SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF INGEGNERIA INFORMATICA,
AUTOMATICA E GESTIONALE "ANTONIO RUBERTI"

Under the bed report

INTERACTIVE GRAPHICS

Students:

Anna Carini 1771784

Adriano Izzi 2048338

Gavriel Di Nepi 2067753

Academic Year 2022/2023

Contents

1 Brief Introduction

Under the Bed (*UTD*) is a **action oriented puzzle-game**, where the goal of the player is to safely escape a haunted house by only using a torch.
Originally conceived to be **developed by four people**, the game was **concluded successfully** by only **three students**.

1.1 Game's main elements

The game tells the story of a **little girl** abandoned by her parents who must manage to escape the monsters that came out from **under the bed** by only using her **torch**. In fact, the monsters are **susceptible to light** and, if their hearts are illuminated by light (even only by a reflection...) they will block for a time, granting Claire a chance to move freely around the house.
If Claire isn't skilled enough, though, the monsters will wound her and rejoice of it. If Claire will be hurt too much, she will faint and the game will be over.
The game consists of **three levels**:

1. *Bedroom*: which serves as tutorial
2. *Hallway*: an actual level which presents the mirror-reflection functionality
3. *Living Room*: a level with a complex and changing planimetry, and which require to retrieve an object to solve it

1.2 Game Conception and Development history

The idea of game came as natural consequence of the intense **studying of light made during the course**, which lead us to the concept of a game where the light was both an atmosphere element and a tool for the player to use.
Soon after, the idea for the scenario appeared too and, in a matter of days, the concept was basically done.
The project development was then delayed until the **end of July** when the three of us gathered for a **full-immersion week of developing**, after which the foundations of the game (such as the player movement, torch and monsters) were basically functional.
At that point, the fourth person **decided to leave** the project, hence, by the end of August till now, Anna, Adriano and Gavriel not only refinished their own sections of the game, but had to **split between them all the parts still missing**, having to refactor some functionalities to adapt for the inconvenience.
Nonetheless, the result it is a **fully playable game**, which gave us a **lot of satisfaction**.

2 Gameplay Presentation and Guide

2.1 Initial Interface

Figure 1: Initial Interface

The UDT initial screen is the one above.
A player has **three choices** regarding how to start the game:

- *START GAME*: they can create a new game that will put them in the Bedroom level
- *Go to level 2*: they can start the game from the Hallway level (with full life and already grabbing the torch)
- *Go to level 3*: they can start the game from the Living Room level (with full life and already grabbing the torch)

But the functions of this interface don't end here.
The player can ask the program to show the controls (see in the next subsections for more details).
Not only that, the player can toggle on and off many options that can impact the gameplay:

- **Play with hints**: which will make **many suggestions appear** for the player, as well as tips on the controls. This greatly improves the game experience for new players.

- **Lighter Version**: in this way, the levels will contain less heavy 3D models, leading to an **improvement of performances** for low-end computers.
- **Play without monsters**: the player will experiment a **monster-free game**. This is useful for familiarizing with a level one finds difficult to win or to explore the house details.
- **Activate Shadows**: Since the shadows can have a small but noticeable impact on performances and they have some minor issues (see chapter **??**), the player can decide to put them on and off at will.

2.2 Controls

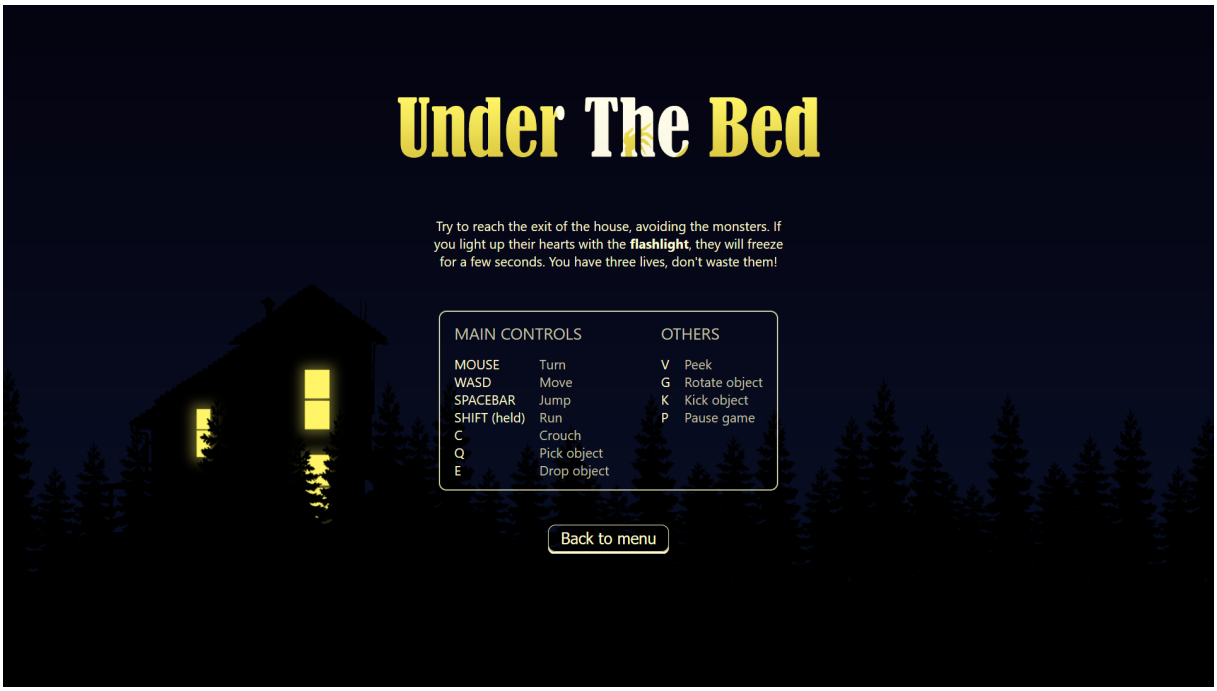


Figure 2: Game Controls

The game controls page (that can also be consulted by pausing the game) contains all the informations a player must known to win the game.
Here, we go a little more in details of what each section of this page says.
The main controls are:

- **Mouse - Turn**: When the pointer of the mouse is captured (see **??** for more details) it allows to move Claire around or to control the peeking function (see underneath)
- **WASD - Move**: as in many games, the player is allowed to move in 8 directions by using the above keys. This movements can be issued **also while jumping and falling**. Moreover, the torch will move accordingly.

- **Wheel - Rotate Torch**: this command is **essential to win** the game! While holding the torch, you can adjust its vertical orientation to hit the monsters right into their hearts! Remember, you can **hit them also from behind**, they are semi-transparent!

- **Space - Jump**: this command is especially useful for fleeing from monsters, since they cannot jump and do not know what to do when the player is outside their reach! Pay attention though, because this will make noise and you may jump only when you are **far enough both from objects and the monsters** themselves!

- **Shift(held) - Run**: By holding Shift the player can make Claire **move much faster**, escaping monster, jump further away and close the distance to the level exit with ease. Be careful though, as this will make Claire's **steps much louder**!

