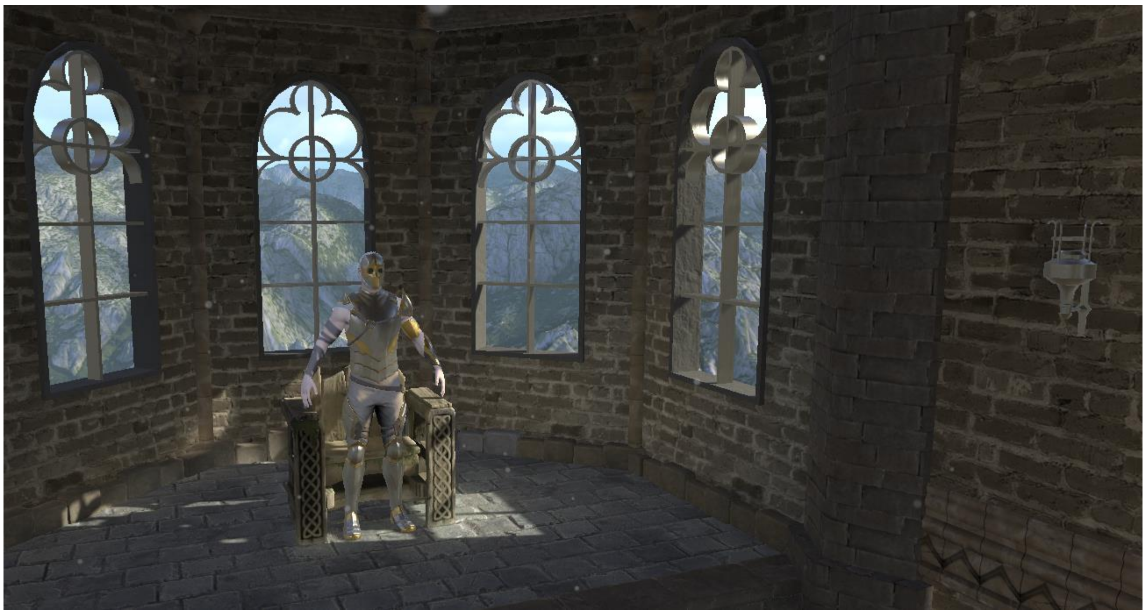


Figure 31: Screenshot without Post-Effects.

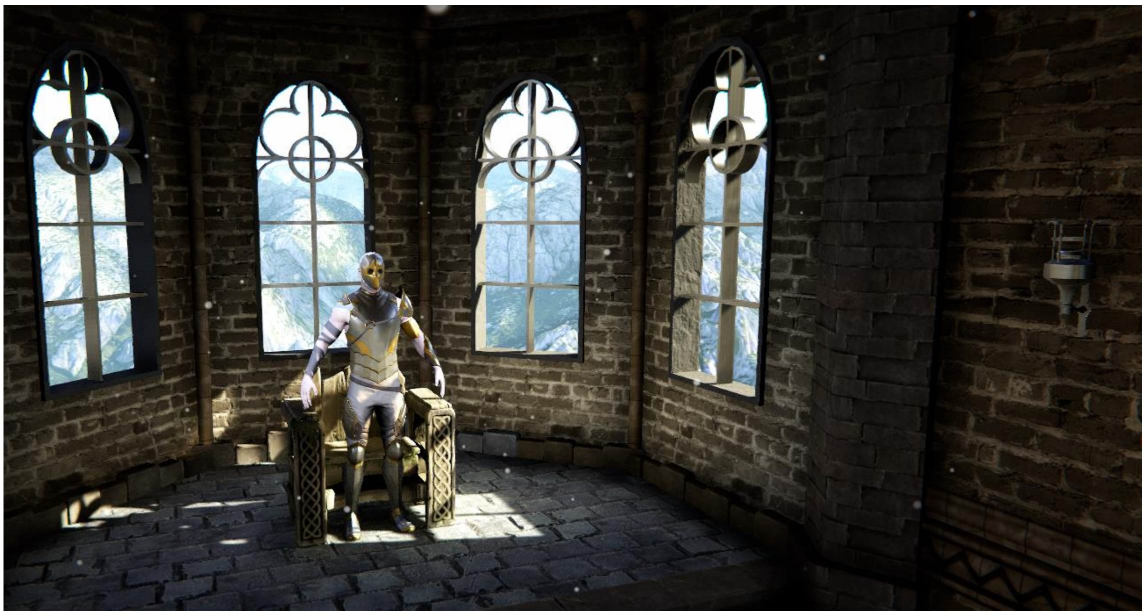


Figure 32: Screenshot with Post-Effects.

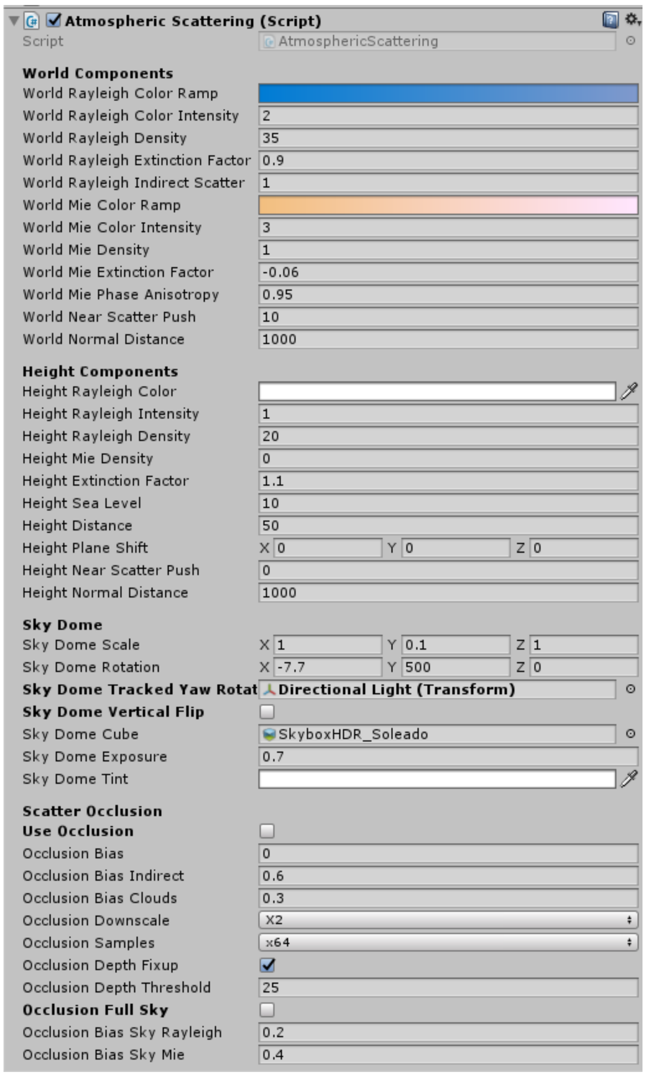
The chosen effects allow to emphasize the light and the flashes of the scene in the surfaces, the color is more intense and the materials seem more realistic. In addition, the ambient occlusion generates shadows that show how the different elements of the scene react between them.

5.1.7. Atmospheric scattering

Atmospheric scattering is an effect that works the scattering of light. It is used mainly to work the atmosphere of the scene obtaining the effect of blurring of the horizon as further away the elements in the scene are seen mixing with the Skybox of the scene.

It achieves a realistic sun and atmosphere simulation by adjusting various parameters from the camera component and the light component.

The component of Atmospheric Scattering looks like *Figure 33*.



Atmospheric Scattering (Script)

Script	AtmosphericScattering
World Components	
World Rayleigh Color Ramp	🟦
World Rayleigh Color Intensity	2
World Rayleigh Density	35
World Rayleigh Extinction Factor	0.9
World Rayleigh Indirect Scatter	1
World Mie Color Ramp	🟧
World Mie Color Intensity	3
World Mie Density	1
World Mie Extinction Factor	-0.06
World Mie Phase Anisotropy	0.95
World Near Scatter Push	10
World Normal Distance	1000
Height Components	
Height Rayleigh Color	⬜
Height Rayleigh Intensity	1
Height Rayleigh Density	20
Height Mie Density	0
Height Extinction Factor	1.1
Height Sea Level	10
Height Distance	50
Height Plane Shift	X 0 Y 0 Z 0
Height Near Scatter Push	0
Height Normal Distance	1000
Sky Dome	
Sky Dome Scale	X 1 Y 0.1 Z 1
Sky Dome Rotation	X -7.7 Y 500 Z 0
Sky Dome Tracked Yaw Rotat	Directional Light (Transform)
Sky Dome Vertical Flip	☐
Sky Dome Cube	SkyboxHDR_Soleado
Sky Dome Exposure	0.7
Sky Dome Tint	⬜
Scatter Occlusion	
Use Occlusion	☐
Occlusion Bias	0
Occlusion Bias Indirect	0.6
Occlusion Bias Clouds	0.3
Occlusion Downscale	X2
Occlusion Samples	x64
Occlusion Depth Fixup	✓
Occlusion Depth Threshold	25
Occlusion Full Sky	☐
Occlusion Bias Sky Rayleigh	0.2
Occlusion Bias Sky Mie	0.4

Figure 33: Window of Atmospheric Scattering script of the camera.

World Components are options to configure the appearance of the sky and create the depth effect that generates the atmosphere, mixing the sky with the most distant elements.

Height Components work the fog that rises from the ground to the atmosphere. E.g. give that feeling of mist in valleys.

Sky Dome is the set of parameters to define the appearance of the sun with its radius, the position given by the directional light and its exposure.

An example of the effect it produces, see *Figure 34*.

