Final group project for “Interactive Graphics” academic course, master degree in “Engineering in Computer Science”, “Sapienza, Università di Roma”, A.Y. 2021/2022.

Authors:
- Ionta Antonio (1469982)
- Laurenzi Giordano (1750070)
- Teglia Simone (1836794)

1. Introduction

ROBOT-ESCAPE is an escape room themed game in which the robot protagonist has to solve some mini challenges in a given amount of time in order to win and collect a trophy.

The robot can wander in a sci-fi environment and interact with objects that may be useful to solve the challenges or may distract it in pursuing its goal, just as in a real escape room.

The player has to open four doors in order to win the game by:
- collecting a key;
- moving an object to reveal a secret code;
- pressing a button;
- knocking on a door.

Dancing tunes and sliding doors sounds, together with different animations played by the character, the doors and even the camera, accompany the player throughout the game.

2. Objects

2.1 Grounds

There are actually three flat grounds:

- the ground of the main space subdivided in smaller room;
- the ground of the narrow hallway beyond the final room;
- the ground of the final space reachable through the above-mentioned hallway.

2.1.1 Create

All the three ones are realized as `PlaneBufferGeometry` with `MeshPhongMaterial`; a specific texture is bound to the material.

The code for realizing them is put in common with the help of a function in `src > App > Utils > RoomFunctions.js`:

```
export function create_ground(w, h, texture) {
    const geometry = new THREE.PlaneBufferGeometry(w, h);
    const material = new  THREE.MeshPhongMaterial({
        map: texture.baseColor,
        bumpMap: texture.normal
    });

    return new THREE.Mesh(geometry, material);
}
```

2.1.2 Instantiate

The rectangular shapes are put in place exploiting some transformations. As an example, let's see some snippets of code for instantiating the ground of the final space from `src > App > Utils > ThrophyRoom.js`:

```
[...]
        targetName = "trophyRoom.ground";
        object = this.objects[targetName];
        objectSize = this.groundsSize[targetName];

        referenceName = "trophyRoom.hallway.ground";
        referencePosition = get_center(this.objects[referenceName]);
        referenceSize = this.groundsSize[referenceName];

        object.position.set(referencePosition.x, referencePosition.y,
referencePosition.z);
        object.rotation.x = -Math.PI/2;
        object.translateY(referenceSize.height/2 + objectSize.height/2);
[...]
```

Note that the actual transformations are parameterized by the dimensions of the other objects of the scene, thus becoming robust to an eventual resizing of the environment.

2.2 Walls

Within the scene there are 14 walls. We can classify them among two main groups:

- Standard walls, i.e. the flat ones, without holes.
- Walls with doors, i.e. the ones which have a hole in correspondence of the respective door.

2.2.1 Standard

2.2.1.1 Create

They are realized as `BoxBufferGeometry` with `MeshPhongMaterial`; a specific texture is bound to the material.

The code for realizing them is put in common with the help of a function in `src > App > Utils > RoomFunctions.js`:

```
export function create_wall(w, h, d, segments, texture) {
    const geometry = new THREE.BoxBufferGeometry(w, h, d, segments, segments, segments);
    const material = new  THREE.MeshPhongMaterial({
        map: texture.baseColor,
        bumpMap: texture.normal
    });

    return new THREE.Mesh(geometry, material);
}
```

2.2.1.2 Instantiate

The parallelepipeds are put in place exploiting some transformations. As an example, let's see some snippets of code for instantiating one of the standard walls of the main space from `src > App > Utils > MainRoom.js`:

```
[...]
        object = this.objects["mainRoom.wall.big.vertical.right"];
        object.translateY(PERIMETER_WALL_SIZE.height / 2);
        object.translateX(GROUND_SIZE.width / 2 +
this.wallSizeMap.get("mainRoom.wall.big.vertical.right").width / 2);
[...]
```

Note that the actual transformations are parameterized by the dimensions of the other objects of the scene, thus becoming robust to an eventual resizing of the environment.

2.2.2 With Door

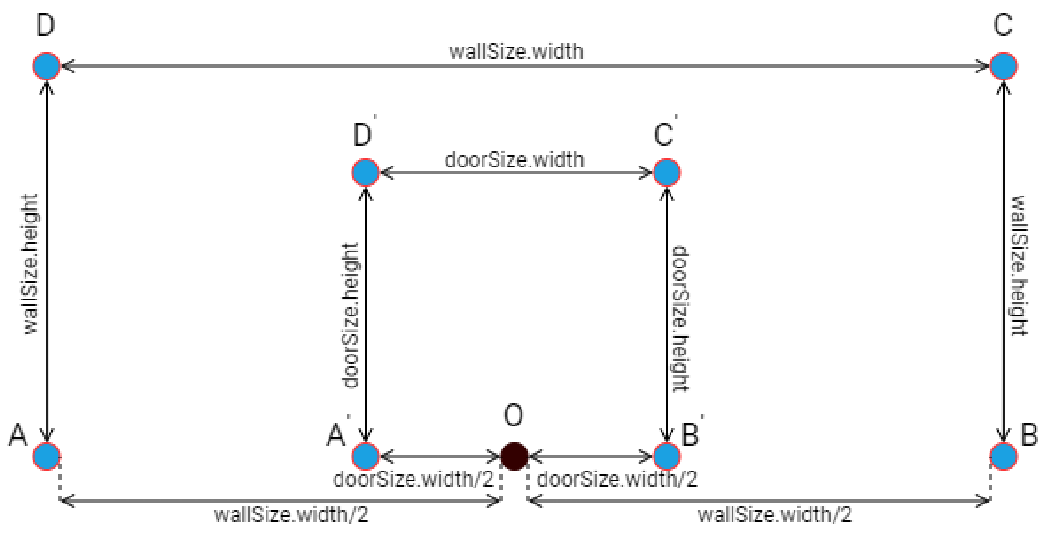
2.2.2.1 Create

They are realized as `ExtrudeGeometry` (*ThreeJS* object for extruding a 2D shape to a 3D geometry), on the basis of a custom-defined `Shape`, paired with `MeshPhongMaterial`; a specific texture is bound to the material.

