University of La Sapienza

Interactive Graphics Project

SAPIENZA
UNIVERSITÀ DI ROMA

Nicolanti Valerio, 1693224

Roggiolani Gianmarco, 1695376

AA. 2018/2019

Contents

Chapter 1

Environment

The entire project is built with html and javascript, using WebGL, Three, and jQuery utils.

WebGL is a Javascript API library for rendering 2D and 3D graphics in any browser without plug-ins. It is compatible with any other html element and is integrated with web standards, giving the possibility to use GPU accelerations. The low-level features of the library contribute to the creation of larger and more advanced libraries as A-Frame (with VR compatibility), Three, BabylonJS and many others.

The potential of WebGL is clear when we think about the numbers of game engines produced including Unreal Engine 4 and Unity. Some light-weight utility library providing the vector and matrix math utilities for shaders is been developed to be used with a WebGL specific extension called **glUtils.js**.

Desktop 3D authoring software such as Blender, Autodesk Maya or SimLab Composer can be used to export scenes to use. Particularly, Blend4Web allows a WebGL scene to be authored entirely in Blender and exported to a browser with a single click, even as a standalone web page. Online platforms such as Sketchfab and Clara.io allow users to directly upload their 3D models and display them using a hosted WebGL viewer.

As stated before **Three.js** is a library based on WebGL, it's open-source and the code can be found on a public repository on GitHub. some of the most important features included are listed below:

- **Scenes** with the possibility to add and remove objects
- **Cameras** both perspective and orthographic with easy-to-add controllers
- Different types of **lights** and the shadow property, both for casting it and receiving it

- **Materials** as Lambert, Phong, with textures and shading properties
- Many types of supported **objects** as meshes, particles, spirits, lines, and bones
- **Geometry primitives** for planes, cubes, spheres and more
- Specifics **data loaders** for external resources import

The **jQuery** library is primarily used for *Document Object Model* manipulation. It simplifies search, selecting and modifying operations and also provides a paradigm for event handling.

It allows javascript to handle the events connected to the DOM elements of the html encouraging a strong separation from the javascript code. It aims to promote clarity, brevity, and extensibility, removing the cross-browsers incompatibilities.

Despite all these libraries are completely cross-browsers, in section 3.1.2 we see that some of them have restrictions or characteristics which prevent the project to effectively be browser independent.

Chapter 2

External credits

2.1 Index Template

The index page has been created using the Bootstrap framework. The template is a full page bootstrap carousel composed by two slides:

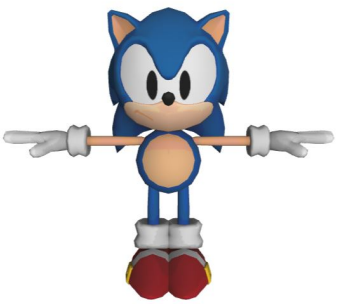
1. The main page, it shows the basic controls for the game and it contains the "Play the game" button and the sliders for the difficulty selection.

2. The tutorial for the game, constituted by a frame and other elements of the game with on hovering bootstrap tooltips with advices for the player.

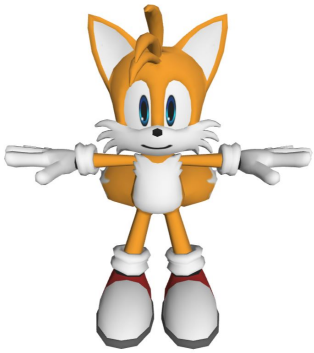
The code is split among the html page and one CSS files. The *vendor* folder contains the ready-to-use compiled code for Bootstrap v4.0.0 and jQuery.

2.2 Models

Most of the 3D models are imported from open source web sites as Sketchfab and model-resources. Below there is a list of all the models, with annotations on specific characteristics we exploited or built-in animations.



Sonic: the model has a hierarchical structure we animated manually (3.2), without importing the existing animations. Its position is changed throughout the game by the user (4.2.1).



