Interactive Graphics Project
Aladdin

Ratini Riccardo, Simionato Giada

{ratini.1656801, simionato.1822614}@studenti.uniroma1.it

Contents

Chapter 1

Introduction

This work takes inspiration from the *Disney* story *Aladdin*. It's a complex multi-level interactive game where the player can interact with the environment thanks to user-friendly commands.
The game is composed by three levels, each of which is based on a scene seen on the homonymous film. In the first level the character, with the appearance of Aladdin that can be commanded by the player, is trapped inside the *cave of wonders* with the only purpose of reaching the genie lamp to be teleported at the beginning of the second level. In this new journey the player can guide the character, now free, by jumping on the roofs around the city of *Agrabah* until it will arrive near the magic flying carpet. Once having grasped it, the character will be allowed to enter in the third and last level, where, on board of the magic carpet, will fly around the world inside nineteen glowing rings for finally landing on the sultan palace.
The purpose of this report is to provide a general overview on the technical aspects that are behind the implementation of this game, along with a useful stand-alone user guide for having a walk-trough shortcut for playing with it.
The report is structured as follows: in Chapter 2 the work structure will be disclosed, especially in Section 2.1 where all the files generated in this work will be discussed and in Section 2.2 where a list of all the used libraries will be presented along with their explanation. In Chapter 3 all the design choices for this work and the steps in the modelling phase will be revealed. In Chapter 4 all the basic elements, e.g. skybox (Section 4.1), lights (Section 4.2), cameras (Section 4.3), musics (Section 4.4), scene loading steps (Section 4.5) and material loading steps (Section 4.6), that compose a scene, will be in turn analysed and all the techniques used to realise the game will be commented. In Chapter 5 the physics of the game, along with the movement system and the game parameters will be deeply explained with particular attention on the functions used at the implementation level, while in Chapter 6 the animations construction process and the main mesh composition will be disclosed. In Chapter 7 how the game deals with input and user commands and how the overall system renders the game will be discussed, especially in Sections 7.1 and 7.2 respectively. Finally in Chapter 8 conclusions will be drawn.
The afore-mentioned user guide can be found in the Appendix A at the end of this report.

Chapter 2

Work Structure

2.1 Files Organization

This work is structured into eight main folders containing the scripts along with useful resources needed to correctly run the game and the main html file to load the live version of it. Specifically, these items are:

- *imgs* folder: it contains the logo of the film used in the different initial menus of the game;
- *js* folder: it contains three *JavaScript* files, namely `L1.js`, `L2.js`, `L3.js` whose in turn contain the code for the scenes and game parameters setup for the all the levels, respectively;
- *L1* folder: it contains the necessary resources for correctly loading the first level, such as texture images, the skybox, the scene exported from *Blender* in the *.babylon* format and the corresponding html file. This latter contains in turn the libraries import commands along with those necessary for the correct import of the level and the canvas to where render the game;
- *L2* folder: it contains the necessary resources for correctly loading the second level, such again as texture images, the skybox, the scene exported from *Blender* in the *.babylon* format and the corresponding html file. This latter contains in turn the libraries import commands along with those necessary for the correct import of the level and the canvas to where render the game;
- *L3* folder: it contains the necessary resources for correctly loading the third level, such as, like the previous two items, texture images, the skybox, the scene exported from *Blender* in the *.babylon* format and the corresponding html file. This latter contains in turn the libraries import commands along with those necessary for the correct import of the level and the canvas to where render the game;
- *lib* folder: it contains all the libraries used for the game implementation;
- *menu* folder: it contains two files, namely `Commands.html` and `Levels.html`. They represent the html scripts of the pages that compose the main menu, such as the single level selection options and the commands recap;
- *musics* folder: it contains three files, i.e. `first.mp3`, `second.mp3` and `third.mp3`, where, as the name suggests, they stand for the music tracks used in the corresponding level;

- *index.html* file: it is the main page of the game that when run, it provides the live version of the game.

The game was implemented combining scripts written in *HTML* language, especially for the menus and for the pages responsible for scene loadings, and in *JavaScript*, for the actual scene loadings and for the gameplay. As regard the former, it was used to build menu pages coupled with *CSS* sheets for providing a more Arabic look, distinctive element in Aladdin. As concerning the *JavaScript* files, already mentioned in the bullet list, in addition to the code for the loading of the scenes, they also contain the script for defining the materials, the animations, the environmental physics, the movement system and how to handle interactions and rendering. Further information on these topics will be disclosed in the next chapters.

2.2 Used Libraries

The libraries used in this work all belong to the *babylon.js* distribution and are clearly imported in the headers of each of the three html files written for the corresponding levels (i.e. `L1.html`, `L2.html`, `L3.html`). They are:

- *babylon.max.js* contains the basics of `babylon.js` along with all the functions necessary to define a canonical scene such as materials initialization or animations implementation;
- *babylonjs.loaders.js* is a library used to import complete scenes or single meshes from external files. In this work it was used to import the final scenes in the `.babylon` format.
- *babylon.waterMaterial.js* contains the shader used in the third level to emulate the water for the lake.

The remaining features, e.g. the animation manager, physics and collisions and interactions, were manually implemented and will be explained in subsequent chapters.

