

# Queen's Fate

3D modeling and development of a multiplayer fighting game

Alexandra Moreno Díez

Final Degree Work  
Bachelor's Degree in  
Video Game Design and Development  
Universitat Jaume I  
May 7, 2021

Supervised by: Diego José Díaz García, PhD.



# Index

## Introduction

- Motivation

- Objectives

- Environment and Initial State

## Planning and resource evaluation

- Planning

- Resource evaluation

## System analysis and design

- Requirement analysis

## Work development and result

- Game concept

- Characters

  - Design

  - Modeling

  - Texturing

  - Rigging

  - Animating

- Environments

  - Design

- Unity Basics

- Menus and HUD

  - Main menu

  - Controls menu

  - Character selection menu

  - Map selection menu

  - End game menu

  - HUD

- Game scene

  - Loading the characters

  - Camera

  - Player mechanics

  - Platforms

- Multiplayer implementation

  - Character selection menu

  - Game scene

- Sound

Music  
Sound effects

**Result**

**Conclusions**

**External links**

# **Introduction**

## **Motivation**

The main motivation for this work is wanting to develop a project in which the artistic section has to be worked on thoroughly, since it is the section I'm most interested in, and in a game of this style it is important that the models and animations are well worked. It also is a project that wraps all the contents of the degree, since it goes through the whole process of developing a video game, including the construction of the plot, the conceptual design of characters, environments, menus and HUD, the modeling and animation of said characters, the design of the game mechanics, the application of gamification techniques to balance the gameplay, and the coding of all the game logic, thus, being able to complete this project would mean that I've acquired all the necessary knowledge that is expected to acquire after finishing this degree.

## **Objectives**

The objective is to develop a complete game with original models and animations in which players fight each other in a 2D combat system, with many interesting mechanics that offer an extended gameplay.

## **Environment and Initial State**

The project is going to be entirely developed by a single individual, myself, working on my own with my own equipment. It started from a completely blank canvas, and the only previous knowledge came from all the projects developed at the degree.

I made the decision to develop it using mainly three softwares: Blender for 3D modeling, Photoshop for texturing and 2D sprites (like HUD icons, menus, etc), and Unity as a game engine. I came to this decision since those are programs I've previously worked with and I feel quite comfortable using them.

## Planning and resources evaluation

### Planning

The task was carried out as shown in the following table. This order was decided based on some logical decisions: before starting to do anything, you need to be sure what exactly it is that you want to do, that's why an initial design of mostly all the elements in the game is a necessary first step. After that, I'll start with the development of all those elements. The order of said development could have taken many forms, but I think it is important to leave the coding of the core mechanics as the last development step. It could be possible to do it before, but if you were about to code, for instance, the movement of the character without having the actual character, you would need to have some model that worked as a placeholder to test the code, and that seems rather less efficient than doing all your characters, environments, menus and HUD and then code directly with them.

<ul style="list-style-type: none"><li>- Establish the concept of the game: Look for references and draw sketches of the environments, characters, menus and HUD elements until I reach some final designs that satisfy me. Raw design of the main game mechanics</li></ul>	1st week of February 10 hrs
<ul style="list-style-type: none"><li>- Environments design Draw the 2D scenarios with the chosen softwares (mainly Photoshop) and model the 3D elements of the scenarios with the chosen softwares (probably Blender, 3DS Max or Sculptris).</li></ul>	2nd-3rd week of February 30 hrs
<ul style="list-style-type: none"><li>- Character design Model the 3D characters with the chosen softwares (probably Blender, 3DS Max or Sculptris).</li></ul>	4th week of February and 1st week of March 50 hrs
<ul style="list-style-type: none"><li>- Character animation Rigging and animating the characters with some software that allows it (for example Blender).</li></ul>	2nd-4th weeks of March 50 hrs
<ul style="list-style-type: none"><li>- HUD and menu design and setting all the elements in a game engine Design all the menus and HUD elements and add the models, scenarios and all the necessary elements (camera, HUDs and HUD etc.) to a game engine (Unity) to get everything ready to start coding.</li></ul>	1st week of April 10 hrs
<ul style="list-style-type: none"><li>- Designing and coding the mechanics Establish the final mechanics and code them, create a well-built animator to manage the animations of each of the mechanics and the transitions between them correctly. Generate the changes in the state of the game caused by each of the mechanics (decrease health, death, etc.).</li></ul>	2sd-4th weeks of April 70 hrs

<ul style="list-style-type: none"> <li>- Multiplayer implementation Make all the necessary adjustments so that the game can be played by two people.</li> </ul>	1st-2nd weeks of May  50 hrs
<ul style="list-style-type: none"> <li>- Testing and correction Allow several people to play the game to detect possible errors or shortcomings to correct them later.</li> </ul>	3rd-4th weeks of June 30 hrd
<ul style="list-style-type: none"> <li>- Development of the final report and presentation</li> </ul>	The report was developed throughout the project as it progresses, and the presentation was during the 1st week of June  20 hrs
TOTAL	<b>320 hrs</b>

### Resources evaluation

As mentioned before, the project is going to be entirely developed by myself. As for the equipment, nothing more than a computer is needed. It has already been mentioned that the main softwares used will be Blender and Unity, both completely free, and Photoshop, which has a free student license available. Having established that, the only cost of this project is the time needed to develop it.

## System analysis and design

### Requirement analysis

#### Functional requirements

FR1	
Input	Play
Output	Start game
Description	The user asks to start a new game and the application forwards him to the new screen to set the conditions of the new game.

FR2	
Input	Play again
Output	Restart game
Description	After a game finishes, the user asks to start a new game and the application forwards him to the new screen to set the conditions of the new game.

FR3	
Input	End playing
Output	Quit game
Description	The user asks to stop playing and thus the application shuts.

FR4	
Input	Adjust volume
Output	Volume update
Description	The user requests to change the current volume settings. The volume settings update to the ones specified by the user.

FR5	
Input	Player inside the game
Output	Correct interaction between the player and the game
Description	All the inputs of the player are properly processed and the application answers to them the way it is expected to.



FR6	
Input	Character selection
Output	Assign a character to the player
Description	The user can choose which character they want to use. The character will be properly assigned and loaded in the new game.

FR7	
Input	Map selection
Output	assign a map to the game
Description	The user can choose which map they want to play on. The map will be properly assigned and loaded in the new game.

FR8	
Input	Control character
Output	Change the player's position/action inside the game
Description	The user can control their character inside the game, deciding which action to take based on keyboard inputs.

FR9	
Input	Characters interacting in the game
Output	Display animation
Description	The game will show different animations for each of the actions the player can do inside the game.

#### Non-functional requirements

NFR1	
Input	Download the game
Output	Device compatibility
Description	The application should be compatible with all devices with any of the following operating systems: Windows 10, linux, macOS

NFR2	
Input	Execute the game
Output	Resolution compatibility
Description	The application should be compatible with different screen resolutions, adapting its elements to each one of them

NFR3	
Input	Navigate inside the game
Output	Simple and intuitive UI
Description	The application will be easy to use and understand

NFR4	
Input	Design the game
Output	Original aesthetic
Description	The application will have its own unique style, provided with original designs.

NFR5	
Input	Design the game
Output	Code simplicity
Description	The code will be as simple and optimal as possible, making it easily rectifiable and expandable.

## Work development and result

### Game concept

The concept of the game has pretty much fully been described in the sections above. It is a multiplayer fighting game in which two players fight each other in a 2D system using many combat mechanics.

Actually, this section's purpose is to talk about the background of the game, the story, it's plot. It may not seem like something too relevant gameplay wise, but that's really not true, it's not the same to tell the player "go fight that monster with no reason", than "go fight that monster because he kidnapped your princess and she's in danger". Context and story actually help players feel the game a lot more, empathising with the story, making it their story and making them feel committed to it.

So for my game, I asked myself two main questions, "What are the players fighting for" and "Why are the characters only females?", I wanted them to fight for something big and important, something that made it impossible for them to lose, and I started building around the concept of "power", as if they would obtain any form of power by winning.

After some thinking and joining that idea of power with the fact that the only ones that are fighting are females, I ended up coming up with the following story:

*"The fate of a country is in the hands of a single fight. Himeji is a country of warriors ruled by women, in which the title of Queen is mainly given by bloodline. However, contenders can challenge the ruler to a fight for the title. When the last Queen passed away, she left no children, so the council decided to celebrate a tournament in which participants will fight each other in order to find the strongest warrior, the one that will rule the whole country."*

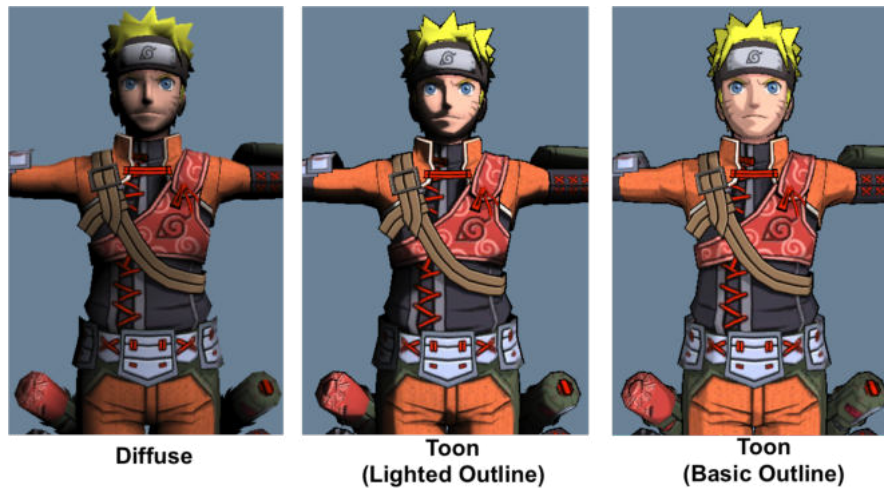
### Characters

In this section we will go through the whole process of creating the characters. Since the steps are rather similar for both of the characters, the process will be in depth explained for one of the characters, mentioning the major differences between the two of them.

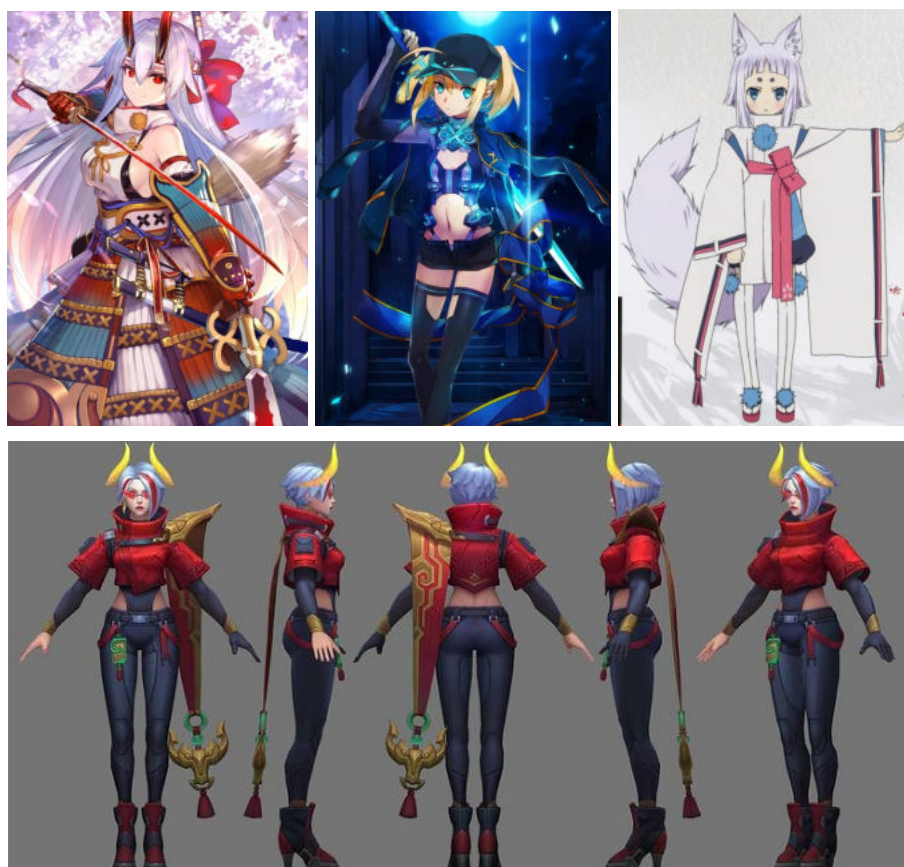
### Design

The characters are two females designed in anime style, japanese cartoons characterized for not being too disproportionate, keeping the body proportions close to real life ones, and also for having big eyes and small mouth and nose.

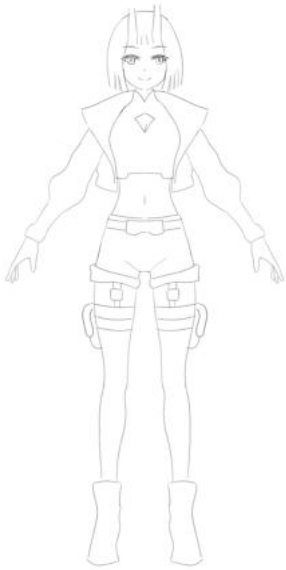
They were modeled in 3D, textured with a painting software and shaded using cell or toon shading, a type of non-photorealistic rendering designed to make 3D models appear to be flat by using less shading color instead of a shade gradient or tints and shades. Here's a reference to the visual differences between a standard diffuse shader and varying configurations of a toon shader. The expected result is the one shown in the last picture.



Since the game is set in Japan, the characters have references to folkloric Japanese creatures, like the Oni and the Kitsune. Clothes however, are a mix between rather futuristic and traditional elements, giving the characters its own, characteristic style. The following pictures provide a full example of the main features of the characters' visual design: the anime style, the traditional Japanese elements and the futuristic clothes.



The combination of all those elements gave room to the first sketches and designs.



After establishing the visual design of the characters, I started working on their personalities. It is important to establish the personality of the characters, even though at first sight it may not seem as something too relevant in a fighting game, but that's actually far from reality, as the body language of a character says a lot about their personality, and a fighting game provides a wide range of actions for the characters, which all together have to build a solid and logic persona.

I wanted them to have rather opposite personalities, being one of them calmed and upright and the other one enthusiastic and impulsive. I also wanted to choose their names according to their personalities. The result for all this construction is as follows:

- Yura: From Japanese kanji 悠 (yū) meaning "permanence" combined with 良 (ra) meaning "good". She's the commander of the imperial troops, one of the strongest women alive. She's an experienced warrior who's participated in hundreds of fights over the years. She's a righteous woman who fights to protect those she cares about. She's a natural leader, capable of managing any situation with a clear mind and a thoughtful reasoning.
- Tamachi: Cognate to standard Japanese たましい[魂] tamashii "soul; spirit", it can also be read as 珠智, 珠 means "pearl", 智 means "smart". She's the niece of the last Queen. She's really clever and has a lot of enthusiasm, but she's still a child, she has never been in charge before so she doesn't really know how to manage a town, and her mind and ideas are not completely developed and established, as well as her fighting skills. She has a lot of raw potential, but she's still developing it.

### **Modeling**

3D modeling is the process of developing a mathematical coordinate-based representation of any surface of an object in three dimensions via specialized software.

Three-dimensional models represent a physical body using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc. Being a collection of data (points and other information), 3D models can be created manually, algorithmically

(procedural modeling), or by scanning. In this case I manually created the models for my video-game using a 3D computer graphics software. The process I used to generate the models is known as Polygonal modeling:

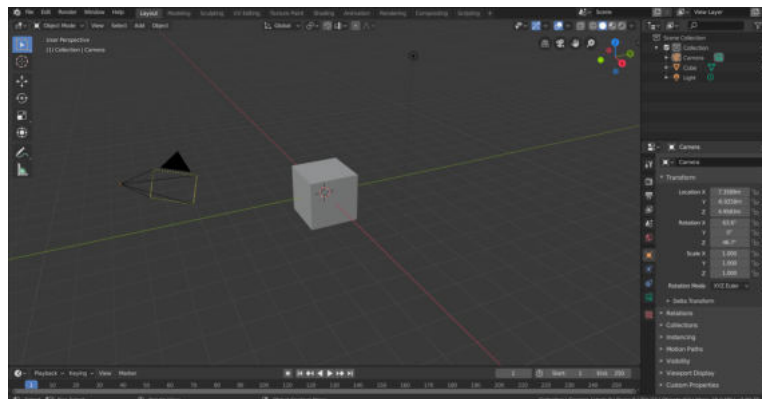
- Points in 3D space, called vertices, are connected by line segments to form a polygon mesh. These kinds of models are flexible and computers can render them so quickly. However, polygons are planar and can only approximate curved surfaces using many polygons.

and, the kind of models generated are shell or boundary models:



- These models represent the surface, i.e. the boundary of the object, not its volume.


The software I chose to create the models is Blender, a free and open-source 3D computer graphics software. I had previously worked with it and I was used to the interfaces and the shortcuts so I feel pretty comfortable working with it.

This is Blender’s default screen when you open it or when you create a new project, it includes a 3D cube, a camera and a light. I deleted the camera and the light and left only the cube, since those elements are only necessary if you are going to use the models inside of Blender, to get some renders or some video clips, and I am only going to use Blender to create the models.






Blender has a wide range of features and tools to modify 3D meshes. One of the main concepts that we need to know before editing our model is the *mode* in which we’re manipulating the mesh. Blender has several modes that allow it to manipulate the mesh in different ways. I’ll be mostly using three modes:

	Object mode	The default mode, available for all object types It is dedicated to Object data-block editing (e.g. position, rotation, size), meaning it allows to manipulate only entire objects.
	Edit mode	A mode available for all renderable object types, as it is dedicated to their “shape” Object Data data-block editing (e.g. vertices/edges/faces for meshes, control points for curves/surfaces, strokes/points for Grease Pencil, etc.). This is the mode that allows you to modify the shape of an object, thus, this is the mode that I used the most.