- **C - Crouch**: Crouching reduces Claire's movement **noise** to a level **not perceivable to the monsters**, as well as allowing her to be less visible and **passing under some objects**. At the same time, this allows for less control over the torch while staying in the same spot and **reduces Claire's velocity**.

- **Q - Pick Objects**: In the game there are few objects that can be picked up. This command allows Claire to take them if she is **close enough** and **no other meshes block** a direct line from her to the desired object. In case there are many pickable objects, the closest one is picked.

Other useful controls are:

- **V - Peek**: This command blocks Claire's movements (or better, **moving will cease the peek function**) but allows to peek around corners and objects without being seen by monsters. Better stay out of sight (and lock the torch over a monster heart) while you study your exit strategy!

- **E - Drop Objects**: At the current time, the only droppable object is the torch. Doing so allows for *certain kind of tricks*, like blocking monsters permanently while Claire moves around (requires some skill though!)

- **R - Rotate Object**: This command allows Claire to manipulate the torch while it is on the ground. In this way, one can invent some cool exit strategies!

- **K - Kick Object**: You can kick the torch while on the ground. It does not have any use if not to have fun (or vent anger). Originally, though, *it had some game design purposes.*

- **P - Pause the Game**: Need to go to the bathroom, take some air or just review the game controls? Press P or Esc, and you'll be served!

2.3 Level Guides

This section **contains spoilers** of the game, so be careful!

2.3.1 Level 1 - Bedroom

The Bedroom level allows the player to explore all the game control possibilities.
Still, those two monsters on the other side of the room can be tricky, so, here's a possible strategy to defeat them!
First, grab the torch on the floor.
Now crouch and go under the desk (do not go too far!). At that point, the monsters might see you but you can block them with your torch! Remember, you can use the mouse wheel to adjust the torch orientation and you can hit the monsters from behind!
Even if you blocked only one monster, you can exit from under the desk, block the monsters again and then run for the door!
Yet, there are more ways to solve this level! Why not trying to go over that wardrobe?

2.3.2 Level 2 - Hallway

The easiest way to solve this level is to illuminate the mirror on the wall without windows.
By doing this, you will block the first monster and also any approaching monsters that passes in front of it.
Now do not run, advance and sometimes light the mirror to keep blocking the first monster.
When you are between the two chests of drawers (doesn't matter if there is the monster), use the peeking function (you can use it while Claire still lights the mirror) to see where the footstool is.
After gathering your forces, run towards it, jump on it and then on the chest of drawers.
From there, light the monsters if you can and jump on the bookshelf, where you'll be safe from monsters.
Now you can light them easily again.
After that, get on the ground, crouch and pass under the desk. Well done!

2.3.3 Level 3 - Living Room

The living room is actually a simple level if you know what to do!
First, go in front of the door BUT NOT TOO CLOSE, or you'll open it.
Now, run on the place (yes, you can), and attract the closest monsters (it may even attract all of them).
At that point, behind the glass, illuminate their hearts (do not confuse the light on the opaque glass for the one that passes trough).

Now you have to open the door (wait a second), run through, and open the door on the left.
You'll find yourself in the kitchen, and it's time to jump on the furniture, and then on the fridge.
From here, you can run again on the place and attract the monsters. Again, light them.
Once you have blocked them, jump near the keys and grab one of them, but be quick! Before the monsters activate again, bolt through the living room, light the last monster if it stands in your way, jump on the chairs and hit the exit door!
Congratulations, you won!

3 Libraries and models

3.1 Libraries

In our project, we utilized two essential libraries to enhance our development process. Firstly, we employed **Babylon.js**, a powerful JavaScript framework, to handle the rendering and interactive aspects of our 3D environment.

Additionally, we integrated **Recast**, a navigation mesh generation and pathfinding library, which greatly improved our game's artificial intelligence and character movement. By leveraging Recast, we were able to efficiently calculate paths for characters within the game world, contributing to a more dynamic and engaging player experience.

We have incorporated the Babylon.js library's **Loaders** module to allow the loading of 3D assets and models. Furthermore, we have integrated the **Earcut** library to facilitate the generation of polygons required for constructing the house.

3.2 Monster and Girl Models

To create our models, we started by downloading a *hierarchical* model from *turbosquid*. Subsequently, we imported it into Blender and performed some remodeling. In particular, we added a heart to the monster and modified the skeleton for a better animation. In addition to adding the heart as a mesh, we also incorporated an animation (in Javascript) that changes the heart's color when it is illuminated and remains colored as long as the monster is immobilized, allowing the player to understand the monster's status.

Figure 3: Starting model

`https://www.turbosquid.com/it/3d-models/3d-room-1456414`

Figure 4: Modified model on Blender

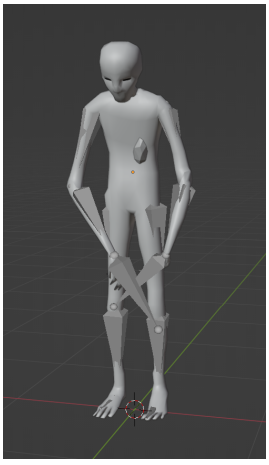
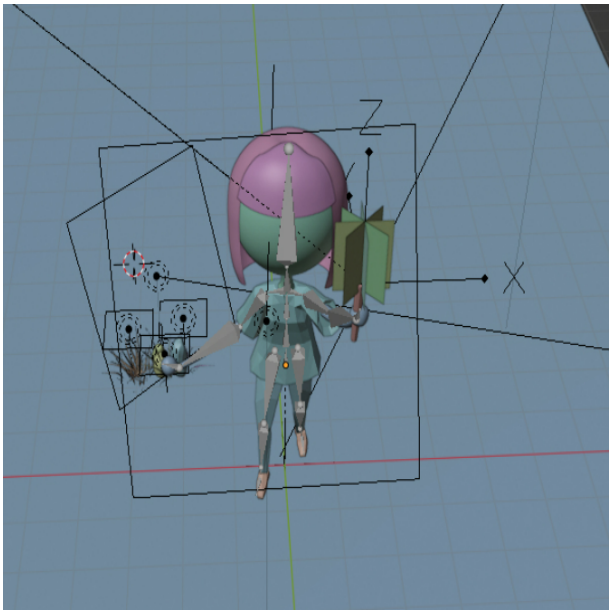
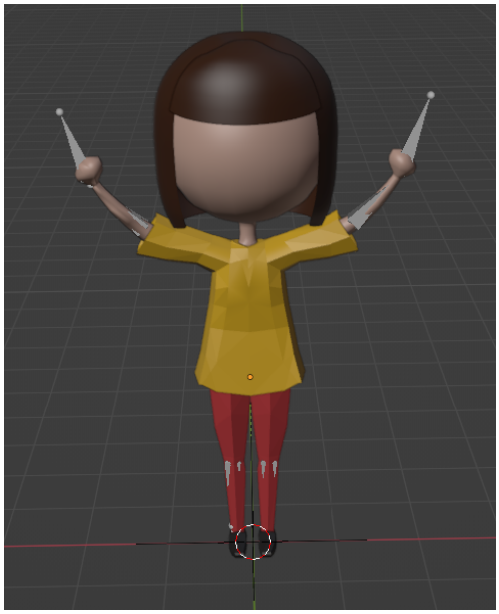


Figure 5: Starting model



https://www.turbosquid.com/it/3d-models/girl-cave-3d-model-1486912

Figure 6: Modified model on Blender



3.3 Furniture Models

To create the furniture, we imported some online models, customizing them according to our needs and preferences. Additionally we designed some models from scratch using Blender.

