Fall Guys: final project

Interactive Graphics

Facoltà di ingegneria dell'informazione, informatica e statistica
Engineering in Computer Science
Sapienza, Rome, Italy

Andrea Panceri 1884749, Francesco Sudoso 1808353, Christian Tedesco 2025232

Academic Year 2022/2023

Contents

List of Figures

Code snippets

1 Introduction

FallGuys is a web game done with Babylonjs. It is the ultimate test of agility, strategy, and survival. Get ready to dive into two thrilling game modes that will challenge your wits and reflexes.

1.1 Survival Game Mode

In this heart-pounding mode, you'll find yourself floating above an arrangement of spaced-out hexagonal platforms. As the initial countdown reaches the value zero, you'll plummet onto your very own platform. But be wary, as each hexagon you touch starts to blink and vanishes instantly. Beware of the water at the bottom of last platform, a single touch will make vanish the player and end the game.
Stay vigilant and keep an eye out for the special spheres hovering above each hexagonal platform. Some are your saviors, like the bubble protection sphere that, when touched, cloaks you in a shield, preventing the platform from disappearing beneath your feet. Others propels you skyward with an explosive jump. Your objective is simple: survive as much as possible, searching every time to surpass the precedent record.

1.2 1 Vs 1 Game Mode

Prepare for head-to-head competition in the 1 Vs 1 mode. As the initial countdown reaches the value zero, you and your opponent will descend onto the hexagonal platforms. With every touch, a hexagon blinks and fades away, so tread carefully to avoid falling into the water below.
Like before there are various kind of spheres.
Your objective: outlast your opponent and survive longer on these platforms. Will you have what it takes to secure victory and claim your spot as the last one standing?
.

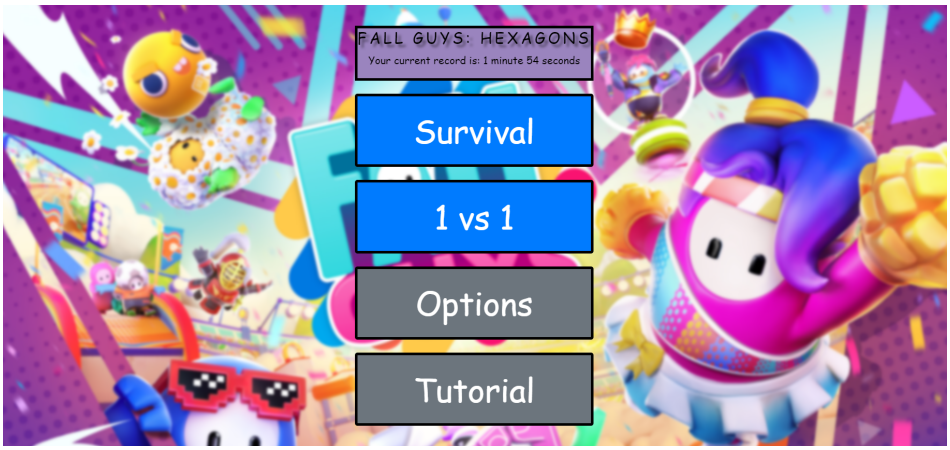


Figure 1: Homepage

1.3 Framework used

There are mainly three frameworks and libraries used in the implementation of the project:

- **Babylon.js:** an open-source, 3D graphics engine and framework for creating interactive 3D applications and games that run in web browsers. It is platform-independent

and works across different web browsers, and it also supports both desktop and mobile devices. It provides a wide range of pre-built meshes and materials that can be easily customized. Supports keyframe animations and various camera types, including the *follow camera* (used in our project).

- **Oimo.js:** a JavaScript physics engine designed for simulating 3D rigid body dynamics, collision detection, and responses in web-based applications and games. This engine allowed us to apply forces, impulses, and constraints to objects. It also includes robust collision detection algorithms that can accurately determine when objects in the 3D space collide with each other. In our project, collision detection between the player and the ground hexagons was of fundamental relevance.
- **Bootstrap:** a framework that provides a set of pre-designed HTML, CSS, and JavaScript components, as well as a responsive grid system, that makes it easier to create consistent and modern user interfaces.

2 Hierarchical model of player

The following diagram shows the structure of the hierarchical model for the player.

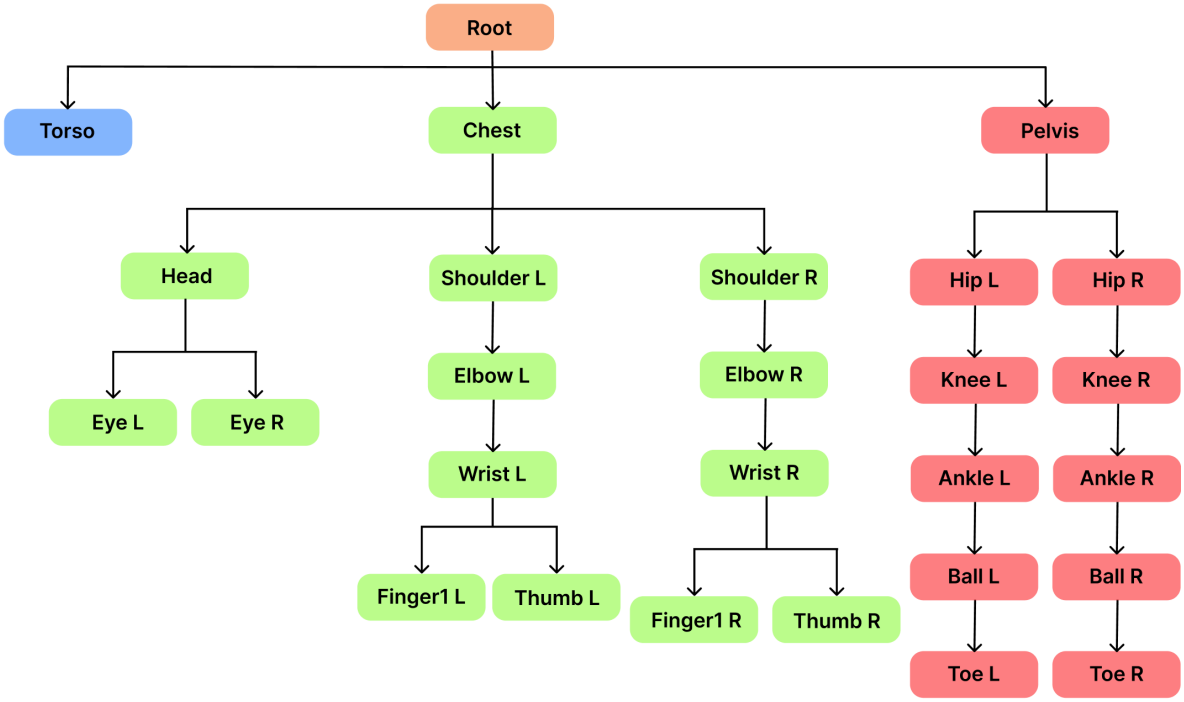


Figure 2: Hierarchical model of player

Specifically, the parent node is the **root** that has 3 children, each of which corresponds to a section of the player's body, specifically we divided the body into **torso**, **chest** and **pelvis**, as we can see from Figure 3.
The most important nodes in this model are the chest and pelvis. The chest because it contains the head and the left and right shoulder of the player (with all the related elements such as arms and hands), while the pelvis because it contains all the configuration concerning the player from the pelvis down to the toes (hips, knees, and so on).

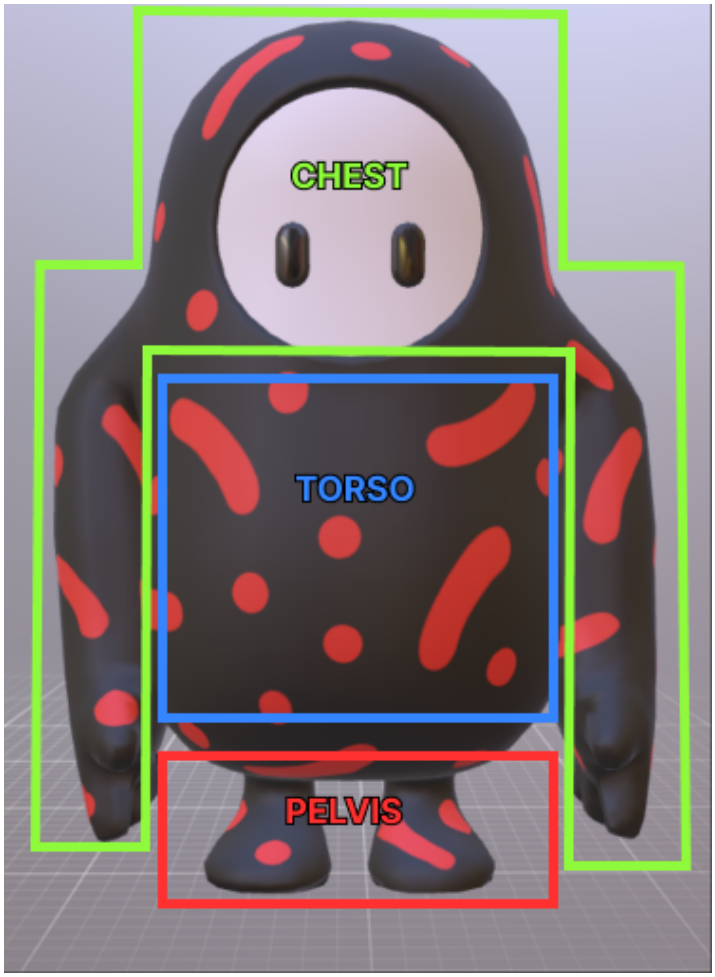


Figure 3: Model of player

3 Models of game

We have included different models in the game, using some external tools like Blender and sandbox.babylonjs.com for editing the hierarchical model and changing the textures. We are going to analyze each model.

