SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

The Boring Game

INTERACTIVE GRAPHICS

Professor:
Marco Schaerf

Students:
Daniele Affinita
1885790
Domenico Meconi
1838058

Academic Year 2022/2023

Contents

1 Introduction

The Boring Game is a browser 3D adventure game where the player has to complete puzzle-like tasks. In the first stage, the protagonist, a robot, has to navigate through a maze and find a way to get out. In the second stage, it has to arrange colored cubes in the correct order under color-altered light conditions.

1.1 Tools and Libraries

1.1.1 Three.js [1]

The game is created using JavaScript and the `Three.js` library, a framework designed for displaying 3D content directly within web browsers. This library builds upon `WebGL` and offers various high-level APIs. In our development process, we mainly utilized:

- **Camera**, which enabled us to manipulate the camera's position and orientation, directing its focus as needed.
- **WebGLRenderer**, which represent the direct link between `Three.js` and `WebGL`, using the latter to render the scene.
- **Geometry & Mesh** were useful to define simple geometric 3D shapes and assigning them a material.
- **Loaders** to load textures and models into the framework.
- **Light & Shadow** provide an interface for managing light position, orientation, and the generation of shadows. They were especially useful since our game heavily relies on lighting effects.
- **Debugger** helped us to develop the game providing a visualization of abstract scene elements to make the debug simpler.
- **Vectors & Math** to perform 3D vector computations and basic utilities.

We also included two add-ons to the core library, namely `FontLoader` for loading custom fonts, `GLTFLoader` for loading GLTF 3D models, and `OrbitControls` for debug purposes.

1.1.2 TweenJS [2]

This library provides a unified interface for animating 3D objects. It facilitates the selection of character joints and the application of a roto-translation transformation to the link, leading to point-to-point elementary animations. The trajectory is calculated using an interpolating function class and an execution time. Specifically, we utilized linear and quadratic functions. It's also possible to stack multiple basic animations to create a complex one.

1.1.3 Cannon [4]

Cannon is an open-source 3D physics engine for Javascript. It provides a set of tools and algorithms for simulating realistic physics in web-based 3D applications. It can handle various aspects of physics, including rigid body dynamics, collision detection, and constraints. In particular, we used it for two main purposes: to detect collision between the main character and various objects in the world and to add gravity.

1.1.4 Stats [5]

Stats is a lightweight javascript library that offers a code performance monitor. In particular, we utilized it to profile our code by visualizing the Frames Per Second (FPS) changes over time. This tool was valuable in spotting bottlenecks within the game loop, which led to reduced performance. By finding these issues we could take action to improve the computational load.

1.2 Game Controls

The controls are exclusively provided through the keyboard. A detailed list of all available commands follows although key button hints will also be provided within the game.

Movement:

- ↑ or **W**: Move forward
- ↓ or **S**: Move backward
- ← or **A**: Rotate Counterclockwise
- → or **D**: Rotate Clockwise

Actions:

- **L**: Toggle torchlight (only level 1).
- **P**: Action (Pick up / Drop item, select cube, press button).

1.3 Browser Compatibility

To assess the performance of our game in the most common browsers, we played it using the fps as an evaluation score. The table **??** summarizes the results.

Table 1: Browser Usage Statistics

Browser Name	Performance
Firefox	★★★
Chrome	★★
Edge	★★
Safari	★

2 Environment

We have adopted a hierarchical approach to structure the development of the framework. The low-level functionalities serve as interfaces to the upper levels, contributing to the overall robustness of the codebase. In this section, we will discuss the core components of the code, focusing on their implementation and management.

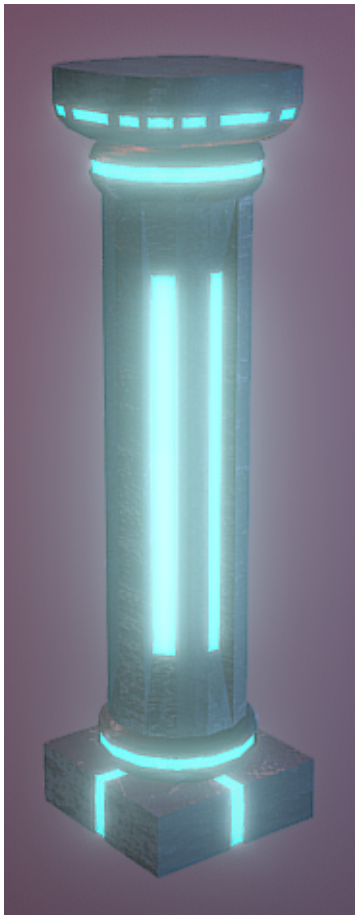
2.1 Models

We employed many 3D models in our game, all of them in GLB format to avoid importing multiple external loaders. All the models have been taken from the website Sketchfab [6]. We tried to find a tradeoff between the model's visual quality and its polygon count. The models used are illustrated within the following images.

2.2 Efficient Loading

Some of the models we used are not lightweight, so preloading was necessary. To do so, we developed an efficient loading mechanism that loads the required models before the game begins. information about the models is compiled into a configuration file. In particular, we store the JavaScript class corresponding to each model (details in section 2.3), the path of the GLTF file, the number of instances of that object to be rendered, and the levels in which the model is used. All of this information is organized into a hashmap, where the class name represents the key and all the corresponding information constitutes the value. During the loading screen, a custom loader utilizes this data structure to load only the models that are required. Specifically, it creates the instances by using the `GLTFLoader` asynchronously, precompiling them, and storing the results into another hashmap. It has the model class name as key and a queue of loaded models as value, in particular, the queue maintains as many entries as specified in the configuration file. This approach allowed us to significantly reduce the slowdowns during the game, creating a loading phase right before each level starts.

