# FINAL PROJECT
# Interactive Graphics

Martina Valleriani (valleriani.1709963@studenti.uniroma1.it)
Irene Bondanza (bondanza.1747677@studenti.uniroma1.it)
Pietro Nardelli (nardelli.1546380@studenti.uniroma1.it)
"Artificial Intelligence and Robotics" - Università La Sapienza

**Abstract**

"The three kingdoms" is our final project and it is a first-person shooter game. The user plays three levels, which are the three kingdoms. In the first level the player is in the desert and has to fight against a cowboy. In the second level we are immersed in the nature, in a mountain landscape, and the user has to fight against two woodsmen. In the third and last kingdom the game takes place in a medieval town with a big castle and the player has to fight against a dragon. If the gamer wins all the three levels, it becomes the king or the queen of the three kingdoms.

In this project we use *JavaScript* (with the *three.js* and *tween.js* libraries), *HTML* and *CSS*.

## 1 Introduction

"The three kingdoms" is the final project of *Interactive Graphics* course held by Prof. Marco Schaerf in La Sapienza, University of Rome. It is a first-person shooter game where the user plays as a character that needs to take back its throne in all of the three kingdoms.

Each kingdom is a level of the game (from the easiest to the hardest one). For the first two levels, when the enemy is killed, the user can proceed to the next kingdom using an archway that will be lightened by a spotlight. At the start, the user will choose if it wants to fight by day or night and if it wants to play as male or female.

The desert is the first level of the game: this is an hostile environment where a cowboy shoots with its gun while it tries to protect himself behind some teepes (Native American tents).

The nature is the second level of the game and can be seen as the level with medium difficulty. Here, there are two woodsmen that reclaim their land. You need to kill them in order to go to the next and last level.

The castle is the third and last level of the game. It was a peaceful medieval town, until one day a dragon showed up. You need to put him down while it casts fireballs in your direction. If you succeed, you will finally be the king/queen of the three kingdoms.

In this report we will first describe the libraries used and the three levels that have been created, the character's weapons, the enemies and their animations. Finally, the techniques created will be outlined, such as the user interaction choices and the main methods that allow us to claim that "The three kingdoms" is a first-person shooter game.

### 1.1 Three.js

Our project is based on three.js that is is an open source JavaScript library that allows to create and render 3D scenes directly in a browser, providing an extensive API with a large set of functions. Three.js uses WebGL if the browser supports it or HTML5 if it's not supported. Given that this is completely a JavaScript library, it doesn't have to depends by other libraries, running in a standalone manner. Moreover it can be used both to create custom geometries from scratch and to create and to import external model, supporting most of the popular format with great performances.[16].

### 1.2 Tween.js

The animation of our project is created using tween.js that is a tweening JavaScript library, supporting animation for javascript object properties with a simple API, making it easy to create complex tweens by chaining commands. To produce the animations, for each property that needs to be animated inside the *animate()* function, we created a Tween object using the constructor *createjs.Tween.get(propertyToAnimate)*. Once the tween object is instantiated, a flag called *alreadyLoaded* is set to "true" to prevent the reinstantiation of it and produce the animation using the *to()* method, defining the ending property value and the duration (in milliseconds). To produce a chain of

tweens, we simply concatenate them one after another.

## 2 First Level

In the first level we are in the first kingdom: the game is set in the desert and the player's aim is killing a cowboy. The gamer has 10 lives and the same the enemy. If the cowboy kills the gamer then the game is over, you can hear an audio which says "game over" and you are redirected to the menu web page. If the player kills the cowboy then the scene becomes dark and a spotlight illuminates the portal. So the gamer has to go to the portal and pass through it. In this way it goes to the second level.

### 2.1 Desert

For the scene, we have imported the "Low Poly Desert" model [14] taken from Sketchfab. Using Blender, we have added a portal [13], always taken from Sketchfab, and four transparent walls to delimitate the scene. For the portal we have not used the textures, we colored it beige and put in the correct position. For the walls we created them from scratch on Blender adding four planes, choosing the material, setting A=0, V=1, alpha=0, roughtness=1 in order to obtain a transparent material.