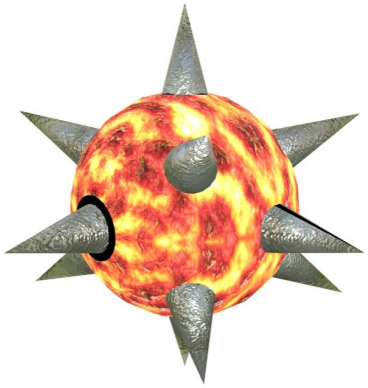
Tails: the model has been animated manually (3.4.4) thanks to the hierarchical structure, no animation was imported. The animation starts at every respawn, which includes a repositioning of the object.

Eggman: the model has no nodes for the chain's rings so it was not possible to animate swings during the motion which was not imported (3.4.1).

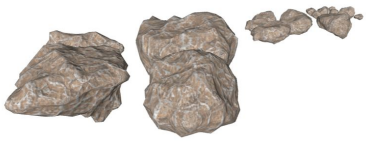
Heart: the model is repositioned during the game, it has not a hierarchical structure. The animation was not built-in, it's a simple rotation around the vertical axis.

Ring: the model is cloned and positioned several times, it has not a hierarchical structure or animation.

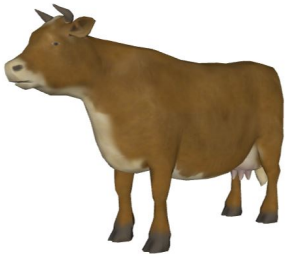
Robot: the model has a respawn function which changes its position on request. It moves horizontally and rotates around the z and y-axis, the animation was not imported (3.4.3).



Fireball: the model is repositioned during the game. The animation of the movement and rotation was manually implemented (3.4.3).



Rubble: the model is not visible when loaded and it's repositioned on request every time an obstacle is broken, there is no animation.



Cow: the model is cloned just once, the two copies are repositioned during the game, the animation has been imported with the model.

2.3 Perlin Noise

The project uses the **Perlin noise**[1], a type of gradient noise developed by Ken Perlin. The algorithm has an organic appearance because it produces a naturally ordered (smooth) sequence of pseudo-random numbers.
In game development, Perlin Noise can be used for any sort of wave-like, undulating

material or texture. For example, it could be used for procedural terrain, fire effects, water, and clouds. These effects mostly represent Perlin noise in the 2nd and 3rd dimensions, but it can be extended into the 4th dimension. An implementation typically involves three steps: grid definition with random gradient vectors, computation of the dot product between the distance-gradient vectors and interpolation between these values.
For this project, the Improved Noise has been used for calculating random plane thickness for the sides (3.6.3).

[1] `https://mrl.nyu.edu/~perlin/noise/`

Chapter 3

Technical Aspects

3.1 Technical Features

3.1.1 System Requirements

The game is launched using the browser, so there are no requirements or restrictions on the operating system. However, due to the number and size of the models, the game could run into slowdowns when the RAM available is not enough.

After different trials we can say that the game is fully enjoyable with at least 8 GB of RAM; it's still playable with 4GB but there would be frequent slowdowns.

3.1.2 Browsers

Browser	Compatibility
Chrome	The game does not work because the CORS policy forbids loading from local origin[2].
IE	The game works on the default release without additional requirements.
Firefox	The game does not work due to CORS policy[3] and lazy loading.[4]
Safari	The game works after disabling the **cross origin restrictions** from *Preferences* → *Advanced* → *Develop Menu Bar*.

[2] It would be possible to reconfigure the game uploading models and textures on a storage online or a public Github repository and changing all the loading paths. Using https request for all the resources would slow down the game, because of this, we decided to not reconfigure it, considering it is still playable on other common browsers.

[3] As for Chrome, Firefox does not allow the loading of local resources. However, this can be solved by modifying the configuration parameters of the browser, setting STRICT_ORIGIN_POLICY to false.

[4] The lazy loading of the resources is a minor problem, the game is still playable with minor adjustments on the variables' initialization, but for the first minutes, most of the textures would not be displayed. This is very unpleasant, for this reason, we discourage the use of Firefox to play the game.

3.1.3 Reuse Objects

Probably the most important rule for efficient web experiences: after the objects' creation in the initialization, no new objects should be created while the game is running, this avoids memory thrashing. In fact, in the project we reuse almost every[5] object we create by repositioning them in the setting when they go behind the camera. On every frame, we check if the objects are behind the camera view, if so we reset their position further on the road.