(a) HMO Man, Lowpoly main character



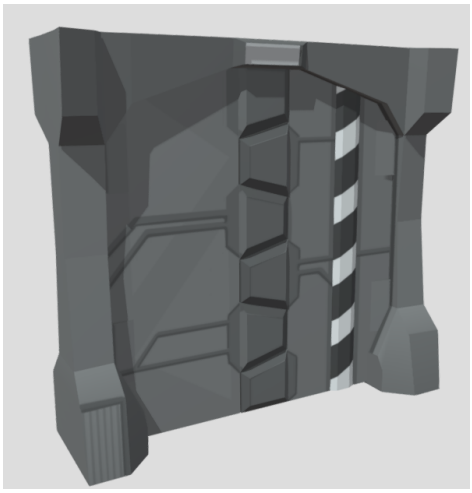
(b) Pillar sci-fi

2.3 Objects and Textures

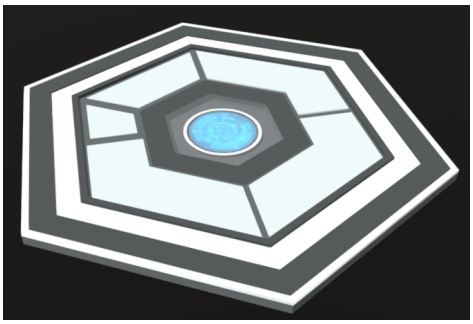
We widely employed the **Factory** design pattern to organize all the game elements. Every factory represents a class of game elements and it represents a unified interface for creating instances. In particular, every element of the game has its own class but the constructor is accessed only from the factory. Two fundamental factories are the ones representing the materials imported as textures and objects. The latter utilizes the loader illustrated in 2.2 to create the elements. The Texture factory creates the Threejs materials which can be attached to geometries. Every `MeshStandardMaterial` is made up of multiple texture components which gives a more realistic appearance to the objects. The texture components used are:

- *Map* for the basic color map, *AoMap* to represent ambient occlusion.
- *bumpMap* for modifying the object's normal vectors providing a depth effect.
- *emissiveMap* for defining the emissive color and intensity.
- *metalnessMap* which alters the blue component to give a metalness effect to the object.

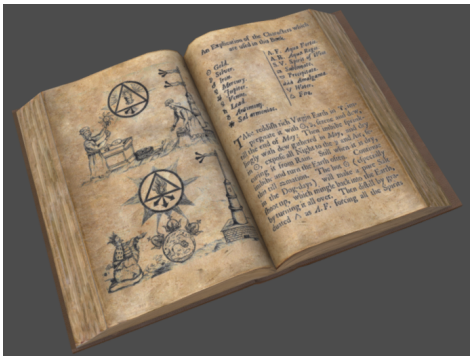
(a) Generator Sci-fi



(b) Door



(c) Platform sci-fi



(d) Open Book

- *normalMap* for further modifying the normals for each channel.
- *roughnessMap* altering the blue channel to create a rough effect.

We created two different materials: one for the floor and the other for the walls. We also defined a field "density" for every texture instance that defines how often the texture should be repeated within the associated surface.

The object's classes use the models loaded by the custom loader and attach several properties to them, such as shadow casting and a physics object. It also defines the object dimensions by applying to it a proper scale.

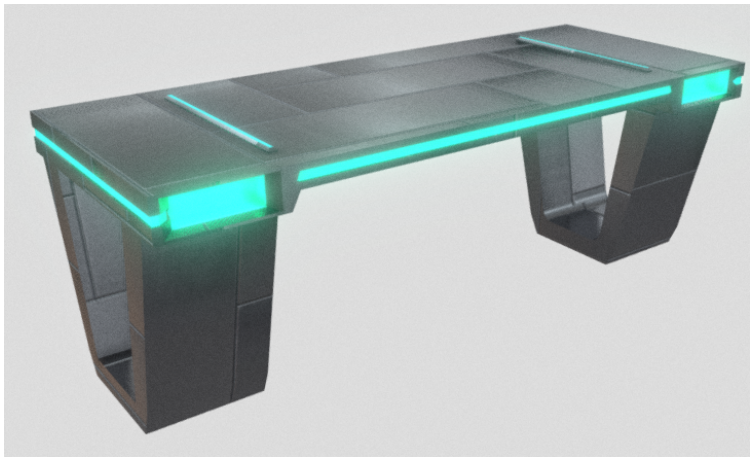
Another important Factory we have defined is the so-called `BuildingFactory`. It creates all the geometric 3D elements that are used to create the room's structure. The classes of such elements are hierarchically defined and represent the floor, the wall, and the wall with door. Each of them has an associated material as well as a physical body.

2.4 Build Engine

In order to quickly create a world structure, we designed a building engine that allows you to define the level environment simply by specifying its structure into a JSON

(a) Button

(b) World Cup Trophy



(c) Desk Sci-fi

file. The engine is basically a **JSON parser** that calls the right factory methods to create elements defined in the structure file and place them in the correct position. Every JSON entry representing an object contains the pose (position + orientation) of the element, the element type, an associated ID, the attached texture, and other parameters object dependent. In order to avoid redundant texture loadings, the parser uses a Lookup Table to reuse textures already loaded that match both the type and the "density" field. This room parser has been useful for effortlessly creating many worlds that change according to the selected difficulty.