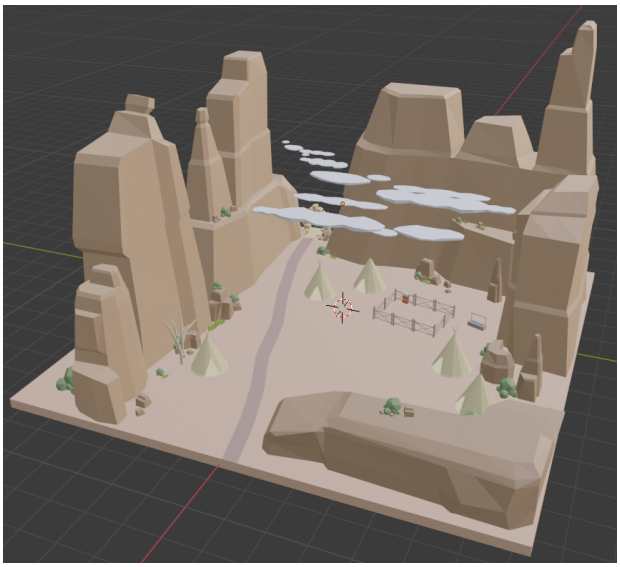

Figure 1: Desert Blender model.

### 2.2 Cowboy

For the enemy we started from a model [10] taken from Sketchfab. We used only the man body and the face of this imported model. In particular, we had to separate the groups that made up the figure and rename the principal components, we selected the right sizes, the desired colors and materials and so we built the hierarchy (see figure 2) choosing "petto" as the root and all the other components as children in a fairly articulated hierarchy (which is visible in the console, setting to "true" the third parameter of the *load_object_gltf()* method, when we load the cowboy).

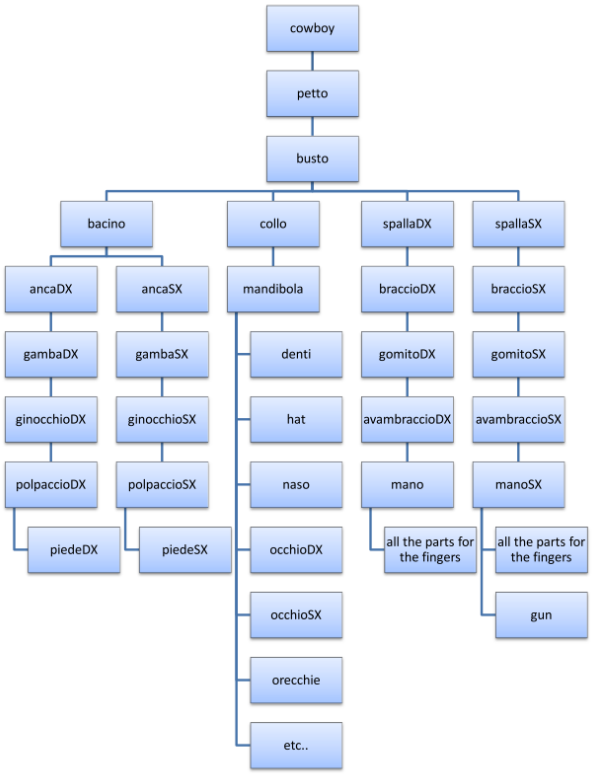


Figure 2: The cowboy's hierarchical structure.

After this, we added a cowboy hat [2] and a gun [1], always taken from Sketchfab. For both the hat and the gun, we didn't import also the textures, but with Blender we assigned to them a material and colored them dark brown and anthracite gray, respectively. Of course, we rotated the fingers of the left hand (in which there is the gun) so that the pose is as realistic as possible.
For the cowboy's animation, we created the instances of parts of the cowboy that we want to position or orient:

- orientation $\longrightarrow$ "spallaDX", "spallaSX", "ancaDX", "ancaSX", "ginocchioDX", "ginocchioSX", "piedeDX", "piedeSX", "petto";
- position $\longrightarrow$ "petto".

