TABLE OF CONTENTS

TheArcher

1 Game Manual and General Description

1.1 Introduction and gameplay walkthrough recording

TheArcher is a rage game third person shooter, it means it must be **difficult**, but not so accurate in the graphics and the models. However, controls should be good and it must be able to **run on every machine**, even the oldest ones. If you encounter some problems going forward into the game because the game is too hard or too demanding for your computer, even with the lowest graphic settings (shadows to minimum quality, wireframes off and clouds off), you can **watch a complete walkthrough** on it, at this link(Google Drive shared .mov video). In **this gameplay** the player never dies and goes from the beginning to the end of the game with the final version of it. Everything that is not visible from this gameplay (e.g. what happens if the player falls down of if he's been hit by an enemy) can be discovered in game or **reading this report**.

1.2 Main menu and settings

In the **main menu** you can see three main buttons: **start, settings and credits**. This last button allow you to see information about the author of the game and let you know where you can find info about the imported models (actually into this document). Settings button instead allows you to tune some parameters about the game that will be sent to the game when clicking the "start" button. The **shadow** parameter allows you to choose between maximum, medium and low, and this changes the shadow quality in the game. The "**clouds**" setting let you disable the moving windows in the game, while the "**effects**" and "**music**" buttons obviously allow you to enable or disable these important characteristics of the game. An additional feature of the main menu is that you can click the **background targets** to make them fall.

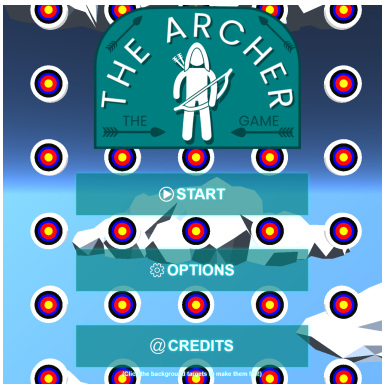

Figure 1: Home menu

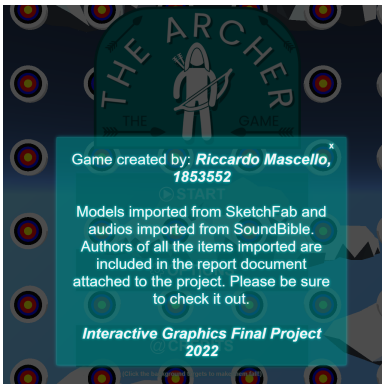

Figure 2: Credits

Figure 3: Settings menu

1.3 Commands

Move the archer using the *WASD* keys of the keyboard and **jump** using the *spacebar*. In order to move the **camera visual** you can move the mouse, and you can also **zoom in or zoom out** the camera using the *mouse wheel*, this will help you to have a better aim in some situations. You can also **run** using the *shift* key, while moving in any direction, to speed up a little bit the gameplay. Pressing the *F* key you will also **shoot an arrow**. When you shoot an arrow, the player will be rotated horizontally in the camera direction (maintaining although his relative direction with respect to the camera, more explanations during the technical discussion). This choices are the typical ones of a third person shooter game. When the player is in the nearbies of a terminal or an hologram projector (explained in details in one of the next chapters of the manual) he can **interact** with it using the *E* key.

1.4 Main components of the game

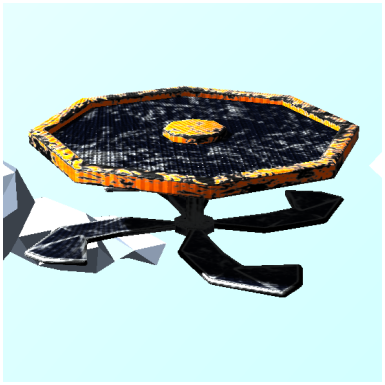

Figure 4: Floating Plaform

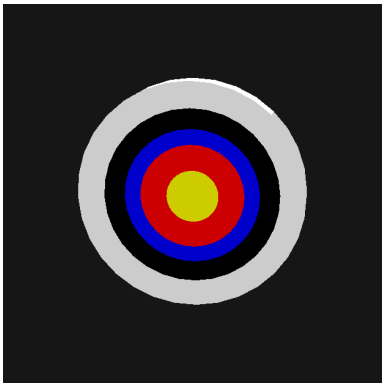

Figure 5: Target

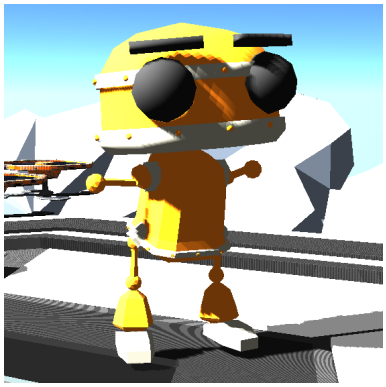

Figure 6: Robot Enemy

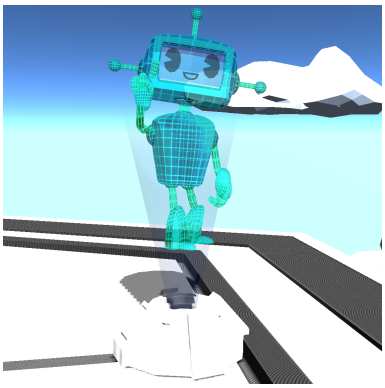

Figure 7: Robot Hologram

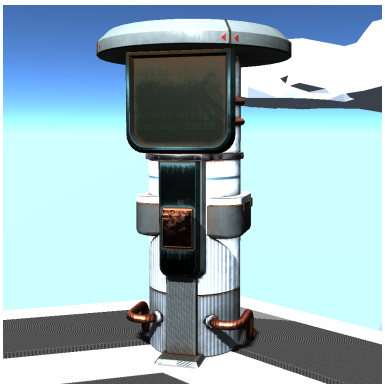

Figure 8: Terminal

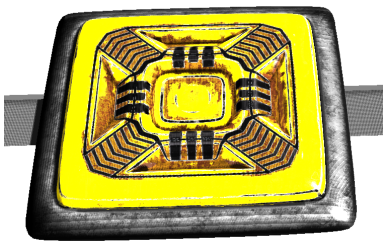

Figure 9: Pressure Plate

Floating platforms: The small floating platforms are very slippery and constantly moving up and down, this causes the player to easily fall from them. They are the main obstacle of the game (and also the hardest).