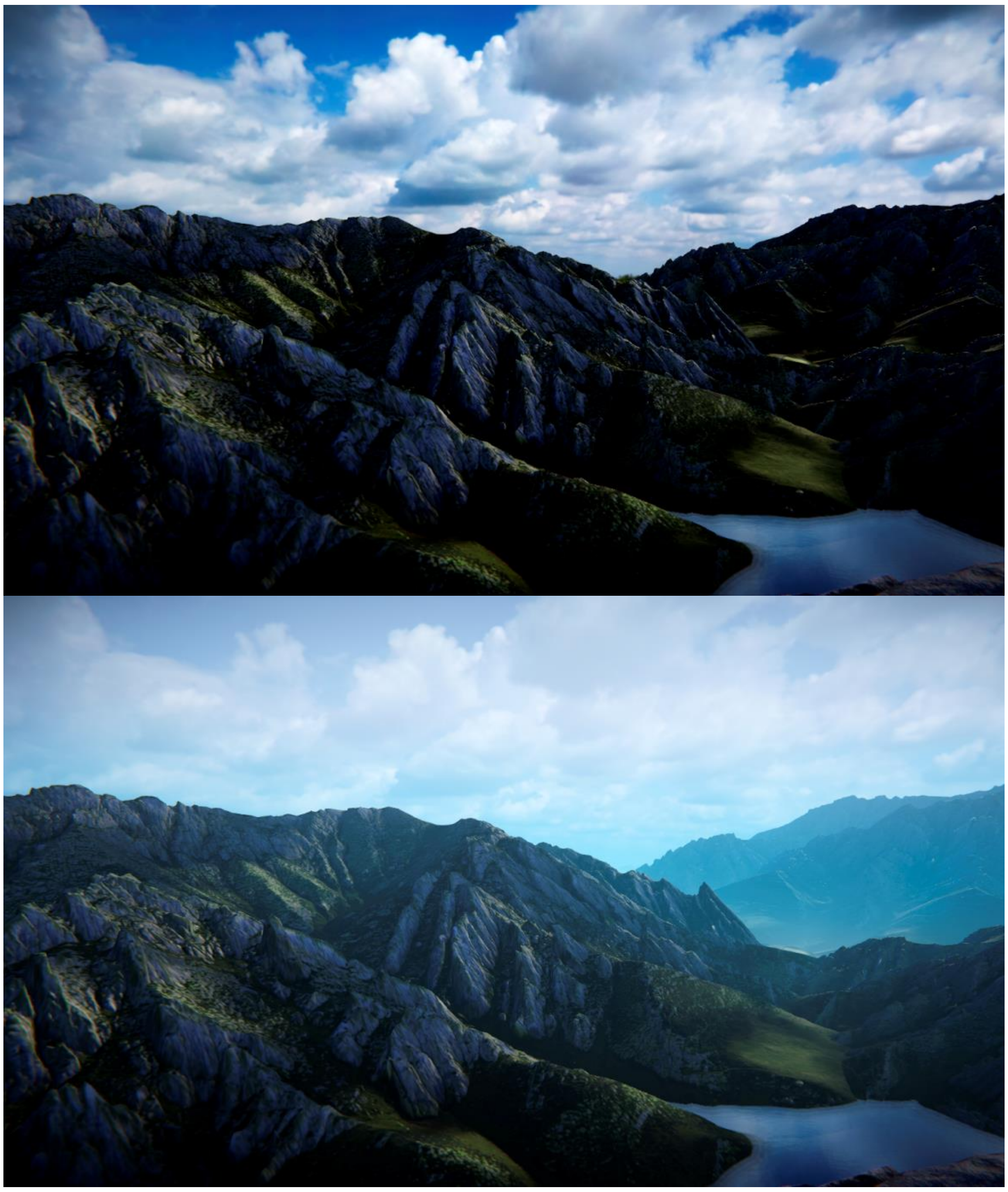


Figure 34: Up image without Atmospheric Scattering and image below with Atmospheric Scattering.

5.2. ILLUMINATION

The demo works lighting from two types of light:

- **Directional Light:** it is the main light of the scene, which illuminates the whole scene and works like a sun thanks to the Post-processing script and the Atmosfering Scattering Sun.
- **Point Lights:** these are the secondary lights distributed by the two rooms that fulfill their function like a torch light, with a reduced intensity.

Baked Lighting

For reducing the loading process of lighting, a bake of scene lighting is performed for objects that are defined as static. The light is activated as Mixed that allows to keep the shadows of non-static objects in real time and for static creates a series of illumination maps that are then activated in the scene to recreate the shadows of these objects.

The first thing to make the illumination bake is to set the scene lights as mixed, then to indicate which objects are static and which are going to be dynamic.

The Reflection Probes elements are used that allow through its central point to transmit the reflex that receives the probe to the other objects that are around him simulating reflexes in real time but which are previously baked, this is means that the elements of around an object are reflected in him, in measure of how reflective the surface is as well as the color that they give off those objects of around, that next to the characteristic of SSR obtains an excellent quite plausible result with the real physics of light.

Subsequently, a problem is presented for dynamic objects. When a dynamic object passes through the shadow of a static object in the scene, it does not affect it over its surface, because it is a texture already calculated and cannot cast shadows on non-static surfaces, the static object does not it shadows in real time on any dynamic object.

Two alternatives can be made for this type of cases:

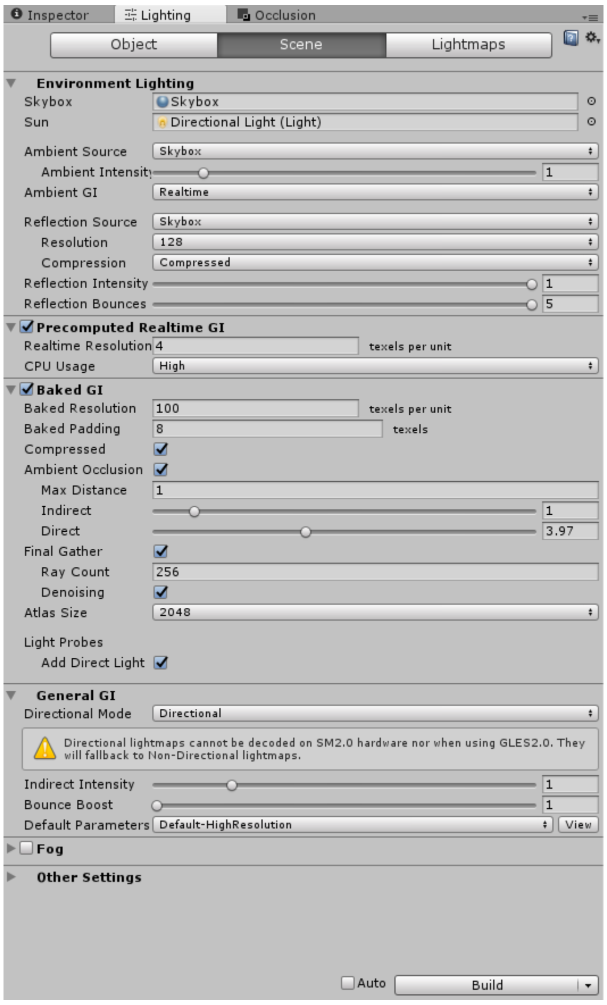
- **Light Probe Group:** groups of points distributed in a way linked in the space of the scene that provides a way to capture and use information about the light that is spread throughout the scene.
  According to the distribution of these points, a more complex mesh is generated. Where there are more points on a given space, it will give more precise information of the intensity of light at that point of the scene.

- **Cast Only Shadows:** If our scene has been set as static, the shadows will no longer appear in real time on a mixed light. But Unity provides the designer with an option with which he pretends to duplicate the scene mesh, with it would have the original mesh already baked and a copy that would stop being static. The new copied mesh located in the same position like the original suffers a change in the Mesh Renderer options of its different objects that compose it, the Cast Shadows feature allows to choose the option Only Shadow that eliminates the original mesh but remains the shadow that projected The mesh on surfaces in real time, this method is more precise than the use of Light Probe Groups, but requires a greater requirement of the hardware.

In this case it is the preferable option to achieve a hyperrealistic result because the Light Probe Groups are more inaccurate approximations for particular illuminations like the light that passes through a window, it isn’t saw a direct light of this window unless the mesh of the Light Probe Group is worked with excessive detail, which in the end is not good compared to the use of Cast Only Shadows.

Finally the models that are to be static, must have the option of Generate Lightmap UVs activated in the original .fbx with the advanced settings of Hard angles to 180 and Angle and Area error to 0. So that the lightmap does not present impurities for each new generated island in the uvs. This option is necessary if the model does not have pre-designed uvs in advance for lightmapping.

Already in lighting window the options to mark are the following to achieve the best quality in the lightmap for the scene. See *Figure 35*.


Inspector | Lighting | Occlusion

Object | Scene | Lightmaps

Environment Lighting
Skybox: Skybox
Sun: Directional Light (Light)
Ambient Source: Skybox
Ambient Intensity: 1
Ambient GI: Realtime
Reflection Source: Skybox
Resolution: 128
Compression: Compressed
Reflection Intensity: 1
Reflection Bounces: 5

Precomputed Realtime GI ✓
Realtime Resolution: 4 texels per unit
CPU Usage: High

Baked GI ✓
Baked Resolution: 100 texels per unit
Baked Padding: 8 texels
Compressed: ✓
Ambient Occlusion: ✓
Max Distance: 1
Indirect: 1
Direct: 3.97
Final Gather: ✓
Ray Count: 256
Denoising: ✓
Atlas Size: 2048
Light Probes
Add Direct Light: ✓

General GI
Directional Mode: Directional
Directional lightmaps cannot be decoded on SM2.0 hardware nor when using GLES2.0. They will fallback to Non-Directional lightmaps.
Indirect Intensity: 1
Bounce Boost: 1
Default Parameters: Default-HighResolution | View

Fog

Other Settings

Auto | Build

Figure 35: Screenshot of Lighting window.

Baked GI works important parameters:

The *Resolution* in texel per unit, which take an x amount of pixels for a space unit, and the *Distance* in pixels of how much space there is between each shape in the Lightmap.

The *Ambient Occlusion* must manage two parameters: *Indirect* to work the intensity of those shadows in a realistic way, or *Direct* to make them larger to the artist's taste without being completely realistic.

The *Atlas Size* indicates the resolution of the baked light textures. 2048 or 4096 are good options to make good baked illumination.

5.3. CHARACTER SETUP - THIRD PERSON CHARACTER

The demo La Torre has a warrior who rises to walk the scene to reach the exit. All this happens through the interaction of the player.

The character moves organically and in a third person view, taking several steps.

The first thing is to introduce the model of the character in Unity, for that the character must already be rigged through Mixamo or any program to animate like Maya and 3Ds Max. Once inside a GameObject is created at the position (0, 0, 0) and named as Player, and the character is also put in the (0, 0, 0) and placed in the hierarchy like the Player's child.

