SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di
Ingegneria dell'Informazione, Informatica e Statistica

Dipartimento di
Ingegneria Informatica, Automatica e Gestionale

Master's Degree in
Engineering in Computer Science - Ingegneria Informatica

Interactive Graphics - Final project

Rotten Souls

A.Y. 2022/2023

Professor: Marco Schaerf

Student (author of the project): Angelo Garcia (matricola: 1812424)

SAPIENZA
UNIVERSITÀ DI ROMA

Index

1. Introduction

1.1. Context and genre of the game

The following project is a simple videogame inspired by the "Soulslike" genre, that is a category of games derivated mainly by "Dark Souls" (made by From Software in 2011), in which the player, exploting the underlying Action RPG ("Role-Playing Game") structure of the game, can fight enemies and bosses by learning their movesets and practising the so called "strategic sword play", i.e. you have to know your equipment and the range, the damage and the speed of your weapons in order to attack strategically, also waiting for an opening in the enemy's movements (for example, if an enemy is doing a very slow and heavy move, almost certainly he will have a long recovery time after that and that is the right window to attack or go back and heal yourself); moreover, this kind of games generally offers a rich world building and a hidden narration to explore (e.g., some important parts of the story-telling are hidden in item descriptions or NPC dialogues), usually set in a dark medieval fantasy world. Obviously, this project is a very simplified version of a "Soulslike" game and the exploration and narration parts are completely absent, but I hope I offered a glimpse on this genre, loved by me and many other people in the world.

1.2. Libraries and tools used

The library used, together with *JavaScript* and *HTML* languages, is only *Babylon.js*; its files (including the basic library plus the GUI and importer parts) are all included in the project directory "basic" and inserted in the header of the launcher file.
Other external tools used are:

- *Blender* (to import, export, modify, rig the characters models and simulate the animations, then manually copied in *Babylon.js*);
- *Ableton 10 Lite* (to cut and elaborate some of the sounds used in the game)

1.3. Quick user manual

The application has been tested mainly on a desktop PC running Windows 10, with an Intel Core i5-7500 CPU (3.40GHz), 16 Gb RAM, and a NVIDIA GeForce GTX 1050 Ti graphic

card (4 Gb dedicated VRAM), using a screen resolution of 1920x1080. The browser used for testing was Google Chrome, but no issues should be encountered with different browsers. The game is not very responsive to dynamic resolution changes (especially the GUI), so if you change it, just reload the page; the optimal resolution to play and make all the graphical elements fit would be at least a 16:9 ratio or similar.
The interaction of the game is based on mouse and keyboard, so I strongly suggest to use these devices.
This is the list of commands:

- **W**: walk foward;
- **S**: walk back;
- **A**: walk left;
- **D**: walk right;
- **Q**: rotate left;
- **E**: rotate right;
- **W+Space bar**: dodge forward;
- **S+Space bar**: dodge back;
- **A+Space bar**: dodge left;
- **D+Space bar**: dodge right;
- **Left click (mouse)**: light attack;
- **Right click (mouse)**: heavy attack;
- **R**: heal.

I also added these debug commands to have more freedom of analysis while examining the project:

- **P**: show all hitboxes;
- **L**: switch between normal follow camera and free camera (the free camera is controlled with mouse clicks and keyboard arrows).

In this demo project, I chose that the player only have one healing herb to restore a bit of health, but I calibrated the difficulty to be reasonable to win with not too much effort. Moreover, while dodging, the character has invincibility frames (about 40 frames at 60 fps), so, if you want to dodge a boss's attack, be sure to synchronize your dodge with the active part of that attack (do some attempts to understand the timing). Finally, the player has 2000 maximum health, 200 maximum stamina (see the following chapters to see about stamina consumption), 750 damage points with light attack and 900 damage points with heavy attack.

The structure of the following parts will follow a "top-down" approach, so a higher level description will be given first, then a deeper explanation will be given later.

2. In-depth gameplay description

2.1. Player commands and event listeners

First of all, even if it will be widely explained in the next chapter, I outline the fact that the movement of both the two hierarchical models in the scene are governed by the “main” bone in their skeletons, which is the parent of all other bones (actually, the pure mesh is never moved) and is animated separately from the other bones.

In the game, the player interprets a warrior wielding a greatsword that can move into an arena (a cathedral) and has to fight an enemy “boss”, trying to win (by dodging his moves and carrying out attacks at the right time) and gain the final treasure.
The player, using the classic “WASD” control system, can navigate the arena, plus he can rotate the character and camera with “Q” and “E” and start a dodge move in any of the four cardinal direction with the correspondent direction key together with the “space” key; the light and heavy attacks to fight the enemy boss are performed by using, respectively, the left and right clicks of the mouse; moreover, the player can heal (only once) by pressing “R”; as mentioned, also a debug mode has been added just to simplify the examination of the project and it is toggled with “P”, to show all the hitboxes which limitates the player (the arena and navigation hitboxes that stop the player from going through walls, columns and the boss himself, they will be better explained in the environment chapter) or compute the hit events of the weapons (body and weapons hitboxes), and “L” to switch the view to a free camera instead of the fixed follow camera at the shoulders of the character.
All these inputs are controlled by event listeners or animation events: infact, as we can see from the code inside “game.js”, I used the accessor provided by Babylon.js “scene.onKeyboardObservable” and added a new observable which read the keyboard inputs, then with two “switch” statements I checked if a key was pressed or released and, if the player character is not dead, which key was pressed. So, inside a key case (we can take the “W” case for example) I check if the “animationIsPlaying” flag is active (flag used to verify if a key is kept pressed, so that an animation is launched only once per prolonged key pression) and then, if the player is not attacking (it is a design choice: once the player has attacked he cannot stop the animation or move until the attack has ended, otherwise you could simply walk to cancel an attack), I stop the current animation group, I switch to the correct animation group and main animatable (recreated every time the key is pressed so that the direction is always computed correctly) and I start the animations, while updating the right flags. All the inputs