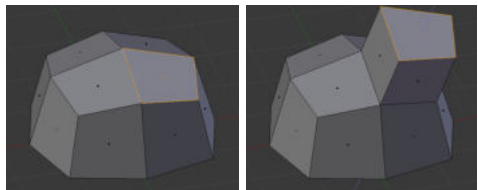
	Sculpt mode	A mesh-only mode that enables Blender's mesh 3D-sculpting tool. 3D sculpting is the use of software that offers tools to push, pull, smooth, grab, pinch or otherwise manipulate a digital object as if it were made of a real-life substance such as clay. It is another way of manipulating the shape of a mesh, typically manipulating several vertices at once. I used it mostly to add some details.
---	-------------	---

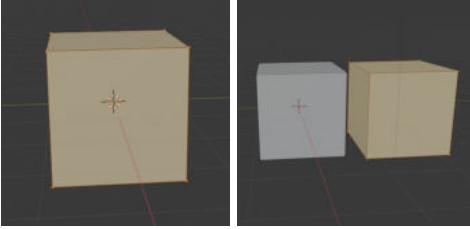
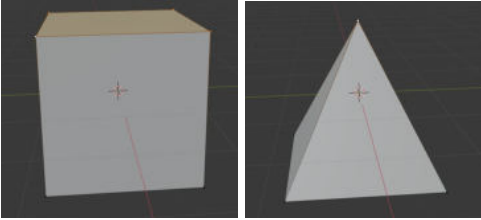
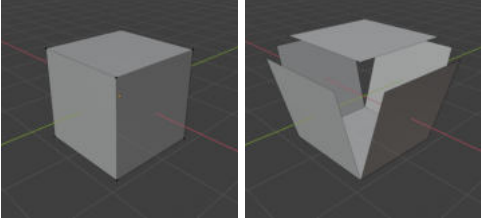
Like it's already been said, the mode I mostly worked at is the Edit Mode. This allows you to modify the mesh at will, with a wide variety of tools. The modifications made to a mesh in Edit Mode are known as Transformations. Transformations refer to a number of operations that can be performed on a selected Object or Mesh that alters its position or characteristics. Basic transformations include:

	Move	In Object Mode, the move option lets you move objects. Translation means changing location of objects. Edit Mode also lets you move any elements that make up the object within the 3D space of the active 3D Viewport.
	Rotate	Rotation is also known as a spin, twist, orbit, pivot, revolve, or roll and involves changing the orientation of elements (vertices, edges, faces, objects, etc.) around one or more axes or the Pivot Point.
	Scale	Scaling means changing proportions of objects. Pressing S will enter the Scale transformation mode where the selected element is scaled inward or outward according to the mouse pointer's location. The element's scale will increase as the mouse pointer is moved away from the Pivot Point and decrease as the pointer is moved towards it. If the mouse pointer crosses from the original side of the Pivot Point to the opposite side, the scale will continue in the negative direction and flip the element.

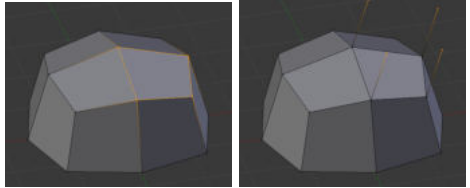
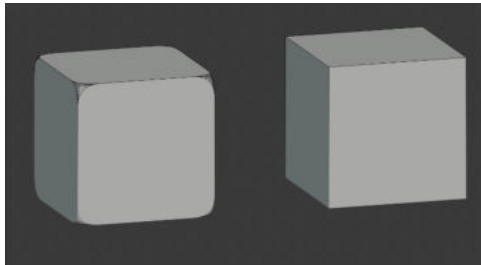
Those are the main transformations that, given a certain mesh, allow for the modification of its elements at will. However, there are more complex and sophisticated tools with which you can not only modify the existing elements, but generate new ones, combine and join existing ones, etc. Those can be categorized as mesh, vertex, edge and face tools. Here follows a list of all the tools used during this project:

#### Mesh Tools

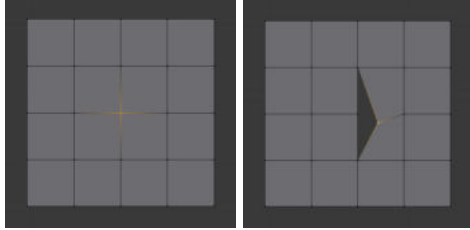
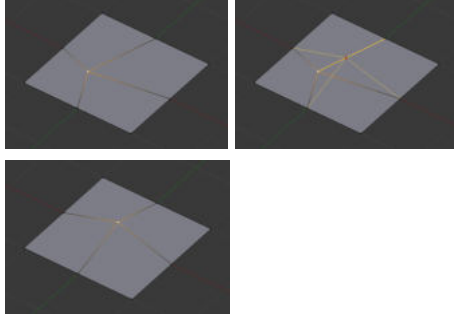
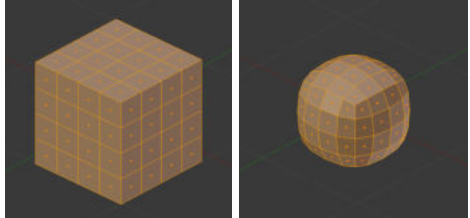
Extrude	Extrude Faces duplicate faces, while keeping the new geometry connected with the original vertices.	
---------	---	--

Duplicate	<p>This tool simply duplicates the selected elements, without creating any connections with the rest of the mesh (unlike extrude, for example), and places the duplicate at the location of the original. Once the duplication is done, only the new duplicated elements are selected, and you are automatically placed in move mode, so you can move your copy elsewhere...</p>	
Merge	<p>This tool allows you to merge all selected vertices to a unique one, dissolving all others. You can choose the location of the remaining vertex in the menu this tool pops up before executing (at first, at last, at center, at cursor)</p>	
Split	<p>Splits (disconnects) the selection from the rest of the mesh. The border edges to any non-selected elements are duplicated. The "copy" is left exactly at the same position as the original, so you must move it to see it clearly...</p>	

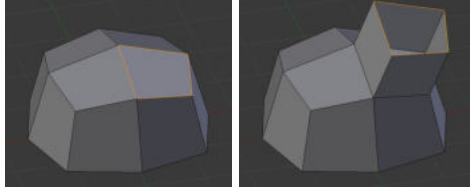
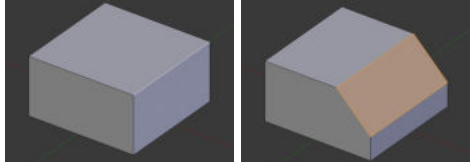
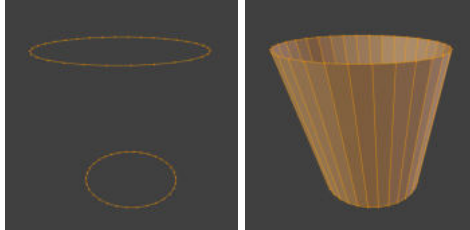
#### Vertex tools

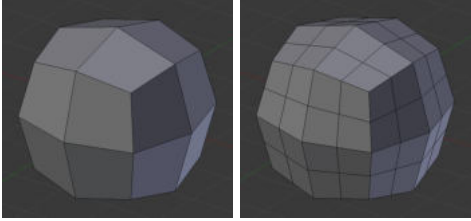
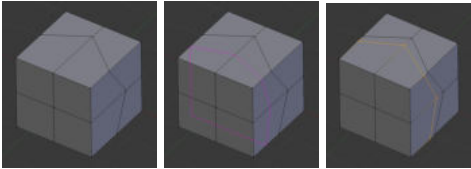
Extrude vertices	<p>Extrude vertices as individual vertices</p>	
Bevel vertices	<p>The Bevel tool rounds off edges or corners of a mesh at the point of the selected vertices. In "vertex only" mode, the Bevel Vertices tool works on selected vertices but the option to switch to Bevel Edges is available. By doing so, more vertices are added in order to smooth out profiles with a specified number of segments.</p>	



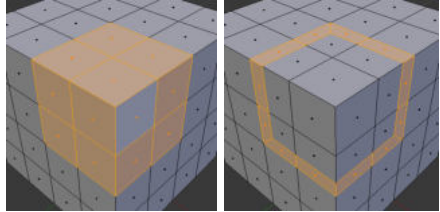
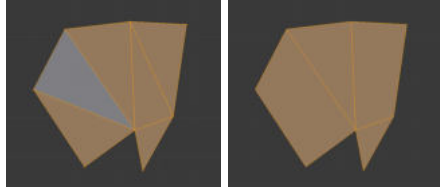
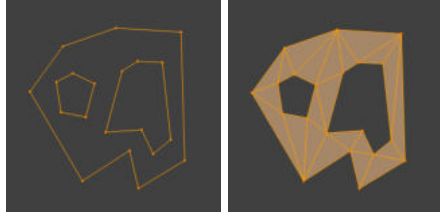
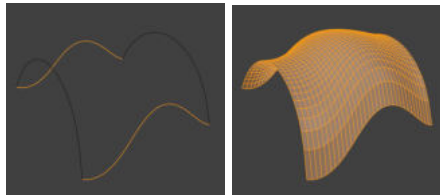
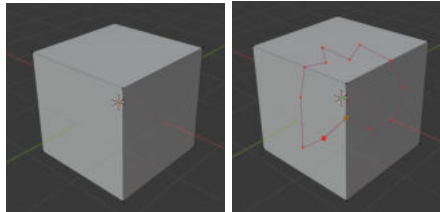
Rip vertices	Rip creates a “hole” in the mesh by making a copy of selected vertices and edges, still linked to the neighboring non-selected vertices, so that the new edges are borders of the faces on one side, and the old ones, borders of the faces on the other side of the rip.	
Slide vertices	Vertex Slide will transform a vertex along one of its adjacent edges. The nearest selected vertex to the mouse cursor will be the control one. Move the mouse along the direction of the desired edge to specify the vertex position.	
Smooth vertices	This tool smooths the selected vertices by averaging the angles between the faces.	

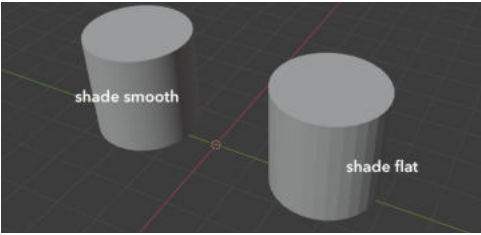
### Edge tools

Extrude edges	Extrude edges as individual edges	
Bevel edges	The Bevel tool allows you to create chamfered or rounded corners on geometry. A bevel is an effect that smooths out edges and corners.	
Bridge edge loops	Bridge Edge Loops connects multiple edge loops with faces.	

Subdivide	Subdividing splits selected edges and faces by cutting them in half or more, adding new vertices, and subdividing accordingly the faces involved. It adds resolution to the mesh by dividing faces or edges into smaller units.	
Loop cut and slide	Loop Cut and Slide splits a loop of faces by inserting a new (or more than one) edge loop intersecting the chosen edge.	

### Face Tools

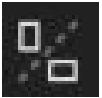
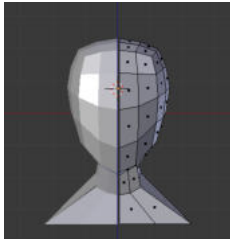
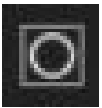
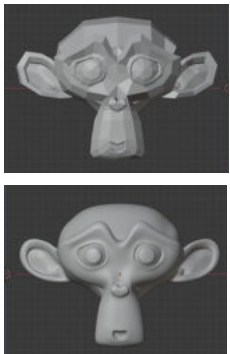

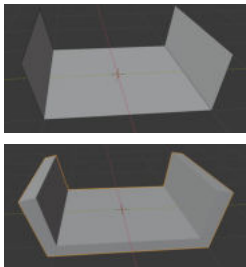
Inset faces	This tool takes the currently selected faces and creates an inset of them, with adjustable thickness and depth. Think of it as like creating an edge loop, but relative to the selected edges, even in complex meshes.	
Triangles to quads	This tool converts the selected triangles into quads by taking adjacent triangles and removing the shared edge to create a quad, based on a threshold. This tool can be applied on a selection of multiple triangles.	
Fill	The Fill option will create triangular faces from any group of selected edges or vertices, as long as they form one or more complete perimeters.	
Grid fill	Grid Fill uses a pair of connected edge loops or a single, closed edge loop to fill in a grid that follows the surrounding geometry.	
Intersect (Knife)	The Intersect tool lets you cut intersections into geometry. It does not calculate interior/exterior geometry. Faces are split along the intersections, leaving new edges selected.	


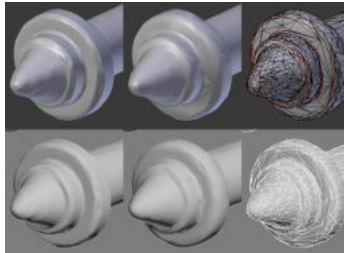
<p>Shade smooth and flat</p>	<p>The appearance of the mesh edges are determined to be evened out or well defined within the 3D Viewport and render. In Edit Mode, individual faces can be selected to determine which faces are smoothed or flattened.</p>	
------------------------------	---	--

With this, almost all the necessary elements to start modeling the characters are defined. One last thing to keep in mind are modifiers. Modifiers are automatic operations that affect an object's geometry in a non-destructive way. With modifiers, you can perform many effects automatically that would otherwise be too tedious to do manually (such as subdivision surfaces) and without affecting the base geometry of your object.

They work by changing how an object is displayed and rendered, but not the geometry which you can edit directly. You can add several modifiers to a single object and apply a modifier if you wish to make its changes permanent.

There are just a few but highly important modifiers used to speed up and ease the modeling process:

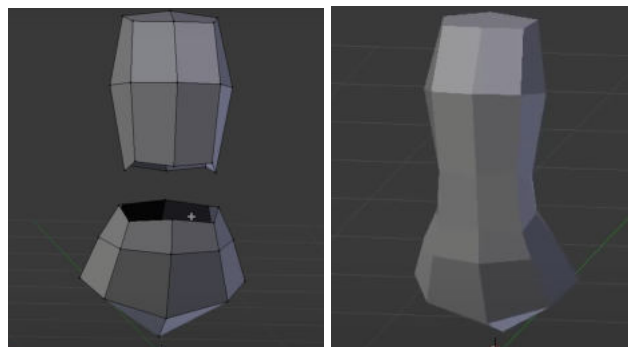
	<p>Mirror</p>	<p>The Mirror modifier mirrors a mesh along its local X, Y and/or Z axes, across the Object Origin. It can also use another object as the mirror center, then use that object's local axes instead of its own.</p>	
	<p>Subdivision surface</p>	<p>The Subdivision Surface modifier is used to split the faces of a mesh into smaller faces, giving it a smooth appearance. It enables you to create complex smooth surfaces while modeling simple, low-vertex meshes. It avoids the need to save and maintain huge amounts of data, and gives a smooth "organic" look to the object.</p>	
	<p>Solidify</p>	<p>The Solidify modifier takes the surface of any mesh and adds depth, thickness to it.</p>	

	Decimate	The Decimate modifier allows you to reduce the vertex/face count of a mesh with minimal shape changes.	
---	----------	--	---

Now that all the tools have been exposed and explained, all that's left is modeling the characters. I will go through the process, showing some steps and explaining the practical application of many of the tools previously explained.

It all started with a 3D cube, I deleted the right side of it and added the mirror modifier along the X axis. Since a T-posed character is practically completely symmetrical, except for in some cases the hair, clothes and some other accessories, the mirror modifier allows for a way faster modeling, not having to model the whole body but only half of it.

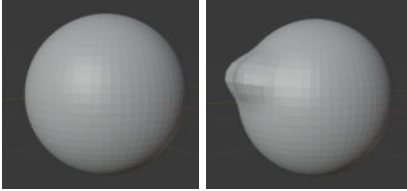
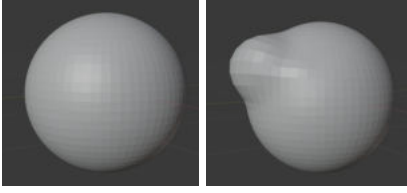
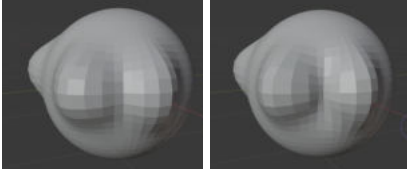
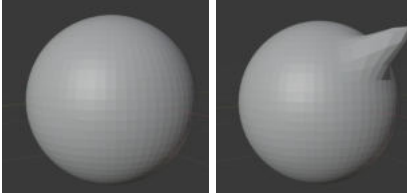
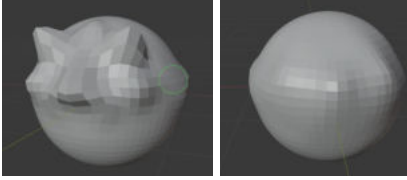
I then used the Subdivide Edge Tool with one iteration to split each of the cube faces into four and the Smooth Vertex Tool to smooth the shape of the cube. I adjusted the cube to fit in the shape of the chest using the Move Transformation on its vertices. After that I duplicated the cube with the Duplicate Mesh Tool and I scaled it negatively along the Z axis with the Scale Transformation. I adjusted this new cube to the shape of the hips. Once both were properly adjusted, I deleted the bottom faces of the chest cube and the top ones of the hips cube, I selected the edge loops generated after deleting the faces and I used the Bridge Edge Loops Edge Tool to generate faces connecting the two cubes, acting as the character's waist.



The next step was to start modeling the arms, legs and neck. In order to do this, I used the Knife Face Tool to generate a circle of vertices from which the arm will grow and the same thing for the leg and the neck. I deleted the faces inside those circles, selected the edges of the circles and used the Extrude Edge Tool to start building the arms, legs and neck.



Before keeping up with the arms and the legs, I applied two iterations of the Subdivision Surface Modifier to smooth the mesh. Once smoothed I swapped to Sculpt Mode to sculpt some details on the torso that would be quite tedious to model in Edit Mode. I used the following brushes to work on the sculpt:

Draw	Moves vertices inward or outward, based on the average normal of the vertices contained within the drawn brush stroke.	
Blob	Pushes mesh outward or inward into a spherical shape with settings to control the amount of magnification at the edge of the sphere.	
Crease	Creates sharp indents or ridges by pushing or pulling the mesh, while pinching the vertices together.	
Grab	Used to drag a group of vertices around. Grab selects a group of vertices on mouse-down, and pulls them to follow the mouse. And unlike other brushes, Grab does not move different vertices as the brush is dragged across the model.	
Smooth	Eliminates irregularities in the area of the mesh within the brush's influence by smoothing the positions of the vertices. The inverse of this tool is to sharpen the details in a mesh by applying a Laplacian smooth in the opposite direction.	

With my base mesh, I first used the grab brush to adjust some general proportions, moving around the shoulders, the hips and the butt.

After that I used the draw and blob brushes to add and reduce volume of more specific areas: I added volume to the chest, the stomach, the hip bones, the shoulder blades and the clavicles and I took volume of the area under the chest, the center of the back, and the navel.

Then I used the crease brush to sharpen and emphasize the navel, the chest, the clavicles, the butt, the shoulder blades, the center of the back, the hip bones, and the muscles of the stomach.

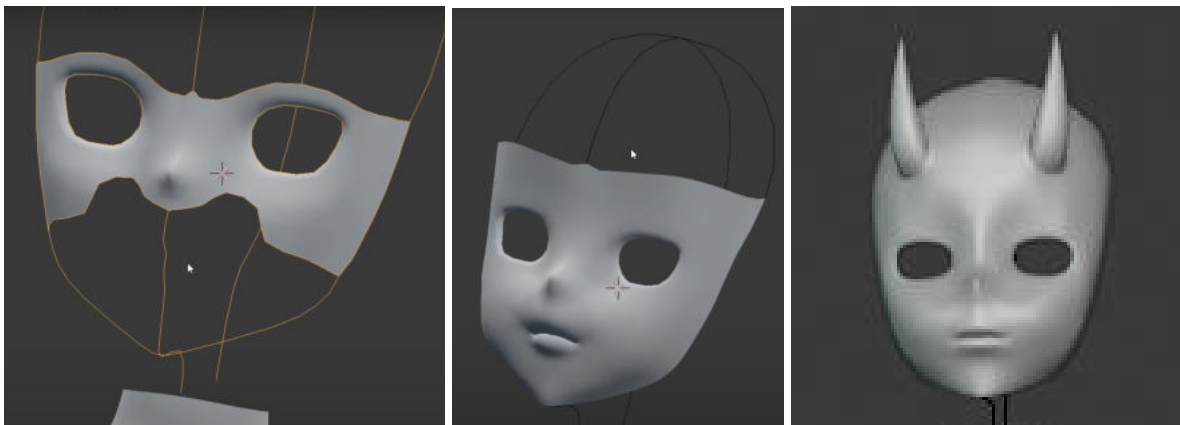
Finally I smoothed the whole piece putting special attention to the sharpest areas.