2.5 Main Game Loop

Main Game Loop is responsible for managing the high-level aspects of the game. During its initialization, its constructor instantiates essential components, including the scene, camera, and physics engine. Furthermore, it maintains information about the current game level and shares this data with the **Loader** and **Build Engine**, which

utilize it to render the appropriate game elements. After all elements have been loaded, the loading screen is disabled, and the actual game loop begins. It invokes the update function for each sub-element, continuously checking if the screen has been resized to dynamically adjust the renderer. The frequency of the game loop is determined by the `requestAnimationFrame()` function, which is dependent on the user's browser.

2.6 Lights

We employ a wide range of lighting techniques in our game, each serving a specific purpose. In the initial level, the player starts in darkness with only a flashlight for illumination. To achieve this effect, we utilize an **AmbientLight** source with a very low-intensity level of 0.005, providing just a hint of visibility to the game world. Additionally, we employ a **SpotLight** source (as shown in Figure 4) that follows the character's movements and points toward the player's orientation, simulating the directional beam of the flashlight. Once the player has completed all the tasks of the first level the Ambient Light intensity rises to 0.8 improving the visibility of the environment. In the second level, the goal was to have a colored light that would alter the color of the cubes in the room. To do so we used the **DirectionalLight**, given the length of the room we inserted three of them. They are placed in different positions pointing toward the room center as shown in Figure 5 .The color of the lights is initially set to red but can be changed by pressing the buttons in the room.