The code for realizing them is put in common with the help of a function in `src > App > Utils > RoomFunctions.js`:

```
export function create_wall_with_door(wallSize, doorSize, texture) {
    const wallPoints = {
        A: {x: -wallSize.width/2, y: 0},
        B: {x: wallSize.width/2, y: 0},
        C: {x: wallSize.width/2, y: wallSize.height},
        D: {x: -wallSize.width/2, y: wallSize.height}
    };
    const doorPoints = {
        A: {x: -doorSize.width/2, y: 0},
        B: {x: doorSize.width/2, y: 0},
        C: {x: doorSize.width/2, y: doorSize.height},
        D: {x: -doorSize.width/2, y: doorSize.height}
    };

    const wallShape = new THREE.Shape();
    wallShape.moveTo(wallPoints.A.x, wallPoints.A.y);
    wallShape.lineTo(wallPoints.B.x, wallPoints.B.y);
    wallShape.lineTo(wallPoints.C.x, wallPoints.C.y);
    wallShape.lineTo(wallPoints.D.x, wallPoints.D.y);

    const doorPath = new THREE.Path();
    doorPath.moveTo(doorPoints.A.x, doorPoints.A.y);
    doorPath.lineTo(doorPoints.B.x, doorPoints.B.y);
    doorPath.lineTo(doorPoints.C.x, doorPoints.C.y);
    doorPath.lineTo(doorPoints.D.x, doorPoints.D.y);

    wallShape.holes.push(doorPath);
    const extrudeGeometry = new THREE.ExtrudeGeometry(wallShape, {depth: wallSize.depth,
bevelEnabled: false})

    // it's necessary to apply these settings in order to correctly display the texture on
a shape geometry
    texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
    texture.repeat.set( 0.05, 0.05 );
    const material = new THREE.MeshPhongMaterial({
        map: texture
    });
    const wall = new THREE.Mesh(extrudeGeometry, material);

    return wall;
}
```

The basic shape and the hole within it are modeled respecting this planar points pattern:



2.2.2.2 Instantiate

The extruded parallelepipeds are put in place exploiting some transformations. As an example, let's see some snippets of code for instantiating one of the walls with hole of the main space from `src > App > Utils > MainRoom.js`:

```
[...]
        objectName = "mainRoom.wall.big.vertical.withDoor";
        object = this.objects[objectName];
        object.rotation.y = Math.PI / 2;
        objectMeasure = get_measure(object);
        object.translateZ((GROUND_SIZE.width / 2 -
this.wallSizeMap.get("mainRoom.wall.small.horizontal").width));

[...]
```

Note that the actual transformations are parameterized by the dimensions of the other objects of the scene, thus becoming robust to an eventual resizing of the environment.

2.3 Light Bulbs

The three light bulbs are realized as rectangular cuboids thanks to the `BoxGeometry` class and positioned right above the doors in the main room with a code similar to the following:

```
const geometry = new THREE.BoxGeometry(0.5, 0.5, 0.5);
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 });
object = new THREE.Mesh(geometry, material);
object.name = "bulbRight";
this.objects["bulbRight"] = object;
object.position.set(10.1, 11.5, 0);
this.scene.add(object);
```

Just as when a door is opened, the color of the material changes from red to green through the following line of code

```
this.main.scene.getObjectByName("bulbLeftRear").material.color.set(0x00ff00);
```

2.4 Imported Models

The imported models used in the project are located in `src > models`.

Here a complete list of them, with the indication of their original source (the names used in the list recall the ones assigned to the corresponding *ThreeJS* objects after the import):

- robotExpressive
- sciFiCrate
- desk
- oscilloscope
- desk3
- caldurun
- redButton
- sign: we created this object in blender since it is a very simple one. It's composed only of two meshes.
- bucket
- hologramConsole
- scifiTerminal
- finalDoor
- slideDoor
- key
- pinPad
- notepad
- scifiTable
- arrow
- cup

2.4.1 Load

All the models are loaded with the help of `GLTFLoader` according with and asynchronous workflow whose progress is tracked with `LoadingManager`. The source code is located in a dedicated JS class `src > App > Models.js`:

```
async loadModels() {
    let promises, models;
    promises = models = [];

    for(const info of MODELS_INFO) {
        promises = promises.concat(this.gltfLoader.loadAsync(info.path))
    }
    const rawModels = await Promise.all(promises);
    for(let i=0; i<rawModels.length; i++) {
        models = models.concat(this.setupModel(rawModels[i], MODELS_INFO[i].name));
    }

    return models;
}
```

For letting the user start the interaction with the full scene already loaded, the above-mentioned functions are awaited since the whole application start-up.