3.1 Platform Model

The Platform Model consists of 300 independent hexagons. There isn't a complex hierarchical structure; instead, a simpler approach was taken to facilitate the addition of colliders and manage the game's dynamics. All 300 hexagons share the same root, allowing the entire platform to be translated by moving a single node. To construct the model, Blender was utilized. The process began with a single cube, to which hexagon faces were added at the top and bottom using geometry techniques. This process was repeated to create columns of adjacent hexagons and subsequently rows. Once the final platform was achieved, each individual mesh was divided. Also we have not add a particular material to the platform we change it in game.

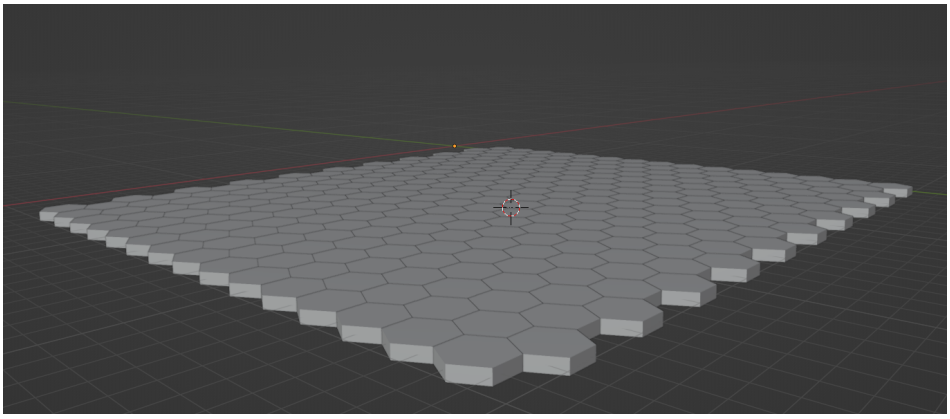


Figure 4: Platform Model

3.1.1 Hexagon Model

We have also incorporated a singular hexagon model into the design. This model serves a dual purpose: during the countdown phase, it positions the player above the platform, and in the endgame, it acts as the foundational base for the player. This hexagon is also the root of his model and like for platform there is any material.

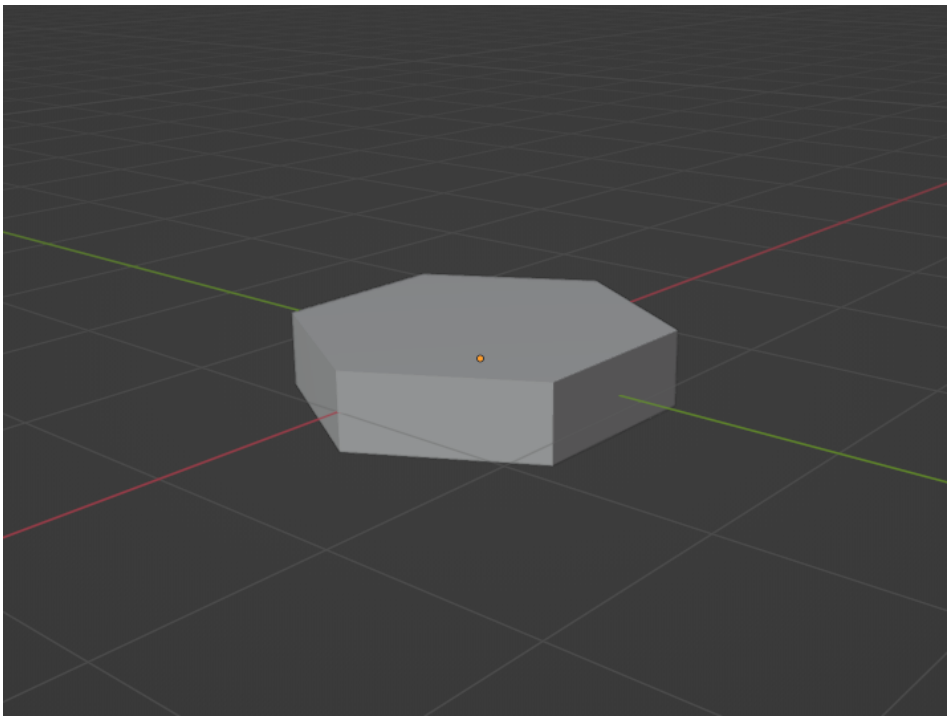


Figure 5: Hexagon Model

3.2 Bomb Jump Sphere Model

The Bomb Jump Sphere model consists of a single root node, designed to resemble a sphere with certain elements to mimic the appearance of an emergency device. While not overly complex, it effectively conveys the intended effect. To enhance its realism, various textures have been applied to achieve a metallic finish, adding depth and detail to the overall design.

This attention to detail ensures that the Bomb Jump Sphere not only serves its gameplay purpose but also contributes to the immersive experience of the game.

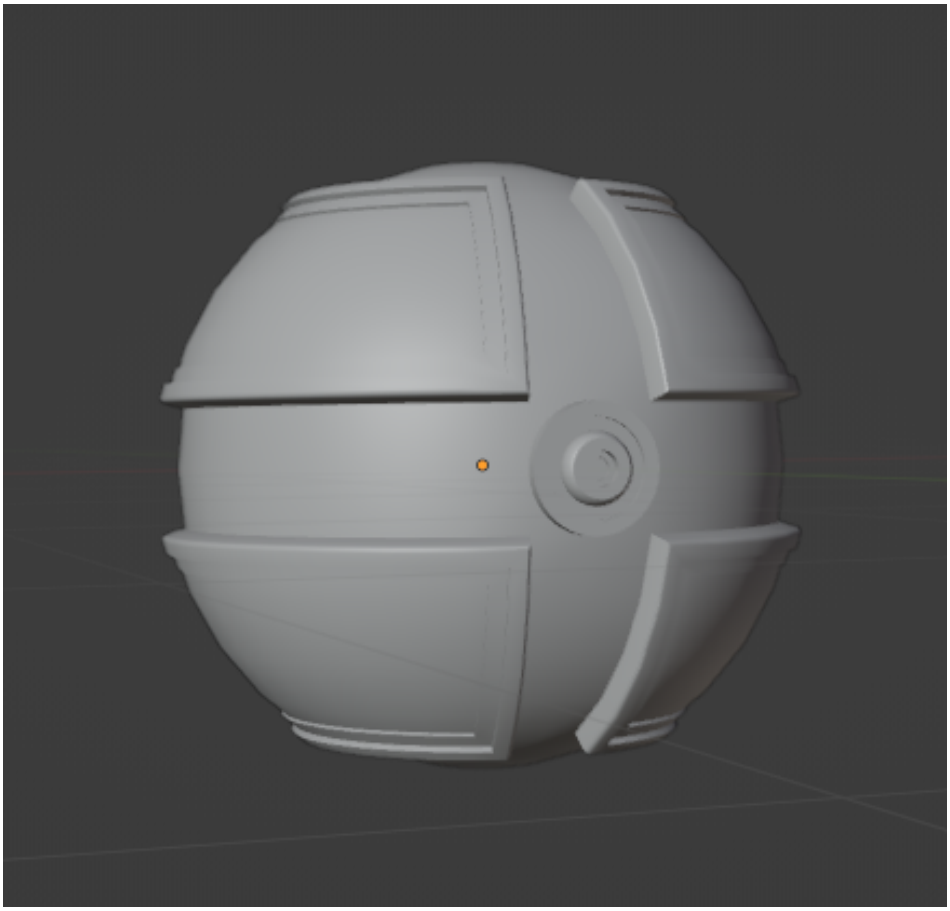


Figure 6: Bomb Jump Sphere Model

3.3 Bubble Protection Sphere Model

The Bubble Protection Sphere model is a highly intricate design, featuring a substantial 1.154.988 vertices, in stark contrast to the 3829 vertices of the Bomb Jump Sphere. This complexity is essential to achieve the desired visual effect and gameplay functionality.
At its core, the model is organized around a central root node, which serves as the foundational structure for the sphere. This root node branches out into 18 distinct sub-nodes, each responsible for housing various components of this intricate sphere. These sub-nodes collectively bring together the different parts necessary to create the complex shape and functionality of the Bubble Protection Sphere.
In terms of aesthetics, the Bubble Protection Sphere is predominantly colored in a pristine white hue. This coloration not only imparts a sense of purity but also serves as a visual cue to players, indicating the presence of a protective shield. Its complexity, combined with the distinctive white color, ensures that players can easily identify and interact with this essential game element.