Once created these instances, we can animate the cowboy thanks to the *to()* method. We move the legs and the right arm like a normal walk, while the left arm is not animated because the cowboy holds the gun with the left hand, so the arm is always correctly posed and ready to shoot. Then we rotate all the cowboy with respect to both the y axis and the z axis. In the first case we manage the fact

that the figure sways slightly to make the walk as realistic as possible, otherwise the cowboy walk was too much rigid and unnatural. In the second case we manage the fact that the figure rotates when it walks from a point to another one, having a frontal walk with respect to the direction, like we usually do. Changing the position of "petto" along the x and z axes, we determine the path that the cowboys does .

Figure 3: Cowboy Blender model.

### 2.3 Player

Since our game is a first-person shooter game, the player does not have a character which represents it but we can see only the weapon and the camera represents the player's point of view.
In this first level the weapon is a gun, in particular a revolver [1] like the cowboy's weapon. Also for the player's gun, we imported the model from Sketchfab without the textures but assigning a material and an anthracite gray color with Blender.
A common choice in first-person shooter games is to place the gun on the right, slightly inclined, in order to have a realistic pose like if the player is holding the weapon with its right hand. The gun loading is done with the *load_ object_ gltf()* function to which we pass the scene (that allows us to set up what and where needs to be rendered by three.js and where we place objects, lights and cameras),the name 'gun', the path to find the gltf object, the positions where we want to put it and the angles with respect to the three axes.
The player's aim is to kill the cowboy. It has 10 lives, like the enemy, but it can shoot faster than the enemy. In fact this is the first level, thus it has to be an easy level. Pressing 'X' or the left button of the mouse, the player can shoot. The direction of the bullet is determined by the viewfinder, which is green if the gamer can shoot and red otherwise.

## 3 Second Level

This level is the middle kingdom in which players have to compete against two woodsmen. These enemies have different methods of defeating their land: the red woodsman walks around the forest shooting with high frequency instead the green woodsman stay still and thanks to that its shooing are more painful and precise. Players must kill these two enemies in order to complete this level. When both woodsmen are dead the ancient portal will be lightened, which means that it has been activated and players can pass through it and access to the next and last level.

### 3.1 Nature

This world has been created merging different parts from models of sketchfab. This forest contains: trees, rocks, clouds and a mountain imported by [11], the house by [8], the carpentry tool by [15] and the portal by [12]. Using Blender we have merged these objects modelling a plane in which we have placed them and in addiction, we have added four invisible walls all around the scenery in order to prevent players to go outside the plane. The figure 4 shows the Blender model of this kingdom. As we can see, we have not chosen any textures but instead we have selected materials and RGB colors.

Figure 4: Nature world Blender model.

### 3.2 Woodsman

The enemies are models we have created from scratch using two images of a cowboy and modelled them as woodsmen (see figure 5). We have built a hierarchical model starting from the torso of the enemy and then we have defined the remaining parts of the body building a hierarchy that is shown in figure 6. We decided not to add textures but instead use RGB colors and materials and to color

with a red jacket and a black shirt one woodsman and the other one with a green jacket and a grey shirt.

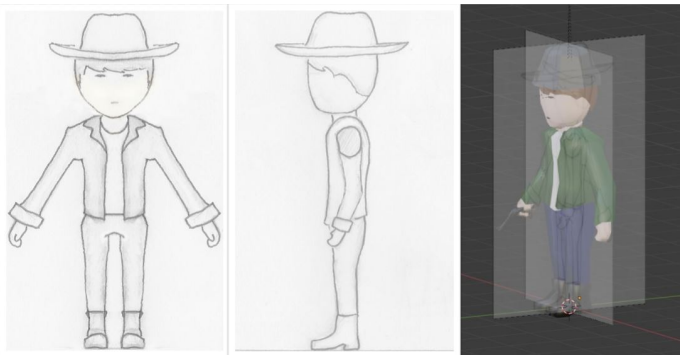

Figure 5: Images used for creating the model and woodsman Blender model.