Chapter 3

Design Choices and Modelling Step

To implement a multi-level game platform such as in this work, a first step of hand-crafted design was of fundamental importance. The choices on the number of levels, the difficulties in bypassing the obstacles or the plot of the game needed to create a convincing story line, were only the first layer in the creation of the game. After having established the environments and the types of the three levels[1], the study of the details led to the possibility of creating convincing landscapes without affecting game performances, by limiting the number of elements in the scene as long as not constraining the character movements.
The following step in this work was to model the scenes along with their details. For performing this phase *Blender 2.80* suite was used, due to its free and open source nature, it allowed to create 3D models, dealing with lights and textures in a simpler way. By editing basic shapes like cylinders, cubes, planes and conics, was possible to build more complex *meshes*. Even due to the presence of several tools in the *Edit Mode* that allowed to work with vertices, edges or faces in a separate way and to modify, introduce and remove parts in a detailed level, where the *proportional editing* tool played a main role. The use of *modifiers* was also fundamental since it allowed to blend basic meshes in such a way not achievable with any other basic operation. In Fig. 3.1 is depicted an example of the time-lapse sequence of the creation of a house.

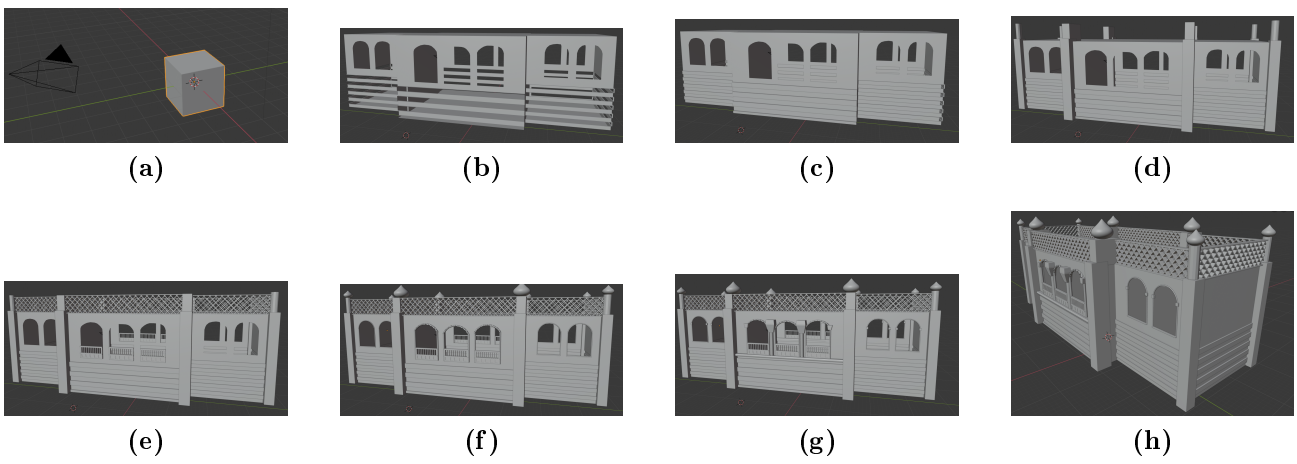

Figure 3.1. House modelling process: (a) initial mesh; (b) addition of other two meshes and first details skeleton; (c) embroidered details superposition; (d) wall decors creation; (e) railings and balconies addition; (f) ceiling and edges details creation; (g) additional details and decors insertion; (h) final house from a different point of view.

After having created several basic meshes the landscape skeletons were built: planes were subdivided into vertices and shaped to represent the ground, according to the specific terrain,

[1]For further details on the appearance of the levels, please refer to the User Guide in the Appendix.

e.g. desert, mountains, cave, lake and street, and, as in level one, internal walls. Once having setted the level scenes, they were enriched with the complex meshes already created, according to the design. Maintaining the number of vertices present in the scenes limited to a maximum threshold, without loosing the detailed and realistic effects, was one of the main challenges in creating the game at the modelling level.
In order to make more appealing the scene, in a prior phase, textures were also added in *Blender*[2]. This step was twofold: it allowed to understand how to combine different textures and to select more promising effects, and to perform the *baking* operation needed to use materials created using *nodes* even with *babylon.js*, whose doesn't support this feature. In fact, the baking operation allowed the creation of an image to substitute the procedural material with, aimed to make useless the computations in the rendering phase by looking like the chosen material under that rendering environment. Images used directly as textures or resulting from baking operations were then exported for being handled in a subsequent step by *JavaScript*. Examples of meshes, along with their textures, that represent details in the levels, are shown in Fig. 3.2.

