Meteorite Apocalypse

Nicolò Brandizzi

Student id: 1643869

Eric Stefan Miele

Student id: 1643696

Interactive Graphics
Project report

DIAG
Sapienza
June 2018

SAPIENZA
UNIVERSITÀ DI ROMA

Abstract

We implemented a game based on the *Babylon.js* framework, using physical engines, meshes, textures, lights, particle systems and much more to create a playable environment.
Our focus was on the cyclic implementation of the character animations manually coded, but also on the challenging aspect of the game.
In the following report we analyze in detail every technical aspect and provide a user manual with instructions for playing the game.

Contents

Chapter 1

Technical part

1.1 Libraries and models

Besides *Babylon.js*, *Cannon.js* was used as physics engine in order to implement collision detection and gravity. *Babylon.js* already uses this library internally as a plugin. We imported the model of the character which comes with a set of meshes and links to which we manually applied rotations and translation in order to obtain animations.

1.2 Babylon.js

Babylon.js is a real-time 3D engine in the form of a JavaScript library for displaying 3D scenes in a web browser via HTML5. It was initially developed by Microsoft employees, on their free time and the project counts in 2016 more than 90 contributors. This 3D engine is naively interpreted by a web browser supporting the HTML5 standard. The programming language used is JavaScript, allowing calculations and 3D rendering via the WebGL programming interface.
The source program *Babylon.js* is itself encoded in TypeScript, but the version available in JavaScript is provided to the end user, so that the latter can directly use Javascript to access the *Babylon.js* API.
Babylon.js offers an online programming space called the PlayGround, which is very useful to integrate code with examples.

1.3 Details and technical aspects

1.3.1 Skybox

Babylon.js provides a custom type of mesh called **BoxMesh**, which we used to wrap the entire scene with. This box is then linked with a material which is composed of six jpegs, each corresponding to a part of the sky. We tuned the various parameters such as *diffuseColor*, *specularColor* and, most importantly, *infiniteDistance*, in order to have a realistic looking sky around the map.
Moreover we disabled the *checkCollision* and *Lighting* attributes for the following reasons:

- The only objects able to collide with the skybox, which is positioned 1000 units away from the map, are the fireballs. We chose to disable collision in order to

induce a feeling of an *endless* sky. The more distance is accumulated between the player and the fireball the less the latter is visible, until it exits the skybox and disappears from sight.

- Finally we disabled the *Lighting* to mimic the real world, so that the sky is not influenced by other light sources inside the map.

1.3.2 Ground and height map

The map you can stand on while playing the game is a combination of multiple objects which we will now illustrate in detail.

Ground Box This box shaped mesh is used as the actual zero level for the map. It has an attribute called *Physics Impostor* which is used to simulate the actual physics of a box, thus making collision with the player and other meshes possible.

Lava This object is a Babylon's custom material which emulates the behavior of the real lava. To use it we first had to assign it to the scene and then to the *Ground Box* as a *material* attribute.

Height Map The height map is also a mesh which requires a gray-scaled image. Each pixel of this image has a white intensity value which is then mapped to the mesh corresponding to the height of that specific point. We generated a custom gray-scaled image to use for as a base of the height map building.
Furthermore we tuned the following attributes to make the map playable:

1. **Ground size** We used the same values for both the x and z directions of the ground, making the map a square. We thought that this would be an optimal choice since the game brings the player to look up the majority of time, and we wanted an easy geometry for the map to remember.
2. **Subdivisions** This attributes indicates the number of subdivisions the maps goes through, the grater the number the smoother the map.
3. **Max Y** The maximum height the height map can assume. Increasing or decreasing this parameter makes the map taller or shorter. This makes the difficulty harder or easier since the ground has no friction and brings the player to slide more towards the lava.

As we did for the *Ground Box* we applied an ice texture to the height map too. Moreover we exploited another *Physics Impostor* with zero friction in order to get a nice sliding effect on ice, which makes harder the survival of the player.

1.3.3 Light

This game implements two different kinds of lights which serve different purposes.

Directional Light

This light is emitted from everywhere in the specified direction, and has an infinite range. An example of a directional light is when a distant planet is lit by the apparently parallel lines of light from its sun. This is why we used it as our moon light source and choose a suitable color to arouse a creepy sensation.

Hemispheric Light

A hemispheric light is an easy way to simulate an ambient environment light. It is defined by a direction, usually 'up' towards the sky. We used this kind of light to let the player see the borders of the map in order to avoid falling into the lava.

1.3.4 GUI

The Graphic User Interface is an essential part to display useful information to the end user such as:

- **Health Bar** This graphical element is tied to an event listener which decreases the player's health by a factor of 30 when any incoming meteorite hits the character. Furthermore it looses 1hp per second each time he collides with either magma or a meteorite on the ground.
- **Mana Bar** Same as before, the mana bar decreases when a button down event is detected.
- **Score** The score interface is composed by a gray background and a dynamic text which shows the time elapsed from the start of the game until the player's death.
- **Game Over Screen** When the player's health reaches 0, a *Game Over* screen appears and shows both the score and the instruction to start the game again.
- **Dynamic Infos** These kind of dynamic texts are documented in the *User Manual* section 2.

