Interactive Graphics Final Project:
BabylonJS Mario Kart

Leandro Maglianella – 1792507

Lorenzo Nicoletti – 1797464

Contents

1. Introduction 3

2. Framework 4

3. Implementation 5

3.1. Camera 5

3.2. Light 5

3.3. How we render a scene 5

3.4. Overview on models, textures and sounds 6

3.5. Menu organization and flow of information 6

4. Generic race development 8

4.1. How the race is organized 8

4.2. Item acquisition and effects on the vehicles 8

4.3. Motion of the CPU cars 9

4.4. Events in a race: overtakes, spins, updating of information and much more 10

5. Specific race designs 12

5.1. Luigi Circuit 12

5.2. Bowser’s Castle 13

5.3. Rainbow Road 14

6. User manual 16

7. Conclusions 17

1. Introduction

The idea of our project takes inspiration of one of the most famous racing games of the *Nintendo* universe: *Mario Kart*. The original videogame is extremely complex, with lots of options that the user can set, including as instance character selection, race and difficulty modalities and so on. Our implementation is a simplification without, however, moving too far away from the original product of Nintendo: our objective is to provide a game that is easily controlled by the user but, at the same time, that keeps the realism of a race, with events like overtakes, spins, out-of-the-street situations and much more. We wanted also to recall the 'true' Mario Kart, therefore, we adopted original models, textures and even sounds in order to satisfy our prerequisite to be faithful to the real game. All the models have been downloaded from *Sketchfab*[1], a platform where 3D models are published and shared, in a GLTF format and straightforwardly imported in our scene. The textures are directly taken from the 'Mario Kart Wii' section of "The Textures Resource" website[2], while the sounds are mainly imported from "The Mushroom Kingdom" inventory[3].

The initial difficulties this project has faced were both general and specific: from the requirements of choosing the best graphic framework for this specific game and how to start from scratch an idea that seemed to be wide and extensive to questions like how to render specific situations that can occur in a car race. This report aims to exhaustively explain all the implementation details, analysing the advantages and disadvantages of our approach in response to the encountered difficulties in a 'top-down' fashion, we will analyse the general complications first and the specific applications later.

[1] Official website available at https://sketchfab.com/.
[2] Available at https://www.textures-resource.com/wii/mariokartwii/.
[3] Available at https://themushroomkingdom.net/media/mk64/wav.

2. Framework

Figure 1: Our game logo, formed by merging together BabylonJS and Mario Kart logos

The choice of the most suitable framework has required a preliminary phase of analysis among the suggested libraries for the project. Reading their documentations, we found out that *BabylonJS* (version 4.2.0) was the best compromise between an easy-to-use environment and the correct way to realize our ideas. BabylonJS is a real time 3D engine written in JavaScript and combined with HTML for displaying graphics in a web browser. We noticed also the huge online support, with lots of playgrounds (instances of BabylonJS application accessible through their website) to practically understand how things work with this library and active forums with conversations that was useful for our difficulties too. The starting point was, therefore, interacting with the first chapters of the wide documentation of BabylonJS and gradually beginning to autonomously work on our specific idea.

The first design choice we have taken has concerned the organization of a web page: instead of splitting it in a HTML and a JS file like we have done with *WebGL* until now, we have found more useful to integrate all the code related to a page in a single self-contained file, structured like a HTML box directly containing the related JS script therein. This different approach has allowed a smoother organization of the work and a better exchanging between pages.

3. Implementation

3.1. Camera

BabylonJS provides a wide set of cameras that the user can choose according to his objectives in the application. We have decided to adopt the so-called *FollowCamera*, that takes a mesh as a target and follows it as that mesh moves. This is a particularly reasonable option if we consider the nature of our application: a user-controlled car that moves on the street following the track; in this way, the experiencer can easily understand where the car is and which type of action it should do to proceed racing.

3.2. Light

As for the cameras, BabylonJS has several options for lights too. We will analyse in detail how lights are handled for each racetrack in the following sections of this report; however, we can anticipate that there is always a 'main' light that gives the ambient illumination: a BabylonJS *HemisphericLight* is the best and easiest way to simulate an environment light by simply specifying the light reflection direction (not the incoming direction). Usually it is towards the sky ((0, 1, 0) in 3D *xyz* coordinates) in an open environment but we proposed some exceptions depending on the specific scene we are rendering (please, refer to the Chapter 5: *Specific level designs* section below for further information). Moreover, the properties of this type of light are *diffuse*, *specular* (shared among all the other BabylonJS lights), to indicate the amount of light going from the centre of an object to the direction specified above, and *groundColor*, that relates the light in the opposite direction. The choice of these parameters is specific of each racetrack and it will be later discussed in the appropriate paragraphs.

One drawback of this type of light consists in its impossibility to cast shadows, making shadowing generation actually unfeasible.

3.3. How we render a scene

A car race is a dynamic situation where vehicles are always moving with accelerations, decelerations, multiple overtakes, spins if a Mario Kart item hits them and so on. All these situations require a time-dependant evolution of the whole scene, therefore we decided to render separately the static and the dynamic parts of the environment. In particular, the static characteristics are immediately created right after the webpage is correctly loaded: these include some elements like the ground and the street, the models of the cars (that will be later animated) and other (static) aesthetic components for the scene. Once this first rendering operation is completed, the 'dynamical' routine can begin when the start-race sound is played: a countdown is shown in the middle of the page to indicate the launching of the race. The dynamic render loop is based on the *registerBeforeRender* function of BabylonJS to automatically catch the evolution of the competition just before a frame is rendered; in its scope we update variables, we handle collisions with items or item boxes, walls, end-of-the-world boundaries, we update information such as number of laps and position with respect to the other vehicles and we focus on those situations that regulate a realistic development of a race. When the user car completes

the final lap, this second render loop is stopped, with all its animations, and the game redirects to the successive webpage of the application.