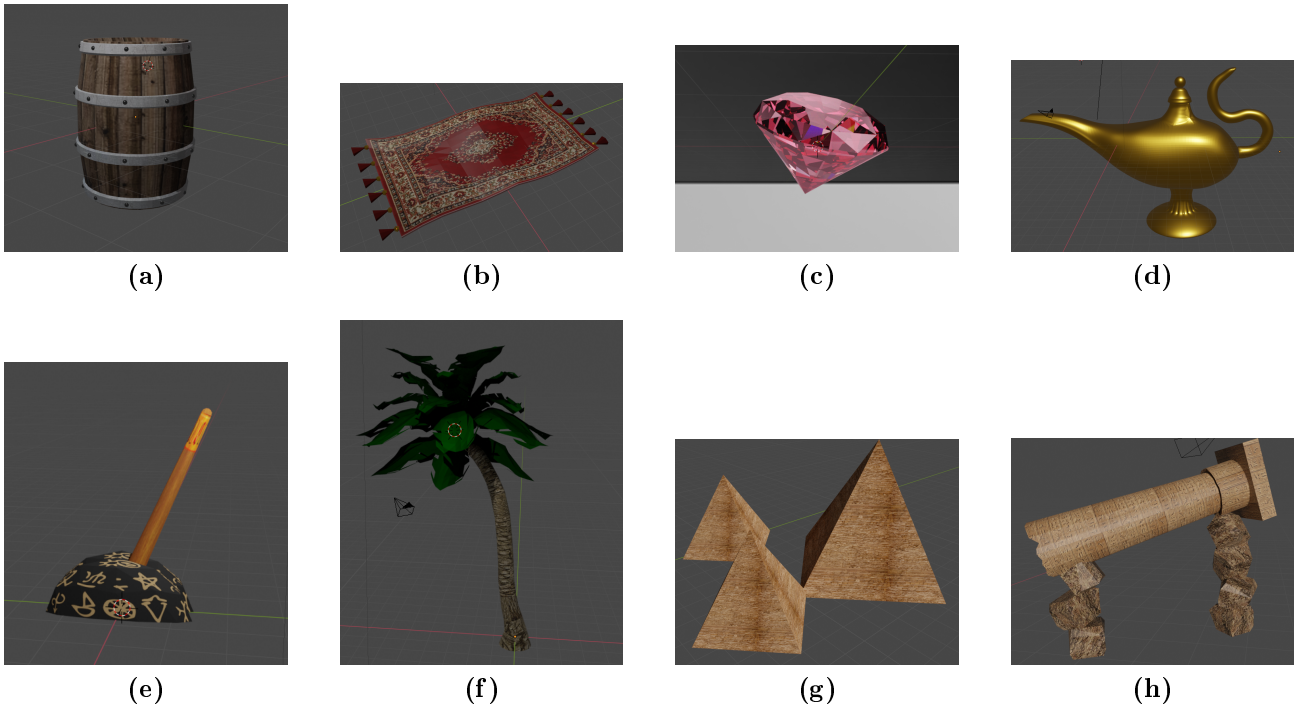

Figure 3.2. Detailed meshes: (a) barrel for the second level; (b) carpet for obstacle in second level; (c) diamond for first level; (d) genie lamp for first level; (e) lever to open the wall for first level; (f) palm for second level; (g) pyramids for third level; (h) ruins for third level.

Once having filled the basic skeleton of each level with details, the scenes were exported to be further handled and inserted in the *JavaScript* files, as explained in Section 4.5.

[2]For textures creation in *JavaScript* please refer to Section 4.6.

Chapter 4

Scene Creation

The creation of the scenes was implemented in the functions `createLx`, where `x` belongs to the set $\{1, 2, 3\}$ and stands for the corresponding level. For each level, inside these functions, firstly a new scene was defined through the `BABYLON.Scene()` method and then the colour of this new scene was setted to white thanks to the parameter `clearColor`. In a second step a customized definition of the objects that compose the scene took place. In the following sections these elements will be deeply analysed and all the technical details will be revealed.

4.1 Skybox

The *skybox* is a cube whose dimensions are much greater that those of the scene, into which this latter will be inserted. In the internal faces of the cube, textures will be applied in such a way to simulate a sky scenario (code lines: *1198-1205* for the first level, *1259-1266* for the second and *608-615* for the third one). For creating the skybox a huge cube was created through the `CreateBox` function of the *babylon meshbuilder* and then textures, such as `reflectionTexture` were applied. The final step was to set the mode *SKYBOX_MODE*.
Please note that the textures used for the skyboxes can be found in the folders `Lx/skybox`.

4.2 Lights

The lights of the game were defined in *JavaScript* files only for the first level, since the complex and detailed nature of the other two levels demanded a simpler way to handle this element. For this reason, for these latter two, the lights were created using the *Blender* suite, then exported and imported along with the other assets. In all the levels the lights are of the type *DirectionalLight* or *PointLight* defined with customized directions, intensities and colours, in such a way to give the illusion of lighting effects and special reflections and to emulate the sun effect, for the last two levels, and the moon one, for the first[1]. In order to define a light using *babylon*, it was used the `BABYLON.DirectionalLight()` function to create the directional type or the `BABYLON.PointLight()` for the point one. The regulation of the intensity was made through the parameter `intensity`, that could vary between *0* and *1* while the changes of the colour of the emitted light was achieved using the parameter `diffuse` and for the one of the specular light was instead used `diffuse` (code lines *1183-1186* of the first level).

[1] In this case the light penetrates into the cave through a hole in the ceiling.

