Final Project of Interactive Graphics

Adriano Pacciarelli 1816493
Fabio Scaparro 1834913

July 2022

Contents

1 Introduction

1.1 Game

The project consists in creating a game with 3D engines based on WebGL, which we chose to be Babylon.js. Our game is called **Spectre Survivor** has been a reinterpretation of the classic arcade shooter based on zombie/ghost invasion, with a spherical map inspired on the well known video game Nintendo *Mario Galaxy* (Figure 1).
The player plays the role of a sole survivor and has to defend himself and his house form a horde of ghosts coming form a red crystal placed outside the barrier of his property. He is provided with three lives at the beginning of the game, he can shoot enemies with a gun and he can walk in every part of the planet.

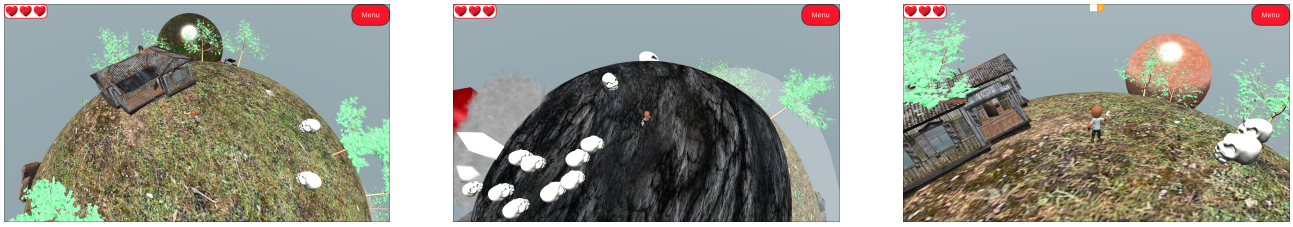

Figure 1: Game playing

At the same time, ghosts will endlessly spawn from the other side of the map, increasing frequency and number while the game goes on in time. Their physical capabilities will depend on the difficulty level inserted at the beginning of the game and their are programmed to aim for the player's lives ether damaging him directly or by infesting the house.
In conclusion, the goal of the game would be to survive as long as possible with the three lives provided at the beginning.

1.2 BabylonJS

For this project we used the BabylonJS framework. It is a web rendering engine that simplifies the process of developing games and graphical applications by providing powerful abstractions to the rendering pipeline and easing many tedious tasks such as writing shaders. It also provides many useful tools that we used extensively to speed up the process of creating the game:

- **Inspector**: GUI tool that can be enabled on every BabylonJS program that allows to inspect the properties of any object in the scene and also interacting with it by moving and rotating meshes and starting animations. It still does not support some useful things such as instancing and it's quite confusing in some regards like how it handles rotations of meshes, but many of its shortcomings can be fixed by using it jointly with the console of the browser.
- **Sandbox**: No more than a simple BabylonJS program that allows to import 3D models to inspect them using the Inspector defined above. It has been used to check the state of the models we downloaded and how they would get imported into the applications.
- **Playground**: BabylonJS lets you write scenes on their servers to quickly prototype small applications to test functions such as camera movements, lighting and transformations. You're just required to write a *createScene* function and the Playground takes care of the rest. Each playground gets an ID that can be used to reference the scene on the internet, to get help online or to show off a project. The most important feature of the Playground is its integration with the API docs: each function description has a link to query the playground server for snippets where that function was used. We found that this feature is invaluable when first approaching a framework like BabylonJS. We found that despite being newer than its alternatives, BabylonJS has a very detailed documentation, which not only includes the API but many use case examples and tutorials.

One thing we found very confusing was the co-existence of right handed and left handed frames in the same scene, and how orientation parametrization is handled. Apparently, you can either specify Euler angles or quaternions for a given mesh, and then you're bound to use that parametrization and the other one is deactivated. The problem is that many of the function calls only work in one of the two ways and in the end we realized that quaternions are always used for the actual computations so we do not understand the need for this separation. We did not use any physics engine.