`src > index.js`:

```
import Main from "./App/Main.js";

init();

async function init() {
    await Main.build();
}
```

`src > App > Main.js`:

```
static async build() {
    let scene = await setScene();
    [...]
    return new Main({
        [...]
        scene: scene,
        [...]
    });
}

[...]

async function setScene() {
    let scene = new THREE.Scene();
    [...]
    let progressBar = document.getElementById("progress-bar");

    const manager = new THREE.LoadingManager();

    manager.onProgress = (url, loaded, total) => {
        progressBar.value = (loaded / total) * 100;
    };

    manager.onLoad = function () {
        console.log("Just finished loading models");
        document.getElementById("loading-screen").style.opacity = 0;
        document.getElementById("landing-page").style.opacity = 1;
    };

    // Wait for models to be loaded
    const models = await new Models(manager).loadModels();
    for (const model of models) {
        scene.add(model);
    }

    return scene;
}
```

2.4.2 Instantiate

The static objects are put in place exploiting some transformations. As an example, let's see some snippets of code for instantiating one of the slide doors of the main space from `src > App > Utils > MainRoom.js`:

```
[...]
        object = this.objects["mainRoom.slideDoor.left.rear"];
        object.scale.set(SLIDE_DOOR_SCALE_FACTOR, SLIDE_DOOR_SCALE_FACTOR,
SLIDE_DOOR_SCALE_FACTOR);
        objectSize = get_measure(object);
        position = get_center(this.objects["mainRoom.wall.small.vertical.rear.withDoor"]);
        object.position.set(position.x, 0, position.z);
[...]
```

Note that the actual transformations are parameterized by the dimensions of the other objects of the scene, thus becoming robust to an eventual resizing of the environment.

2.5 Utils

Several different functions are exploited to accomplish common tasks relative to object instantiation, etc. We report here some of the most meaningful

2.5.1 get_center

`src > App > Utils > Functions.js`:

```
export function get_center(object) {
    [...]
    let center = new vec3();
    let box = new THREE.Box3().setFromObject(object);
    box.getCenter(center);
    [...]
    return center;
}
```

This function exploits the properties of *ThreeJS* bounding boxes for computing the position center of a given object.

2.5.2 get_measure

`src > App > Utils > Functions.js`:

```
export function get_measure(object) {
    [...]
    let measure = new vec3();
    let box = new THREE.Box3().setFromObject(object);
    box.getSize(measure);
    [...]
    return measure;
}
```

This function exploits the properties of *ThreeJS* bounding boxes for computing the sizes of a given object with respect to the three cardinal axes.

3. Lights and Textures

3.1 Lights and Shadows

The scene contains the following lights:

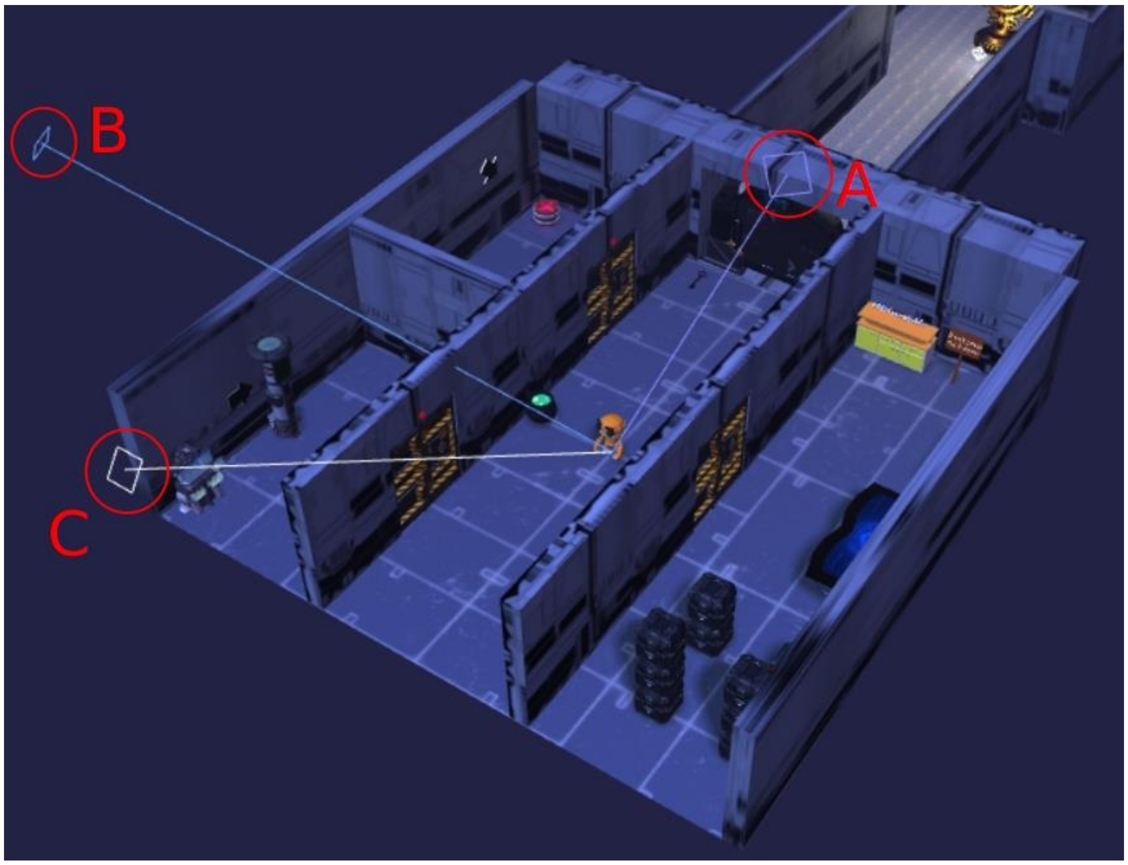
- 1x `AmbientLight`
- 3x `DirectionalLight`
- 3x `PointLight`

Shadows are realized through the built-in *ThreeJS* renderer shadow map. Just a single directional light casts shadows. All the objects, except from the walls and the grounds, cast shadows. The grounds receive shadows.