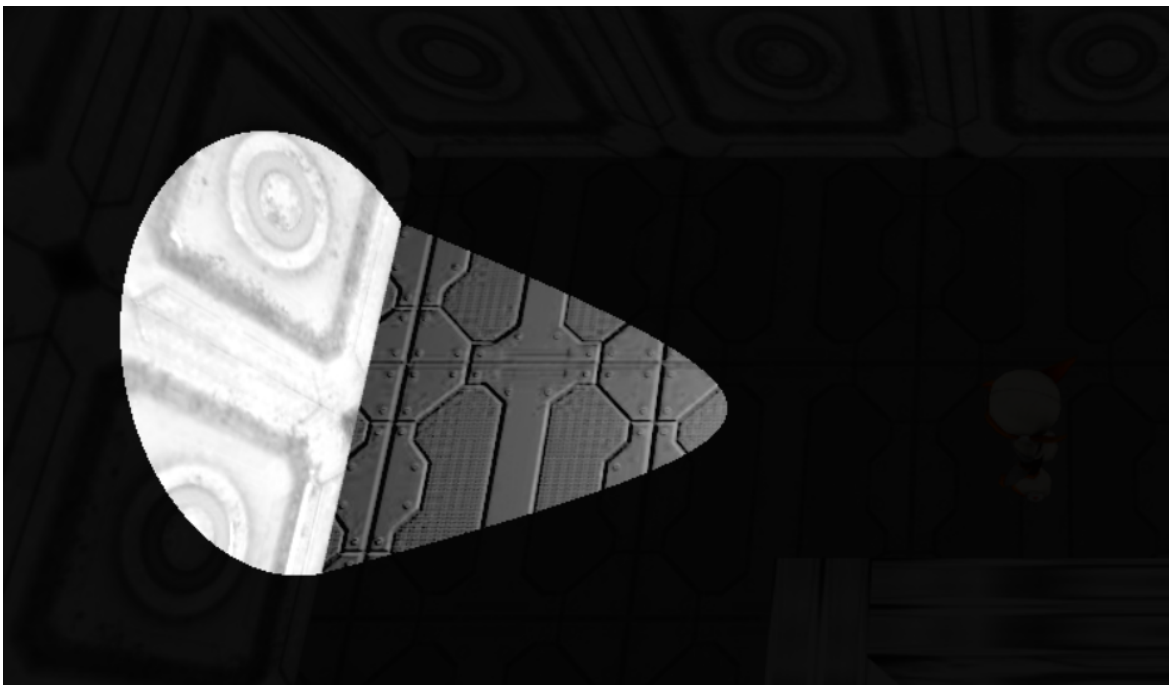


Figure 4: Flashlight

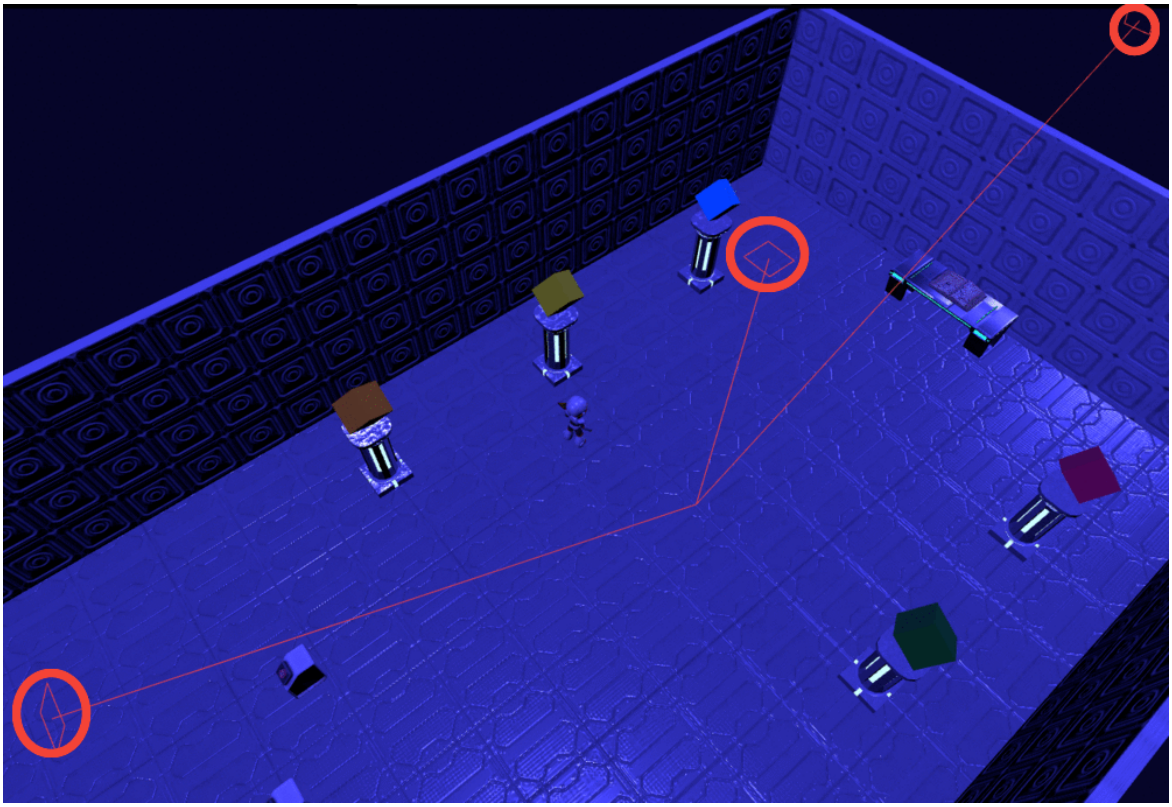


Figure 5: Directional Light placement in Level 2

2.7 Camera

We utilized a **Perspective Camera** to generate the *Projection Matrix* during each iteration of the Game Loop. This camera object is provided by `Three.js` and implements a Perspective Projection simulating the way the human eyes see. The Perspective Camera interface offers customization options to control the view of the scene, with the primary attributes being the Field of View, Aspect Ratio, near plane, and far plane. In our game, we chose the far plane distance to fit the whole room while we set a non-zero value for the near plane to prevent the rendering of walls or objects that might obstruct the view of the main character.

During gameplay, the camera is consistently positioned in close proximity to the main character, maintaining a fixed relative position with respect to the player, which is represented as $C = (0, 150, 50)$. This setup ensures that the camera effectively follows the player's movements, always maintaining the same distance.

The camera's position is only altered during a close-up animation of a door opening and approaching the book. In this case, the camera changes translation and rotation to focus on the object. Additionally, when the player successfully completes the second level, the camera is adjusted to point towards the trophy.

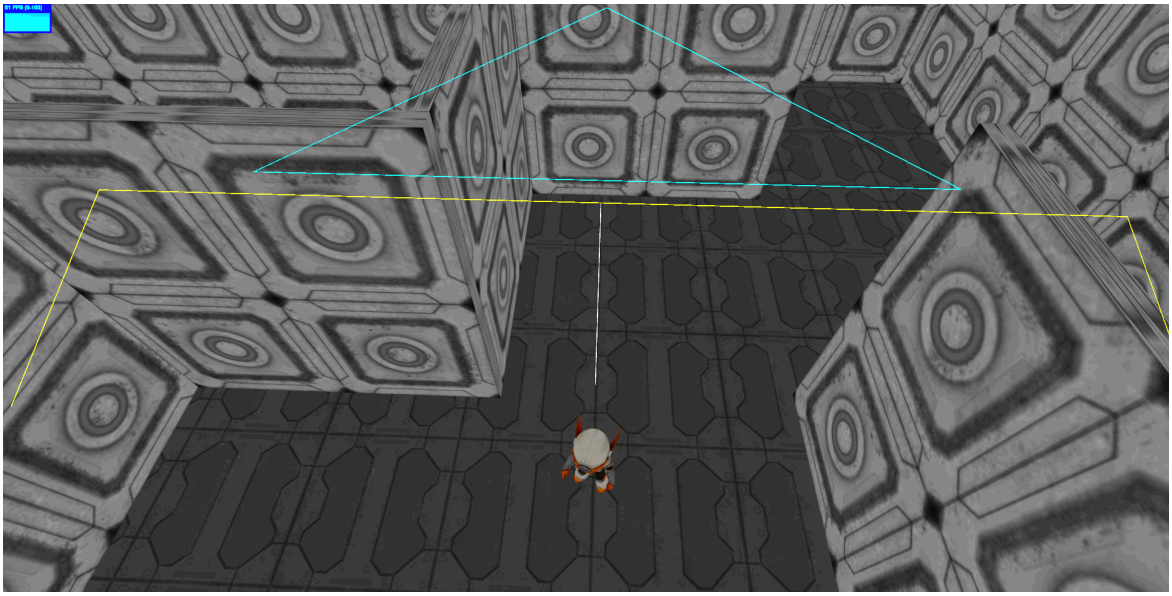


Figure 6: Camera Position

2.8 Music

To make the gameplay more engaging, we also added music. The original soundtrack has been sourced from [3] and modified using the software Audacity. To include it, we used `audioLoader()` setting the reproduction in a loop. Since the audio in `Three.js` is directional, we ensured that it's perceived always in the same way, by attaching the audio loader to the camera, which remains at a fixed distance relative to the player. Some browsers prevent from executing an audio without a user interaction, to overcome this issue we start the music only when the first user input is caught.

2.9 Physics

To make the game more realistic, we have included physics. **CannonJS** defines any object with a positive mass as a rigid body. In addition, we also need a bounding box associated with the element to define collision regions. By binding these two elements to every instance of our game we were able to handle collisions preventing the main character from passing through rigid bodies. To assign a bounding box to elements, we took advantage of the **Box3** function from ThreeJS which provides the object's height, width, and depth.