[5] *The only exception to this rule is the line object representing the laser. More details in chapter 3.4.3.*

3.2 Sonic

Sonic is the main character of the game and he's controlled by the user. That's why his animations are the most accurate and complex. Some of the values are fixed for most of the time, like the ones for fingers which are closed in a fist.

One big dictionary allows access to the nodes of the hierarchical model {*Head, Thighs, Calves, Arms, Forearms, Hands, Fingers' first and second phalanges* }. Two main arrays are created when the script is processed to store the interpolation values for the running and jumping animations.

The function **move_sonic()** is called at every rendering frame and set for every node the correct value extracted from the arrays. The array containing the values for the y position, which is changed only when sonic is jumping, is accessed only if the variable **jump** is true, which happens the first time when the spacebar is pressed (4.2.1) and then is set to false when the jumping time has reached the length of the array.

When sonic has been damaged he turns invincible for a fixed number of frames, during this time the main animation loop removes and adds again three times the model from the scene. This blinking shows the user the collision occurred and is handled by a variable incremented at every frame. After this variable reached the desired value, the invincibility is turned off and the variable is set again to zero.

At the end of the game, sonic slows down before stopping. During the slow down the running animation is performed as before, but the speed along the z-axis of sonic is

halved. The final pose uses no interpolation, this was partly due to a large number of possible starts. Not knowing how and in which frame sonic dies, we should perform the linear interpolation run time to have a really smooth animation towards the final pose. Instead, we decided for a sharp switched in the camera view and the immediate setting of the values for the final pose of the character.

3.3 Items

During the game some items will spawn randomly along the road to help the player:

- The **heart** will restore one of the lives, if the user already has the maximum number of lives available (based on the difficulty chosen) he will not be able to pick it up.
- The **rings** are the only way to get score points, every ring gives 10 points.
- The **shield** provides invincibility for some time, during this time the user is granted to not lose hearts even after an enemy hit.

As for every object we use, the items are not created every time they spawn, but they are reused changing their position in the game world.

3.3.1 Life

The game gives the user the possibility of choosing the difficulty, this affects the number of lives in the game. Once the game is started from the index page, the difficulty information, in terms of the maximum number of hearts available, is passed as a parameter in the URL.

The lives are displayed in the upper right corner of the window and are represented with two different png images: one when the life is available, the other when it is not. The in-game heart model is initially positioned along the negative y-axis, far from the camera view (hidden by the road), then every time the z-position of sonic is greater than the heart's z-position, the heart is repositioned with a certain probability using the **heartSpawn(start)** function. This function takes as input the current z-position of sonic to generate a feasible random position for the heart to be placed, then if that position does not collide with obstacles, rings, and shield, the heart is actually repositioned.

When sonic is damaged by an obstacle or an enemy (3.4) the **getDamage()** and **damageFeedback()** functions are called. The first is in charge of removing one life (changing the corresponding displayed .png image), the second sets to true the variable **invincibility**. In fact, as said in the previous chapter to avoid multiple damages from

the same or others sources, sonic is invulnerable for many frames.

The function **getHeart()** checks if sonic is in the hitbox of the heart, when this happens **checkLives()** is called to restore one life in the HUD.

When sonic loses the last life a variable is set to change the animation loop, from **animate()** (3.2) to **death()**. For the first 70 iterations, the animation of sonic is only slowed down; this allows sonic to pass the enemy or obstacle and to terminate the jump if he was jumping.

Then sonic is shown in a final pose, created using specific values for his arms, forearms, hands, thighs, calves, and head. The camera position and orientation changes to be in front of sonic, directed towards his face, centered on his position. Being this the final pose, we do not need to keep calling the **requestAnimationFrame()** function which is called for the firsts 70 steps.

After setting the final pose for sonic, the function **clear_area()** remove any objects or enemy in the surroundings of sonic. Without this, the 70 steps after the real death of the player could lead the character near an item or obstacle which could obscure the view of sonic. The trees are removed because if saw from this direction they seem unnatural, being oriented to be seen from behind the character, not from the front.