3.1.1 Ambient Light

Color	*Intensity*
White (#FFFFFF)	0.3

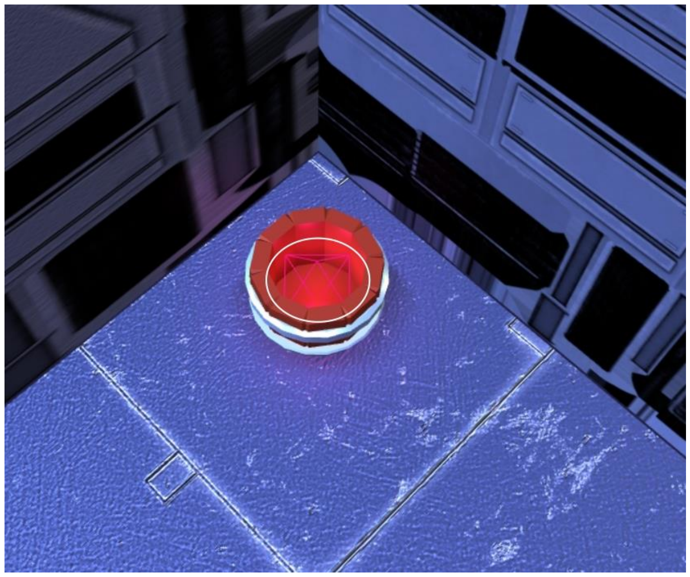
3.1.2 Directional Lights



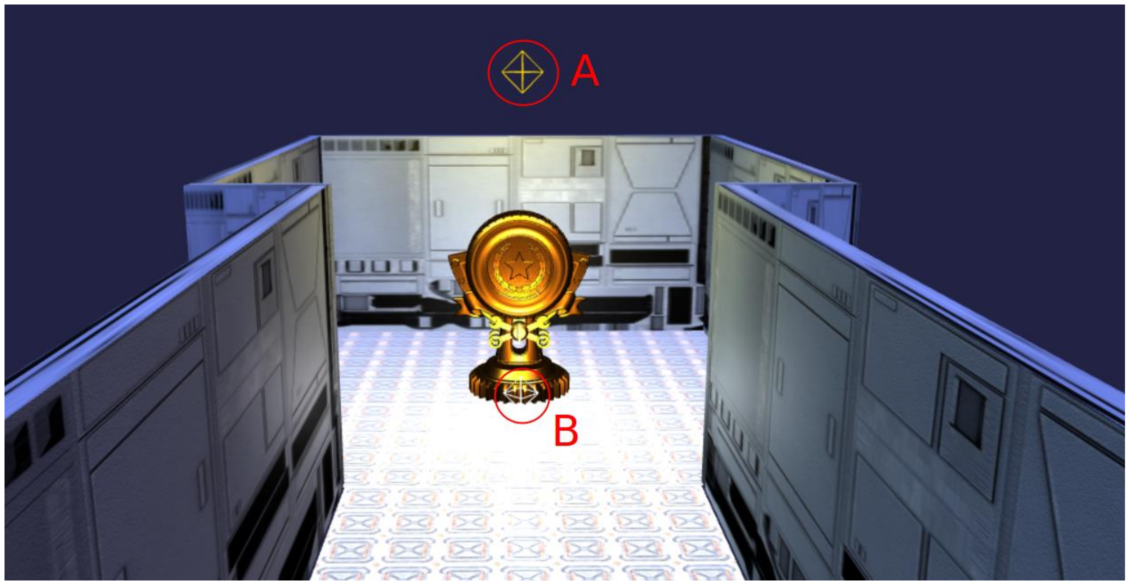
Light	*Color*	*Intensity*	*Position*	*Cast Shadow*
A	Malibu like (#8888FF)	2	(30, 60, 30)	Yes

B	Cornflower Blue (#6495ED)	2	(-30, 40, 30)	No
C	White (#FFFFFF)	0.3	(0, 40, 50)	No

3.1.3 Point Lights



Color	*Intensity*	*Max Range*	*Cast Shadow*
Deep Pink (#FF1493)	2	4	No



Light	*Color*	*Intensity*	*Max Range*	*Decay*	*Position*	*Cast Shadow*
A	Gold (#FFD700)	2	120	2	(0, 20, -80)	No
B	White (#FFFFFF)	2	30	1	(0, 2, -85)	No

3.1.4 Shadows Configuration

From `src > App > Main.js`:

```
setRenderer() {
    [...]
    this.renderer.shadowMap.enabled = true;
    this.renderer.shadowMap.type = THREE.VSMShadowMap;
    [...]
}
```

From `src > App > World.js`:

```
setLights() {
    [...]
    this.directionalLight.shadow.camera.near = 10;
    this.directionalLight.shadow.camera.far = 150;
    this.directionalLight.shadow.camera.right = 60;
    this.directionalLight.shadow.camera.left = -60;
    this.directionalLight.shadow.camera.top = 60;
    this.directionalLight.shadow.camera.bottom = -60;
    this.directionalLight.shadow.mapSize.width = (1024 * 2) / 3;
    this.directionalLight.shadow.mapSize.height = (1024 * 2) / 3;
    this.directionalLight.shadow.radius = 4;
    this.directionalLight.shadow.bias = -0.0005;
    [...]
}
```

3.1.5 Utils

Several different functions are exploited to accomplish common tasks relative to lighting. We report here one of the most meaningful.