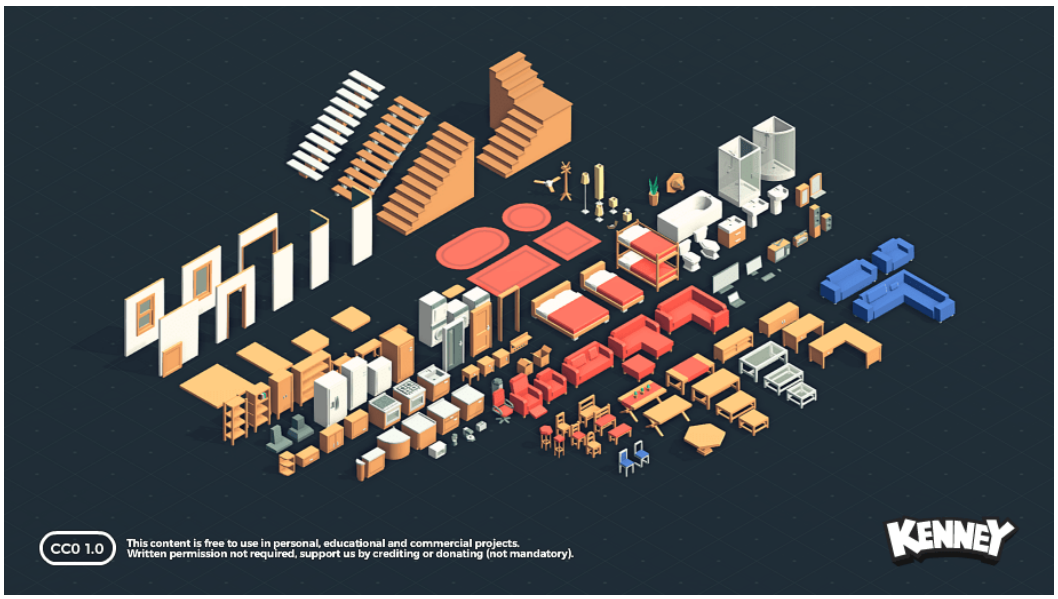


Figure 7: Link to the imported models: https://kenney.nl/assets/furniture-kit

4 Animations

Since we couldn't use any pre-made animations, we decided to use a direct Javascript approach.

4.1 Skeletons

Even if Babylon actually has a Skeleton class to animate, when importing a .glb file any eventual skeleton is not imported as an instance of that class, but rather as a complex hierarchy of Transform Nodes.
This allows to move the mesh basically as a Skeleton, but with several limitations, such as not having a simple clone method for multiple model instances.
This brought us to having to create several instances of the same monster model in Blender but with different bone-node names, so that we could retrieve them in Babylon with the getTransformNodeByName method and reconstruct each skeleton in a dictionary.
If we cloned the same model, in fact, we could move them separately but could only animate them as a single entity.

```
this.nMonst += 1;
var skeleton = {};
skeleton["bc"] = scene.getTransformNodeByName("M" + this.nMonst + " Bacino");
skeleton["pt"] = scene.getTransformNodeByName("M" + this.nMonst + " Petto");
skeleton["cl"] = scene.getTransformNodeByName("M" + this.nMonst + " Collo");
skeleton["ts"] = scene.getTransformNodeByName("M" + this.nMonst + " Testa");
skeleton["clSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Clavicola Sx");
skeleton["clDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Clavicola Dx");
skeleton["brSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Braccio Sx");
skeleton["brDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Braccio Dx");
skeleton["avSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Avambraccio Sx");
skeleton["avDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Avambraccio Dx");
skeleton["mnSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Mano Sx");
skeleton["mnDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Mano Dx");
skeleton["anSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Anca Sx");
skeleton["anDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Anca Dx");
skeleton["csSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Coscia Sx");
skeleton["csDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Coscia Dx");
skeleton["gmSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Gamba Sx");
skeleton["gmDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Gamba Dx");
skeleton["tlSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Tallone Sx");
skeleton["tlDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Tallone Dx");
skeleton["pdSx"] = scene.getTransformNodeByName("M" + this.nMonst + " Piede Sx");
skeleton["pdDx"] = scene.getTransformNodeByName("M" + this.nMonst + " Piede Dx");
```

Figure 8: Skeleton Structure

The monster and Claire model have a really similar structure to help animation.

4.2 Animation Structure

We decided to opt for a rather simple but effective animation mechanism.
Based on the behaviour of the player/monster, some global or monster specific variable would be modified. In this way, when at each render the animation function would be called, a different set of key frames will be chosen.

```
var animationManager = function(){
    switch(moveStatus){
        case 0: player.resetPose(); break;
        case "rest": player.restAnimation(); break;
        case "move": player.moveAnimation(); break;
        case "run": player.runAnimation(); break;
        case "jump": player.jumpAnimation(); break;
    }
};
```

Figure 9: Player animation function

```
restAnimation(){
    switch(standingPosition){
        case "stand": this.standAnimation(); break;
        case "crouch": this.crouchStillAnimation(); break;
    }

}
```

Figure 10: Player switch animation based on standing position

```
animateMonster(monster){

    switch(monster.behav){
        case "Stay": this.stayAnimation(monster); monster.hands.position.y = eyeHeight/3; break;
        case "Happy": this.happyAnimation(monster); monster.hands.position.y = eyeHeight/3; break;
        case "Roaming": this.moveAnimation(monster); monster.hands.position.y = eyeHeight/3; break;
        case "Searching": this.searchAnimation(monster); monster.hands.position.y = eyeHeight/3; break;
        case "Grabbing": this.grabAnimation(monster); break;
    }

}
```

Figure 11: Monster animation function

The key frames have a structure similar to the skeleton one, plus the the ”i” key, which tells the moment in which the animation must correspond to the key frame. Some key frames have also the ”li” key, that stands for ”last index”. In this way, the useFrame function knows when an animation has to start again and can use the first frame also as the last one.