Targets: Try to hit these targets using your bow. Once you hit them you will unlock the next phase of the game, so they can be considered your main objective in this game. You should have a good aim because usually they're moving.

Robot enemies: Robot enemies are idle until you go sufficiently near to them. Once you activate them, they start running against you, and if they manage to touch you, you will be repulsed (usually falling down). They are not so fast but in some cases they work in group.

Holograms and Holograms projectors: Once the archer is sufficiently near to an hologram projector, he can interact with it, reading hints and information about what he has to do next.

Terminals: Terminals are elements with which the archer can interact to trigger events, more details about each terminal are obtainable from the hologram projectors in the game.

Pressure plates: Pressure plates can be activated positioning on them sufficiently heavy objects. You can simply stand up on them or put on it the metal cubes that you find in the map.

1.5 Game sections

All these sections contain robot holograms that will **explain you what you have to do**, to be short I will avoid to talk about them in each game section.

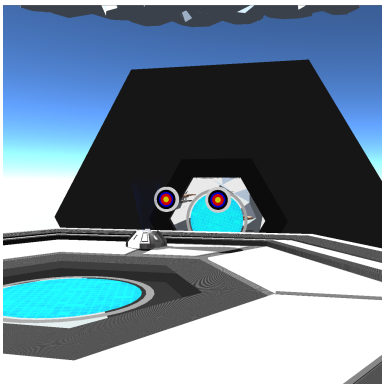

Figure 10: First section

Figure 11: Second section

Figure 12: Third section

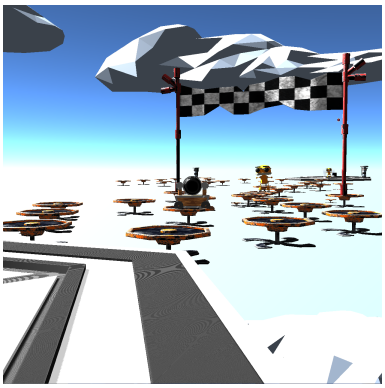

Figure 13: Fourth section

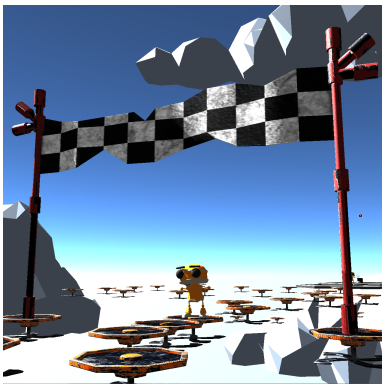

Figure 14: Fifth section

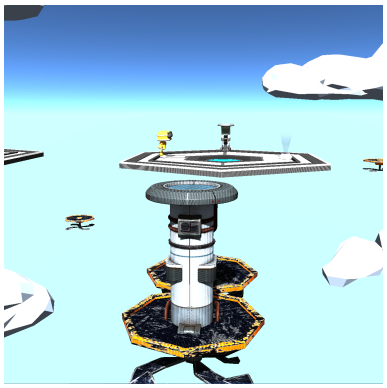

Figure 15: Sixth section

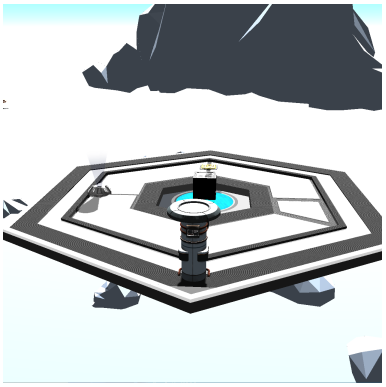

Figure 16: Seventh section

First section: The player starts on a platform where he has simply to shoot at two targets moving in front of him. Once he shot at them, the platform in front of him will rotate to allow him passing.

Second section: In this section the player encounters many floating platforms, only some of them are effectively reachable, while others will animate when both the targets moving in front of the player will be hit. More specifically, when the objective is complete, six platforms will form stairs to go to the next section.

Third section: Here the archer will find three idle enemies that will be activated going near them. The player will have to shoot them and make them fall in order to eliminate them.

Fourth section: Once the player shot out the enemies in the third section, he will face the threat of a shooting cannon in front of him. The cannon will shoot a cannon ball to the player, at every regular interval of time. The player will have to be fast in order to avoid to be thrown out of the platforms by the cannon balls.

Fifth section: The fifth section is immediately after the fourth one, and it's characterized by a starting line and an enemy positioned under it. Once the player will cross the starting line, the floating platforms under its feet will start falling at regular intervals. If the player will run and jump fast enough, he'll reach the next section.

Sixth section: Onto this main platform, the player will find a terminal, that, if activated, will trigger some floating platforms for a certain amount of time. After that timeout, the platforms will fall and the terminal will be again available for use. So, once the terminal is activated, the main objective is the one of running to activate the second one. The second terminal will position a floating platform that will allow the archer to arrive at the next main platform.

Seventh section: Once arrived to this main platform, the player will see a pressure plate and a metal cube. The pressure plate can be enabled either by the weight of the user or by the weight of the metal cube. Since the metal cube could fall down there is also a terminal that allows the player to restore the initial position of this important object. The metal cube is useful to the player, since the pressure plate will show up a target unreachable for the arrows of the player if he's standing directly on the pressure plate. For this reason he should move the metal cube directly on the pressure plate, enabling himself to shoot directly at the target. Once the target has been shot, the game is finally ended, the music will change and some balloons will be shown in the game.

2 Game Technical Description

During the following discussion, firstly the general solutions will be discussed, then I'll deepen the singular sections of the game introduced in the game manual, to explain how these ones were developed and which technical methods have been adopted.

2.1 Libraries

The entire game is based on the use of three main libraries, each one covering important fundamentals of the game environment, they are: ThreeJS, Cannon-ES and TweenJS.