Once I finished sculpting I kept working on the legs. The process was quite simple: I kept selecting the circular edge loop and extruding it, scaling each loop to adjust to the size of the leg at that point and moving around some vertices when needed to some particular shapes, like the knee.



When I finished the legs I began to work on the face. I did not know exactly where to begin, and after some research I found out what some people do is to trace an outline of vertices for the front view, another one for the side view and two for the eyes, and then they start filling the spaces between the outlines. So that is what I did. It is a simple process that only requires Vertex and Edge Extrusion and Move Transformation of the vertices.



I did not finish the entire head at first as I thought maybe the hair would cover a huge area of it so it would not be necessary to model it as it would not be visible. So I left it like that and kept going with the body. I finished the arms, it was the exact same process as the legs, extruding edge loops and adjusting some vertices by hand.

Once I had the arms I went back to the head to finish it. I started modeling the hair so once finished I could consider whether I should model the back of the head or not.

Each hair strand is a cube, subdivided once, with some added edge loops, adjusted by moving, rotating and scaling the loops. To do the small strands inside each big one I did the same as the first steps to create the arms: using the knife to cut a circle on the strand, deleting the faces inside the circle and extruding the loop a couple of times, scaling and moving it accordingly.



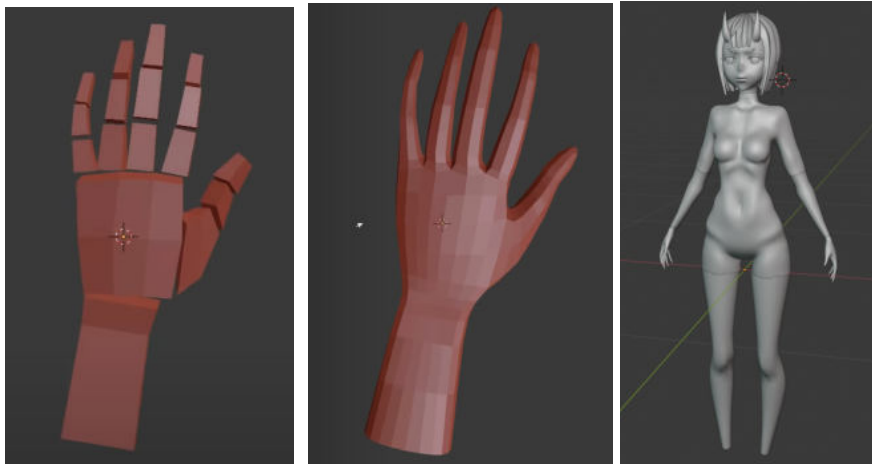
At the end I decided to model the whole head, since I was going to animate the hair so it was very likely that most of it would end up being visible at some point.

I also added the final details to the face: eyebrows, eyelashes and pupils. The eyebrows and eyelashes are practically tiny hair strands and so they were modeled following the same steps. The pupils are spheres scaled down on the Y axis and slightly moved and rotated to fit in their position.



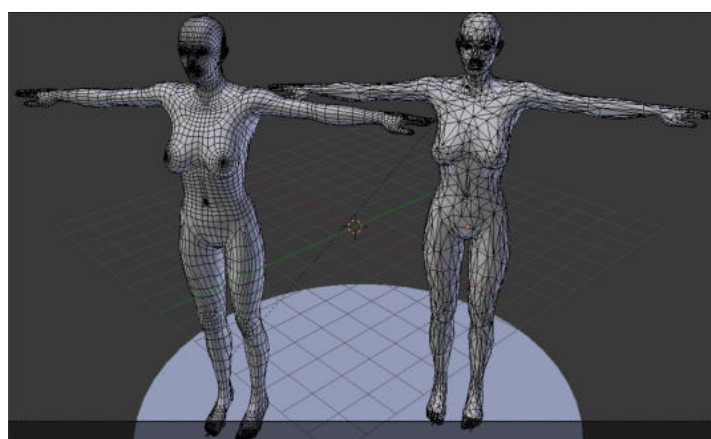
The only body parts left are the hands and feet. I used mostly the same process to model both: I started from a cube for the base of the hand, I subdivided it once I added two vertical edge loops, that way the base has three vertical loops and four faces on top of it, from which the fingers, except the thumb were modeled. The thumb comes from one of the side faces of the base. I selected those four top faces and the side face and I used the Duplicate Mesh Tool to generate a copy of them separated from the base cube. I extruded those faces to generate cubes. I then duplicated each of those cubes three times, being the three phalanges of each finger, and I adjusted their rotation, position and scale. I kept the three divisions on the thumb too, even if it only has two phalanges the

result looked better with the three of them. I then joined all the cubes of each finger between them and with the base of the hand with the Bridge Edge Loop tool. Finally I added a subdivision surface modifier and I made some final adjustments to the shape of the hand.



To finish with the body I modeled the feet, mostly the same way as I modeled the hands, making a cube for the base of the foot, using that cube to create small cubes for the fingers and joining it all together at the end.

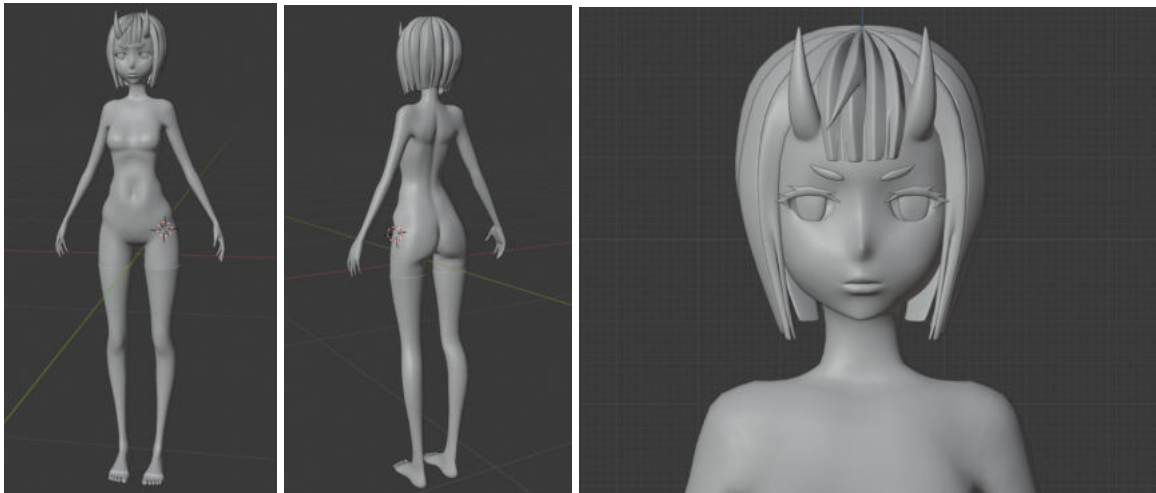
After everything was done I realized that the polygon count grew a bit more than I expected. It still was an amount that would have worked just fine inside a game engine, but since I wanted the game to work fluently in any kind of computer I thought it would be alright to try and reduce it a bit. Here's when I found the decimate modifier: it reduced the polygon count considerably without changing the quality of the model too much. What I did not know and only found out long after applying the modifier was the fact that it redistributed all the polygons of the mesh. A visual example will make it much clearer:



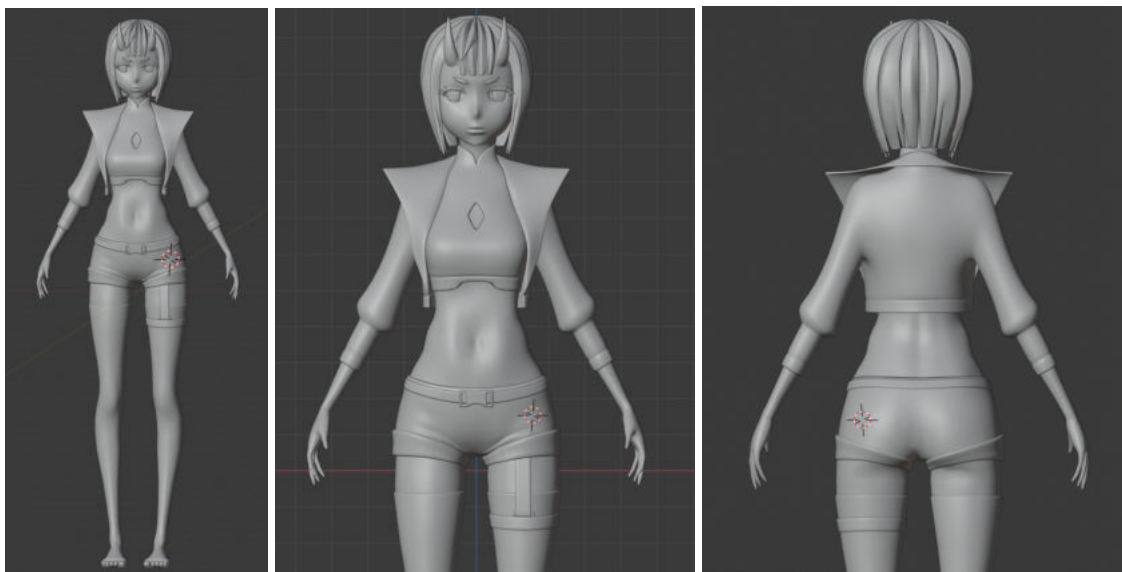
On the left there's an organized mesh of a character. On the right, the generated mesh after applying the modifier. It is completely disorganized and while it still looks good on T-pose, if we check the joints of the body, like the knees, elbows or phalanges, rigging and animating them with that topology would have been terrible, because they would deform as soon as they start bending.

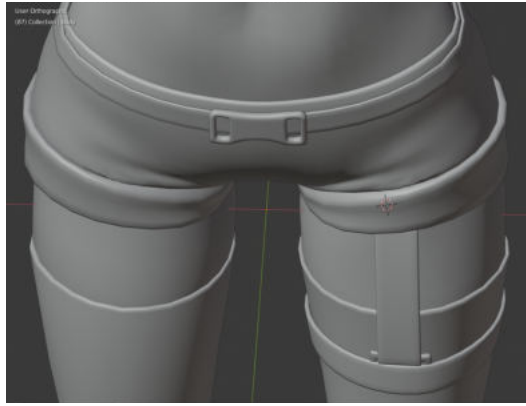


When I found out it was already too late and I could not revert the application of that modifier, so I had to go back to a different copy of the file, since I fortunately saved several copies in different files as I progressed, if I did not have those copies I would have had to redo the whole model. This version had everything done except the hands and feet, which I had to do again. So after redoing them I reduced the polygon count manually, deleting some edge loops. The model lost a bit of quality, but now the polygon count was considerably good and at the end, the area that lost most of the detail was the torso, which would be mostly covered by the clothes.



To finish the model I modeled the clothes. I did something quite similar to what I did to model the face: I did a vertex outline for the front view and another one for the side view and I filled the areas inside those lines by extruding edges, moving vertices and adding edge loops mainly. At the end each piece of clothing was actually flat, so I added a Solidify modifier to each of them to give them thickness, which can be appreciated in the last one of the following photos.





## Texturing

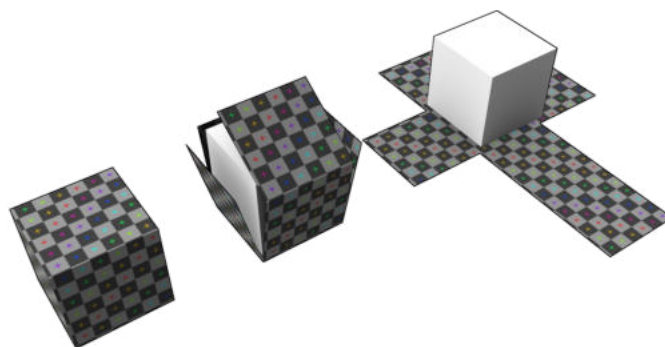
In order to texture the model, I generated the UV texture of it and then paint over it with a painting software.

UV mapping is the 3D modeling process of projecting a 2D image to a 3D model's surface for texture mapping.

UV texturing permits polygons that make up a 3D object to be painted with color (and other surface attributes) from an ordinary image. The image is called a UV texture map. The UV mapping process involves assigning pixels in the image to surface mappings on the polygon, usually done by "programmatically" copying a triangular piece of the image map and pasting it onto a triangle on the object.

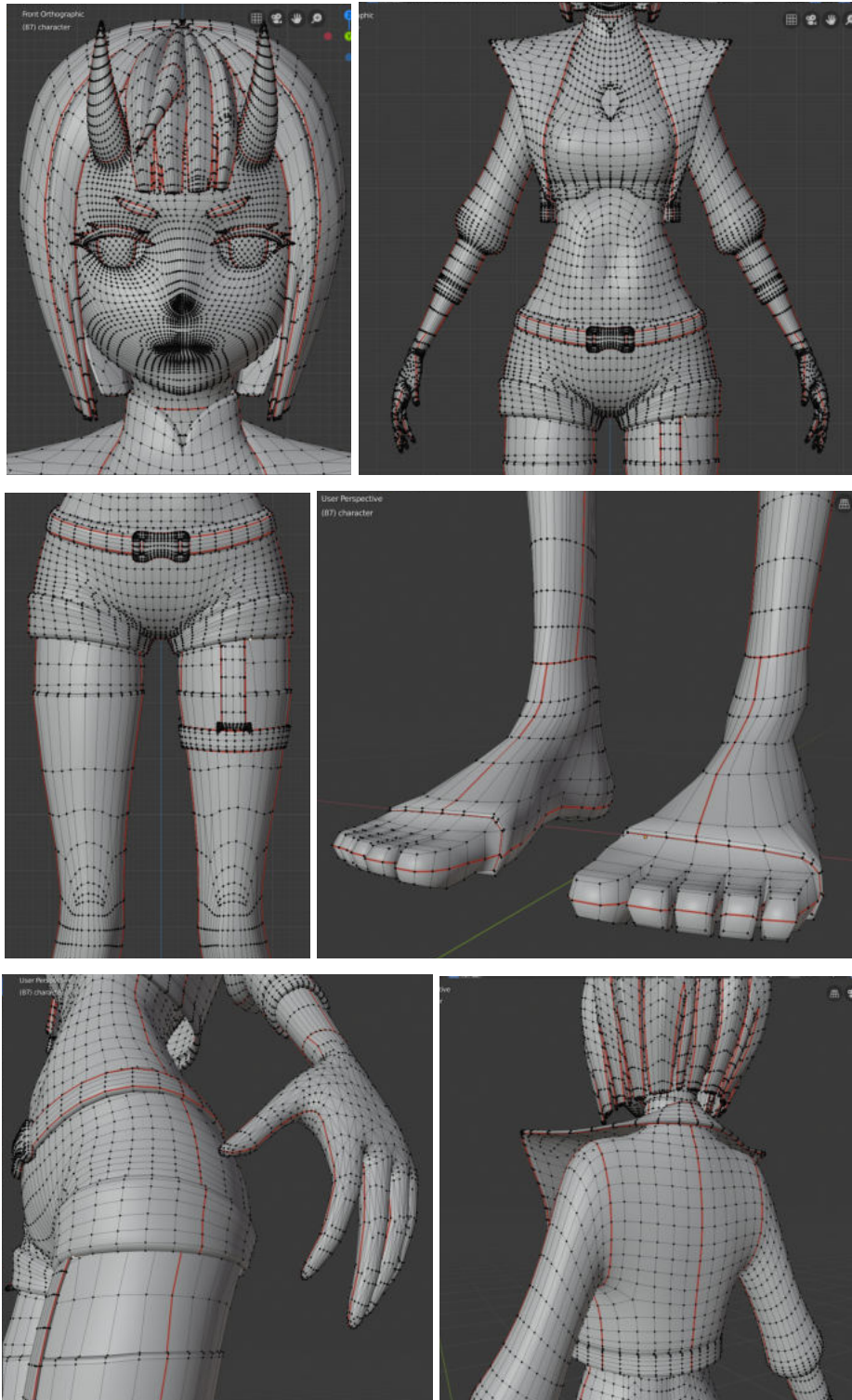
The process to generate the UV texture is known as UV unwrapping. When a model is created as a polygon mesh using a 3D modeller, UV coordinates (also known as texture coordinates) can be generated for each vertex in the mesh. The way I will be doing it is by unfolding the mesh at the seams, automatically laying out the triangles on a flat page.

Just like in sewing, a seam is where the ends of the image/cloth are sewn together. In unwrapping, the mesh is unwrapped at the seams. Think of this method as peeling an orange. You make a series of cuts in the skin, then peel it off. You could then flatten it out, applying some amount of stretching. These cuts are the same as seams.



Once the model is unwrapped, It generates a 2D stencil (like the one on the second picture) that can then be downloaded as a PNG and loaded in a painting software, Photoshop in my case, to paint over it.

I will now show the unwrap of my models, the final UV textures and the painted textures of the models.



The red edges are the seams. There are a couple of things to keep in mind when establishing the seams. First of all, it is usually best to try to hide them away if possible, to avoid having to paint over them later on in the process. You have to consider that a seam is the part where the mesh splits in the final texture, so if you plan on painting the model using not only plain colors but for example patterns, when painting over your UV texture you need to make sure that in the areas where a seam cut a continuous part of the mesh, you properly adjust the pattern so they are at the exact same position and there is not a visible discontinuity on the final model.



This model has a seam on the center of the body and it was painted not only with plain colors but with a texture, and the texture was not properly applied over the UV map, making sure that the two splitter parts of the body ended with the exact same part of the texture, so since they picked a very visible part to place a seam and they did not apply the textures properly, a discontinuity is clearly visible and it makes the model look wrong.

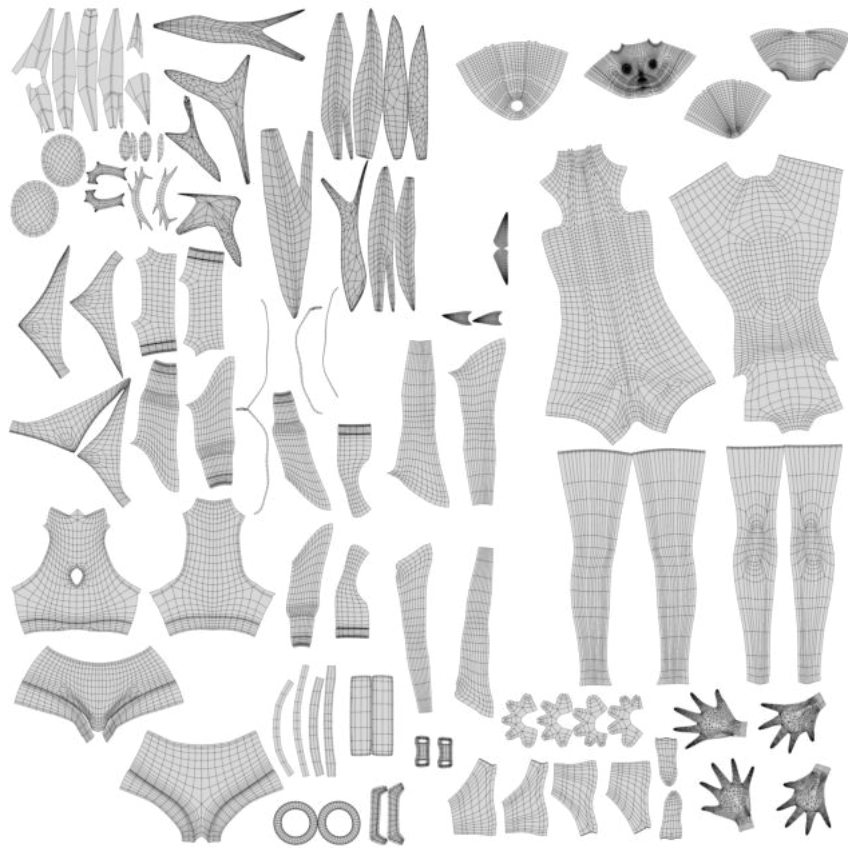
As well as choosing edges that are less prominent or that will be hidden, you can also select entire edge loops, which is helpful if you want to separate complex pieces of geometry that may not otherwise unwrap so easily.