4.2.1 Player details

All the player's key frames have the same structure, but each animation function has a version for when Claire holds the torch and when she doesn't.

```
crouchStillAnimation(){
    var keyFrames = [];

    if(this.torch == null){

        keyFrames[0] = {
            "i" : 0,
            "li" : 384,
            "bc" : new BABYLON.Vector3(-1.61, -3.14, 0),
            "pt" : new BABYLON.Vector3(0.56, -0.02, 0),
            "cl" : new BABYLON.Vector3(0.02, 0, 0.01),
            "ts" : new BABYLON.Vector3(0.42, 0, -0.01),
            "clSx" : new BABYLON.Vector3(-0.85, -0.03, 1.55),
            "clDx" : new BABYLON.Vector3(-1.45, 0.02, -1.59),
            "brSx" : new BABYLON.Vector3(1.38, 2.95, 0.17),
            "brDx" : new BABYLON.Vector3(1.41, -2.46, -0.17),
            "avSx" : new BABYLON.Vector3(0.66, -2.34, -0.51),
            "avDx" : new BABYLON.Vector3(0.66, 2.39, 0.45),
            "mnSx" : new BABYLON.Vector3(0.4, 1.53, 0.05),
            "mnDx" : new BABYLON.Vector3(0.46, -1.47, -0.02),
            "csSx" : new BABYLON.Vector3(1.31, 0.14, 3.13),
            "csDx" : new BABYLON.Vector3(1.27, -0.09, -3.14),
            "gmSx" : new BABYLON.Vector3(1.38, 0.1, 0.03),
            "gmDx" : new BABYLON.Vector3(1.55, 0, 0.02),
            "tlSx" : new BABYLON.Vector3(0.16, 0.01, -0.03),
            "tlDx" : new BABYLON.Vector3(-0.43, 0.01, -0.03),
            "pdSx" : new BABYLON.Vector3(-1.1, 0, 0),
            "pdDx" : new BABYLON.Vector3(-1.1, 0, 0)
        };
```

Figure 12: Part of crouch still animation

When Claire holds the torch, as stated also in other sections of this document, the torch is regularly stabilized based on her standing status and movement mode outside of the animation position.
This is needed because we want to re-position the torch only once when the standing or the movement mode changes.

4.2.2 Monsters details

The most particular monster animation detail is the grabbing function.
Here, in fact, we have the lower part of the skeleton that always moves the same, while the upper part animation must change depending on the player height.
Hence, we do a first interpolation before calling the actual useFrame function, so that we generate correct key frames and position the hands where they should be.

```
grabAnimation(monster){
    var keyFrames = [];

    keyFrames[0] = {
        "i" : 0,
        "li" : 96,
        "anSx" : new BABYLON.Vector3(-1.1, -1.69, -0.33),
        "anDx" : new BABYLON.Vector3(-1.54, -1.5, -3),
        "csSx" : new BABYLON.Vector3(1.34, -0.16, -3.14),
        "csDx" : new BABYLON.Vector3(1.47, -0.03, 3.14),
        "gmSx" : new BABYLON.Vector3(0.11, 1.15, 0.09),
        "gmDx" : new BABYLON.Vector3(0.12, -1.37, 0),
        "tlSx" : new BABYLON.Vector3(1.18, 2.76, -0.3),
        "tlDx" : new BABYLON.Vector3(1.41, 2.55, -0.64),
        "pdSx" : new BABYLON.Vector3(0.51, 0.02, -0.11),
        "pdDx" : new BABYLON.Vector3(0.2, 0.08, -0.19)
    };
    keyFrames[1] = {
        "i" : 48,
        "anSx" : new BABYLON.Vector3(-1.1, -1.69, -0.33),
        "anDx" : new BABYLON.Vector3(-1.54, -1.5, -3),
        "csSx" : new BABYLON.Vector3(1.3, -0.01, -3.14),
        "csDx" : new BABYLON.Vector3(1.62, 0.24, 3.14),
        "gmSx" : new BABYLON.Vector3(0.11, 1.15, 0.09),
        "gmDx" : new BABYLON.Vector3(0.12, -1.37, 0),
        "tlSx" : new BABYLON.Vector3(1.18, 2.76, -0.3),
        "tlDx" : new BABYLON.Vector3(1.41, 2.55, -0.64),
        "pdSx" : new BABYLON.Vector3(0.51, 0.02, -0.11),
        "pdDx" : new BABYLON.Vector3(0.2, 0.08, -0.19)
    };
```

(a) Fixed lower part movement

```
var lowFrames = {
    "bc" : new BABYLON.Vector3(0.05, 0.01, 0.14),
    "pt" : new BABYLON.Vector3(-0.06, -1.6, -0.3),
    "cl" : new BABYLON.Vector3(-1.01, -1.47, -0.06),
    "ts" : new BABYLON.Vector3(0.68, -0.47, -0.3),
    "clSx" : new BABYLON.Vector3(-2.5, -2.51, 1.99),
    "clDx" : new BABYLON.Vector3(-1.68, -0.63, -2.11),
    "brSx" : new BABYLON.Vector3(-1.05, -0.84, -0.46),
    "brDx" : new BABYLON.Vector3(-0.93, 1.12, 0),
    "avSx" : new BABYLON.Vector3(0.05, 2.34, -0.19),
    "avDx" : new BABYLON.Vector3(0.33, -1.47, 0.72),
    "mnSx" : new BABYLON.Vector3(0.17, -1.1, -0.58),
    "mnDx" : new BABYLON.Vector3(-0.18, -0.54, 0.41)
}

var mediumFrames = {
    "bc" : new BABYLON.Vector3(0.05, 0.01, 0.14),
    "pt" : new BABYLON.Vector3(-0.06, -1.6, -0.3),
    "cl" : new BABYLON.Vector3(-1.01, -1.47, -0.06),
    "ts" : new BABYLON.Vector3(0.68, -0.47, -0.3),
    "clSx" : new BABYLON.Vector3(-0.44, -1.92, 1.99),
    "clDx" : new BABYLON.Vector3(-0.23, -0.95, -2.11),
    "brSx" : new BABYLON.Vector3(-1.05, -0.56, -0.46),
    "brDx" : new BABYLON.Vector3(-0.84, 0.98, 0),
    "avSx" : new BABYLON.Vector3(1.26, 1.5, -0.19),
    "avDx" : new BABYLON.Vector3(1.9, -1.8, 0.72),
    "mnSx" : new BABYLON.Vector3(-0.8, -0.79, -0.58),
    "mnDx" : new BABYLON.Vector3(0.89, -0.78, 0.41)
}
```

(b) Changing upper part frames

```
var yHeight = this.player.getTorsoPosition().y;
monster.hands.position.y = yHeight + 0.75;

if (yHeight < 2){
    importance = yHeight/2;

    for (const k of Object.keys(lowFrames)) {
        var v = new BABYLON.Vector3(lowFrames[k].x * (1 - importance) + mediumFrames[k].x * (importance),
        lowFrames[k].y * (1 - importance) + mediumFrames[k].y * (importance),
        lowFrames[k].z * (1 - importance) + mediumFrames[k].z * (importance));

        keyFrames[0][k] = v;
        keyFrames[1][k] = v;
    }
} else if (2 > yHeight < 4){
    importance = yHeight/4;

    for (const k of Object.keys(mediumFrames)) {
        var v = new BABYLON.Vector3(mediumFrames[k].x * (1 - importance) + highFrames[k].x * (importance),
        mediumFrames[k].y * (1 - importance) + highFrames[k].y * (importance),
        mediumFrames[k].z * (1 - importance) + highFrames[k].z * (importance));

        keyFrames[0][k] = v;
        keyFrames[1][k] = v;
    }
} else {

    for (const k of Object.keys(highFrames)) {
        keyFrames[0][k] = highFrames[k];
        keyFrames[1][k] = highFrames[k];
    }
}

this.useMonsterFrame(monster, keyFrames);
```

(c) Interpolating upper part

Figure 13: Grab Animation sample

We were actually limited in the monster model animation due to distortions in the mesh after raising the arms too high.