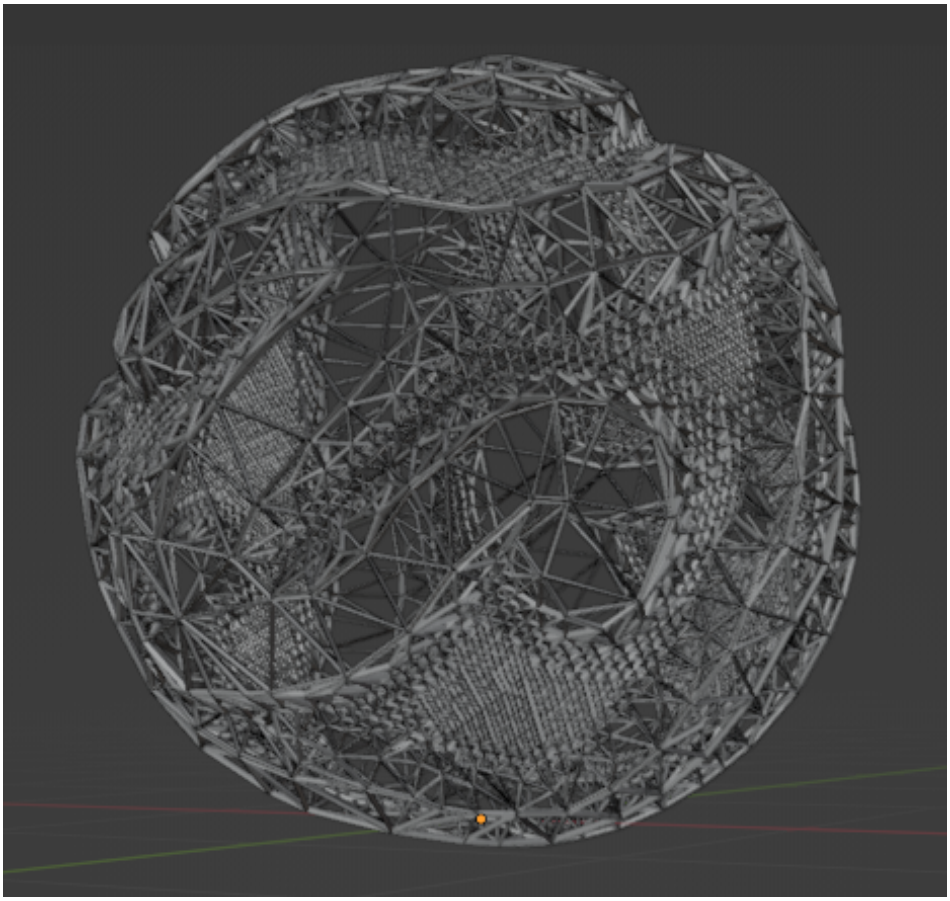


Figure 7: Bubble Protection Sphere Model

3.4 Player Model

The Player Model is indeed the centerpiece of the game, combining a complex hierarchical structure with careful attention to detail in both geometry and textures. This model, while not overly dense with vertices (2384), possesses a hierarchical structure crucial for delivering smooth and expressive animations. It plays a fundamental role in bringing the game's characters to life with a charming and cartoony aesthetic.

The Player Model boasts a well-thought-out hierarchical structure. This structure is indispensable for creating intricate animations that capture the essence of a dynamic and bouncy cartoon character. Each component of the player, from limbs to facial features, is meticulously organized within the hierarchy. This enables precise control over each part's movement, making animations fluid and lifelike.

Despite its moderate vertex count, the Player Model's geometry is designed with care. The model's shape and proportions are optimized to evoke the playful and whimsical appearance of a cartoon character. It strikes a balance between complexity and performance, ensuring that the character moves convincingly while maintaining optimal game performance.

The Player Model takes advantage of different textures applied to various parts of the character. These textures add depth, character, and vibrancy to the model, contributing to the overall cartoonish appearance.

The primary goal of the Player Model is to encapsulate the essence of a cartoon character. This includes not only the physical appearance but also the dynamic and playful animations. The hierarchical structure and careful design allow for a wide range of movements,

from running and jumping to comical gestures and reactions, all with that delightful cartoon bounce. The model of player two is equal, what is change is only the textures.

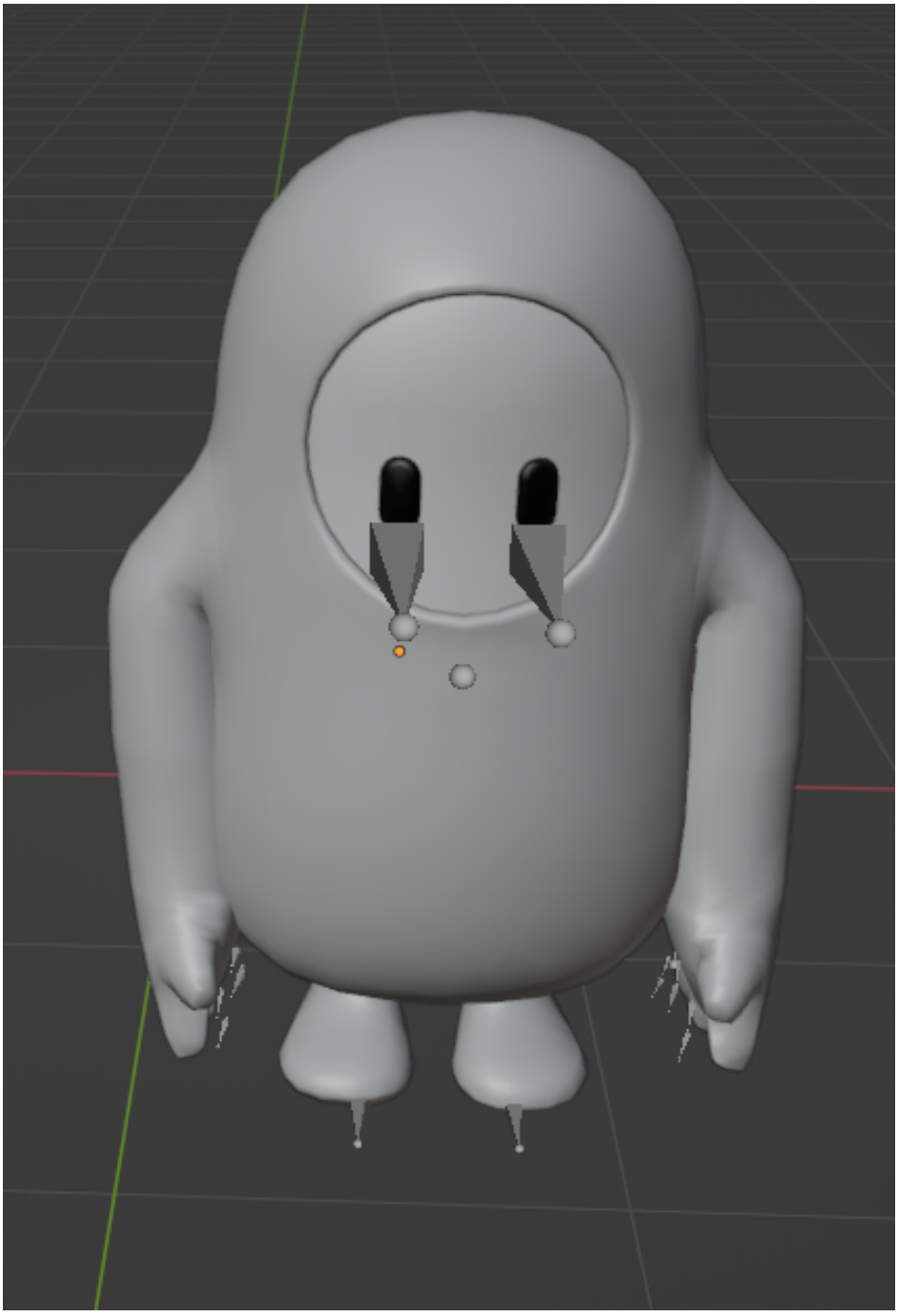


Figure 8: Player Model

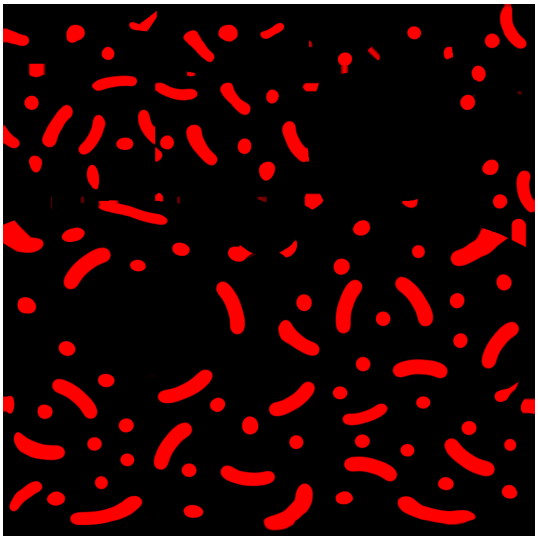
4 Textures of game

We have included different textures in the game. Using external tools we added normal map and add also metallic effects. We focus on materials for choose the correct color palette and reflection of each one. We cover also other special textures we have not seen at lesson like

the water and the sky-box.

4.1 Player One Textures

For player one we have used three different material: one for Body, one for the head and on for the eyes. We have reported in figure (a) the texture for the body, simple texture without use of a normal texture that use red a black color as base. For the eye we have choose to add a high reflection parameter and a base color of black. For the Head a simple white color. Overall the 3 materials give a cartoon effect to the player. In figure (b) we can observe the final result. We have not applied the textures and material in game but directly in the mesh.