This function also creates and positions, using the **game.css** file (3.7.2) the final score and the button pointing to the index page.

3.3.2 Rings

As said before, the rings give the player 10 points and they are the only way to increase the score.
In the game, the rings never spawn isolated but they are always in a group of five. The groups can be of two kinds: a straight line of rings with fixed height and a series of rings following a parabolic curve pattern.

In the initialization through the **randomCoinInitialization()** function, five groups are created and positioned in the game with random group type. As every object we use, when the rings are left behind the player's current z-position, the rings can be repositioned ahead on the road.
Every time the first ring of a group must be repositioned, the new group type and the new position are randomly calculated again. For a stronger collision avoidance with the obstacles, the collisions are checked not only for this first ring but also for every other following rings in the group since their positions are already been decided, even if they are not been repositioned yet. In fact, in order to keep the rings of a group always chained together, the last four rings in a group will never be repositioned before

the first one.
The function **check_line()** is responsible for checking the collisions with the obstacles, and it takes into account their height and width, so that the rings can be repositioned above the short ones (3.4.2).

Finally the **check_ring()** function checks if sonic is in the ring's hitbox, if true the ring is removed from the scene and ready to be repositioned.

3.3.3 Shield

The shield is created using the Three primitives on objects and materials: it is a sphere geometry, transparent and with opacity set to 0.35 of a light blue color (#0C67FB). A variable **initialization** checks if the shield is spawning for the first time or not, if it's the case it needs to be added to the scene, otherwise it's already in it and we only need to set the position. When spawning, a random position is generated among the feasible one for the player. Then we check that the position generated is not colliding with obstacles, rings or other important items. If there are no collisions we can place the shield there.

The function **get_shield()** checks if sonic is in the shield hitbox, in that case it sets to true the variables **shield_on** and the invincibility. This second variable is crucial to avoid the player getting damage when having the shield on.

The **update_shield()** function moves the shield with sonic if the player has the shield on, sets the variables needed for the respawn of the shield if it has expired of the player didn't manage to get it, handles the intermittence of the shield when it's expiring and its disappearing at the end.

3.4 Enemies and Obstacles

3.4.1 Eggman

The function **egg_spawn()** is invoked every time the position of the player (sonic) has advanced for 150 units on the z-axis from the last time. The attacks of Eggman are subject to probability, so there may be cases in which the function is called while Eggman is already on the scene; that's why the function checks his presence through a variable.

The function plays an audio and use a loaded texture to display a warning of the incoming enemy. The texture is placed on a plane transparent geometry and he's placed on the ground; at every rendering while Eggman is moving towards the player, the warning follows his position.

The **turn_off_eggman()** function handles the visibility of the warning; when Eggman is visible from the player the warning is not needed anymore and can be dismissed.

The function **eggman_moves()** is the one in which Eggman's position is updated while he's on the scene. While the y position stays constant, the x position is created only once using the linear interpolation: Eggman moves from one side to the other of the screen repeatedly.

The z position of Eggman increase until he's some steps in front of the player. When he reaches this position there's a random probability that he tries striking an attack to sonic, this probability is checked at every frame if the attack is not already performing.

The attack consists of a movement towards the z position of sonic. The attack follows the movement of Eggman along the x-axis, without checking for the previous sonic position; Eggman is one 'easy' enemy with fixed behaviour, only the start of the attack is random.

Then Eggman goes back to his position in front of the player and keeps on moving, checking again for the probability of at every new frame.

After a fixed number of attacks Eggman disappears going behind the player, from where he came and from where he can appear again later.

The function **check_eggman()** is called only if sonic is not invincible, which means the player can be hurt. In that case, the function checks at every frame if sonic has been hit by Eggman. Due to the dimension of both models, the position check cannot be on the precise value, but it requires a hitbox.

If sonic has been hit, the function call **getDamage()** and **damage_feedback()** which decreases the number of lives, trigger the end of the game if it's the case or turning on the invincibility of sonic for a fixed number of frames.