4.3 UseFrame function

After choosing the right key frames, the useFrame function is called.
This function slightly differs between the player and monsters, but the functionality is basically the same.

```
//reset
if(this.#frame >= keyFrames[0]["li"]){
    this.#frame = 0;
}

for (var i = 0; i < keyFrames.length; i++){
    if(keyFrames[i+1] == null){
        kf = i;
        last = true;
        break;
    }else{
        if(this.#frame >= keyFrames[i]["i"] && this.#frame < keyFrames[i+1]["i"]){
            kf = i;
            last = false;
            break;
        }
    }
}

if(!last){
    firstKf = keyFrames[kf];
    secondKf = keyFrames[kf+1];
    var indexDifference = (firstKf["i"] - secondKf["i"]);
    var inBetweenIndex = this.#frame - firstKf["i"];
}else{
    firstKf = keyFrames[kf];
    secondKf = keyFrames[0];
    var indexDifference = (firstKf["i"] - secondKf["li"]);
    var inBetweenIndex = this.#frame - firstKf["i"];
}
importance = (inBetweenIndex)/indexDifference;
importance = Math.abs(importance);

var addRot = 0;
```

(a) First part of Player useFrame function

At that point, based on the current frame (and eventually other factors like player movement direction or player height for the grabbing animation of monsters) the two relevant key frames are interpolated, leading to an actual transformation on the nodes.

```
var addRot = 0;
if(document.getElementById("sliding").innerHTML == "true"){
    var dirRot = document.getElementById("dirRot").innerHTML;
    switch (dirRot){
        case "avDx": addRot = -Math.PI/4; break;
        case "dtDx": addRot = 5*Math.PI/4; break;
        case "avSx": addRot = Math.PI/4; break;
        case "dtSx": addRot = -5*Math.PI/4; break;
        case "Dx": addRot = -Math.PI/2; break;
        case "Sx": addRot = Math.PI/2; break;
        case "dt": addRot = Math.PI; break;
    }
    document.getElementById("sliding").innerHTML = "false";
}

//interpolate frame
for (let k in firstKf){
    if(k == "i" || k == "li"){
        continue;
    }else if(k == "bc"){
        this.#skeleton[k].rotation = new BABYLON.Vector3(firstKf[k].x * (1 - importance) + secondKf[k].x * (importance),
        firstKf[k].y * (1 - importance) + secondKf[k].y * (importance) + addRot,
        firstKf[k].z * (1 - importance) + secondKf[k].z * (importance));
    }else{
        this.#skeleton[k].rotation = new BABYLON.Vector3(firstKf[k].x * (1 - importance) + secondKf[k].x * (importance),
        firstKf[k].y * (1 - importance) + secondKf[k].y * (importance),
        firstKf[k].z * (1 - importance) + secondKf[k].z * (importance));
    }
}
this.#frame +=1;
```

(a) Second part of Player useFrame function

Finally, the transformation of the nodes is automatically transferred onto the interested meshes.

5 Environment

5.1 Scene creation

When the game starts, the game's scene is initialized using a function that we called *basicScene*. Inside this function we instantiate the scene object, and we set the camera and ambient light.

This function also creates the objects *player* and *torch*, importing their GLB models. It also imports the monster's models, setting them as invisible. This speeds up the process of passing from one room to the other.

The ambient light is a hemispheric light provided by Babylon. Its intensity is very low, and its color is blue, because the game is set at night. We also added a skybox, representing the woods at night.

5.2 Camera

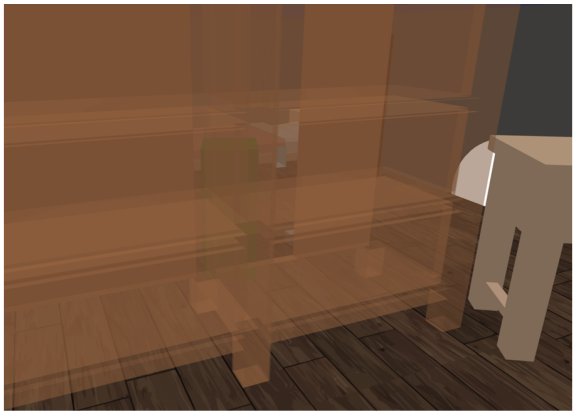
To make the camera follow the player, we used Babylon's *FollowCamera* class. We were supposed to set as target of the camera the player's mesh. But the player's mesh has its origin at the base of the model, and we wanted the camera to follow the head. To solve this problem we created an abstract mesh, *cameraTarget*, we set the player mesh as its parent, and incremented its y coordinate by a percentage of the player's height. Then we set *cameraTarget* as the locked target of the camera.

5.3 Keeping the player visible

Since the camera is always behind the player, and the player can move around, it could happen that a piece of furniture or a wall gets between the player and the camera. In order to keep the player visible at all times, inside *basicScene* we registered a function that gets executed before each render. This function uses Babylon's raycasting to cast two rays toward the camera: one starting from the player's feet, one from the player's head.

All the mesh intersected by these two rays, plus all their child meshes, get their *visibility* changed. We distinguished two cases: the walls (and their children, i.e. the objects attached to a wall) get assigned a visibility equal to 0. All the other meshes get a visibility equal to 0.2.
We did this because we still want to be able to see if there is an obstacle, but letting the player see half-transparent walls would be superfluous and worsen the gameplay.

To make the player even more visible, we also edited every furniture model in order to remove all the internal faces which were not necessary. We also exploited Babylon's backface culling, to make the furniture even more transparent. These changes made a huge difference, that can be seen in the two comparison pictures below.



Unmodified closet model. The player (a green box) is behind the closet, and can barely be seen.

The player seen from the same angle, after removing the internal faces of the closet.

6 Player

To add a player to the game, we defined the class *Player*. The constructor of this class takes as one of the inputs an imported GLB model, which becomes the mesh associated to the player.
To manage the movements of the player, we kept track of two variables: *forward_direction* and *right_direction*, to know toward which direction the player is turned.

6.1 Rotating the player

In order to rotate the player, and thus the camera, we used the *Pointer Lock API*. As soon as the game starts, the pointer lock is requested on the canvas (and is automatically released and requested every time the game is paused and unpaused).
When the pointer is locked and the mouse is moved, the player's *forward_direction* is rotated using Babylon's *rotateByQuaternionToRef* function. The mesh is rotated as well by an equal quantity.

