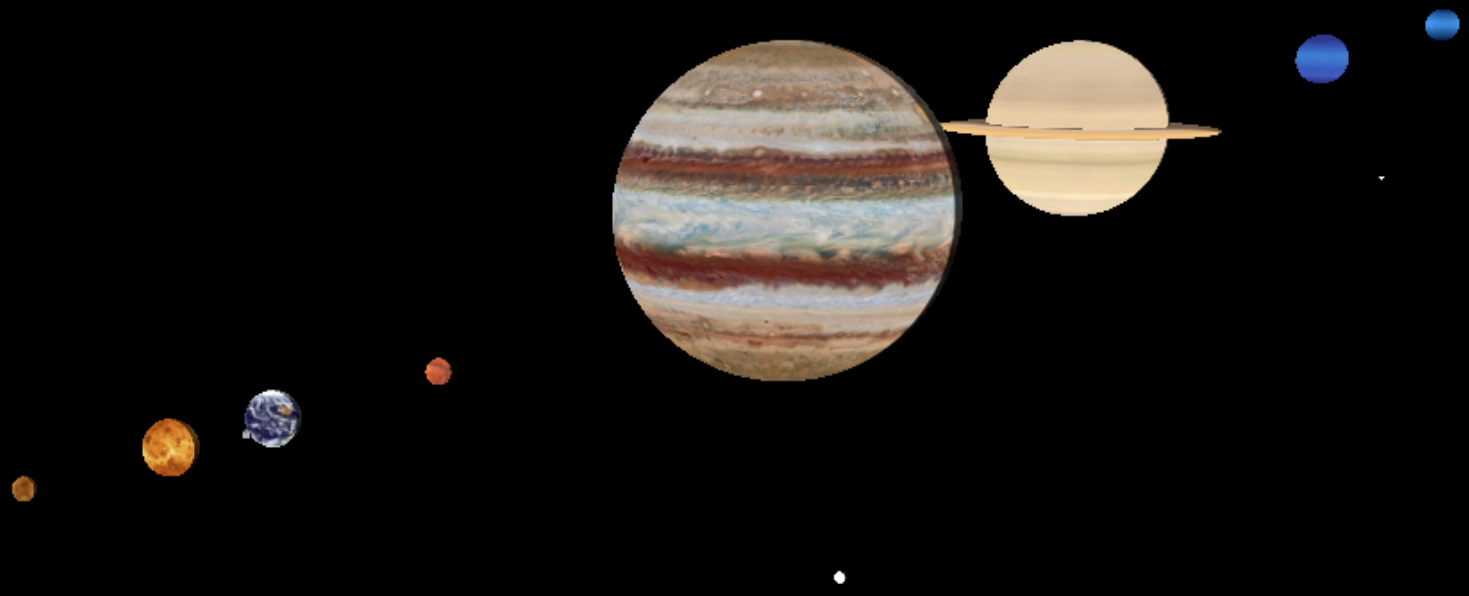
SpaceY

Interactive Graphics Course

Sapienza, University of Rome - a.a. 2019/2020



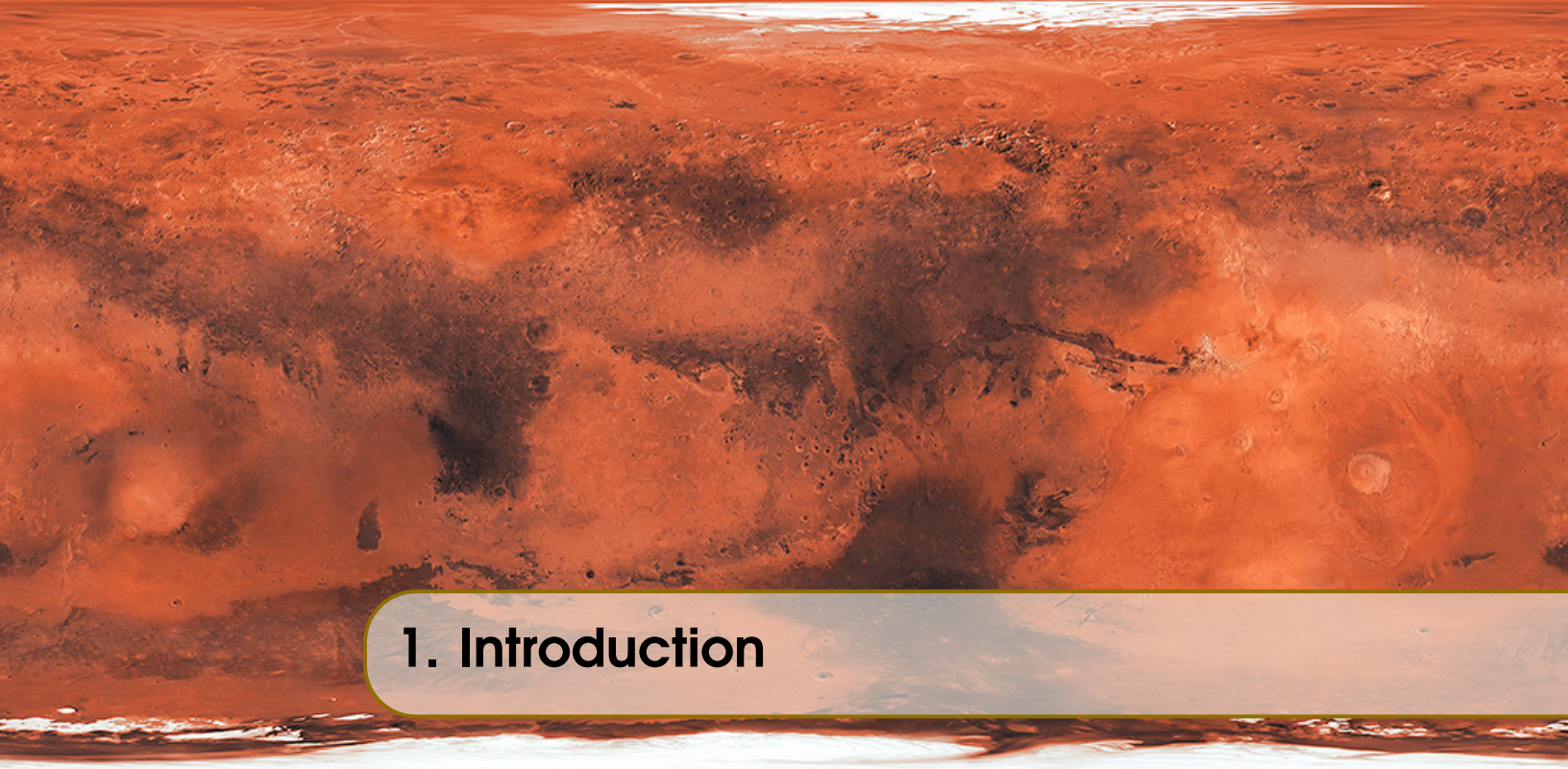
Final Project

Members:

Akash Garg, Simone Rossetti

{garg.1892171, rossetti.1900592} @studenti.uniroma1.it

Contents



1. Introduction

1.1 Motivation

The following work presented as the final project for the Interactive Graphics course takes inspiration from the recent space missions and projects being carried out by a number of space organizations around the world. The members of the team *Space Y* share a common interest in space exploration field and therefore decided to apply all what they learned in the *Interactive Graphics* course to create this multi-mission interactive game.