4.3 Cameras

The cameras used in the game were of the special type of *ArcRotateCamera*. This is a particular type of camera that always points in the direction of a specific target, chosen as reference. The straightforward analogy with an object that rotates around a landmark represents the reason for which it has found employment in this work. The camera position, with respect to the target, can be governed through the parameters `alpha`, that represents the rotation angle with respect to y-axis, `beta`, that stands for the rotation angle with respect to the x-axis and `radius` that expresses the distance from the target[2]. By setting the main character as target, the resulting view will be Aladdin-centered even when the player will perform changes on the point of view. In order to define a camera, the function `BABYLON.ArcRotateCamera(name, alpha, beta, radius, target_pos, scene)` was used (code lines: *1190, 1268, 600* for the three levels respectively). For the parameters setting please refer to Chapter 6.

4.4 Musics

In order to engage more deeply the player, a soundtrack for this game was created. Each level has its own music track tied with the film scene from which such level was inspired. For the loading of the audio files, the function `BABYLON.Sound()` was used: the parameters to be specified are: the name to assigned to it, the path of the audio file, the scene into which to play it, a *JSON* dictionary for the parameters setting and finally a callback function that can be executed once the file is completely loaded. In this work, inside the callback function, the method `play` of the audio files was executed in such a way to allow the starting of the sound, while in the *JSON* dictionary the parameter `loop` was set to *True* to make the song restart each time it finishes (code lines: *1208-1212, 1276-1280, 617-621* for level one, two and three respectively).

4.5 Scene Loading

As stated in Chapter 3, once having completed the design and the modelling step of the layers, in order to load the meshes from *Blender*, in this work was used the method `Append()` of `BABYLON.SceneLoader`. This function loads all the meshes found in the export file taken as input[3] and it adds them to the chosen scene. Once having loaded all the specified meshes, it will call a callback function in which, strong of the assumption that all the meshes were already been loaded, all the references to the main meshes will be defined and the various game parameters, animations and input elements will be setted. For further details on these latter items please refer to the next chapters. The function, finally, returns the completed scene (code lines: *1214, 1282, 623* for the three levels respectively).

4.6 Materials Loading

As mentioned in Chapter 3, after a first texturing aimed to understand in a easier way the combinations of the materials, the application of these textures was achieved in the *JavaScript* environment. The definition and the assignment of the materials were performed by the self-made function `create_materials(scene)`. This method creates the materials for the scene

[2]More details on how this camera works can be found at https://doc.babylonjs.com/babylon101/cameras#arc-rotate-camera.

[3]In this work the format of the export file was `.babylon`.

through the constructor `BABYLON.StandardMaterial()`, loads and assigns the `diffuseTexture`, `specularTexture` and `bumpTexture`, if present, otherwise it does the same for the `diffuseColor`, `specularColor` and `normalColor`. It also, if necessary, regulates the scale of the UV mappings of the textures by changing the values of `uScale` and `vScale` and modifies the intensity of the textures through the parameter `level`, that can take values between *0* and *1*. Finally it obtains the pointer to the meshes which assign the material to, through the method `scene.getMeshBy Name(name)` and it actually assigns the material.
In order to obtain the glowing effect typical of the interactive objects and of the rings in the third level, the shader `BABYLON.GlowLayer` was used. To select which meshes will possess this property, it was used the selector `customEmissiveColorSelection` of *GlowLayer* that assigns the glowing effect, with a specific colour, only under determinate circumstances, such as in levels one and two whenever the character is near to an interactive mesh[4] or in the third, if the object has the ring material.
Finally, creating the material to emulate the water in the third level, was more challenging. It was achieved through the shader `WaterMaterial` for which it's possible to define several parameters such as wind strength and direction, waves height and length, colour of the water and for which meshes to show the reflection.
This work counts *106* different materials, individually designed and crafted. The definition of `create_materials` can be found in the code lines: *170, 160* and *116* for the three levels respectively. The water shader is inside the mentioned function for the last level while the glow shader can be found inside the callback function of the `Append()` and `triggers_manager` methods.

[4]For further details please see Chapter 7.1.

Chapter 5

Physics, Movements and Game Parameters

A widely used technique for reducing computational requirements in videogames consists on split the *physical* part from the *aesthetic* one. This idea translates in the creation of several *bounding box*[1], one for each mesh, and then to perform computations required for the movements, collisions and gravity onto these low-polygonal density boxes and use the more complex high-detailed original meshes only for aesthetic purposes. In this way was possible to drastically diminish the computations required without loosing the appeal of the game.
The bounding box used for the main character was a sphere named `OBJ_Controller`. The reference for this mesh was defined at the beginning of the callback function `Append()` while in a following step also the `spawn_point`, hence the initial position of `OBJ_Controller`, and its material, named `MAT_Wireframe`, were defined. Finally the parameters of the camera were setted in such a way that this stayed closer to `OBJ_Controller`, thanks to the methods `setPosition` and `radius`, and the origin of the controller was set as target point.
The next step was to initialize all the game parameters by assigning to the controller the self-made dictionary `gameParameters`. This structure contains several parameters with user-friendly names, that manage every aspect of the game: from the forward to the lateral directions, from the speed of the movements to the states of the character.
As regards the other meshes of the game, they have their own bounding boxes, identified by the prefix `CX_name` and they possess the same material as `OBJ_Controller`. The peculiarity of this texture is that it is invisible, since this kind of meshes only pertain to a physical part and don't play a role in the aesthetic.
Once having setted up the controller, the other bounding boxes and the game parameters, it was possible to define the physics of the scene and a complex movement system. *Babylon.js* possesses a rudimental physics engine and a collision-detection system enabled by the parameter `collisionEnabled` of *scene*. In addition it was possible to define the gravity in the levels through `gravity`. After having enabled the collision system on the scene, as in the first lines of the `createLx` functions, it was mandatory to select for which meshes collisions must be checked, thanks to the parameter `checkCollisions` owned by each mesh. In this work this flag was enabled for the `OBJ_Controller` and for all the meshes whose name begun with `CX_`.
To guide the controller the function `moveWithCollisions` was used for each mesh object: it applies a translation vector to the specified mesh, keeping into account the collisions with the enabled meshes. This translates in the fact that if in the path it encounters a `CX_` it stops. The intensity and the direction of the above-mentioned vector strictly depend on the game parameters, specifically, they can be computed according to Eq. (5.1):