To adjust the new hierarchy of the character, the original inspector components when entering the .fbx in the scene must be deleted and in the Player parent, place the following components:

- **Animator:** component that allows assigning an *AnimationController* object to the character. With this, the animations can be select of the character and create a flow in which these are interspersed according to the action of the player.
- **Rigidbody:** component that controls the position of an object through the physical simulation offered by the Unity engine.
- **Collider:** for this demo a capsule collider has been chosen to use, it only offers us a total collision of the body like a large block because the character will only

have move through the scene. But to increase the level of demand of detection of the body, several colliders would be placed throughout the skeleton of the rig (this skeleton comes enclosed in the same .fbx that is being used for the setup of the character and its animations)

- **Motion Scripts:** *Third Person User Control* and *Third Person Character*. These scripts record the input of the player and update the movement and rotation of the character and determine the corresponding animation for each action of the player.

```
// FixedUpdate es llamado para trabajar con las físicas
private void FixedUpdate()
{
        // Inputs de movimiento y agacharse??
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");

        // Calcular la dirección de movimiento para pasar al script m_Character
        if (m_Cam != null)
        {

                // Calcular la dirección relativa de la cámara para moverse:
                m_CamForward = Vector3.Scale(m_Cam.forward, new Vector3(1, 0,
                1)).normalized;
                m_Move = (v*m_CamForward + h*m_Cam.right).normalized;
        }

        // Multiplicador de la velocidad al andar
        if (!Input.GetKey(KeyCode.LeftShift)) m_Move *= 0.5f;


        // Pasar todos los parámetros al script ThirdPersonCharacter
        if (GameFlow.jugar == true)
        m_Character.Move(m_Move);

}
```

Figure 36: Method to update the player input and call the motion function in *Third Person Character* script.

This method detects when the player presses the WASD keys to trace the direction vector towards which the character moves. To calculate this vector, it is used the relative direction of the camera using its axis Z to normalize it and this redirects the movement of the character depending where the camera looks.

```
public void Move(Vector3 move)
{

        // Convertimos el moveInput relativo al mundo vector en un local-relativo
        // Cantidad de giro y cantidad de avance requerida en la deseada direccion

        if (move.magnitude > 1f) move.Normalize();  // Normaliza el vector move

        move = transform.InverseTransformDirection(move);   // De direccion global a
        direccion local

        CheckGroundStatus();        //Comprueba si pisamos el suelo

        move = Vector3.ProjectOnPlane(move, m_GroundNormal);    // Proyecta move en
        función del plano en el que se encuentre el personaje

        m_TurnAmount = Mathf.Atan2(move.x, move.z);     // Cantidad de giro = grados
        entre el eje x y el vector move (x,z)
        m_ForwardAmount = move.magnitude;               // Cantidad Forward/Delante

        ApplyExtraTurnRotation();       // Ayuda para girar más rápido.


        // El control y la velocidad de manejo es diferente cuando está agachado y en
        el aire/airborne

        if (m_IsGrounded)
        {
                HandleGroundedMovement();   // Manejo/Movimiento si estoy agachado
        }
        else
        {
                HandleAirborneMovement();               // Manejo/Movimiento si estoy
                en el aire
        }

        ScaleCapsuleForCrouching(crouch);   // Escalar la capsula para cuando se
        agache
        PreventStandingInLowHeadroom();     // Comprobar que no se ponga de pie en
        zonas de sólo agacharse

        // Envía un input y otro state parameters al animator
        UpdateAnimator(move);
}
```

Figure 37: Method to do movement of the character in *Third Person Character* script.

This method writes the vector in its local direction with respect to the character, checks if the character is in the ground to be able to start moving, and calls to the function *CheckGroundStatus()*, soon the vector direction projects it on the plane in which is the character.

Later the power is calculated to advance and rotate the character.

Adapt the movement with the walk and run animations is called the method *UpdateAnimator(move).*

```
void UpdateAnimator(Vector3 move)
{
        // Actualiza los parámetros del animator
        m_Animator.SetFloat("Forward", m_ForwardAmount, 0.1f, Time.deltaTime);
        m_Animator.SetFloat("Turn", m_TurnAmount, 0.1f, Time.deltaTime);
        m_Animator.SetBool("Crouch", m_Crouching);
        m_Animator.SetBool("OnGround", m_IsGrounded);

                if (!m_IsGrounded)      // Si no está en el suelo, jump se actualiza
                con la velocidad en Y
                {
                        m_Animator.SetFloat("Jump", m_Rigidbody.velocity.y);
                }

        // ---Calcular qué pierna está detrás, para dejar esa pierna que arrastra en
        la animación del salto---
        // (Este código depende del " del ciclo de la animación específica" en las
        animaciones,
        // Una pierna pasa a la otra en los tiempos de clip normalizados de 0,0 y 0,5)

        float runCycle =
        Mathf.Repeat(m_Animator.GetCurrentAnimatorStateInfo(0).normalizedTime +
        m_RunCycleLegOffset, 1);

        float jumpLeg = (runCycle < k_Half ? 1 : -1) * m_ForwardAmount;     // Pierna
        de salto

        if (m_IsGrounded)   // Si está en el suelo actualiza jumpLeg
        {
                m_Animator.SetFloat("JumpLeg", jumpLeg);
        }

        // El multiplicador de velocidad de anim permite que la velocidad total de
        caminar / correr sea ajustada en el inspector,
        // Que afecta a la velocidad de movimiento debido al movimiento de la raíz.

        if (m_IsGrounded && move.magnitude > 0)
        {
                m_Animator.speed = m_AnimSpeedMultiplier;
        }

        else
        {
                // No se utiliza mientras esté en el aire
                m_Animator.speed = 1;
        }
}
```

Figure 38: Method to update the animations of the character in *Third Person Character* script.

This method updates the values of the parameters of the Animator and allows the different animations to be synchronized according to the values that the player has introduced.

The moving leg used by the character when walking or running, it is calculated taking into account the cycle of animation.

If the character is not on the ground, the speed of the animator will stop until it collides with the ground.

```
void ApplyExtraTurnRotation()       // Rotación del personaje
{
        // Ayuda al personaje a girar más rápido (esto se suma a la rotación de la
        raíz en la animación)

        float turnSpeed = Mathf.Lerp(m_StationaryTurnSpeed, m_MovingTurnSpeed,
        m_ForwardAmount);

        transform.Rotate(0, m_TurnAmount * turnSpeed * Time.deltaTime, 0);
}


public void OnAnimatorMove()
{
        // Implementamos esta función para anular el motion root predeterminado.
        // Esto nos permite modificar la velocidad de posición antes de que se
        aplique.
        if (m_IsGrounded && Time.deltaTime > 0)
        {
                //**************************************
                // Mover hacia delante con motionZ / Move forward with motionZ--------
                ---------

                Vector3 moveForward = transform.forward *
                m_Animator.GetFloat("motionZ") * Time.deltaTime;

                Vector3 v = ((m_Animator.deltaPosition + moveForward) *
                m_MoveSpeedMultiplier) / Time.deltaTime;

                //**************************************

                // Preservamos la parte Y existente de la velocidad actual.

                v.y = m_Rigidbody.velocity.y;
                m_Rigidbody.velocity = v;

        }
}


void CheckGroundStatus()        // Comprobar que estamos en el suelo
{
        RaycastHit hitInfo;     // Elemento detectado por Raycast

        if (Physics.Raycast(transform.position + (Vector3.up * 0.1f), Vector3.down,
        out hitInfo, m_GroundCheckDistance))
        {
                m_GroundNormal = hitInfo.normal;
                m_IsGrounded = true;
                m_Animator.applyRootMotion = true;
        }
        else
        {
                m_IsGrounded = false;
                m_GroundNormal = Vector3.up;
                m_Animator.applyRootMotion = false;
        }
}
```

Figure 39: Methods to rotate the character, to move him independent of the motion root, to verify if the character collides with the ground, in *Third Person Character* script.

ApplyExtraTurnRotation() allows to rotate the character using the rotation that the player wants when he presses A or D (horizontal input) normalized with W or S (vertical input). *OnAnimatorMove()*move the character forward using the state parameter animator *motionZ*, a variable assigned to certain animations which represents the speed for each animation. *CheckGroundStatus()* checks that the character is colliding and has a ground near a given distance in the Inspector window.

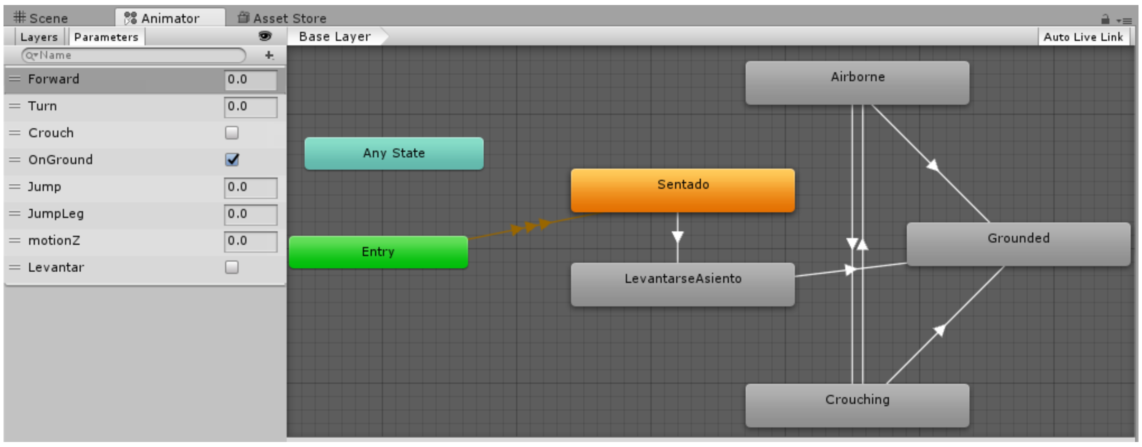
Set the camera in third person, Unity has a package where there are various camera types to import. The one used like basic to then work on it and improve it is the MutipurposeCameraRig. This camera gives the basics such as collision detection to avoid clipping with the walls of the stage and also allows the tracking of the character without needing to be in the same hierarchy Player like the character mesh. This independence allows effects such like moving away from the camera according to the speed of the character or the adjustment of the camera behind the character.