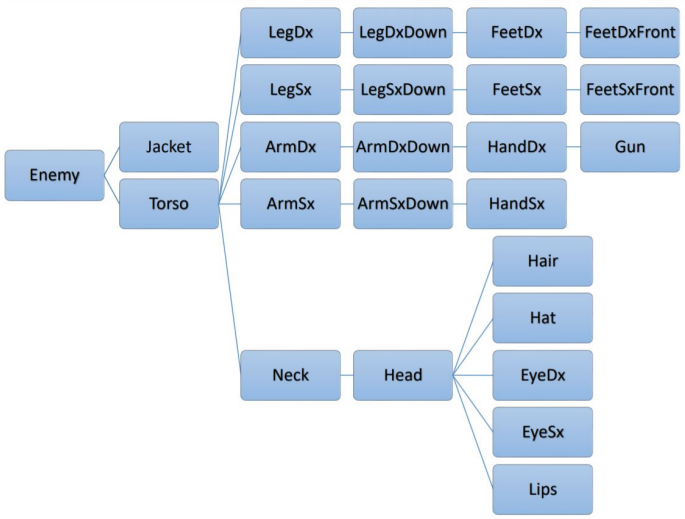


Figure 6: Woodsman hierarchical structure.

As concern the animations of the enemies we have decided to animate just one of the two men with a walking function and both with a function that moves forward their right arm (which is the one that holds the gun), in order to shoot towards the player. For both enemies we have loaded the body parts we would have needed. Thus, for the red woodsman we have loaded: torso, right arm and both legs; instead for the green one we have just loaded the torso and the right arm because as it won't walk we wouldn't need its legs. The animation used for rotating the arm towards the player was done changing the rotation and position of the right arm of the model, ensuring that the enemy has been loaded and then taking the right arm from the scene and changing the parameters we need. In order to allow the red woodsman to walk we defined a circular path that this enemy will do until he dies. In addition, both enemies change their views according to player's position.
As the animations are different between the two woodsmen we have decided to change also their shooting damage and reloading shots, in fact the red enemy will decrease the player's life by 12.5% every 96 frames and the green one will decrease user's life by 17.5% every 122 frames (we use different reloading times in order to avoid enemies to shoot at the same time).

### 3.3 Player

As we said before this game is a first-person shooter game so to represent our character we just need a weapon placed in front of the camera (that is the player's point of view). The weapon was loaded by [7] and is the same gun that the two enemies hold in their right hands. We have chosen this kind of weapon because this level has a nowadays world and this arm reflects modern revolver's shapes (as concern loading, shooting and cross-hair are the same described in subsection 2.3). As for the previous kingdom the player's aim is to kill enemies but this time its shoots are less damaging (decreases an enemy's life by 7.5%, in the first level player's shoots decrease by 10% the cowboy's life) and enemies are more precise when they shoot.

## 4 Third Level

The last level of the game takes place in a medieval town with a big castle. The aim of the player is to kill the final boss of the game, a dragon, that flies in front of the castle. If the player manages to kill the dragon, the user will be redirected to the final page (starting and ending menus are described in section 5). If the player has been killed by the dragon, likewise the other two levels, it has to restart the game.

### 4.1 Castle

The last scene has been imported from the "Castle Town" model [3] taken from Sketchfab. From Blender, we have added a transparent cube that covers the entire scene, to prevent the player to go outside the boundaries of it. The model of this scene can be seen in figure 7. In this case, some textures are applied directly from Blender mainly for the windows and doors of the town buildings .

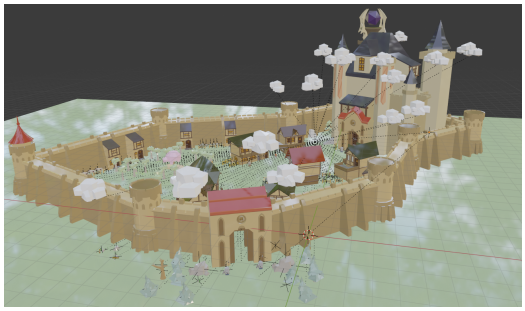

Figure 7: Castle town Blender model.