ThreeJS: an interface library based on WebGL to create 3D graphics and visualize it. All the game graphics is built on it.
Cannon-ES: a fork of the older CannonJS library. This last one is not maintained anymore, so Cannon-ES is a better alternative. It is a physics engine that is useful to make the in-game environment interactions more realistic. The presence of gravity, contact materials and collisions make the player actions work correctly (e.g. shoot arrows, jump and fall).
TweenJS: a Javascript library that makes it available for the developer to use interpolation functions in keyframe animations. It also allows to repeat infinitely animations, create groups of them or simply chaining one to the other. All the animations of the game have been built using this fundamental library.

2.2 Models

All the models of this project have been imported from external sources in the GLTF format, using the GLTFLoader provided by default into the ThreeJS library. Here it follows a list of the models included into this project. Under each model you will find a link that brings you to the page from where the model has been downloaded. Following the links you will be able to see also the creator of each model. Some of these models contained already a hierarchical bone structure, that made easier the task of creating animations. In order to be able to manage all the bones of a structure, I created a dedicated associative array to manage the

rotations and movements of each component. One of the most important structures for example, was the one of the archer, that contained many different bones, for each of which I created an entry in the dedicated associative array. The functions that associate the bones of the hierarchical structure, to the entries of the associative array, have a name starting with the keyword *populate* (e.g. *populateArcherStructure*) and they perform this very basic but fundamental task. Once the structures have been populated, the animations can be created using the library TweenJS, but the different behaviours of them will be discussed in the dedicated section "animations".