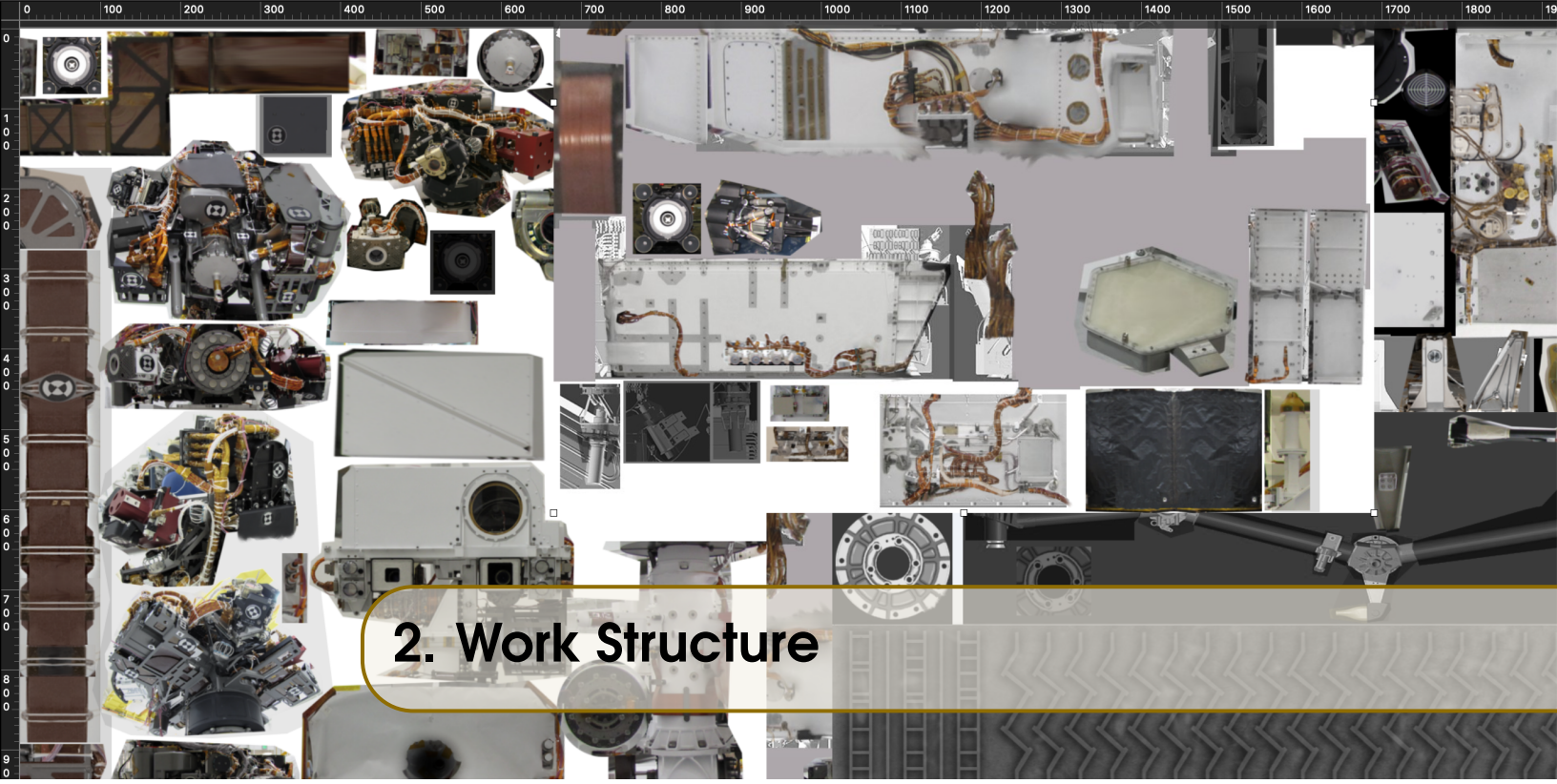
> I don't think the human race will survive the next thousand years, unless we spread into space. –*STEPHEN HAWKING*

1.2 Overview

The game touches upon various concepts of the field of *Interactive Graphics* like modeling, scene creation, animation, physics setting etc. The game consists of two levels which defines two different missions, *Mission Moon* and *Mission Mars*. The two missions are marked with different modeling and environment as well as quite distinct tasks to be carried out by astronauts. The user controls the activities of the astronauts like walking, turning, stopping, grabbing and throwing an object with the help of interactive keys. *Mission Moon* consists of two astronauts namely, Astronaut A and Astronaut B, an oxygen tank. Astronaut B has run out of oxygen. The user controls the Astronaut A and the task is to pick an oxygen tank lying near to Astronaut A on the Moon crater. Once having grasped the oxygen tank the user has to carry it to the fellow astronaut and save his life. On the other hand *Mission Mars* has an astronaut and a rover on the Mars surface. The rover is malfunctioning and needs to be repaired. The astronaut controlled by the user, therefore, has to reach the rover and start repairing it.

The report is structured as follows, in Chapter 2 we discuss about the Work Structure of the project. In subsection, Files Organization, all the folders and the contained files in our github repository are talked about whereas the subsection, Libraries Used, lists all the libraries that were

employed to accomplish the task in hand. Chapter 3, Scene Creation and Design, introduces all design related aspects and the modelling phases involved in the project. We try to provide a brief description of all the major elements like skybox, lights, cameras, shadows, heightmaps and ground, sounds, GUI and optimizer. Chapter 4, Physics, Movements and Game Parameters, focuses on a detailed explanation of the physics and the movement related aspects in the game. A number of parameters used to set up the desired movement behaviour of the objects in the project are explained. In Chapter 5, Main Mesh and Animation, we talk about the main mesh as well as various Animations used. This includes riggifying the main mesh, animating the corresponding skull as well as camera and text animations. Chapter 6 labelled as Input, User Interactions and Rendering describes the user interface of the game and how the input commands are integrated to produce the desired movements and activities. Along with this the chapter also explains about the overall rendering of the game. We conclude with chapter 7, Conclusions, and talk about the message drawn and things learned during the project.



2. Work Structure

2.1 Files Organization

The repository of the work is present at the github link [1]. We have tried to organize the files and resources in four different folders that make locating and navigating through them much easier. The top level files and folders are as follows:

- *babylon*: this is the most crucial folder from the point of view of the code. It contains all the *JavaScript* and *HTML* files associated to the project. The first file in the hierarchy is *index.html*. It is the starting point of the game and loads the menu. The associated *Javascript* file, *index.js*, contains the menu creation code as well as captures the user preference to select the mission. Depending on the user preference one of the two *html* file associated to either of the two missions is called. Eventually *game.js* is evoked which contains the scene creation and user interaction code.

- *images*: it contains the images used in the project mostly used as texture to the meshes for example planets displayed in the menu, ground heightmap and it's texture etc.

- *models*: this folder contains all the 3D models useful for the project. Mostly models imported from *NASA* website are put in this folder. Models which required animation to be applied, like that of astronauts, were rigged before that.

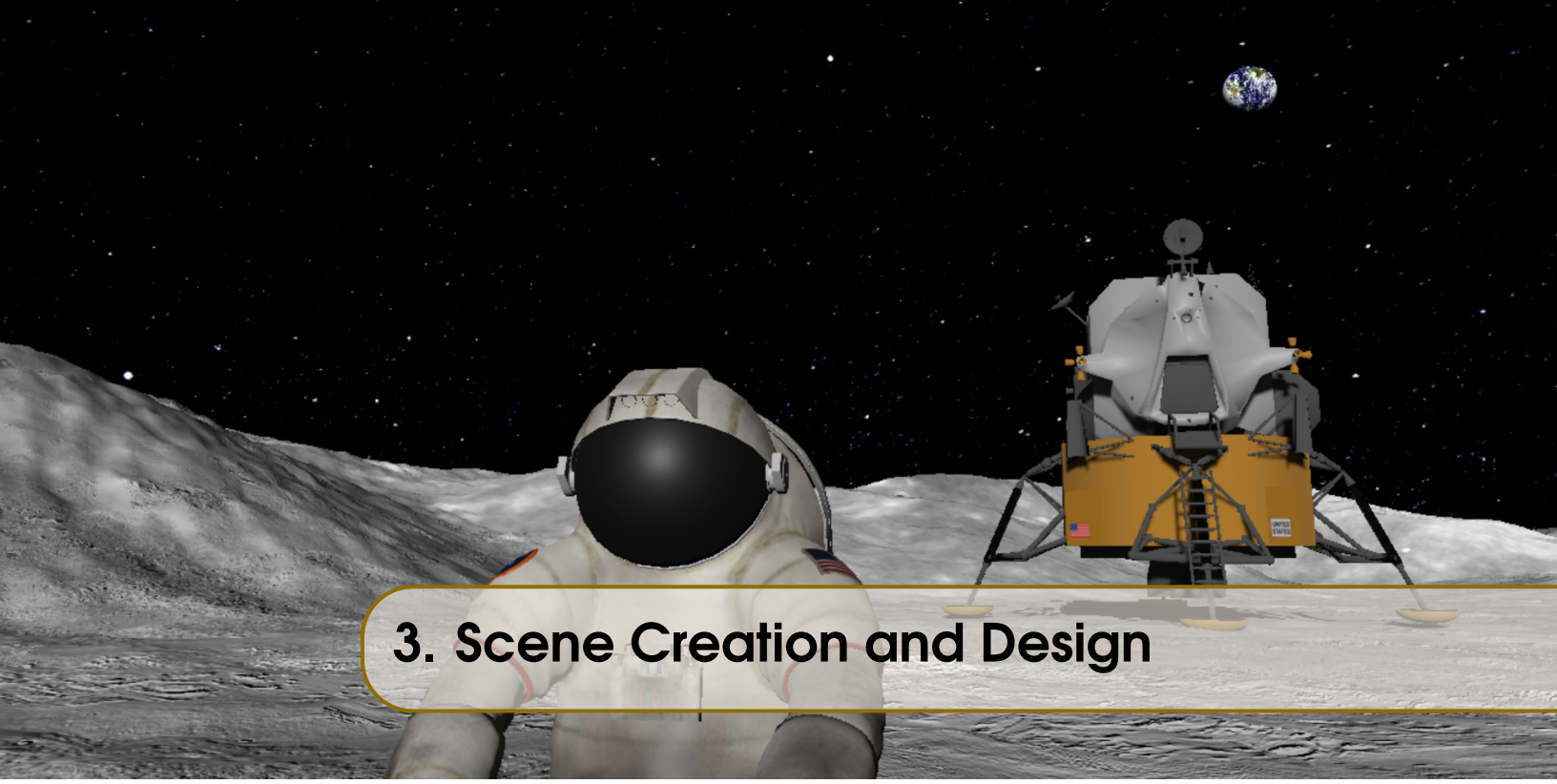
- *sounds*: it contains the sound files played during the game. The menu of the game has a sound file whereas the two missions have associated distinct sounds being played in the background.