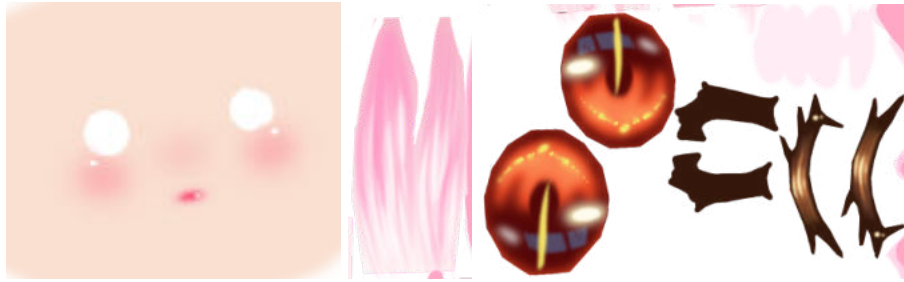
In my case I tried to place most of the seams on the sides of the characters, like under the arms, between the legs and between the fingers. However it was not too much of a problem if a seam was too visible because I was going to paint my models mostly using flat colors, so ensuring the continuity of the texture would be rather easy. I also used edge loops as seams to completely separate the arms, legs, head, hands and feet and deal with them independently.

When generating the PNG of the stencil, the default size of it was 1024x1024 pixels, but I changed it to 4096x4096 pixels, it is a really huge size and it would make the final texture very heavy but even though I was going to paint the model mainly with plain colors, I wanted to add some details and shades to small areas like the eyes, lips, cheeks and hair, since I was planning to make a zoom in of the character's face on the character selection screen of the game, and the image needed a lot of definition for those details to be visually noticeable.

I then proceeded to paint over that stencil. I did not use many special tools, just the default Photoshop brush, the gaussian blur filter to generally blur the shades and details, the finger tool to

blend some colors, specially the ones on the hair and the gradient tool to make some color transitions on the shirt and the socks, so the entire model did not look extremely plain and dull.





Once this texture is finished, all that's left is to assign it to the albedo channel of a material. As previously mentioned, the shader used for the material would be a toon shader. I downloaded a free toon shader for Unity, creating my own shader would have been an extremely hard task and it would have taken too much time, as I only have so little knowledge about the subject.



## Rigging

Rigging is a general term used for adding controls to objects, typically for the purpose of animation. Many features are involved in the process of rigging. For my rigging I used the following ones:

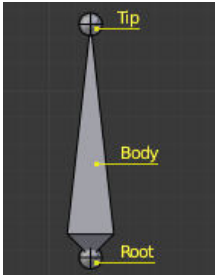
- Armature: an Armature in Blender can be thought of as similar to the armature of a real skeleton, and just like a real skeleton an Armature can consist of many bones. These bones can be moved around and anything that they are attached to or associated with will move and deform in a similar way.
- Constraints: Constraints are a way to control an object's properties (e.g. its location, rotation, scale), using either plain static values (like the «limit» ones), or another object, called «target» (like e.g. the «copy» ones).

I first created the armature. As said, an armature is a collection of bones parented between them. The bones with a parent will follow the movement of this one, just as in the human body, if you move your upper arm, the rest of the arm will follow.

An armature is like any other object type in Blender, it has an origin, a position, a rotation and a scale factor and it has an Object Data data-block that can be edited in Edit Mode. The bones in an Armature can be generally classified into two different types:

- Deforming bones: are bones which when transformed will result in vertices associated with them also transforming in a similar way. Deforming Bones are directly involved in altering the positions of vertices associated with their bones.
- Control bones: control Bones are not directly used to alter the positions of vertices; in fact, they often have no vertices directly associated with themselves.

Bones have three elements:

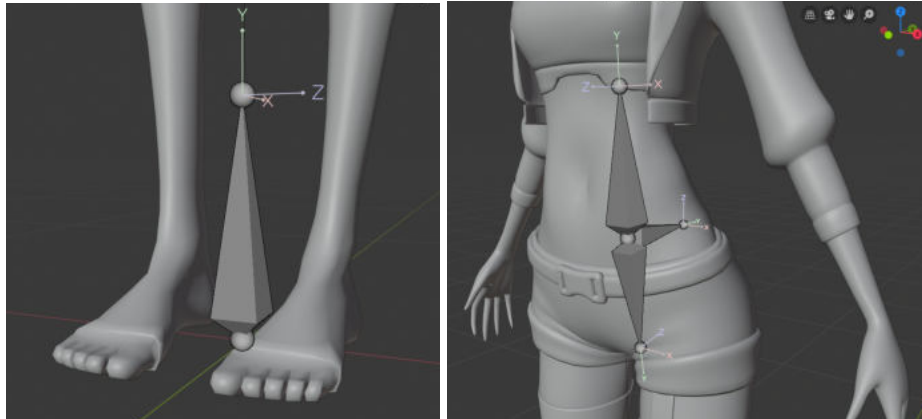
<ul style="list-style-type: none"><li>- The «start joint» is named root or head.</li><li>- The «body» itself.</li><li>- And the «end joint» is named tip or tail.</li></ul>	
---	---

In edit mode the three of them can be edited. Moving the body will move the whole bone while moving any of the joints will only move that specific joint, changing the bone's size, position and orientation.

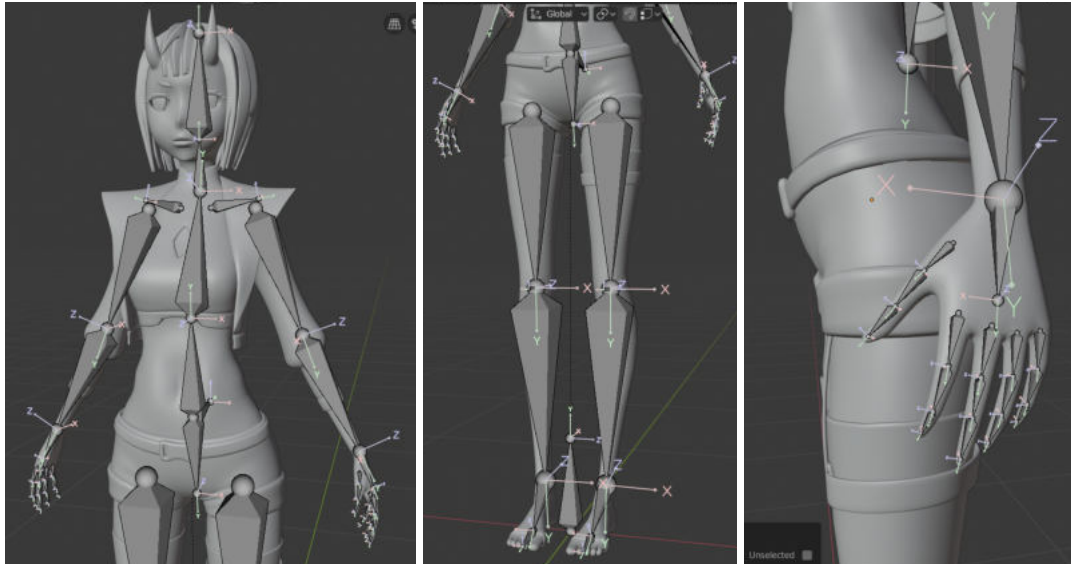
Extruding one of the joints will generate a new child bone parented to the first one. That's the main point for creating the armature. When you create an armature object in Blender, it creates just one

single bone, so you have to switch to edit mode and start creating new bones from that one. I used that first bone as a control bone placer between my character's feet. That bone will not deform the character but will be used to displace the whole character as it is displaced.

I then extruded its start joint to create another control bone that will work as the base to start building the rest of the armature. I then created two children of this bone, one for the hips, pointing downwards as all its children will be below it, and one for the lower spine, pointing upwards as all its children will be above it.



I just kept extruding and moving the new bones to their positions until I was satisfied with the result.



Once the armature is set, the next step is to assign the influence of the bones. Basically, a bone controls a geometry when vertices «follow» the bone. To do this, you have to define the strength of influence a bone has on a certain vertex. The simplest way is to have each bone affecting those parts of the geometry that are within a given range from it.

A first approximation to the right result is automatically generated by Blender. By selecting both the armature and the character (first the character and then the armature) and pressing Ctrl + P, you can

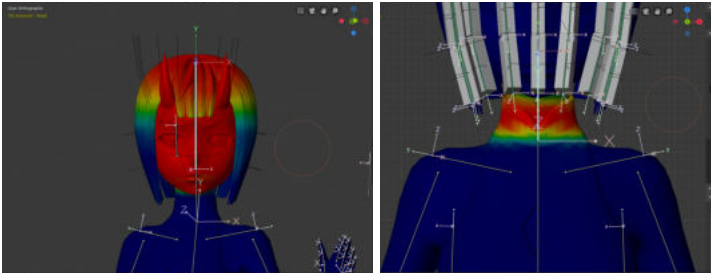
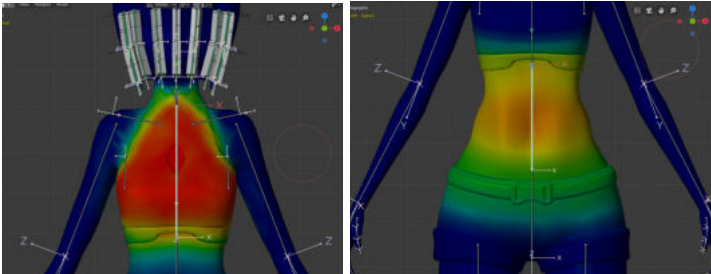


parent your character to the armature. And selecting the option Parent With Automatic Weights, Blender automatically calculates the influence of each of the armature's bones on each of the character's vertices. There is not much need to say that the result is quite far from being satisfactory. The auto generation does not assign the bones' influences, which will be called weights from now on, properly, especially if there are clothes involved. So I had to adjust the weights manually, for what Blender offers what is known as the Weight Paint Mode.

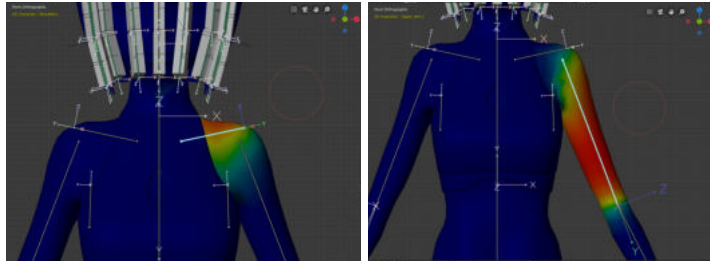
Inside the Weight Paint Mode the selected mesh object is displayed slightly shaded with a rainbow color spectrum. The color visualizes the weights associated with each vertex in the active bone. By default blue means unweighted, meaning that area won't be affected by the bone, and red means fully weighted, meaning that area will be completely affected by the bone.

This mode, then, allows the user to select a bone of the armature and paint its influence over the character with a brush. The size and strength of the brush can be adjusted. It is also important to allow for it to paint through the object and not just the front faces, that way you make sure that on the cloth areas, the skin under the clothes is also being painted.

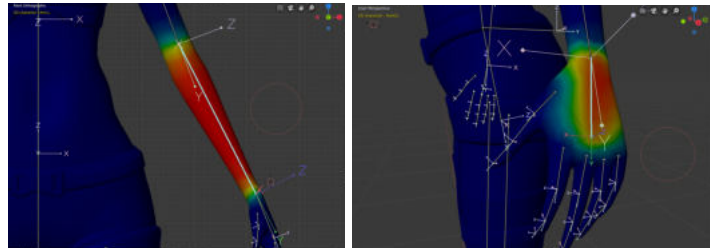
I started painting over my character, there was a lot to correct, and after spending some time and going through all the bones carefully I started getting the result I was expecting. One important thing to mention is that I ended up removing the jacket from the original design of one of the characters, since I was having a lot of trouble adjusting the bones to its irregular shape.

Bone	Weight
Head and neck	
Upper spine and lower spine	

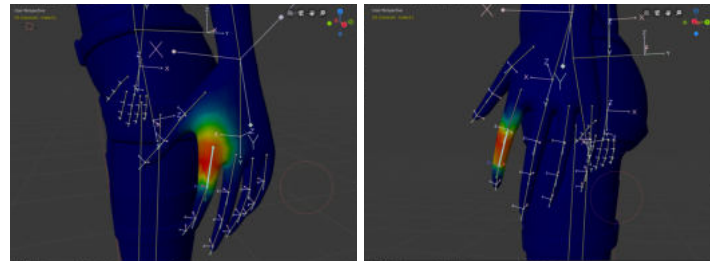
Shoulder and upper arm



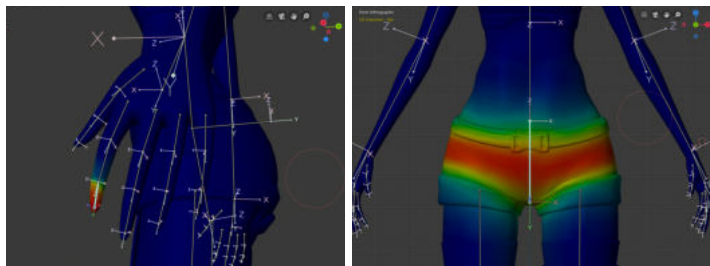
Lower arm and hand



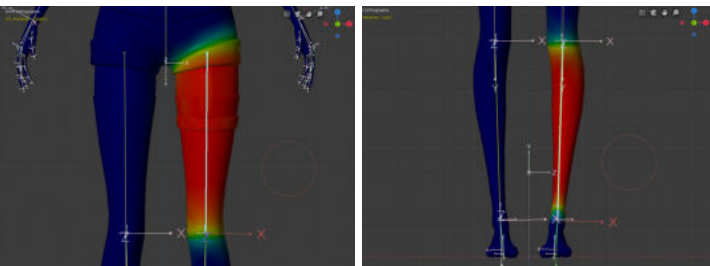
Phalange 1 and phalange 2



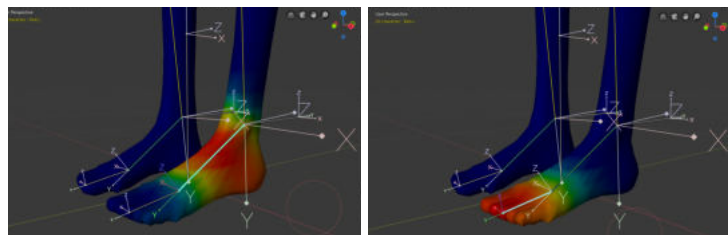
Phalange 3 and hip



Upper leg and lower leg



Foot and foot fingers



Once the weights are properly set the rig is practically complete. However I decided to add a couple more features. The first one is an Inverse Kinematics (IK) system for the legs.

Most animation is produced by rotating the angles of joints in a skeleton to predetermined values. The position of a child joint changes according to the rotation of its parent and so the end point of a chain of joints can be determined from the angles and relative positions of the individual joints it contains. This method of posing a skeleton is known as forward kinematics.

However, it is often useful to look at the task of posing joints from the opposite point of view: given a chosen position in space, work backwards and find a valid way of orienting the joints so that the end point lands at that position. This approach is known as Inverse Kinematics (IK).

Basically, with the current configuration of the armature, which uses forward kinematics, if I wanted to move all the bones in the leg, I would have to manually change the position of the upper leg, the lower leg and the foot. IK, however, makes it possible to, by just moving the end of the chain, in this case, the foot, calculate the correct position for the rest of the bones of the chain based on the new position of the foot.

The process to make a leg IK is as follows: First, select the joint on the knee and extrude a new bone from it, Do the same with the heel joint. Then unparent the two new bones and move the knee one away from the body. Make them control bones, not deform. After that select the lower leg bone and add a bone constraint called Inverse Kinematics. The object this constraint will be affecting is the armature, and the main bone that will be controlling the whole IK system is the new bone we created from the heel. The length of the chain is two since moving that bone will affect itself and two more bones (the lower and the upper leg). Furthermore, I added a pole target, the bone I created from the knee and separated from the body. This will make the whole IK chain to point towards that pole, meaning by just moving that separated bone, the knee and the foot will move too pointing it that bone's direction. Also, to have a better control of the foot, I selected the foot bone and the new heel bone and parented them. Finally, at this point if you move the heel bone downwards, it will end up separating the foot from the rest of the armature. In order to avoid this, I added a bone constraint to the foot bone called copy location targeting the lower joint of the lower leg. That way even if you move the knee IK too far away, the foot will remain attached to the rest of the armature.

To finish with the rigging, I decided to rig the hair. I thought it would be a really interesting feature, and it would make the final model look much better.

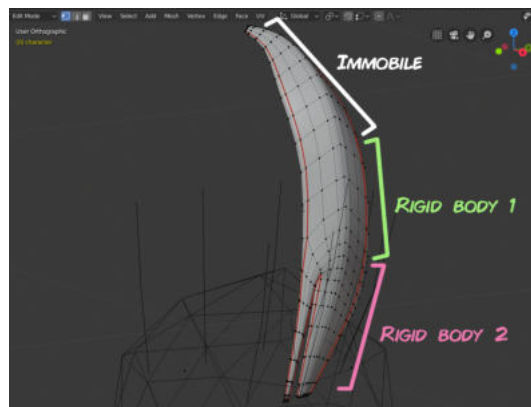
Rigging the hair is quite different from rigging the rest of the body. Of course, you could do it the same way, assigning bones to the hair strands and painting the weights for the bones. But then, once you started doing the animations, for each keyframe you created you would have to manually place each of those bones. That is an absurd amount of work and it can be avoided by using physics.

Blender's physics system allows you to simulate a number of different real-world physical phenomena. Basically, I will add physics to the hair, so that when I move the body of the character, the hair will follow along, affected by forces like gravity and some others.