Figure 17: Archer LINK

Figure 18: Arrow LINK

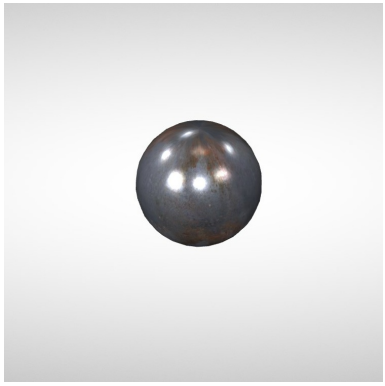

Figure 19: Cannon ball LINK

Figure 20: Cannon LINK

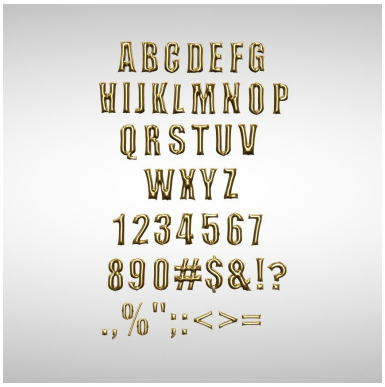

Figure 21: Letter balloons LINK

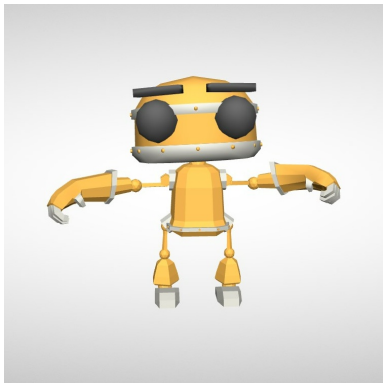

Figure 22: Robot Enemy LINK

Figure 23: Platform LINK

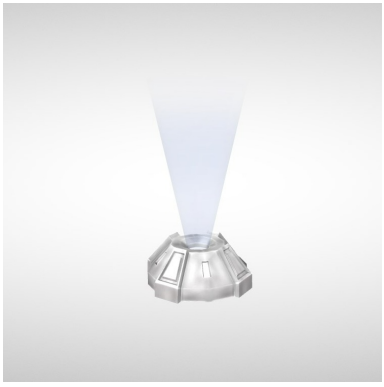

Figure 24: Hologram base LINK

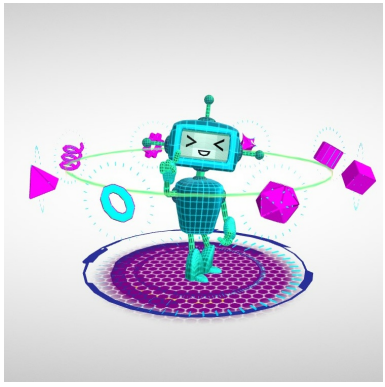

Figure 25: Hologram robot LINK

Figure 26: Metal cube LINK

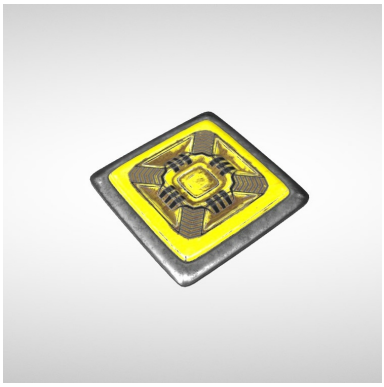

Figure 27: Pressure plate LINK

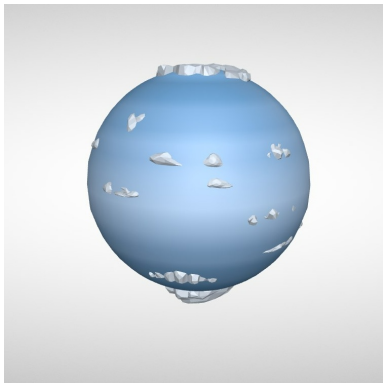

Figure 28: Skybox LINK

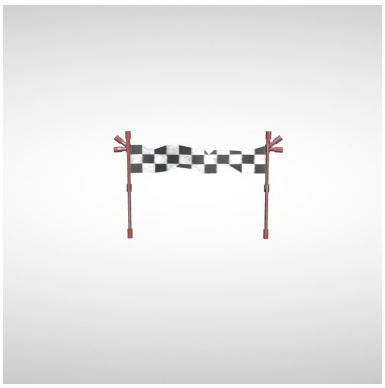

Figure 29: Starting line LINK

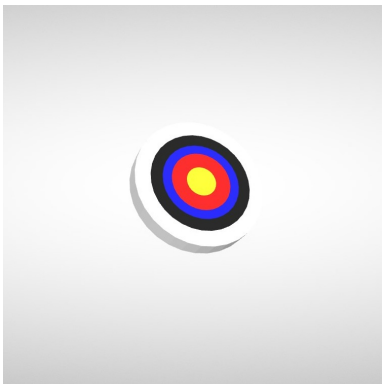

Figure 30: Target LINK

Figure 31: Terminal LINK

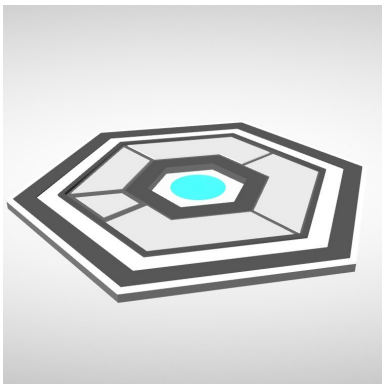

Figure 32: Main platform LINK

Now I will present shortly each model. The **archer model** and the **enemies model** had a very complex structure in terms of bones and meshes, but it was necessary to create realistic animations. **Arrows**, **cannon balls**, **cannons**, **metal cubes**, **pressure plates**, **starting lines**, **targets**, **terminals**, **main platforms** models were considered as unique pieces, not exploiting their structure (very minimal in most cases). Even if the **hologram robot** has a very complex structure (also with particles effects) I removed everything maintaining only the robot in the initial position, animating it very minimally. I had to remove the background sky from the **skybox model** since it was affected by the light and it created a very strange effect. I replaced the background sky with the tunable sky example from the ThreeJS library. **Floating platforms** had the helix as a separate component, so it was possible to make it rotate with a loop animation.

2.3 Textures

Each model imported, contains dedicated textures, it means, like for the models and the sounds, I didn't create any texture, they were made by the models creator already mentioned with the links above. When a model is cloned inside the code (e.g. there are different targets using the same model), each copy maintains the same materials for its meshes, so the materials are not replicate. It means that changing properties like the opacity of a material means modifying every clone, and this in most cases is not an approach to follow. Inside each model there are many different types of textures, and they differ about what they are used for, in the rendering. I'll deepen how the terminal model is composed in terms of textures, since it is one of the most complete examples this project includes. It includes four different types of textures. Let's see for what each texture is useful, considering the terminal example.

- The **base color** which represents the desired color of each pixel for the model at full luminosity, but the effective color will be obtained when applying the lighting model. The terminal contains two base color textures, one for the glass and one for the rest of the model.

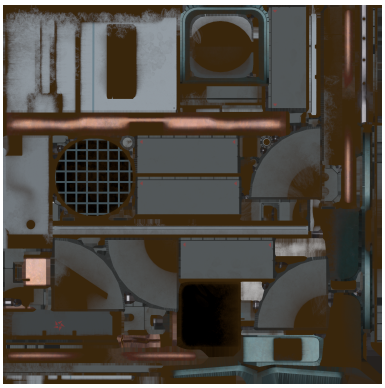

Figure 33: Terminal general base color texture

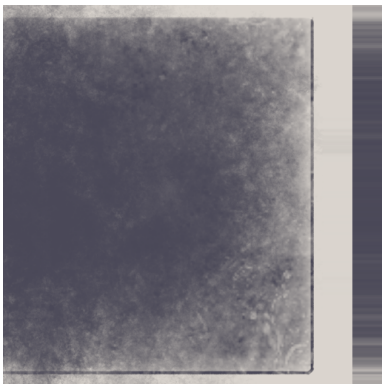

Figure 34: Terminal's glass base color texture

- The **normal map** is used to indicate the normals of each point artificially, to make it seems that you're facing an high poly surface, while instead it's only simulated by the lighting effect. So the normal map is describing the normal direction of each pixel of the surface, to create shadows or in general irregular surfaces.

Figure 35: Terminal normal map texture

- The **metallic roughness** texture indicates how sharply each point should reflect the light. The terminal contains two metallic roughness components, also in this case, one is for the glass and one is for the rest of the model.

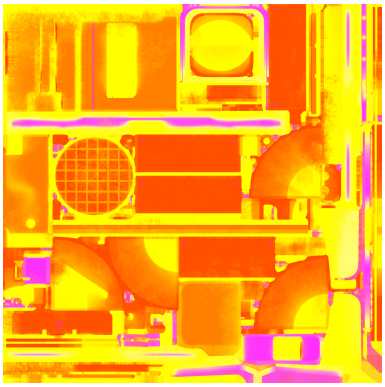

Figure 36: Terminal general metal roughness texture

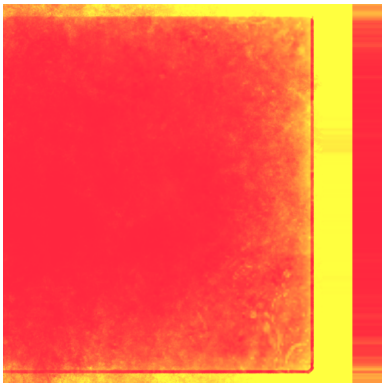

Figure 37: Terminal's glass metal roughness texture

- The **emissive** texture indicates that a model, in particular points, has to emit some light. The color and the intensity of the light are described into the texture.

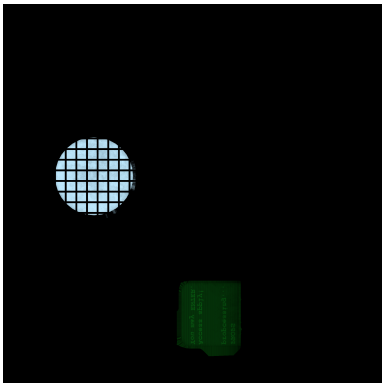

Figure 38: Terminal emissive texture

2.4 Wireframes

Each element in the game environment is characterized by three main components: the model (already mentioned in the previous section), the mesh and the body. These three components must be in the same position and oriented in the same direction in every moment, since they represent effectively the same entity. I decided to implement, for each body, a wireframe that delimitates the physical body, to make it visible in some sense, in order to easily see if a body is colliding with another one or to see if the hit-boxes are correctly implemented. There is an option (mainly for developers) in game to activate the wireframes of every body/model. Of the most important examples of wireframe that I had to design in order to make the shooting work is the one of the arrow. The mesh, so also the body, must be built according to the model dimensions and to the body dimensions. Since the body is not visible, making the model not proportioned to the it and the mesh, would mean having not expected behaviours for the user. For example in the case of the arrow, it would mean to hit an enemy even if the arrow model didn't effectively arrive to the enemy's body, or simply that the arrow model would penetrate the other models.