Improve the experience, a script was added to the Pivot object that allowed the camera to rotate around the character freely on the horizontal axis and maintaining a limit of x degrees on the vertical. The automatic rotation is disabled by entering the value of 0 in the SpeedTurn variable in the inspector.

5.4. ANIMATION CONTROL

Animate the character, the object *AnimationController* is used to display on it all the animations imported to Unity of the character.

Previously the animations of the character must have been processed properly in the inspector. In this case, the animations came in individual .fbx, then to make them work in the .fbx of the character, character's avatar is used to transfer the animation.



Forward	0.0
Turn	0.0
Crouch	☐
OnGround	✓
Jump	0.0
JumpLeg	0.0
motionZ	0.0
Levantar	☐

Figure 40: The Animator Window, Base Layer and on the left animation/state parameters.

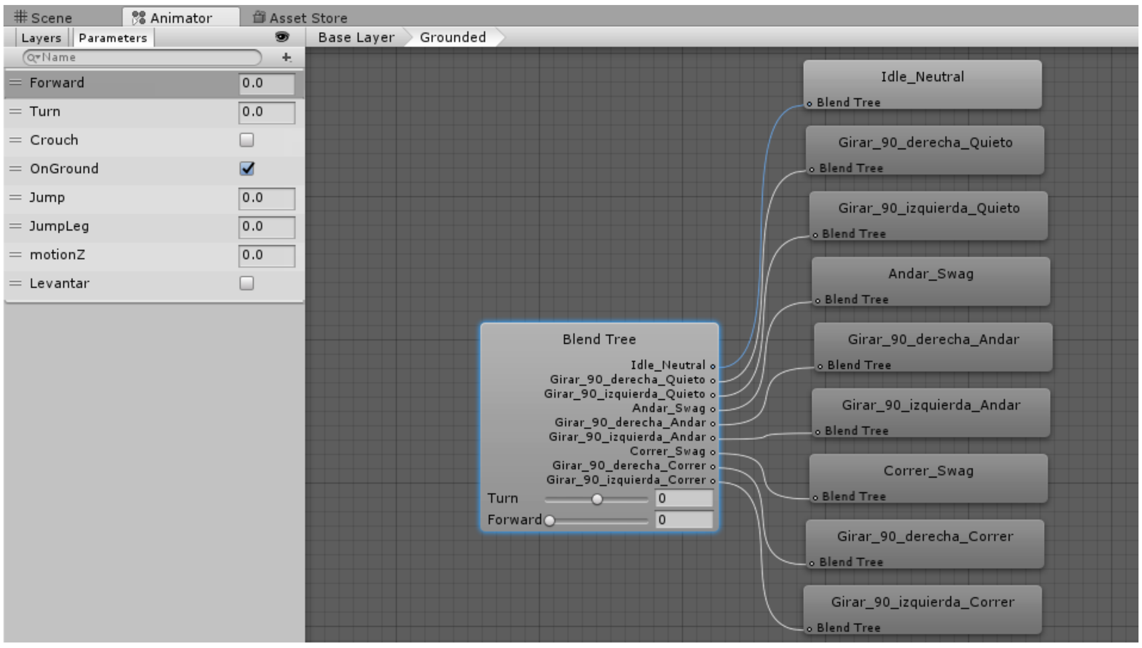
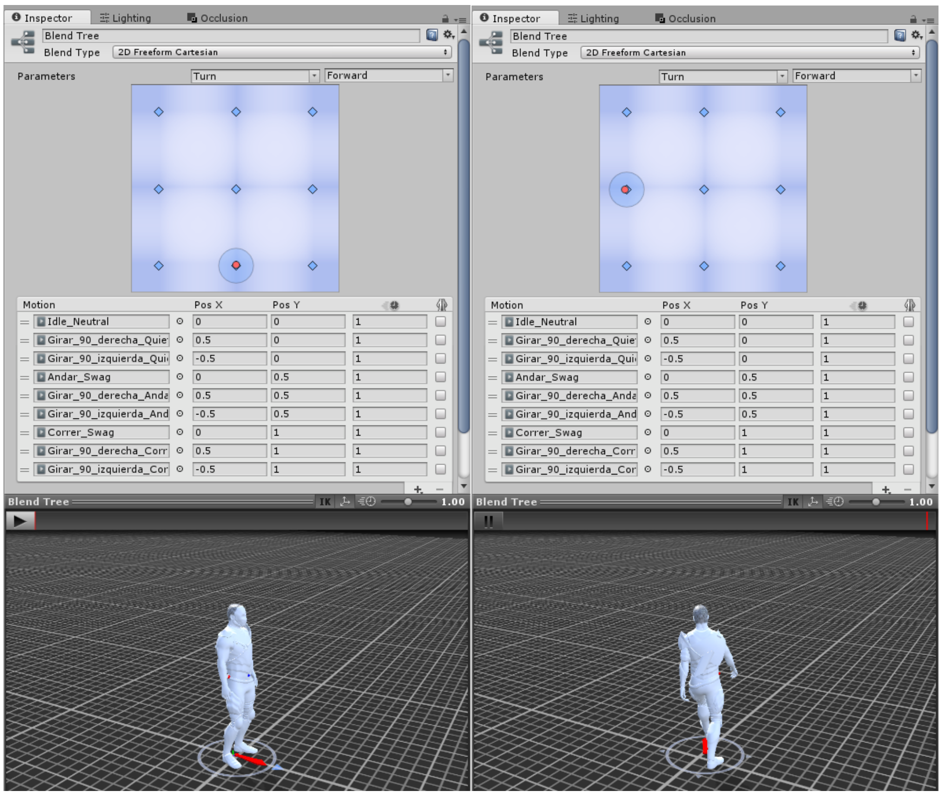


Figure 41: The Animator Window, a BlendTree.

The controller has references to the animation clips used within it, and manages the various animation states and the transitions between them using a so-called State Machine, which could be thought of as a kind of flow-chart or a simple script.

It is important to distinguish between Transitions and Blend Trees. Although both used to create soft animations, these are used for different types of situations.

- **Transitions** are used to make a smooth transition from one Animation State to another for a given period of time. A transition from a completely different movement is fine and the transition is fast and smooth.
- **Blend Trees** are used to allow multiple animations and blends smoothly by incorporating all parts of different grades. The amount of each movement is controlled by *blending parameters*, animation/state parameters with an *AnimatorController*. In order for the mixed motion makes sense, these are mixed must be of a similar in movement and time. Blend Trees are a special type of state in an Animation State Machine.



Blend Tree

Blend Type: 2D Freeform Cartesian

Parameters: Turn, Forward

Motion	Pos X	Pos Y	
Idle_Neutral	0	0	1
Girar_90_derecha_Quie	0.5	0	1
Girar_90_izquierda_Qui	-0.5	0	1
Andar_Swag	0	0.5	1
Girar_90_derecha_Anda	0.5	0.5	1
Girar_90_izquierda_And	-0.5	0.5	1
Correr_Swag	0	1	1
Girar_90_derecha_Corr	0.5	1	1
Girar_90_izquierda_Cor	-0.5	1	1

Figure 42: The Blend Tree configuration in the Inspector Window.

A special detail, walk animations of character have assigned a value of 2 when the animation is executed that 2 is assigned to the *motionZ* and indicates a speed of 2 in the *Third Person Character* script. Then run animations have a 5 to increase the speed with respect to the walk animation and it seems that he really runs.

5.5. SETTING THE STAGE

The scene setting is based on the placement of the different elements of the scene scattered around the three-dimensional space of Unity.

If the design of the scenario has been taken into account since the beginning it has a basis to respect to locate each element.

The stage consists of an interior zone formed mainly by two rooms, a corridor and a balcony, and an exterior formed by mountains, rocks, sea and elements like the light and the camera. The design of each object has been defined in three dimensions from the program of 3Ds Max and now in Unity each element must be made at all. That is, it is time to add the final details for each object like the material and its corresponding transformation (its position, rotation and scale in the scene).

Unity provides a simple editor that allows to place elements in the scene by dragging them from the project folder window. Then through the Unity Inspector window, the location of each element is reset.

In order for the object to show its final appearance, a material is created for each type of object or surface in the scene. Thanks to the textures created previously it is time to use them for this purpose, each material has to have in general except for particular cases, four textures, the albedo to define the color and the type of object that is, the specular to indicate how much the surface is reflective, the normal map to simulate the volume or roughness of the surface and the occlusion to add a realistic shading to the material in the areas closest to the object.

In order to understand how materials work in Unity it is necessary to take into account if in the albedo the assigned color is totally black and on the contrary the color of the specular has enough intensity in its range of illumination, that is, it is a quite clear color near the white, the surface is metallic; and if on the contrary the surface is of a different color to black, its specular is not so bright defines a nonmetallic surface. The most notorious effect between both types is the type of reflection that occurs and the dispersion of light on the surface, the latter is not a constant from the tone of the specular, but it is the alpha of this texture that works the Intensity of brightness. The alpha is different for a metal than for a non-metal but a particularity that shares this layer is that a black and white image, this is so to indicate that the higher the target the brighter the surface is and the blacker the less it shines the surface.

For particular cases like walls or floors is usually added a texture plus the height map, Unity material has this option assignable in a box to generate depth effects on the surface of the object. This texture indicates through a range of gray how the surface sinks by the darker the tone in this texture, is generated in xNormal as the normal map and together with it defines the shape of the surface of the material that will have the object corresponding.

Then textures like the occlusion are not used in materials like the skin, because the effect it causes gives the sensation that the skin loses the tone of *translucency* that it should possess. The occlusion map is still used but in the albedo texture multiplied between its layers and as a detail to remember instead of being black and white this time the texture is in a range of red to simulate the most translucent areas of the body. This is done because Unity does not provide a square for *translucency map* PBR in its materials, which would generate that effect inside the engine.

To assign the materials, it’s necessary to do the same like placing the objects in the scene, each one is dragged to its corresponding object and automatically Unity fits in them and they get the textures giving the final appearance the material.

5.5.1. Assets of official demos

The exterior and the particles were design like assets of the official Unity demos, e.g. the exterior scenario of the Adam. The elements that were used by way of test were the denominated Cliffs, these are several rocks that are used to give detail to the flat lands like in the scene of the Adam and a mountain to decorate the horizon.