3.4. Overview on models, textures and sounds

All the models adopted in this project are characterized by complex hierarchical structures where meshes are formed by elements where child or sibling relations are established between each other. The animations fully exploit all the advantages of the hierarchical modelling: if an object is required to move, this motion is provided by controlling the root mesh only which apply a consequent motion to the other parts of the body as well[4]. In this way, all the models belonging to the 'dynamic' part of a race are easily regulated through their parent nodes in the hierarchy. Another consistent advantage of the adopted models is that they already contain all the information about lighting: each mesh has a predefined material which specifies its light properties with elements like *baseColorFactor*, *emissiveFactor*, *metallicFactor*, *roughnessFactor* and so on, making the illumination of a model easy and automatic.

The (original) textures used in the game are mainly of the *diffuseTexture* type because of the poor availability of the other texture categories (*specularTexture*, *bumpTexture*, etc.). The texture is rendered by attaching it to the correspondent BabylonJS *Material*, which is in turn attached to the correspondent mesh. This is a recurrent and easy routine to instantiate a model with an imported texture.

Finally, the sounds are very easily loaded and played through the BabylonJS *Sound* constructor: a *Sound* instance can be processed with useful methods to play, pause, restart and stop the related sound file.

3.5. Menu organization and flow of information

In addition to the game, various other HTML pages which the user can interact with have been implemented. As soon as our website is launched, a *Main Menu* is shown from which the user can decide whether to start playing or whether to access the *Game Controls* page, the *Credits* page or to mute/unmute the sounds. The *Game Controls* page (Figure 3) consists of a screen comprising various Super Mario style images in which our beloved protagonist informs the player about the keys to use during the game (for more information, see Chapter 6: *User Manual*). The *Credits* page (Figure 2) is

Figure 2: Credits page

[4] Regarding cars, we have tried even more complex animations involving rotation of the tires too. However, the predefined organization of the imported models makes the handling of each tire impossible because they group all the four tires as a single mesh. Consequently, a rotation to simulate their movement resulted unrealistic. We have therefore decided to rely on simpler but more close-to-reality animations.

a simple screen with the *Sapienza* logo as a background where the user can be directed to the *LinkedIn* profiles of us two creators of the game by clicking on the right buttons.

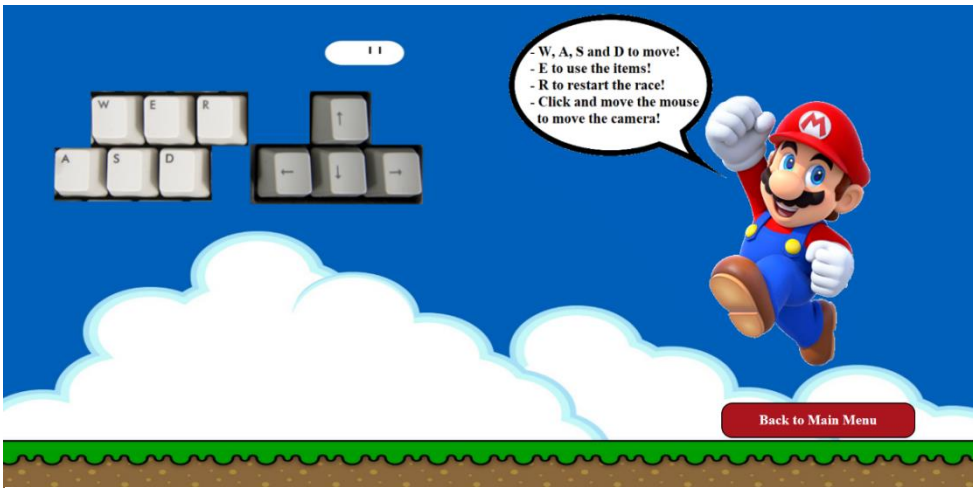

Figure 3: Game Controls page

Instead, by deciding to start playing, the users are directed to the *Game Selection Menu* (Figure 4) where they can select which track they want to play in (or play them all in a *Grand Prix*) and which character they want to use among the available ones (Figure 10). From all these secondary pages, you can conveniently return to the Main Menu with the appropriate button. In all of the webpages the buttons have been programmed to play a sound when the user passes over them with the cursor and for their text to change colour when they are selected: this was done to give visual and sound feedbacks to the users regarding the reception of their choices. Furthermore, in the Game Selection page the buttons of the tracks have customized backgrounds and the buttons of the characters reproduce specific sounds belonging to the chosen character. Starting the race, you are redirected to the real game: the user's choices can be passed between the different HTML pages by inserting them in the link as variables which are retrieved in the destination website by manipulating the link string; this mechanism is also used to store the player's score in the Grand Prix mode. This flow of information technique, although it has its strengths in the effectiveness and simplicity of implementation, is not a very safe way of communicating on the web; however, in this case we can adopt this solution as it is a simple game and therefore not an important application that requires a careful data protection and processing.