Another useful functionality we used from CannonJS is gravity. In particular, it is used for the second part of the transition between the two levels where the characters drop from the top of the room. In this case, CannonJS controls the acceleration and velocity evolution according to the underlying physics model.

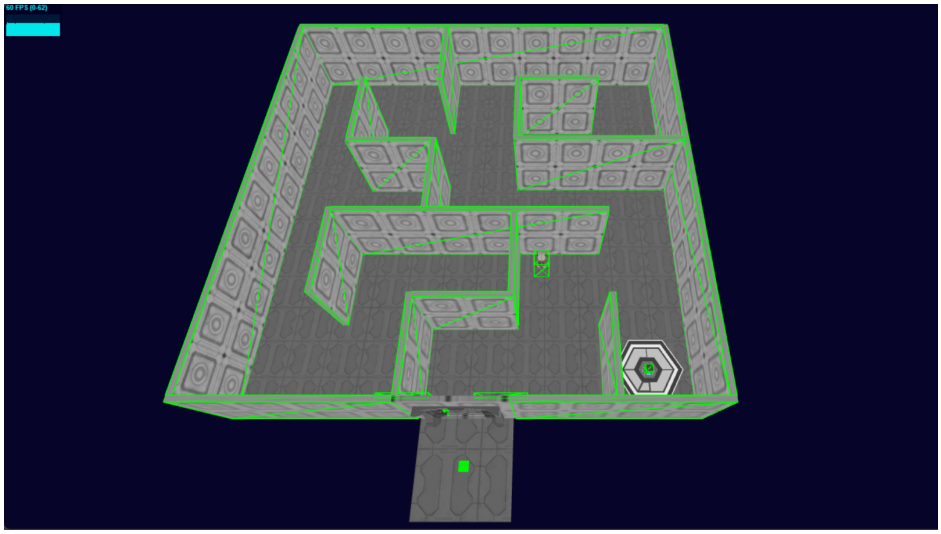


Figure 7: Level 1 Physics view

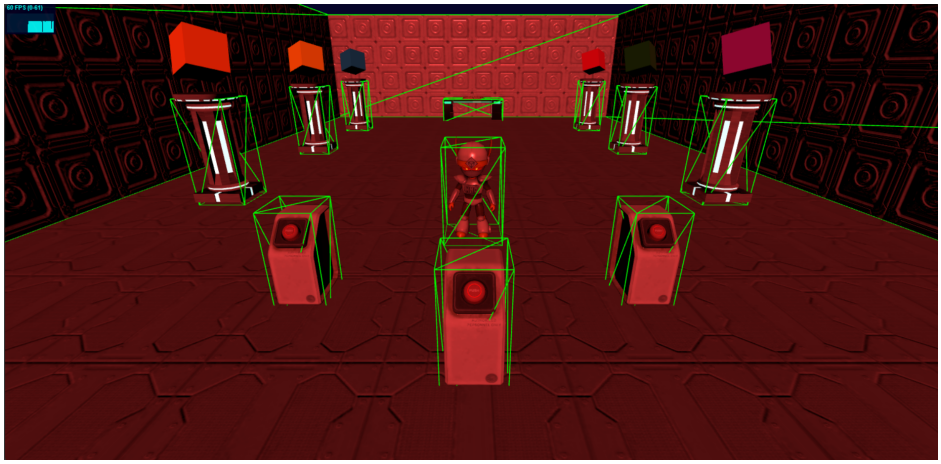


Figure 8: Level 2 Physics view

Figure 9: End Level 2 Physics view

3 Player

3.1 Structure

The model of the main character shown in Figure 11 is built with a human-like hierarchical structure. The skeleton represents the underlying structure of the character's body, defining the placement and orientation of bones in a hierarchical manner. The model consists of 72 joints, which allow for articulation and movement, such as rotating or bending limbs. However, only the main ones were used for the animations, such as: head; chest; shoulders; upper and lower arms; upper and lower legs; feet. A diagram of the simplified hierarchy tree, including only the joints we used, is shown below in Figure 10