[1] https://github.com/SapienzaInteractiveGraphicsCourse/final-project-spacey

2.2 Libraries Used

All the libraries used in the project come from the *Babylon.js* package. The package is wholesome and provides the necessary functionality because of its comprehensive library stack. The following libraries were included in the *html* files belonging to the two missions as well as the menu (i.e. index.html, mission1.html, mission2.html).

- *babylonjs.materials.min.js*: it is used to provide materials for the sky, planets, and other meshes in the game.
- *babylonjs.loaders.min.js*: this library is used to import complete scenes or single meshes from external files. In this work it was used to import the rigged as well as non-rigged meshes in the *.blend* format.
- *babylon.inspector.bundle.js*: *Babylon.js* inspector is a visual debugging tool created to help pinpoint issues related to a scene.
- *babylon.gui.min.js*: this GUI library is an extension used to generate interactive user interface. It is build on top of the DynamicTexture.
- *babylonjs.serializers.min.js*: this library is an extension that we used so as to serialize Babylon scenes.



3. Scene Creation and Design

The file `game.js` implements the scene creation and the logic of the game. Instead `mission1.js` and `mission2.js` files encase the necessary global variables required by the respective scenes, which are needed to specify distinct values for each common element such as: meshes positions, textures locations, materials, maps, sounds, texts, etc. Global variables are loaded before the scene.

The game has a fixed template: the astronaut should move on surface of different planets to achieve a goal. Therefore the environments are clearly different depending on the planets but the setup and the elements which compose the scene are similar (i.e. sky, ground, sun, etc.).

3.1 Scene and engine

It's important to specify what a scene is. First of all when we use `BABYLON.CreteScene()` we tell to the framework (*Babylon.js*) to create a scene class bin which we can fill with lights, cameras and meshes. A scene always require a camera to be active, otherwise we have no scene. An engine, instead, is created when we use `BABYLON.Engine()`, it is a class responsible for interfacing with all lower-level APIs such as WebGL, inputs and audio. Therefore the engine is the one responsible for the rendering of our scene.

3.2 Camera

The type of camera we use is called `BABYLON.ArcRotateCamera()`. You can see its working looking at the figure 3.1. This camera always points towards a given target position and can be rotated around it with the target as the centre of rotation. It can be controlled with cursors and mouse, or with touch events. Think of this camera as one orbiting its target position, or more imaginatively as a spy satellite orbiting the earth. Its position relative to the target can be set by three parameters, alpha the longitudinal rotation, beta the latitudinal rotation and radius the distance from the target position. The camera target is locked to the astronaut in order to follow him during motion.

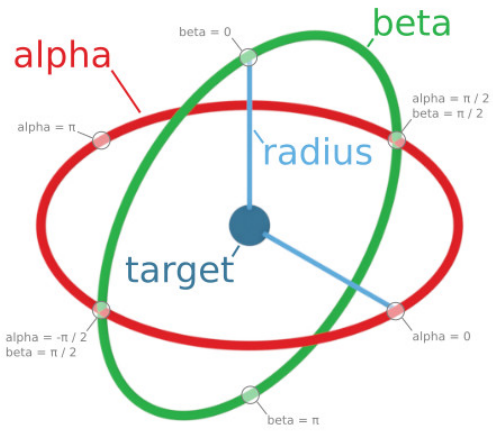


Figure 3.1: Picture which shows how `BABYLON.ArcRotateCamera()` works. Source taken from `https://doc.babylonjs.com/babylon101/cameras`

3.3 Environment

It is not simple to build a realistic scene. Furthermore it is not simple to make it light and fast on general devices. Our scene is the result of the compromise between efficiency and realistic rendering. The whole scene encloses a lots of details which we are going to analyze in this chapter and which make the scene fascinating to see but computationally a bit heavy. That's why we had to adopt a lots of optimization techniques (which we are going to treat as the last topic of this section) to reduce the memory footprint, but the quality too. Fortunately, our optimization is adaptive, thus the better is the device, the better the result, obviously.

3.3.1 Lights

Lights are what really create the scene, using those in a proper way can really improve the result. Assuming we are on Moon or Mars, the only light source is the sun. We have no atmosphere or just a little of it, the sun shines very hard and both the scenes should look very barren.
For this aim we used a very intense `BABYLON.DirectionalLight()`, which is the unique light in *Babylon.js* that simulate the sun light. A directional light is defined by a direction. The light is emitted from everywhere in the specified direction, and has an infinite range.

Now, considering that we are in a game, and that we want to keep as realistic as possible the scene, we wanted to add one more light called `BABYLON.HemisphericLight()`. This light simulates the ambient environment light, so the passed direction is the light reflection direction, not the incoming direction. Thus we added a very feasible light directed from the ground to the sky, which creates a more pale and bright effects and which permits to the user to see also the meshes that are under a shadows. In a kind of sense we added this global light to contrast the intensity of the shadows produced by the sun light.

3.3.2 Volumetric light scattering

Then, we added lights, result is nice, but there is still something missing. We forgot the mesh of the sun! Then, let's add a white sphere at a very high point in the space. Now one would say that we are done. Doesn't look too flat? What a nice trick would be add to the scene a sun rays scattering looking? Very nice, and we did it. Thanks to `BABYLON.VolumetricLightScatteringPostProcess()`, we applied a post-process that compute the light scattering according to our light source mesh, the sun, and attached to the camera. Light comes exactly from the sun position and is directed towards the ground, now, has the realistic effect of a shining, ray scattering sun too. See figure 3.2.

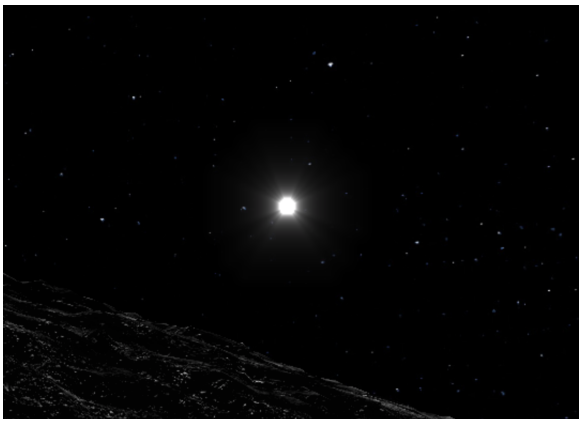


Figure 3.2: Volumetric light scattering of the sun mesh in lunar mission.

3.3.3 Shadows

As we well know already, WebGL still does not support automatic shadows mapping. But what realistic scene would be with lights and no shadows?
Luckily *Babylon.js* comes in our help. The framework has a very good shadow mapping tool: `BABYLON.ShadowGenerator()`, which creates a map of your scene generated from the light's point of view.

To add a mesh into the shadow casting we used the two functions `shadowGenerator.addShadowCaster(*)` and `shadowGenerator.getShadowMap().renderList.push(*)` which permits to generate a shadow from the selected meshes. To let a mesh receive shadows instead we had to enable the property `*.receiveShadows`.

We made a lot of tests, considering that there are a lot of mapping types, in order to obtain the best result. Plus we encountered a lot of difficulties dealing with exponential shadows maps and Poisson sampling because of the very far light source of the sun.

At the very first moment, shadows generated from a far directional light looked intractable. In particular all the meshes were very dark. And the shadows were very imprecise and full acnea points. But now the result is awesome, in particular using the last available shadow mapping type called `shadowGenerator.usePercentageCloserFiltering`, which takes benefit from the new hardware filtering functions available in WebGL2 and produce a smoother version of Poisson sampling. It falls back to the standard Poisson Sampling when WebGL2 is not available on the target device. The result is excellent working with very far directional light sources. See figure 3.3.

3.3.4 Skies and fog

To simulate the sky we created a very big low poly spherical mesh (to not affect the memory performance) using `BABYLON.CreateSphere()`. After we applied a `BABYLON.StandardMaterial()` covered by a reflection texture (`BABYLON.Texture()`) (a starry sky for Moon and an orange gradient for Mars which can be found in the `\images` folder) which are mapped to the sphere with a fixed equirectangular mapping (`BABYLON.Texture.FIXED_EQUIRECTANGULAR_MODE`), and because the framework applies back face culling by default, we had to turn it off, in order to reflect the texture inside the spherical mesh of the sky.