In order to do so, we will need two tools from the Blender's physics window:

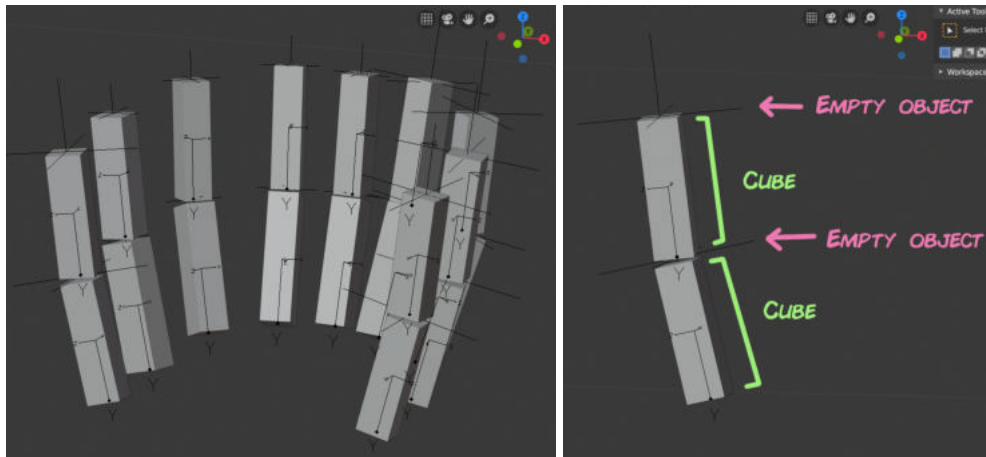
- Rigid Body: The rigid body simulation can be used to simulate the motion of solid objects. It affects the position and orientation of objects and does not deform them. Unlike the other simulations in Blender, the rigid body simulation works closer with the animation system. This means that rigid bodies can be used like regular objects and be part of parent-child relationships, animation constraints and drivers. Only mesh objects can be part of a rigid body simulation. There are two types of rigid bodies: active and passive. Active bodies are dynamically simulated, while passive bodies remain static. Both types can be driven by the animation system when using the Animated option. During the simulation, the rigid body system will override the position and orientation of dynamic rigid body objects.
- Rigid Body generic constraints: Constraints (also known as joints) for rigid bodies connect two rigid bodies. The physics constraints are meant to be attached to an Empty object. The constraint then has fields which can be pointed at the two physics-enabled objects which will be bound by the constraint. The generic constraint has a lot of available parameters. We will use it to limit the translation and rotation of the rigid body along the X, Y and Z axis.

Now that we know the theory, it is time to apply it. We are going to think about each hair strand as a chain of parented rigid bodies, each of them will move on their own affected by the external forces and will also follow their parent's movement. I chose to use two rigid bodies for each strand since the hair of the two of my characters is very short. Also, rigid bodies will not start affecting the hair right from the base of the head. There will be a certain margin and the base of the hair closer to the head will remain immobile. To put it graphically, in one strand of hair we will have:



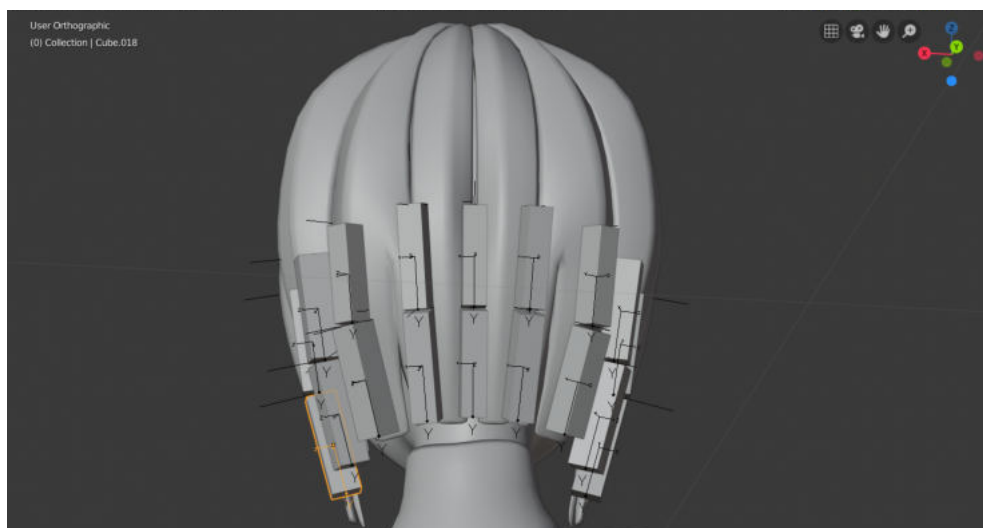
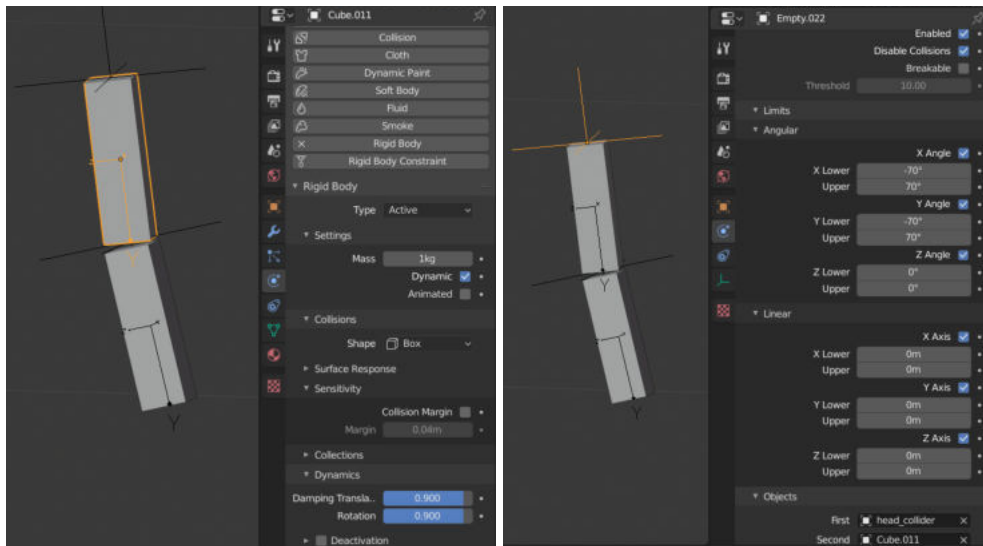
After some research, what it seemed to me as the simplest approach for this was to create two cubes adjusted to the two desired rigid bodies on the previous picture, and then add the physics rigidbody tool to those cubes; add two empty objects on top of each cube, to act as the rigid body constraints; add two bones of the exact same size and placed on the same position as the cube, parent all the system together, the constraints with the rigid bodies, and then with the bones, and finally parent all the bones to the head and paint their weight over the hair strands.

So, I started by creating the cubes and the empty objects and placing them all properly adjusted to the position of each hair strand.

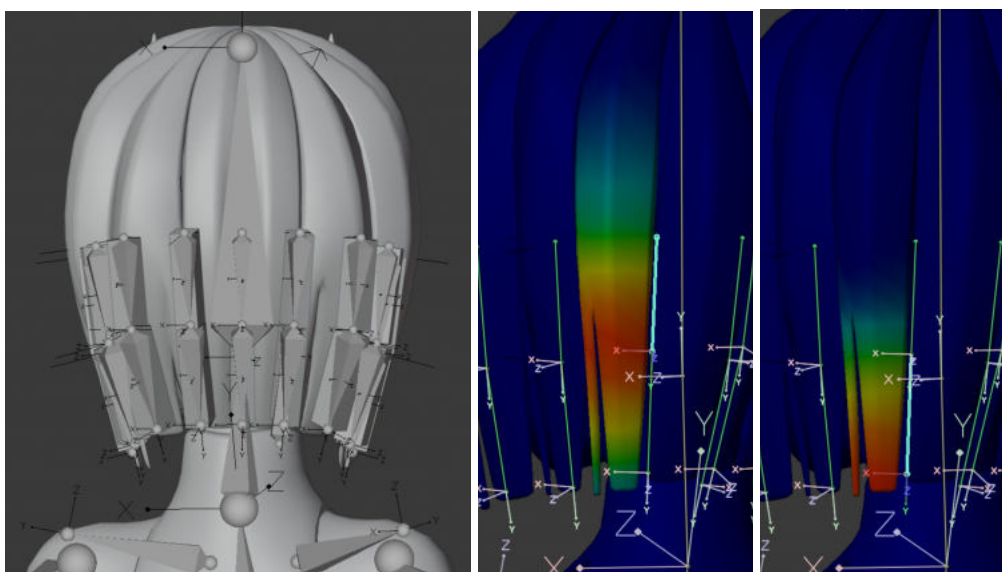


Once they were all properly placed I parented them properly: the topmost empty object is the father of the whole chain, affecting everything below it. It is also the child of the head. The upper cube is the father of the lower empty object and the lower cube, affecting the two of them. The lower empty object is the father of the lower cube, affecting it only.

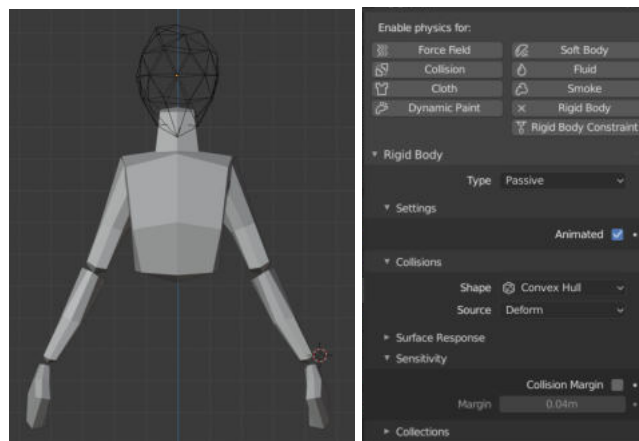
After that I added the physical properties to each of the objects of the chain: the two cubes have an active rigid body. Like mentioned before, active rigid bodies are dynamically simulated, and since hair is a dynamic object we need this kind of rigid body. Rigid bodies have many parameters that can be adjusted and will affect the final result of the simulation, like, for example, the mass. I played around with them until I liked the visual result. The two empty objects have generic rigid body constraints mostly to limit the rotation and translation of the rigid bodies. The translation is limited to zero, this makes sense because hair is not elastic, if we are splitting the hair strands into two rigid bodies to control them, it will never happen that half of a hair strand will change its position without the other half changing too. The rotation however, can be changed, except on the Z axis. I limited it on the X and Y axis to a range of  $[-70, 70]$  degrees since I don't want extremely exaggerated rotations but rather soft and smooth ones, and any rotation surpassing 90 degrees would look really weird and deformed, so I thought that was a nice range. Here follow the final configurations of the rigid bodies and the constraints and the whole system assembled on the head of the character.



Now all that's left is to create the bones parented to the head and placed over the rigid bodies and paint their weights over the hair strands.



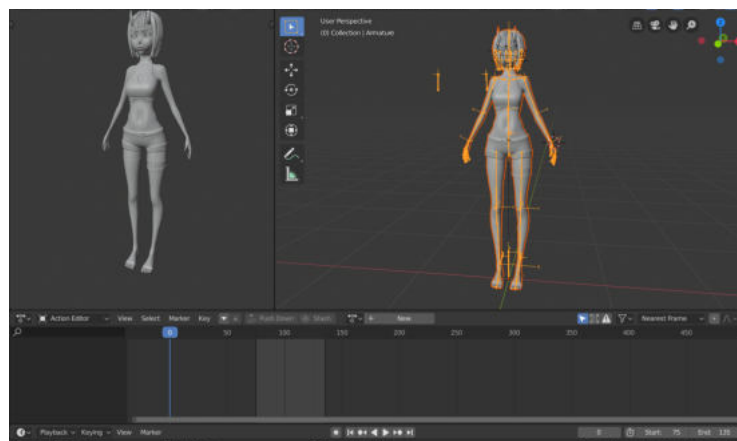
One last really important thing to do is adding colliders to the body. Colliders are objects with rigid bodies that will interact with the hair strands making them collide instead of going through. This way the hair won't go through the head of any part of the body. One simple way to do this would be by adding a rigid body directly to the character, but it would be expensive and unnecessary since it is quite unlikely for the hair to go through the lower half of the body. Instead, I created a few new cubes and spheres and adapted them to the head, neck, arms, hands and chest, and I added a passive rigid body to them. With that, the characters are fully rigged and ready to be animated.



## Animation

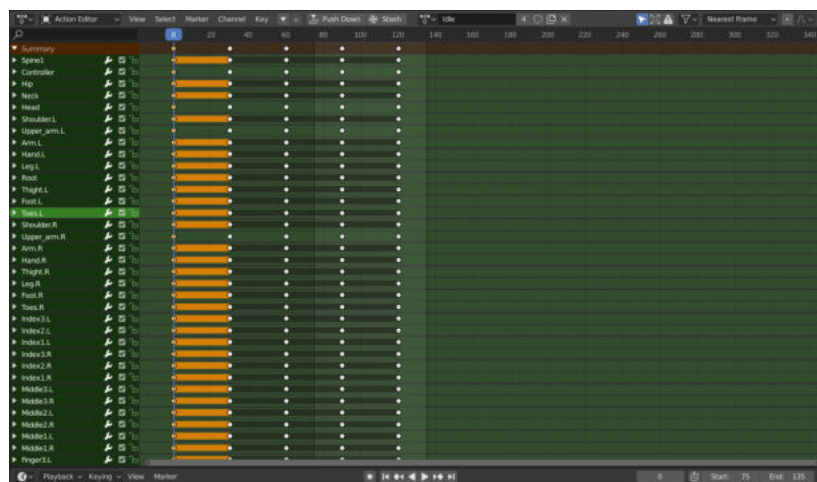
Now that we have our characters rigged, it is time to create the animation clips for all the possible movements of the characters. This will also be made in Blender since it provides an animation window I have previously worked on, and I find it convenient to execute all the steps of the character creation process in one single software, to avoid continuous exports and imports that may somehow affect the model.

The process to create an animation clip is simple: you need to change to Blender's animation window. This window will display your model; a version of the model only with the parts that will be rendered, meaning only the parts that will be visible in the game, in this case, only the mesh, without the armature and the whole physics system; and finally, a timeline showing the frames of an animation.



The “New” button on top of the timeline will create a new action, it is, a new animation clip. Inside each animation clip we will be moving through the frames on the timeline, changing the position of the character on certain frames. Those frames will be called keyframes.

Keyframes can be generated manually or automatically. By enabling the circle button under the timeline, it will automatically create a key on a certain frame whenever a bone is moved on that frame. However, it will only add keys for the bones that were moved, leaving the rest blank. That means that the position of those bones on those frames is not being saved, so when the program interpolates the positions of the character between the keyframes, the position of the bones that did not have a key may be altered, leading to undesired results. That is why, in each of my keyframes I manually added a key for all the bones on the armature, even if I had not moved them. As shown below in a photo of the timeline expanded, the white circles are the keys for the bones, and as you can see, on the keyframes in which I registered the positions of the bones, I did it for all the bones in the armature.


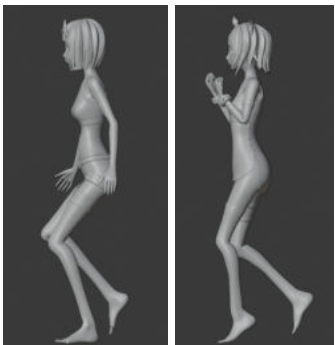


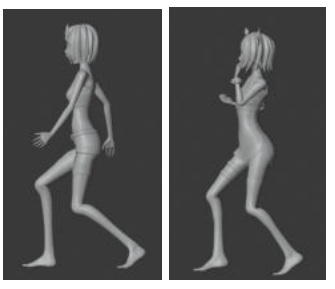




There is really nothing much to mention about the animation process. I will now show some of my animations, showing the keyframes I generated and adding some annotations explaining the main features of each keyframe to make a smooth and fluent animation.

### Walk cycle




Frame	Annotations	Image
0	The left leg is in front of the body and the right one is behind. The hip is slightly down and slightly rotated towards the right. The arms are wide open, the left one is behind the body and the right one is in front of it. The feet aren't fully grounded.	




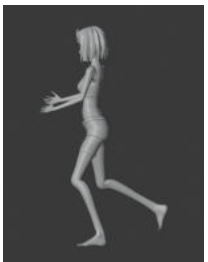


10	<p>The feet are fully grounded, the hip is at its lowest position and completely bent towards the right, the arms are fully extended.</p> <p>Note that the second character is not extending the arms because she walks keeping her fists up. She also isn't grounding her feet completely, because I wanted to make her walking animation look like she's making small jumps.</p>	
20	<p>The right leg starts passing over the left one, the hips start going up and stop bending to the right. The arms start closing up</p>	
30	<p>The right leg passes the left one. The arms are completely aligned. The hip is at its highest position and it is completely straight.</p> <p>Note that the second character is not touching the ground at all. again, because of wanting to create the effect that she is making little jumps while she walks.</p>	
40	<p>From now on, the frames are just inverted versions of the previous ones starting from frame 0. That means, the position of the hip is the same, but its orientation is the opposite. The position of the left arm is the same one as the position of the right arm on frame 0 and vice versa. And the position of the left leg is the same one as the position of the right leg on frame 0 and vice versa.</p>	
50	<p>Inverted frame 10</p>	



60	Inverted frame 20	
70	Inverted frame 30	
80	Copy of frame 0 to properly loop the clip	



#### Kick 1

Frame	Annotations	Image
0	Default pose to smoothly change from the idle animation to any other clip.	
10	Whole torso rotates towards the right. Right knee also rotates towards the right. Left foot moves forwards. Arms get in a sort of guard position, getting ready for the impulse of the kick.	
20	Left foot slightly lifts from the floor and arms are fully closed, getting all the impulse for the kick.	

30	She starts kicking. Left foot lands on the ground again. The whole torso starts rotating towards the left. Right leg lifts from the ground and bends to take an impulse. Arms start opening.	
40	Left foot slightly lifts, standing on tip toes. Torso is fully rotated towards the left. Right leg is fully extended. Arms are lifted.	
50	Torso starts going back to normal, arms and legs get relaxed and start falling down.	
60	Torso is straight, looking forwards. Left leg is fully on the ground. Right leg is again behind the body, going back to its original position. Arms are also going back to their original position.	
70	Copy of frame 0, to properly connect animation clips.	

### Jump

Frame	Annotations	
0	Default pose to smoothly change from the idle animation to any other clip.	
2	Hip goes down, taking impulse for the jump. Arms move backwards and bend, also to help getting impulse.	

5	Right after starting to jump. The whole body stretches after releasing the strength of the impulse. Arms go up as a result of releasing the impulse.	
15	Legs bend back again in the air after full stretching when releasing the impulse. Arms stay up.	
30	A copy of frame 15. Since she will remain in the air for a while, she will keep this position for a brief time before start falling and landing	
43	A copy of frame 5 as she starts falling down.	
46	A copy of frame 2 when she touches the ground and her body falls abruptly because of the gravitational force.	
48	Copy of frame 0, to properly connect animation clips.	

As you can see, animating is just a long process that requires an extremely careful treatment of the keyframes, adjusting over and over again, putting a lot of attention to the small details, as a simple 5 degree rotation of a bone can make the difference between a jumpy and a smooth, clean transition between frames.

## Environment

### Design

My first intention for the environments was to make them 2d background images with maybe a few 3D elements like platforms to jump over them. Besides, I would be doing those images with a 2D drawing software like photoshop.