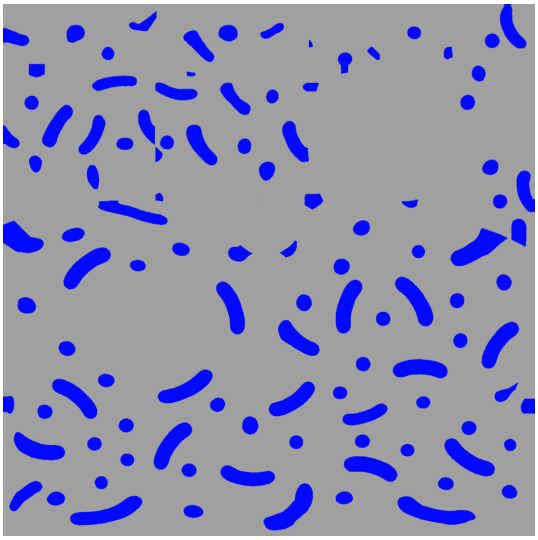
(a) Player One Texture

(b) Player One Model

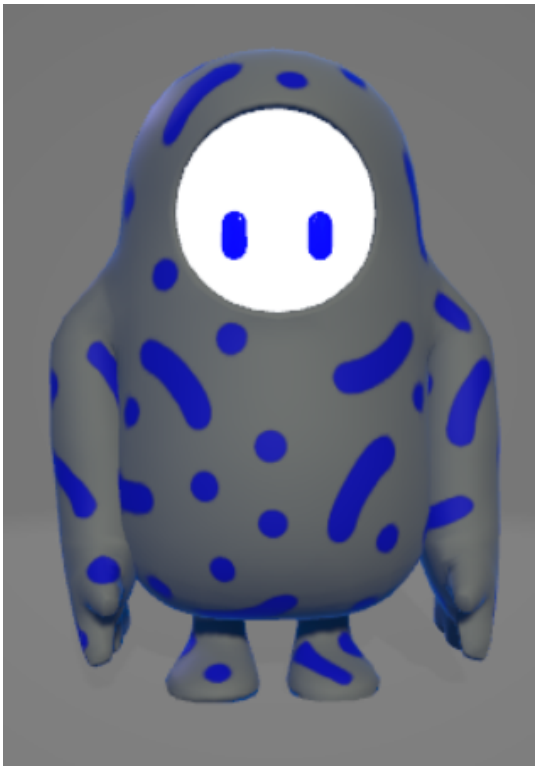
Figure 9: Player One Model with Materials

4.2 Player Two Texture

For player two we have used three different material: one for Body, one for the head and on for the eyes. We have reported in figure (a) the texture for the body that use blue and grey color as base, simple texture without use of a normal texture. For the eye we have choose to add a high reflection parameter and a base color of blue. For the Head a simple white color. Overall the 3 materials give a cartoon effect to the player. In figure (b) we can observe the final result. We have not applied the textures and material in game but directly in the mesh.



(a) Player Two Texture



(b) Player Two Model

Figure 10: Player Two Model with Materials

4.3 Bomb Jump Sphere Texture

For the Bomb jump sphere we have used a single material but different textures for reach the final goal.

Base Color Texture: The base color texture for the Bomb Jump Sphere provides the main visual appearance of the sphere. It uses a combination of colors to give the sphere its overall look.

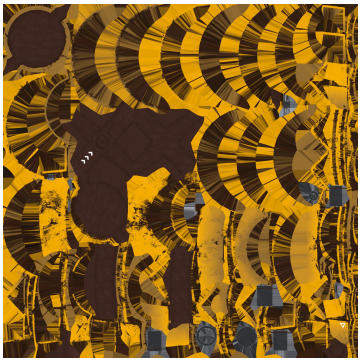
Emissive Texture: The emissive texture adds an emissive or glowing effect to parts of the sphere. This can be used to make certain parts of the sphere appear to emit light or glow in the game.

Metallic Roughness Texture: The metallic roughness texture is used to control the metallic and roughness properties of the Bomb Jump Sphere's surface. It determines how light interacts with the sphere, giving it a metallic shine and controlling how rough or smooth its surface appears.

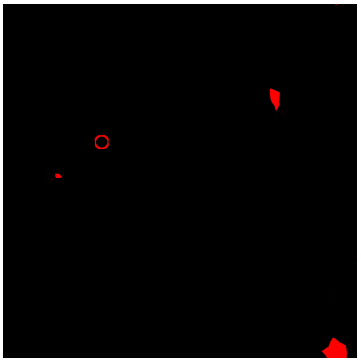
Normal Texture: The normal texture adds detail and depth to the Bomb Jump Sphere's surface by simulating small surface imperfections and bumps. This enhances the sphere's realism and makes it look more tactile.

Occlusion Texture: The occlusion texture helps simulate how ambient light is occluded or blocked by nearby surfaces. It adds shadows and shading to the sphere, making it appear more integrated into the game's environment.

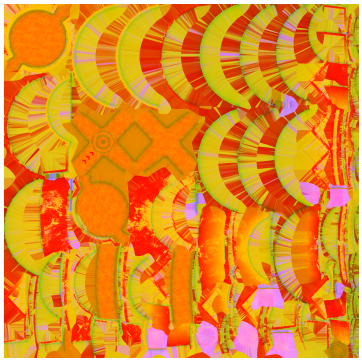
Bomb Jump Sphere Model: This is the 3D model of the Bomb Jump Sphere itself. It's the three-dimensional representation of the sphere that combines all of these textures to create the final visual appearance of the Bomb Jump Sphere in the game.



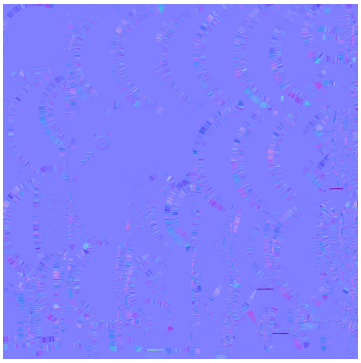
(a) Base Color Texture



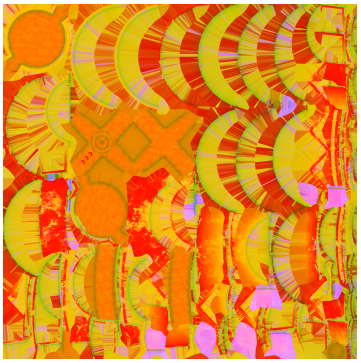
(b) Emissive Texture



(c) Metallic Texture



(a) Normal Texture



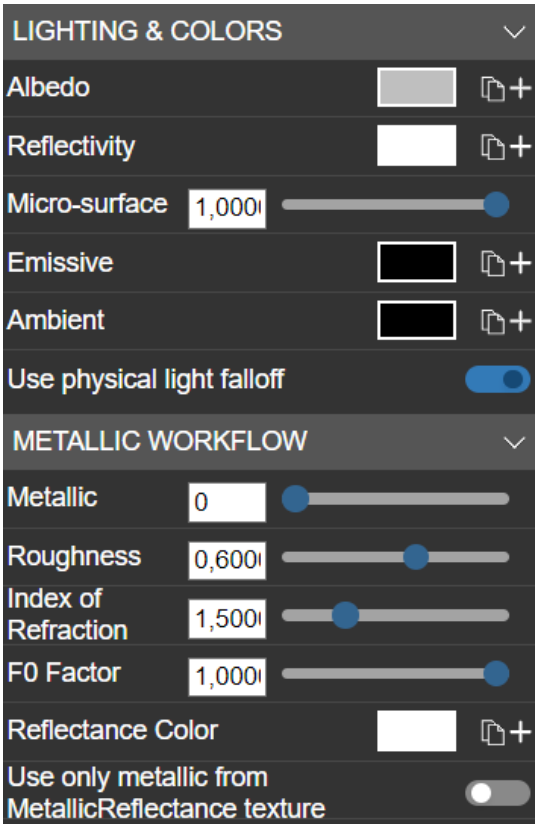
(b) Occlusion Texture

(c) Bomb Jump Sphere Model

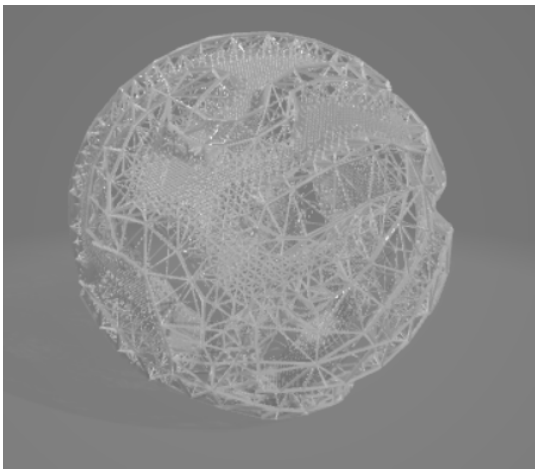
Figure 12: Bomb Jump Sphere Model with Materials

4.4 Bubble Protection Sphere Texture

The Bubble Protection Sphere doesn't rely on textures for its visual appearance. Instead, it uses a specialized material with a color that's close to white. This material is carefully chosen to emphasize the complex structure of the sphere model and create a specific visual effect. The complex hierarchical structure of the Bubble Protection Sphere is a key part of its design. Using a near-white material allows the player to clearly see and appreciate the intricate network of nodes and components that make up the sphere. This emphasizes the technical and protective nature of the sphere.



(a) Bubble Protection Sphere Material



(b) Bubble Protection Sphere Model

Figure 13: Bubble Protection Sphere Model with Materials

4.5 Water Texture

In the world of 3D graphics and game development, creating realistic water surfaces is a common challenge. Water is a complex element, reflecting light, exhibiting surface ripples, and adding depth to scenes. To achieve this level of realism in your 3D scene using Babylon.js, we utilize water materials and textures.

In the code snippet provided, we are configuring a water material for a specific mesh, presumably representing a water surface.

Graphics rendering can be computationally intensive, but we can optimize it by enabling back face culling (backFaceCulling = true). This means that only the visible sides of polygons are rendered, improving performance by not rendering the backs of surfaces that are unlikely to be seen.