2 Assets

2.1 3D models

The game contains some 3D models (Figure **??**) available in the internet, reported here with the name, the author and the website:

- **Animated Character 1** (survivor male), from *kenney.nl*.
- **Old House** by *"All4Map"*, from *clara.io*.
- **Evil Skull** by *"leopoly"*, from *clara.io*.
- **Tree 05** by *"Jason Shoumar"*, from *clara.io*.
- **M1911 Handgun** by *"Nathan101"*, from *clara.io*.
- **Agrisphere Stones** by *"sleekinteractive2"*, from *clara.io*.
- **Three crystals** by *"Amirvenom1385"*, from *clara.io*.
- **Crystal** by *"dryangore"*, from *clara.io*.
- **Unknown crystal** by *"ToYu"*, from *clara.io*.
- **Tractor** by *"hegab"*, from *clara.io*.
- **Old Stone** by *"sunraider3"*, from *clara.io*.

Those models have been first uploaded in the game using the *ImportMeshAsync* function from the library *BABYLON.SceneLoader* and disabled. Then instances are created when needed.
Some of the previous models originally contain animations, that we removed to follow the project requirements.

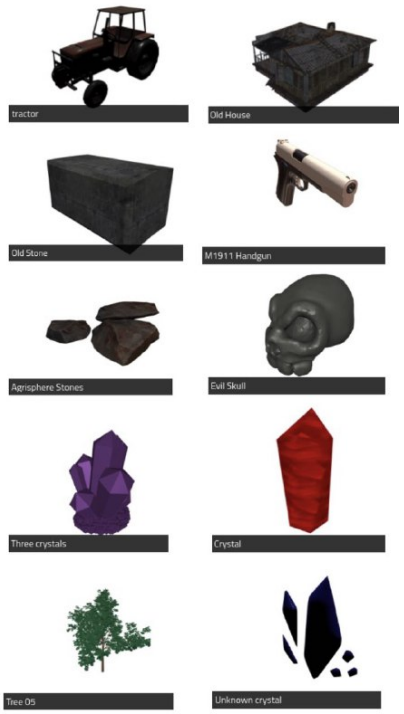

Figure 2: Models

2.2 Sounds

To be more engaging, we added also some sound effects and musics downloaded from the internet website *freesound.org*, that we redport with the name and the author:

- **Human male scream** by *"JohnsonBrandEditing"*.
- **Moaning Ghost** by *"MaxDemianAGL"*.
- **Foot steps Grass** by *"mikaelfernstrom"*.
- **Ghost Appearance** by *"TheRunner01"*.
- **Gun shot** by *"schots"*.
- **Ghost Breath** by *"Argon098"*.
- **Hammer Hitting a Head** by *"WhiteLineFever"*.
- **Metal Song Short** by *"bainmack"*.
- **Suspance Background** by *"Casonika"*.

All those sounds are loaded with the *BABYLON.Sound()* constructor.

2.3 Textures

Several textures have been applied, especially regarding the planet. This last one, has two different ambient textures depending on the hemisphere: grass (Figure 3), for the upper hemisphere and rocky (Figure 5), for the downer hemisphere) and the respective bump textures (Figure 4 and 6).

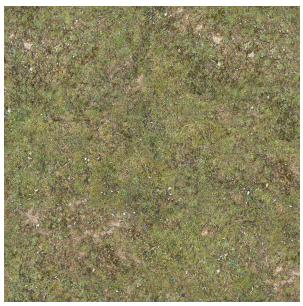

Figure 3: Grass texture

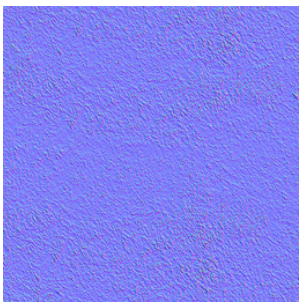

Figure 4: Grass bump tex

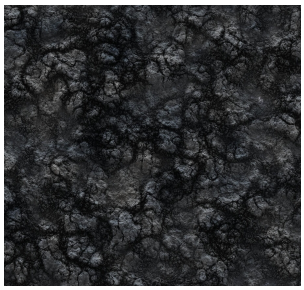

Figure 5: Rock texture

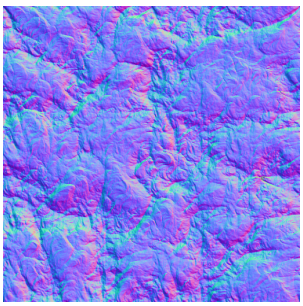

Figure 6: Rock bump tex

Then other textures have been used for the smoke (Figure 10), for the planets (Figure 7 and 8) and for the menu (Figure 9),.