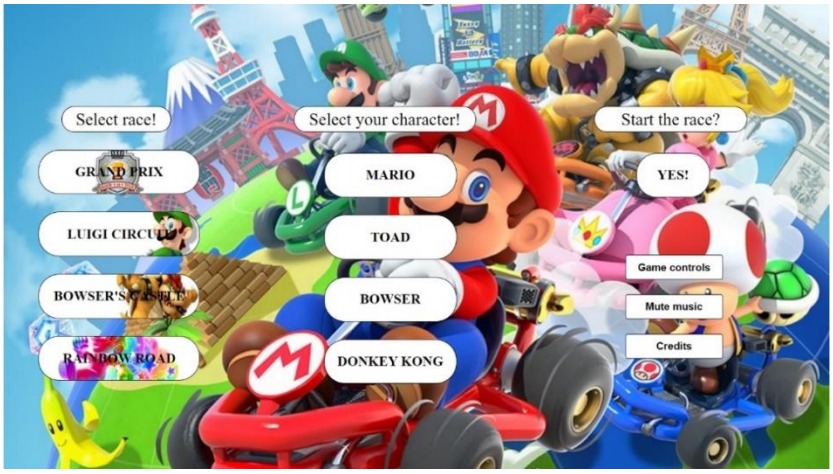

Figure 4: Game Selection Menu

4. Generic race development

4.1. How the race is organized

The general structure of the races that we have implemented starts with a screen where a welcome button is shown, which when pressed starts the competition: this is announced by a classic Mario Kart sound followed by a countdown. This button in particular has a dual purpose in this case since it both delays the start of the race, thus allowing the scene to load completely in the meantime, and both forces the player to click the button which, being part of the scene, subsequently allows the acquisition of game inputs. At the end of the countdown all the cars start to move at a base speed; the player's goal is to overtake the cars driven by the CPU and to follow the path by accelerating, decelerating and turning appropriately since leaving the path would entail penalties, which differ according to the game track.

The race ends when the player finishes his third lap and his final position determines the score he obtained, which is shown on screen and added to the scores of any previous races. Only the *Grand Prix* mode in particular ends when the user has completed all the three races of the game and is finally redirected to the *Prix Results* page (Figure 5) where his victory is announced, accompanied by a trophy and the total score obtained.

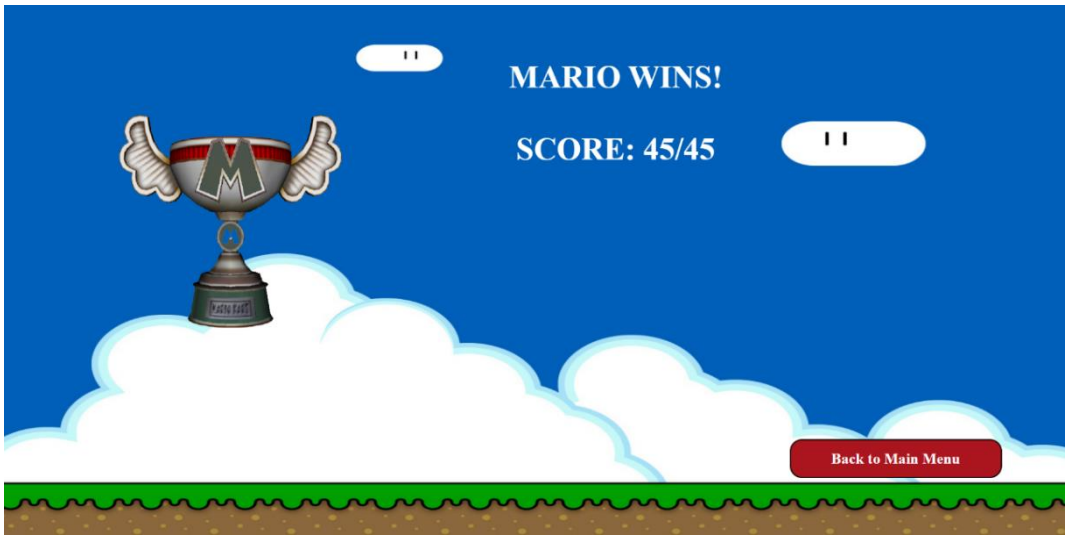

Figure 5: Prix Results page

4.2. Item acquisition and effects on the vehicles

The official Mario Kart platform contains a huge set of items that the user can use in the original game. We have restricted this set to four elements only, deciding to import only the most known items that did not require too many difficulties with reference to their implementation. Therefore, the available items are:

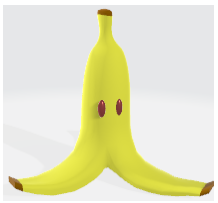

Figure 6: banana item

- **Banana,** a banana peel (Figure 6) that can be thrown by the player behind his car and causes a spin to the car which passes over it. After this effect, the banana is destroyed from the racetrack.

- **Green shell,** a green-coloured turtle shell (Figure 7) that can be thrown on the track. This object follows a straight-line trajectory with constant velocity (but higher than car velocities) and hits an eventual intercepted car. After this effect, the green shell is destroyed.

Figure 7: green shell item

- **Red shell,** in the original literature of Mario Kart, the red shell (Figure 8) is a powerful item that follows the enemy car in front on the user until it hits its target. We have strongly simplified such behaviour: in our game, the red shell aligns to the target car first and then follows a straight-line trajectory like the green shell. In this way, the probability of an enemy-hit event is much higher; however, there are cases where the correct execution of this pattern is not always realized due to misalignment in the user car and target car orientations. After this effect, the red shell is destroyed.

Figure 8: red shell item

- **Mushroom,** the power-up mushroom (Figure 9) is the only non-lethal item in our environment. If used, it provides to the user a huge sprint that rapidly increase the car velocity. This sprint gradually reduces making the effect slowly disappear after some seconds. Once the mushroom is used and the sprint starts, the item is destroyed from the scene.

Figure 9: mushroom item

The interaction with the items has revealed to be much more complex than the expectations. An item is available to the user car only if it crosses over one of the item boxes systematically located on the track. Once the item box is used, it is destroyed from the scene and the user cannot cross it anymore. The selection of the object is randomly taken: each item has an equal probability of 0.25 to be created. If an item is created but still not used, the object follows the user trajectory along the x- and z-axes and is positioned slightly on top of the car (higher value on y-axis). If, moreover, an item is not used and the player crosses another item box, the previous item is no longer available, it becomes a lethal object (or positive, if the dropped item is a mushroom) on the track, causing a spin to the car that will pass over it, and a second item is generated for the player to use.