work in almost the same way, apart from the dodge command, which first verify if the player has enough stamina and if another movement key is being pressed to decide in which direction activate the dodge, and the attacks commands, light and heavy, which are controlled by *JavaScript* event listeners for, respectively, “click” and “contextmenu” events (also here it is checked if the player has enough stamina). Moreover, I added event listeners on the end of most animation groups (we will talk more in depth about animation groups in the animation chapter), using “onAnimationGroupEndObservable” accessor, to reset to the default “idle” animation when another animation ends (except for walk, whose transition is handled when the corresponding key is released, so it is located in the “BABYLON. KeyboardEventTypes.KEYUP” case in the keyboard listener). In particular, when a dodge animation ends, it is also checked if the corresponding walk key is still pressed (by checking the correspondent flag) so that it can switch again to the walk animation instead of the idle animation.

Now we can describe the part of event listeners that use the *Babylon.js* class “AnimationEvent”, which enables to trigger events at specific frames inside an animation (for example, for the healing animation, the actual recovery of health happens at 2/3 of it, so after 90 frames if you are at 60 FPS). This method is used in particular to detect hit and dodge’s invulnerability events. In fact, the player’s hit (for both light and heavy attacks) on boss is detected by creating a sequence of animation events, one per frame in which the attack is active (i.e., the damage interval is not as long as the entire animation, but it is a subset of it), in which it is checked if the player’s sword hitbox intersects the boss’s body hitbox and, by using a flag called “playerSwordIntersecFlag”, it is guaranteed that the intesection detection is computed only once per attack (otherwise the damage would have been subtracted in every frame that the sword was intersecting the hitbox); then, since I couldn't attach these events to an animation group, I attached them to arbitrary single bones (with the “addEvent” method). The boss’s hit on player events are almost identical, except that they also verify if the player is dodging by checking the “playerIsDodgingFlag” flag. Speaking of dodging, the animation events for dodge animations, instead, are structured as couples, since an event sets the flag mentioned above to “true”, starting the invulnerability frames interval, and a second event sets it again to “false”, finishing the said interval; in particular, the invulnerability frames, at 60 FPS, start at 5 frames and end at 45 frames of the animation, resulting in a total of 40 frames of invulnerabilty (since the application is frame rate independent, this calculation is only indicative and should be recomputed in case of lower or higher frame rates).

2.2. Gameplay parameters

Usually, in this kind of videogames the following data are partially unknown (especially boss’s health and specific damages) to the player in the moment he/she is playing, unless these are searched through wikis on the internet; anyway, obviously, for the sake of examination of the project, they will be explained in detail.

The player's character has 2000 maximum health points, 750 damage points with the light attack (with a faster animation), 900 damage points with the heavy attack (but with a slower animation than the light attack) and 200 maximum stamina points. The stamina is a bar that replenish continuously and it is consumed by actions like dodges and attacks; in particular, the light attack consumes 50 stamina points, the heavy attack consumes 75 points and the dodge (in any direction) consumes 50 points; moreover, dodges don't stop stamina regeneration, conversely it is stopped by attack animations until they end; I want also to point out that stamina can go below zero in the case of an action performed when there is less stamina necessary for that action (in this case the player is "punished" by waiting the regeneration without the possibility of attacking or dodging); in addition, the stamina regeneration is more or less 40 points per second at 60 FPS and it is computed in the main render loop by dividing 40 by the frame rate, so that, regardless of the frame rate on the running system, the same amount of stamina recovered in a second is guaranteed (just as an example, it is 0,667 stamina recovered per frame at stable 60 FPS). Concluding the player's stats, they have only one healing item (a healing herb) which recover 500 health points instantly; obviously, if the healing exceeds the maximum health, it is not counted and the health is set to its maximum (all these things are checked in a corresponding animation event).

On the other hand, the boss character has 12000 maximum health and can perform two attacks similarly to the player: the lighter one deals 250 damage points, the heavier one deals 350 damage points; finally, the boss can neither dodge nor heal himself.

2.3. Boss's behaviour

All the boss's interactions and behaviour (in gaming jargon, also called "AI", with abuse of nomenclature, since it is not a real artificial intelligence in the sense of the machine learning definition) are programmed inside the main render loop, controlled through the library function "registerBeforeRender" (also other player's interactions are checked here, but we will see them in the next chapters).
First of all, it is checked if the boss is alive through the use of a death flag. Then, the distance between the boss and the player is calculated (using the usual cartesian formula $\sqrt{(x2-x1)^2+(z2-z1)^2}$) and the rotation of the model is computed so that it always faces the player's model: the method used implies first the calculation of player's position in the boss's coordinate system (I simply subtract the player's position on X and Z axis by the one of the boss); so, the new angle is computed, using the trigonometric thorems on the right-angled triangle (these will be used later also for player main bone animations), with the arccosine of player's local Z dimension divided by the distance (since this last division returns the cosine of the angle); then, if the boss is not attacking, the new angle is set by modifying his main bone rotation on Y axis and inverting its sign in the case the player's local X dimension is negative (accordingly to the arccosine domain). After this, a "link line" which connects the boss to the player is created and, since I didn't implement a real path finding or

crowd agents, it is used to check if the boss can walk towards the player's character: if both the boss's hitbox and the line are intersected with a column or a wall, the boss is blocked; if he is stuck for too much time (I set a timer of 4 seconds), he is instantly set to a new arbitrary position, possibly a free one.
To give the player more breathing room, I also added the possibility that the boss pauses his attacks for 3 seconds and stays in idle animation; this is achieved through a flag and a counter, decreased at every frame the boss is in pause.
Afterwards, the position and the attacks of the boss are handled as follows. Using the just calculated distance, after verifying that the boss is not in pause, it is checked how far the boss is from the player, distinguishing three cases:

- the distance is greater than 2.5 units (considerable as meters) - in this case the boss, if he is not performing an attack animation, should chase the player, so, using the same trigonometric theorems, I computed the movement on X and Z axes as, respectively, 0.022 multiplied for the sine of the just calculated boss's rotation angle and 0.022 multiplied for the cosine of the same angle (0.022 is the movement increment per frame that I decided to give to the boss, so that he is slightly slower than the player, who makes about 0.03 units per frame at 60 FPS), then, checking the return value of an auxiliary function called "bossCanChase" (contained in the "gameHitboxes.js" file), verifies whether he can chase the player or he is stuck in one of the arena hitboxes and, in positive case, the "stuck" timer is reset, the current animation is switched to the walk forward animation and the "main" bone position is updated with a similar method to the rotation (the X dimension increment is changed in sign accordingly to the sine here too), otherwise it switches to the idle animation and the counter for "stuck" condition is decremented; if this counter reaches zero, the boss's position is updated "manually" (with arbitrary increments);
- the distance is between 2.5 and 2.3 units - here the boss can attack and the attack to perform is chosen based on an random number between 0 and 1 (using the function "Math.random"), just to add a bit of unpredictability to the boss's behaviour (I put the number generation only when the boss is not attacking, otherwise it was too fast and the probability of the "pause" interval was too high): if the random number is lower than 1/6 (0.167) the boss doesn't attack and goes in pause for 3 seconds switching to idle animation, otherwise, if the number is between 1/6 and 3/4 (0.167 and 0.75), he performs a light attack (the light attack is a bit more probable than the heavy one), else again, if the number is greater than 3/4, he performs a heavy attack;
- the distance is smaller than 2.3 units - in this case the boss is too close to the player to attack, so he takes some steps back until he is in the right range (the "main" bone back movement is computed with the same method as the walk forward).

Just as a side note, if the boss is in pause, all the previous checks are skipped and only the pause timer is decremented, resetting it and finishing the pause when it reaches zero.

Finally, if the player is defeated, it is a “game over” and the boss stops attacking and returns to his idle animation. On the other hand, if the player wins, the boss plays his death animation, checking with the help of a flag that it is played only once.

2.4. Additional features

As already mentioned, a debug mode was added to simplify the examination of the project. It consists of two features: the first, triggered with the “P” key, shows all the hitboxes (map and characters hitboxes) and also the link line between the boss and the player; the second, triggered with the “L” key, makes possible to switch between the default follow camera to a free camera to navigate the scene more freely and which is controlled through the keyboard arrows to move on the axis and the mouse clicks to rotate the view (also the wheel click works).

Beyond this, I implemented also a “low spec” mode in case a not very powerful system is used to run the application. It is realized in the main render loop and it checks if the frame rate is under 20.0 for more than 10 seconds (in total), the program asks the user through a *JavaScript* popup if he/she wants to run the “low spec” mode and, in case of positive answer, redirects to a version of the application (“index_low_spec.html”) in which fog effects, shadows and particle systems are disabled, lightening up the execution of the program; if the user answers negatively, the popup is never showed again until the page is reloaded. Anyway, if the system is very old and poorly performing, it is not guaranteed that this mode will improve the application frame rate.

3. Imported models and *Blender* usage

3.1. Player’s and boss’s models

First of all, I want to say that all the animations and the entire application are frame rate independent, in the sense that all the variables that represents a duration in time (like key frames in animations, or timers) are a fraction of the frame rate, so, regardless of the fact that it is high or low, the important actions take the same “real” time.
Moreover, I want to point out that the entire scene is built using the function “createScene” which is made asynchronous (using the “async” key word); in this way I could use the “promise” of the function as return value, I didn’t lost the reference to the loaded meshes’ variables and the meshes could be rendered only when they were fully loaded (improving the loading look and performance).

The player's character model is a darkwraith model, an enemy from the original Dark Souls game; instead, the boss's model is a silver knight, another enemy from the same game (evidently, I chose humanoid models to simplify the rigging process for both meshes). All the models have been downloaded, in ".obj" format, from the site "models-resource.com" (by the Videogame Resources community). The ".obj" file, which basically contains the vertices and the geometry, is always accompanied by another file in ".mtl" format (referenced at the beginning of the ".obj" file), which instead contains all the data about materials and lighting, including bump/normal maps and diffuse, specular and ambient components of the light on the material (respecting the Phong's model). To elaborate, correct and export all the imported meshes I used the program "*Blender*" (Steam version).

Blender is a free and open-source 3D computer graphics software tool set used for creating animated films, visual effects, art, 3D-printed models, motion graphics, interactive 3D applications, virtual reality, and, formerly, video games. *Blender*'s features include 3D modelling, UV mapping, texturing, digital drawing, raster graphics editing, rigging and skinning, fluid and smoke simulation, particle simulation, soft body simulation, sculpting, animation, match moving, rendering, motion graphics, video editing, and compositing. In addition, it is very well supported and promoted by the *Babylon.js* community.

As both characters meshes and also other environmental objects meshes received the same treatment in *Blender*, the following description is general and valid for almost all the meshes.