### 4.2 Dragon

The enemy of this level is a dragon that is built anew by us, using Blender. The model can be seen in figure 8.

Figure 8: Dragon Blender model.

The parts of the model has been created from simple meshes, applying a subdivision modifier. The subdivision modifier is used to split the faces of a mesh into smaller faces, giving it a smooth appearance, enabling the creation of complex smooth surfaces while modeling simple, low-vertex meshes. A Catmull-Clark is applied with a viewport (the number of subdivision levels shown in the 3D Viewport) greater than one. (e.g. legs and arms have viewport equals to 2 while the torso has viewport equals to 4). Of course, higher levels of subdivisions results in more vertices, which means that the model becomes heavier. This trade-off has been balanced to make the game as fluid as possible. Once the main structure of the dragon has been created, additional details has been added. The spines and the horns, for example, has been created from cone meshes but each cone has been modelled differently to produce imperfections that make the model more realistic. Finally, the hierarchical structure has been realized. The torso is the main part of the body from which the remaining parts has been defined, as we can see in figure 9.

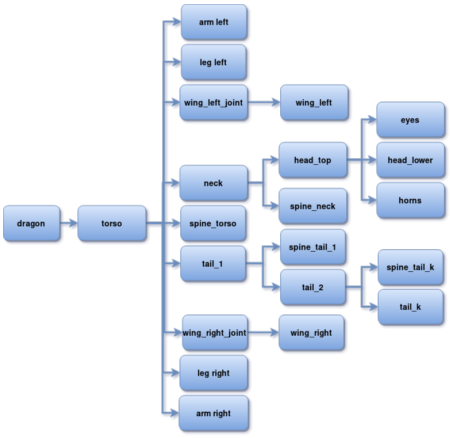


Figure 9: The dragon's hierarchical structure. We omitted some tail and spine-tail nodes that are repeated such that $1 \leq k \leq 4$.

With regard to the animations, we have decided to animate the dragon in a simple and effective way. The dragon flies over the town in front of the castle so that it will face the town center, where the players will play more likely. When the dragon is hovering, wings are moving alternatively from the top to the bottom and vice-versa. Simultaneously, the dragon is moving in the opposite direction as a result of the flapping wings. After some seconds, the dragon is moving on the left, then on the right. The torso is rotating and the wings are slowing down like a bird that is glinding. The legs and the arms are slightly rotating in the opposite direction of the torso movement. To complete the animation, that is composed of a repetition of hovering and glinding, other parts of the body are rotating simultaneously such as tail, head (head-top), mouth (head-lower) and neck. The peculiarity of the dragon is that it will cast fireballs from its mouth in the direction of the character. The fireball is extracted from a model of a weapon called fireball launcher [6] in Sketchfab: the weapon is discarted using Blender while the fireball with the attached textures are selected. Given that this is the last level and the dragon is the final boss, each fireball will decrease the player's life of 25%.

### 4.3 Player

As already said in the previous sections, the character is characterized by a weapon that depends on the kingdom that is playing. In this case, the weapon chosen is a crossbow, that is a classical ranged medieval weapon, taken from a model [4] in Sketchfab. In this particular case, the player needs to shoot an arrow that is extrapolated from the previous cited model. This is the last level of the game so, in addition to the increased damage from the enemy, each part of the dragon will decrease its life of a particular percentage:

- shooting to the head of the dragon will produce a decrease of its life by 20%;
- an hit to the torso will produce a decrease of 10%.
- if the wings are hitten, the dragon's life is reduced by 0.5%;
- for all the other parts, a damage of 0.25% is applied.