4.3. Motion of the CPU cars

While the user car is actively moved by the user through proper keyboard buttons, the other three cars are CPU-controlled, meaning that they follow a standard and fixed pattern with the usage of a predefined animation. An animation in BabylonJS is created with the useful keyframing technique: it requires only some specific 'important' frames and the other in-between frames are automatically drawn while executing the animation. In this framework, keyframing is realized with the *setKeys* method of a BabylonJS *Animation*, which takes as input an array of frame-value pairs, defining the key frames, associates them to the animation and automatically creates the in-betweens. Keeping this in mind, a CPU car motion is created by simply considering some points on the racetrack and can be executed by defining a frame rate to regulate the velocity and a *speedRatio* parameter to differentiate the timing among different cars. A final helpful property of Animation consists in the possibility to

pause it, start another animation (for instance, a spin) and then restart it and make the car proceed in the race.

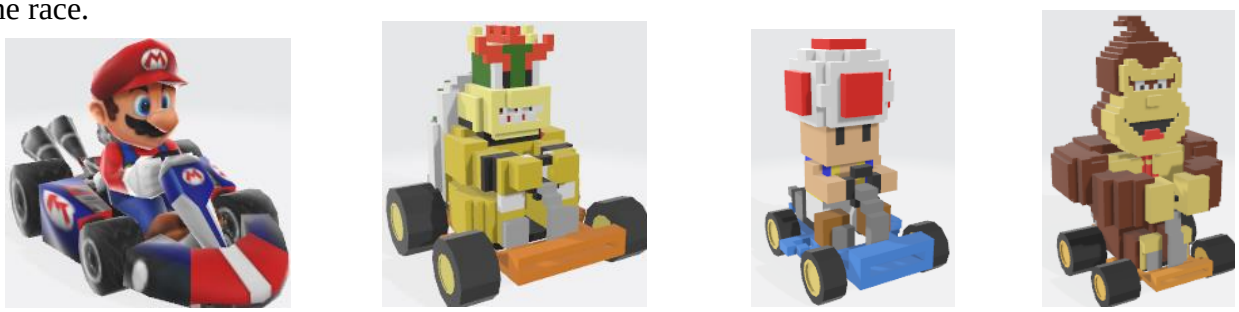

Figure 10: Game's car models

4.4. Events in a race: overtakes, spins, updating of information and much more

The player's car is an active agent within the race and can cause various events with effects on itself and its opponents; let's analyse them all one by one. One of the most important and necessary features in a car racing game is being able to establish the player's position within the track and be able to correctly update it in real time when the player overtakes or is overtaken by an opponent. In our implementation, the logic of overtaking varies according to the player's position: each track is made up of various vertical or horizontal straight road in which the right pace is "direct" or "opposed". Knowing the player's coordinates, we can therefore establish in which straight road he is located and whether there the best position is relative to a greater or lesser x or z coordinate. In every track the straight roads are found in different positions from each other and, above all, the *Bowser's Castle* has a more complex shape and therefore the overtaking was obligatorily made slightly differently with respect to the other tracks. In addition, the situation in which the player (or opponents) laps the other cars was also considered: the position is updated if and only if the user and the CPU are on the same race lap.

The position information is always visible to the player in the graphical user interface of the game (Figure 11), where the race time and the lap being covered are also shown; the lap's increment was again made in relation to the coordinates of the car: passing through the finish line the lap value increases by one. The GUI also includes a frame per second indicator.

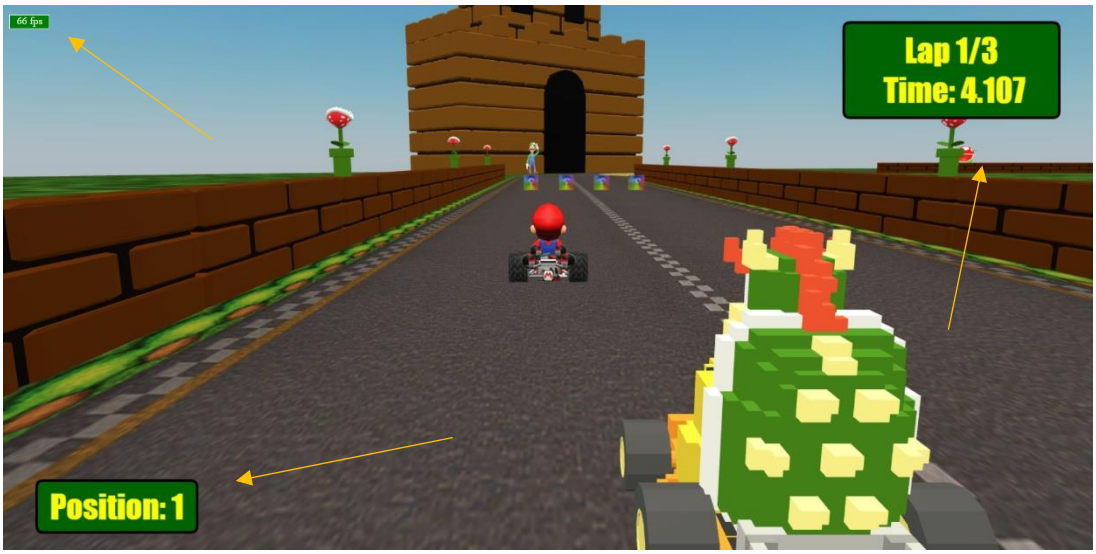

Figure 11: particular of the GUI

To oblige the player to respect the track's path (so that the lap could not increase simply by moving back and forth by the finish line) we have imposed that the increase can only occur if the player has first passed through an area approximately in the middle of the circuit.

As anticipated in the previous sections, when a car collides with a lethal item on the track, it starts a spin animation (Figure 12) in which the car is subjected to two quick full rotations clockwise and counter clockwise and then repositioned in its initial orientation. As for the player's car, we only need to start the animation of the spin when necessary; while instead for all the CPU controlled cars, we first need to pause their motion animations, start the spin and then resume their automatic motion.