Figure 7: Planet 1 text

Figure 8: Planet 2 text

Figure 9: Menu texture

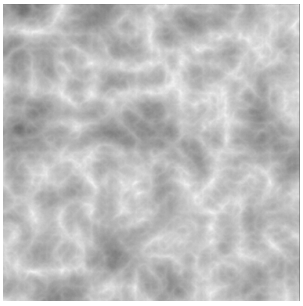

Figure 10: Smoke texture

3 Game components

3.1 Scenes

The structure is subdivided into three scenes, communicating through menu and events.
The first scene is the **Main menu**, presenting a user interface in front and the environment in the background. Then from the main menu you can access the **Game** scene, which contains many different elements, such as the environment, the player, the enemies and the secondary menu. The passage between those two scenes is triggered by the *"Start" button*.
The third and final scene (reachable after loosing in the game scene) is the **Lose menu**, which presents an other user interface in front and the environment in the background. From this interface, you can either go back to the game scene using the *"Yes" button* (for another match) or go to the main menu scene with the *"No" button*.

3.2 Game user interfaces

The game presents multiple game user interfaces (GUIs). Each one of them is realized with the *BABYLON.GUI* library using *"AdvancedDynamicTexture"* objects in *"Fullscreen"* mode, where control objects are added.

Most of the GUIs are employed in the three menus. The first one is the **Main menu** (Figure 11) (in the file "primary-menu.js"), which presents a rectangle as base to which is attached a grid with two columns. The right column is used a sort of "screen" to show the continuity triggered by interface buttons. The left column is attached a panel with four buttons: *"Start"* button to get to the game, *"Rules"* button (Figure 12) to trigger the rules' description, *"Commands"* button (Figure 13) to trigger the list of commands and the *"Settings"* button (Figure 14) to trigger a selection box where you can regulate sound, light and difficulty levels.

Figure 11: Main menu

Figure 12: Rules

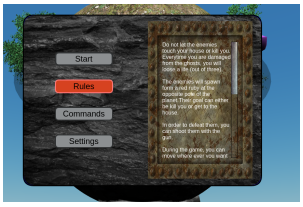

Figure 13: Commands

Figure 14: Settings

The second menu is the **Game menu** (Figure 15) (in "secondary-menu.js"), which is accessible through a *"Menu"* button present in the right top corner in the game scene. It presents a structure similar to the main menu, but with different buttons in the left column: *"Quit"* button (Figure 16) to get back to the main menu (it requires a confirm through a pop menu), *"Commands"* button,*"Settings"* button (Figure 17) and the *"Resume"* button to dispose the menu.
The third menu is the **Death menu** (Figure 18) (in the file "life.js"), which is present in the loss scene. It has a rectangular base, with a grid attached with two rows. The first row presents the score and the second one two buttons to get to the main menu of to the game.
Other GUIs are the life panel in the left-top corner of the game scene and some pop menus.

Figure 15: Main menu

Figure 16: Quit

Figure 17: Settings

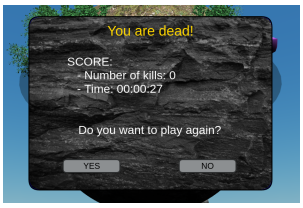

Figure 18: Death menu

3.3 Environment

The map of the game is stored in *environment.js*. First of all, it presents a spherical ground (*planet*) which is parent to everything else. Then, on top of that there are two half spheres representing the two different hemispheres (*Upperworld* and *underworld*) with different textures, representing the 2 sides of the map. The subdivision is present as a semi-transparent spherical disc (*barrier*), with a higher radius in order to be visible.