The game is going to have two maps, and I wanted them to be set in Japan, as it has already been mentioned that this game would be based and inspired by japanese culture. In my first trip to Japan, one of the things that fascinated me the most was how you could be walking on a street full of skyscrapers and neon lights, feeling like you are in the future, and then you walk two streets away and you find yourself in front of a temple surrounded by magnificent gardens. So when I decided to make two maps I was clear about how they would be:

- A map in the middle of a city at night, with some buildings, a lot of artificial lights and a shiny starry sky.
- A map in a quiet place, in a traditional temple at dawn, with lots of vegetation.

The main ideas can be clearly appreciated on the following pictures<sup>1</sup>:



I did one first check picking one of the images I found when searching for ideas and putting it in a scene in Unity, also placing the characters on it to see how it would look and the result was rather weird. It looked pretty unnatural to put 3D characters over a 2D background, it felt like they were just flying with a random image on the background. The main problems were the lack of the shadows the characters should cast, without them, mainly without the floor shadow, they don't really look integrated on the scene. Besides, since I had the idea of making platforms so the characters could jump over them, when doing so, the perspective of the image obviously won't change, when it by logic should, since the camera will move to keep the two characters framed all the time, and that effect also looked pretty strange.



Being that, after some considerations and hesitation, I decided to make the backgrounds 3D as well. Sadly, it was impossible for me at that point to model the backgrounds by myself, as it is an enormous amount of work, and I was already a bit delayed on my original schedule. So I kept the original idea for their design and started looking for free 3D models. Here are the ones I ended up using:

---

<sup>1</sup> Sources for the images:

[image 1](#)

[image 2](#)

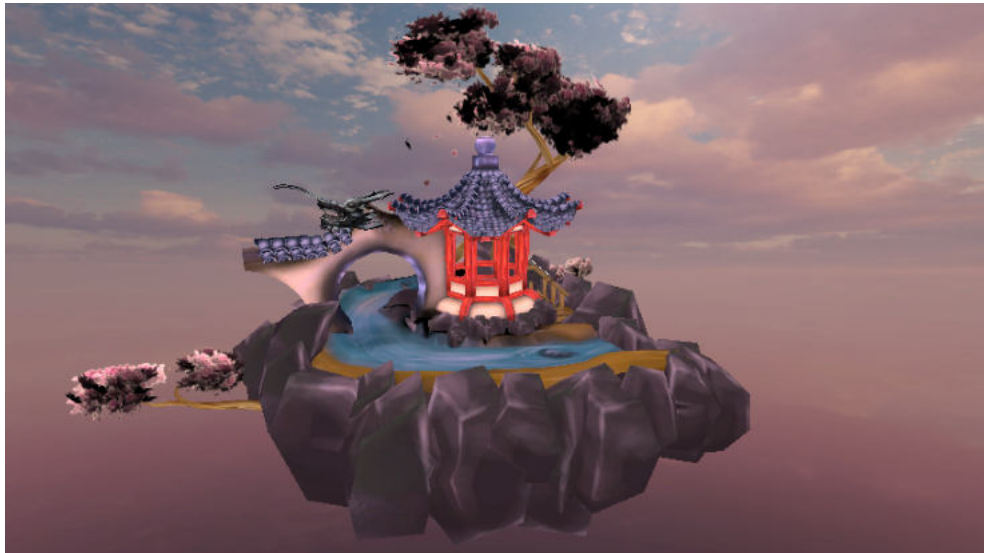
[image used to test](#)



They are free to use and pretty well optimized, with a decently low amount of polygons. However, even though I did not do the models, I had to dedicate some work to them. What you see in the pictures above are high quality renders probably made with a dedicated software that allows for extremely high quality shaders, materials and lights. Unity is not a software specialized on lighting, texturing and creating materials, the options it offers for those areas are rather limited, so when I exported the models to unity, all the lights did not get exported and the materials were converted to Unity standard materials, which didn't look too good. So I had to redo all the lights and materials by myself, which took some time since it was hard to find proper shaders and materials, and configure them, so that the models were displayed the way I wanted.

I also had to look for the texture of the background, the sky. I picked an animated starry night sky for the urban map and a pink dawn sky for the temple.

After assembling them in Unity this is how they ended up looking like:



## Menus and HUD

### Main Menu

For the design of the menus I kept the Japanese aesthetic, using elements of traditional Japanese ink drawings, like clouds or fire. I posed my characters in a cool looking pose, placed them over a cloudy sky ink background and added some smoke effects<sup>2</sup>.

I then designed a logo for the game. I put the title of the game in a Japanese style typography, and added a slight 3D style and a black border. I placed it over a red circle and added some ink clouds. I also added the silhouettes of my characters in a fighting pose over the clouds.

---

<sup>2</sup> Source of the images used to create the main screen:

[background](#)

[golden clouds](#)

[red smoke](#)

Finally, I added the buttons, which I designed drawing a red geometric border and a slightly transparent red background. I used a japanese-like font for the texts. There are three buttons: one to start a new game, one to go to the controls screen and one to exit the game. There is a script in the scene called manageScenes, attached to an empty GameObject, which consists of several short functions each of which change to a determinate scene (to the main scene, to the character selection menu, to the controls scene and to close the game). I called the proper functions to change to the proper scenes from each of the buttons.



### Controls menu

As its name says, the controls menu is a screen showing all the controls of the game. As it is obvious, it will also be designed following the line of the japanese and ink-like elements.

It has a red background with golden japanese ink waves, an image of a keyboard, ink crosses in different colors to mark the used keys over the keyboard and a box with a similar design as the buttons used in the main menu but this time in gold, same geometric border and semi-transparent background, and from now on in each of the screens, filled with the same ink crosses as the ones in the keyboard to reference them, and accompanied by a text explaining what each of the keys do<sup>3</sup>.

Finally, there's a back button to go back to the main screen. Like the rest of the buttons we've created until now, it will call a function of the manageScenes script that changes to the main menu scene.

---

<sup>3</sup> Images used to create the controls menu:

[background](#)  
[keyboard](#)  
[ink cross](#)

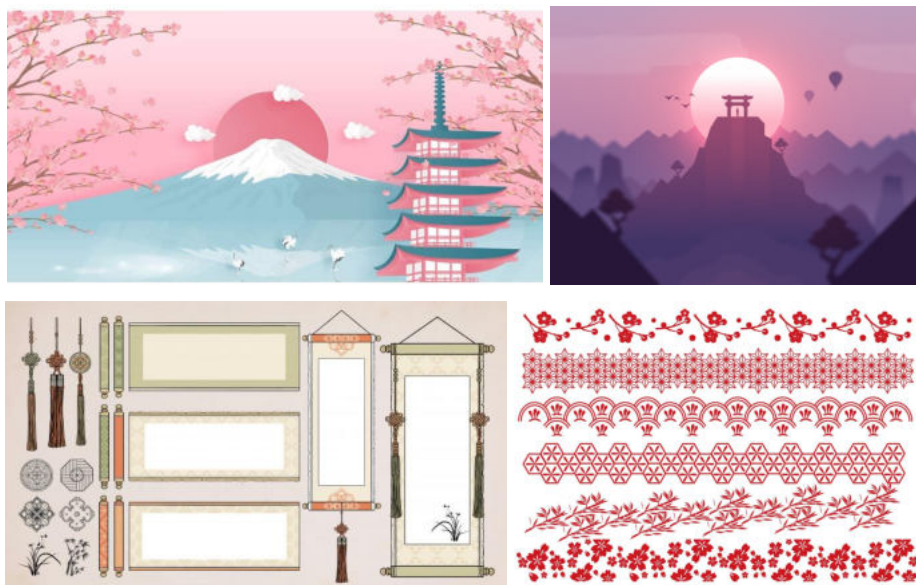




### Character selection menu

For the design of this menu, I decided to go with a design that included some Japanese traditional items. I had a blurry image in my head about how I wanted it to look: I wanted the characters to stand over a background with a soft drawing of some Japanese landscape like a temple or a garden. Also, showing on one side of the screen, the list of available characters. For that I thought it would look good to put the characters inside a scroll, like the ones they formerly used to write letters and documents.

I started searching for the elements to assemble the idea in my head, and I selected a few ones that I thought could look good together.



And I did one first design that I liked enough and I thought I would use it as the final design. I included it on the project and worked with it for a while. But after plenty of time looking at it, I knew there was something that did not completely convince me. It was too dark and the division between the section with all the characters and the one displaying the selected character looked very weird. So I ended up changing it for a new version with lighter colors and in which the sections are not separated but integrated together instead. Here you can see the first design and the final one<sup>4</sup>:



All that was left for the design part was to add the buttons displaying the different characters and the button to start playing. I designed those manually with Photoshop, drawing a geometric border, filled with an 80% opaque red background, and, for the character ones, a picture of the face of the character, and for the one to start playing, a “Start” text using a free to use japanese looking font.



---

<sup>4</sup>Sources for the images used:

[background1](#)

[background2](#)

[wall scrolls](#)

[floral borders](#)

Now that we are over with the design part, it is time to start coding the logic of the menu. The idea is quite simple: pressing any of the buttons of the characters will display the adequate character in the middle of the screen. Pressing the play button will go to the next screen saving the currently displayed character as the selected character.

In order to do so, we will start by adding the models of the characters to the Unity scene, disabling all of them except for the first one on the list, which will be visible as soon as we enter the menu.

We will then add an empty object to the scene and will attach a c# script to that object. The script could be attached to any object of the scene actually, but creating an empty object named "MenuManager" and making it hold the management script seems more organized and clearer.

Inside the script we will have many components. We will start by defining the variables: we will need an array of GameObjects which will contain all the characters on the scene; an integer pointing at the currently selected GameObject of the array; a GameObject representing the currently selected character and an Animator component.

We will then define the Start method, which will only initialise the currently selected character index to zero and the currently selected character GameObject to the one with that index inside the array. That means that, right when entering this menu, the first character of the list, which is in fact the one that we kept enabled, will be selected by default. This way, if the user pressed the play button without having set the character by himself before, the first character would be automatically assigned and there would be no error.

After that, we will create a method called ChangeCharacter. This method receives an integer and basically what it does is to disable all the characters in the array except the one with the passed index, which is then set as the currently selected character.

Now to understand what follows, a bit of a more concrete explanation about the menu is necessary. My idea was not to simply select the character and immediately exit the menu. I wanted to do something a bit more elaborate that made the menu way cooler. So my idea was: when pressing the character buttons, the character displaying in the middle of the screen would be playing some idle looped animation. Then, once the character is selected and the user presses the play button, the selected character plays a characteristic animation while the camera starts zooming into its face. Once the animation is finished, the game changes to the new screen. This is carried out in our script this way:

The function PlayAnimation gets the animator of the selected character and changes its state so that the idle animation, played by default, stops and the special one starts playing. It then calls two Coroutines; the first one, ZoomCamera, changes the position of the main camera by, giving an initial position, a desired position and a time period, interpolating the camera's current position during that time period. Finally, after two seconds, the time that the zoom in lasts, it calls the Coroutine GameStart, which changes to the next scene and saves the index of the selected character so that the in-game scene can recover that information to use it to load the proper character for the player.

One last thing to do so that all we did actually works is to call the described functions so that they actually execute. The proper way to do this is: calling the ChangeCharacter passing the index 0 when pressing the button of the first character and calling this same function passing the index 1 when pressing the button of the second character. If more characters were added we would just need to keep the same process incrementing the index by one with each character. Finally, we call the function PlayAnimation when we press the start button. This one automatically executes the other two functions so we don't need to call them.

And finally, a couple of pictures of the final menu with a character displayed and the menu right before changing the screen, after doing the zoom in and playing the animation.



### Map selection menu

The design for this one is quite more simple than the character selection one. It was not necessary to create a background and elaborate a layout with various elements, since the map itself was going to be the background. And just like in the previous menus, there will be buttons with images of each map, a button to go to the next screen and a button to go back to the character selection menu. The buttons will keep the same design line, having the same red geometric border.



And the whole menu assembled:

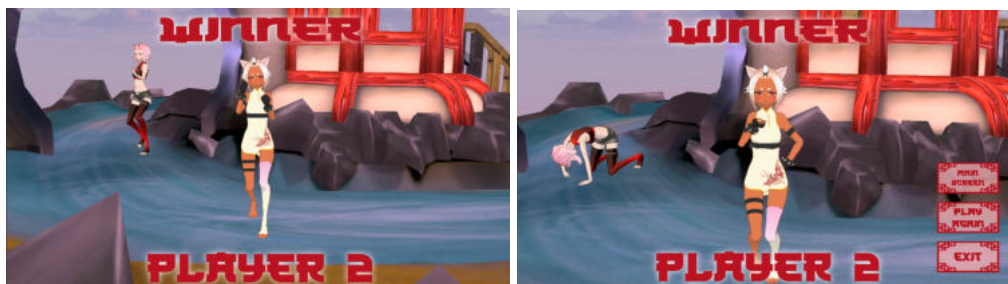


About the functioning of the menu, there is not much to say other than that it is practically identical to the character selection menu. The only difference is that there is no animation to play after pressing the start button, it just loads the next screen automatically, so the ZoomCamera and the GameStart Coroutines are not necessary, and the PlayAnimation function, now becomes the GameStart function and what it does is storing the information about the selected map to again, be able to use it in other scenes that need to know that information; and load the proper game screen depending on the selected map.

Like on the characters menu, the first map will be displayed and selected by default, the function ChangeMap function is called passing index 0 when pressing the first map button and passing index 1 when pressing the second map button, and the GameStart function is called when pressing the button start.

### End game menu

The end game screen is the one that will be loaded once a game is finished. It is a simple screen with a text displaying the winner, the two characters inside of the map in which the game was played, one of them in the center of the screen playing a cocky winning animation and the other one in the back playing a sad losing animation. There is a slight movement of the camera towards the winning character. After the movement, three buttons are displayed; one to play again, which will lead to the character selection menu; one to go back to the main screen; and the last one to close the game. Here are a couple of pictures of the screen right when it loads and after the translation of the camera.



As an annotation, there is not just one end game screen, there are several of them covering all the possible combinations of the winning character and the map. At the end of a game, that information is gathered up and it is used to load the corresponding end game screen. Here are all the possible options:

- Map 1, Yura win, Tamachi lose
- Map 1, Tamachi win, Yura lose
- Map 1, Yura win, Yura lose
- Map 1, Tamachi win, tamachi lose
- Map 2, Yura win, Tamachi lose
- Map 2, Tamachi win, Yura lose
- Map 2, Yura win, Yura lose
- Map 2, Tamachi win, tamachi lose

The information about which player, not character, won is also stored and recovered on the end game screen to properly set the text with the winner.

There is not much code on this screen. It has one single script with a Start method that recovers the information about what player won and sets the winner text properly. After that it starts the Coroutine ZoomCamera, which moves the camera towards the winner during two seconds. After those seconds, the Coroutine GameReset starts. This coroutine simply makes the three buttons visible.

There are also three methods that will execute when each of the three buttons are pressed: ToMain loads the main screen when the button “Main screen” is pressed; Restart loads the character selection screen when the button “Play again” is pressed and Exit closes the application when the “Exit” button is pressed.

## HUD

The HUD of my game is quite simple, it is simply a display of the status of the characters during a game, showing their face, their HP, the amount of charge for a long range attack and the amount of charge for the ultimate attack (those two can only be executed when their bar is full). The hp is a circular bar surrounding the face of the character, and the long range and special attack bars are simple straight bars at the bottom of the circle, pointing towards the right for the player one and towards the left for the second one. Finally, the three bars change their color depending on the fill amount; the hp bar goes from turquoise when full to red/orange when empty; the long range attack one goes from cyan when full to dark blue when empty and the special attack goes from pink when full to yellow when empty.

That change in the colors is managed by a function that, given the full and the empty color, interpolates the current color based on the current amount of hp.



## Game scene

In this section we will go through all the elements of the game scene, starting with the general configurations of the scene and then focusing on the concrete settings and coding of the players. Even though we are talking about one game scene, there are actually two of them, one for each of the maps, but the two of them work exactly the same way, the only difference is the model used for the map.

## Loading the characters

The most important and logical thing to do at first is obviously to load the proper characters (and their HUD) in the level. Characters are not static, they are not going to be the same each time we load the level, they will change based on what the characters the players selected in the selection menu.

So the way we will be doing it is quite simple: we will keep all the possible characters and HUD elements in the scene, properly placed and hidden, and depending on which one each player chose, we will display the proper ones and destroy the rest.

To do that we will need a script, attached to an object on the scene, we will call this object "GameManager". The script will receive four arrays of GameObject, to store the characters of player 1, the characters of player 2, the HUD elements of the characters of player 1 and the HUD elements of the characters of player 2. The way the characters and HUD elements are arranged inside the arrays is as follows:

- Element 0: Yura model/HUD
- Element 1: Tamachi model/HUD

Knowing this, we will do the following in the Start function of our script: when talking about the menus, we said that "we would be saving the information about the selected character to recover it and use it in another scene". Unity provides the class PlayerPrefs, that stores player preferences between game sessions. It can store string, float and integer values into the user's platform registry. It means that we are able to store two integer values referring to the selected characters in the character selection menu and those values will still exist in the game scene. This way, the function recovers those values and makes the proper characters visible and destroys the rest. The integers saved in the character selection menu were 0 for Yura and 1 for Tamachi, so for those values can be directly used to get the elements of the array with that index: if the index is 0, the model of Yura and her HUD elements are made visible and the model of Tamachi is destroyed and vice versa when the index is 1. This is applied to load the characters of both players.

## Camera

For the game camera, a basic approach would be to create a camera on the scene, point it towards the map and make the whole map fit inside. But this way we will always be seeing the whole map, this makes sense for when the characters are far from each other, but they would always look pretty small.

So a more sophisticated solution is to have a camera pointing to the two characters, keeping both of them always on screen, zooming out when they are far from each other and zooming it when they get close.

At first I tried to do it creating a script for the camera that received a list of targets and kept the camera always looking towards those targets, but I did not get it to work 100% right because the boundaries of the camera were too imprecise. So I ended up using Cinemachine. Cinemachine is a suite of modules for operating the Unity camera. Cinemachine solves the complex mathematics and

logic of tracking targets, composing, blending, and cutting between shots. It is designed to significantly reduce the number of time-consuming manual manipulations and script revisions that take place during development.

This way, you add a Cinemachine component to your camera and instead of coding the behaviour of the camera, you are provided with an interface that lets you set this behaviour by simply adjusting parameters and options. The most important parameters I adjusted are: The Follow and Look At parameters which receive the GameObject or group of GameObjects that the camera will follow and look at. I set those GameObjects to be the characters in the scene. Other than that I just played around with all the settings, adjusting the borders, the offsets on the axis and the maximum and minimum zoom until I was satisfied with the result, making sure that the characters were fully visible anywhere in the scene, without getting cut anywhere.