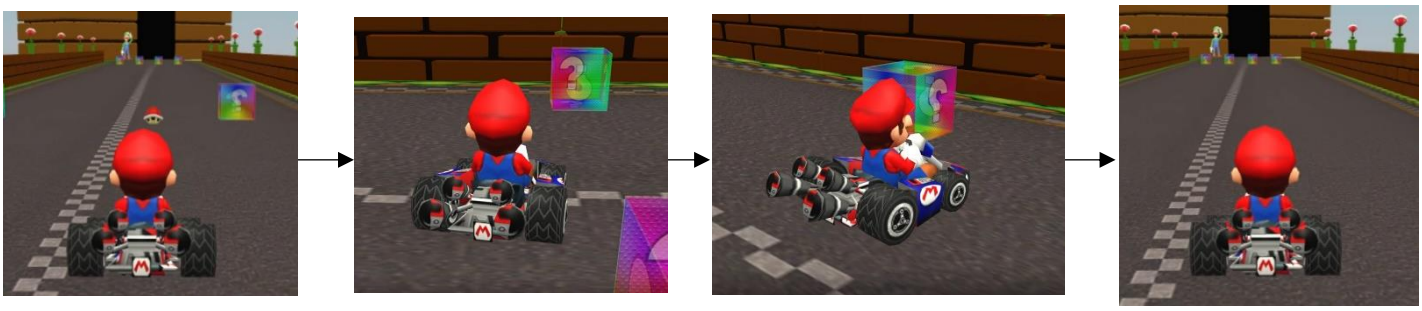

Figure 12: some example frames of spin animation

Many other interactions have been created in the game; we can briefly anticipate them but we will describe them in the next section as they are track-specific. In *Luigi's Circuit*, when exiting the track, the speed of the car decreases due to the greater friction on the grass (or sand). Furthermore, for each track, a particular behaviour has been created which is triggered in the case that the player leaves the viable world: the car can be re-teleported on the track or burn itself in the lava and jump back into the circuit or again fall into sidereal space and then be (luckily) recovered and moved back in the race.

5. Specific race designs

5.1. Luigi Circuit

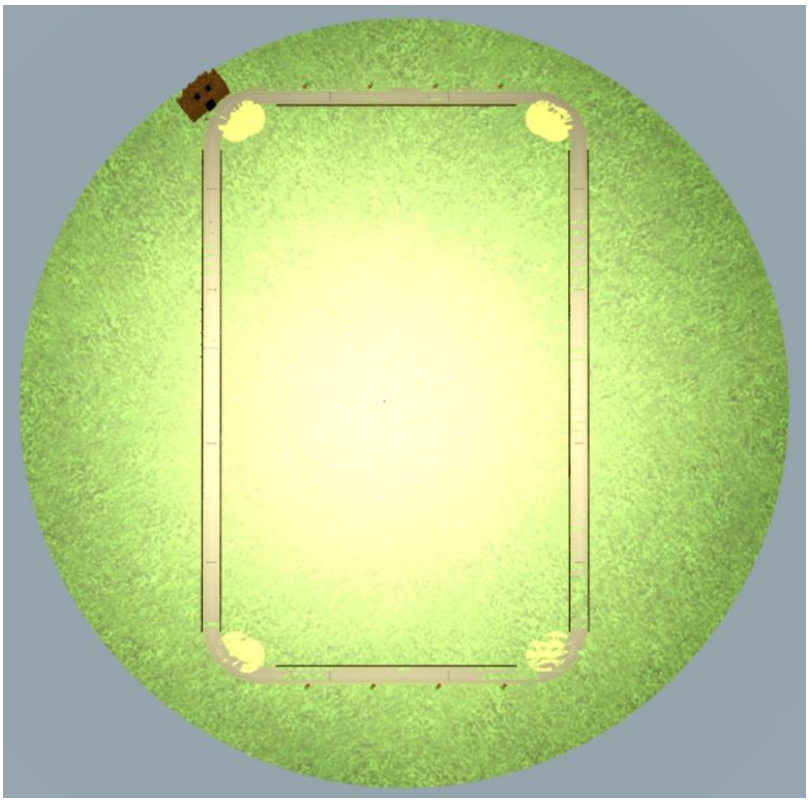

Figure 13: Luigi Circuit seen from above

Luigi Circuit (Figure 13) is the first track that the user faces; therefore, it has been designed to help the player to understand the commands and gain confidence in the proceeding of the game. The race takes place in an open environment where the light simulates a sunny day by properly setting the aforementioned diffuse and specular parameters to mimic a yellowish sunlight. The light intensity is doubled. The open space requires obviously the presence of a sky, realized through the *SkyMaterial* element of BabylonJS and, in particular, setting its inclination (namely the sun inclination) to 0 to give the appearance of a clean sky with no clouds. The other aspects of the scene involve a circular ground representing the grass, the street and some aesthetic models to improve the environment look, like the piranha plants, the bleachers, Luigi and his castle.

The peculiarities of this circuit mainly rely on how the out-of-the-street events are handled. During the race, the user may notice that the road can be bounded by walls, as sequences of brick blocks, or by natural elements like grass or sand. To improve the realism, we have introduced the concept of friction: if the car gets out of the street and moves on the grass or on the sand, its velocity will consequently decrease, making its motion consistently slower and suggesting the user to try to go back to the street path. On the other hand, when the car hits a wall around the street, we have implemented a collision-aware system that simply inverts the orientation angle of the vehicle by an angle specular to its current rotation (Figure 14), similar to the behaviour of a ray that hits a mirror and is symmetrically reflected along its axis. Finally, if the user tries to leave the viable world (the grass circle), he is sent back in a position inside the track: this position varies based on whether the player reached the half lap area or not (Figure 15).