The Upperworld (Figure 19) presents on the surfaces some objects (models) such as an house, a truck, a stone and several trees which are placed randomly in that side of the map (the implementation will be discussed later).
The Underworld (Figure 20) presents other objects such as skulls, crystals and a *"smoke"* object, implemented as a particle system with a fog texture.
Then other planets (Figure 21) are placed around the ground (still parented to the main planet).

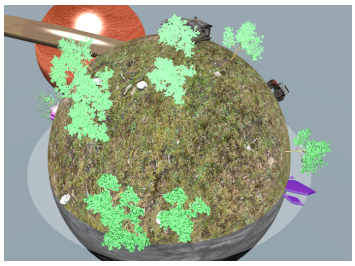

Figure 19: Upperworld

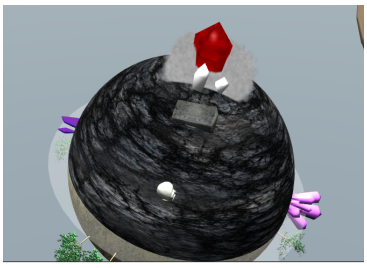

Figure 20: Underworld

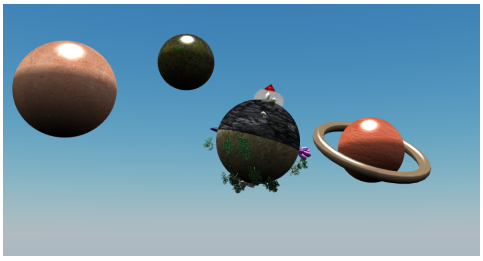

Figure 21: Planets

In addition, sounds are loaded to be played in loop (as for the background *suspense music*) or triggered by either the keyboard (like the *walking sound*) or the mouse (like the *hammering sound*).

3.4 Skybox

The skybox (Figure 22) has been constructed by building a large box around the planet so that the camera is always inside this box.

Then, it as been assigned a particular material called *SkyMaterial*, which has many properties that can be set and customized to allow a different look and feel for the skybox. It is possible to press keys `1` to `4` to change between daytime and night time, although this feature has not been used in the game. This skybox does not actually light the scene but it's just a visual effect, two lights were added in the traditional configuration of **DirectionalLight** plus **HemisphericLight**. The directional light follows the sun movement so when it goes down you just see the small intensity hemispheric light. A possible improvement would be to include a day/night cycle into the gameplay, possibly switching enemy type and difficulty of the game.

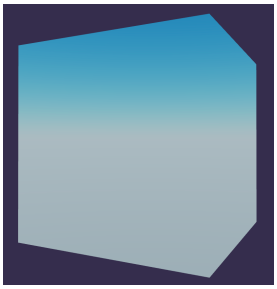

Figure 22: SkyBox

3.5 Animated objects

The animated objects in the game are the player and the enemies, both present only in the game scene.
The player is a hierarchical model, provided with a skeleton inside, which can be fully animated (more details in the next chapter). It will spawned at the beginning of the game in the "Upperworld" and it can be commanded to shoot and walk.

The enemies are instances of a single model. Those will spawn depending on the difficulty level from the "Underworld" top and are provided with a pseudo AI that will bring them to attach either the player or the house (more details in the next chapter).
Naturally, those objects are provided with sounds triggered by keyboard events (like "shooting") or game related events (like "dying").

3.6 Camera

Our framework provides many options for the camera, we chose the **ArcRotateCamera**, which targets a point and moves around it parametrized by spherical coordinates (r, θ, φ). This is especially convenient to use when you intend to move the camera with the mouse, as it basically implements a third person camera. Another option we considered was the **UniversalCamera**, which is the most general one and allows to build custom behaviors. Due to the fact that we fixed the player in place (see technical aspects), we did not benefit from the freedom gained with the universal camera so the arc rotate one was just fine.

4 Technical aspects

4.1 The game

The game has a state, it can be in `MAIN_MENU`, `PLAYING` or `LOST`. Depending on the state, the BabylonJS engine renders a different scene, and upon switching state it ditches the old one and sets up all the assets for the new one. You can transition to every state from each state, except for `MAIN_MENU` to `LOST`, of course. This allows a very clean execution of the game and the possibility to do many runs without refreshing the page.