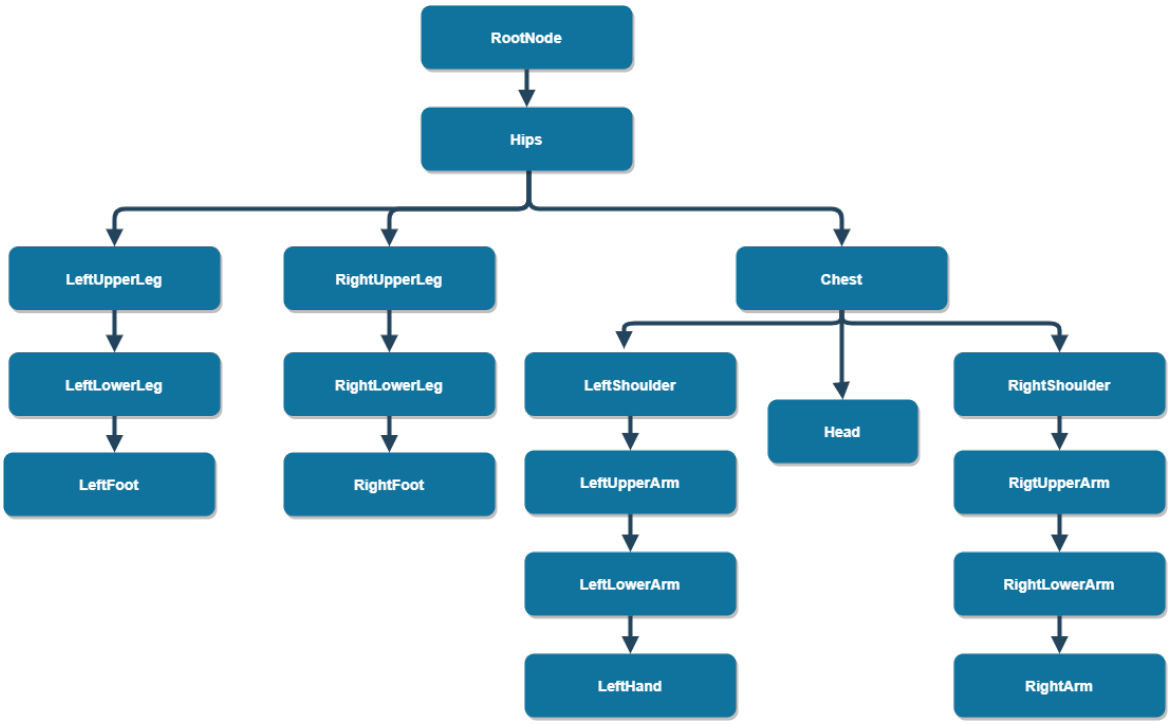


Figure 10: HMO Man Simplified Hierarchy Tree

3.2 Movement

To implement a smooth player movement we used a model based on simple physics equations. Assuming to have already received the desired command as specified in the section 6.1 we can compute the character displacement and rotation. Several parameters are associated with the character, each identified as follows: `walkSpeed` (referred to as v), `acceleration` (referred to as a), `maxSpeed` (referred to as v_M), `bodyOrientation` (referred to as θ), `angularSpeed` (referred to as ω). If the command is ↑ or ↓ then we have to compute a translation, to do so we use the following velocity

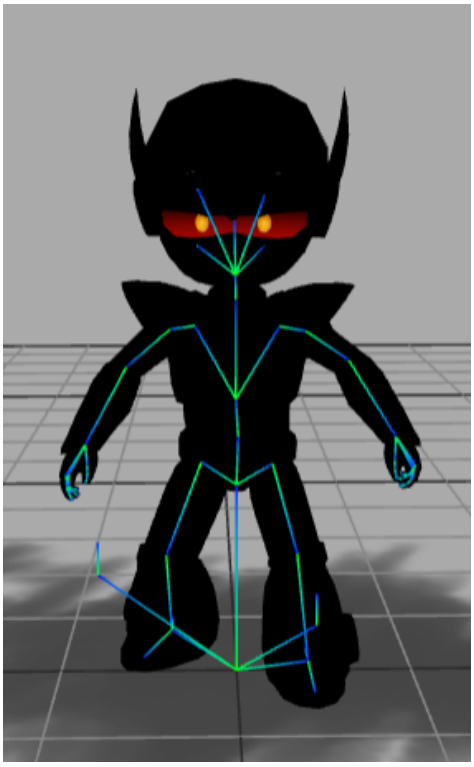


Figure 11: HMO Man whole structure

update rule:

$$v_t = max(min(v_{t-1} + \Delta t * a, v_M), -v_M) \tag{1}$$

In such a way we obtain a physically correct velocity without exceeding the bound. The acceleration a is intended positive when the command is ↑ and negative when ↓. Otherwise, if the command is not a translation one, then the velocity decreases exponentially according to the following equation:

$$v_t = max(v_{t-1} - (1 - 2^{-10 * \frac{v_{t-1}}{v_M}}) * \Delta t * a, 0) \tag{2}$$

Again, The acceleration a is positive when $v > 0$ and negative when $v < 0$. Finally, the body orientation evolves as:

$$\theta_t = \theta_{t-1} + \Delta t * \omega \tag{3}$$

Where $\omega > 0$ if the command is ← and negative is the command is →

4 Gameplay

We structured the game into 2 levels which propose 2 different tasks. In this section, we will present the gameplay highlighting the implementation details.

4.1 Level 1

At the start of the game, the main character finds himself in a dark room, which he can explore using a torchlight to illuminate his way. To restore the power, the player has to navigate through a maze to locate the generator and place it on the designated platform. Once all these steps are completed, the lights will be reactivated, and the door leading to the next level will open. Thanks to the build engine we designed, we have created three different rooms, each with increasing dimensions and complexity. These rooms correspond to the three different difficulty levels of the game, which can be set from the initial screen. To enhance the user experience, we have included in-game hints that suggest to the user which key to press to perform an action. For example, when the character is near the generator, a vanishing hint appears, suggesting to press 'P' to pick up the generator. To implement these actions, we continuously check during every game iteration which objects are near the character's origin to trigger the appropriate action.

Each room is organized into a specific factory, and every class includes the method `isCleared()` which determines whether a level is cleared or not. This information is used from the main game loop to trigger the level transition. For example, during level 1 the function `isCleared()` returns `True` function returns True once the character has passed through the door.