[1]Even if the name is *box*, the principle extends to the basic shape that best approximates its mesh.

$$(fw + rt, \quad fallSpeed * dt + 0.3 * jumpPower, \quad bw + lt) \tag{5.1}$$

where *fw, rt, bw* and *lt* are the values of the movement of the controller along the four directions, i.e. forward, backward, lateral right and left, *fallSpeed* is the gravity strength, *dt* is the time variation while *jumpPower* is the jumping power. As will be stated in Chapter 7.1, these parameters vary according to the input provided by the player and they are continuously updated through the functions `updateJump` and `update_gameparameters`. Both the functions along with `moveWithCollisions` are called inside the method `registerBeforeRender` of *scene* that manages which functions must be performed before rendering each frame.
The function `update_gameparameters` updates the values of the movements along the four directions, with respect to the camera, by updating the x and z-axes through the normalization of the difference between the positions of the controller and the camera. In a subsequent step it updates the four final values by multiplying the camera axes for the speed of the movements, hence `moveSpeed`, and changes the values of the vertical and horizontal axes, namely `vAxis` and `hAxis`, that depend on the input provided by the player. As concerning the `jumpPower` parameter, whose handles the vertical movements with respect to the gravity, it is updated inside the `updateJump` function. It monotonically diminishes the value of `jumpPower`, that was setted at its maximum at the beginning of the jump, according to the time elapsed since the beginning of the animation[2].
The controller is also endowed of four different states that stands for the action performed by the character: `isJumping` is enabled when it's jumping, `isGrounded`, enabled when the controller is touching the ground, `isFalling`, enabled when the character is falling and, finally, `isMoving` enabled when the controller is moving. In order to update these flags, it was used the `update_grounding` function to determine whether the controller is touching the ground, since using only `moveWithCollision` made not possible to determine uniquely this feature through the `state_manager` function. `update_grounding` is called inside the latter, while, in turn, it is called inside `registerBeforeRender`.
In summary, the `state_manager` works in such a way that whenever the controller is grounded and at least one of the input axes is enabled, it sets `isMoving` to *True*, otherwise to `False`. Then, it updates the other states through `update_grounding`.
In a similar fashion, the latter works by identifying whether the mesh which is in collision with, is positioned under the controller, through the drawing of a virtual line from the origin of the sphere to the lower bound. Then it checks the intersections between this line and the positions of all the collision boxes: if at least one intersection is detected then this method sets to *True* `isGrounded` while to *False* `isJumping` and `isFalling`, otherwise, if the character is not jumping, nor it's grounded then it sets to *True* `is Falling`.
In this work, for the third level, gravity was not implemented due to the willing of make realistic to fly with the magic carpet. This led to the fact that jumping parameters, states like `isJumping`, `isFalling` and `isGrounded` were not defined. In this case the `state_manager` only checks whether the controller is moving, while for the vertical movements `jumpPower` was substituted by the parameter `up`, whose value depends on a third axis, i.e. `eAxis`, that allows to increase or decrease the altitude of the controller.
All the functions mentioned in this chapter were defined at the beginning of each *JavaScript* file, while the function `registerBeforeRender` can be found at the end of the callback function of `Append()`.

[2]This was possible due to the saving performed by the game parameter `jumpStartTime` of this data, and it's continuously updated every time a jump input is received.