## 5 Starting and ending menus

The game starts from a starting menu in which the user can select if it wants to play during day or night, and if it wants to play as male or female. If the users

doesn't select both the preferences (or one of the two), an alert will appear waiting for the selections of the player. To make the menu more appealing, a video [5] is playing in background with a fantasy-like music [9]. A different menu is showing if the player wins the game. When the dragon is beaten, the user will be redirected in this final page where appears: "Your are the king of the three kingdoms!" if the selected sex at the starting menu was male, or "Your are the queen of the three kingdoms!" otherwise. A restart button is also available if the user wants to play again, taking him to the initial menu. Similarly to the starting menu, a video [5] is showing in background while a victory music is played.

## 6 Day/Night

At the beginning the player is in the menu web page and here it has to select the setting and the sex. For the setting it is possible to choose "day" or "night". The variable *light* is passed to the next HTML page (from index.html to index_desert.html) thanks to the GET method, which is one of the most common HTTP methods. So the query string (name/value pairs, in our case the name is *light* and the value is *day* or *night*) is sent in the URL of a GET request.
In the JavaScript files, we have a *get_light* variable which contains the information passed from the HTML page through the URL.
In the *init()* function we handle this fact: if *get_light* is equal to 'night' then we delete the directional light and the ambient light so that the scene is darker; otherwise we add these lights. In the *delete_lights()* and *add_lights()* functions, in addition to removing or adding the lights, the background color is changed: in the first function it is set to dark blue (as the night sky), while in the second to light blu (as the sky in daytime).

Figure 10: The desert scene during day.

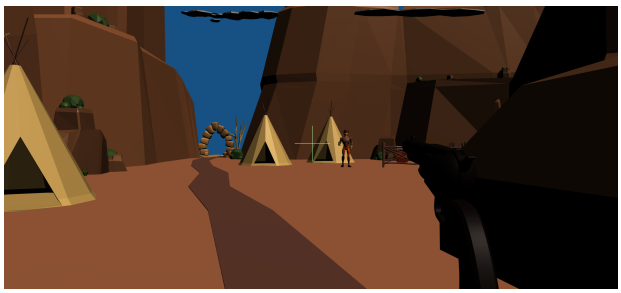

Figure 11: Same scene as figure 10 but during night.

## 7 Portal illumination

In the first and second level, if the player wins, the scene becomes darker and a spotlight illuminates a portal. So the gamer has to go to the portal and pass through it, so it goes to the next level.
To implement this fact, when the enemy is died, we deleted the directional light and the ambient light. If the gamer has chosen the night setting at the menu web page, we added the hemisphere light so that the scene is not completely dark. Then we added the spotlight using the *load_object_gltf()* function. We have to pass to this function the scene, the name *'spot'*, the path to find the gltf object, the positions where we want to put it and the angles with respect to the three axes.

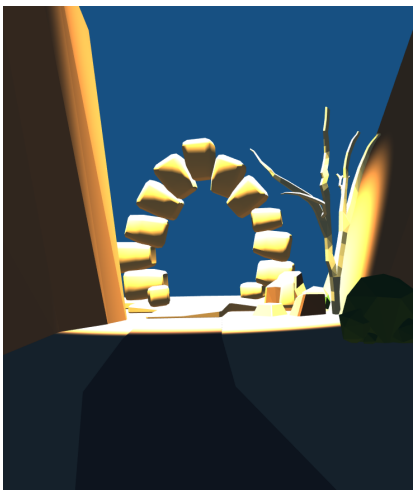

Figure 12: The portal illumination when the player has beaten the enemy in the first level.

## 8 Lives

In order to manage the player and enemy's lives, we chose to add two health bars on the top of the page. In particular, on the left corner there is the player's health bar and on the right corner the enemy's one.
The value of the health bar changes according to *enemyLifes* and *characterLifes* variables. The back-

ground color of the health bars changes in this way:

- when the player or the enemy has sufficient lives, it is green;
- when it has not many lives, the color is orange;
- when it has very few lives, the color is red.

The graphic of the health bars is defined in the *style.css* file.

## 9 Mouse movements

Since this is a first person game we have decided to use a controller for manage user's movements. Our idea was using both key buttons and mouse movements and more precisely using key buttons for moving in the four directions and the mouse movements to change the point of view.
We have used a three.js controller called *PointerLockControls* and a raycaster that is used to set the ground in which the camera walks on. Then we have added *EventListeners* in order to manage five different functions:

a) Block screen.

b) Unblock screen.

c) Key buttons pressed.

d) Key buttons unpressed.

e) Mouse click.