Adding Surface Details with Bump Texture: To create the illusion of water waves and ripples, a bump texture is employed. This texture, set as water.bumpTexture, defines the height variations on the water surface. By altering the content of this texture, you can control the appearance of the water's surface.

A key aspect of realistic water is simulating the effects of wind. Here, we set water.windForce to -10, indicating a significant force that impacts the water's surface. Additionally, water.windDirection determines the direction from which the wind is blowing, influencing the orientation of waves and ripples.

To control the overall height of the water waves, we use water.waveHeight. This parameter affects the prominence of waves in the scene. Surface ripples are adjusted with the water.bumpHeight parameter. This value determines the intensity of ripples created by the

bump texture.
Water isn't just about waves; it's also about color. We set the water.waterColor to a specific shade of blue, represented by RGB values. This color provides the base hue for our water surface.

```
//WATER CONFIGURATION
function configure_water_material(scene, skybox, waterMesh) {
    var water = new BABYLON.WaterMaterial("water", scene, new BABYLON.Vector2(512,
     512));
    water.backFaceCulling = true;
    water.bumpTexture = new BABYLON.Texture(Assets.textures.water.Url, scene);
    water.windForce = -10;
    water.waveHeight = 1.7;
    water.bumpHeight = 0.1;
    water.windDirection = new BABYLON.Vector2(1, 1);
    water.waterColor = new BABYLON.Color3(0, 0, 221 / 255);
    water.colorBlendFactor = 0.0;
    water.addToRenderList(skybox);
    waterMesh.material = water;
    return water;
}
```

Listing 1: Water configuration

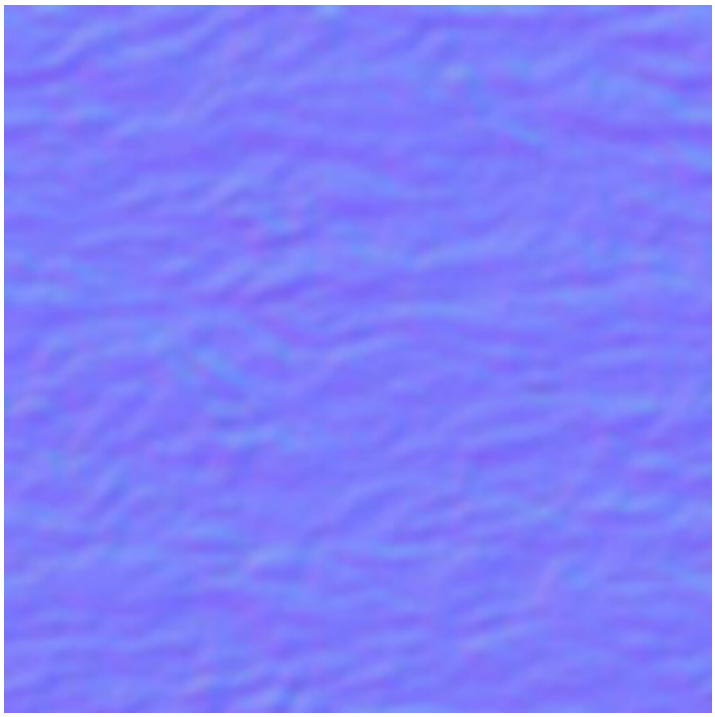


Figure 14: Water Bump Texture

4.6 Sky-box

In Babylon.js, a skybox is a technique used to simulate a distant background or environment around a 3D scene, making it appear more immersive and realistic. Skyboxes are commonly used in 3D graphics to create the illusion of a vast and continuous environment. Let's delve into how they work and why .env files are used.
A skybox in Babylon.js is essentially a textured cube or sphere that surrounds your entire

scene. It's a static background that provides the illusion of a distant environment. This technique is used to create a seamless and immersive experience for the viewer, especially when dealing with outdoor scenes or open-world environments.
Textures are a crucial part of creating a realistic skybox in Babylon.js. They provide the visual elements that make up the distant environment. These textures are usually wrapped around a cube or sphere, creating the appearance of a 360-degree environment. These textures can include features like distant landscapes, skies, clouds, or other elements that contribute to the scene's atmosphere.
We used five different skybox that can be selected by user in the settings. In particular we have used a cube as base mesh to apply the texture, all the enviroments have in common that has a bottom face water for consistency.

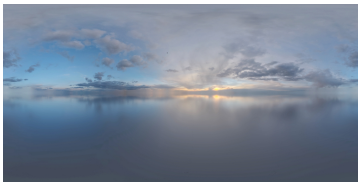
```
//  SKYBOX
var skybox = BABYLON.Mesh.CreateBox("skyBox", 5000.0, scene2);
var skyboxMaterial = configure_skybox_material(scene2, skybox);

...

//SKYBOX CONFIGURATION
function configure_skybox_material(scene, skybox) {
    var skyboxMaterial = new BABYLON.StandardMaterial("skyBox", scene);
    skyboxMaterial.backFaceCulling = false;
    skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(Assets.textures.sky
    .Url, scene);
    skyboxMaterial.reflectionTexture.coordinatesMode = BABYLON.Texture.SKYBOX_MODE
    ;
    skyboxMaterial.diffuseColor = new BABYLON.Color3(0, 0, 0);
    skyboxMaterial.specularColor = new BABYLON.Color3(0, 0, 0);
    skyboxMaterial.disableLighting = true;
    skybox.material = skyboxMaterial;
    return skyboxMaterial;
}
```

Listing 2: Skybox configuration

These are the skyboxes used in the games, and that can be selcted from player:

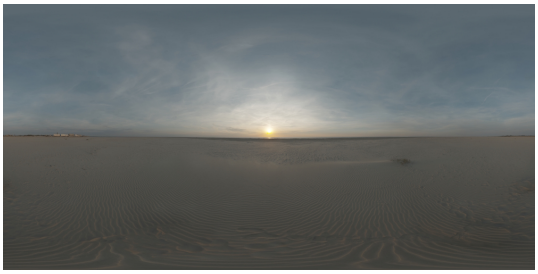


(a) Skybox 1

(b) Skybox 2

(c) Skybox 3

(a) Skybox 4



(b) Skybox 5

Figure 16: Skyboxes Textures

5 Animation

In order to make the use of the game as smooth and realistic as possible, we designed several animations. For the implementation of the animations we used the keyframes, that define specific points in time within an animation sequence.

5.1 Animation of Player movement

Player movement animation means all those animations that are performed when the player presses the WASD buttons on the keyboard. Therefore, they correspond to the forward or backward movement(W and S) and the rotation of the body to the right or left(A and D). As for the forward or backward movement, we simply perform a translation of the player's mesh position toward the direction of the camera, obtained by the function *camera.getDirection(BABYLON.Vector3.Forward()) or camera.getDirection(BABYLON.Vector3.Backward())*.

```
    function move_player(camera) {
    if (player == null) return;
    if (keyStatus[87]) { // 'W' key or up arrow key
        var cameraForward = camera.getDirection(BABYLON.Vector3.Forward());
        var speed = 0.04;
        var currentPosition = player.position.clone();
        var deltaPosition = cameraForward.scaleInPlace(speed);
        deltaPosition.y = 0;
        player.position = currentPosition.add(deltaPosition);
    }
    if (keyStatus[83]) { // 'S' key or down arrow key

        var cameraForward = camera.getDirection(BABYLON.Vector3.Backward());
        var speed = 0.04;
        var currentPosition = player.position.clone();
        var deltaPosition = cameraForward.scaleInPlace(speed);
        deltaPosition.y = 0;
        player.position = currentPosition.add(deltaPosition);

    }
    if (keyStatus[65]) { // 'A' key or left arrow key
        rotatePlayer("left")
    }
    if (keyStatus[68]) { // 'D' key or right arrow key
        rotatePlayer("right")
    }
}
```

Listing 3: Movement of the player

In this way, the player will move forward or backward following the direction of the camera. Specifically, the camera is a **FollowCamera** locked on the player's chest.

```
    camera.lockedTarget = playerScene["transformNodes"][5]; //chest
```

In this way, we will have that the direction will always be in front of the player.
In addition to the translation of the player, to make the movement more realistic we decided to perform animations on the arms and legs to simulate a person's walk. In this case, we used keyframes because they allowed us to more easily model each frame of motion with just a few lines of code.

```
var animationShoulderRight = new BABYLON.Animation("rotationAnimation", "
rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3,
BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);

var animationShoulderLeft = new BABYLON.Animation("rotationAnimation", "
rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
ANIMATIONLOOPMODE_CONSTANT);

var animationKneeLeft = new BABYLON.Animation("kneeLeftAnimation", "rotation",
 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
ANIMATIONLOOPMODE_CYCLE);

var animationKneeRight = new BABYLON.Animation("kneeRightAnimation", "rotation
", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
ANIMATIONLOOPMODE_CYCLE);

animationShoulderLeft.setKeys([
    { frame: 0, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI) },
    { frame: 5, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI * 220 /
180) },
    { frame: 10, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI) },
    { frame: 15, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI * 140 /
180) },
    { frame: 20, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI) }
]);
animationShoulderRight.setKeys([
    { frame: 0, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI) },
    { frame: 5, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI * 140 /
180) },
    { frame: 10, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI) },
    { frame: 15, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI * 220 /
180) },
    { frame: 20, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI) }
]);
animationKneeLeft.setKeys([
    { frame: 0, value: new BABYLON.Vector3(-Math.PI / 4, 0, 0) },
    { frame: 10, value: new BABYLON.Vector3(-Math.PI / 2, 0, 0) },
    { frame: 20, value: new BABYLON.Vector3(-Math.PI / 4, 0, 0) }
])
animationKneeRight.setKeys([
    { frame: 0, value: new BABYLON.Vector3(Math.PI / 16, 0, -Math.PI / 2) },
    { frame: 10, value: new BABYLON.Vector3(Math.PI / 16, 0, -Math.PI / 4) },
    { frame: 20, value: new BABYLON.Vector3(Math.PI / 16, 0, -Math.PI / 2) }
])

kneeLeft.animations.push(animationKneeLeft);
kneeRight.animations.push(animationKneeRight);
shoulderRight.animations.push(animationShoulderRight);
shoulderLeft.animations.push(animationShoulderLeft);
var animationGroupW = new BABYLON.AnimationGroup("rotationAnimationGroup");
animationGroupW.addTargetedAnimation(animationShoulderRight, shoulderRight);
animationGroupW.addTargetedAnimation(animationShoulderLeft, shoulderLeft);
animationGroupW.addTargetedAnimation(animationKneeLeft, kneeLeft);
animationGroupW.addTargetedAnimation(animationKneeRight, kneeRight);
```