In Mars mission, once we created the full scene, we decided to add a linearly attenuated yellow fog effect to simulate the dry atmosphere of Mars. See figure 3.3.

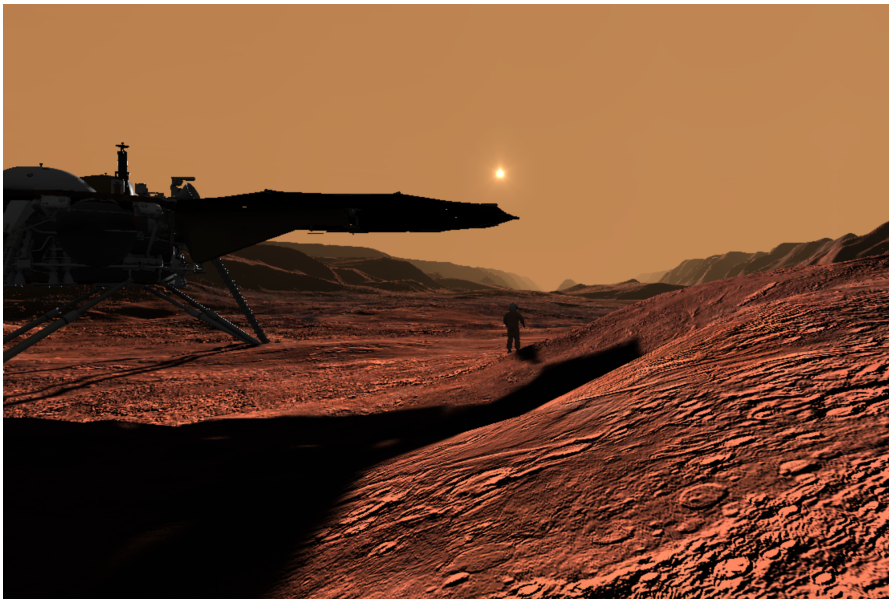


Figure 3.3: Volumetric light scattering, shadows, fog, textures and bump map.

3.3.5 Heightmaps and grounds

We encountered a lot of difficulties regarding the ground meshes to use. In the very first moment we decided, in order to maintain the scene as much realistic as possible, to adopt meshes of famous lunar and martian sites, captured and reconstructed by NASA into 3D models and now available for everyone at this site `https://nasa3d.arc.nasa.gov/models`. See figure .
This choice has been particularly expensive, in terms of time, space and memory performances too. To deal with these meshes, we adopted a software called *Blender* (which is a very famous open source in 3D graphic, if you don't know it, you can find it here: `https://www.blender.org`).

All of these NASA 3D sites models are in `.obj` of `.stl` format, no textures, no materials, no bump maps, just vertices, hence, we had to work a little bit with those.
The first solution to reduce the heavy memory impact that the introduction of these huge meshes had into the scene, has been remove all those vertices and faces which do not contribute into the scene, obtaining a thin convex hull. But the real turning point has been discovering that *Babylon.js* has a really nice method for optimizing vertices storage of the subclass `BABYLON.Mesh.GroundMesh()`,

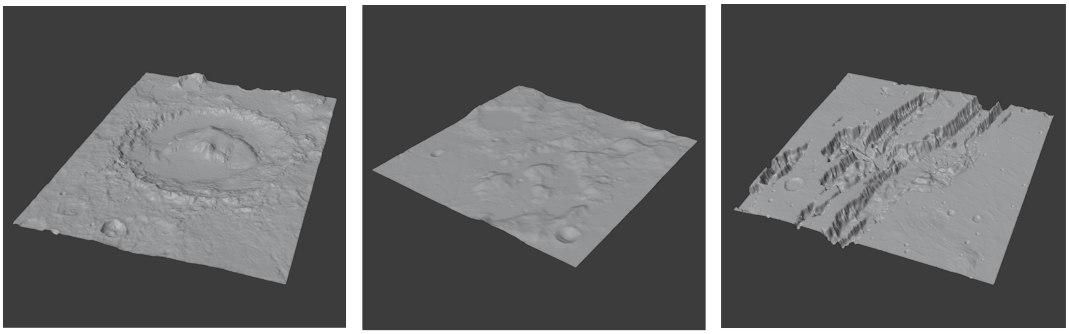


Figure 3.4: The first image shows *Gale Crater* on Mars, is the mesh we originally adopted and which we particularly worked on, this one has been abandoned and substituted with the second and third meshes, *Apollo 17 Landing Site* and the *Valles Marineris*, after replaced with their heightmaps.

which is `*.optimize()`.
This function updates an octree to help to select the right submeshes for rendering, picking and collision computations.

Thus, we decided to move to `BABYLON.Mesh.GroundMesh()`. And what's the best way to keep a mesh without storing all the vertices? Easy, height maps.
We used *Blender* software for color the meshes with a gray scale gradient along the height and rendered a picture from the top.
This is how we obtained the height maps for the ground, to reproduce as close as possible the original NASA meshes. See figure 3.5.

Once we had the images, we used `BABYLON.Mesh.CreateGroundFromHeightMap()` to obtain the ground mesh and use optimization function to make it light. Nice trick! Obviously after that, we applied to the two meshes (one per mission), a new `BABYLON.StandardMaterial()` and using a `BABYLON.Texture()` for each different textures (`*.diffuseTexture`) and different bump maps (`*.bumpTexture`) we created nice lunar/martian rocky grounds for each mission. Plus we controlled the intensity of the bump mapping using the parameters `bumpTexture.level`, `bumpTexture.uScale` and `bumpTexture.vScale`. Which gave to the ground a more realistic aspect.
In order to obtain the desired bump mapping we converted some lunar and martian surfaces pictures into bump figure, thanks to this free online tool `https://cpetry.github.io/NormalMap-Online`. See figure 3.5.

3.4 Imported meshes and materials

Babylon.js provides a lot of efficient tools to import external meshes with various kind of extension (such as `.obj`, `.gltf`, `.glb`, `.3ds`, `.stl`, `.babylon`, etc.). We chose to convert all the models into `.babylon` format, in order to exploit as much as possible the compatibility with the framework. All the meshes we used are available here: `https://nasa3d.arc.nasa.gov/models`.
In particular we adopted the *Z2 Spacesuit* and the *Apollo Lunar Module* in the first mission, and the *Advanced Crew Escape Suit*, the *Insight Cruise Lander* and the *Curiosity Rover* into the second one. The height map we made picture the *Apollo 17 Landing Site* and the *Valles Marineris*. See figure 3.4.

We really encourage the reader to read more about these topics, and to read at least the descrip-

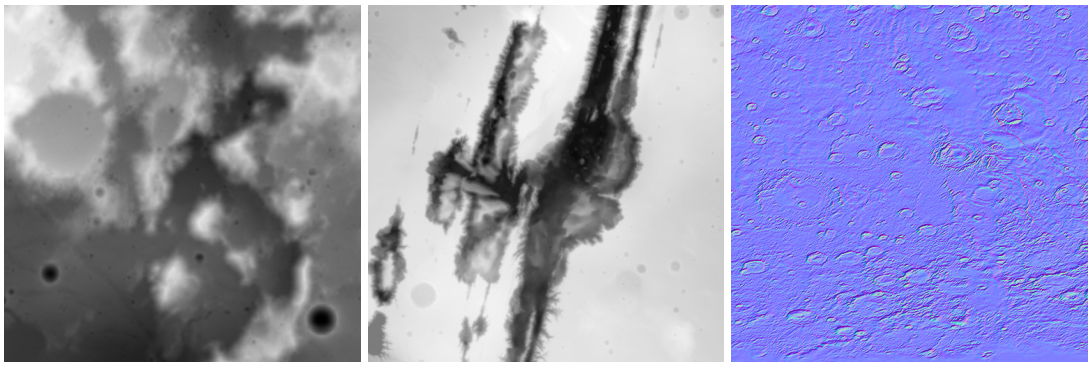


Figure 3.5: The first two images are the height maps used to build the ground meshes respectively for moon and mars missions; the third image is a piece of bump map we used.

tions related to these sites, object, missions, etc. at the NASA website.

Using the function `BABYLON.SceneLoader.ImportMesh()` we could import various kind of meshes. Our scenes are made by five imported meshes: two different astronaut, two different landers, a rover, and some code built meshes, the sun, the earth, an oxygen cylinder and some hoops.

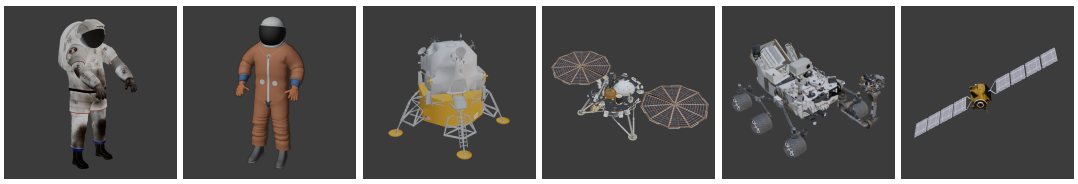


Figure 3.6: These figures shows the meshes we got by `https://nasa3d.arc.nasa.gov/models` and imported into the game. From left to right: *Z2 Spacesuit*, *Advanced Crew Escape Suit*, *Apollo Lunar Module*, *Insight Cruise Lander*, *Curiosity Rover* and the *Deep Space 1*.