(a) Top view of Level 1 in easy mode

(b) Top view of Level 1 in medium mode

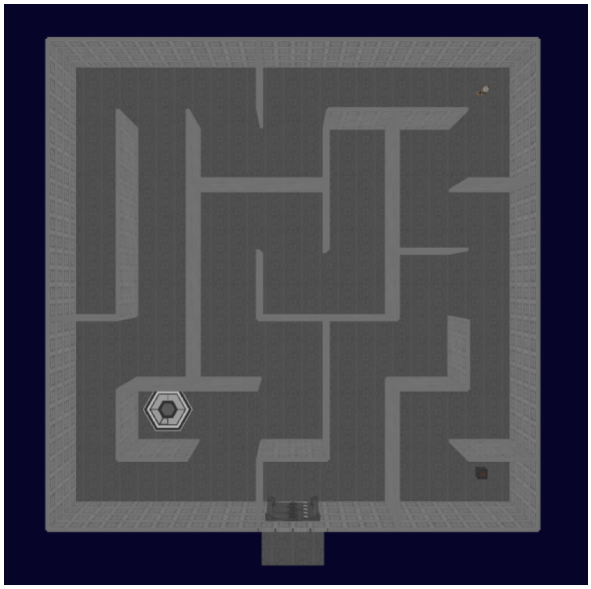


Figure 13: Top view of Level 1 in difficult mode

4.2 Transition

When the main game loop detects that the current level is cleared, it initiates a switch routine that can be summarized in 3 steps:

1. Clear the physics of the world, unbinding the physical object associated with each element in the scene.
2. Clear the elements of the scene.
3. Call the load function of the same class, passing the ID of the next level as an argument.

Thanks to the high modularity of our code, the transition phase is straightforward and does not require further specific handling.

4.3 Level 2

In the second level, the character enters a rectangular room filled with floating cubes supported by columns. Each cube has its own distinct color, which is influenced by non-white lighting. In fact, players have the option to select a light color from red, green, or blue to illuminate the room. Although each cube has a base color, the lighting alters its apparent color, making it challenging to recognize its original colors. The goal of this room is to insert a right cube sequence representing a password, under these adverse light conditions. The password is dynamically written into the open book which is located over a desk at the end of the room. The password length depends on the game's difficulty, in the easy case you have to select only 2 cubes, 3 for medium and 4 for hard.

The correct cube sequence is written in a book located on a desk at the end of the room. This sequence is displayed using the 'Great-Vibes' font and is generated as a 'ShapeGeometry.' Once we create the 'Mesh,' we apply a homogeneous transformation to project the text onto the pages of the book.

During the game loop, we constantly compute the distance to the nearest cube, allowing the user to select it only if they are close enough and standing right in front of the column. If the user fails three times to enter the code correctly, they will lose, and a 'Game Over' screen will appear. Otherwise, clearing the sequence triggers the final animation, where the character walks toward the trophy and celebrates with an animation once they reach it.

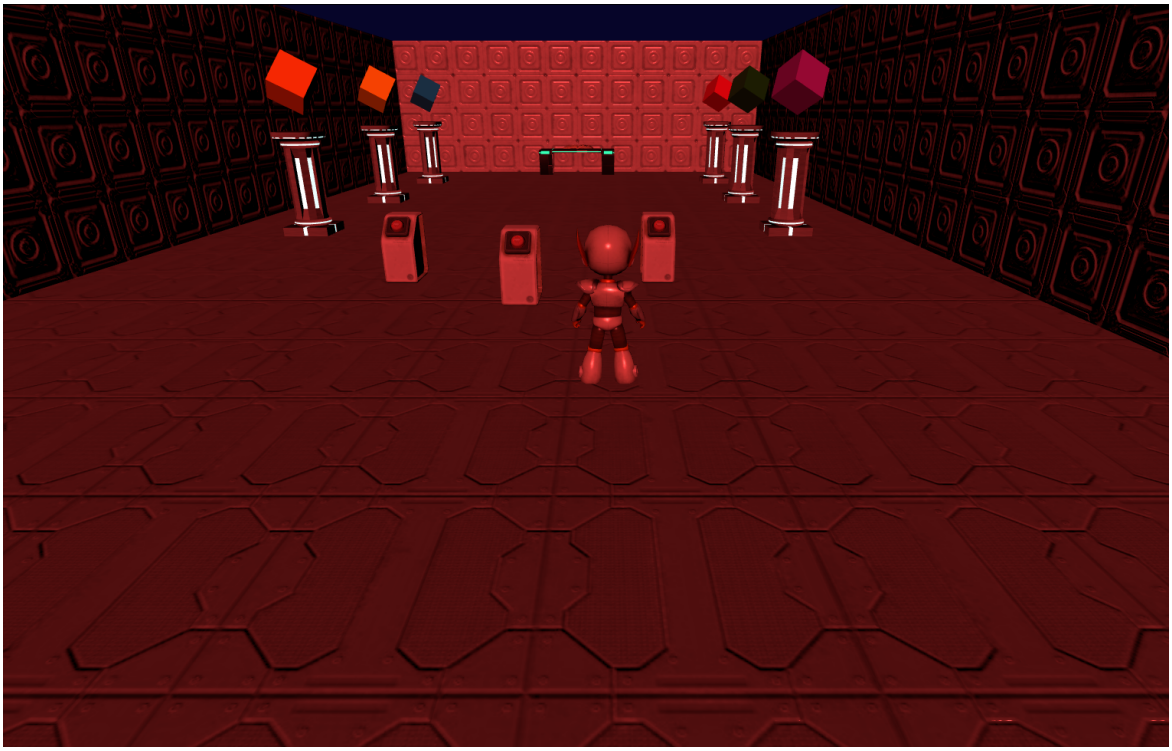


Figure 14: Frontal view of level 2

5 Animations

5.1 Basic Animations

We implemented several animations in our game application, starting with the basic ones, which can be found in **AnimationUtils**, such as translation, rotation, and rotation on one axis. On top of that, we have implemented other animations that are explained in detail as follows:

- **yes and no**: occurs when you pick a cube. It is implemented simply by rotating the neck joint from one side to another by using a quadratic interpolation