Listing 4: Animation of arms and legs during the walk

Finally, when the player presses the A or D buttons, it will perform player rotation as well as arm and leg movement animation.

```
function rotatePlayer(direction) {
    transformNodes = player._scene.transformNodes
    chest = transformNodes[3]
    pelvis = transformNodes[25]
    torso = transformNodes[36]
    if (direction == 'right') {
        chest.rotation.y -= 0.02
        pelvis.rotation.y -= 0.02
        torso.rotation.y -= 0.02
    } else {
        chest.rotation.y += 0.02
        pelvis.rotation.y += 0.02
        torso.rotation.y += 0.02
    }
    chest.rotation = chest.rotation.clone()
    pelvis.rotation = pelvis.rotation.clone()
    torso.rotation = torso.rotation.clone()
}
```

Listing 5: Rotation of the player

Figure 17: Animation of player movement

5.2 Animation of Player falling and jumping

To fully simulate the correct behavior of the player when making a jump using the spacebar or when falling from a platform, we implemented several animations that go into changing the rotation of the arms and feet.
In particular, with regard to jumping, we have divided the animation into three parts: the ascent, the slowing down of the ascent, and the descent. The following code shows how we used keyframes to correctly construct the jump animation.

```
function jump(camera) {
    if (jumping === false) {
        jumping = true;
```

```
        root = player._scene.transformNodes[0]
        shoulderLeft = player._scene.transformNodes[7]
        elbowLeft = player._scene.transformNodes[8]
        shoulderRight = player._scene.transformNodes[16]
        elbowRight = player._scene.transformNodes[17]

        var animationShoulderLeft = new BABYLON.Animation("shoulderLeftAnimation", "rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3,
        BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);

        var animationShoulderRight = new BABYLON.Animation("shoulderRightAnimation", "rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);

        var animationElbowLeft = new BABYLON.Animation("elbowLeftAnimation", "rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);

        var animationElbowRight = new BABYLON.Animation("elbowRightAnimation", "rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);

        var jumpAnimation = new BABYLON.Animation("jumpAnimation", "position.y", 30, BABYLON.Animation.ANIMATIONTYPE_FLOAT, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);

        var jumpHeight = 2.0
        var jumpDuration = 45
        var framesPerStep = jumpDuration / 3; // Divide the jump animation into three steps (start, peak, and end)

        var jumpKeys = [];
        var shoulderLeftKeys = [];
        var shoulderRightKeys = [];
        var elbowLeftKeys = [];
        var elbowRightKeys = [];

        // Add keyframes for the first step (ascending)
        for (var i = 0; i <= framesPerStep; i++) {
            jumpKeys.push({
                frame: i,
                value: root.position.y + (jumpHeight / framesPerStep) * i,
            });
            jumpAnimation.addEvent(new BABYLON.AnimationEvent(
                i,
                function () {
                    var cameraForward = camera.getDirection(BABYLON.Vector3.Forward());
                    var speed = 0.07;
                    var currentPosition = player.position.clone();
                    var deltaPosition = cameraForward.scaleInPlace(speed);
                    deltaPosition.y = 0;
                    player.position = currentPosition.add(deltaPosition);
                },
                false
            ));
        }

        // Add keyframes for the second step (descending)
        for (var i = 1; i <= framesPerStep; i++) {
            jumpKeys.push({
```

```
                frame: framesPerStep + i,
                value: root.position.y + (jumpHeight - (jumpHeight / framesPerStep
) * i),
            });
            jumpAnimation.addEvent(new BABYLON.AnimationEvent(
                framesPerStep + i,
                function () {
                    var cameraForward = camera.getDirection(BABYLON.Vector3.
Forward());
                    var speed = 0.07;
                    var currentPosition = player.position.clone();
                    var deltaPosition = cameraForward.scaleInPlace(speed);
                    deltaPosition.y = 0;
                    player.position = currentPosition.add(deltaPosition);
                },
                false
            ));
        }

        // Add keyframe for the last step (return to original height)
        jumpKeys.push({
            frame: jumpDuration,
            value: root.position.y
        });

        shoulderLeftKeys.push(
            { frame: 0, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI) },
            { frame: 15, value: new BABYLON.Vector3(-Math.PI / 2, Math.PI / 2,
Math.PI) },
            { frame: 30, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI) }
        )

        shoulderRightKeys.push(
            { frame: 0, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI) },
            { frame: 15, value: new BABYLON.Vector3(-Math.PI / 2, -Math.PI / 2, -
Math.PI) },
            { frame: 30, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI) }
        )

        elbowLeftKeys.push(
            { frame: 0, value: new BABYLON.Vector3(0, 0, 0) },
            { frame: 15, value: new BABYLON.Vector3(0, 0, -Math.PI / 3) },
            { frame: 30, value: new BABYLON.Vector3(0, 0, 0) }
        )

        elbowRightKeys.push(
            { frame: 0, value: new BABYLON.Vector3(0, 0, 0) },
            { frame: 15, value: new BABYLON.Vector3(0, 0, Math.PI / 3) },
            { frame: 30, value: new BABYLON.Vector3(0, 0, 0) }
        )

        jumpAnimation.setKeys(jumpKeys);
        animationShoulderLeft.setKeys(shoulderLeftKeys);
        animationShoulderRight.setKeys(shoulderRightKeys);
        animationElbowLeft.setKeys(elbowLeftKeys);
        animationElbowRight.setKeys(elbowRightKeys);

        root.animations = [];
        shoulderLeft.animations = [];
        shoulderRight.animations = [];
        elbowLeft.animations = [];
```

```
        elbowRight.animations = [];

        root.animations.push(jumpAnimation);
        shoulderLeft.animations.push(animationShoulderLeft)
        shoulderRight.animations.push(animationShoulderRight)
        elbowLeft.animations.push(animationElbowLeft)
        elbowRight.animations.push(animationElbowRight)

        scene.beginAnimation(shoulderLeft, 0, 30, false, 1, isJumping);
        scene.beginAnimation(shoulderRight, 0, 30);
        scene.beginAnimation(elbowLeft, 0, 30);
        scene.beginAnimation(elbowRight, 0, 30);
        scene.beginAnimation(root, 0, jumpDuration);

    }
}
```

Listing 6: Animation of the player jump

Figure 18: Animation of player jump

On the other hand, as for the falling of the player(for example from a platform), we simulated the fact that the player raises his arms up being in the presence of gravity in accordance with the physics of the game.

```
function fallingAnimation(amountEndingFrame) {
    if (jumping === false) {
        jumping = true;
        shoulderLeft = player._scene.transformNodes[7]
        elbowLeft = player._scene.transformNodes[8]
        shoulderRight = player._scene.transformNodes[16]
        elbowRight = player._scene.transformNodes[17]
        kneeLeft = player._scene.transformNodes[27]
        kneeRight = player._scene.transformNodes[32]