Figure 39: Arrow's body approximated by mesh

2.5 Bodies

The main element of the Cannon-ES library is the body. A body can be configured based on different properties, like the mass, the dampings and the size. It will be influenced by the world's gravity and by the other bodies. It is possible to modify also the interaction between different bodies, creating contact materials. Each body is characterized by physics materials (different from the material of the ThreeJS meshes) constituted by properties like the friction and the restitution. The contact materials are instead composed by friction, restitution and different other properties, all of them will be used to manage the interaction between two materials if the respective property of each singular material is negative or null. Let's consider as an example the archer's body. Since the gravity of the world has been set to -20 (instead of -9.8, approximately the realistic value) and the mass of the player has been set to a realistic value, which is 80 it means that the jump velocity has been set to an higher value than the realistic one, to make the player jump like in the reality. The unrealistic values derive from a manual and approximated tuning of these ones. The player has linear and angular damping factors set to 0.5, to make the movements of the body easy but not too much. The player's body is considered as a *dynamic* type, which means it's influenced by the gravity and the other bodies. Other bodies like the main platforms are set to *static*, so that they act like if they had infinite mass, and the player can't move them. An example of contact material is the one created to impose rules between the archer and the platforms (all the types of them). The archer must have maximum friction, to avoid sliding, and minimum restitution, to avoid bouncing. The minimum restitution was although not sufficient to completely avoid bouncing, so I had to set the vertical velocity to zero when the player touches the ground. Fundamental bodies of the game are the arrows. These last ones are generated at every shooting action of the player and they are initially positioned at the end of the bow. Their horizontal velocity is then modified to a fixed value in the correct direction, to make them have a parabolic trajectory. In order to generate the arrows in the correct position, and to give them the correct velocity settings, it's needed to know the direction of the camera and the one of the player. In this way, having a normalized vector and multiplying it a factor to have a fixed displacement (or a fixed velocity when shooting), with respect to the archer's body, we obtain the expected realistic behaviour. The same logic has been applied to the cannon, to make it shoot the iron balls in the direction of the player. In this case it obviously doesn't consider the camera angle, but the cannon simply rotates in the direction of the player with a fixed velocity (using the ThreeJS primitive *rotateTowards*). So in this last case the ball must be directed to the player, and in order to make it more precise I didn't use the velocity, but a tween towards the last known position of the player. This behaviour is deepened later in the report, when talking about the cannon game phase.

2.6 Animations

Animations in this project were mostly realized using the TweenJS library. The simplest ones, like for example the skybox rotating was simply realized adding a constant factor, in the animation loop, to the direction of its model. More complex animations, like for example the states of the archer (e.g. running, walking etc.) were realized interpolating some key rotations of the bones composing the model's hierarchical structure. One of the most complex animations implemented is the archer running. It is composed by four keyframes that repeat in loop until the player state changes (e.g. to idle or walking state). In the following images you can see the four sequential keyframes that are interpolated using a linear easing function.

Figure 40: First Run Keyframe

Figure 41: Second Run Keyframe

Figure 42: Third Run Keyframe

Figure 43: Fourth Run Keyframe

Another complex animation to which I would like to dedicate attention, is the shooting. The archer in order to shoot, has to pull out an arrow, place it in the bow, pull the bow cord and release it. In order to realize such an animation I cloned a dedicated hidden arrow model, that appears only when the player is in a certain phase of the shooting animation. This initially hidden arrow has been attached to the right hand of the archer hierarchical structure, in order to make it follow the movement of the hand. Obviously the arrow has been also translated and rotated correctly to be placed inside the hand of the archer, and to remain inside it during the animation. Some screenshots of this animations follow this paragraph (the last state is exactly equal to the first one, with the archer in the idle pose).

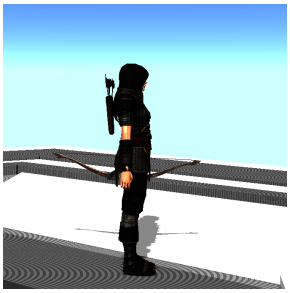

Figure 44: First Shoot Keyframe

Figure 45: Second Shoot Keyframe

Figure 46: Third Shoot Keyframe

Figure 47: Fourth Shoot Keyframe

Some examples of other animations implemented are: the player walking, jumping, idle state oscillating, the enemies running and idle oscillating, the cannon sliding, the platforms floating, the targets moving. I will not treat individually these tweens in this discussion but they can be found commented in the code and in the game.

2.7 Sounds

All the sounds available in this game have been imported from external sources, when mentioning each sound included into this project, it will be also provided the link from which the sound has been downloaded. There are two main categories of sounds in this game: the music audios and the effect ones. Both can be enabled or disabled through the settings inside the game menu. Technically speaking, two classes of sounds have been used, provided by the ThreeJS library: the first is the general *Audio* class, the second instead is a subclass of the previous and it is called *PositionalAudio*. The Audio class provide the basic functions to work with audios, but the *PositionalAudio* version is more sophisticated since it adds the possibility of embedding the audio inside a mesh. This means that when that sound is played it will come directionally from the mesh where it belongs. As classic audios we have the two main themes of the game, one which is playing continuously (Main Theme Music Link) and the other playing only at the end of the game (End Theme Music Link). The only other non-positional sound is the menu click (LINK). For what concerns the positional audios instead

we have all the other sounds:

- **the archer shot** (LINK), **the walk** (LINK), **the run** (LINK), and the **objective completed** (LINK) sounds emitted from the player's mesh,
- the **cannon shot** (LINK) emitted from the cannon's mesh,
- the **metal impact** sound (LINK) emitted from the enemy robot's mesh when this last one impacts the player or an arrow,
- the **terminal** sound (LINK) emitted from the terminal's mesh when the player interacts with it,
- the **click** sound (LINK) emitted from the pressure plate's mesh when pressed or released,
- the **robot blip** sound (LINK) emitted from the hologram projectors when the player interacts with them.