3.5 Musics and sounds

In to propagate from one place to another, the sound requires a medium or a fluid to move through. The air on Earth allows sound waves to move from one point to another. However, on the Moon atmosphere is very thin. Thus there is no sound on the Moon. The only sounds that an astronaut can hear from there are the communications from the control station and with the fellows. Thus we tried to stay as much realistic as possible and included in the lunar scene (and in the menu too) a low background sound which is actually electromagnetic noise. These electromagnetic waves are called *Chorus Radio Waves*. The electromagnetic waves are a type of natural radio waves, vibrations of electric and magnetic energy occurring at the same frequency as sound. They are strongly audible from the Earth's atmosphere.
Instead on Mars, we mostly can hear earthquake sounds. All of those effects and sounds are available at this link: `https://www.nasa.gov/connect/sounds`.

To reproduce and customize the dialogues and technical communications between the control station and the astronauts we used a more complex approach.
First of all we wrote and translated into audio files the dialogues, thanks to an online TTS (text-to-speach) reader (you can find it here: `https://wideo.co/text-to-speech`) with various voices

templates.
After that we edited the files using a free open source software named *Audacity* (you can find it here: `https://www.audacityteam.org`). Here we equalized the sounds to give a telephone call effect, removing frequencies which are under 300Hz and up to 2KHz. We added background white noise, delays and different kinds of distortions to simulate radio transmissions.
Sounds has been imported into the game thanks to the library `BABYLON.Sound()`, which permits among many things to manage the sounds, to give a direction to the audio and to loop over it.

3.6 GUI

Babylon.js provides few tools to interface with users, they all come from the library `BABYLON.GUI` which is an extension you can use to generate interactive user interface. It permits to organize and include into the scene 2D and 3D elements, such as buttons, panels, texts and images. This is how we created the navigator on the bottom right of the scene, all the messages and interfaced with the user interactions. Plus we applied to those elements some animations in order to create a game template to use in each scene. See figure 3.7.

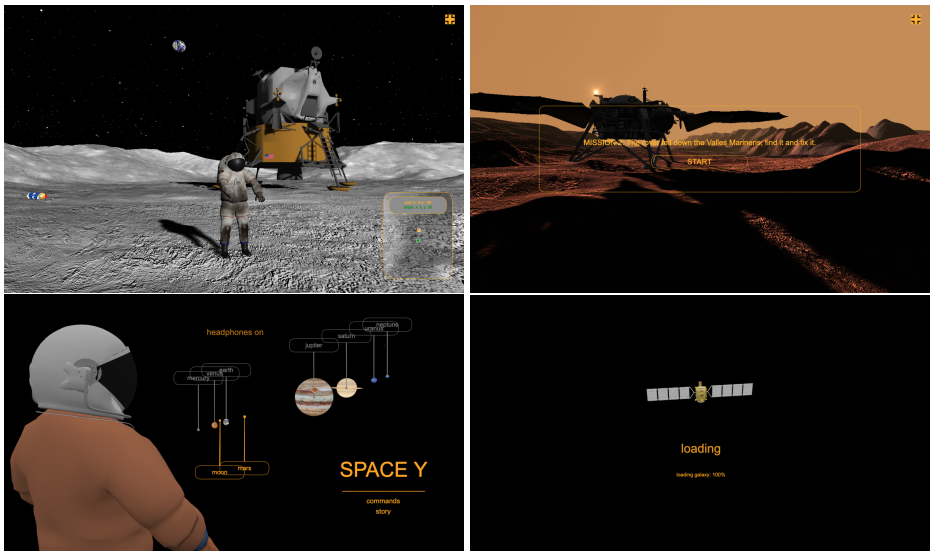


Figure 3.7: Buttons, panels, texts and images.

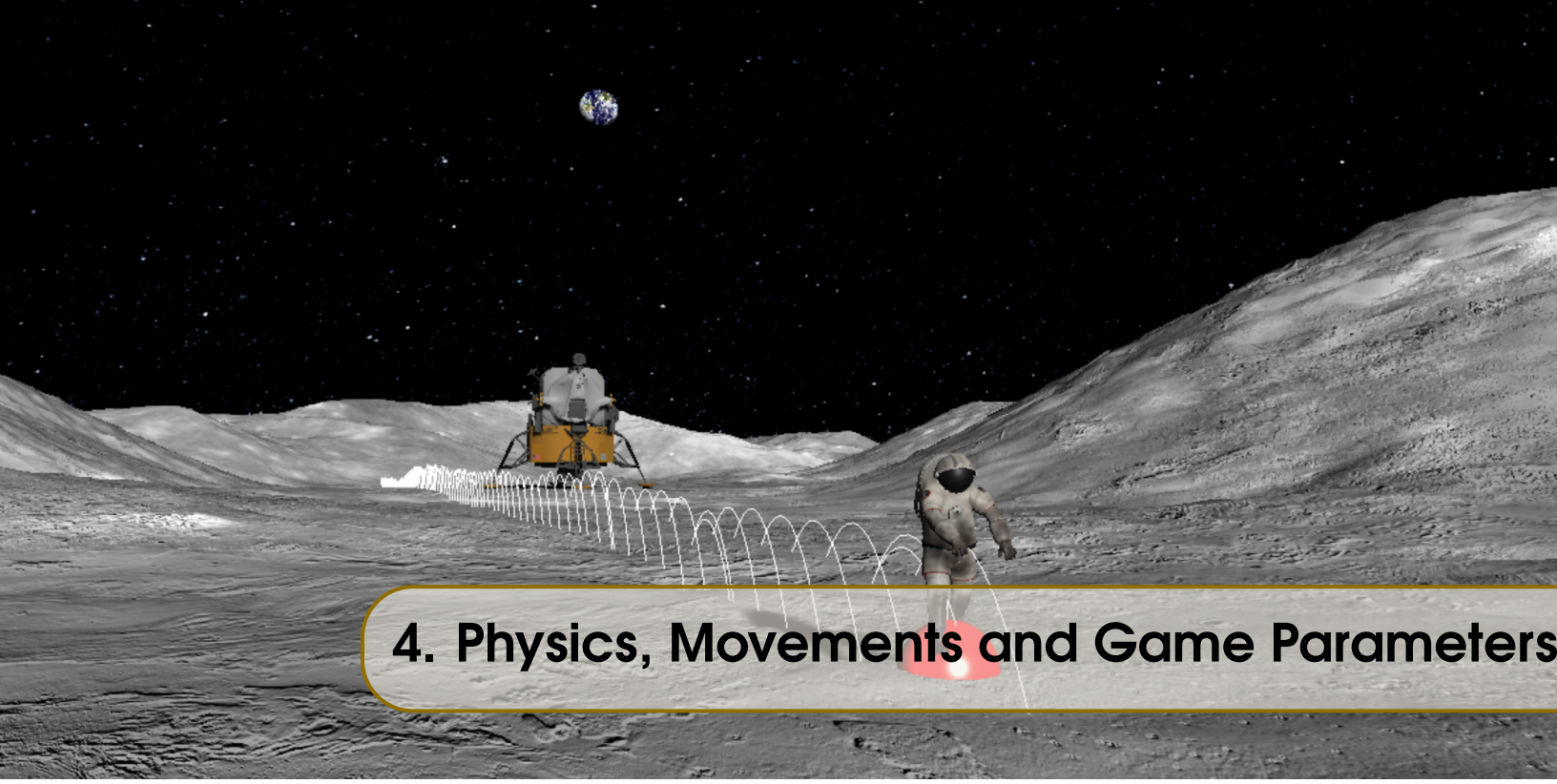
3.7 Optimization

Rendering a scene on a browser is a great experience because you can reach a lot of different users and hardware. But the main associated caveat is that you can encounter very low-end devices. The principal objective is to reach and guarantee for each device the canonical 60 FPS (Frames per second), which means guarantee a good graphic experience. As we previously said, we adopted multiple techniques to lighten the scene, i.e. we made use of built in height maps optimization instead of import huge amount of vertices for ground meshes.
Now we will go into detail by talking about the `BABYLON.SceneOptimizer()` tool, which is designed to help the engine to reach a specific framerate by gracefully degrading rendering quality at

runtime. We have available a set of optimization options to apply in a default of customize order. As soon as the target FPS is reached, the `BABYLON.SceneOptimizer()` stops. There are different layers (or passes) that are applied one after another. The `BABYLON.SceneOptimizer()` pauses between each layer to ensure a stable FPS, for measuring. To create a custom optimization buffer we can assign to each option a determined priority level. The optimizer will go into the optimization process per priority. Our custom buffer is made by the following five levels:

- level 0: `BABYLON.TextureOptimization()`: This optimization tries to reduce the size of render textures. This optimization has a huge impact.
- level 1: `BABYLON.HardwareScalingOptimization()`: This optimization increments or decrements the value of hardware scaling. This is a really aggressive optimization that could really help if you are GPU bound.
- level 2: `BABYLON.PostProcessesOptimization()`: This optimization disables post processes. This optimization has a huge impact too.
- level 3: `BABYLON.RenderTargetsOptimization()`: This optimization disables render targets.