In function a) players can't shoot or move and also enemies can't shoot against players, when a player clicks with the mouse on the screen the window is unblocked b) and now it can start playing. When a player presses a key button between WASD or X, the program calls the common function *onKeyDown()* that enebles the moving towards a specific direction or the shooting according to the button pressed c) and when the button is unpressed is called the opposite function *onKeyUp()* that disable the variables activated by *onKeyDown()* d). *EventListener* e) is used to enable a global variable which starts the shooting function when it is activated.
After initialising the controller we managed how players will be moved: if players press a button between W, A, S or D the camera will move in front, left, back or right respectively. If it is pressed the key button X the player will shoot. We have chosen to enable the shooting both with a key button and the left mouse click because we thought that players may have different preferences and for the same reason users can use both WASD and arrow keys to move themselves around the kingdom.
As a player will move the camera according to in which direction the mouse is pointing, we have used a raycaster to ensure that the camera won't never go under the ground.

## 10 Shooting player

To implement the player shooting, we have loaded the weapon, created the bullet or loaded the arrow, put it in the correct relative position, animated it, added the sound and managed all the consequent actions.
First of all the weapon loading is done with the *load_ object_ gltf()* function, as explained in the previous sections.
Then, in order to implement the shooting done by the first-person player, we add a raycaster class used for create a ray with a selected origin (in our case the camera position) and a selected direction. Thanks to the *intersectObjects()* method, we can know where the created raycaster intersects an object (one of the children of the scene). In this way, the bullet will follow the direction of the ray from the camera to the intersected point.
Once the direction of the bullet/arrow animation is determined, we check if the intersection is different from "undefined", that means that we check if the raycaster actually intersects a children of the scene. Then we check if the gamer has pressed 'X' or the left mouse click (*movements[4]==true*). If it is so, in the first and second level we create the bullet with the *create_ bullet()* function, while in the third level we load the arrow object and add it to the scene with the *load_ object _ gltf()* function. So we put the object in a relative position and a sound (of the shot or of the bow) plays.
If the bullet/arrow is loaded but not even shot, we create the tween instance for the animation and save the position of the intersected point, i.e. of the point where the bullet/arrow has to arrive at. Now the object is ready to be shot, so the color of the crosshair changes so that the player understands that it cannot shoot. Then the animation is completed thanks to the *to()* method and , when the bullet/arrow is arrived at the final position, we do some checks. In particular, we have to check if the intersected object is our enemy or a part of it, because in this case we have to decrease the enemy's lives. If the enemy's lives finish, that is the enemy is died, we set to false the *canShotEnemy* variable so that it cannot shoot anymore and we remove the enemy from the scene and also the bullet or the arrow. Since the enemy died, the player won so we have to make the setting darker and illuminate the portal, as explained in section 7.
When the bullet/arrow is arrived at the end position, we have to remove the object and change the color

of the crosshair because now the player can shoot again. The crosshair has been created directly from three.js, defining a geometry and a material. Once the crosshair is defined and positioned exactly in the middle of the screen, it has been attached to the camera so that when the player is moving the crosshair is moving too. As it will be shown in section 12, each level should have a defined *collisionDistance* variable to prevent the user to collide with an object of the scene. For this reason, the crosshair position along the z-axis should have been at least 1 unit bigger than the collision distance value, to prevent that the character will not get stuck in the crosshair object: *crosshair.position.z = -(collisionDistance+1)*. As already said, the crosshair will be initialized with different material color if the player can shoot its bullet/arrow or if it cannot.