Chapter 6

Animations and Main Mesh

After having defined the physics of the game, the movement and collision-avoidance systems, it's possible to set the controller of the main character as *parent*. Using this technique made possible to interact with the character in an easier way, since its movements will be devoted to follow the controller.
A closer look at the main mesh reveals that the character is a complex hierarchy of simpler meshes, as shown in Fig. 6.1.
The pointers to all the meshes in Fig. 6.1 are collected in the `body_parts` dictionary, with an exception for the level three where is also present the mesh for the carpet, also with its controller setted as *parent*. One of the key steps on implementing animations is the reference frames definition. As regards the first two levels, the function `update_gameparameters`, explained in Chapter 5, turns the controller along the vertical axis to force the character to show the back to the camera, even when this moves, or to force the character to point in the input direction. In the last level, instead of turning the controller according to the input, it is moved along the x and z-axes.
In this work five different animations were implemented for the main character: the first named *idle* allows, during the first two levels, the main mesh to assume a default position when the character is not performing any action as shown in Fig. 6.2(a) while the second, the *falling* animation, represents the sets of positions assumed by the character whenever it falls, an excerpt of this can be seen in Fig. 6.2(b). The third animation is for the *jump* action and it's partially shown in Fig. 6.2(c) and (d) from two different points of view. The fourth animation was implemented for the *run* action and some excerpts could be seen in Fig. 6.2(e) and (f) again from two different points of view. The last animation was a different *idle* version for the third level and can be seen, from two points of view in Fig. 6.2(g) and (h).
The animation definition can be found inside the functions `idleAnimation`, `runAnimation`, `jumpAnimation` and `fallingAnimation`, each of these behaving in the same way. The first step in the actual implementation of the animation is the definition of an `animationGroup`, hence a structure that contains all the single animations that must be performed concurrently: this was of fundamental importance since the main character is composed by several different meshes and for creating a complete animation was also necessary to create one version for each submesh in the hierarchy. For each part was possible to define an animation through the *babylon* function `Animation()`. In particular these animations regarded the rotations of the meshes, except for the mesh root, for which was also included the position. Each of these animations was setted to 30FPS in the `ANIMATIONLOOPMODE_CYCLE` in such a way that could automatically restart whenever it will finish, when requested.
In order to define how the values of the mesh will vary during the animation, a *keyframe* assignment step was performed. An array of five frames was defined for every mesh, namely *(1,8,15,23,30)*, and for each of this the values of the rotation or of the position was setted. Once

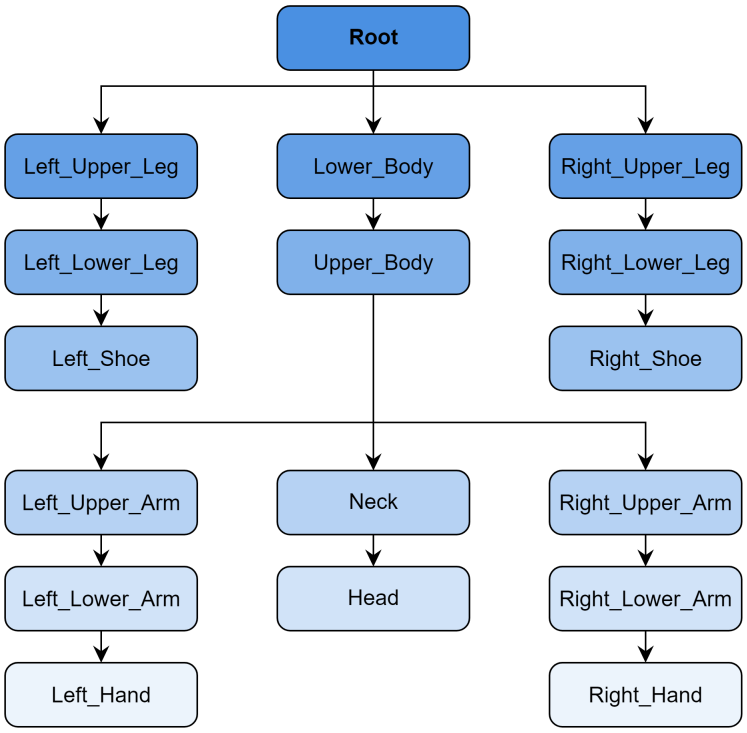

Figure 6.1. Main mesh hierarchy structure.

the frames were assigned to the animations of the submeshes, through the `setKeys` method, these were in turn assigned to the corresponding `animationGroup` thanks to the function `addTargetAnimation`. Finally the animations inside each group were normalized onto the FPS themselves and the method returned the group.
The implemented animations were then assigned to the corresponding parameter into the `gameParameters`.
In addition to the animations for the main character, in level one, were also defined the animations for the lever, as rotation, and of the wall that initially prevent the character for entering in the cave, as position. To start the animations the function `animation_manager` was defined: it is called inside the `registerBeforeRender` in such a way to be performed at each frame. It starts the correct animation according to the current controller state and stops the others. In order to obtain a smooth transition effect between two animations, at the beginning of `createLx` was enabled the *animation blending*.
The methods for the definitions of the animations and of the `animation_manager` can be found at the beginning of the *JavaScript* files, while the calls of the main character along with the use of the animation functions are inside the callback function of `Append()`.