Firstly, I imported the ".obj" together with the ".mtl" files into *Blender*, creating a project file for each model, I removed the automatic smoothing of the normals (which was creating some defects in the visualization of textures and lighting) and, since the imported models were often composed of mutliple meshes, I unified them into a unique mesh, also to simplify all the next steps, and I adjusted some textures to be a bit brighter and shinier (I modified various options in the material tab on the right in *Blender*). Then, learning from a tutorial on Youtube[1], I started "rigging" the models, i.e. the process of creating a "skeleton" (also "armature", it is like a special invisible mesh, composed of smaller parts, the "bones", that governs a normal mesh to which is connected), following these steps: I started creating the bones of the skeleton in a way that they were fully hierarchical, beginning with a "main" bone (which is not a real bone that deforms the mesh, but it is the parent of all the other bones and it is needed to control them all in once), then continuing with the "chest" as a child of "main" and so on, also checking the "deform" option so that the following bones can manipulate the mesh around them (see the scheme in *Figure 1* to see the tree of all the bones; the two characters skeletons differ only in two bones, the "sheath" bone for the darkwraith is raplaced with the "cape" bone for the silver knight; only the darkwraith one is showed in the scheme); finished the rigging on the left side of the character, I used the *Blender* feature to mirror the bones on X axis to automatically create the symmetric bones on the right side, adding manually the asymmetric bones (like the sword or the sheath); just as a side note, in an early phase, I created

[1] The mentioned video: https://www.youtube.com/watch?v=mYgznqvbisM

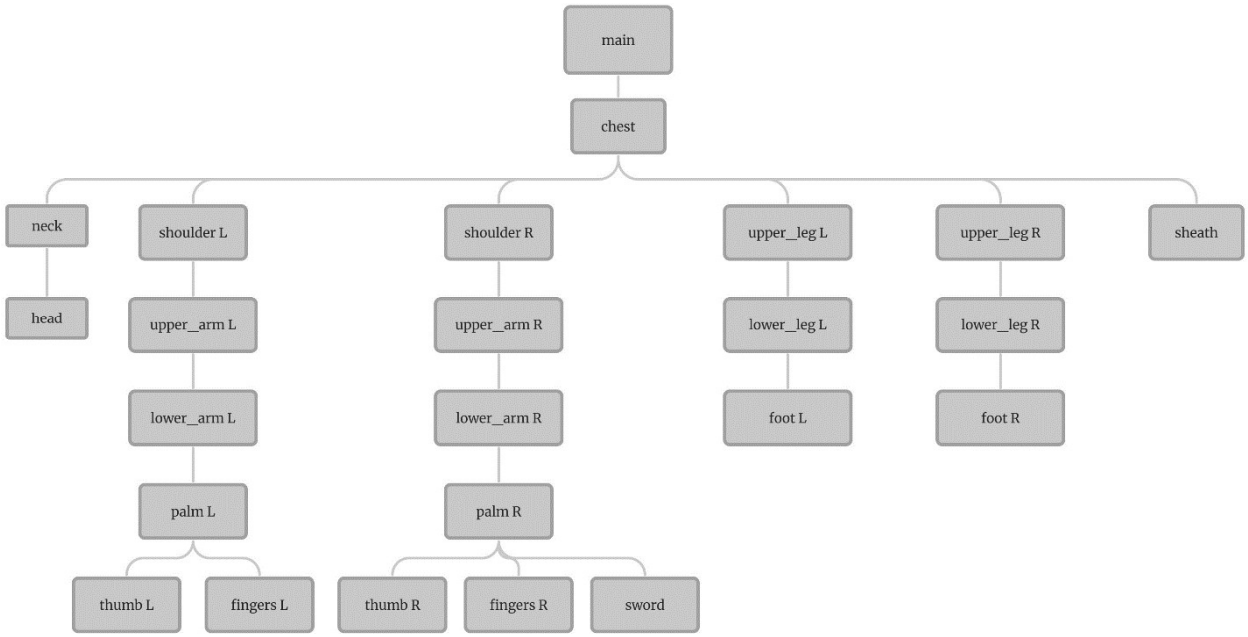

Figure 1: Bones' tree of the hierarchical player's character mesh

also the so called IK bones ("Inverse Kinematics"), i.e. special bones that don't deform the mesh but control a group of other bones and helps to ease the creation of animation without thinking of moving the single bones, for the arm (through the palm and the elbow) and for the legs (through the foot and the knee), but I had to remove this feature since these bones don't work when the skeleton is exported into Babylon.js; after completing the skeleton (see *Figure 2*), I parented it to the mesh and, since the automatic weight assignement in *Blender* didn't work due to some error in the mesh detection, I started a long phase of "weight painting", i.e. drawing with brush-style tools the influence of every single bone on how much it deforms the vertices of the mesh around it, going from dark blue (no deformation) to red (full deformation of those vertices), as showed in *Figure 3*.

Figure 2: Full skeleton for the boss's model (silver knight)

So, after finishing the rigging and all the adjustments on the models, I chose to export the models in ".babylon" format because it also contains the skeleton (differently from other