This optimization procedure allowed us to reach good performances in low memory conditions preserving the quality and usability of the game. Thus we want to aware the user that it's possible that in case of cheap memory devices the graphic quality may decrease.



4. Physics, Movements and Game Parameters

4.0.1 Collision System

We use built-in collision system, a lightweight system, which is basically doing collisions between ellipsoids and meshes. The *ellipsoid* for the astronaut mesh is adjusted to fit the character properly with the help of parameters *boy.ellipsoid* and *boy.ellipsoidOffset*.

Collision detection for the scene is enabled by setting the parameter *scene.collisionsEnabled* to *true*. Now for each mesh for which collision detection property needs to be activated we set the parameter *checkCollisions* for that particular mesh to *true*.

4.0.2 Gravity setting

The movement of astronaut in the scene is implemented using basic parabolic equation of motion. A special attention is given to the fact that our character is operating on the Moon and the Mars and not on the Earth. This affects particularly the walking behaviour of the character since gravity on both, Moon as well as Mars, is less than the gravity on earth. We set the gravity value for scene using *scene.gravity*. Moreover the parameter *GRAVITY* holds value in m/s^2, this therefore is converted to $coordinates/s^2$ by multiplying *GRAVITY* with *SCALE_FACTOR*. The latter parameter denotes the ratio of height of the astronaut in coordinates to the height in meters. The height in meters is taken as constant in the parameter *HERO_HEIGHT*.

4.0.3 Parameters

The next step is to initialize all the game parameters which define the state of the astronaut as well as the overall game. First of all we set the parameter *MISSION_STATUS* to 0 which is 1 only if the mission has been successfully completed by the astronaut and 0 otherwise. A specific flag, *flagImp*, is set when the user presses the key 'W' whereas by pressing 'S' the user brings the *flagImp* value back to 0. The parameter is used to identify if the astronaut should be provided with velocity/impulse to continue its walking behaviour or be stopped. Similarly whenever the user presses key 'Q' or 'E' an event is triggered and we also set the corresponding flag, *flagQ* or *flagE*, respectively. Next, the parameter *flagGb* signifies the state of the oxygen tank. It acquires the value 1 if the tank has

been grabbed by the astronaut in Mission Moon and 0 if not grabbed. In Mission Moon when the parameter *flagGb = 1* and the astronaut controlled by the user reaches close to the struggling astronaut the mission is considered complete which sets the above parameter *MISSION_STATUS = 1*. Similarly in Mission Mars when the astronaut is close to the malfunctioned rover and the *flagE = 1* the parameter *MISSION_STATUS = 1*. All the parameters are update in *scene.registerBeforeRender()* function which is executed before rendering each frame.

4.0.4 Movements

Thanks to *Babylon.js* inbuilt function *mesh.moveWithCollisions(velocity)* we can move a mesh with with desired velocity. This function will try to move the mesh according to given velocity and will check if there is no collision between current mesh and all meshes with *checkCollisions* activated. We denote a separate parameter *boy.speed* which is a 3 element vector signifying speed of astronaut in x,y,z axes of world coordinates. As discussed earlier, because of nature of walking on the Moon and the Mars crater we try to impart a parabolic behaviour to motion of astronaut. Now, in order to implement the said behaviour there arises a need to determine the height of the astronaut above the surface. We make use of the concept of *ray* in *Babylon.js* in which a ray originating from the astronaut's feet is directed in the direction (0,-1,0) in local coordinates of astronaut . The ray upon intersecting any mesh in the said direction, ground/surface in our case, returns the point of intersection. This can be used to evaluate the height of astronaut above the ground. If the height is greater than the threshold *Y_THRESH* the astronaut is considered to be in air otherwise on ground. When on ground we impart an initial velocity to the astronaut. The initial velocity in the x and z world axes is maintained since there is no acceleration acting in those directions whereas the velocity in y direction keep getting diminished because of the gravity. The parameters *SPEED, SPEED_ANGLE, SPEED_DIR_ANGLE* denote the overall magnitude of initial velocity in m/s, direction of application of initial velocity w.r.t. the ground and direction of horizontal component of the velocity w.r.t. the z axis in radians. The *SPEED* in m/s is converted to $coordinates/s$ using the parameter *SCALE_FACTOR* as discussed earlier. The parameter *SPEED_DIR_ANGLE* is initialised to 0, however by pressing key 'A' or 'D' the user changes its value by +15 or -15 degrees, respectively, i.e. the orientation of the astronaut about the y axis is changed. The velocity is updated before each frame is rendered and therefore is computed inside *scene.registerBeforeRender()*. The computation is done with the help of *scene.getEngine().getDeltaTime()* which returns the time in milliseconds between two frames. Finally the updated velocity of the astronaut is imparted as *boy.moveWithCollisions(boy.speed)*.

In summary, it is important to determine the input from the user. After that the distance between the astronaut and the ground can be computed to modulate the speed accordingly.

5. Main Mesh and Animations

We can honestly say that this step has been the one that required our greatest effort. We had to convey a lot of patience and willingness to build and adapt 3D models to our needs. As we already told in the previous chapters we choose to use models taken from the following website: `https://nasa3d.arc.nasa.gov/models`, in `.obj` format, which do not provide any kind of nested hierarchical model.
We decided to adopt an open source software called *Blender* (in particular we adopted the version 2.83, `https://www.blender.org`), which permitted us to create and merge virtual skeleton to the astronauts meshes in order to make them animatable.

5.1 Main mesh

To realize what a *hierarchical model* is we can think about a undirected and acyclic graph, called *tree*, in which all nodes have a parent and some children except for the *root*, which has no parent and the *lefs*, which have no children. The root is in general the body part which has more connections with the others, the leafs are the body parts which have just one connection (with the parent). Thus in general we can think that the hierarchical model of a human has as root the top or bottom backbone, and as leafs the limb endings.

We applied the same hierarchical scheme to both the astronaut meshes.

5.1.1 Hierarchical Model

We idealized a simplified anthropomorphic hierarchical model. We decided to cut from an ideal human like model the body parts which were not required by the scene. For instance since the scene does not require the motion of the bones of the face (since are hidden by the helmet of the astronaut), we decided to simplify the entire head with one bone, the neck. The same we did for the hands and the foots. Our model is described in the picture 5.1.

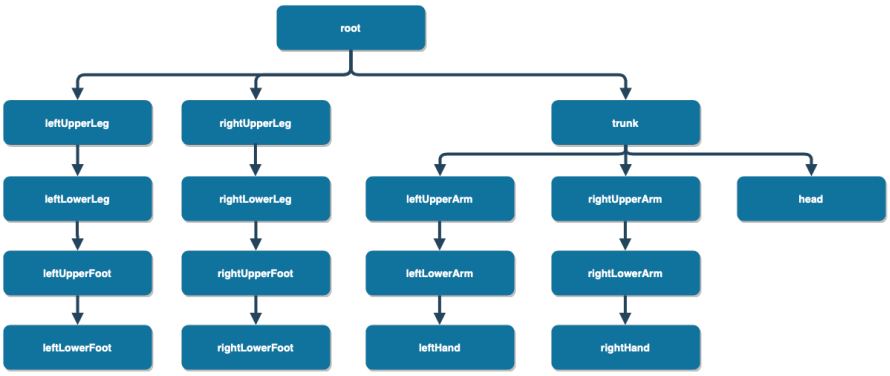


Figure 5.1: Picture which represents the hierarchical model of the astronauts meshes.

5.1.2 Rigify and Skinning

Rigify is a *Blender* open source library plugin (`https://docs.blender.org/manual/en/latest/addons/rigging/rigify`), which helps automate the creation of character skeleton (*rig*). It is based around a building-block approach, where you build complete skull out of bones parts. See figure 5.2.// It is important to note that *Rigify* only automates the creation of the skull and bones. It does not attach the skeleton to a mesh, so we still had to do *skinning* (weighting) by our self. The *skinning* is the process to associate portion of meshes to the respective bone, we used the word weighting since what is done in practice is compute the percentage of influence that each bone has on each vertex of the mesh, thus collect groups of weights, and thus groups of vertices, each of which represents a part of the body hierarchy. *Rigify* provides an automated solution to the skinning operation named '*Automatic weights*', which could be very helpful and save you a lot of time, especially if you are new to the topic and to the software. But it was not so for us, since the automated solution fails in presence of overlapping vertices, and our meshes contain a lot of them (because of the details and the construction of the meshes). Thus after some failures, we finally got the solution and we had to '*weight paint*' the meshes.
Weight painting is a method to generate large amounts of weight information in a very intuitive way. In which you literally have to color, based on an RGB weight scale, the influence each bone has with respect to the body parts, *Blender* will translate it into weights (you can find it here:

Figure 5.2: Rig and meshes of the astronauts.