1.3.5 Camera

The camera is a fundamental object in every graphical environment.
We make use of two cameras, one for First Person [FP] point of view [POV], while the other is used for Third Person [TP] POV.

First Person Camera

The camera used for FP is the *Universal Camera* object which is also the parent of the body mesh that sticks to it and permits the movement of both with a single command.
Since this camera does not offer any collision detection, we wrapped it into an ellipsoid which will be our collision mesh detector. Moreover we set an offset on the Y axis of the ellipsoid to avoid having the legs of the body mesh entering the ground.
Finally we changed the key listeners which move the camera in order to include the *WASD* movement commands.

Third Person Camera

For this kind of POV we explited the *Arc Rotate Camera* object, assigning as its parent the FP camera itself for the reasons we discussed previously.
We applied some tuning to the parameters of the camera (especially the parameters $alpha = \pi/2$, $beta = 1$ and $radius = 10$) in order to let the game start with the camera looking the body mesh from behind.
Lastly this type of camera is used also for the 2D POV, changing the $radius$ value to 200 units.

POV changing

The POV can be changed with the *V* key. Although it may seems an easy task it is very tricky to achieve, since it is necessary to deactivate the TP camera and perform some spacial transformations to the body mesh.

Disposing TP camera In order to change from FP to TP we need to remove the *Arc Rotate Camera* object from the scene camera's array, with a simple for loop. To restore the TP POV we just compute the inverse process.

Body Mesh RotoTranslation When switching to FP POV the camera position is set at the body mesh's chest height, thus obscuring part of the field of view. To fix this we displaced the mesh backward and rotated the arms in order to position them correctly for FP.
These transformations required the body in its rest configuration. You can achieve this either by inverting the transformation or using the mesh *Resting Position* attribute and setting the current homogeneous matrix to the resting one. Finally you can change from a position to another by switching back to the resting position and then applying your custom transformations.

1.3.6 Character

The character was created importing an existing model with extension *.babylon*. This last came with a set of *meshes* and *bones* which were rotated and translated in order to obtain animations. A *bone* is a *Babylon.js* object type used expressly to represent links and bones.
The character is a hierarchical model composed by more than 50 elements, including the eyes and the fingers, but since we just needed few links in order to implement animations we just considered the following elements:

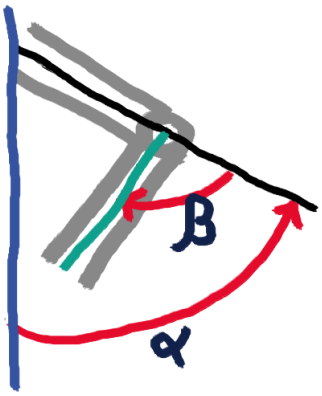
- Body
- Head
- Upper, lower arms
- Upper, lower legs

The main mesh is the body which has as children the head, the upper arms and legs. The lower arms and legs are of course children of their upper components. In this way, during the animation when we translate the body (parent), all of its children are translated as well. But if we rotate or translate in space the children (e.g the upper right arm), then the body will not be affected by this operation. This guarantees a great modularity of the model.
The body has as parent the main camera so that, when the camera is moved through the user's interaction, the character follows the camera.
A texture was applied to the character in order to resemble a golem-like humanoid made of magma.

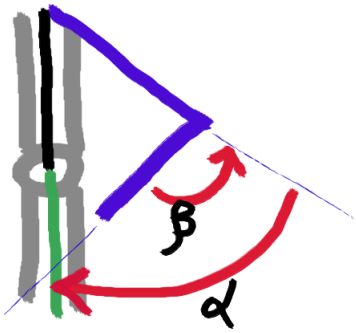
1.3.7 Animations

The following animations were implemented and will be discussed in the next paragraphs.

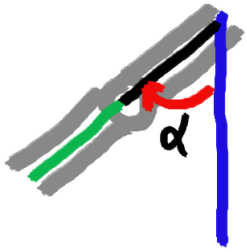
Running The running animation was implemented through incremental rotations around the Z axis of the upper and lower legs and arms components.
The legs animation is cyclic, subdivided in 4 phases.
In the first phase the upper leg is rotated incrementally until it reaches an angle of $\alpha = \pi/4$, while the lower leg is incrementally rotated until it reaches an angle of $\beta = -\pi/4$, as show in Fig. 1.1a.
In the second phase both legs are rotate incrementally until they reach an angle of 0 (rest configuration) (Fig. 1.1b).
Then in the third phase only the upper legs rotate incrementally until they reach an angle of $\alpha = -\pi/4$ (Fig. 1.1c) and finally in the last phase the upper leg brought back to the rest configuration (Fig. 1.1d).
The same pattern is applied to the other leg in a symmetrical way.