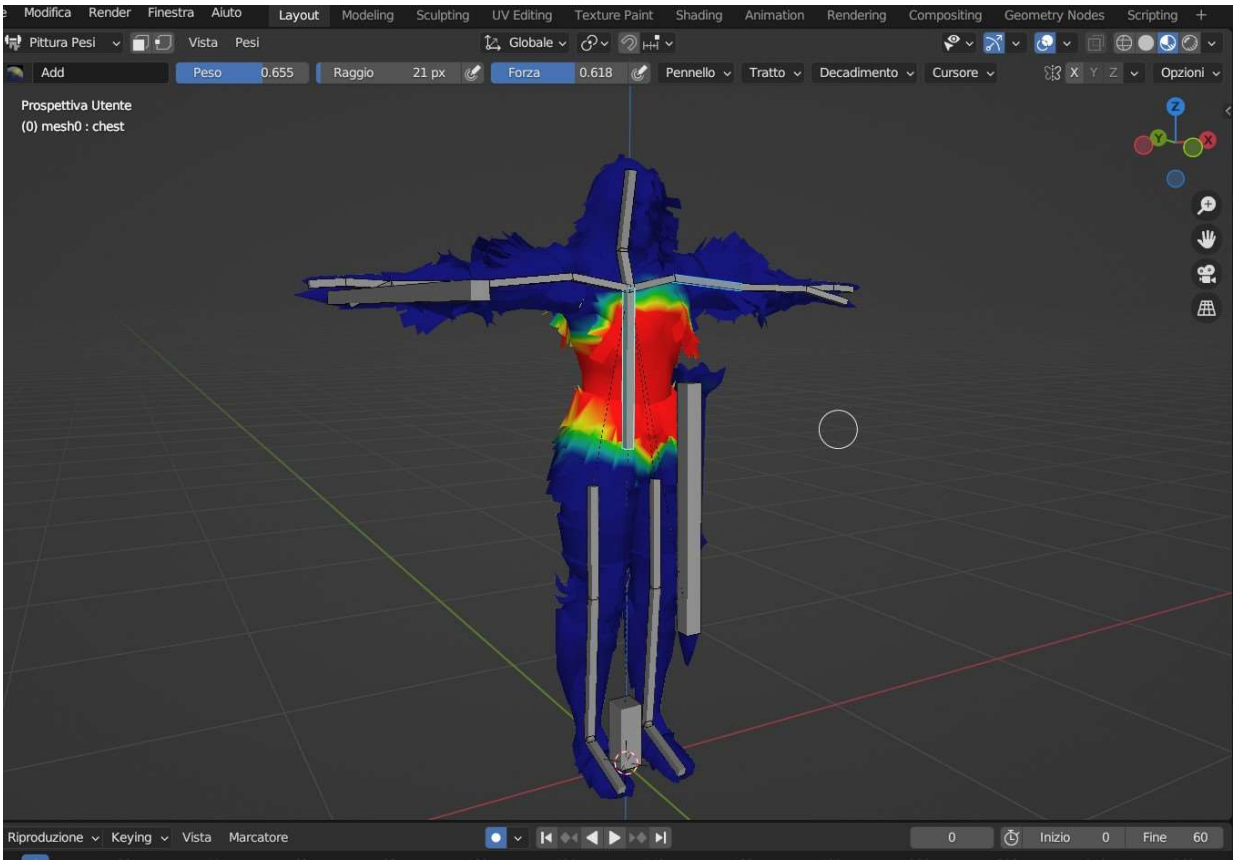

Figure 3: Example of weight painting for the "chest" bone of player's model

formats) and is better supported in *Babylon.js* libraries, but, since *Blender* can't natively export in that format, I had to install an extension (called "Blender2Babylon") and tweak some parameters for this exporter (as some settings, like back face culling, "force baking" or PBR materials, produced many errors in the visualization of the models in *Babylon.js*). The models exported with this extension contain all the geometry and are associated with all the appropriate normal map textures, diffuse textures and specular textures (they are all consultable in the "assets" directory).

Afterwards, inside the main game file ("game.js"), I imported the models (previously located in a dedicated directory inside the project) using the library function "ImportMeshAsync", which takes as parameters basically a path, a file name and the associated scene, preceded by the "await" key word to respect the asynchronous nature of the "createScene" function (also "ImportMeshAsync" returns a "promise" that can contain meshes, skeletons, particle systems, animation groups and transform nodes, if any). Moreover, as for some reason linked to the axis order used in *Blender* the imported meshes were mirrored on the X axis, I set the scaling of the "main" bone to -1 to solve this problem (mirroring only the mesh didn't work). Finally, I set the mesh property to receive shadows to "true", I set the max simultaneous lights for all the materials to 6 and, most importantly, I set for the main meshes the "alwaysSelectAsActiveMesh" property to "true", otherwise the camera didn't prioritize the other meshes excepts the first imported and they disappeared partially from certain camera angles.

3.2. Player's model animations

To better organize the application and keep the code modular, I created a *JavaScript* file for each individual animation, everyone located in the "animations" folder. In particular, the animations for the player's model are: an idle animation, four walk animations (one for cardinal direction), two rotation animations (one for clockwise, one for counterclockwise), four dodge animations (again, one for cardinal direction), a group of animations for the "main" bone (movements and rotations), light attack and heavy attack animations, a healing animation and a death animation. Since almost all the animations were produced following the same process, I will give the description of a generic animation creation in the project.

First of all, inside the main application file, I initialized two arrays for each rigged character model (four in total), one filled with the bones' initial position taken from the default T-pose and one with the bones' initial rotation, so that, when I had to specify the positions and the angles in the animations keyframes, I had only to compute the increment from these initial values (the imported models didn't start with bones at 0.0 rotation, so I had to think at this solution). So, in separate files (one for each animation, named accordingly), I created a function corresponding to each animation, which takes as parameters the skeleton of the interested model, the frame rate of the application (taken from the *Babylon.js* engine, which was initialized before the main game function as a global variable, as well as the canvas variable), the initial bones' positions array and the initial bones' rotations array. After this, I opened *Blender* on the interested rigged character and I started doing prototypes of the animations, just by selecting the skeleton/armature, going in "Pose mode", setting a frame rate and an animation length and grabbing or rotating the bones until I reached a satisfying pose for each key frame I was designing. Returning to the *JavaScript* animation file, here I read the correct bone to animate from the skeleton, I read the initial position or rotation from the arrays passed as parameters (in the whole application, I always obtained the bone's index by using its name and the library function "getBoneIndexByName"), I initialized a new "Animation" object, passing to the constructor the animation's name, the property to animate (which is always either "position" or "rotation", that are accessors in the "Bone" class), the type of the property (which is always a "Vector3"; I decided not to use quaternions for simplicity) and the type of looping for the animation, which can present three main cases (actually four, but one is never used) through library macros: "ANIMATIONLOOPMODE_ CONSTANT", that makes the animation pause at the final values when it ends, "ANIMATIONLOOPMODE_CYCLE", that makes the animation restart from the initial values (used mainly for animations that loop, like walk and idle), and "ANIMATIONLOOPMODE_RELATIVE", that makes the animation repeat starting from the final values (useful for animations that progress in time); then, I initialized the key frames array in which every element was formed by two fields, the frame number and the value of the property, manually filling them with the values read from *Blender* (as shown in *Figure 4*); just as a side note, since the values from *Blender* were often inverted or on wrong axis, I

Figure 4: Example of animation in Blender (only the "chest" bone is showed)

had to manually adjust them time to time for different bones until they matched a satisfying movement; so, I set the key frames for the current bone's animation using the "setKeys" function and I added it to the "animations" array inside the correct bone. I repeated this process for all the necessary bones, leaving untouched the ones that were not animated for the current animation.