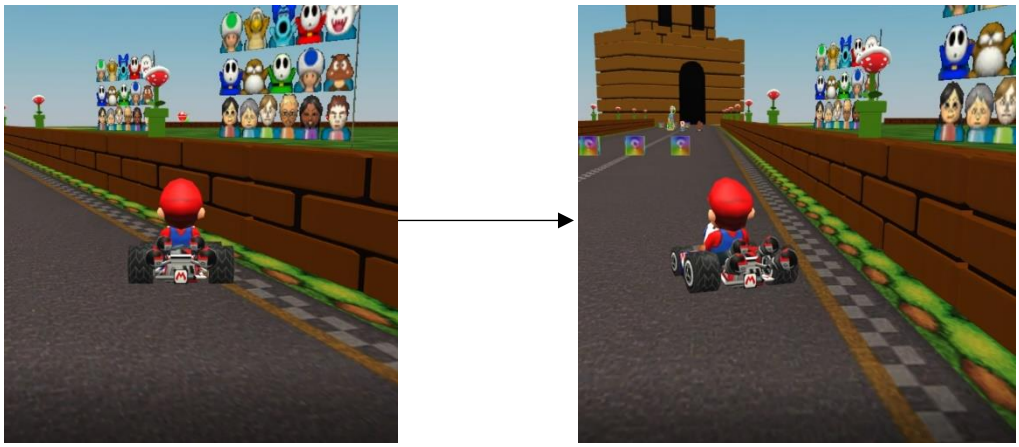

Figure 14: wall collision

Figure 15: out-of-the-street routine

5.2. Bowser's Castle

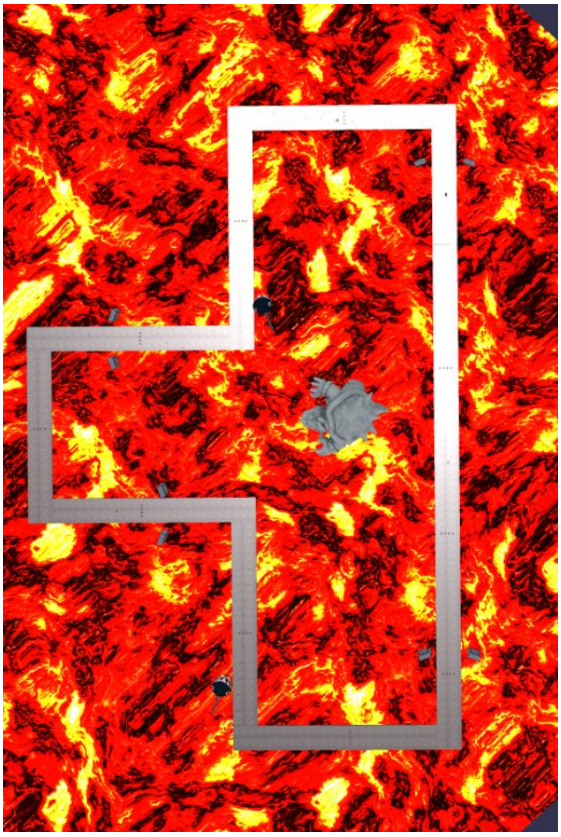

Figure 16: Bowser's Castle seen from above

When the user enters in the Bowser's Castle world (Figure 16), the first noticeable thing is the lava/magma movement under the street. This effect has been created assigning to the world's disc the special BabylonJS's material *LavaMaterial* which uses two textures: a diffuse texture gives the colour of our lava while a noise texture represents the lava deformation applied to the mesh. We have also set the lava's *speed* and *fogColor* to recreate a realistic behaviour. We have imagined this world like a closed environment: the racers are inside the castle of the famous villain and, therefore, there is no sky and the light simulates a darker scene with respect to the sunny Luigi Circuit. The hemispheric light source is located on the ceiling of that fictitious castle, rendered as a black semi-sphere placed on top of the magma ground, with diffuse, specular and intensity parameters properly damped. In this track some aesthetic models to improve the environment look have been used as well, like the *Thwomps*, the *Chain chomps* and the giant statue of Bowser.

The second main innovation of the circuit is the implementation of checkpoints, invisible points on the street that we have defined to handle out-of-the-street events. In this new situation, if the user gets out of the road, the car will be burnt by the lava provoking a big jump back to the last checkpoint, with the camera following the action of the 'bounce' (Figure 17).