The new assets are designed with the same structure like those of Adam to define the hills of the outer scenario of the demo of La Torre. First the model of the mountain that was conformed by three LODs (levels of detail) was replicated. LODs are different models that represent the same shape but with different amount of polygons to allow a lower load of polygons when the object is some distance from the camera and reduce the weight of the graphic load.

To do the first level, a plane with enough subdivisions is needed to apply a displacement map; this texture is a height map done by hand or with Zbrush, sculpting in the same plane. Later, the ProOptimizer modifier is assigned to the mesh in 3Ds Max, this modifier reduces the amount of polygons preserving the shape of the grid that has the base model through a variable that goes from 100% to 0% indicating the number of polygons with respect to the original. The original model is saved to later obtain the normal map for each level of detail.

Then the remaining levels of details are obtained by applying the same method so that the previous level of detail is used like the base of the next one. The texture is painted in Photoshop or by adjusting the texture so that the greenest areas are seen on the mountain peaks and the rocky surface on the hillside and the gulf.

With the rest of rocks the same process is carried out, except the rock on which is the model of the Tower, which was made from a bucket and was sculpted in Zbrush.

The textures and models are taken to Unity and the LODs are assembled with the Unity component to adjust to what distance each level of detail should be activated and also indicate which effects can disappear such as removing the collider at the lowest level or adjusting what textures appear in the material for each type of LOD.

Afterwards, each element is scaled and placed in the scene in a different way to cover the horizon in 360º from the point of view of the Tower model, which is located in the center of the scene.

5.5.2. Colliders

Collider components define the shape of an object for physical collisions. A collider, which is invisible, needs to be not in exactly the same shape, like the mesh of the object and in fact, an approximation is usually more efficient and indistinguishable in the game.

The simplest colliders, and least expensive for the processor, are the so-called primitive collider. In 3D, these are Box Collider, Sphere Collider and Capsule Collider.

To assemble the game mode it is necessary that the character respond to the physical correctly for it is necessary to use the colliders. Each collider represents a barrier against which the character collides and stops to prevent clipping between him and the stage. The collider is also used so that the character can interact with the doors of the scenes and thanks to that they have as component a collider and a rigidbody can be opened to the passage of the personage using the physical ones.

When configuring the colliders, it is necessary to remember that the colliders when interacting, their surfaces need to simulate the properties of the material to which they supposedly represent. For example, a wooden board will be slippery than a block of stone, because it will offer more friction. Although the shape of the colliders is not deformed during collisions, their friction and rebound can be configured using Physics Materials, which generate the fiscal qualities to define the frictions of each type of surface. Unity offers a pack of several options to choose from, among which is the wood type and both ends of zero friction and maximum friction.

Triggers

Using the *OnCollisionEnter* function in a script can be detected when there is a collision between objects and execute an action accordingly. However, Unity can also use the physics engine simply to detect when a collider enters another's space without creating a collision. A collider configured like a Trigger (using the Is Trigger property) does not behave like a solid object and will simply allow other colliders to pass through it. When a collider enters its space, a trigger will call the *OnTriggerEnter* function in the object trigger scripts to perform some action for the result of the detection of an object.

This property is very useful for performing cutscene scripts, performing actions when the character reaches a certain point in the scene. The trigger is located at the point where the character traverses and the script detects him and can execute an animation or effect pre-calculated accordingly to interlace actions.

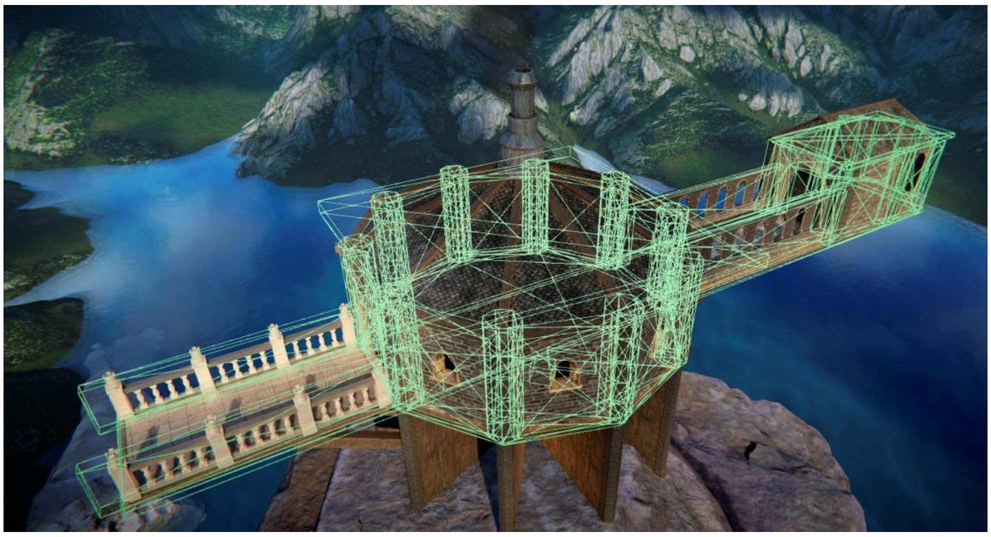


Figure 43: Colliders in the stage.

5.5.3. Game flow

The demo of La Torre has a gameplay that begins and ends interlacing cutscenes and gameplay.

This gameplay gives the player an interesting gameplay so that he gets involved in moving forward and finishing the demo. The difference here is that the dynamics of the game to be performed will be implicit within the demo, i.e., by exploring the player will advance certain guidelines to finish the demo.

The guidelines that the player must do are: start the demo, at that time a small cutscene is managed that puts us controlling the character, move from the starting room to the final where a second cutscene will appear in which the character with the sword, which is in the middle of the room, then the character must open the way to the exit and there jumps the final cutscene, in which the exterior landscape of the demo is observed. The demo when the demo title and the author's title appear, fade out and the final screen appears with the buttons to restart the demo or exit it.

5.5.4. Cutscenes

The cutscene sequences are phases in the demo in which the control of the character is retired to the player to obtain camera angles and different planes for the basic camera of the game mode. With these cutscenes are emphasized the beginning and end of the demo, and also show the acquisition of the sword of the big room in a more interesting way than with an animation using the basic plane behind the character of the ingame camera.

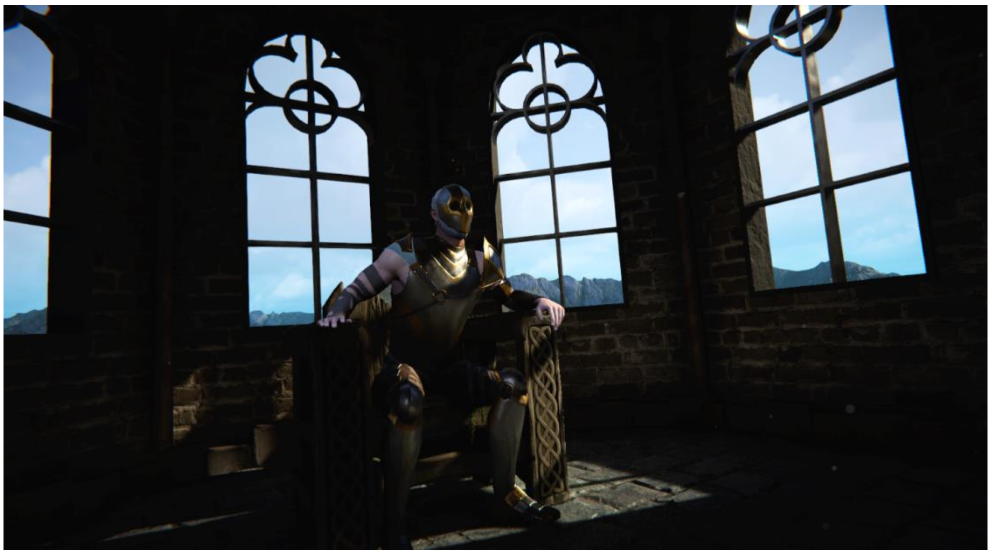


Figure 44: Screenshot of the first cutscene.

5.6. SWITCH CAMERAS

In Unity the change of plane is not through different cameras but by the repositioning of the third person camera in different points of the scene through the activation of triggers. At the beginning of the demo the camera is located from a different angle and position to the basic position behind the character, but at the beginning of the demo after a few seconds the character stands and the camera changes position to be behind the character. At that moment the player is given control of both the movement of the character and that of the camera to rotate it around him.

At the moment the character enters the second room is triggered by the second cutscene. The camera resets in the scene in pre-defined points to focus the sword and the character entering and reach the sword. The rate of change of plane is managed the same like before, by means of several counters the duration of each plane is controlled and the appropriate animations for each moment. When the sequence ends the character is again seen in the third person and controlled by the player.

The last sequence also appears through a trigger near the last door when trying to open it. At this point the control is already completely taken away from the player and the character automatically moves to a waypoint on the balcony of the Tower while the camera change is still measured through timers that cause the camera to be relocated to continue focusing the last plane which realizes a travelling by zooming out to show a general map of the whole scene.

5.7. CONSECUTIVE ANIMATIONS

During the cutscenes the character acts without the will of the player, so that the character's animations are predefined so that they have a certain duration and happen at a point x of the stage and happen naturally without forcing a sudden transition. That is why counts are used just like in the camera change to make a sequence measured in seconds.

5.8. LOADING TIME

The cutscenes happen in an organic way, that is, each sequence happens with the gameplay part without loading times. Once the cutscene ends the control and camera are instantly resisted like a change of plane more but the player already understands how the moment of interacting in the demo. This is achieved thanks to all animations and camera changes are guided by script and are not pre-rendered videos that need a preload to be able to be seen between each ingame sequence of the demo.

5.9. CANVAS

To finish it is necessary to define the user interface so that the player can interact off the demo the field.

There are three screens accessible to the player:

- **The main screen:** The initial scene in which the character seated with a message at the bottom tells that it’s possible to start the demo by pressing any button. When the button is pressed the message disappears and the demo starts without charging time.
- **The pause menu:** this screen is accessible during the game mode and gives the options to the player to continue playing, restart the demo or exit to the desktop.
- **The final menu:** this screen appears after the fade to final black giving the options to restart and exit the demo.

Ingame elements:

- **Tag / marker:** these are indicators that warn the player if he delays in moving forward in the demo of where he should go.
- **Credit titles:** The name of the demo and the author appear in fade in at the end of the last cutscene.
- **Fade in and fade out:** When the demo is started and finished there is a fade to black that softens the beginning and end of the demo to be more pleasing to the human eye.