Inside the animation file, I also added a function that creates an animation group ("AnimationGroup" class), adds all the single bone animations to it (through the "addTargetedAnimation" function) and returns it as result, so that I logically gather all the single animations into one entity that can be easily played or stopped.

A special mention goes to the "main" bone animations, which in fact are located in a separate file and are independent from the other animations to guarantee more freedom to control collisions with hitboxes and movement generated by input keys. They are used to make the player's character actually move in the game environment and consist in four animations for the four walk directions, two for the rotations and other four for the dodges in the four directions. Differently form the other animations, the loop mode is set to "ANIMATIONLOOPMODE_RELATIVE", since they have to progress in time and space. Moreover, the movement is computed by using again the trigonometric theorems of the right-

angled triangle, obtaining the movement on X axis with the desired distance multiplied by the sine of the current angle (Y component) and the movement on Z axis with the same distance multiplied by the cosine of the angle, which are then fragmented on the key frames (see the code for more details); this computation is inverted in the case of lateral movements.

Finally, all the animations are initialized in the main game application by calling all these functions and putting the animation groups in different variables (keeping one variable for the current animation group that changes contionuously).

3.3. Boss's model animations

Since the boss does not need to walk laterally (he follows the player's model by tracking its position), dodge or heal, the boss's character model animations are: an idle animation, two for walk forward and walk back, one for the light attack, one for the heavy attack and one for death. The process to create the boss's animations is the same followed for the player's animations. The only thing that I want to point out is that the boss's "head" bone is never actually animated, because it is governed by a look controller (similar to a "look at"), created through the class "BoneLookController" that is then updated at every frame in the main render loop with the "update" function; the only exception is the death animation, since the bone look controller is no more updated when the boss dies and I had to animate the "head" bone manually.

3.4. Other models

Beyond the player's and boss's models, the project contains other imported meshes to create the environment of the game, including: a cathedral (the actual arena in which the player fights the boss, erroneously called "chapel" in the files) (see *Figure 5*), a chandelier (not very visible from the player's perspective), a treasure chest (the prize for winning the game), another sword (the Moonlight Sword, contained in the treasure chest) and a couple of "easter eggs" objects (i.e. another chest and a special helm, both located in the back of the arena, in front of the three statues). Also all these models were elaborated in *Blender* (to correct the normals smoothing and unify multiple meshes into one) and then exported in ".babylon" format, together with their textures. Finally, I want to point out that also the treasure chest mesh has a skeleton (very simple, composed by only two bones), in order to have the possibility to create a nice opening animation when the player gain the prize; this animation, together with the one for the sword coming out of the chest, are unified into a unique animation group and are located in the "chestAnimation,js" file.

Figure 5: View of the cathedral arena from outside (the strange look is due to the back face culling)

4. Environment creation

4.1. Models' placement, cameras and skybox

The entire game setting is built inside a single *Babylon.js* scene, which is initialized at the beginning of the main function "createScene", using the engine variable as parameter (in turn, initialized outside of the function as global variable, together with the canvas, used to create the engine, and the frame rate). The environment consists of a cathedral placed at the center (matching the height to align the floor in 0.0), which contains the arena with its hitboxes, the characters' models (boss and player, located one in front of the other at a certain distance), two instances of a chandelier mesh, the treasure chest and the reward sword (invisible until the player wins) and the two "easter egg" objects in the back; underneath the cathedral, even if it is not much visible from the player perspective, I put a ground object matching the stairs at the entrance to give a bit of depth when looking outside of the door (the ground uses the same bricks diffuse texture taken from the cathedral mesh, but the normal texture applied to it was generated by me). All the placements of the meshes are achieved by setting a new "Vector3" object to the "position" or "rotation" properties of the meshes, except for the character models, which are moved through their skeletons and "main" bones.

The skybox is usually used in videogames to represent the sky with a simple large box (somethimes a half-sphere is used in its place), Here in the project, I realized the skybox by

following the official *Babylon.js* tutorial, so I found six images to represent the sky (I chose to set the scene at the sunset), I created a very large box, I disabled back face culling to it (since the box must be visible from the inside) and I set the reflection texture of the material as follows: I created a "Skybox" directory (inside the "assets" directory) containing the six chosen images named as "skybox_px", "skybox_nx", "skybox_py" and so on, which symbolize the images to locate in the positive X, in the negative X, etc.; I initialized a new "CubeTexture" object, passing as argument the path "/assets/Skybox/skybox", so that it automatically detect the suffixes in the names; lastly, I set the coordinate mode of the texture to "SKYBOX_MODE" (so that it applies a cube texture internally) and the material and the position of the cube.

As already mentioned, the cameras usable in the game are two: a follow camera, which follows a specified target, that, in this case, is the frontal player hitbox (set later when it is created; it is explained in the following paragraphs), and whose radius, offsets and speed are adjusted to match an acceptable point of view; a free camera, not locked to any mesh and controllable for debug purposes (it is not advised to play the game with this camera). These two cameras are swapped by modifying the "activeCamera" property of the "scene" variable.