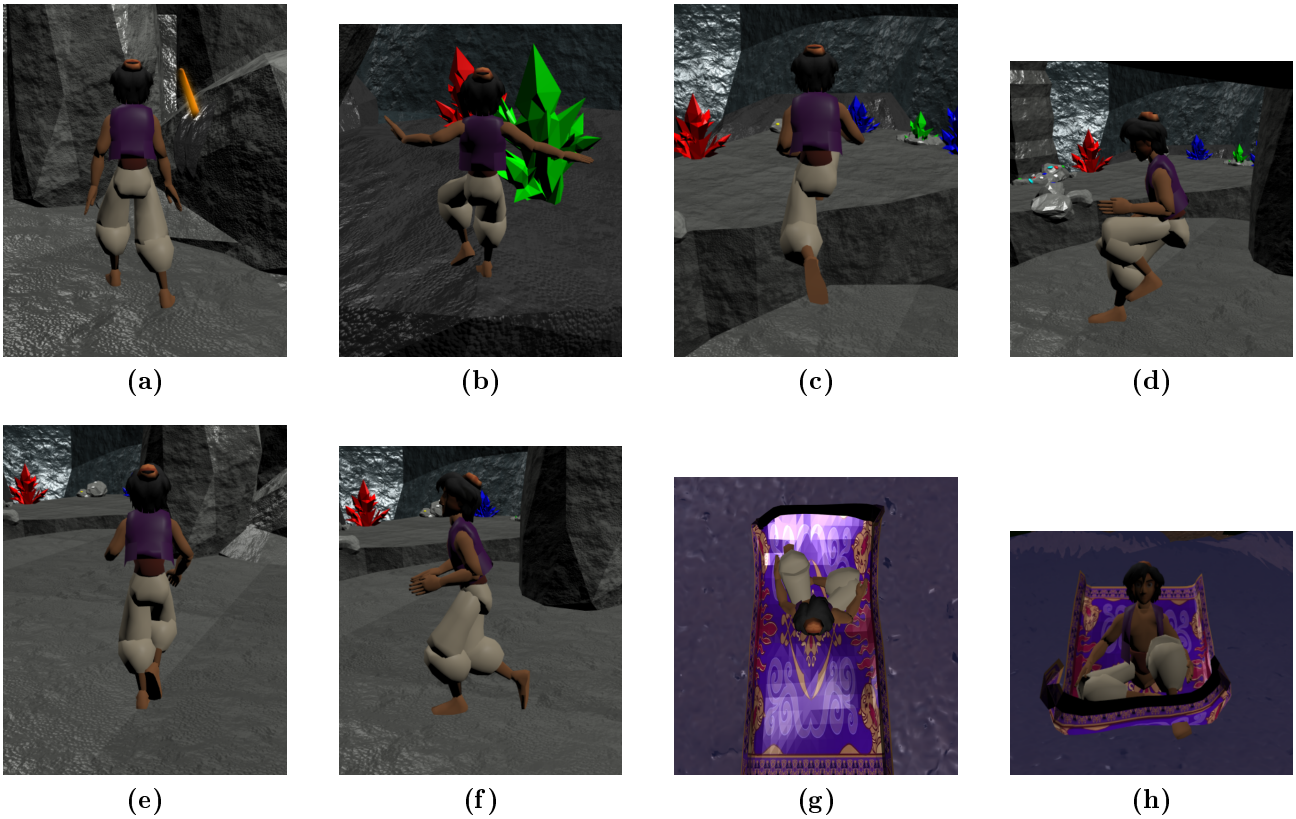

Figure 6.2. Animations excerpts: (a) idle position for the first two levels; (b) falling animation; (c) jump animation from behind; (d) jumping animation from a lateral point of view; (e) run animation from behind; (f) run animation from a lateral point of view; (g) idle position for the last level; (h) idle position for the last level from a frontal point of view.

Chapter 7

Input, User Interactions and Rendering

7.1 Input and User Interactions

The control system chosen is this work is the most known in the RPG computer games community: with the employment of the keys *W, A, S, D* for the movements, *Space* for jumping or for increasing the altitude in the third level, *E* for interacting with glowing objects and $<$ for decreasing the altitude of the carpet in the last level[1].
The input is collected through the *JavaScript* functions `eventListener`, `keyPress` and `keyUp`, that trigger an event whenever specific keys are pressed or released. In particular, when *W* will be pressed, the vertical input axis `vAxis` will be setted to *-1*, while when *S* will be pressed, it will be setted to *1*, or to *0* if both are released. In a similar manner also `hAxis` will take the same values according to the states of *A* and *S*. In the third level it's possible to see the same behaviour for the `eAxis` according to *Space* and $<$. Whenever the space bar will be pressed, in the case in which the character is not already jumping and it is grounded, then `jumpStartTime`, `jumpPower`, `isJumping` and `moveSpeed` will be updated. Finally, by pressing *E*, the animation involving the enabled glowing object will be performed. As regards the lever on the first level, this action starts its animation and, if not already performed, also the one of the wall. The lamp in the first level and the carpet in the second work as a trigger to define when the corresponding level is over.
For activating the objects, *trigger boxes* were used. The concept behind these structures reminds the one for the bounding boxes, explained in Chapter 5, but without the collisions part. When the controller intersects one trigger box, it activates an event handled by the function `triggers_manager`. According to which trigger box was activated, a variable in the game parameters will be enabled, e.g. `lever_enabled` and `lamp_enabled` for the level one and `carpet_enabled` for the level two. This function also contains the possibility to manage the application of the glowing shader for the first two levels while in the third it is only used to check whether the last ring has been traversed, in such a case the level will be ended.
Another feature of this work was the positioning of a collision box, `CX_Floor`, on the bottom of the scene as to prevent the character to fall in an endless way. When this box is touched by the controller, it triggers an event that sets the character back to the initial `spawn_point`. This mechanism is handled by the `reset` function. In the third level this latter is substituted by the function `trigger_circles` in which the parameter `curCircles` was added. This allows to keep track of the current last traversed ring, and whenever a future ring will be crossed, the function checks whether it is the successive of the `curCircles` one, in this case the material of the ring will be switched to an invisible one, otherwise the character will restart from the beginning `spawn_point` and all the materials of the already traversed rings will be restored.