5.10. OPTIMIZATION

To finish the demo should work at a fairly stable rate of frames per second higher than 30 fps and with a resolution of 1080p.

The demo has been developed in a PC with the following characteristics:

- 3.60GHz i7-4790 processor
- Nvidia 970 GTX Graphics Card
- 16GB RAM Memory
- Windows 7 64-bit operating system

The final state of the demo is a frame rate ranging from 50 fps to 60 fps at a resolution of 1080p.

To achieve a good performance, it is necessary to have defined the level of polygons used by each object in the scene to the minimum necessary, preserving the shape of the element it represents.

Then Unity has the following tools to optimize:

- **LODs:** apply the level of detail needed in the objects to compensate for the graphic load depending on the need within the ingame camera frame.
- **Baked illumination:** reduces the load on the illumination of the Unity scene allowing lighting pre-calculation by using lightmaps that store the light value of each element in the scene.
- **Occlusion culling:** this option prevents the graphic loading of certain elements of the scene depending on whether they enter the camera plane or are covered by other objects of larger size.

CHAPTER 3: ART

In this part 3d models will be shown in the final state with his materials and textures and with especial details in their images or their final appearance:

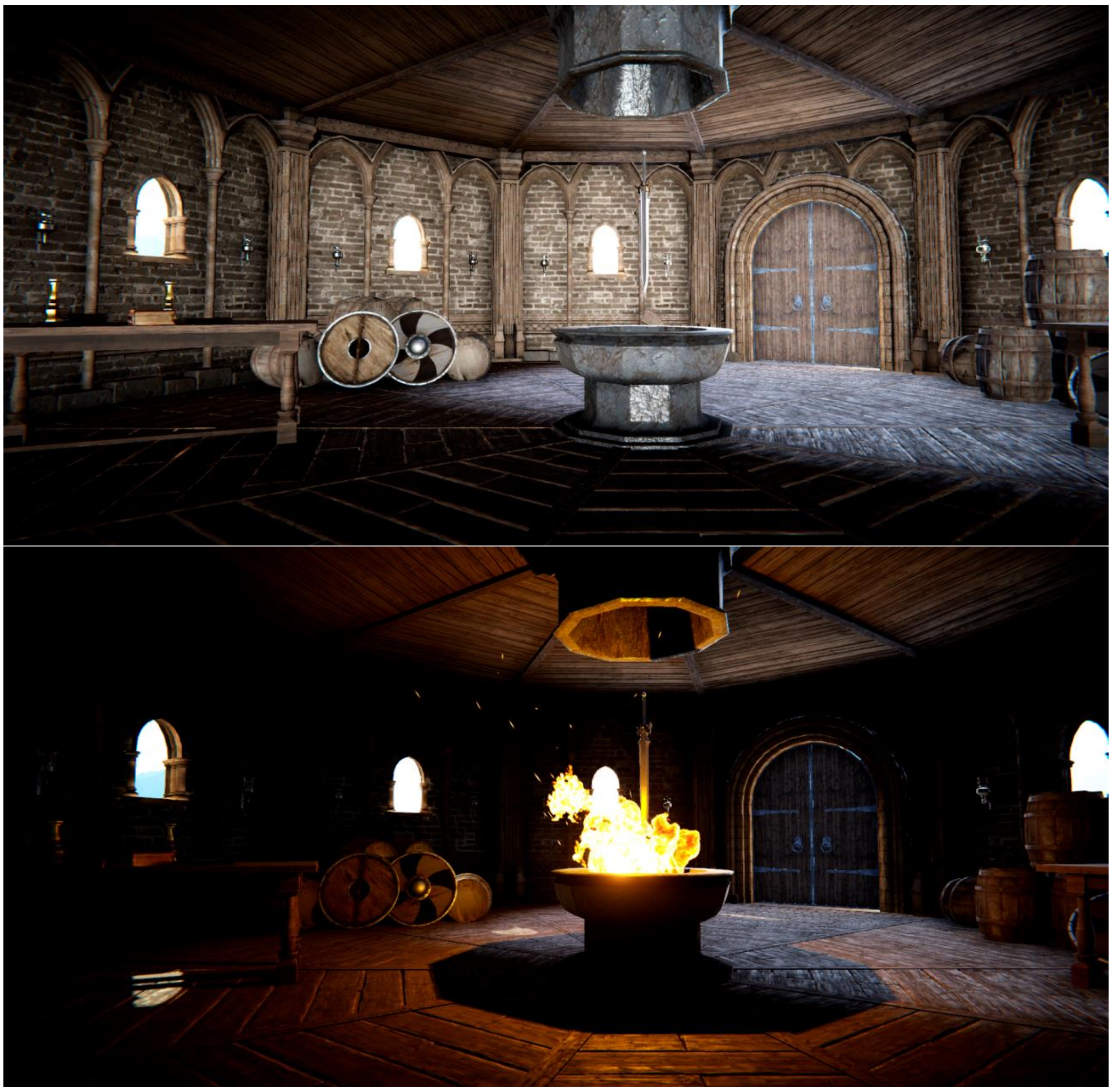


Figure 45: Top image large room interior without direct light and image below with light and effects.

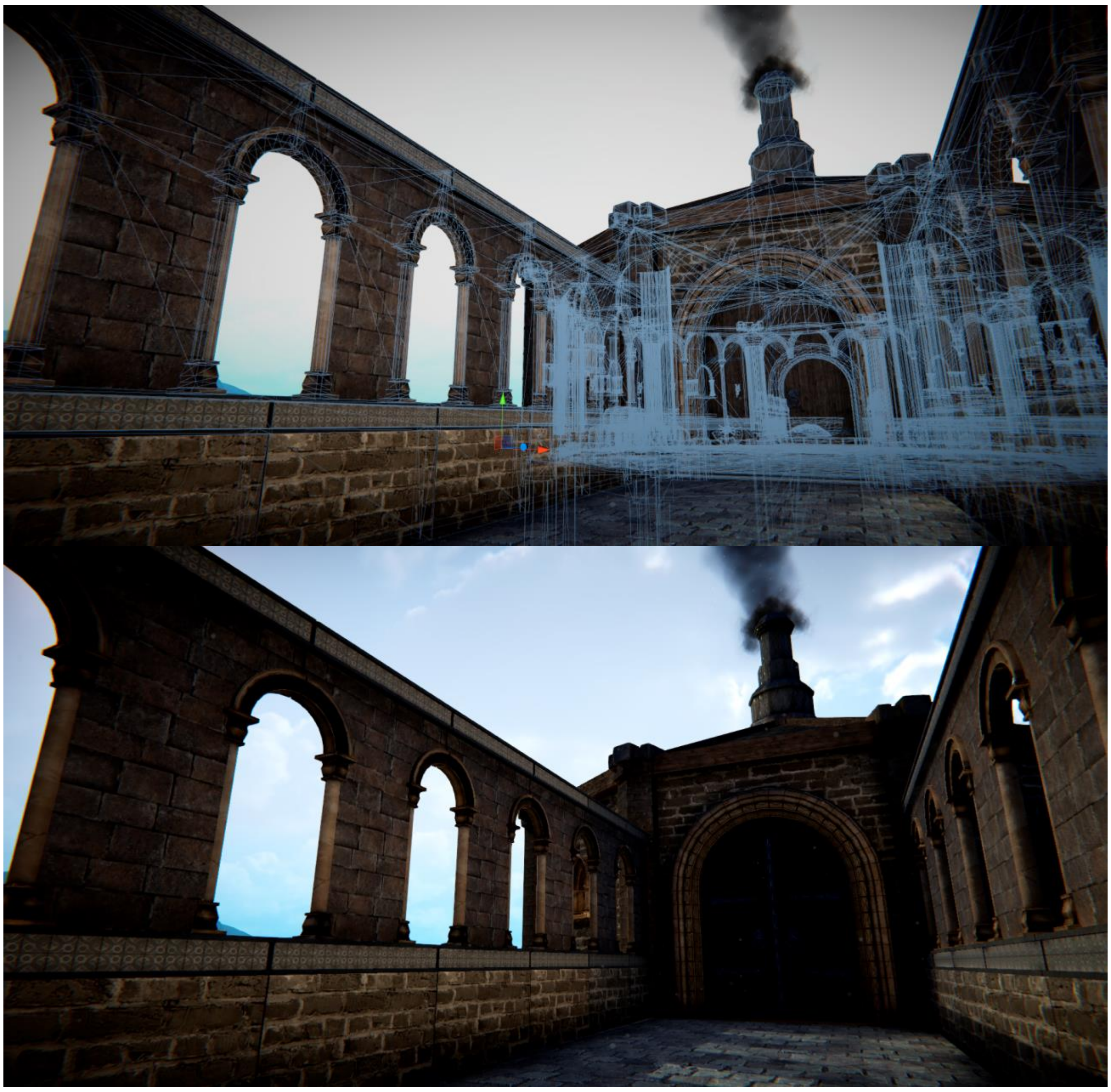


Figure 46: Corridor between the two rooms.