4.2. Lights, shadows and particle systems

The lighting of the scene is organized basically in three lights: a hemispheric light, that acts as a external ambient light and is slightly colored orange to match the sunset skybox; two point lights, that represent the lights given by the chandeliers and are limited in range. In addition, to give a glow effect to the moonlight sword when it comes out of the chest, I created other two point lights with smaller range, one in front of the sword and one behind it, that turn on (using the "setEnabled" method) only when the player opens the treasure chest.

To generate shadows from these lights, following again the official *Babylon.js* documentation, I had to create a "ShadowGenerator" object (specifying the dimension of the texture used to draw the shadow as parameter) for each light that could project a shadow and add to it, with the "addShadowCaster" method, every mesh that I chose to have a shadow. Moreover, to obtain smoother shadows, I also set the Poisson sampling option to "true".

As an aesthetic addition, I also created two particle systems, using a flare image as texture: one for the treasure chest, which starts when the chest appears and stops when the player opens it (it is used also for user friendliness to indicate where the chest is after the player has beaten the boss); a second one that is used to emphasize the healing effect when the player uses a healing herb and whose emitter is updated at every frame to match the player's chest position. For every initialized particle system, various parameters are modified, like the minimum and maximum emitter box, the emission rate, two directions, the gravity and others.

Lastly, also a subtle fog effect is implemented by setting the fog mode (exponential, in this case), the fog's density and the fog's color properties inside the "scene" variable.

4.3. "Combat", arena and navigation hitboxes

One of the most important part of the game interactions is the set of hitboxes inserted in the scene, i.e. invisible boxes that represent an interaction or a limitation around another object. First of all, we can differentiate the hitboxes inside the project into two categories: the "combat" hitboxes and the arena/navigation hitboxes (both visible in wireframe visualization in *Figure 6*). The first category, already mentioned in the gameplay chapter, is composed by the body and the sword hitboxes of both the player and the boss; they are used to compute, through animation events, if an intersection between a sword and a body happens when the player or the boss is attacking and they consist of two boxes for each mesh, one parented to the "chest" bone and one to the "sword" bone (adjusted manually to fit the blade of the sword). The second category includes the walls and columns hitboxes (which constitute the limits of the arena), the treasure chest hitbox and the player's "navigation" hitboxes; in particular, I designed these last ones because, after several experiments with the body hitbox only, I noticed that the player's character got stuck every time it touched an arena hitbox, since the function used to compute the intersection ("intersectsMesh"), after the mesh intersected even only one pixel with the hitbox, continued to return "true" and so the character could not get out of it; therefore, I created four smaller hitboxes for the player's mesh, one for cardinal direction and parented to the "main" bone, which are checked (at every frame) inside the main render loop to see if the navigation hitbox corresponding to the current movement key the player is pressing is intersected with one of the arena hitboxes or the boss's body hitbox (using the support function "playerCanMove") and, in negative case, the "main" bone animatable is

Figure 6: View with all hitboxes shown (except the treasure chest one)

paused, otherwise it means that the current direction is free and the character can move, restarting the animatable. All these hitboxes are generated in adequate functions, inside the "gameHitboxes.js" file, and returned as arrays (excluding the one for the treasure chest which is returned as a single constant variable).

4.4. Sounds

The application hosts a wide range of sounds, not only for ambience and musical purposes, but also for gameplay purposes. In fact, I included three soundtracks, one for the battle (launched in autoplay when the game is loaded), one for the player's defeat and one for the player's victory, but also sword swing sounds (with multiple choices for each attack), sword hit sounds, footsteps and dodge sounds, boss's scream sounds and a treasure chest opening sound. Every sound is initialized by creating a new "Sound" object, specifying the path of the sound file, the scene and the options (including the loop, the autoplay, which is set only for the main soundtrack, and the volume), and played later when needed; in particular, every non-musical sound effect reproduction is controlled by animation events: without listing every sound trigger, for example, the footsteps sound effect is triggered in two events (the walk cycle include two steps) at certain frames of the four walk animations (the events are attached to the upper leg bone only because they can't be attached to animation groups) and, inside the events, not to annoy the player with too many repetitive sounds, a random number is generated and, depending on the result, a sound among a group of three (with the same probability) is chosen and reproduced; the same process is followed for all the swing, hit, dodge and scream sound effects; instead, the treasure chest sound effect is simply played when the player interacts with it, without a specific animation event. Finally, I want to point out that some of the boss's scream sounds are cut and elaborated (I also added some reverb and delay to the death sounds) by me inside another program called *Ableton 10 Lite*, which is a DAW ("Digital Audio Workstation") useful to do music composition.

4.5. Graphic User Interface (GUI)

To present to the player all the information that he/she needs to know while playing, I implemented a GUI ("Graphic User Interface"), sometimes called "HUD" in the gaming jargon, to show the player's health bar, the player's stamina bar, the boss's health bar, the player's currently available healing herbs and different messages on screen; in particular, these last ones include the player's death message, the victory message and the final message when the player opens the treasure chest.

To create a 2D GUI in *Babylon.js*, again according to the official documentation, first I had to create a special texture, an advanced dynamic texture, by using the helper method "CreateFullscreenUI", to make the program render a 2D GUI on its own layer in fullscreen mode. Then, I designed the GUI to be organized in various containers, exploiting the

"BABYLON.GUI.Rectangle" class to create generic rectangles. In particular, also the structure of the GUI is a tree: for example, the player's bars are contained inside a larger invisible container, which again holds two containers that represent the empty health and stamina bars and, finally, these last ones contains the actual coloured rectangle that changes in size according to the current health and stamina (since they are aligned on the left, they deplete from right to left); also the other bars and other parts of the GUI have a similar structure and are dynamically modified (in size or in content of text) according to the same animation events in which the respective health, stamina or healing herbs values are changed. After their initialization, the elements of the GUI can be attached to their respective containers through the "addControl" function. On the other hand, to create text areas in the GUI, I used the "TextBlock" class, setting the right font family (I mainly used "Palatino Linotype") and size; I want to point out that every text area is not attached to its container (they are attached directly to the main advanced dynamic texture) only because I gave a transparency effect to the containers by modifying the alpha component and elements inherit the values of the properties from the parents, so, if I attached to them, also the text areas would have been half-transparent (an example is in *Figure 7*). Moreover, of course, I modified every element of the GUI in width, height, alignment and colors to match the screen dimensions and the look of the game.