3.1.5.1 enable_shadows

`src > App > Utils > Functions.js`:

```
export function enable_shadows(object, cast, receive) {
    object.traverse(function (child) {
        if (child.isMesh) {
            child.castShadow = cast;
            child.receiveShadow = receive;
        }
    });
}
```

It handles the shadow configurations for an object with a hierarchical structure. `cast` and `receive` are boolean parameters used to establish if the given object should cast shadows, receive shadows, both or none.

3.2 Textures

The imported textures used in the project are located in `src > textures`.

Here a complete list of them, with the indication of their original source:

- ground
- ground-metal
- wall

3.2.1 Load

All the textures are loaded with the help of `TextureLoader` according with an asynchronous workflow. The source code is located in a dedicated JS class `src > App > Models.js`:

```
async loadTextures() {
    let promises, textures;
    promises = textures = [];

    for(const info of TEXTURES_INFO) {
        promises = promises.concat(this.textureLoader.loadAsync(info.path))
    }
    const rawTextures = await Promise.all(promises);

    for(let i=0; i<rawTextures.length; i++) {
        textures = textures.concat(this.setupTexture(rawTextures[i],
TEXTURES_INFO[i].name));
    }

    return textures;
}
```

For letting the user start the interaction with the full scene already loaded, the above-mentioned functions are awaited since the whole application start-up.

`src > index.js`:

```
import Main from "./App/Main.js";

init();

async function init() {
    await Main.build();
}
```

`src > App > Main.js`:

```
static async build() {
    let textures = await setTextures();
    [...]
    return new Main({
        [...]
        textures: textures,
        [...]
    });
}

[...]

async function setTextures() {
    return new Models().loadTextures();
}
```

3.2.2 Apply

3.2.2.1 Ground

The texture named *ground* is applied to the ground of the main space, providing also a bump mapping. We can see here some code snippets from `src > App > MainRoom.js`:

```
[...]
    create() {
        let mesh, texture;

        /**
         * Ground
         */

        texture = {
            baseColor: this.main.textures.find(e => e.name ==
"mainRoom.ground.baseColor"),
            normal: this.main.textures.find(e => e.name == "mainRoom.ground.normal")
        }

        mesh = create_ground(GROUND_SIZE.width, GROUND_SIZE.height, texture);
        [...]
    }
```

and from `src > App > Utils > RoomFunctions.js`:

```
export function create_ground(w, h, texture) {
    [...]
    const material = new  THREE.MeshPhongMaterial({
        map: texture.baseColor,
        bumpMap: texture.normal
    });
    [...]
}
```

3.2.2.2 Ground-Metal

The texture named *ground-metal* is applied to the ground of the rear space, the one which contains the trophy, providing also:

- normal mapping,

- roughness mapping,
- metalness mapping,
- ambient occlusion mapping.

We can see here some code snippets from `src > App > TrophyRoom.js`:

```
[...]
    create() {
        let objectName, mesh, size, texture;

        /**
         * Grounds
         */

        texture = new Map([
            ["baseColor", this.main.textures.find(e => e.name ==
"trophyRoom.ground.baseColor")],
            ["normal", this.main.textures.find(e => e.name ==
"trophyRoom.ground.normal")],
            ["metallic", this.main.textures.find(e => e.name ==
"trophyRoom.ground.metallic")],
            ["roughness", this.main.textures.find(e => e.name ==
"trophyRoom.ground.roughness")],
            ["ambientOcclusion", this.main.textures.find(e => e.name ==
"trophyRoom.ground.ambientOcclusion")]
        ]);
        for(let [label, tex] of texture) {
            configure_texture(tex, {u: 7, v: 15}, THREE.RepeatWrapping);
        }

        objectName = "trophyRoom.hallway.ground";
        size = this.groundsSize[objectName];
        mesh = create_ground(size.width, size.height, null);
        apply_texture(texture, mesh);
        mesh.name = objectName;

        [...]

        texture = clone_texture(texture);
        for(let [label, tex] of texture) {
            configure_texture(tex, {u: 12, v: 12}, THREE.RepeatWrapping);
        }

        objectName = "trophyRoom.ground";
        size = this.groundsSize[objectName];
        mesh = create_ground(size.width, size.height, null);
        apply_texture(texture, mesh);
    }
```

Observe that this texture is applied following a repetition pattern which has different parameters for the hallway ground and for the main room ground, with the aim of adapting the theme to the different planar dimensions.

3.2.2.3 Wall

The texture named *wall* is applied to all the walls within the scene, but with different mappings for the flat walls and the extruded ones.

The standard walls receive the texture exactly with the same bump mapping already discussed for the ground.