## 11 Shooting enemy

To implement the enemy shooting we have defined a function similar to the one described in section 10 except for the fact that here we don't need a raycaster because we know both initial and final position of the bullet and we added a variable that enables the shooting each specific amount of seconds. The function loads the enemy model and create a bullet. If the enemy shooting is enabled, it computes the direction in which it will shoot according to the position of the camera and then it shoots.
If the enemy bullet hit the player, then the player's life is decremented according to the damage of the enemy shot and if the player's life is lower or equal to zero (which means that is dead) player will hear a sound that says "game over" and it will be redirect to the initial page of the game.

## 12 Collisions

Collisions are an important feature in a first-person shooter game: they provides obstacles that the character cannot get through, improving the realism of the game. three.js doesn't provide a system for collision detection that usually is provided by a physics engine. To not further weigh down the game, we decided to produce the collisions by scratch directly in three.js, using raycasters. Like described in section 9 and more in details in section 10, each raycaster has an origin (camera position) and a direction. Four raycasters are defined, one for each direction along the y-axis (front, back, left and right). Starting from the front raycaster $\boldsymbol{r_F}$, it is possible to define all the other ones from it:

- front raycaster: $\boldsymbol{r_F} = (x, y, z)$
- back raycaster: $\boldsymbol{r_B} = (x, y, -z)$
- right raycaster: $\boldsymbol{r_R} = (-z, y, x)$
- left raycaster: $\boldsymbol{r_L} = (z, y, -x)$

The *intersectObjects()* method provides us a list of all the objects that a particular raycaster is intersecting. Each intersected object has a property called *distance* that define the distance between the origin of the raycaster and the object. Using this simple method and property, we can check if an object is too close to the character (defined by *collisionDistance* variable, that is adjusted for each level of the game), activating the collision. If the collision is detected, a flag for that particular direction is initialized to 0 (instead of 1) so that, in the *motion()* function it becomes a multiplication factor for the z and x directions. In fact, if the value is 0 that particular direction along an axis is forbidden, otherwise if it is 1 the character can move freely along that direction. Of course the implementation of this method is just a starting point for a more accurate collision detection: it is possible to just increase the number of raycasters that starts from the camera. Hovewer, we decided to use the minimum number required to provide the collision detection feature in order to keep the game lighter, fluid and to improve the gameplay across devices.

## References


[1] URL `https://sketchfab.com/3d-models/revolver-b8d459dfb3504d4093c8ecac85e35c59`.

[2] URL `https://sketchfab.com/3d-models/cowboy-hat-c048d51a2e9a4ee99262dc27058d8b9e`.

[3] URL `https://sketchfab.com/3d-models/castle-town-dc23d240d9b54fb594f0a953a46e7588`.

[4] URL `https://sketchfab.com/3d-models/crossbow-bolt-medieval-game-asset-22da51b06e674052affd5dd4d86bdae8`.

[5] . URL `https://www.pexels.com/video/glitter-dancing-in-the-air-854769/`.

[6] . URL `https://sketchfab.com/3d-models/fireball-launcher-9ca2d3a0be4243a7aee0a0c8a68ef155`.

[7] URL `https://sketchfab.com/3d-models/toon-revalber-8d22839630824acab026aa6312628e0b`.

[8] URL `https://sketchfab.com/3d-models/cliff-house-b6e7b196229d43ca896cd2372c5c5abe`.

[9] URL https://freesound.org/people/theojt/sounds/511311/.

[10] URL https://sketchfab.com/3d-models/free-stylized-basemesh-for-blender-sculpting-fc45334ab9fd4f24acb91eb7e17222b3.

[11] URL https://sketchfab.com/3d-models/low-poly-nature-493b488007f14e3b98502ccc60f8582.

[12] . URL https://sketchfab.com/3d-models/teleportation-portal-ad6b0ee7079441cfaaae0453a9c58064.

[13] . URL https://sketchfab.com/3d-models/portal-991938f5f22b4c869246b0c03a9c5944.

[14] URL https://sketchfab.com/3d-models/low-poly-desert-87f3124d9d2743c4986cdcedce269bae.

[15] URL tps://sketchfab.com/3d-models/forester-wooden-house-f404ef2ca80d4f0398f6d942f2e582.

[16] J. Dirksen. Three.js essentials. 2015.