[1] For deeper details on the commands please refer to Appendix: User Guide.

All the functions described in this chapter can be found in the beginning of the *JavaScript* files while the `addEventListener` ones inside the callback function of `Append()`.

7.2 Rendering

Once having completed all these steps, the functions `createLx` will return complete scenes, composed by meshes endowed with materials, by musics, lights, robust physics and movement systems, input structures and systems for environment interaction. In order to run a scene, the method `render()` must be evoked inside the `runRenderLoop()`, one of the game engine.

Chapter 8

Conclusions

This work had the purpose of designing and implementing a complex multi-level interactive game by exploiting most of the techniques pertained to the field of *Interactive Graphics*.
The construction of all the aspects needed to run such game was challenging and required a considerable amount of time. One of the main features of this work was the number of resources used: knowledge about the *HTML* language was required along with the one of *JavaScript*, especially for the *babylon* libraries. Notions of *CSS* were exploited but also the understanding of the usage of the *Blender* suite was essential.
Not only the actual implementation part was important but also the time spent in the study and in the research of customized solutions that allowed to make this game the most similar to Aladdin's world as possible.

Appendix A

Users guide

In this appendix a walk-through guide for players of the game will be provided.
The game implemented in this work is inspired by the *Disney* story: *Aladdin*. It consists on three different levels representing parts of the adventures of the main character whose the name of the film is inspired of. The interactive nature of this game allows the player to command *Aladdin*, the character, thanks to a sophisticated animation system that provides the possibility to run, jump as long as it manages the fall and to remain idle. Specifically, the command structure is thought to be used by both hands and varies among the levels. For the first and second layers:

- `W`: by pressing this letter the character will go forward towards the centre direction of the point of view, even if it will change during the execution of the action, until it stays pressed.
- `S`: by pressing this letter the character will go backward by turning in front of the camera until the key is pressed.
- `D`: by pressing this letter the character will move right, perpendicular with respect to the camera centre axis, until pressed.
- `A`: by pressing this letter the character will move left, perpendicular with respect to the camera centre axis, until pressed.
- `Space`: by pressing the space bar the character will jump.
- `E`: by pressing this letter in the proximity of a glowing target object, it starts an animation or it starts the loading of the next level.

For the third layer the player can use:

- `W`: by pressing this letter the carpet will move forward, until pressed.
- `S`: by pressing this letter the carpet will move backward, until pressed.
- `D`: by pressing this letter the carpet will move right, until pressed.
- `A`: by pressing this letter the carpet will move left, until pressed.
- `Space`: by pressing the space bar the carpet will go up.
- `<`: by pressing this symbol the carpet will go down.

For all the levels the use of the *mouse* is of fundamental importance: by clicking and dragging the pointer is possible to change the direction toward which the character is moving, or the point of view, even in a continuous fashion, i.e. also while performing another action. With the scroll wheel is possible also to regulate the zoom on the scene.
As in the beginning of the film, *Aladdin* is trapped inside the *Cave of Wonders* with the purpose of retrieving the genie lamp to exit from this landscape. In level one the player must arrive near the lever and press `E`. Once the wall will go down, it will open the cave full of gemstones. After having jumped on all the platforms of increasing height and crossing the wood bridge, it's possible to collect the lamp by pressing again `E`. This will end the first level and the second one will be loaded.
The second level is inspired by the scene of the film where *Aladdin* tries to escape from the guards by jumping on the roofs of the houses in the middle of *Agrabah*. In a similar way the character must jump over all the roofs without falling. It must go through the corridor and jumping on the flying carpets. By stepping on narrow wood boards and cane platforms, it's possible to reach the last house where the magic flying carpet will glow to allow the access to the third and last level, i.e. once having pressed `E`.
For both these levels, whenever the character falls, the game will restart from the beginning of the corresponding level.
The third, and last, level is inspired from the worldwide trip that *Aladdin* and *Princess Jasmine* made on the flying carpet. For this reason, the character, on the magic carpet, must go through nineteen glowing rings, arranged on four different scenarios, linked in a gradient fashion: a winter mountain, a forest with a lake, a desert and the sultan palace where the game will finish. If the player goes through different rings not consecutively, the game will restart at the beginning of the level.
After the ending of the third level the game will resume on the main menu page. In the following rows there will be presented the instruction on how to setup the game and how to navigate in the main menu.
After having launched the `index.html` file, you will see, in the bottom-centre of the black page, the main menu composed by three entries. Each of these will load a different page: once having dragged the pointer over the chosen box, until it started to glow, you have to click on it:

- *Main Story*: will load the whole game, starting from the first level and automatically loading the subsequent ones, when completed, until the game is not over. This menu entry will remain available during the gameplay of all the levels in order to skip one of those or to explore different parts of the menu options.

- *Single Level*: by clicking on this item, another menu will appear containing the name of the single level that is wanted to play or the possibility to get back to the main menu.

- *Commands*: selecting this option will open a page where all the commands to play the game will be disclosed and with the possibility to get back again to the main menu.

For all the technical details on how the game works, please refer to the main corpus of this report.