Texture application on the extruded walls, instead, consists in a custom mapping which exploits texture's base color only in `src > App > Utils > RoomFunctions.js`:

```
export function create_wall_with_door(wallSize, doorSize, texture) {
    [...]
    texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
    texture.repeat.set( 0.05, 0.05 );
    const material = new THREE.MeshPhongMaterial({
        map: texture
    });
    const wall = new THREE.Mesh(extrudeGeometry, material);
    [...]
}
```

The reason why these adjustments are needed can be summarized stating that `ShapeGeometry` uses vertex position values in order to generate UV data; this means the texture coordinates of `ShapeGeometry` exceed the usual range of [0,1]; thus without scaling you won't get proper results.

3.2.3 Utils

Several different functions are exploited to accomplish common tasks relative to textures. We report here some of the most meaningful

3.2.3.1 configure_texture

```
export function configure_texture(texture, repeat, wrap) {
    texture.repeat.set(repeat.u, repeat.v);
    texture.wrapS = texture.wrapT = wrap;

    return texture;
}
```

`wrapS` and `wrapT` properties define, respectively, how the texture is wrapped horizontally/vertically and corresponds to "U"/"V" in UV mapping. Wrapping parameters determine what happens if the texture coordinates *s* and *t* are outside the (0,1) range. In many cases in this project the "repeat" approach is adopted: the texture repeats itself infinite times along both axes. `repeat` property defines how many times the texture is repeated across the surface, in each direction U and V.

3.2.3.2 apply_texture

```
export function apply_texture(texture, mesh) {
    if(texture.get("normal") != null
        && texture.get("metallic") != null
        && texture.get("roughness") != null
        && texture.get("ambientOcclusion") != null) {
            mesh.material = new THREE.MeshStandardMaterial({
                map: texture.get("baseColor"),
                normalMap: texture.get("normal"),
                roughnessMap: texture.get("roughness"),
                metalnessMap: texture.get("metallic"),
                aoMap: texture.get("ambientOcclusion")
            });
    }
    else if(texture.get("normal") != null) {
        mesh.material = new THREE.MeshPhongMaterial({
            map: texture.get("baseColor"),
            bumpMap: texture.get("normal")
        });
    }
    else {
        mesh.material = new THREE.MeshPhongMaterial({
            map: texture.get("baseColor"),
        });
    }

    return mesh;
}
```

It applies the given texture to the given mesh. Note that `texture` parameter is actually a `Map` which could contain information about several mappings to be bound to the texture. If many mappings are present, the mesh needs a `MeshStandardMaterial` for a correct application of all of them. On the contrary, if the texture is paired with just a normal mapping, a `MeshPhongMaterial` with a bump mapping is chosen as performance-saving solution.

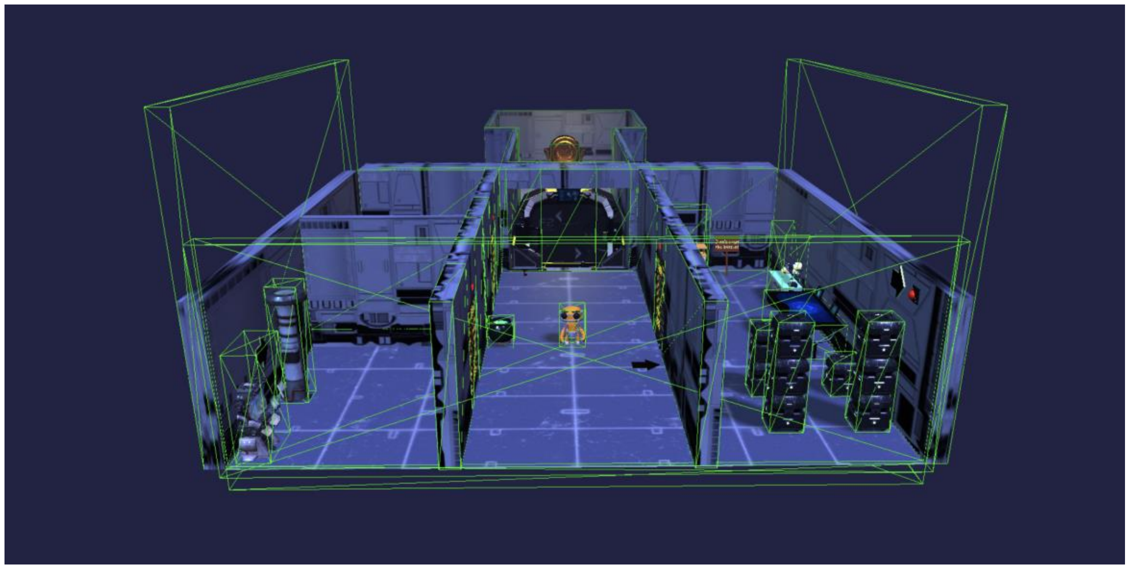
4. Physical Engine

In order to give a realistic feel to the game we decided to implement a physical engine. After some research we elected `cannon-es` as best candidate.

`cannon-es` is a lightweight engine and a maintained fork of the more famous library *cannon.js*. It creates an alternative world where physics is present and each object inside this world is affected by the laws of physics decided by some parameters. Any object in the physical world is called `body` and it can be linked to any *three.js* mesh. In this way we obtained a realistic game with gravity and collisions.

Each body have different properties like `mass` or `material`. If a body has a mass equals to 0 it means that it is static and it will not be affected by any kind of force.

In the following image we can appreciate all the bodies linked with *three.js* meshes in the scene.