6.2 Collisions

We didn't use a physics engine, because it would add more features than needed, making the project too heavy. Instead, we used Babylon's raycasting to implement the player's collisions.
In order to do it, we defined three properties of the object player: *height*, *width*, *depth*, which determine how big is the player's "collision box".
Using these properties, we defined the function *checkCollisions* that casts 9 rays starting from 9 different positions around the center of the player, as can be seen in the picture on the left. The function returns several values used to determine whether the rays have intersected a mesh different from the player's, and thus if a collision has occurred.
The same function is used for both horizontal and vertical collisions, by simply changing its parameters.

6.3 Movements

All the movements of the player are managed inside a function called *step*. This function is initially called when *player.start()* is executed, and then, if the game is not paused, it is continuously called by Javascript's *requestAnimationFrame* function.

The player has three types of movements: walking, falling, jumping. Falling and jumping obviously exclude each other, but the player can still move horizontally while it jumps or falls.

1. **Walking:** It is possible to walk (or run) in 8 possible directions: forward, backward, left, right, and the four diagonal directions. When one of the *WASD* keys is pressed, first of all we check whether in the corresponding direction there is an obstacle. This is done through *checkCollisions*, casting rays whose length depends on the player's speed. If there aren't obstacles, the player is moved by adding to the position of its mesh a vector scaled by the speed of the player.
   If there is an obstacle, but the collision is only detected by the rays on one side (left or right), the movement isn't blocked! Instead, the player gets turned slightly toward the opposite side. In this way, it is possible to walk near an obstacle with ease, without having to "align" perfectly the player.

2. **Falling:** Inside the step function, we also check whether the player is falling. To do it, we defined a function called *coordinateObjectBelow*, which casts a ray from the player towards the floor. The function returns the y coordinate of the collision point with the first object it finds.
   In this way we know that the player must fall if the difference between its y coordinate and the coordinate of the object below is bigger than a quantity we called *heightOffset*.
   The player has a falling speed which, while falling, gets increased until it reaches a maximum value. This adds gravity to the movements.
   If the player is supposed to fall, we check whether there is a collision beneath it using again *checkCollisions*, casting 9 vertical rays whose length depends on the current falling speed. Then we distinguish three cases.
   In the first case there isn't a collision, and the player is translated vertically of a value equal to the falling speed.
   If instead there is a collision, but the distance from the object below is still bigger than *heightOffset*, it means that the player has to fall for a smaller distance than the one given by the falling speed. In this case the player is translated vertically of a value equal to the distance from the object below.
   In the third case, there is a collision with only 1 or 2 of the 9 rays. It means the player is almost falling from an obstacle. In this case, the player is moved horizontally toward the direction where it's supposed to fall. In this way we avoid cases where the player appears to be stuck mid-air because only one corner of its "collision box" is on the obstacle.

3. **Jumping:** To handle the jumping, we defined a value called *jumpHeight*, which determines how high the player is able to jump.
   We first of all check that there isn't an obstacle over the player. To do it we use

checkCollisions, and cast some rays upward. The length of these rays is equal to *jumpHeight*.
If there aren't collisions, the player is moved upward by a value equal to the jumping speed. This value is decreased until it reaches a minimum value. Again, this gives the impression of gravity.
When the *jumpHeight* is reached, we set the flag *jumping* equal to false, so that the jump ends and the player can start falling instead.

6.4 Walking variants

Besides walking normally, the player is also able to run and crouch.

1. **Crouching:** Crouching changes the value of the player's *height*. In this way, the "collision box" appears to be shorter, and the player is allowed to pass under objects like tables. The use of the *height* variable allows to manage crouching without having to change the way we check the collisions.
   When crouching is active the speed of the player is also made slower, by the use of a variable called *speedFactor*, which multiplies the speed of the player. This variable is set as 0.7 by the function *setSpeedCrouching*.
   If the player tries to jump or run while crouching, it automatically stands up. If it can't stand up, because there is a collision above it, it doesn't jump or run.

2. **Running:** To handle running, we simply change the variable *speedFactor* to 2.5 through the function *setSpeedRunning*.

6.5 Interaction with objects

Instead of creating ad-hoc solutions for each pickable object of the game (originally they were supposed to be more) we decided to create a sort of *Interface* named **Pickable Object**.
In this way, inside the Player class, we could write a single section of code that deals with *potentially infinite* pickable objects.
Anyway, the **grabObject** function checks if there is a pickable object in the vicinity that is not blocked by other meshes.
If there is such pickable object, it is added to the **Owned Objects** list and is not considered an obstacle for the cast ray function of other grab actions.
The list of pickable objects can dynamically change even during a level.
Finally, there is the interaction with the doors and the mirror. These are not pickable objects and they will be both discussed in other sections.

6.5.1 Torch

The torch is clearly the most important Pickable Object (and, in fact, the **only droppable one**).
When the torch is dropped, its parent is set to null, it is released in front of the player and many parameters are reset.
Once the torch is picked up, its parent becomes the right hand bone of the kid skeleton, hence, if *we animate the arm, the torch is moved too.*
Since the player can go up on the furniture, we *allowed the torch to rotate up and down.* This rotation, in fact, is not fixed and it is recalculated each time Claire rotates since the axis reference changes.
The sub-direction of the *spotlight changes automatically* with respect to the mesh rotations.
Finally, if the *torch is on the ground*, it can be still rotated on Y-axis and **kicked** (also falling from furniture). This functions are not actually needed to solve the game (in fact they are relics of the original game concept for four-people development), but they show that we could emulate physics and deal with objects separated from the player.

6.5.2 Key

The key interaction is fairly simple: if the key mesh is close enough, the key is added to the player inventory and the mesh is disposed.
If the player tries to approach the exit door with the key, they win, otherwise, the door does not open.

6.6 Peeking

The peeking function is meant to simulate the action of, well, peeking over or on the side of an object to *see what's behind without being seen.*
Hence, the peeking function **detaches** the mouse movement from the **player rotation** and just moves the **camera instead** (within some limits).
If the player moves or does any other changes to its own position (such as crouching) the peeking will stop.

6.7 Interaction with doors

When the player is walking, the central ray cast by the function *checkCollisions* has actually a double function.
In fact, besides checking if there are collisions, this ray also checks whether it hits an object whose name contains the string "door". If it hits such an object, the function *hitDoor* is called, and the name of the door is passed to it. This function executes a different piece of code for each of the several doors of the game.

The different behaviors associated to the doors will be better discussed later in the report.

7 Monster

To create the monsters, we opted to leverage Babylon.js' Crowd Agents feature. This functionality enabled us to generate a group of monsters, referred to as a "crowd", capable of assessing their surroundings during movement. In essence, these monsters can cooperate and interact with the surrounding world, enhancing the game play experience significantly. Thanks to this capability, the monsters can intelligently respond to player actions and environmental stimuli. We decided to add each monster in a list "monsterList" in order to save and keep track of every monster's status.