4.2 Player

The player (Figure 23) moves on the surface of a big sphere that is its planet. The movement scheme is derived from the well known video game series *Mario Galaxy*, which shows similar movement styles.

There are many ways to achieve this effect of moving around a planet, but basically you either keep the planet or the player still and move the other.
The most straightforward approach is to move the player on the planet,and this is what we tried first. It ended up being a poor choice because of the strange behavior of the camera we chose, which is Babylon's **ArcRotateCamera**.

When the player was in the lower hemisphere it would do strange movements to avoid reaching the spherical coordinates limits and it was very unpleasant to play and also very hard to fix. One solution we tried is to parent the player to the camera so it's orientation would remain the same; this implies that every rotation of the player had to be canceled in the camera to keep the user in control of the camera movement and we thought that moving the planet below the player was way easier. In this setup the player always has the same world coordinates and the planet is rotated under his feet. The camera always keeps the same orientation because the player is still and this avoids any locking configuration. The downside of this approach is that when you need to reference the player position (e.g. enemies that track him down) you need to transform its coordinates in the planet's frame, otherwise they would be constant regardless of the player's position on the planet.

Figure 23: Player model

The user can input movements through `WASD` keys and by rotating the camera with the mouse. It can shoot bullets with `SPACEBAR`. The movement of the player follows a simple algorithm. Upon receiving some desired movement input:

- The player orientation is set to point at the forward direction of the camera.
- From this configuration he may turn even more if the requested input includes a strafe component. The right direction of the player is then fetched (its x-axis) as a rotation axis
- A rotation angle is calculated according to the input and the player's velocity constant. The planet is then rotated by (negative) that angle around the rotation axis, applied at it's origin.

The player has **three lives** (Figure 24) . The information about player lives is stored in an instanced class that also records player score and running time.
The player also shoots bullets (Figure 25) from his gun, which are used to kill enemies. Each bullet (Figure 26) travels, like the player, on the surface of the sphere and has a certain lifetime after which it disappears. By tweaking this parameter it is possible to have the bullet make one or more full turns. An interesting modification to the gameplay of the game would be that the player can be hit by his own bullets that come full circle. It would require some changes in the bullet parameters because as of now, we imagined that the bullets are small and fast thus hard to see and avoid.

Figure 24: Lives

Figure 25: Shooting

Figure 26: Bullet

4.3 Collisions

To handle collisions, we used BabylonJS's collision detection system. For each mesh the engine draws a bounding box that is an approximation of the mesh's shape, then you can query two meshes and check if their bounding boxes intercept. Sometimes we had to adjust some of these bounding boxes to have a volume that better represents the mesh. In the case of trees for example the bounding box has been modified to capture only the trunk, not the branches and leaves.

If, after a position update, the player's bounding box were to collide with that of one part of the environment, it's old position is restored so it does not enter into objects. Actually, this caused the player to sometimes be stuck forever into bounding boxes he enters in. We assume it is due to numerical approximations so we don't just rollback the last position but a little bit further back than that, to make sure that the player lands in a free position.

4.4 Enemies

The enemies are floating skulls-ghosts (Figure 27) and there can be two kinds of them: those that track the player and those that track the house. In the unfortunate case that an enemy reaches its target, it will be removed from the game and the player will lose a life.

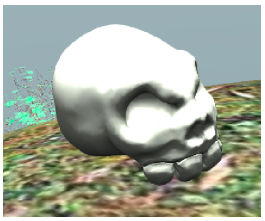

Figure 27: Enemy

Moreover, the house or the player gain some invulnerability time to avoid losing too many lives at once.

The enemy movement is a little bit different from the player's. The enemy is first rotated toward its target then it's translated in its forward direction, which is tangent to the sphere. Finally, it is projected back onto the sphere surface by scaling its position vector so that is has length equal to the radius of the planet and it is reoriented to be normal to the sphere. This is visually the same as the player movement and we kept both styles to outline how it is possible to implement movement in many ways.

Enemies spawn in hordes; the number of enemies that spawn at each horde is function of time passed and game difficulty. This is the same for the interval between hordes, which decreases as time passes to make the game harder, and for the speed of the enemies, which is constant throughout the game but depends on the difficulty.