It’s clearly visible that the lateral walls are higher that the meshes, this is to prevent the player falling out of the map. There is also an invisible wall in front of the map that has the same scope.

4.1 Robot Collision Box

In order to make the robot interact with the environment we created a body that acts like a collision box, then we linked the movement of the collision box to the movement of the robot’s mesh. The movement of the box is obtained by updating the velocity of the body along the x and z axes.

4.2 Jump

In the same way as we update the horizontal velocity to make the robot move, we can update the y velocity to make the robot perform a jump. In order not to make the robot jump even when it's in the air we created a list that contains the objects on top of which the robot can jump. Every time the robot collide with one of this, the variable `canJump` is updated to true. Obviously, when the robot is touching the ground the variable `canJump` is always true.

5. User Interaction

5.1 Difficulty Settings

As the user opens the application the landing page shows up. Here the user can read the set of rules or select the difficulty of the game that he is going to play. There are three levels of difficulty: Easy, Medium and Hard (2020). The difficulty setting will affects the timer and the hints in the scene according to the following schema:

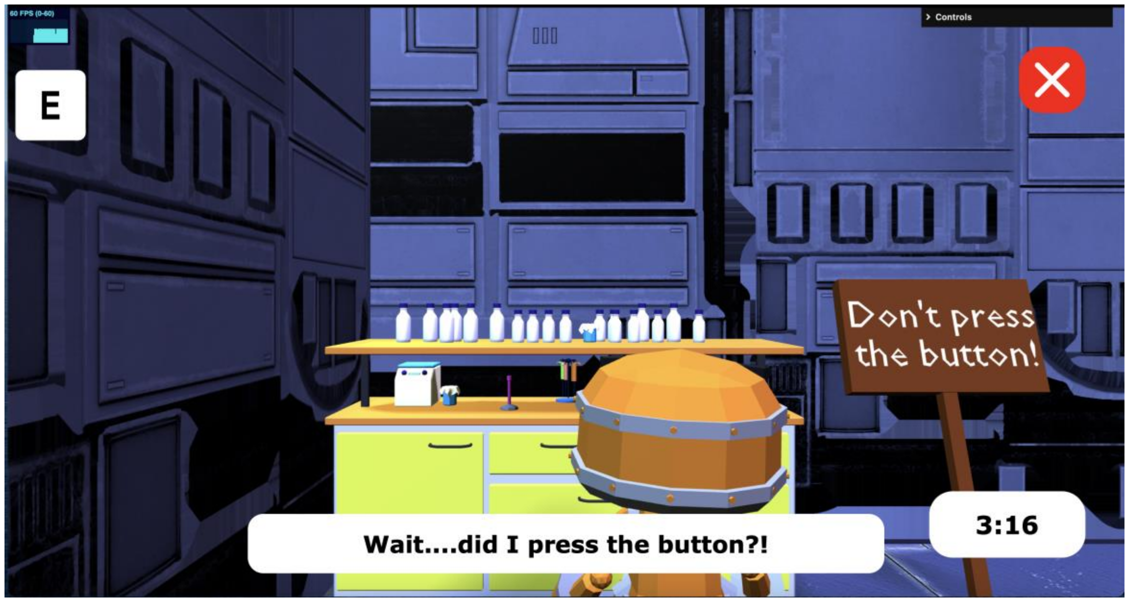
	Easy	Medium	Hard (2020)
Time	4 min 30 s	2 min	90 s
Hints	4	2	0

5.2 Interaction with Objects

During the game the robot needs to interact with some objects in order to complete the challenges.

When the robot is inside the activation zone of an object the user can press `E` to activate an animation that brings the camera near the robot and shows the interaction. If the `E` key is pressed but the robot is not inside the activation zone, the robot will shake its head. Pressing the exit button on the top right part of the screen will resume the game bringing the camera to the original position.

To let the user know when he's inside an activation zone a icon of the `E` key will appear on the top left part of the screen.



Wait....did I press the button?!

6. Animation

The animations are implemented using *tween.js*, by changing over time the values of rotation or position of different parts of the models. In order to have smooth animations, different easing functions were used thanks to the `easing` method.

All animations are realized inside `src > App > Animations.js` that contains all the functions needed to produce the desired effect, by combining tweens of different parts of the model that is passed as input.

A useful tool that allowed to quickly add a very basic user interface to interact with the 3D scene is *lil-gui*, a drop-in replacement for *dat.gui*. With the formula

```
let boneFolder = this.gui.addFolder("Bone");
boneFolder
    .add(this.bone.rotation, "x")
    .min(-Math.PI)
    .max(Math.PI)
    .step(0.0001);
boneFolder.close();
```

it was added to the user interface the possibility to modify the values of rotation and position of the different parts of the robot before writing them in the code, which, together with the fact that all the parts of the robots were assigned to homonym class variables to rapidly access them, really simplified the work.

6.1 Robot

6.1.1 Head

The functions `yesTween(robot)` and `noTween(robot)` realize the animations for making the robot's neck rotate to simulate positive and negative affirmations. In both cases, the neck is rotated twice before returning to the initial position, but, while in the former the rotation happens along the x axis, in the latter it happens along the y axis.

6.1.2 Death

The function `deathTween(robot)` simulates the death of the robot by making it fall to the ground. It has been realized by rotating and positioning the body in such a way that its back ends right above the surface of the floor. To complete the animation, shoulders, upper and lower arms and upper and lower legs are rotated to follow-through, and the head is rotated and positioned in such a way to fall next to the body: thanks to the use of the easing function *Bounce Out*, it seems like it is bouncing on the ground before stopping.