2.8 Controls

The game controls are based on a modified version of the Orbit controls available in the ThreeJS library. I modified the Orbit controls in order to make them meet my needs. Since I wanted to realize a third person shooting game I had to make the camera follow the player and the user had to be able to rotate the camera moving the mouse. Since in the beginning of the project I started implementing the *WASD* commands, I made the camera move with the player when pressing the dedicated keys. The basic movement implemented took into account the camera angle and the current key pressed (*WASD*) to make player move in the correct relative direction with respect to where the camera is pointing at. Obviously the player state changes based on the key pressed, for example the *shift* key, together with a movement button, makes the archer run. The *spacebar* instead changes its state to jumping, until it touches again the ground. Importing the Orbit controls, the player was also able to rotate the camera direction using the mouse, although a problem occurred. The very important problem was that in order to rotate the camera the player had to hold the right button of the mouse. So I modified the source code of the Orbit controls to make the camera rotate even if the mouse button wasn't pressed. Another problem occurred although, while moving the mouse, the player could make the mouse go out of the game window, interrupting the control. So looking at other controls available in the ThreeJS library I found the *PointerLock* controls. These are commands made for first person shooter games, through which the player is able to move the character and also rotate the camera with window locking. This couldn't replace in my project the Orbit controls although, because the camera wasn't able rotate around an object, but around itself (since it was made for first person point of view), in order to modify such a behaviour I would have had to modify a lot of code and also remove the moving commands already implemented in the project, for this reason I decided to take some listeners and general code from the *PointerLock* and put them into the Orbit. In this way, now the player was finally able to lock the pointer, or unlock it (showing the pause interface). The camera will follow immediately horizontally the player, but along the Y axis, I made it reach the player only if he stands at a particular height (within a certain threshold) for a specified amount of time, to not make the camera move too much, causing motion sickness to the user.

2.9 Details and implementation of every game phase

For each phase of the game I'm going to explain how everything was implemented.

2.9.1 Game home menu

The home menu is a mixture of HTML structures, CSS style and some Javascript libraries already mentioned (ThreeJS and Cannon-ES). There are listeners for each button, and in the settings menu, when changing some parameter we have an immediate effect on the background scene, reflecting the game world. The settings are

then held into variables and sent to the game inserting them into the *URL* as *GET* parameters, when the start button is pressed. The shadow setting allows the user to choose between three different types of shadow map size, it is a property of the directional light that characterizes how much the shadows should be detailed. The bigger is the shadow size, the more the shadows will be detailed. Sounds, clouds and wireframes are instead managed through a simple flag that conditionally allows the game to respectively play specific sounds, show the wireframes or add the cloud model to the environment. The home background scene is composed by many targets sliding to right. Each target is given 0 mass, it means that it's not reacting to gravity and to the other bodies. Once a target is touched although, its mass is set to a finite number different than zero to make it fall (possibly also against the others). The touching functionality has been applied thanks to the ThreeJS *raycaster*, which allows, given a camera and a pointer position (normalized) to detect intersections between: a ray shot from the camera position to the pointer selection direction, and any object in its trace. In this case I was interested only in the first intersection, since there aren't elements overlapping. So I see if the first element intersected is a target and I update its physical properties to make it fall. Targets are generated continuously since there is a window in which they are allowed, if a target goes out of this window, it will be removed to not make the scene too demanding computationally.

2.9.2 Initial targets

Once the game starts the player is spawned onto a static main platform and he can interact with an hologram projector or directly shoot at the targets he is facing.
The hologram projector is constituted by a very simple model, but when the player interacts with it, an animated transparent object appears onto the hologram projector selected, with an apparition animation but also a loop moving animation. Once the archer's body is near enough the hologram projector, an HTML prompt is shown inviting the player to press the *E* button. If the player presses the dedicated button the robot's hologram position will change and also the opacity of the model will be tweened to appear more realistic. The robot hologram (even when it's not visible) is constantly repeating the same tween, also opening and closing eyes and mouth, changing easily the scales of the meshes already provided with the model. This transparent figure will also rotate according to the player's position while the text attached to that specific hologram is shown inside another HTML prompt. The player can close it simply pressing the *E* button.
The targets are animated by tweens and they have a collision listener that makes it possible to react if they're hit by a body. In particular if they're hit by an arrow, the arrow will be removed. Once a target has been hit, it will be repulsed in the opposite direction (with respect to the collision, this happens to simulate the clash between the target and the hitting body, otherwise the target would simply fall where it is, not realistically), its mass will increase to a positive value to make it fall, and all the tweens moving it will be stopped, otherwise it wouldn't be free in the falling movements. Once the two targets have been hit, the next platform is subject to a tween animation, rotating and bouncing (thanks to the easing function) allowing the player to go on.
Since targets are replicated entities they could have been inserted into an array of structures, like I did with the floating platforms and other entities, this can be considered as a future improvement for the project.