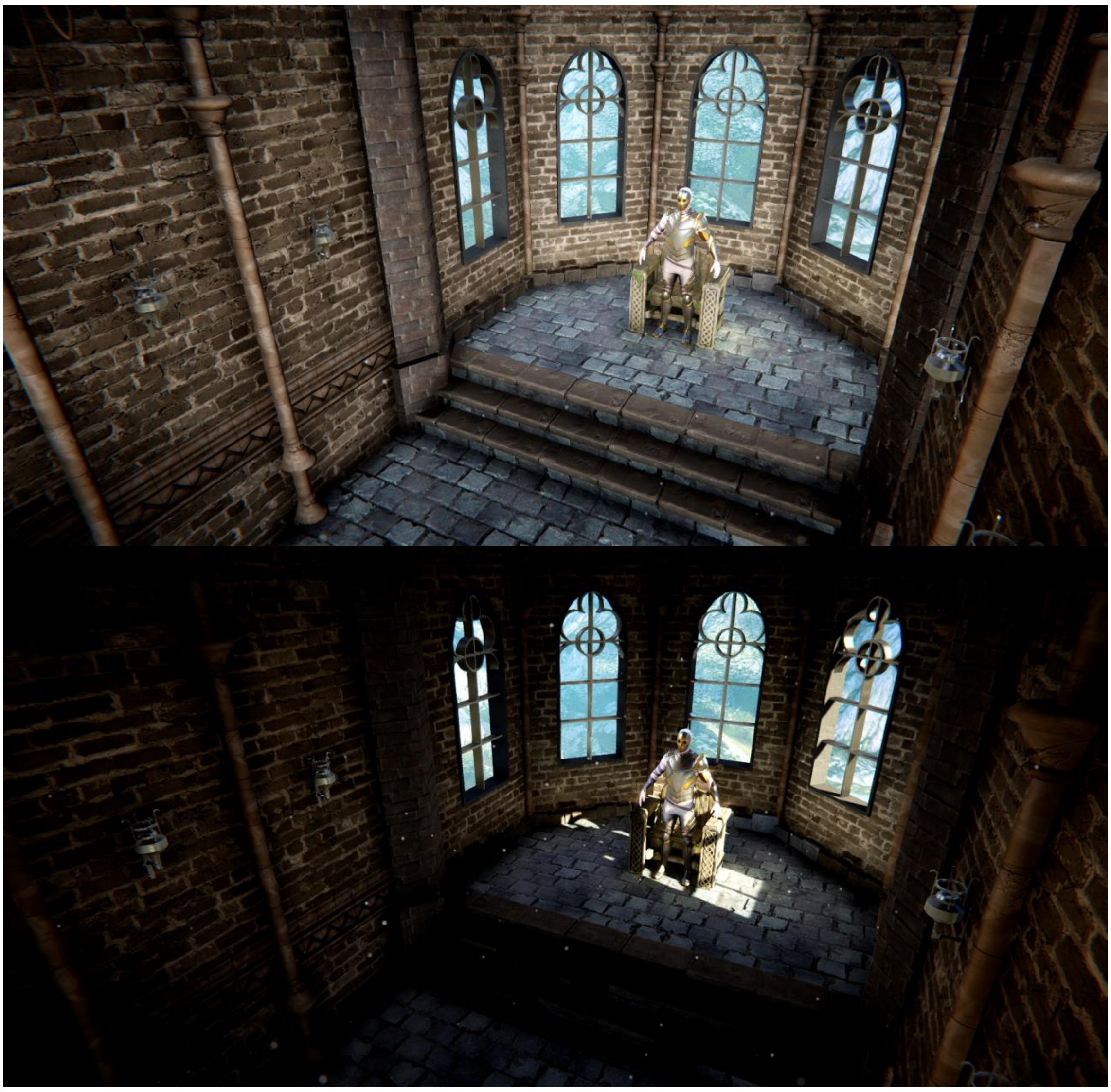


Figure 47: Small room, Top image without direct lights and image below with it.

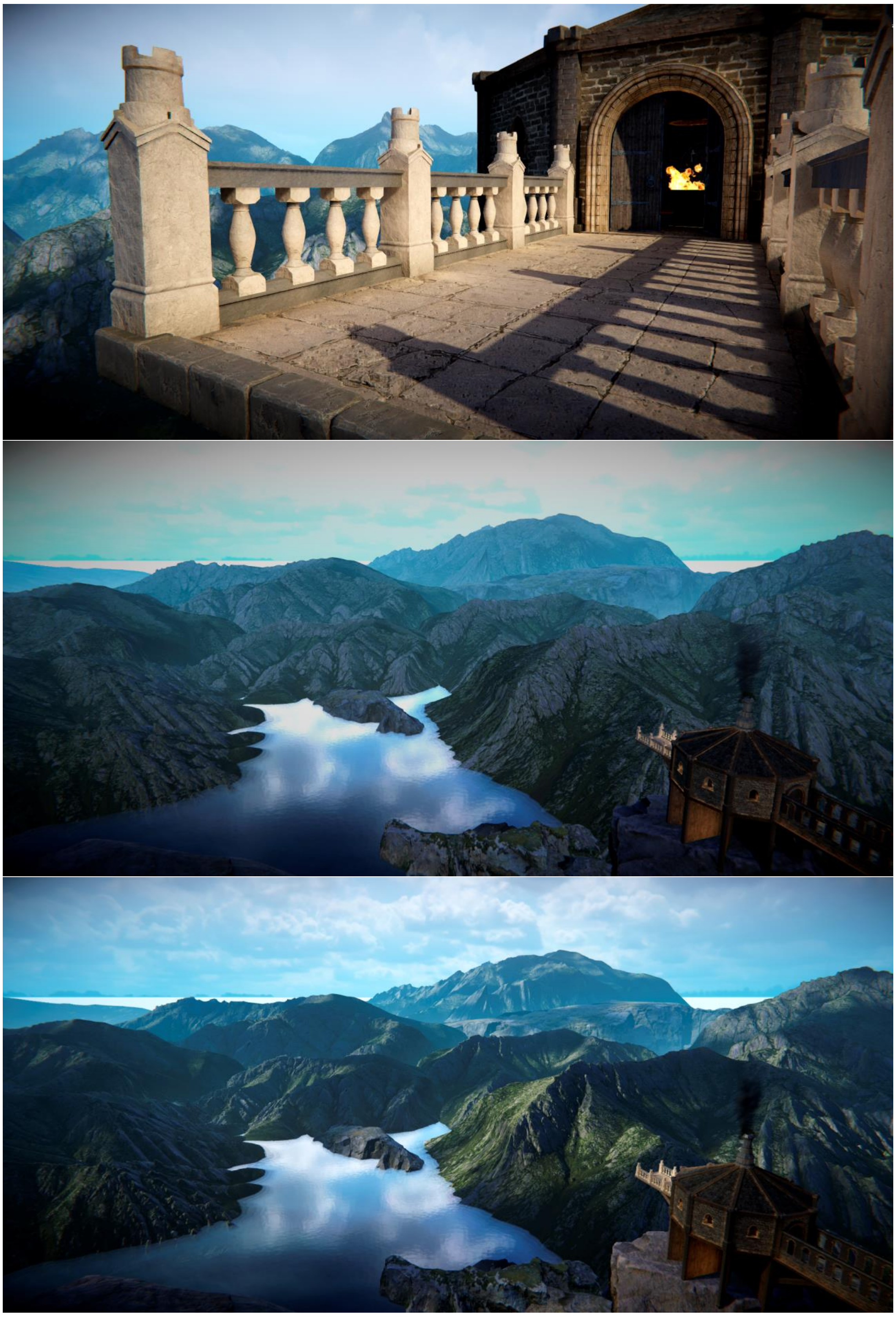


Figure 48: General landscape and tower balcony.

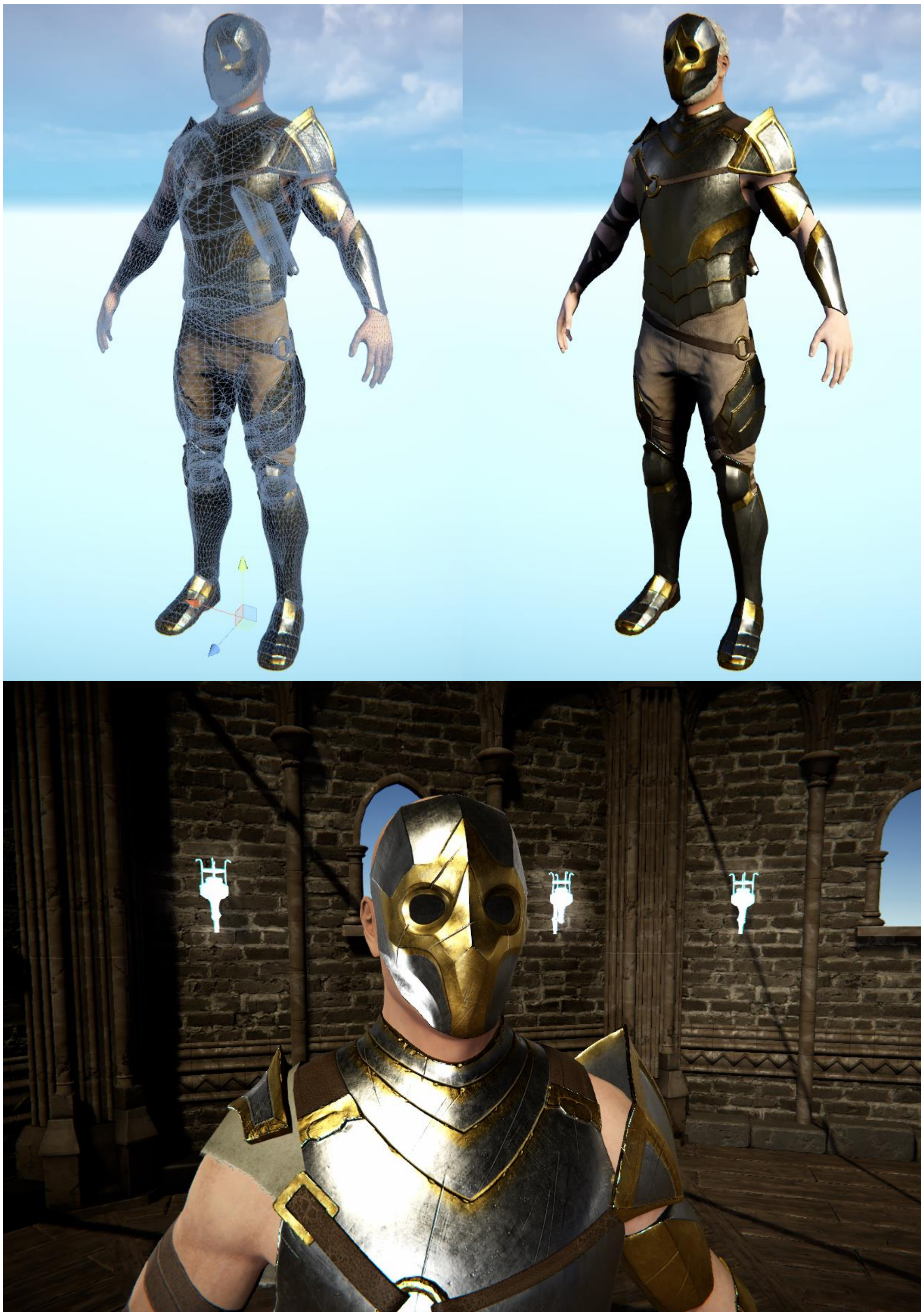


Figure 49: Baltor, playable character.