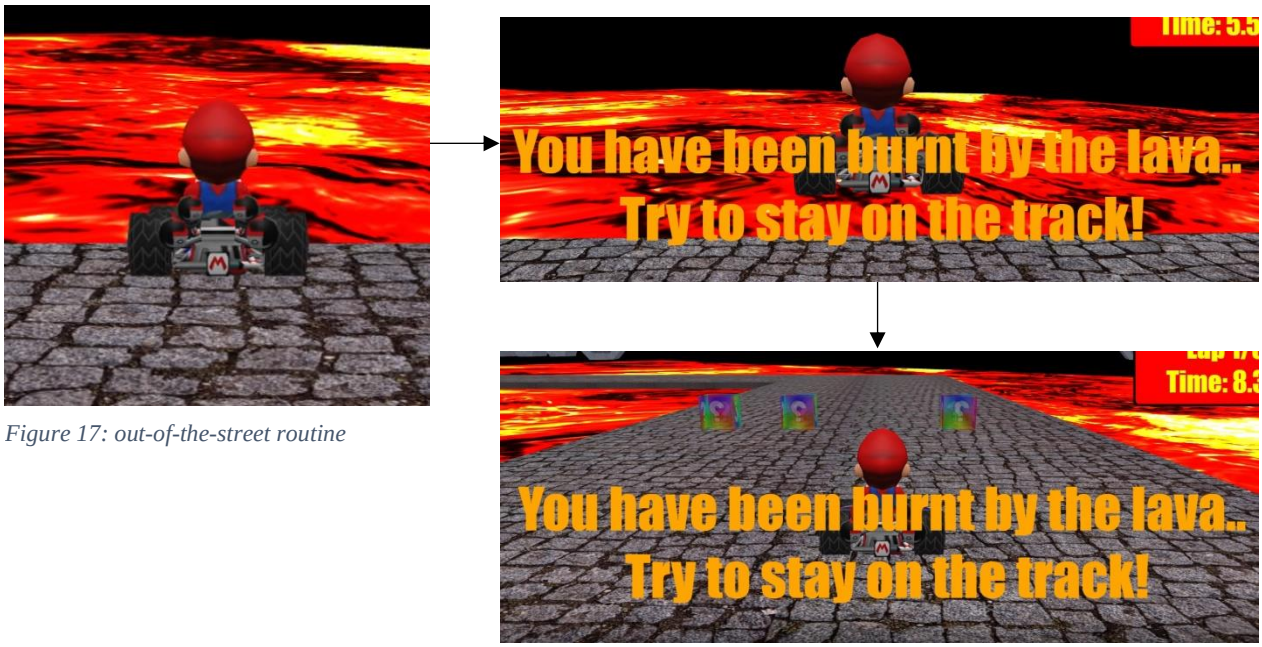

Figure 17: out-of-the-street routine

5.3. Rainbow Road

Rainbow Road (Figure 18) is the last circuit of the Grand Prix and maybe provides the most suggestive landscape. The environment is in the free space, in an undetermined point between the sun and the Earth planet where the Milky Way and lots of stars are visible. The light is located near the sun on the back of the user car and is characterized by a blueish diffuse and an orange specular, accordingly to the colours that the rendered sun emits. The sky is again present and is totally black in the areas far from the sun or slightly illuminated in the areas near the light source. Since in the real-world stars emit light too (as Milky Way does), we have instantiated other three[5] fictitious sources in the corners of the scene to guarantee an additional illumination. These three further lights are *PointLight* variables, types of light that emit in every direction from their position, and their intensity is consistently reduced to simulate the infinitely high distance of the Milky Way and the stars. Moreover, these two objects are rendered through the *Point Cloud System (PCS)*, a set of particles of equal size that can assume the shape of an input mesh: in particular, the Milky Way is realized with a torus (3D 'donut') attached to the PCS, while the stars are randomly distributed over a sphere.

Another particularity of Rainbow Road is the composition of the street, created with colourful textures that simulates a rainbow. Since this specific 'material' is translucent, we have properly halved the alpha value to 0.5 to give a realistic almost transparent effect of the track.

[5] The maximum number of lights a BabylonJS Material can handle is four by default (if not explicitly modified). The choice of adding only three lights to simulate the whole stars illumination takes into account this limit.

In this track we used some ‘semi-circular rainbow star’ models to even more improve the environment look.

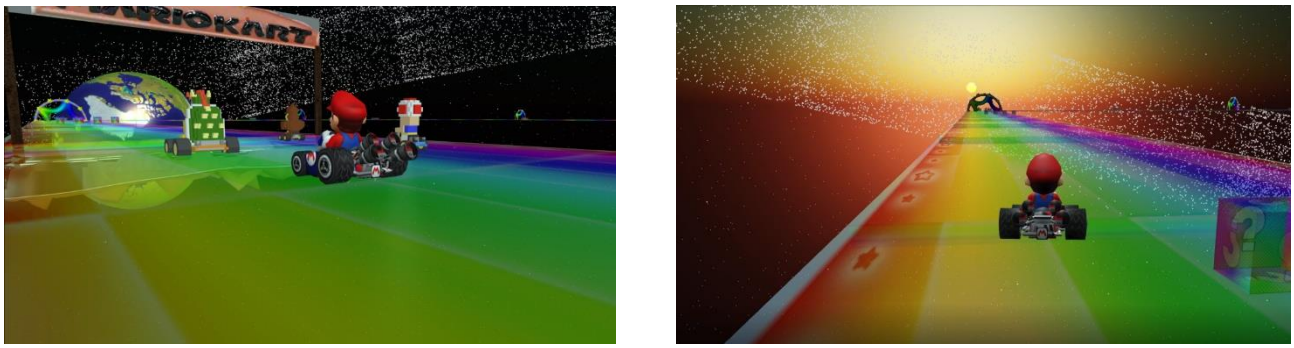

Figure 18: Rainbow Road lateral views; notice the Earth, the Milky Way, the translucent street, the Sun and its illumination

Finally, out-of-the-street events exploit again the usage of checkpoints to resume the car position when it gets out of the road: this is actually a fall in the dark space that stops the race for the user that has to wait until the vehicle is re-positioned on the track, causing a relevant delay for the player (Figure 19).