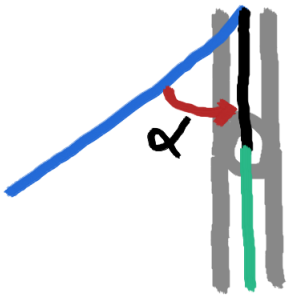
(a) Leg animation phase 1



(b) Leg animation phase 2



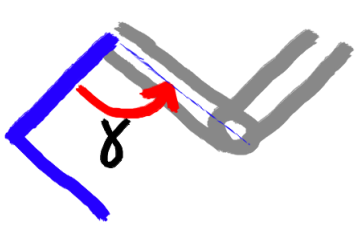
(c) Leg animation phase 3



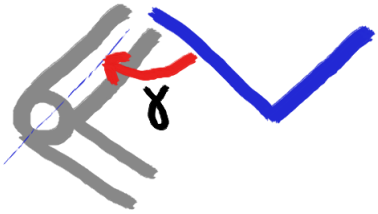
(d) Leg animation phase 4

Figure 1.1: Cycle phases of leg movement

The arms animation is cyclic as well but has only 2 phases. In the first one the upper arm is rotated incrementally from an angle of $\gamma = -\pi/4$ to an angle of $\gamma = \pi/4$ (Fig. 1.2a) while in the second (Fig. 1.2b) the inverse rotation is computed incrementally. The rest configuration is reached when the upper arms are rotate by an angle of 0, so the animation cycle starts at the first phase's half.
In order to resemble the arm position during the running action, the lower arms are always rotated by an angle of $\pi/2$ as shown in the figures.



(a) Leg animation phase 1



(b) Leg animation phase 2

Fireball charge When the fireball animation in triggered two separate, but coordinated, animations start: one for the fireball and one for the arms.
The fireball starts with a radius of 0.1 units and then increases until it reaches a maximum of 1.5 units. During this radius increment the ball rotates around the 3 axis with an increment of 0.03 units so that it resembles a floating animation.
The upper arms instead are rotated both around the X and Z axis of their frames in order to simulate an opening animation. These rotations are incremental from the rest configuration to a maximum angle of $\pi/4$.

Jump The charge animation of the legs is implemented in a similar way as the first 2 phases of the run animation. The only difference is that it is applied to both legs in the same way and not symmetrically and that the body is translated along the Y axis in order to decrease slightly its height.
When the jump button is pressed this animation is triggered, creating a bending effect of the knees, and when the button is released the legs are stretched bringing the character to the rest configuration.
Afterwards the body is translated incrementally along the Y axis of its frame until it reaches the maximum jump height of 10 units. During this translation the legs are animated so that they become open wide.
Then the gravity acts on the body decreasing its height and simultaneously a reverse animation is applied to the legs to bring them to the rest configuration again.

Landing When the character falls and collides with the ground, the landing animation is triggered. This last is like the charging animation of the legs during the jump action, but faster.

1.3.8 Meteorites

In order to make the game more challenging meteorites rain from the sky. At the beginning 10 meteorites are created in random positions above the ground at height 1000 units. Each meteorite has a particle system attached to it to simulate a trail of fire and has a unique ID number to handle collisions. This was done because the physical engine, during a collision with the character, fires multiple collision events and this would damaged the player multiple times. When a meteorite collides with the height map a explosion is simulated through a red particle system. When it collides with the lava it generates an white explosion instead. Some seconds after the

explosion the meteorites disappear and are created again. This also happens when a meteorite gets below -500 units along the the Y axis.

Chapter 2

User manual

This game is about surviving a meteorite apocalypse as long as you can.
You need to avoid being hit while watching out for the lava underneath you. Each direct hit by a meteorite will cause you 30 damage, while making contact with either a fallen meteorite or the lava will cost you 1hp per second. Both mana and health have a regeneration factor of 1 unit per second.
As you go forward the number and dimension of the meteorites increases until you cannot avoid them anymore...
You can die either by loosing all your health points or by falling off the map. When you die the game will display a *Game Over* screen where you can check the score you reached.

2.1 Character Movement

You can move with the *WASD* key configuration:

- $W \rightarrow Forward$
- $A \rightarrow Left$
- $S \rightarrow Backward$
- $D \rightarrow Right$

Moreover you can jump by pressing the space bar key and moving the camera with your mouse.

2.2 Shooting Fireballs

You can shoot fireballs by clicking the mouse button (either the left or the right one). Fireballs are used to deflect meteorites.
The more you keep the button pressed the bigger your fireball will be, this means you can deflect bigger meteorites more easily, but be aware, casting fireballs cost you mana, when your mana limit is reached you will not be able to cast fireballs until it regenerates.

2.3 Other Possible Interaction

Here you can find how to toggle between different modes in the game.

Toggle Snow The game start with the snow mode activated, but if you notice some lag you can deactivate it by pressing the Z key.

POV switching You can toggle between three POV modes pressing V:

- **Third Person** The camera is located above the character shoulders.
- **First Person** The camera is located in the head of the character.
- **2D** The camera is located far above the character's head, giving the impression of playing a 2D game.

Play/Pause Music This games comes with some background music in it, if you find it annoying you can pause/play it with the M key.