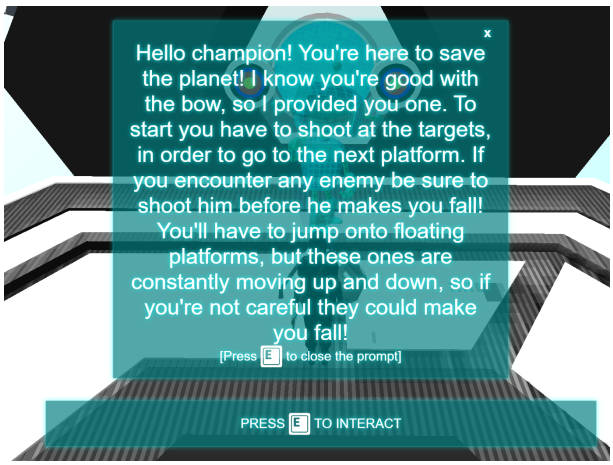

Figure 48: Interaction with an hologram

2.9.3 First floating platforms and targets

The floating platforms have models characterized by an helix that can be rotated infinitely with a tween, making it float realistically. The platforms are constantly moved horizontally to follow a specific path and make that specific phase of the game difficult for the player, since he has to shoot the targets standing up on them. Since the archer can go over the platforms, but the platforms are moved by tween changing their velocity, the archer slides on them, even if the contact material has been set with the highest value of friction possible. To make it seems a desired factor, I added a tween to make the floating platforms go up and down, in this way it seems that the player is pushed down by that movement, and he's not sliding on a surface which should not be slippery. It makes the gameplay harder but it's the best that I could do, since making the platforms slower would make that obstacle very slow and boring.
Once the two targets have been shot (with the same mechanism described in the previous section) some floating platforms are animated to make them compose some stairs. Jumping on the stairs made by platforms the player can arrive to the next game section.
Floating platforms entities are placed in a vector as structures, that contain references to: the body, the model, the wireframe, the rotating helix animation and the vertical floating animation. Some floating platforms' entities can contain additional elements due to the phases of the game in which they are placed. For example there are some platforms needing the velocity, the current phase of the animation and a flag indicating if they are animated at the moment to facilitate the coding. These platforms need these extra information because they have to maintain the velocity indicated by the structure and at specific points they have to start an animation to change their direction to the opposite. So they need to maintain the constant velocity only when a tween is not taking care of it, this is why we need an animating flag into the structure.

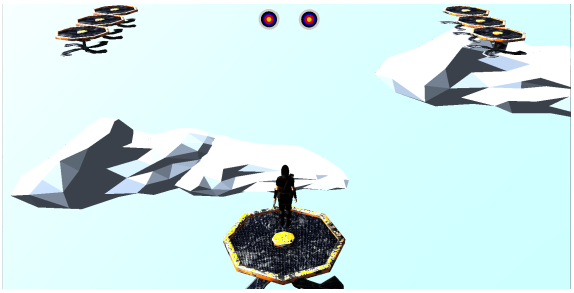

Figure 49: Moving platform 1

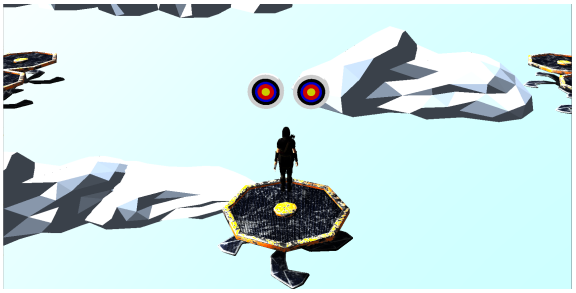

Figure 50: Moving platform 2

2.9.4 First enemies encountered

Each enemy is constantly playing an idle animation (like the archer) when it's not activated by the player. If player is sufficiently near an enemy, it's triggered, and it starts following the player (using as a direction the enemy position subtracted to the archer's position). The robot enemy moves at constant velocity with a running animation. If the robot enemy manages to collide with the archer, than the archer is repulsed in the opposite direction (the objective for the enemy is to throw the player off the platforms). If the robot enemy collides with an arrow shot by the archer then it must be repulsed in the same direction of the arrow. When the collision verifies, I had also to check if the arrow was still "able to kill", and an arrow can be considered unable to do that once it touches the ground. This check is necessary otherwise the enemy would be repulsed also by the arrows left on the ground. An enemy (like the archer) is removed from the game if it falls under a certain value of y. This removal happens also for floating platforms, arrows and cannon balls. Floating platforms can fall only in a specific phase in the game, it will be deepened later in the report.
Also enemy entities are structures composing an array. In this way its easier to animate all of them, simply scrolling the enemies array. The structure of each enemy contains the model, the body, the wireframe, the tweens, the current state (idle or running) and the sound attached to the mesh.

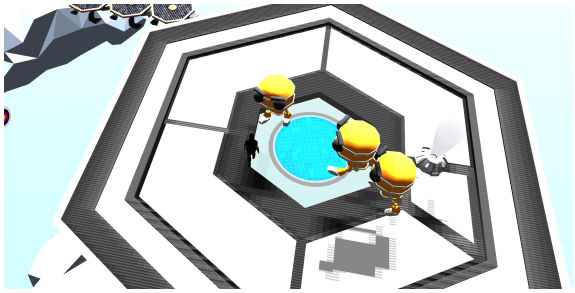

Figure 51: Archer followed by enemies

Figure 52: Archer repulsed by enemies

2.9.5 Cannon obstacle

Once the player approaches the platforms around the cannon, it will start shooting iron balls at the player. The cannon is animated to move constantly, and rotate in the direction of the player. Once the player is in a specific area, the cannon will shoot horizontally in the direction of the player. The shooting has been implemented tweening the position of the spawned cannon ball instead of setting the velocity of the ball only initially. This was done to be sure that the ball reached the player, thing that could not happen is the velocity was too low. So the position of the ball is tweened to the position of the player known at the shooting moment. The cannon shoots at regular intervals, using timeouts. The cannon balls have a collision event listener that repulses the player in the correct direction if they hit him while being still able to kill (same mechanism of the arrow described above).
Every bomb entity characterizes a structure containing the model, the body, the wireframe and the flag indicating if it's still able to kill. All these entities are then part of the bombs array.