Finally, I implemented also some simple animations for the GUI, which are contained in the "GUIAnimations.js" file. These include the defeat message box fade-in animations and the victory message box fade-in and fade-out animation; the animated property to create the effect of fade-in or fade-out is the "alpha" of the respective elements, which are passed as parameters to the functions; here as well, I grouped the animations into animation groups inside their respective functions to simplify their reproduction (then launched in specific event listeners).. In conclusion, for the defeat and the final messages, I created two buttons (using the "CreateSimpleButton" function) to restart the game; their listeners are located in the GUI part of event listeners inside the main game file and use the "onPointerClickObservable"

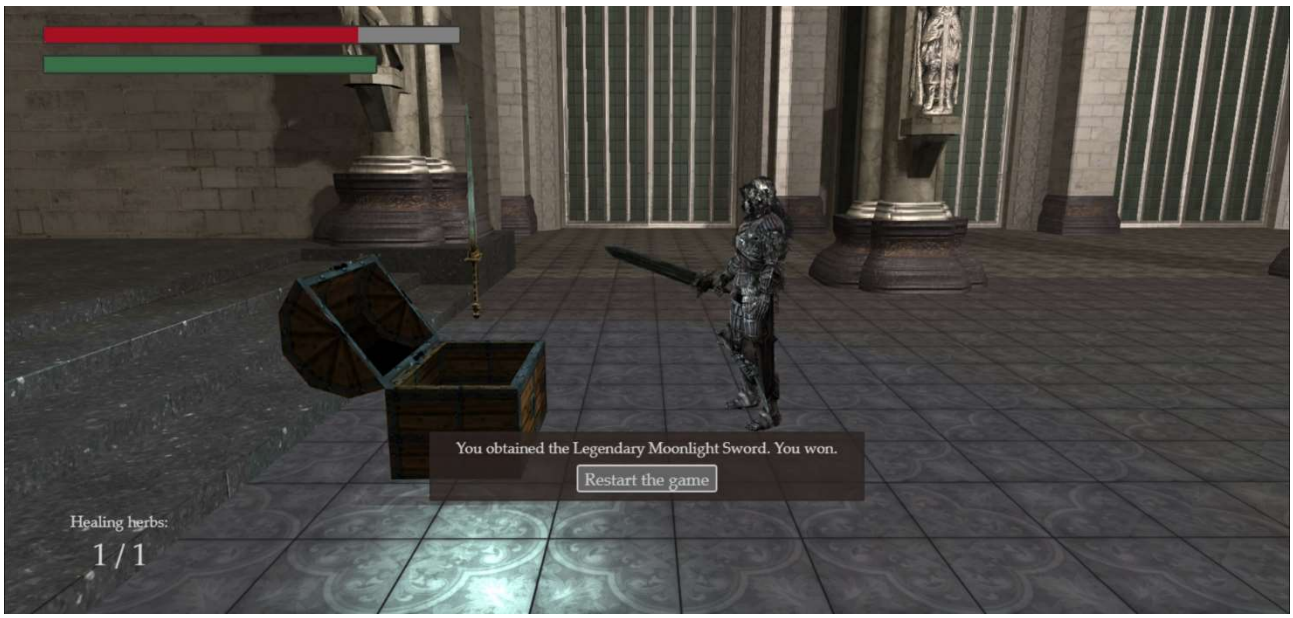

Figure 7: Final message with half-transparent container

observable to interact with it; inside the listener, the *JavaScript* function "location.reload()" is used to reload the current page.

5. Conclusion

5.1 Possible things to improve in the future

Just as an estimation, as a solo project, the presented game was developed in about two months, including the time to learn the *Babylon.js* library and the *Blender* environment (even if I already knew it a bit). In the development, many ideas came in my mind, but some of them, due to the time limit and to keep the project simple enough to debug it, were not implemented. Surely, some possible features to include in the future could be: smoother and better animations for characters models; more animations in general, maybe including magic attacks or boss's long range attacks; a true path finding mechanism for the boss; the possibility to change weapon or choose different character classes at the beginning of the game; a main menu to modify the settings and start the game; expand the environment and the exploration of the map, also giving the possibility to move on stairs and change room or building. So, in conclusion, even if these ideas were not implemented, I hope this final version of the project will be enjoyable anyway.

5.2 Credits

I especially thank the Videogame Resource community ("models-resource.com" in particular), from which I downloaded all the raw models. Then, I also thank the original owners of the assets, so I thank From Software and Bandai Namco for the various "Dark Souls" models, Nintendo for the chest model from "The Legend of Zelda: Ocarina of Time", Square Enix for the other chest model from "Final Fantasy X"; for the sound assets, I thank again From Software for the Twin Princes bossfight soundtrack from "Dark Souls 3" (author: Yuka Kitamura) and the "Gwynevere, Princess of Sunlight" theme from "Dark Souls" (author: Motoi Sakuraba), again Bandai Namco for the "Mission Failed" music from "Ace Combat 7: Skies Unknown", Bethesda Game Studios for the footsteps and weapons sound effects from "The Elder Scrolls V: Skyrim", Capcom for the scream sound effects from the Cavaliere Angelo bossfight from "Devil May Cry 5" and Level-5 for the treasure chest opening sound taken from "Dark Cloud".