7.1 Navigation Mesh

In managing the movement of monsters within the game environment, we harnessed a technology known as the "navigation mesh". The navigation mesh is a three-dimensional representation of the game environment, enabling characters (or monsters in our case) to intelligently plan and follow their paths. This technique ensured that our monsters move realistically.

To create our navigation mesh, we created a merged version of the game environment's meshes, known as the "static mesh". This static mesh represents accessible surfaces and passageways within the environment. In essence, it defines where monsters can navigate and where they cannot. Once this static mesh was passed to the monsters, we disposed it to ensure that the *cast ray functions worked properly.*

7.2 Behaviour

Each monster exhibits four distinct behaviors that are contingent upon the girl's actions. Specifically, at each frame, the monster evaluates whether it can capture, observe, or detect the girl's presence, leading to varying responses and actions.

1. **Capture:** The monster can successfully capture the girl only when its hands are near her (actually an Abstract Mesh is used to determine the hands position). If this occurs, the monster inflicts a loss of one life upon the girl and becomes immobilized for a duration of 5 seconds.

2. **See:** This behavior involves assessing whether the monster can visually perceive the girl. To achieve this, a two-step process is employed. Firstly, two separate Vector3 objects are created—one representing the monster's viewing direction and the other aligned with the girl's position. By measuring the cosine value, the angle of the monster's field of vision is determined. Subsequently, if the girl falls within a potential line of sight, the monster generates a ray from its own position towards the girl and inspects the environment for obstacles. If there are

no obstructions in the path, the monster proceeds towards the nearest position to the girl within the navigable space, effectively closing the distance between them.

3. **Hear:** The monster's ability to hear the girl comes into play when she produces auditory cues, such as walking or running, provided that the monster is in close proximity. When the monster detects audible cues from the girl, it initiates a response. However, due to the lack of precise knowledge regarding the girl's exact location, the monster calculates a position in close proximity to the girl's presumed location and proceeds towards it.

4. **Searching:** When the monster has no information about the girl's potential whereabouts, its default action is to conduct a search in the vicinity of its own position in an attempt to locate the girl.

At each frame, the 'Behaviour' function checks the conditions to execute each of the above behaviors. To make the monsters more realistic, we use a variable 'behav' and 'patience' to store the last behavior of the monster. Since it's possible that the monster only sees the girl for a few seconds, it's necessary that, in case it doesn't have any other clues about her, it remembers what it was doing (its behavior), where it saw or heard her (girl position), and through the 'patience' variable, we define how long it should continue in that state. All of these behaviors are active when the monster is in its normal state.
However, when the girl points her torch toward the monster's heart, the monster ceases all of its actions for a duration of 5 seconds.

8 Torch (and Mirror)

If Claire (and hence the Player class) is the protagonist of the game, nothing could be done without her co-protagonist, the **Torch class**!
As for the Pickable Objects, we opted for a more general approach to non-ambient lights and lights sources, even if in the end there is only one class for each for development reasons (see **??**).
So, first of all, let's talk about the BedSpotlight Class.

8.1 BedSpotlight Class

This class is more of a **helper class** for the Babylon's Spotlight and Shadow Generator Classes. As such, it is easy to create and dispose both the torch and the mirror light-reflections, as well all the shadows related to them.
It encapsulates them and provides some utility functions that **reduce the amount of code** to be written in other parts of the game zones.
In particular, for the Spotlight Class it is important to modulate the range, exponent and intensity to **emulate the fading light** of the torch.
For the Shadow Generator this class is useful to avoid redundant code.

8.2 Torch Class

Hence, let's talk about the Torch class, which relies on a *BedSpotlight class instance*.
This massive chunck of code follows a general approach to functions. In fact, it would be **possible** for the torch **not only to block monsters**, but maybe also to set fire to a tend, if we wanted to.
This is achieved through the **addEffect** function, which takes as arguments an *objectHitList*, an *excusedList*, a *mirrorList* and an *action*. Let's talk individually about this arguments:

- action: this is a piece of code (potentially completely unrelated from the torch code) that is executed if the torch spotlight hits some specific meshes.
- objectHitList: this is the list of meshes that need to be "hit" to activate the action.
- excusedList: this is a list of meshes that must be ignored by the light. Normally, this is the torch mesh itself, but if the torch gets picked we add the player mesh to it and, potentially other objects owned by Claire or door glasses. Not only that, if we want to make some furniture "transparent", we could.
- mirrorList: this is a list of objects which create a light-reflection effect. At the state of art, this creates only one kind of reflection, but it may be expanded if needed.

Once an effect is added, an **onRenderObserver** will be added that checks if the desired meshes are hit by the light or not.
This observer will call the **lightHit** function, which actually activates the torch functions only *one time out of six* renders. In this way, we avoid to go heavy on the computer resources.
The observer actually follows a **two step procedure**: first it uses the utility function **isInPossibileSight**, that simply checks if the line that links the torch and the object center is inside the light cone and then, if this is true, calls the actual **CastRayTorch** (or MirrorCastRay) function. This process allows to *spare the resources* on useless castRay functions.
The CastRayTorch actually uses a **simple mechanism**: leveraging on the **Babylon Ray Class**, we check that inside some range there is one or more of the target objects (which is certain since we used isInPossibileSight, that also gives us the direction of the ray in which to cast) and if there are non-excused objects in the middle. If the first object really hit is a target one, then we activate the effect.

8.2.1 Rotations and Animation

While in Claire's hand, the **torch is moved constantly** since it's attached to the hand bone. To deal with the transformations due to the animations and to keep the torch steady (and usable), we had to manually *adjust the torch rotation* and lock the right arm.

8.3 Mirror (not a class)

As stated above, the mirror light reflection effect is not due to a real class, but it is just a side effect of the lightHit function called at each render.
In fact, by avoiding another observer, we reduced the load on the beforeRender function of Babylon.
Finally, the **MirrorRayTorch** actually **works the same way** as the **CastRayTorch** function, with the only difference being that, if the target meshes are hit, then we call the **MirrorReflectLight** functions.
The latter, in turn, illuminates the objects (that we assume are mirroring surfaces) and re-propose the same pattern for the CastRayTorch (so first check ifInPossibleSight, then cast ray).
We do not actually allow for multiple light-reflections.

8.4 Shadows

If the appropriate toggle is set on, the spotlight of the torch will start to create shadows. We used several options and tricks to make shadow generation fast and properly

working, but, unfortunately (and for unknown reasons) **some times the shadows flicker**.
This happens only with the imported models of the furniture even if we manipulated all the child meshes. On the other side, this does not happen with simple box meshes. Anyway, it is important to remember that **monsters do not cast shadows for game design purposes**, since in this way light can traverse multiple monsters and block them in a line if the player is smart enough!
Finally, **the mirror doesn't cast shadows** too since it's an **emissive texture**. This choice lead us to a faster performance and the **strategic position** of the mirror prevents the player from realizing the absence of shadows!

9 House

After creating the game scene through *basicScene* (as described previously), each room of the house is created through a custom function.
We defined in fact three functions: *BedroomScene*, *HallwayScene* and *LivingroomScene*. Each of them mainly carries out these tasks:

- Adds to the scene the corresponding room's objects (ground, walls and furniture).
- Positions the player correctly inside the room.
- Creates a navigation mesh for the monsters.
- Adds the monsters to the scene.
- If necessary, creates and positions the torch.