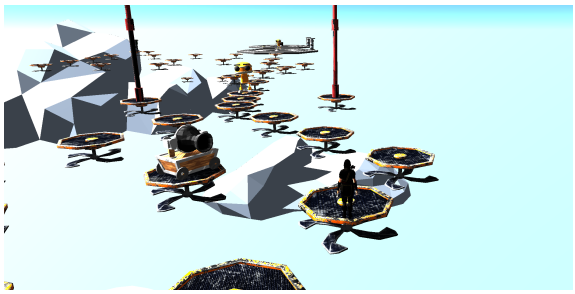

Figure 53: Cannon shooting 1

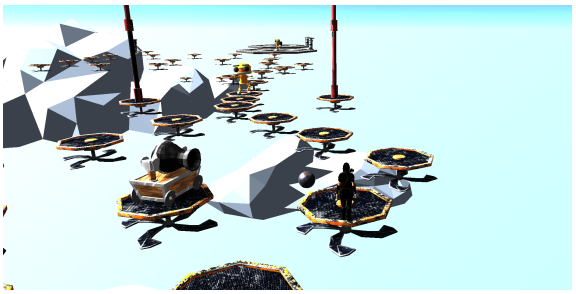

Figure 54: Cannon shooting 2

2.9.6 Race against the time

If the player survives to the difficulty of the automated cannon, he arrives to a starting line. When the player crosses the line, the floating platforms after the starting line will start to fall sequentially. The mechanism is simple, there is a timer that, when it expires, auto reactivate itself and make the next platform fall. The first platform to fall is the one immediately after the starting line. The falling effect is simply due to a change in the mass, so the platforms go from having zero mass (which means being a static object) to a finite number. It's required, like it happened when a target was hit, that all the tweens involving the floating platform are disabled before changing its mass, otherwise it wouldn't be free to fall since its position parameters would be modified by the tweens. The timeout delay has been tuned playing tests on my friends, and now it should be at the right difficulty. The path has been designed to make the player obliged to jump at the right moment on each platform, but still maintaining the correct reachability between the platforms.

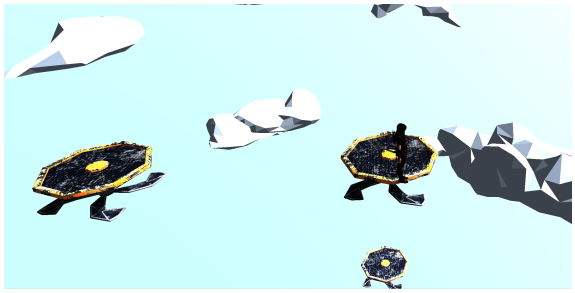

Figure 55: Falling platform behind the archer

2.9.7 Terminals game

If the player ran fast enough, then he managed to arrive at the first terminal in the game. The terminal can be activated in the same way as the hologram projectors, but the behaviour of each terminal is different. So the player has to be near enough to the terminal and press *E* to trigger its effect. In this case (as explained from the hologram projector of the platform) the terminal animates two floating platforms that position in a way so that the player can reach the other terminal. The moved floating platforms remain in that fixed position for a limited amount of time, before falling back to the initial position. After the timeout expires, the terminal became active again and so ready to be triggered another time. If the player manages to arrive to the second terminal, then activating it, it positions the floating platforms in the correct position for the player to return to the platform. But it also positions another platform that allows the user to reach the other phase of the game. Going more into details, each terminal entity is characterized by a structure that contains: model, wireframe, body, the sounds to play, the trigger function and a boolean value that indicates if the terminal has already been activated. Once the player is in the nearbies of an active terminal it means that it will trigger its effect function.

Figure 56: Archer interacting with a terminal

2.9.8 Pressure plate enigma and the end animation

Once the player arrives to the last platform he has to activate the pressure plate to make the designated target go down and be reachable from one of his arrows. However the hologram tells the player two suggestions: the first one is that the player can't shoot at the target simply standing on the pressure plate because the arrow would not arrive to the destination, the other is that if the metal cube falls from the main platform then he can restore its original position using the terminal. The terminal in facts has as a simple trigger function, the one of changing the position of the cube to the initial one. The pressure plate instead is an entity represented as a structure containing: the model, the body, the wireframe, the activation function and the deactivation one. This pressure plate is activated only if the metal cube or the archer are placed within a certain range from the plate's center. They are triggered by position and not by collision in order to avoid that simply touching the pressure plate it's activated. In short we want that the cube or the archer are entirely placed on the platform, so in this way we can manage it changing the threshold in terms of distance, thing that we couldn't manage using the colliders. The metal cube has unitary mass, and it has been created a new contact material to make it slide better on the ground. In this way it's easier for the player to move it, simply going towards it. Once the cube has been placed triggering the target (another tween animation) and the player shot the target, the

ending animation starts. It adds to the scene the balloons model and animate them to make them float in the center of the landscape. The balloons model has been imported as a complete alphabet, I then copied only the necessary letters to compose the sentence I needed, positioning them correctly in a new empty model.

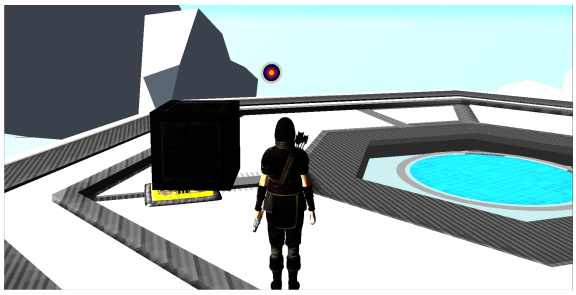

Figure 57: Metal cube triggering the pressure plate

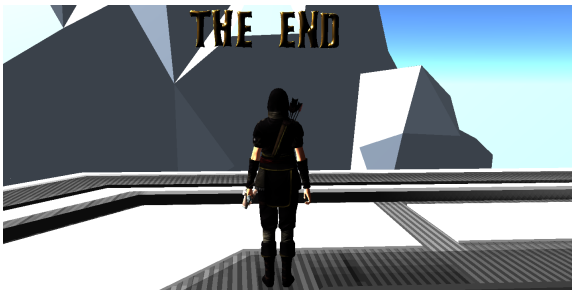

Figure 58: The end balloons

Figure 59: Logo of the game

3 Conclusions and final thoughts

Thanks to this project I learned how to make different entities interact in a world with physical laws. I acquired some additional knowledge about keyframe animations and user interaction.
The code could have been structured better but due to time constraints I didn't manage to apply an high amount of tests on a better structured code. The last part of the code developed is better structured because I acquired some basic experience with the previous parts.
An additional future improvement for this project is the difficulty level, that could be introduced in the settings menu. It could simply change factors like the movement time of the animations (simply changing the tweens time), or the area for activating the enemies (or their speed). Also the interval of shooting for the cannons could be changed, together with lower hints in the hologram projectors for higher difficulties. Also checkpoints could be added to make the overall experience less frustrating.