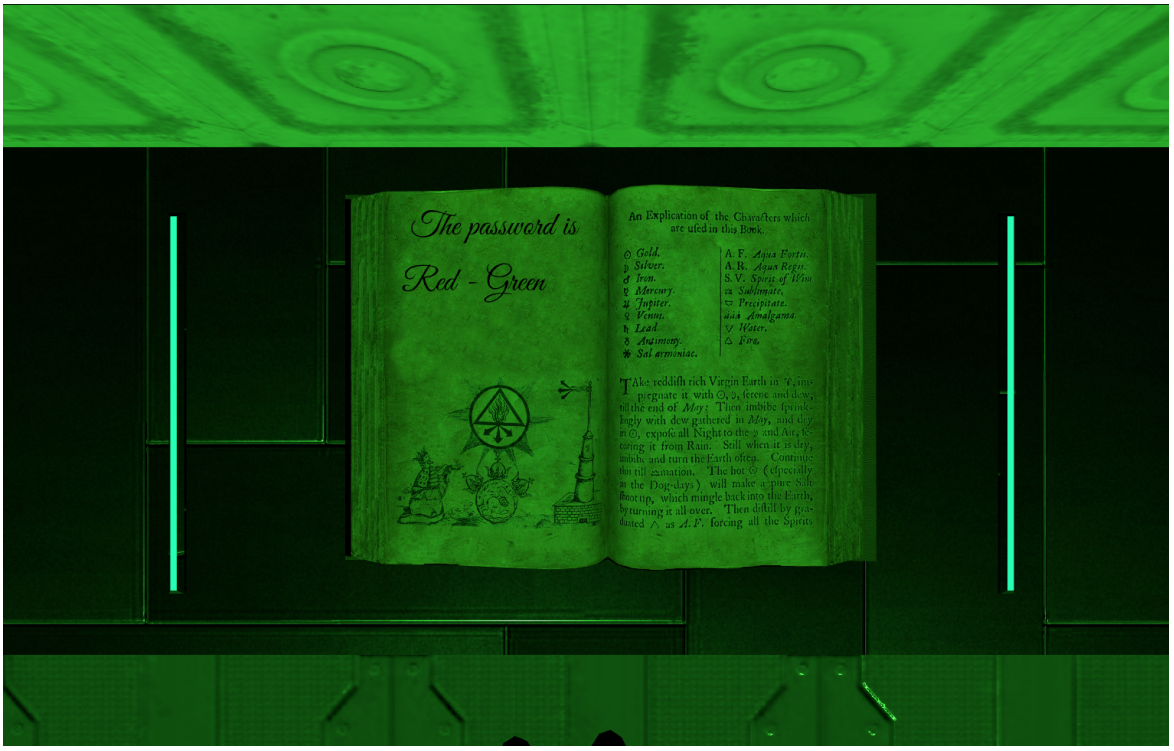


Figure 15: Book closeup

function.

- **stand**: occurs when the character velocity drops to 0. The character brings all limbs back to a rest position, in particular, all those joints that are involved in other animations and thus subject to possible changes in angles are modified.

- **holding a torch**: triggered when the user presses the 'L' key during level 1. To implement it we rotate the lower right arm joint to lift the right hand. Since this angle would be overwritten by the stand or walk animation, we had to explicitly handle these cases. In particular, in the walk animation, we have a minor movement of the right arm, while in the stand the right lower arm is not modified.

- **pressing button**: occurs when the user presses the 'P' key in front of a button. Before starting the animation we slightly adjust the character pose to be in the desired position, then we change the shoulder and arm joint angles to first bring the hand to the desired height and then move it back to simulate pressing.

5.2 Finite State Machine Animations

For cyclic animations such as walks and celebrations, we used a Finite State Machine (*FSM*). In particular, we defined a set of animation phases modeled as the states of the machine and we defined conditions that allow transitioning among states. If the *FSM*

is cyclic then the last state has a transition to the first state. In our implementation, we identify the state with an integer number, while a `switch case` encapsulates all the states. Every state is divided into 2 parts: an 'action' section that contains all the animations that have to be executed and a 'transition' section with modifies the state number depending on which conditions are met. Every *FSM* is represented by a class, this representation allows us to easily define variations. For instance, the class `Walk` represents the basic *FSM*, whereas `MainCharacterWalkWithLight` extends `Walk` adding the slight shoulder oscillation we mentioned before.

6 User Interaction

6.1 Key handler

Since Javascript is an event-based programming language, the key press event is caught by using an event listener. A character property is changed as a consequence of receiving the event, in this way the update loop can recognize what the command is. We can also disable the key handler when we run animations or text boxes.

6.2 Text box

In order to explain the rules to the player and provide background information for the game, we extensively used text boxes. The text box is implemented as a hidden HTML . When we want to display text, we make the visible and disable all key listeners. The text is revealed progressively by introducing a small timeout after each character.

6.3 Hints

To make the game easier we show hints about which key the user has to press at a certain time. Again this is implemented as a hidden HTML which is shown only when the character is close enough to a certain item. When the player moves away the text opacity gradually decreases at every `setInterval()` call.

6.4 Home Screen

Before the actual game begins, the player lands on a Home Screen with three clickable options:

- **PLAY**: redirects the user to a different page that loads the actual game.
- **RULES**: shows information about the game, including instructions on how to play.

- **SETTINGS**: allows the user to modify the difficulty of the game and enable or disable audio.

All the dynamic elements of the page are handled by Javascript which also maintains the state of settings. When the user clicks play all the setting values are passed in GET adding parameters to the URL.

References

[1] Three.js documentation.

[2] Tweenjs documentation.

[3] Chosic - free music for games, 2023.

[4] Stefan Hedman. Cannon.js documentation.

[5] Mr.doob. Stats.js github repository.

[6] Sketchfab. Sketchfab - 3D Models for Download, 2023.