```

```
11          var animationShoulderLeft = new BABYLON.Animation("shoulderLeftAnimation",
     "rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CYCLE);
12
13          var animationShoulderRight = new BABYLON.Animation("shoulderRightAnimation
    ", "rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CYCLE);
14
15          var animationElbowLeft = new BABYLON.Animation("elbowLeftAnimation", "
    rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CYCLE);
16
17          var animationElbowRight = new BABYLON.Animation("elbowRightAnimation", "
    rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CYCLE);
18
19          var animationKneeLeft = new BABYLON.Animation("kneeLefttAnimation", "
    rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CYCLE);
20
21          var animationKneeRight = new BABYLON.Animation("kneeRightAnimation", "
    rotation", 30, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CYCLE);
22
23          var shoulderLeftKeys = [];
24          var shoulderRightKeys = [];
25          var elbowLeftKeys = [];
26          var elbowRightKeys = [];
27          var kneeLeftKeys = [];
28          var kneeRightKeys = [];
29
30          shoulderLeftKeys.push(
31              { frame: 0, value: new BABYLON.Vector3(0, Math.PI / 2, Math.PI) },
32              { frame: 25 + amountEndingFrame, value: new BABYLON.Vector3(-Math.PI,
    Math.PI / 2, Math.PI) },
33              { frame: 35 + amountEndingFrame, value: new BABYLON.Vector3(0, Math.PI
     / 2, Math.PI) },
34          )
35
36          shoulderRightKeys.push(
37              { frame: 0, value: new BABYLON.Vector3(0, -Math.PI / 2, -Math.PI) },
38              { frame: 25 + amountEndingFrame, value: new BABYLON.Vector3(-Math.PI,
    -Math.PI / 2, -Math.PI) },
39              { frame: 35 + amountEndingFrame, value: new BABYLON.Vector3(0, -Math.
    PI / 2, -Math.PI) },
40          )
41
42          elbowLeftKeys.push(
43              { frame: 0, value: new BABYLON.Vector3(0, 0, 0) },
44              { frame: 25 + amountEndingFrame, value: new BABYLON.Vector3(0, 0, -
    Math.PI / 3) },
45              { frame: 35 + amountEndingFrame, value: new BABYLON.Vector3(0, 0, 0) }
46          )
47
48          elbowRightKeys.push(
49              { frame: 0, value: new BABYLON.Vector3(0, 0, 0) },
50              { frame: 25 + amountEndingFrame, value: new BABYLON.Vector3(0, 0, Math
    .PI / 3) },
51              { frame: 35 + amountEndingFrame, value: new BABYLON.Vector3(0, 0, 0) }
52          )
53
```

```
        kneeLeftKeys.push(
            { frame: 0, value: new BABYLON.Vector3(-Math.PI / 4, 0, 0) },
            { frame: 25 + amountEndingFrame, value: new BABYLON.Vector3(-Math.PI / 2, 0, 0) },
            { frame: 35 + amountEndingFrame, value: new BABYLON.Vector3(-Math.PI / 3, 0, 0) }
        )

        kneeRightKeys.push(
            { frame: 0, value: new BABYLON.Vector3(Math.PI / 16, 0, -Math.PI / 4) },
            { frame: 25 + amountEndingFrame, value: new BABYLON.Vector3(Math.PI / 16, 0, -Math.PI / 2) },
            { frame: 35 + amountEndingFrame, value: new BABYLON.Vector3(Math.PI / 16, 0, -Math.PI / 3) }
        )

        animationShoulderLeft.setKeys(shoulderLeftKeys);
        animationShoulderRight.setKeys(shoulderRightKeys);
        animationElbowLeft.setKeys(elbowLeftKeys);
        animationElbowRight.setKeys(elbowRightKeys);
        animationKneeLeft.setKeys(kneeLeftKeys);
        animationKneeRight.setKeys(kneeRightKeys);

        shoulderLeft.animations = [];
        shoulderRight.animations = [];
        elbowLeft.animations = [];
        elbowRight.animations = [];
        kneeLeft.animations = [];
        kneeRight.animations = [];

        shoulderLeft.animations.push(animationShoulderLeft)
        shoulderRight.animations.push(animationShoulderRight)
        elbowLeft.animations.push(animationElbowLeft)
        elbowRight.animations.push(animationElbowRight)
        kneeLeft.animations.push(animationKneeLeft)
        kneeRight.animations.push(animationKneeRight)

        scene.beginAnimation(shoulderLeft, 0, 35 + amountEndingFrame, false, 1, isJumping);
        scene.beginAnimation(shoulderRight, 0, 35 + amountEndingFrame);
        scene.beginAnimation(elbowLeft, 0, 35 + amountEndingFrame);
        scene.beginAnimation(elbowRight, 0, 35 + amountEndingFrame);
        scene.beginAnimation(kneeLeft, 0, 35 + amountEndingFrame);
        scene.beginAnimation(kneeRight, 0, 35 + amountEndingFrame);

    }
}
```

Listing 7: Animation of player falling

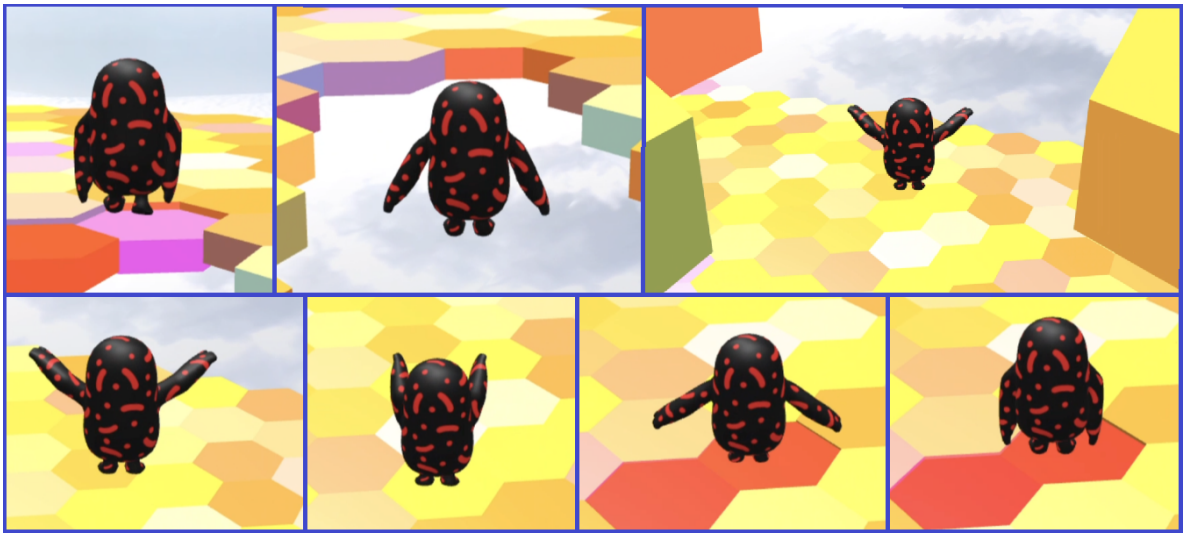


Figure 19: Animation of player falling

It is important to note that in order to start this animation, we perform a constant check that the player is colliding with the ground at all times. As soon as it detects that there is no entity under the player (colliding), then the animation is triggered. To do this, we defined the **isPlayerFalling()** function shown below.

```
function isPlayerFalling() {
    const raycastLength = 9.5; // The length of the ray to cast downward
    const raycastDirection = new BABYLON.Vector3(0, -1, 0); // The direction to
    cast the ray

    const origin = player.position.clone(); // Start the raycast from the player's
     position
    origin.y += 0.75 // Offset the starting position slightly above the player's
    feet

    const ray = new BABYLON.Ray(origin, raycastDirection, raycastLength);
    const hit = scene.pickWithRay(ray, (mesh) => mesh.isPickable && mesh !==
    player);
    return !hit || hit.distance > 2.5; // Return true if no collision or if the
    distance is greater than 1.0 (player is likely falling)
}
```

Listing 8: Check if the player is falling

5.3 Animation of Player endgame

```
function endGameAnimation(player_scene) {
    var transformNodes = player_scene["meshes"][0]._scene.transformNodes;
    var chest = transformNodes[3];
    var pelvis = transformNodes[25];
    var torso = transformNodes[36];

    var playerEndBones = player_scene["transformNodes"];
    var root = playerEndBones[0]
    var leftEye = playerEndBones[6]
    var rightEye = playerEndBones[7]
    var shoulderLeft = playerEndBones[8]
    var shoulderRight = playerEndBones[17]
    var kneeLeft = playerEndBones[28]
```

```
    var kneeRight = playerEndBones[33]

    var animationShoulderLeftEnd = new BABYLON.Animation("
    EndGameAnimation_shoulderLeft", "rotation", 240, BABYLON.Animation.
    ANIMATIONTYPE_VECTOR3, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE);


   ...

    var animationKneeRightEnd = new BABYLON.Animation("EndGameAnimation_kneeRight"
    , "rotation", 240, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CONSTANT);

    var initialPosition = root['_position'];
    var initialPositionLeftEye = leftEye['_position'];
    var initialPositionRightEye = rightEye['_position'];
    var initialPositionHexagon = hexagonEnd.position.clone();
    var initialColorHexagon = hexagonEnd.material.diffuseColor.clone();

    animationPositionHexagonEnd.setKeys([
        { frame: 0, value: initialPositionHexagon },
        { frame: 120, value: new BABYLON.Vector3(initialPositionHexagon.x,
    initialPositionHexagon.y - 0.06, initialPositionHexagon.z) },
        { frame: 240, value: new BABYLON.Vector3(initialPositionHexagon.x,
    initialPositionHexagon.y - 0.12, initialPositionHexagon.z) },
        { frame: 360, value: new BABYLON.Vector3(initialPositionHexagon.x,
    initialPositionHexagon.y - 0.06, initialPositionHexagon.z) },
        { frame: 480, value: initialPositionHexagon },
        { frame: 600, value: new BABYLON.Vector3(initialPositionHexagon.x,
    initialPositionHexagon.y - 0.06, initialPositionHexagon.z) },
        { frame: 720, value: new BABYLON.Vector3(initialPositionHexagon.x,
    initialPositionHexagon.y - 0.12, initialPositionHexagon.z) },
        { frame: 840, value: new BABYLON.Vector3(initialPositionHexagon.x,
    initialPositionHexagon.y - 0.06, initialPositionHexagon.z) },
        { frame: 960, value: initialPositionHexagon }
    ]);

    animationColorHexagonEnd.setKeys([
        { frame: 0, value: initialColorHexagon },
        { frame: 120, value: new BABYLON.Color3(1, 0, 1) },
        { frame: 240, value: initialColorHexagon },
        { frame: 360, value: new BABYLON.Color3(1, 0, 0) },
        { frame: 480, value: initialColorHexagon },
        { frame: 600, value: new BABYLON.Color3(0, 1, 1) },
        { frame: 720, value: initialColorHexagon },
        { frame: 840, value: new BABYLON.Color3(1, 1, 0) },
        { frame: 960, value: initialColorHexagon }
    ]);

    ...

    animationRotationEnd.setKeys([
        { frame: 0, value: new BABYLON.Vector3(0, Math.PI * 2, 0) },
        { frame: 240, value: new BABYLON.Vector3(0, Math.PI * 3 / 2, 0) },
        { frame: 480, value: new BABYLON.Vector3(0, Math.PI, 0) },
        { frame: 720, value: new BABYLON.Vector3(0, Math.PI / 2, 0) },
        { frame: 960, value: new BABYLON.Vector3(0, 0, 0) }
    ]);

    ...
    animationGroupEndGame.onAnimationGroupEndObservable.add(function (event) {
```

```
        animationGroupEndGame.start();
    });

    animationGroupEndGame.start();
}
```