`https://docs.blender.org/manual/en/latest/sculpt_paint/weight_paint`). Once this process were done, we could finally move the body parts of our meshes according to the skull hierarchy and play with the degrees of freedom of the skeleton.
After, we exported our models into *.babylon* format, which was the most compatible with *Babylon.js* framework (you can do it directly in *Blender*, thanks to the *Blender* plugin by *Babylon*: `https://doc.babylonjs.com/resources/blender`).

5.2 Animations

Keyframe is a term used in animation making (more generally in creation of movies, videos, cartoons, etc.) and is a special type of frame that defines the initial, final or intermediate state of an animation. Once the initial and final keyframes of an animation have been established, it is possible to create intermediate frames, this operation is called *interpolation* and in computer graphic it is the process made by the computer algorithm to form a smooth curve through the points of the keyframes.

5.2.1 Character

Now we start having an idea of the complexity of the process that animation involve. The processes we described above are a flow of simplifications that we have done in order to reduce a complex problem, such as move thousand of vertices in ordered way, to a more feasible one. Animations move body parts, each body part is linked to a huge set of vertices.
`Babylon.js` helped us in this hard process of keyframe definitions, and obviously in interpolation rendering. We could define main mesh animations thanks to `BABYLON.AnimationGroup()` function, which permitted us to group multiple animations, one for each bone, and to play them all together. Every time we created a new animation group, we started generating the `BABYLON.Animation()` slot for each bone and then fill it with the keyframes chosen. In particular, we had to specify for each animation the property we were concerned in animating (rotation, translation, etc.) and the respective framerate. Once the keyframes buffer of the animations were ready we could assign them to the respective meshes thanks to the `BABYLON.AnimationGroup()` instruction `*.addTargetedAnimation()`, which as said before, takes as parameter a `BABYLON.Animation()` type and a mesh reference. The reference to the meshes to animate are stored into a *JavaScript* dictionary, which is passed to the animation groups as input. In particular we pass two dictionaries, one called `actualBones`, which contains the reference to the bone, thus the current rotation/position, the other one called `bonesOffset` which contains the bones offset of the main mesh, in order to have reference values and to loop over it. But mainly to make the code reusable for both astronaut meshes, since they should perform same movements but with different skull offsets.
The search of the keyframes has been a particular tedious work, since we literally had to code and visualize the results by trial and error, multiple times per body parts. At the end we had got some nice results:

- `walkAnimation()`: represent the walking animation of the astronauts, obviously the arm, head and trunk movements are constrained by the stiffness of the suit. It is the one which required our best effort. We choose the canonical five keyframes per leg cycle (contact-down-pass-up-contact) to develop in total nine keyframes as shown in 5.3.
- `jumpAnimation()`: simulates the astronaut performing a jump. See figure 5.4.
- `standAnimation()`: represent the default position in which the astronauts doesn't perform any moves, it is needed to reset rotations for all the joints after other animations took place. See figure 5.5.

- `grabAnimation()`: the astronaut perform a grabbing with the arms, it is used in the lunar mission to handle the ovygen tank. See figure 5.6.
- `struggleAnimation()`: this animation is used in mission one to simulate the fellow astronaut struggling on the ground for the lack of oxygen. See figure 5.8.
- `repairAnimation()`: used in the martian mission, simulate the astronaut repairing the rover. See figure 5.7.

We decided to add animation blending to give a natural effect between the animation transitions thanks to the property `*.enableBlending`. By default *Babylon.js* uses linear interpolation function: `BABYLON.Vector3.Lerp(startValue,endValue,gradient)`.

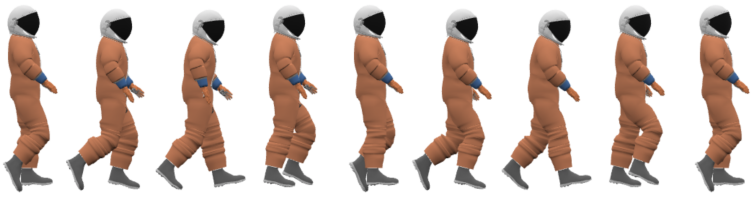


Figure 5.3: Walking cycle.

Figure 5.4: Jumping cycle.

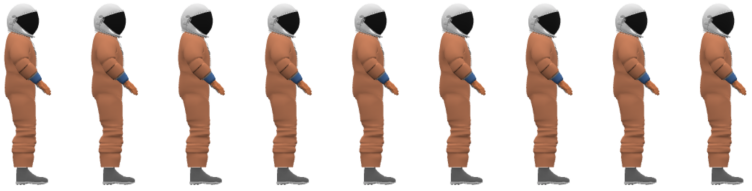


Figure 5.5: Standing cycle.

5.2.2 Camera

We decided to create a nice introduction to each game level using camera animations and sounds effects. As done before we started generating the `BABYLON.Animation()` slot for the camera and then fill it with the position keyframes chosen. We defined exactly three animations to deal with zoom effect:

- `slowZoomIn()`: this animation opens the scene, camera is far from the character and slowly moves along z axis.

Figure 5.6: Grabbing cycle.

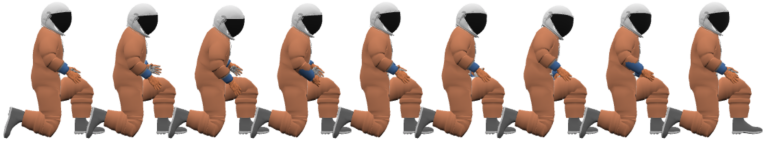


Figure 5.7: Repairing cycle.

Figure 5.8: Struggling cycle.

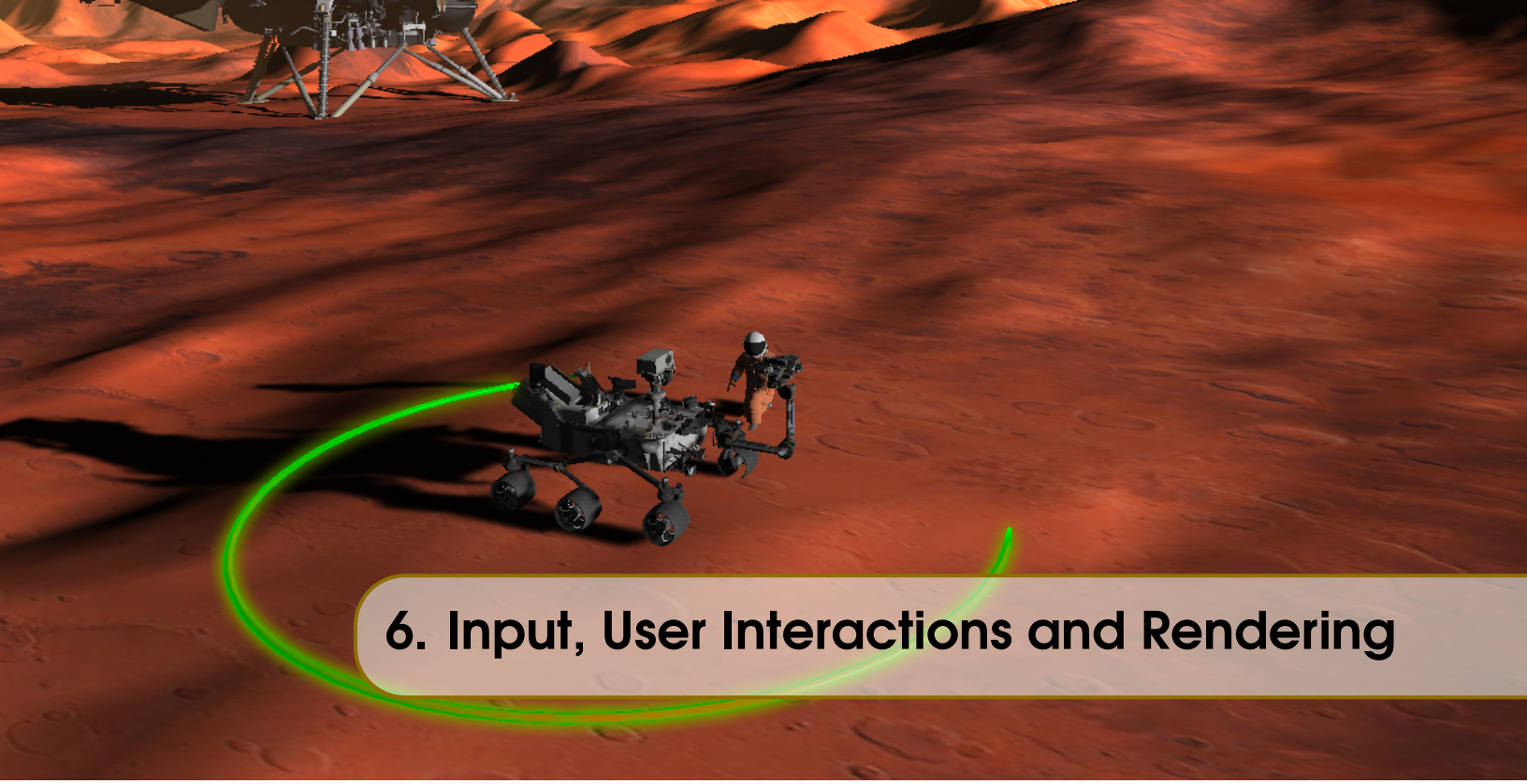
- `fastZoomIn()`: this animation follows the *START* button pressing and rapidly interpolate the position of the camera with the character position. We choose keyframes in order to obtain a fast zoom in camera which slows near the character.
- `slowZoomOut()`: this animation closes the scene, after the task is completed, the camera slowly goes back to the initial position.