3.4.2 Obstacles

The obstacles are cylinders create with the Three primitives, the texture is loaded from an image to resemble the marble material.
The function to generate the obstacles is called only once, it fills an array with **max_obs** objects to be referenced from other scripts when needed.
For every obstacle the first thing is generating a random xz position and orientation, the obstacles can stand vertically or horizontally and this defines their y position.

The obstacles are the second item generated after the rings, so during the initialization, the function must only check their position, and the position of the obstacles already placed, to avoid any collision. If all these checks are satisfied, the object is created, inserted in the array and placed in the right position.

The function **delete_obs()** is called every frame and if an obstacle is not visible anymore, it calls the function **repositioningObstacle()**. This function is called even if the obstacle is broken by Eggman. As before, a random orientation and position are generated and checked.

The check considers the new orientation and the line type of the rings, in this way, there is the possibility to have a curve of rings above the horizontal obstacles. If the new position is not feasible, instead of calling again the function right away the obstacle is set as invisible to avoid false collisions. This is possible because every frame we call the function, we can wait a few frames to change the position of the obstacle and avoid recursive calls which tend to slow the game and give stack memory problems.

Two functions check for the collisions with the player and with Eggman, who can break the obstacles down. The collisions depend on the orientation of the obstacles, the hitbox is created so that the horizontal ones can be jumped over and considering that they are wider and shorter. If the player is inside the hitbox the functions handling the damage and the invincibility of sonic are called.

If Eggman is inside the hitbox and the obstacle is visible, we try to change his position. At the same time, the object **rubble** is placed at the old position and the correct audio is played to display that the obstacle has been broken.

3.4.3 Robotenemy

Very similarly to Eggman, the robot enemy spawns every 100 units on the z-axis and 150 units forward to the current position of sonic, then quickly comes towards him stopping 13 units before him. From this moment on the robot can shoot with a certain probability, when its lifetime runs out it continues on the negative z-axis disappearing behind sonic.

During its lifetime (handled by the **robotEnemy()** function) the robot moves left and right, and rotate on the z and y-axis in order to keep looking towards the camera.
When the robot is about to shoot a red line object is created through the **laser()** function, the starting, and ending points are constantly updated to follow the robot's and sonic's movements. A new line is created for every shoot, to avoid memory leaks the geometry and material are disposed every time and the line removed from the scene, when needed.
Once the laser lifetime is over, the current sonic's position is taken as final destination

for the trajectory of the projectile, then the projectile is actually shot.
The **check_fireball()** function checks if the projectile is in sonic's hitbox.
Both the robot and the projectile's movements are calculated using linear interpolation.

3.4.4 Tails

Tails can spawn at any time with a fixed probability, if he's not in the scene a random value is extracted and, when it's higher than the probability, the model is placed in the distance in front of sonic, with x value generated randomly among the possible values. The function handling the spawn also sets the variables indicating that tails is in the scene and that we need to move him.

The main function for Tails' animation uses two auxiliary functions, **tails_jump()** and **move_aside()** to let tails change his position to avoid obstacles, plus the function **check_tails()** if the player is vulnerable, to verify a possible damage when sonic hits tails. Whenever tails is not visible anymore, when he's behind the player, the variables are set again to give the possibility for a new spawn.

Using a dictionary we have access to the singular nodes of the hierarchical model to animate tails as if he was running. We created the movement for eight nodes {*Arms, Forearms, Thighs, Calfs*} using linear interpolation, which is computed when the script is first processed, only once. Then at every frame, the movement is given by the correct value in the array for every node, plus the update of tails' position along the z-axis, which decreases so that he moves towards the player.

If tails is not already jumping a check verifies if a horizontal obstacle is in front of him, if it's the case only the y position of the model is changed using a pre-computed linear interpolation for the jump. If tails is already jumping the function is skipped, while if he doesn't need to jump the function check whether he needs to move along the x-axis to avoid vertical obstacles.

Moving tails right or left requires additional information, after checking if he needs to move because of a vertical obstacle, we must know his x position and if he was already moving aside. To decide whether going right or left we check if tails is already moving, in the case he will keep on moving in the same direction; if he was not moving we check if he's allowed to move in that direction without passing the border of the road. After the lateral movement is completed, the function does not enter the if statement, being tails outside of every hitbox, and enter in the else statement which settles to false the variables used to verify an existing movement.