4.5 Animations

The player has three kinds of animations: one when he's **idle**, one when he's **walking** and one when he's **hit**. The hierarchical model of the player has many joints so we decided to animate only some of them, mostly legs, arms and neck. For each joint you have to define a set of keyframes (Figure: 28) to put into the animation. This results in many animations and not a clear way to synchronize them. Luckily BabylonJS provides an

Figure 28: Walking frames

AnimationGroup class that allows to group many animations and start and stop them all together. So we defined three animation groups for the three kinds of animations, and each time we start the necessary animation group. The way this works in practice is that the transition between animation is not smooth and visually unappealing. To fix this problem we allow all the three animation groups to play together, but we assign a weight to each of them that varies according to the player situation. As an example when the player is still the only non zero weight would be on the idle group. But when the user issues an input we smoothly change the weights so that the walking group takes over and this results in a smooth interpolation of the player hierarchical configuration.

The enemy also has a small animations which is just bobbing up and down like a ghost would. To accomplish a ghostly looking animation we have used **Easing functions**. Those are a special class of functions that enable

certain behaviors when interpolating keyframes such as manipulating the tangents to make the animation have a certain shape. In this particular case I have used a *Quadratic Ease In/Out* function that allowed the enemy to have continuous smooth tangents all across the animation and achieve a *floaty* feel.

4.6 Instances

BabylonJS provides a powerful interface for instancing meshes. Instanced meshes point to the same assets but have different world matrices among themselves. This allows to have many of the same objects in a scene without loading the asset many times. We have used these functions to scatter trees (Figure: 29) and other objects on the planet, by writing a random scattered function:

$$\varphi = rand(0, \pi)$$

$$\theta = rand(0, \pi) - \pi/2$$

$$p(\varphi, \theta) = \begin{pmatrix} R\sin\varphi\cos\theta \\ R\sin\varphi\sin\theta \\ R\cos\varphi \end{pmatrix}$$

where R is the radius of the planet. Their orientation has then been fixed by providing the columns of the rotation matrix in this way:

$$v = randVec3()$$

$$\hat{y} = normalize(obj.position)$$

$$\hat{x} = \hat{y} \times v$$

$$\hat{z} = \hat{y} \times \hat{x}$$

We just care that the object is normal to the sphere so the $\hat{x}$ and $\hat{z}$ versors do not matter that much and it's also better if they're chosen according to a random factor.

The θ interval can be shaped to target different portions of the planet, for example we only spawned trees in the upperworld and (fixed) skulls in the underworld (lower hemisphere).

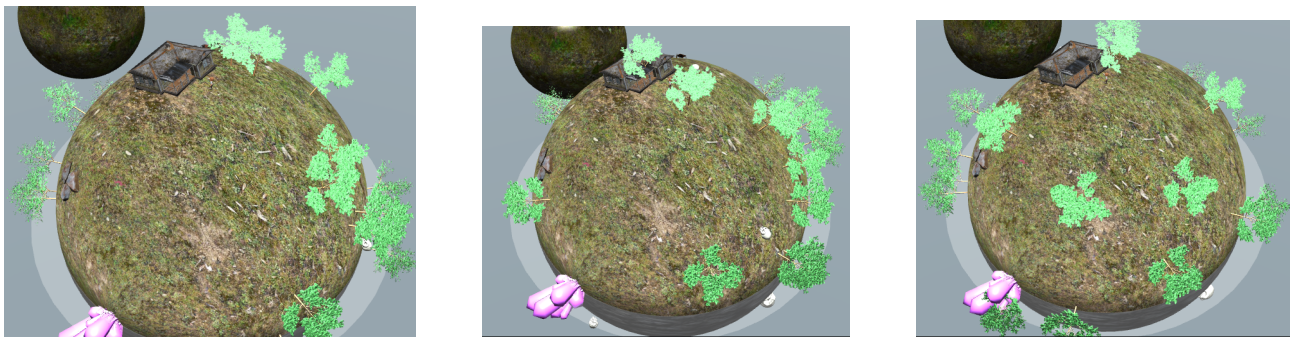

Figure 29: Random trees spawning in different places of the planet