### Player mechanics

Here we will see all the possible movements that the characters can do and the way they're implemented

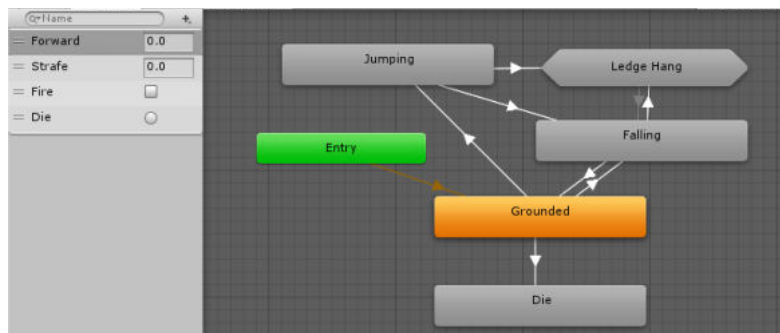
Name	Description	Trigger	Notes
Move	Change the position of the player along the X axis both when standing on the ground and in the air.	Pressing left and right arrow	Can move when jumping. Can't move while attacking, covering, receiving damage, firing and ulting.
Jump	Change the position of the player along the Y axis.	Pressing the jump button	Can't jump while attacking, covering, receiving damage, firing and ulting
Attack	Hit the enemy player, reducing their health if the hit is successful. Attacks can concatenate up to a three hit combo.	Pressing the attack button	Can't attack while covering, receiving damage, firing, ulting and jumping.
Cover	Cover yourself with a magical shield that mitigates all the damage from basic attacks	Pressing the cover button	Can't cover while receiving damage, attacking, jumping,



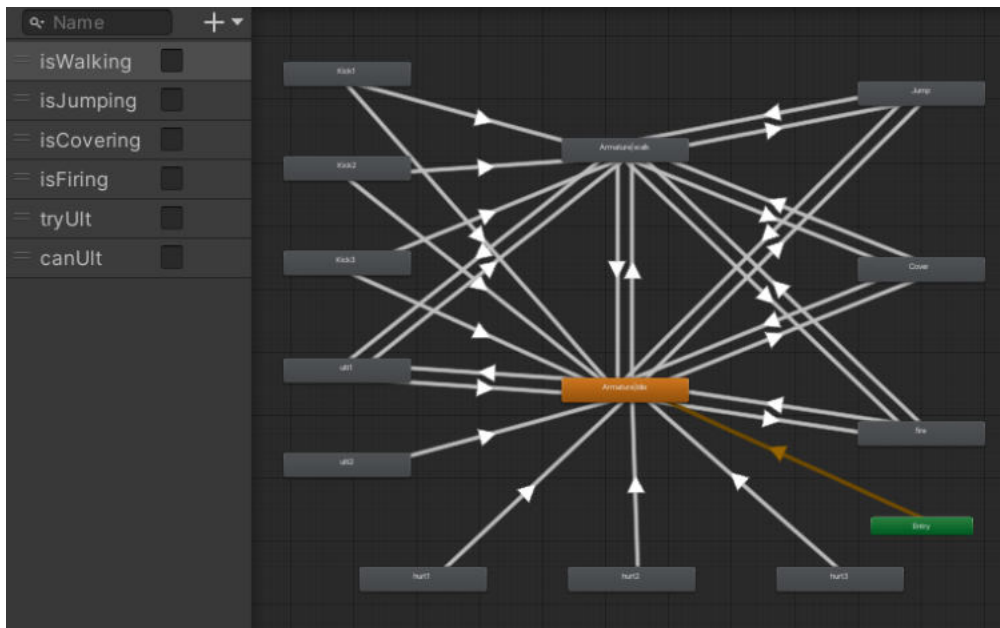
			firing and ulting.
Fire	Shot a long range attack that can only be fired when a charge fire bar is full, dealing damage. If the enemy is covering only half of the damage is inflicted. The bar charges slowly over time and when hitting the enemy. After firing, the bar empties.	Pressing the fire button	Can't fire while jumping, attacking, covering, ulting and receiving damage.
Ultimate	A short range attack that can only be executed when a charge ultimate bar is full. If it hits, it triggers a special animation and deals a high amount of damage. The bar charges slowly over time and when hitting the enemy. After ulting, the bar empties even if it did not hit.	Pressing the ultimate button	Can't ult while jumping, attacking, covering, firing and receiving damage.
Receiving damage	A fraction of the player's hp is taken and they are unable to move for a brief time.	Getting hit by any kind of attack	Disabled time is shorter for the first two basic attacks of the combo and longer for the rest.

Before starting coding the mechanics, we need to have the animations of our mechanics organised somehow, with connections between the mechanics that can directly change form one to another, for example, the idle animation should be connected to the walk animation since you can change to a walking state from a standing state.

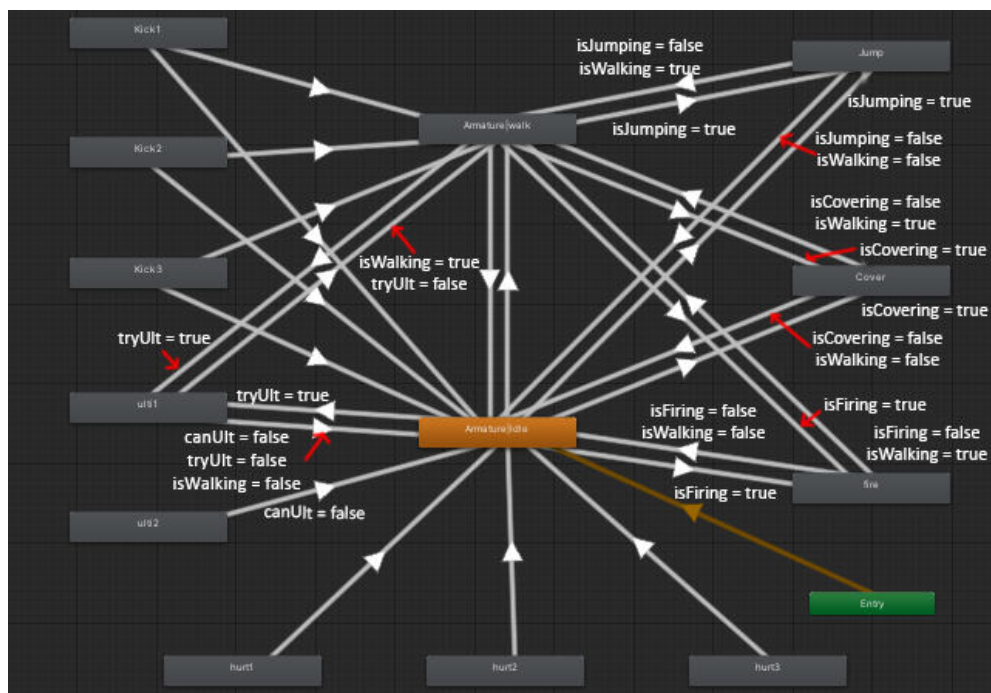
To do that we will use an Animator Controller. An Animator Controller allows you to arrange and maintain a set of Animation Clips and associated Animation Transitions for a character or object. The Animator Controller has references to the Animation clips used within it, and manages the various Animation Clips and the Transitions between them using a State Machine, which could be thought of as a flow-chart of Animation Clips and Transitions, or a simple program written in a visual programming language within Unity.



Here we can see an example of an animator controller. The boxes are the animations, the yellow one being the one that will play at first. The arrows are the transitions between animations. Those transitions can receive conditions. If there are no conditions, the transitions are executed automatically. If there are conditions, the transitions will only execute when the conditions are met. Those conditions can be set with the parameters on the left. In this case we will only use Bool parameters. We will set the bools to true or false by code, executing the transitions between animations right when we want them to execute.



and the conditions for my transitions:



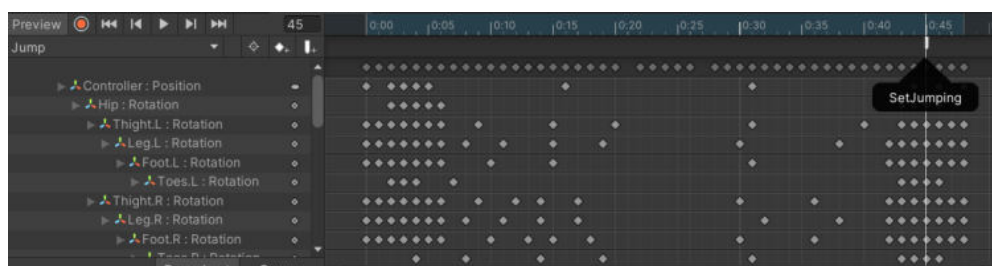
Finally, another important thing to explain, since we will be using it a lot, are animation events. Animation Events allow you to call functions in the object's script at specified points in the timeline. This means, in the script that controls the character we will have many functions that will be called from the animation clips of the character, at the specified frame.

Now it is time to start coding. We will have the main behaviour of the character on a script called CharController, attached to the character. This script is mainly composed of an Update function to constantly check for the player's inputs and the functions to call from the animations.

First of all, I did the move and jump mechanics. For the two of them it is important to know if the character is grounded or not, so we will only walk and jump when we are on the ground. To do that we will create an empty object attached to the feet of the character. From the script we will check if from the center of that object, to a certain radius we can set (0.6 in this case), that imaginary sphere collides with a specific layer, in this case the "environment" layer, to which belongs the model of the map, this way basically if the feet are at a distance smaller than 0.6 from the floor, thack check will return true. We will assign that return value to a "grounded" bool.

For the movement, I set a max walk speed and I multiplied it by the current input on the left and right arrows of the keyboard. I then set that value for the velocity of the character on the X axis. If that value is not zero, meaning there is input on the keys, and the player is grounded and not jumping, we turn the isWalking bool to true thus changing to the walk animation. Moreover, we will have to change the orientation of the character depending on the value of the speed. We will do it by changing the scale along the Z axis of the character if it is not ulting, not covering, not attacking and not firing, as it would not make much sense to suddenly change the orientation in the middle of any of them. Having checked that, if the speed is less than zero, we will make the scale negative, and if it is more, we will make it positive.

And for the jumping, we will check if the character is on the ground, and if it is not ulting, attacking, covering or firing. If all conditions are met, we will set the bool isJumping to true to change the state of the animator controller and the bool jumping to true to manage the functioning of the code. We will also define a value for the character's velocity vertical velocity. I picked a value of 6 because I liked the visual result of it. However, if those conditions are not true but the character is grounded and not jumping, we will set the isJumping bool to false and will make the velocity of the player on the Y axis equal to zero. Finally while we are not grounded, the velocity on the Y axis will be constantly updated. For all this to work properly, we still need one more thing, we need to turn the jumping bool to false again at the end of the jump, for that we will call a function SetJumping() from one the last frames of the jump animation clip.



Now we are going to go through the attack mechanic. Attacking can be done as a single attack or as a combo: if we press the attack button one, we do a single attack, but if we press the attack button again when we are in any frame from the middle of the first attack animation to the end, we concatenate the attack with another one. In this new attack, again, if we don't press the attack button the attack ends, but if we do press it again in any frame from the middle of the second attack animation to the end, we concatenate it with a third and last attack. After this one the combo ends and if the button is pressed again the attack will start again from the first clip.

This behaviour is mostly going to be managed by animation events. The update of our CarController will only check if the attack button is pressed and if the character is not jumping, covering, firing or ulting. If so, it will call a function called Attack to start managing the combo.

This attack function will check at which of the combo steps, the attacks, we are. If we are on the step 0, that means we have just pressed the attack button for the first time to start attacking, then the function will tell the animator controller to play the animation of the first attack, and will increase the combo step by one. And if we are not at the first step, meaning we are pressing the attack button when an attack animation is already playing, it will check if we pressed it in a valid time. Like we said, to do the combo we have to press the attack button at some point between the middle of the current attack animation and the end of this one. It could have done that if you pressed the attack button again at any point during the current attack, the combo was triggered, but this way the time the user has to make the combo is shorter, adding a bit of difficulty. So, if we pressed it when the bool comboPossible is true, we will immediately turn that bool to false, to avoid error if the user presses it more than once, and we will increase the combo step.

There are still a couple of things left to do. A function called ComboPossible will turn that mentioned comboPossible bool to true. The same way, a function called ComboRested will turn that same bool to false and will set the current combo step back to 0. Finally, for now we have just played the first attack, we still need to play the other two. The function Combo will play the second attack if the current combo step is 2 and the last one if it is 3.

Another important thing that we will need not only for the attack mechanic but also for the cover, fire and ultimate ones is to make sure that the character does not move while doing them. I decided to make a function to adjust the constraints of the character. The constraints control the movement and rotation of an object along the three axes. I set the default constraints for my characters so they have the rotation blocked around the three axes and the movement around the Z axis blocked too. I then created a function SetFreezePos to block the movement of the character around the three axis and a function SetConstraints to bring back my default constraints.

We also need to manage a bool called kicking that will be used inside the code to manage all the behaviours. A function called SetKicking will turn it to true and another one called SetNotKicking will turn it to false.

Finally, when attacking, there will be colliders attached to the parts of the body that do the attack (legs or arms). We will use them to check if the player hit the other player with their attack. Those colliders can not be always active, if they were, simply by getting close to the enemy we may hurt

them. We need to activate the proper one during the proper attack. We will do it with a function called `activateHit` that will receive an integer and use it to activate the collider with that index inside an array of colliders that we need to create and set at the beginning of our code.

Now, except for the `Attack` function, the rest have been created but not yet called. As mentioned before, they will be called from the animation clips the following way:

#### Attack1

- Frame 0: function `SetFreezePos` to block the movement of the player and function `SetKicking` to turn the kicking bool to true.
- Frame 35 (middle of the animation): `ComboPossible` to turn the `comboPossible` bool to true and `activateHit(0)` to activate the collider of the first attack. The collider is not activated at the beginning of the animation because, if the attack is a punch, for example, if we activated at first, it may look like the other player got attacked when the player has not even raised the arm, which looks weird, so I simply adjusted the activation to the frames where it look visually coherent.
- Frame 55: `deactivateHit(0)` to deactivate the hit collider.
- Frame 65: function `Combo` to play the next attack animation clip if we pressed the attack button during the combo possible time. We call this function close to the end of the animation so that the transition between clips does not look too harsh. Also, if we switch to another clip here, the last frames of the current one won't be played, which is exactly what we want because the last frame of the attack animations is used to end the combo and this only has to be reached if the next step of the combo was not triggered or if there is no next step.
- Frame 70 (last frame of the clip): reached only if the combo does not continue. Ends the attack state. It calls `SetConstraints` to bring back the default constraints, `SetNotKicking` to turn the kicking bool to false and `ComboReset` to restart the attack behaviour.

Attack 2 works exactly like attack 1, adjusting the functions to its frames, having:

- Frame 0: `SetFreezePos` and `SetKicking`.
- Frame 13: `activateHit(1)`, this time activating the collider of the second attack
- Frame 20 (middle of the animation): `ComboPossible`.
- Frame 38: `deactivateHit(1)` and `Combo`
- Frame 41 (last one): `SetConstraints`, `SetNotKicking` and `ComboReset`.

Attack 3 works a bit different, as it is the last attack of the combo so it does not need the functions ComboPossible and Combo, it just plays until the end and resets the combo all the times:

- Frame 0: SetKicking and SetFreezePos
- Frame 18: activateHit(2) to activate the collider of the last attack.
- Frame 34 (last one): deactivateHit(2), ComboReset, SetConstraints and SetNotKicking.

Next we will talk about the cover mechanic. It will cover the player with a shield to prevent them from getting damaged. This one is a bit different from the ones that work by simply pressing a button. This one requires the player to hold down the button for as long as they want the shield to last. The main difference is that now we cannot simply turn the bool covering to false, reset the constraints of the movement and hide the shield at the end of the animation clip, because what we have now is that the covering clip will play and if it reaches its last frame it will stay there until the button is no longer held down. So we need to do all those things when the button is not pressed. This way, again in the Update of our CharController, we will check if the cover button is down. If it is down and the character is not attacking, firing, jumping or ulting, we call the function SetFreezePosition to block the movement of the character, as it has to stand still when covering, we will set the covering bool to true and the isCovering bool to true to manage the script and change the animator controller state and we will make the shield, a hidden GameObject attached to the player, visible. And if the button is not being pressed, if the player is not attacking, ulting or firing, as those also need the movement to be blocked, we will call the function SetConstraints to bring them back to default, we will hide the shield and we will turn the bool covering and IsCovering back to false.



The fire one works pretty much like the attack when doing just one single attack, not a combo. And just as the shield, it has a hidden GameObject that needs to be shown and hidden back. So from the Update, we will check if the fire button was pressed and if the character is not attacking, covering, jumping or ulting. If those conditions are met, we will check if our fire bar is full. If it is not nothing will happen. If it is full, we will call SetFreezePos to block the movement of the character, as it fires standing still, we will set the bools firing and isFiring to true to manage the script and the animation controller and we will empty the fire charge bar. Finally, at the animation clip, we will call the functions ActivateFire, that will make the shot object visible, orientating it properly depending on which character we are playing and the orientation of said character at the moment of the shot. A

few frames later we will activate the collider of the fire, the one that will check if it has collided with the enemy and take some of their hp. We do this a few frames later because if we enable it right when we make the shot visible, if the enemy is far, it will detect the collision a bit earlier than the shot visually reaching the enemy. Finally, at the end of the clip we will deactivate the collider and hide the shot object with the Deactivate fire function, set the firing and isFiring bools to false with the EndFiring function and reset the constraints of the character with the SetConstraints function.



Now, for the ultimate, the logic is pretty much the same as the firing one: we will have a charge bar and one it is full we will be able to execute the ultimate. In this case though, since it is the strongest attack in the game, we will do something extra to make it look cooler. We will add a special ultimate animation in which the character gets ready to fire the ultimate with a new camera that focuses on the character and has a custom layout. So the whole process will work as follows: once the bar is charged, if we press the ultimate button we will do a very short range hit. If we miss that hit, the ultimate is not triggered and the bar empties. If we do hit it, we trigger the special animation and after that we release the ultimate, which is an unavoidable wide attack that deals high damage.

To code that process, as usual we will check from our update if the ulting button was pressed, if it was, and if our ultimate charge bar was full, we will turn the ulting bool to true, block the character's movement with SetFreezePos, empty the ultimate bar and turn the tryUlt bool to true to change that state of the animator controller so it plays the short range attack of the first step of the ultimate.

From that animation we will need to call some functions:

- Frame 4: activateHit(3) to activate the collider of the short range hit of the ultimate.
- Frame 14 deactivateHit(3) to deactivate the collider of the short range hit of the ultimate.
- Frame 20: SetConstraints to reset the constraints of the character and TryUltEnd to change the value of the ulting and tryUlt bools to false.

Note that if the short range attack hits, the functions on the frame 14 and 20 won't execute, as the animation will change to the special one.

We will need a new script in our collider object to check if the attack hits. This script will use a function called `OnTriggerEnter`, a kind of function that executes any time a collider with the property "isTrigger" collides with an object with a non-trigger collider. In this case the trigger collider is the one set in whatever part of the body of the character that is doing the attack and the non-trigger one is a collider covering the body of each character. So the script will check if the trigger collider hit another collider. It will check if that collider is the one of the other player by checking the tag of the object containing that collider. In Unity you can tag objects to then use those tags in the code to find out what object we are dealing with. I tagged one player with the tag "player" and the other one with the tag "enemy". So the collider of the one tagged as "player" will check if it collided with an object tagged as "enemy" and vice versa. If it did, it will deactivate itself, as, like we said, the functions on the frame 14 and 20 of the attack animations won't execute, so the attack collider does not deactivate and it has to for obvious reasons. It will also disable the `CharController` script of the object it hit, because once that attack hits, we don't want the enemy being able to just walk out while the special animation is being played, dodging the damage of the ultimate. If this attack is hit, the other player won't be able to move until all the ultimate is done. Finally, it will change the state of the animator controller so it plays the second animation of the ultimate.