5.2.3 GUI

To make the game more engaging we decided to add GUI animations too, in particular we used element alpha property to show and hide panels, texts and buttons. As well as before we made use of the class `BABYLON.Animation()`. In particular we defined:

- `fading()`: which, accordingly to the speed value, increase or decrease the alpha property of the element passed as parameter.
- `blinking()`: loops over a fast fading effect.
- `typeWriter()`: types one character at time of a string, giving a dynamic effect to the texts.

In the menu we animated the planets too, making them rotate at rendering; this time we did not use `BABYLON.Animation()` because it simply required increasing rotation along y axis.

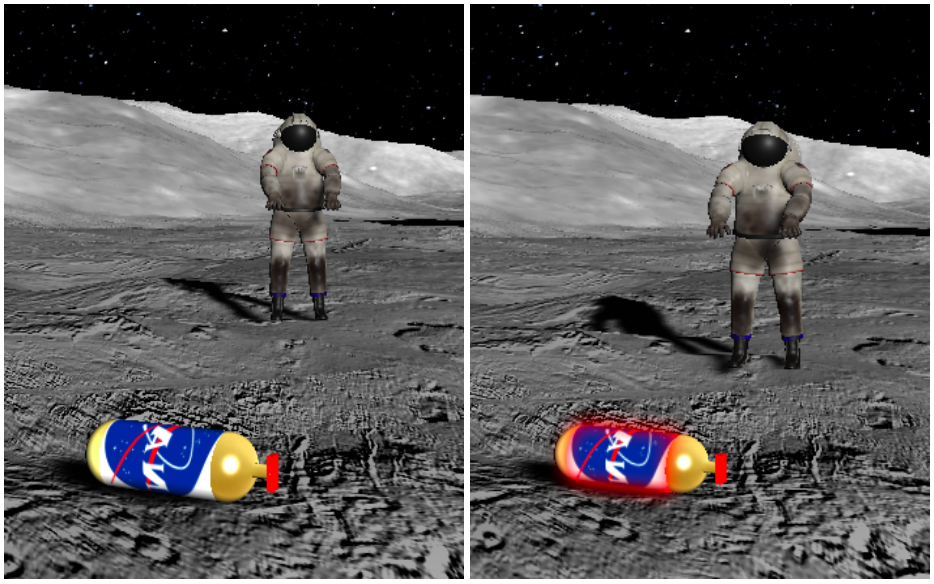


6. Input, User Interactions and Rendering

6.1 Input and User Interactions

The user/player interacts with astronaut and control its behaviour using the conventional keys used in most of the computer games. The movement of the astronaut is controlled using keys *W, A, S, D* which make the astronaut move forward, turn slight left, stop, and turn slight right, respectively. Also the astronaut interacts with the environment like grabbing and throwing oxygen cylinder in Mission Moon which are accomplished using keys *Q and E*, respectively. Also repairing rover in Mission Mars is controlled by the user by pressing key *E*.

In order to process the input from user it is first captured in the Javascript file. We make use of eventListener and keyPress that trigger an event whenever specific keys are pressed. In particular we set the parameter *flagImp* on pressing key *W* whereas the parameter is assigned 0 on pressing *S*. The parameter is essential to determine if the velocity/impulse should be imparted to the astronaut or not. *flagImp = 0* brings it in the idle mode otherwise the astronaut remains in the walking state. This, therefore, also control corresponding animations namely, *WalkAnimation* and *StandAnimation*. At the same time the user can turn the astronaut to slight left or right by pressing *A* or *D*, respectively. Pressing *A* or *D* updates the parameter *SPEED_DIR_ANGLE*. In Mission Moon the grabbing behaviour, achieved by pressing key *Q*, is activated only when the distance between the astronaut and the oxygen tank is less than a predefined distance. In case the distance is key *Q* remains deactivated. In order to make the player aware of the above rule we highlight the oxygen tank with red color as soon as the key *Q* becomes active i.e. the tank can be grabbed by the astronaut. On pressing *Q* the astronaut performs the *GrabAnimation* and the highlight color of the tank is changed to green. Similarly, we have created a hoop of radius *R_1* around the struggling astronaut which glows red signifying that the astronaut is short of oxygen supply. The hoop color turns to green as soon as the helping astronaut enters inside the hoop and the struggling astronaut gets the oxygen tank. The similar visual aesthetics are applied to Mission Mars. There's a hoop of radius *R_2* around the rover which needs to be repaired. The hoop has a highlight color red until it is repaired. As soon as the distance between the astronaut and the rover becomes less than *R_2* the user can press the



(a) (b)

(c) (d)

Figure 6.1: (a) The grab option for the user remains deactivated (b) As the distance between astronaut and the oxygen tank is less than R_1, the tank can be grabbed and thus glows to indicate that (c) Oxygen tank grabbed by the astronaut as the user presses the key Q (d) The astronaut throws away the oxygen tank as the user presses the key E

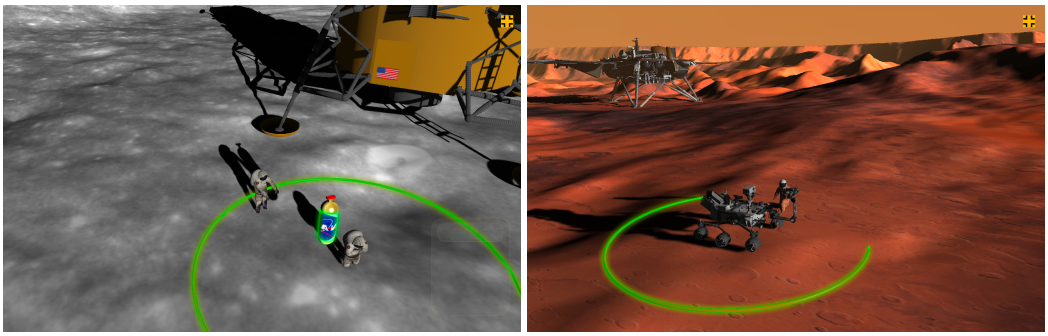
key E to start repairing the rover. This will stop other activated animations at that moment and the astronaut starts to perform the *RepairAnimation*. Also the hoop highlight color changes to green to indicate the successful completion of the task.

6.2 Rendering

To create a complete 3D scene consisting of a camera, lights, 3D shapes (meshes), corresponding physics, user input we call the function *createScene()*. After that we use *Babylon.js* in-built function *runRenderLoop()* which is used to register and execute a render loop. It is inside this function that we evoke the method *render()* so as to run the scene.

7. Conclusions

The presented work is an attempt of the members of the team *SpaceY* to visualize, design and implement a multi-mission interactive game set in the space. The process was quite challenging, however the zeal to learn and implement from the scratch kept the presenters going.



(a) (b)

Figure 7.1: Successful completion of tasks as (a) Oxygen tank is delivered to the struggling astronaut in the *Mission Moon*, and (b) Rover is repaired in the *Mission Mars*

The task required understanding and exploiting a number of concepts pertaining to the field of Interactive Graphics. Some of the key concepts learned during the project were modelling, animation, physics setting, user interaction etc. A basic knowledge of *JavaScript* and *HTML* languages was required for the project. We extensively used *Babylon.js* library to accomplish the task. Though in the beginning it was quite challenging to get along with as comprehensive library as *Babylon.js*, however its well documented concepts, *Playground* demonstrations and active forum expedited the process of learning. Along with that we used other resources like *NASA* website to import 3d meshes, *Blender* to riggify the meshes.

Last but not the least we would like to express our gratitude towards **Prof. Marco Schaerf** for

introducing to us many interesting concepts during the course of *Interactive Graphics*. The lectures from the course were quite constructive for us to understand base concepts and eventually to write our customized solutions.