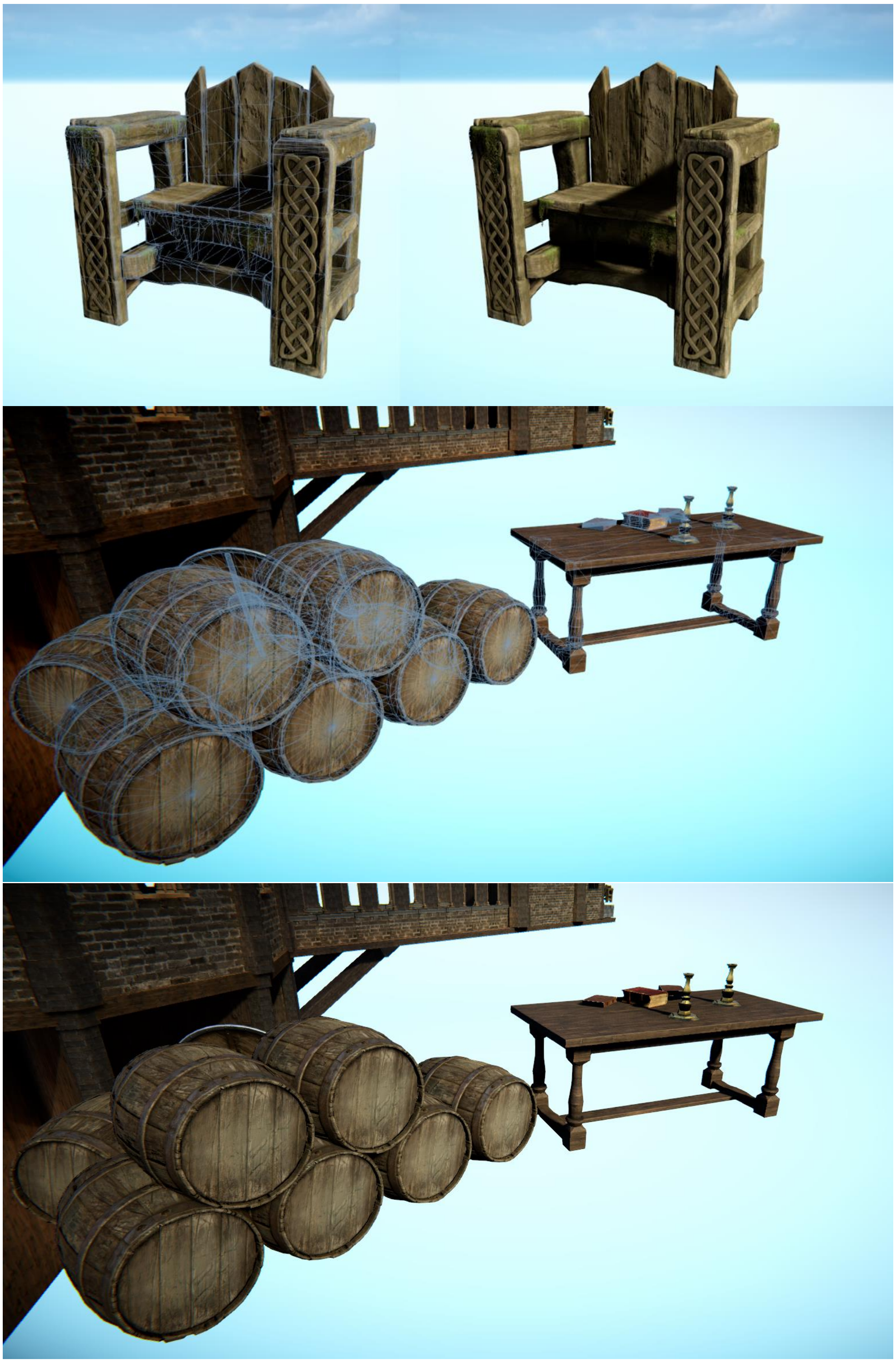


Figure 50: Assets distributed by the demo: barrels, tables, candelabras, books and the seats.

CHAPTER 4: RESULTS

The purpose of the demo was to achieve a high-quality visual and playable level, which expressed the Unity engine maintaining a good performance throughout the playable experience, with a high-quality in scene and character modeling, and animations.

The final result has achieved to respect the basis of the technical proposal.

In addition, it presents more of a cutscene, one in the beginning, with a high-graphic quality like in the rest of the demo and without load times.

During the demo, it has been able to develop a soft control and that responds well of the character of the demo from a perspective in third person so that it is observed and interacted with the stage.

The estimated duration has ended up being about five minutes variable depending on the speed with which we cross the stage, where we can observe the high-quality of the visual section of the demo in both modeling and animations and effects.

In the development process to get the final visual section, the demo has gone through several test tests to see how this reacted Unity to various graphic options.

The post-effects tests varied the result of color and contrast in function of several options and parameters until obtaining the final tone more in line with the landscape.

Then several baked illumination tests were done to reduce the lightmap detail and intensity faults like in Unity's automatically generated uvs to make the Lightmap of all static objects.

Then tests were done to make the occlusion culling with the dimensions of the scene, but unfortunately the culling occlusion of Unity in the free version is quite defective and produces serious failings generating enough clipping during the whole scene for any object, being hidden or not, and outside or within the plane of the chamber. So this option is discarded from the final demo by not working correctly occlusion culling the camera.

For materials mention the limited options that Unity poses regarding generating materials with textures that in Shader for PBR does not recognize like the Translucency map for the skin material. In engines like Unreal the options are endless to customize materials, because the use of a system of nodes that are added to achieve the desirable visual quality. Unity is limited to give all these possibilities to the artists so to get a greater range of different materials it is necessary to work on new shaders but they do not reach the possibilities and simplicity with which an artist is able to work in Unreal.

CHAPTER 5: DEVIATIONS

STAGE	TASK	TIME	TOTAL
1	Find out how to achieve the best viewing / shaders configuration	22 h	22 h
1	Work options to improve performance	28 h	50 h
1	Study the forms of lighting and shading used by Unity	30 h	80 h
1	Find out how to use effects and post-effects optimally visually	20 h	**100 h**
2	Design an attractive and quality gameplay	40 h	40 h
2	Work models, materials, textures and animations in order to achieve a professional finish	150 h	**190 h**
3	Realize the scripts that create the game flow	20 h	20 h
3	Program character movement and player actions	35 h	**55 h**
4	Make final technical proposal	7 h	7 h
4	Make the final memory	60 h	67 h
4	Perform the defense of the project.	10 h	**77 h**
			422 h

Figure 51: Final planning.

In the development of the demo several impediments have arisen that have made the section of testing and tests to take longer than due.

When working on the design of high-poly models. My work at Zbrush was limited to whether the program crashed or simply could not carry the graphic level that required it. After time of several problems I got through the last version of the program that let me export and work with the requirements that I asked for professional level.

In Unity I had to spend a lot of time perfecting the lighting of my scene due to mistakes that came from baking the lighting in my stage. Unity does not handle parameters to allow quality lightmaps to be done by generating rather imprecise uv problems that do not correctly define the value of the light intensity at the limits of the uvs. Producing black edges that are not natural to the shape and affect the result obtained.

The culling occlusion is another disaster to improve on Unity not only does not work well but produces serious popping effects for all elements that are considered in the culling occlusion of the scenario.

The models have had to be remade to improve the graphical load to allow Unity to generate good ligthmaps on them. They have also had to be exported again because the pivots of symmetrical elements produce localization errors causing some of these objects to be rotated 90 or 180 degrees in one of its axes and even displaced from the original zone in which were placed in 3Ds Max. Because of the fact that Max does not initially adjust several of its pivots and it must be done manually to avoid these errors.

For the animation of the character it had to manually adjust the weight of several of the vertices in the rigging automatically generated by Mixamo, in joints such like wrists or shoulders. And because the character has a small layer this was animated with the skeleton but its weights produced errors of overlapping vertices, which generated impossible shadows in a mesh like the layer.

Unity shaders do not allow generating image for backfaces in PBR materials and appear transparent. It was necessary to look for alternatives to solve this defect like to generate the same mesh but with the inverted normals, or thanks to a user that rose in the Unity store, to use four types of Shader similar to those of the PBR of Unity that allow the representation of the backfaces.

CHAPTER 6:
CONCLUSION

My TFG has enabled me to figure out how to develop high-quality graphics games on Unity.

Within the initial difficulty that the program shows to configure a high visual level, I have been able to supply it thanks to the extensive search of information by the demos of Unity and its official tutorials. These tell how to manage the program to achieve this quality like professionals.

The objectives covered in the technical proposal have been fully achieved. The graphic quality obtained is much higher than the one provided by Unity by default. The cutscenes maintain an equal quality of good that the rest of the demo and do not present load times. The gameplay of the demo is controlled in a natural way and is quite professional with a pretty good finish in the animations and the transition between them.

This knowledge any interested developer can start to develop a game with this caliber of lighting and graphics in Unity. Without expecting to be limited by the graphical options unlike using the default settings that Unity gives start in projects.

With all that I know that I can do now games with an impressive visual graphics, but my Unity knowledges have exceeded expectations too and I see myself able to start on the development of a video game with hipperrealistic graphics optimized like the main goal.

REFERENCES & BIBLIOGRAPHY

[1] Unity	https://unity3d.com/es
[2] Visual Studio	https://www.visualstudio.com/es/
[3] 3Ds Max	http://www.autodesk.es/products/3ds-max/overview
[4] Maya	http://www.autodesk.es/products/maya/overview
[5] Zbrush	http://pixologic.com/zbrush/features/overview/
[6] Photoshop	http://www.adobe.com/es/products/photoshop.html
[7] Quixel	http://quixel.se/
[8] Unreal Engine	https://www.unrealengine.com/what-is-unreal-engine-4
[9] TopoGun	http://www.topogun.com/
[10] Unfold3D	http://unfold3d.boards.net/
[11] xNormal	http://www.xnormal.net/
[12] Mixamo	https://www.mixamo.com/

Tutorials links:

Adam Interior tutorial:	https://www.youtube.com/watch?v=0mxr6bFenfE
Particles tutorial:	https://www.youtube.com/watch?v=_cAf5jMqZyM
Third Person tutorial:	https://www.youtube.com/watch?v=7NktwerZFro