This second animation will have two steps: the first one will be to play the special clip with the special camera and layout and the second one will be to release the actual ultimate attack. We will need to call various functions of our `CharController` to manage all the behaviours:

- Frame 0: we will call a function named `EnableMyCam`, this function takes the special camera and makes it visible, making it the new main camera of the scene.
- Frame 161 (middle of the animation): we will call the function `DisableMyCam` to hide the special camera and bring the main camera back to normal, and the function `EnableUltEffects`, that will make the special effects for the ultimate visible.
- Frame 205: `activateHit(4)` to activate the collider of the ultimate, again, later that the activation of the effects so it does not look like the enemy gets hit before the effects even reach them.
- Frame 250: `deactivateHit(4)` to deactivate the collider of the ultimate.
- Frame 291: `disableUltEffects` to hide the visual effects of the ultimate.
- Frame 309: `TryUltEnd` to bring the booleans `ulting` and `tryUlt` back to false, since if this clip triggered we did not make it on the other clip.

Finally, the special camera is a camera attached to the character that always points at their face with a simple line of code inside the `Update` of the `CharController`:



This camera also has a canvas attached. That canvas has a slightly transparent white background, a border of speed lines, typically used in manga and anime to represent action scenes, and Japanese letters, also used for action and danger scenes. In addition, in the culling mask, being the layers that the camera will render, I unchecked the layer of the other player (player2 when the player one is ulting and player1 on the other case), so that the enemy does not show on the special animation, since they will be covering the character ulting which would not look good.

To make it all clearer, here are some images of the special animations and the ultimate effects:





The final behaviour of a character is to receive damage. This one is the only one that is not managed by the CharController, but by the objects that inflict the damage, those are the colliders placed on the enemy parts with which the enemy character will attack. Just like we have seen in the ultimate mechanic, when a player attacks, a collider activates. That collider then checks what it has collided with in a script by checking the tag. If the object of the collision is the other player, if that player is not covering, it will obtain the animator controller of that other character and change it to play the corresponding animation of receiving damage. When playing the receive damage animation, the character will be disabled, they will not be able to do anything. It will also obtain the script of the healthbar of the character it hit and will reduce the current health of the character. All the script of the colliders work very similar, but there are slight differences, like playing different animations and taking different amounts of damage:

- Getting hit by the first attack of the combo will make the character play the first receiving damage animation and will take 2 hp if the character is not covering. It will also load the ultimate bar by 1 and the fire bar by 2 of the player attacking.
- Getting hit by the second attack of the combo will make the character play the second receiving damage animation and will take 5 hp if the character is not covering. It will also load the ultimate bar by 2 and the fire bar by 4.
- Getting hit by the third attack of the combo will make the character play the third receiving damage animation and will take 7 hp if the character is not covering. It will also load the ultimate bar by 5 and the fire bar by 7.
- Getting hit by the fire attack will make the character play the third receiving damage animation and will take 12 hp if the character is not covering. If the character is covering the receive damage animation will not play and the damage will be reduced by half, taking just 6 hp. This attack won't load the ultimate bar nor the fire bar.
- Getting hit by the ultimate attack will make the character play the third receiving damage animation and will take 30 hp. No need to check if the character is covering because their controller will be disabled so they won't be able to do it. This attack won't load the ultimate bar nor the fire bar.

Each damage animation is longer, meaning that with the first one the character will be disabled for a brief time and with the last one it will be disabled a bit longer.

## Platforms

Inside the levels, the characters will be able to jump over some elements in the map, for instance, jumping over some awnings to reach the top of a building. To do this, the players will be actually jumping over invisible floating colliders placed so that it looks like they are standing on the actual models of the map.

Now, those colliders can not be always active, if they were, if our character was placed right underneath a platform and we jumped, the platform and the character would collide as soon as the head of the character touched the platform, not letting the character through, which would be really annoying and weird in this style of gameplay. So, what we need here are known as one way platforms: platforms that are disable when the character is under them and enabled when it is over them. This way, when we jump, we go through them as if they were not there, but when we are falling we collide with them, landing over them, just like the following picture shows.



An effective and not too difficult way to do it is to have one script that will be attached to each platform. That script will receive the position of the character, in this case the position of the feet of the character, since we need the whole character to be over the platform for it to be active. It will then compare the value of that position on the Y axis, meaning its height, and the value of its own position on the Y axis. That comparison will be made on the Update function since the character's position can change at any time so it needs to constantly check for it. If the Y position of the character's feet is bigger than its Y position, then the platform's collider enables, otherwise it disables.

One last important thing to do is to assign the character's feet position properly in the script. Like we have mentioned before, we do not know which one of the two characters we will be keeping in the scene until we load the scene, when we will leave the one that the player picked and delete the other. So what we will do is to pass the two possible feet positions, and in the Start function we will call a Coroutine that will execute one second later. This Coroutine will check which one of the two feet positions is not null, that will be the one of the character that did not get deleted, thus the one that the player picked, and will assign that position as the one used on the Update for the checking

of the Y positions. The reason why we call it one second after executing the Start is because if we did it straight when the start executes, it would execute a bit faster than the code that manages the display and elimination of the characters, meaning the non-picked character would still not be deleted and its feet position would still not be null, so our code may end up taking the wrong feet position. This way we ensure that the characters are already properly set before picking the feet position.

Finally it is important to add the new platform to the “environment” layer, since it is the one we used to check if the player is in contact with the ground, which is the base for its movement and mechanics to work properly.



## **Multiplayer**

When I started this project I was not 100% sure whether it would be single player or multiplayer, in fact, I was opting for single player playing against a bot rather than player versus player, so I started developing the game as if it was made for one single player. But as time passed, I started liking the idea of making it multiplayer more and more.

And since I had already developed a lot of things for single player, I had to do many changes to adjust everything to the multiplayer system.

## **Character selection menu**

First and most obvious thing before getting inside the game screen and the mechanics, my character selection screen was for one player only, letting just one player pick their character and setting the other one automatically as the opposite of the one that the player picked.

Now let's remember how this screen worked for one player: it displayed buttons for each one of the characters, by clicking each button, the character would be displayed in the middle of the screen playing an idle animation. Once the button to pick the player was pressed, it locked the character, played a special animation while zooming in on the character and once the animation was finished it loaded the next screen.

Now, besides displaying buttons and models for the two players, the main difference is that we need to wait until the two players have pressed their lock character buttons to play the special animations and zooming in. So, now the function PlayAnimation that triggered the Coroutines ZoomCamera and

StartGame, won't be called when pressing the lock character buttons, instead, these buttons will each call a function for each player called LockChar1 and LockChar2 that will disable the character buttons and the lock in button of the corresponding player and will turn a boolean player1Locked/player2Locked to true. Then, the Update function will constantly check if booleans are both true. Once both of them are true, meaning the two players have pressed their lock character button, then the PlayAnimation function will be called, which now will play the special animation for the two picked characters, and it will call the functions ZoomCamera and gameStart the same way as it did before.

The character buttons work exactly the same as before, only now we will have an array of characters for player 1 and another one for player 2, and two functions called ChangeCharacter, to display the models for player1, and ChangeCharacter2, to display the characters of player2, so the buttons of the player one will call to the first function and the buttons of the second one will call to the second function.

And of course, StartGame will now save the information about the character each player picked.



### Game scene

Inside the game scene, we will also need to make a few adjustments, concretely on the control of the player two, the functioning of the platforms and the interactions between the players.

The first one is the most obvious one. At first, the mechanics were coded by putting only one character in the scene, and picking random keys of the keyboard to test them. Now we need to include the two players, with different controls for each one. In order to do that, we could have modified the CharController script to, before checking the inputs, check which player is the one with the script attached and then check the inputs depending on the player. However, I preferred to make

a copy of the script, call it CharControllerPlayer2 and just change the input keys, it seemed more organised.

I tried to choose the keys so that the two players are comfortable when playing. This way we have:

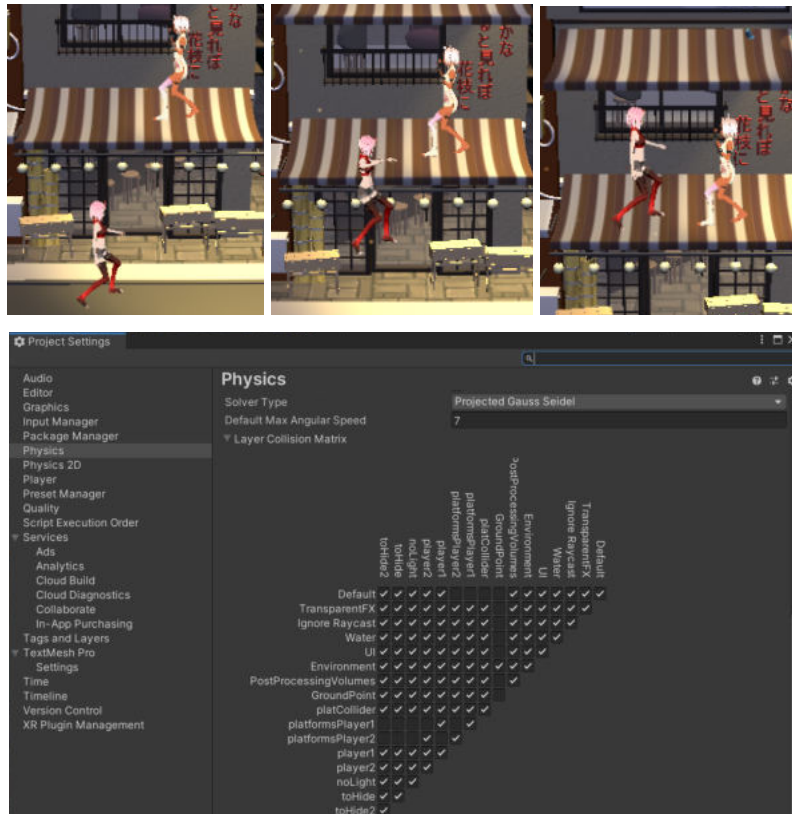
- Movement: A and S for player one, left and right arrows for player two.
- Jump: W for player one, up arrow for player two.
- Cover: S for player one, down arrow for player two.
- Attack: space bar for player one, numeric 0 for player two.
- Fire: 4 for player one, numeric 1 for player two.
- Ultimate: 5 for player one, numeric 2 for player two.

Also, for the scripts that check if the different attacks hit the other player, I was coding and testing as if I was player one so I was only checking if the colliders hit the object tagged as player two. Now we need to extend those scripts so that if the player calling them is player one, they will check if they collide with player two and vice versa.

Moving to the next point, when we implemented the platforms, we did it with one player in mind. Now we need to think of a way to make it still work with two players. Using the same platforms for both players, passing to the script the feet positions of the two of them would not work properly, because either we activate the platforms if one of them is above, meaning the other one would collide with the platform when trying to jump over and could not jump, or we activate them when both player are above, which really makes no sense gameplay wise.

So what I did was to clone the platforms, having two sets of platforms placed in the exact same positions, and I had them both working exactly as described in the earlier section about platforms, one receiving the feet positions of player 1 and the other one receiving the feet positions of player 2. Then I added one important feature: I changed the layers of this platforms groups, instead of being in the "environment" layer, I placed the first set into a "platformsPlayer1" layer and the second set into a "platformsPlayer2" layer, and I made those layers to be only interactable with the layer of the proper character, having "platformsPlayer1" interactable only with the layer "player1", inside of which I put the characters of player 1, and "platformsPlayer2" interactable only with the layer "player2", inside of which I placed the two possible characters of player 2. The intractability between layers can be adjusted inside the Project settings->Physics->Layer Collision matrix, and only the collisions between layers marked inside this matrix will be considered, meaning "player1" layer won't collide with "platformsPlayer2" and vice versa, so if one of the platforms of player1 is active and player two jumps from below, it will pass through it without trouble.

One last important thing to do is to add the two new layers of the platforms to the ones used to detect if the character is grounded, since now the platforms are not in the "environment" layer and we also need to be able to walk and act normally on top of them.



Finally, when I had both players in the scene and started playing around with the two of them at the same time, I realized that there were many situations that could lead to undesired results or bugs that I was not properly dealing with because they only happen when the two players interact with each other:

- The most important bug happens when the two players attack at the same time. Like we said, the attack is handled by a collider that activates and deactivates at the proper frames of the animations. Then, if the two players attack at once, it can happen that one of them, or even the two of them, reach the frame of the attack animation that activates the collider, but then they change to the getting damage animation without getting to the frame that deactivates the collider. Then the collider would remain active and we could hurt the other player by simply walking close to them. Actually, generally speaking, getting interrupted of any action by getting damaged could lead to animation events not being called, so what I did was to create a function inside the CharController script called ResetPlayer that deactivates all the attack colliders, resets the attack combo and turns all the bool variables that manage the actions of the player to false, except the hurt one of course. I then called this function at the beginning of all the damaged animations and that solved the bug.
- Another bad interaction is also related to the fact of getting hurt. It does not happen often, but if you hurt a player when they are jumping, the jump animation would be interrupted, the damage animation would play while in the air and at the end of that animation, since the jump bool wouldn't have turned to false, the jump animation will play again when already on the ground. After considering the possible solutions, I decided that hurting someone who is jumping would "cancel" the jump, putting them back in the ground. To do that, I used a

variable that saves the grounded position from where the player did the last jump. Then when they get hurt, I call a function of their CharController called CancelJump, that checks if the damaged player was jumping when getting damaged, that is, if the jumping bool was true. if it was, I changed the player's position to the position I previously saved, the one they had before jumping, bringing them back to the ground immediately when getting hurt.

- Another thing that is not a bug, but is visually weird, is that if you attack the other player from the back, since the hurt animations are thought to be for front attacks, the visual result seems illogical. To fix that, I created a function in the CharController called adjustOrientation that is called when getting hurt and that checks the orientation of the two players, so that it changes the orientation of the one getting hurt to point towards the one attacking them.

## Sound

Soundtracks are a very important feature of any game. There are lots of elements that need a sound, and there are many different types of sounds. I split the sounds of my games into the following categories:

- Background music: music playing in the background in a not too high volume, in the menus and in-game. This music, like many of the elements of the game, will have a traditional japanese style. The themes playing in the menus will be slow, soft and chill, while the in-game music will be faster, thrilling and tense.
- Character voices: all the voices of the characters, including hurting sounds when getting damaged, brief words when firing and trying to ultimate, a sentence when hitting the ultimate, a cocky sentence when winning and a sad sound when losing.
- Special effects: sounds for the in-game effects, including the basic attacks, the shield activation, the effects of the firing attack and the effects of the ultimate.
- Menu sounds: sounds for the buttons, one sound for the general buttons and another one for the character and map buttons inside the character and map selection menus.

Adding those sounds to Unity is rather simple, to add a sound you need to add an AudioSource to the scene. An AudioSource is attached to a GameObject for playing back sounds in a 3D environment. In order to play 3D sounds you also need to have an AudioListener. The audio listener is normally attached to the camera you want to use.

You can play a single audio clip using Play, Pause and Stop. You can also adjust its volume while playing using the volume property, or seek using time.

Basically, an AudioSource plays audio clips, allowing it to adjust the parameters of the clip. We will have different audioSources for each of the categories of sounds, the one with the music will play right when the scene loads, and it will play in loop, meaning if the clip ends, it will restart. The rest will be shut until a sound needs to be played. In the code, we will need to set the proper clips for the



audiosources and play them whenever we want them to play. The hard task will be to set the proper pitch and volume for each of the clips so that the volume of each clip fits properly inside the whole sound system, and so that they play at the proper speed and time to adjust to what we see.

## **Result**

After all this explanation, it is time to take a look at the obtained results and add some perspective to them using the initial objectives.

I managed to entirely model, texture, rig and animate two original characters, just like I planned. I'm quite satisfied with the result, although I have to say, initially I had some more elaborated ideas in my head about the clothes and accessories of the characters, but at the end I had to keep them a bit more simple because the rigging process is quite complex when having several different elements, and since I lost some time repeating the rigging of the clothes of one of the characters, I preferred to keep a simple but properly developed character design.

I also have a working fighting system in which two players can fight each other using various mechanics. It is true that at the beginning I did not exactly define all the mechanics that the game would include, but in my head I thought of around fifteen possible mechanics I knew from other fighting games. I was not sure about how many of them I wanted to implement but at the end I felt like maybe I should have implemented a couple more, to make the gameplay more interesting. But, like I've mentioned before, I wanted the art to be the main point of this work, and when I started to do the mechanics, I was a bit tight on time and I had some troubles coding them, since there were a lot of variables to take into account, so I decided to, instead of adding a bunch of mechanics fast and not too carefully implemented, implement the most important ones taking extreme care of all the aspects of them.

Another thing I achieved was to keep the same aesthetic in the whole project. I did not include it in the initial objectives but I think it is something important to make every element in the game have a visual coherence with the rest of the elements.

I am also quite satisfied with how the menus ended up looking, they seem quite professional and have interesting elements, like the animations of the characters or the original art of the main screen.

To sum up, the results adapt pretty well to the initial objectives and since it is a project I feel quite satisfied with, I would like to make it available at some online platform so anyone can play it and leave feedback about possible bugs or improvements.

## Conclusions

After everything I have been through, I have to say this was an ambitious project, doing an entire game all by myself, taking care of all the different sections, making sure all of them work and look just as expected was an enormous amount of work.

As expected from practically any big project, there were many mishaps, some things did not result as initially planned, for example the maps that were initially meant to be 2D images drawn by myself, but ended up being downloaded free to use 3D models; some others had to be rebuilt as the first versions were not correct or valid somehow, which made me lose a lot of time that I had to recover working really hard.

But with everything, I managed to achieve a pretty satisfactory result, getting a functional multiplayer game with original menu designs, original characters, and smooth hand made fighting animations, all coherently put together inside one same aesthetic.

I really enjoyed developing this project and I think it has been really useful thinking towards my working future, since I learned a lot of new things both related to the contents of the project and to the management of this one, as I had to carefully organize my schedule to try to adjust as much as possible to the original schedule and deadlines I planned, the same way I would have to do in any professional project.

It has also been personally satisfying to me, because it helped me realise how much I've improved from the day I started this degree. I am perfectly aware of my weak spots and I know they have been compensated before by the fact that most of the projects we did were in groups, so each one of us helped the others with the things we did better. But this time I was on my own, and I was not entirely confident that I would be able to make everything work properly. And not only I did it, but I found it easier than I expected it to be, of course that does not mean the project has been easy at all, it just means that I was more prepared to develop it than I expected, and that is hugely rewarding.

## **Future work**

At this point I am uncertain about the future of the project. I think it actually can work as the base for a way bigger project, with a lot of different characters and maps, and with some more fighting mechanics, but right now and after all I've learned developing it, I think it would be more productive to start a new project from zero, because I think many of the things I did could be improved or done in a different and better way.

## External links

Link to the github's repository of the whole project:

[dazaichan/Queen-s-fate \(github.com\)](https://github.com/dazaichan/Queen-s-fate)

Link to the source code:

[Queen-s-fate/Assets/Scripts at main · dazaichan/Queen-s-fate \(github.com\)](https://github.com/dazaichan/Queen-s-fate/tree/main/Queen-s-fate/Assets/Scripts)

Link to the application:

[Queen-s-fate/Ejecutable at main · dazaichan/Queen-s-fate \(github.com\)](https://github.com/dazaichan/Queen-s-fate/tree/main/Queen-s-fate/Ejecutable)