6.1.3 Lean

The function that handles this animation, `leanTween(robot, angle)`, simply takes an angle of rotation to be used to make the robot lean forward and then go back to the initial position.

This animation is coupled with the interaction with the bucket: the robot leans forward to see what the bin contains and then has to go back to the initial position.

6.1.4 Idle

In order to create such an animation, different functions were created to handle the rotation of the head and of the upper and lower legs of the robot and the position of its body. In order to produce an effective outcome, indeed, while the legs are moving to simulate the bending of the knees, the body is translated along the y axis and the head is rotated. The function `Math.random()` has been used to decide whether the robot should rotate its head to its left, to its right or sideways with different easing functions like the `bounce.inOut` to produce different effects.

6.1.5 Turn

`turnRightTween(robot)`, `turnFrontTween(robot)`, `turnBackTween(robot)` and `turnLeftTween(robot)` realize the animation to change the orientation of the robot. They are often used to put its body in the right position for an effective interaction with the objects in the scene, as it happens with the pin-pad, the bucket and the red button, but they are important especially because they are associated to the arrow keys, to make the robot turn in the direction of the walk.

6.1.6 Walk

The walk animation is the one that required most attention as it had to be synchronized with the movement triggered by the press of the arrow keys. In order to do so, a semaphore is used in such a way that a new walk animation can not begin if the previous has not been completed yet. Moreover, to handle the fact that the arrows can be pressed once to make the robot do just one step, a boolean variable `leftToFront(robot)` is used to understand which leg has to be put forward and which backward but also to realize an alternation between them, not to make the robot begin the walk always with the same leg.

To make the animation run as smooth as possible, while the left leg goes forward and the right leg goes backward, the left arm goes backward and the right arm goes forward and the body is rotated along the z axis to simulate the weight of the robot shifting between legs during the movement. All of this is realized in the functions `rightToBackWalkTween(robot)`, `leftToBackWalkTween(robot)`, `rightToFrontWalkTween(robot)`, `leftToFrontWalkTween(robot)`, `bodyWalkTweenPhase1(robot)` and `bodyWalkTweenPhase2(robot)`.

6.1.7 Wave

`waveTween(robot)` and `waveFinishTween(robot)` realize the animation for the greeting of the player at the beginning of the game. While the latter simply drives the animation for making the robot return to a neutral position, the former really does the work handling the rotation of the right shoulder and upper arm, the tilt of the head, the opening of the right hand through a movement of the components of the fingers as well as the rotation along the *y* axis of the lower part of the right arm to create a waving movement.

6.1.8 Thumbs Up

This animation is realized through the use of two functions `thumbsUpTween(robot)` and `thumbsDownTween(robot)`. The former one handles the movement of the robot to raise the upper part of the right arm and incline its head. At the end, the fingers of the right hand are closed in a fist while the two parts of the thumb are rotated to produce a thumb up. The latter function is used to bring the robot to the initial position, restoring the values of rotation of the parts involved in the animation.

6.1.9 Jump

Following the jump triggered by the space-bar, `rightArmJumpTween(robot)` and `leftArmJumpTween(robot)` handles the rotation of the shoulders to simulate the effect of the gravity on the arms of the robot.

6.1.10 Dance

The dance animation is used to make the robot celebrate its victory once the trophy is reached. It is realized in the function `danceTween(robot)` that handles the rotation of the body, the legs, the shoulders and the head. To realize it in a way to represent a convincing robot dance, once the head is rotated sideways, the robot lets the lower parts of its arms fall with a bouncing movement as they reach the final position and the head does a 360 degrees spin. Once again, different easing functions such as the `bounce.out` were adopted to produce a smooth and realistic animation. Finally, the second part of the function restores the initial values of the components involved in the animation.

6.1.11 Knock

Accompanied with a tune of someone knocking on a door, the animation for the knocking is used for solving a challenge. It is realized thanks to the `knockTween(robot)` function that simply handles the tilt of the robot’s head and the rotation of the left upper and lower arm. In order to make it seem like the robot is indeed knocking, the upper arm is quickly rotated forward and backward twice before every body part involved in the animation returns to its original position.

6.2 Door

The functions that handles the doors animations are `openLeftDoor(door)`, `operRightDoor(door)` and `openFinalDoor(door)` that simply change the position along the *z* axis of the parts of the doors. To improve the effect created, the animations for the sliding doors are coupled with a tune for the opening of the doors, a *thumbs-up* and a change in the camera view to see the doors opening.

6.3 Camera

Even the camera is subject to changes in values of rotation and position over time to change the point of view. This is done to see more clearly objects like the desks, the bucket and the pin-pad but also to accompany and highlighting the beginning of the game, the doors animations and the final victory dance.

In these cases, the animations are realized in the file `src > App > Camera.js`

7. Libraries & Tools

- three.js - r141
- tween.js - v18.6.4
- cannon-es.js - v0.20.0
- cannon-es-debugger.js - v1.0.0
- lil-gui - v0.17.0

8. Browser Testing

Browser	*Performance*
Firefox	⭐⭐⭐
Chrome	⭐⭐⭐
Safari	⭐⭐
Edge	⭐⭐

9. Play the Game

Play the game here ➡️ https://sapienzainteractivegraphicscourse.github.io/final-project-ags-team/