9.1 Rooms: models, textures, materials

The creation of the room's objects, and the managing of their materials and textures, is handled inside three functions, one for each room: *createBedroom*, *createHallway*, *createLivingroom*.

9.1.1 Creating the room objects:

These three functions create and add to the scene the ground and the walls. The walls are created through the function *createWall*, that actually creates two walls: one visible and one invisible, positioned on top of the visible one. This is done to prevent the player from jumping over the walls of the house.

For each piece of furniture, these three functions add two meshes to the scene. One is an imported GLB model, fully visible. The other is a box mesh, slightly bigger than the model, and set as invisible. The invisible box mesh completely envelops the imported model on every side.

There are two reasons justifying this choice.
First of all, since the invisible box mesh can still create collisions, this ensures that the player's collisions work correctly. Otherwise, if the mesh of an imported model has a too complex shape, it wouldn't be guaranteed, .
Secondly, we needed very simple meshes that are connected to the ground in order to generate a working navigation mesh.

To sum up, the functions *createBedroom*, *createHallway* and *createLivingroom* return

two arrays: *objects* and *navMeshObjects*. The first one contains every single mesh of the room. The second one only contains the ground, the walls, and the invisible box meshes. This array is in fact used by the functions *BedroomScene*, *HallwayScene* and *LivingroomScene* to create the navigation mesh.

9.1.2 Adding textures:

While the textures of the furniture models have been added by us using Blender, there are some textures that we imported directly via code. In particular: we used Babylon's *Texture* class to assign to the ground a diffuse texture and a bump texture. We did the same, inside *createLivingroom*, to also assign a diffuse texture and a bump texture to the tiles on the wall (in the bathroom and in the kitchen).

To create the reflection in the hallway's mirror, we used Babylon's *MirrorTexture* class. Inside the function *hallwayScene*, in fact, we created a reflective texture by defining the reflection plane and the normal of the mirror.
In order for the mirror to know which objects must be reflected, the function *createHallway* actually returns a third array: *mirrorObjects*, that is used to define the mirror's render list. To improve the performances, we added to this array only those meshes that are close to the mirror.

A fourth kind of texture that we used, that was actually added using Blender but that we still think is worth mentioning, is an emissive texture, that we used to make the "hints" signs glow.

9.1.3 Modifying the materials:

At the end of each of the three functions (*createBedroom*, *createHallway*, *createLivingroom*), we iterate over all the materials present in the scene, in order to apply a couple of modifications.

The first one is setting the flag *usePhysicalLightFalloff* of every material equal to false. This is necessary to make the imported mesh receive the light from the scene.

The second modification is applied to all the materials named "glass", which, from how we set the models, are only the windows. What we do is assign a light blue emissive color to the glass material. In this way, the windows appear to be glowing.
To emphasize the glowing effect even more, we also added a glow layer.

9.2 Sliding doors

In the living room there are three sliding doors. When the player collides with them they open, allowing both the player and the monsters to pass. This means that the navigation mesh of the monsters must change after a door has been opened.

In order to do this, we didn't add the mesh of the doors to the static mesh used to generate the navigation mesh. It would be in fact impossible to update the navigation mesh without destroying and re-creating the agents crowd.
Instead, after creating the navigation mesh of the living room, we created three obstacles, corresponding to the three doors. We added them to the navigation mesh through the *RecastJSPlugin* function *addBoxObstacle*.

The opening of the doors is managed by the function *openDoor*. This function animates the door mesh using Babylon's *Animation* class. The animation simply consists of two keyframes, and makes the door mesh "slide" until it disappears inside the wall.
After the animation ends, we used a callback function to dispose the two meshes associated to the door, so that the player can pass. The callback function then uses the function *removeObstacle* to remove the corresponding obstacle, allowing the passage of the monsters.

9.3 Changing room

As mentioned earlier, when the player hits a door the function *hitDoor* is called. If the door that has been hit is a door that connects two rooms, *hitDoor* calls another function called *switchScene*.

The function *switchScene* first of all shows the loading screen. Then it pauses the player, and stops the engine render loop.
Then it disposes all the objects of the current room, also disposing their materials and textures, and it disposes the monster crowd. The torch only gets disposed if the player isn't holding it. If it gets disposed, its position and rotation are saved so that if the player walks back into the room the torch is created where the player left it.

Then the function sets up the new room, by calling one of the three functions mentioned earlier (*BedroomScene*, *HallwayScene*, *LivingroomScene*), and positions the player correctly inside the new room.
Lastly, it hides the loading screen, and starts the player and the engine render loop.

10 Optimizations

Being an FPS-dependant game, we adopted several techniques to optimize the execution of the code.

10.1 Freezing the meshes' world matrices

After creating a room, we iterate over all the objects of the room (including their children) and we freeze the world matrix of their meshes, using Babylon's *freezeWorldMatrix* function. We do this because every mesh has a world matrix that specifies its position, rotation and scaling, and this matrix is automatically evaluated at every frame.
Since almost all the objects in a room are never moved, rotated or scaled, it would be a waste of computational power.
In order to animate the sliding doors of the living room, though, we had to call the function *unfreezeWorldMatrix* on their meshes.

10.2 Not updating the meshes' bounding info

Since, again, the objects of a room are never moved, it's not necessary to continuously synchronize their bounding info. Since Babylon does it by default, we deactivated this function by setting *doNotSyncBoundingInfo* equal to *true* for every mesh.

10.3 Blocking the shaders' updates

After creating a room we also iterate over all the materials of the scene, and for each material we call the function *freeze*. This function prevents the material's shader from continuously updating, which is done by default. Since almost all materials are static, it's not necessary to update their shaders.
In order to make the monster's heart glow white, and to make the mirror work, we had to "unfreeze" their materials.

10.4 Not checking whether a mesh is pointed by the mouse

By default, every time the pointer, is moved Babylon checks all the meshes present in the scene, to see whether a mesh under the pointer has an action or event to execute. We didn't need this functionality, so we set *skipPointerMovePicking* equal to *true*.

10.5 Not clearing the color buffer

Babylon by default clears the color buffer before each render. Since we use an opaque skybox, our scene always fills completely the whole viewport, making this mechanism

useless. We deactivated it by setting the *autoClear* flag equal to *false*.

10.6 Blocking the "dirty" mechanism for the materials

Babylon continuously analyzes all the materials, to flag them as "dirty" if one of their properties gets changed. As we said before, most materials are static, and since we have a big number of materials we set the flag *blockMaterialDirtyMechanism* equal to *true*.

10.7 Other optimizations

We decided to allow, in the start menu, to choose a "lighter" version of the game. This version loads the furniture models without textures, and it skips the creation of several unnecessary objects (like the paintings and pictures on the walls).
For performance reasons, we also decided to avoid the use of realistic shadows by default. They can be turned on from the start menu.