After this, if sonic is vulnerable we check his position against the tails hitbox. If the two models are colliding **getDamage()** and **damage_feedback()** are called to handle the lives and the invincibility variable.

3.5 Audio

The audio tracks played during the game have been downloaded from web sites providing soundboards as *Realm of Darkness*, *Zapsplat*, *Sounds-Resource* and *Game Theme Songs*. The project is not for profit and all the sounds, as well as the characters used, belongs to SEGA[6].

[6] `https://www.sega.com/`

3.5.1 Background Music

The theme song **Green Hill Zone** is directly loaded from the html page of the game, setting the auto-play option. Int his way the audio is played even when the resources are loading and then throughout the game. The auto-play also assures the re-start of the background music after it ends.

3.5.2 Audios from Interactions

Most of the audio tracks are launched during the game on specific occasions. In doing so we just need to upload the resource and then play it when we want to, after the appropriate checks to avoid overlapping and interference. the volume is set as to avoid strong sounds and not to overcome completely the background music, being anyway understandable.

Some of the audios are alternative tracks, their use depends on a random number generated at run time. This creates a variety during the game, which we preferred to a fixed catchphrase that would bore a player after a couple of times.

Whenever the player loses a life, sonic emits a pain sound. This enforces the knowledge of what happened, together with the blinking of the character for some time on the screen.

3.5.3 End Game

When the game is over a catchphrase from Sonic encourages the user to play again. The small length of the tracks, together with the steps between losing the last life and the final pose, allows this sentence to be heard without complications from other sounds.

3.6 Scenario

The game setting is composed of many different elements and objects:

- Road
- Background and clouds
- Sides
- Lights

3.6.1 The Road

The road is the only place where the player can freely move and the obstacles, the items, and the enemies can spawn.
In the initialization, two planes with the repeated road texture are created and placed in sequence. To give to the user the illusion of an infinite road, every time sonic goes beyond the first plane, it is placed in front of the second one, and so on as the game progress.
The *receiveShadow* property of the road property is enabled.

3.6.2 Background and clouds

The two backgrounds and clouds are plane objects with textures. The first ones are big enough to cover all the camera view from the top to the ground and move at the same speed of the player, the back background is used for the final pose since the camera view changes.

The clouds have transparent property enabled to hide the cloud image background. There is a maximum of 50 clouds at the same time in the game. As seen before with other elements in the game, the clouds are initialized at the start, then when the player goes beyond them along the road, they are randomly repositioned ahead. The **checkClouds()** function checks collisions to avoid textures flickering.

3.6.3 Sides

The sides are planes populated by trees and cows.

The sides thickness is calculated through the **Improved Noise** (2.3) algorithm.
The basic idea is to divide the plane in *FLOOR_RES* x *FLOOR_RES* squares and calculate the thickness with the noise. The noise is determined by computing a pseudo-random gradient at each of the four nearest vertices and then doing splined interpolation. The sections closer to the road are flatten for smoothing the sides with respect to the road.

The trees are simple plane objects with textures and transparent property randomly positioned in the flattened section of the sides. There are a total of four textures which

are always randomly chosen to be applied to the trees' planes when they need to be repositioned.

The last objects in the sides are two cows with built-in animations. The cows' position and z-direction are randomly chosen.

3.6.4 Lights

There is a total two light:

- Ambient light
- Spotlight: positioned behind sonic, with high y-position. The light's target is sonic and it follows him moving at the same speed along the z-axis. This is the light that casts dynamic shadows since the *castShadow* property is enabled.

The planes that receive shadows are the road planes. The objects that cast shadows instead are: sonic, rings, heart, eggaman, the robot enemy, and the projectile.

3.7 Utilities

3.7.1 Functions and Declarations

The **utils.js** file contains the interpolation function, the definition of the loaders for gltf models and textures, and the variable scene which is referred throughout the scripts. The loaders are just declared from the Three primitives; the models' loader has the crossOrigin parameter set to true to avoid incompatibilities with some browser's releases.