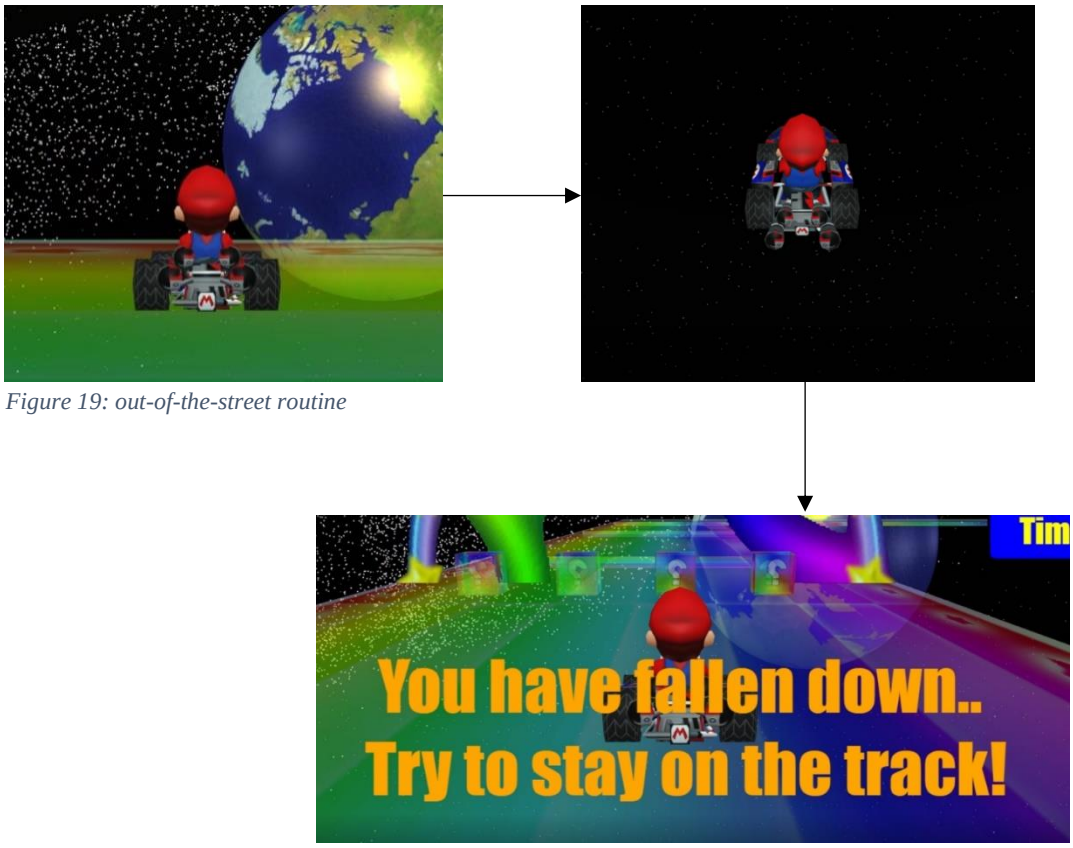

Figure 19: out-of-the-street routine

6. User manual

Now let us talk in general about techniques to optimize the use of our game and the controls to use for the right and best experience. First, our application can be run in any browser; however, we have noticed that on *Google Chrome* it loads faster and allows you to experience more frames per second (FPS). This last feature is extremely important in our game: in fact, the updates to the movements of the car happen every time a new frame is rendered, so if you have a low value of FPS the car will move slower than expected, obtaining an incorrect gaming experience. In particular, the hardware specs used to test the game include an Intel® Core™ i7-10875H Processor 2.3 GHz (16M Cache, up to 5.1 GHz, 8 cores) CPU of which the integrated GPU was used in performance mode (the dedicated GPU of our architecture was therefore not used) and a 17.3 inches display (16:9, 300Hz refresh rate, 3ms response time). Finally, a fairly good internet connection is expected to truly enjoy the game. The recommended FPS per racetrack are shown in the table below:

Race Track	Recommended FPS
Luigi Circuit	60-70
Bowser's Castle	140-150
Rainbow Road	140-150

In Luigi Circuit we obtain a lower value of FPS probably because a lot more models are loaded in it (especially walls and plants). An *alert* (Figure 20) has been added to the game's Main Menu, which informs the user that for a better gaming experience we recommend Google Chrome (this alert does not appear if the user is already using Google Chrome). Another message appears if the player tries to start to race without having selected track, character or both (Figure 21).

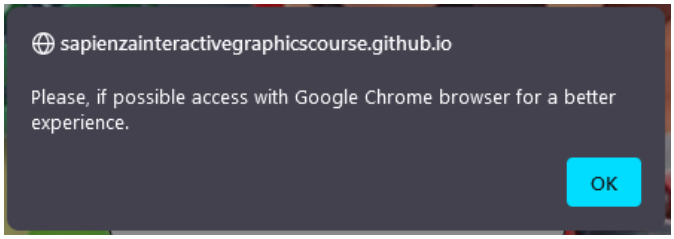

Figure 20: Browser alert

Figure 21: Select track/character error message

Finally, let us see the game controls:

- **W**, **A**, **S** and **D** to move the car along the racetrack;
- **E** to use the item collected if one of the item boxes is hit;
- **R** to restart the race;
- ←, ↑, → and ↓ or click and move the mouse to move the camera;
- **C** to continue to the Main Menu (or to the next track in the Grand Prix) when the race finishes.

Only a small comment needs to be added regarding the game controls: the application can only receive one input per frame, therefore the user does not have the ability to, for instance, accelerate and turn at the same time.

7. Conclusions

The final result of our project is reasonably acceptable while keeping the preliminary conditions: provide a realistic and interactive experience for the player combined with an easy-to-use environment. The required simplifications with respect to the original game of Nintendo have been exhaustively explained and discussed together with the difficulties we have faced during the implementation. The interaction with BabylonJS is the real strength of the work, where this graphic library has revealed to be the most appropriate choice for our ideas. Every aspect of the gaming experience we are providing has been carefully considered and the overall outcome is very satisfactory: some selected external test players have reported a general easiness in understanding how to play and how to interact with the menus we have created, with a definitive appreciation about the structure and the development of the game. All the graphical features in the project are characterized by a theoretical background to better understand the most suitable and feasible way to implement common properties like lighting or animations, always taking into account how the official game handles such situations. To finally conclude our report, the proposed project looks enjoyable and accurate, especially if the user has a background about Nintendo and its videogame platforms, all the inevitable drawbacks have been discussed and the motivations that have led to some simplifications have been explained and reasoned. At the end, the global results are coherent with the starting prerequisites and also adequate for the requested project.