The interpolation function requires the starting value, the ending value and the speed of the interpolation. It returns an array whose length depends on the speed, in which every element is composed as ***starting_value * (1 - step) + ending_value*step***.

3.7.2 Style

The CSS file for the game page can be split in two, the first part contains the style for the loading of the game, the second part contains the style for the displaying of the final score and the button redirecting to the main page.

To display the loading and then the game without overlapping, we created a new canvas class which is not initially displayed. Only the loading elements are shown: a loader element with appropriate attributes is animated with a flex in the column direction, the animation changes considering the percentage of the loading. Under the loader, a brief text informs the user that the game is loading and everything is

going fine. This solution has been considered to avoid the black screen while waiting for the loading of the scene, which can be slow depending on the computer capabilities.

The **onLoading()** function in the html file verifies if the scripts have all been loaded, then it sets to none the display attribute for the loader, while it sets to init the canvas display where the game is.

The final score is modified from the element displaying the score throughout the game, changing its position, size, font, color, and animation. The absolute position is set using percentage values to avoid discrepancies on screens of different resolutions. The animation effect is linear, infinite but alternate. It is divided into five steps, with the first one equal to the last one to avoid sharp interruptions performing the loop.

The animation modifies the shadow of the text which is decomposed using different shades of colors between red and yellow. At every step. the inner color needs more than one line of code to cast the shadow effect in different directions; step by step the animation enlarges the shadow and then retires back giving the effect of a burning flame. The values for the shadows have been decided using trials until a final pleasant effect was reached.

The last element is the style description of a button, which is the same as the index page and that redirects there. The button has a background color, a solid colored border, a scale transformation and the description of the span element inside the button, to have it centered in the button with 'large' font size.

Chapter 4

Interactions

4.1 Index

The index page is made up of two sections, the first one contains only the base information to play the game and the possibility to change the game difficulty and start the game. The second section contains a tutorial, showing the objects present in the game and how the user is supposed to interact with them (avoiding enemies, collecting points and items).

4.1.1 Difficulty

The user can change the difficulty of the game pressing two arrows, one linked to the **lowerDifficulty()** and one to the **increaseDifficulty()** function. Between the two of them, there is a text element in HTML which displays the currently selected difficulty among Easy, Normal, Hard and Extreme.

The two functions above change the displayed text using the innerHTML attribute of the text element which is obtained by its id. Then, depending on the new text displayed the function changes the color of the arrows: blue (#0506DB) if pressing that arrow the difficulty can change, white (#FFFFFF) otherwise.

4.1.2 Play the Game

The button 'Play the Game!' can be clicked on to trigger the **start()** function which launches the game. The function check for the selected difficulty, then it changes the window.location.href parameter passing also the difficulty information as the number of lives parameter.

4.1.3 Sections Navigation

The users can switch between the two sections of the page in two ways:

- Using the previous and next arrows displayed on the left and the right of the sections.
- Using the navigation bar on the bottom which displays two dots, one for each section.

4.1.4 Tool-tip Text

In the second section, there are a total of six tool-tip texts which appears when hovering six elements.

Three of them are boxes overlaying a game screen, the others are simple images presenting the items.

4.2 Game

The game is displayed with a Three.js scene, rendered in a loop. The users' interactions are aimed to control the character in the game (sonic) to move it on the right or left or to make it jump over obstacles. After the game over a button gives the possibility to play again.

4.2.1 Game Commands

The listener for the keyboard pressing is added at the end of the model loading, otherwise, the keyDown event could trigger the function which uses an object still undefined, throwing an error. After pressing the letters 'A' or 'D' on the keyboard, the function checks the **keyCode** for the event and move sonic in the desired direction if it's allowed. There are only two cases in which the movement is not possible; sonic is at the border of the street or the game is over and the screen is displaying the final score.

As well as for moving right and left, when the spacebar is pressed, if the game is not over yet, the function sets a variable which allows the right animation for the jump to start.

4.2.2 Play Again

The final user interaction is displayed only at the end of the game. A button gives the possibility to go back to Index and play a new game. The button has a listener for the on-click event, triggering the setting of window.location.href to the previous HTML page.