Listing 9: Animation of Player endgame

The provided function called endGameAnimation define a complex animation sequence for various elements of player and hexagon. The final result is the hexagon that go up and down flashing different colors, and player jumping on it moving up the arms and legs. The function begins by obtaining the transform nodes of player model, such as chest, pelvis, torso, and various body parts, using their indices.
For each animation, a series of keyframes are defined, specifying how the animated property (e.g., position, rotation, color) changes over time frames. Each keyframe includes the frame number and the target value for that frame.
Each animation is associated with a specific element in the 3D scene by using addTargetedAnimation. For example, the shoulder animations are assigned to the shoulder nodes, and so on. An animation group named "EndGameAnimationGroup" is created. This group includes all the previously defined animations and their associated targets.
We report now some keyfram of the animation:

Figure 20: Animation of Player endgame

5.4 Animation of Hexagons

```
function hexagon_pressed(hexagon) {
    var animationHexagon = new BABYLON.Animation("positionAnimation", "position",
    BabylonEngine.getFps().toFixed(), BABYLON.Animation.ANIMATIONTYPE_VECTOR3,
    BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);
    var initialPosition = hexagon.position.clone();

    animationHexagon.setKeys([
        { frame: 0, value: initialPosition },
```

```
        { frame: 15, value: new BABYLON.Vector3(initialPosition.x, initialPosition.y - 0.06, initialPosition.z) },
        { frame: 30, value: new BABYLON.Vector3(initialPosition.x, initialPosition.y - 0.12, initialPosition.z) },
        { frame: 45, value: new BABYLON.Vector3(initialPosition.x, initialPosition.y - 0.06, initialPosition.z) },
        { frame: 60, value: initialPosition }
    ]);

    var animationColor = new BABYLON.Animation("colorAnimation", "material.diffuseColor", BabylonEngine.getFps().toFixed(), BABYLON.Animation.ANIMATIONTYPE_COLOR3, BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT);
    var initialColor = hexagon.material.diffuseColor.clone();

    animationColor.setKeys([
        { frame: 0, value: initialColor },
        { frame: 10, value: new BABYLON.Color3(1, 0, 0) },
        { frame: 15, value: initialColor },

        ...

        { frame: 55, value: initialColor },
        { frame: 60, value: new BABYLON.Color3(1, 0, 0) }
    ]);

    hexagon.animations.push(animationHexagon);
    hexagon.animations.push(animationColor);

    var animationGroupHexagon = new BABYLON.AnimationGroup("hexagonAnimationGroup");
    animationGroupHexagon.addTargetedAnimation(animationHexagon, hexagon);
    animationGroupHexagon.addTargetedAnimation(animationColor, hexagon);

    animationGroupHexagon.start();
}
```

Listing 10: Animation of hexagons

The hexagon_pressed function is responsible for creating and initiating an animation sequence for a hexagon when it's pressed by a player.
The function begins by setting up keyframes animations for the hexagon's position and color. Keyframes animations allow us to define a sequence of frames and values that the animation will interpolate between.
AnimationHexagon is created to animate the position of the hexagon. It starts with the hexagon's initial position and moves through a series of keyframes. The hexagon descends slightly (y coordinate decreases) at frames 15 and 45, simulating a bouncing effect. Finally, it returns to its initial position at frame 60, creating a bouncing animation loop.
AnimationColor is set up to animate the hexagon's material color. It changes the diffuse color of the hexagon's material to red between frames 10 and 60. This animation causes the hexagon to change color like flashing until is disposed When function is called, it initiates a visually appealing animation for the hexagon object. The hexagon appears to bounce while changing color, creating a feedback effect when the player interacts with it. This kind of animation adds interactivity and engagement to your game, making the gameplay experience more enjoyable and dynamic. Now we report visually the keyframes:

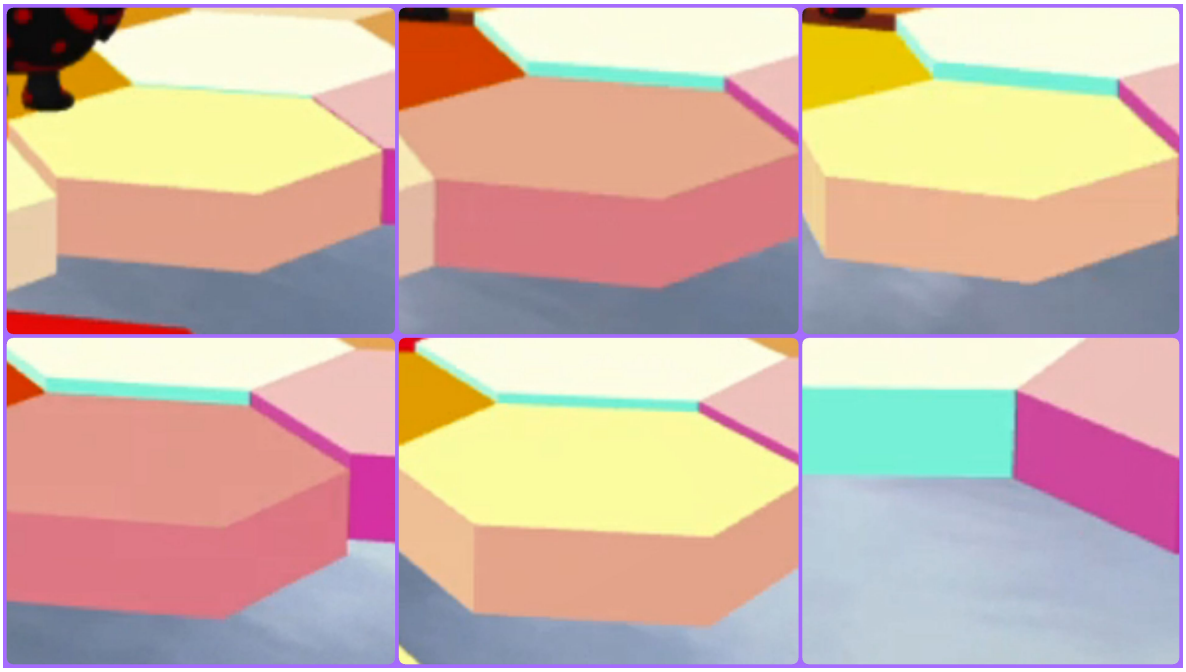


Figure 21: Animation of Hexagons

From the collage, it's evident that the hexagon descends, then ascends while dynamically changing colors, until it finally returns to its initial position before vanishing.

5.5 Animation of Spheres

```
function spheresAnimation(sphere) {

    var animationSpherePosition = new BABYLON.Animation("AnimationSphere_position"
    , "position", 60, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CONSTANT);

    var animationSphereRotation = new BABYLON.Animation("ANimationSphere_rotation"
    , "rotation", 60, BABYLON.Animation.ANIMATIONTYPE_VECTOR3, BABYLON.Animation.
    ANIMATIONLOOPMODE_CONSTANT);

    var initialPosition = sphere.position;

    animationSphereRotation.setKeys([
        { frame: 0, value: new BABYLON.Vector3(0, 0, 0) },
        { frame: 30, value: new BABYLON.Vector3(0, Math.PI / 2, 0) },
        { frame: 60, value: new BABYLON.Vector3(0, Math.PI, 0) },
        { frame: 90, value: new BABYLON.Vector3(0, Math.PI * 3 / 2, 0) },
        { frame: 120, value: new BABYLON.Vector3(0, Math.PI * 2, 0) }
    ]);


    animationSpherePosition.setKeys([
        { frame: 0, value: new BABYLON.Vector3(initialPosition.x, initialPosition.
    y, initialPosition.z) },
        { frame: 15, value: new BABYLON.Vector3(initialPosition.x, initialPosition
    .y + 0.2, initialPosition.z) },
        { frame: 30, value: new BABYLON.Vector3(initialPosition.x, initialPosition
    .y + 0.3, initialPosition.z) },
```

```
        { frame: 45, value: new BABYLON.Vector3(initialPosition.x, initialPosition.y + 0.2, initialPosition.z) },
        { frame: 60, value: new BABYLON.Vector3(initialPosition.x, initialPosition.y, initialPosition.z) }
    ]);

    sphere.animations.push(animationSpherePosition);
    sphere.animations.push(animationSphereRotation);

    var animationSphere = new BABYLON.AnimationGroup("AnimationSphereGroup");
    animationSphere.addTargetedAnimation(animationSpherePosition, sphere);
    animationSphere.addTargetedAnimation(animationSphereRotation, sphere);

    animationSphere.loopAnimation = false;


    animationSphere.onAnimationGroupEndObservable.add(function (event) {
        animationSphere.start();
    });


    animationSphere.start();

    return animationSphere;
}
```

Listing 11: Animation of Spheres

This function is the core of the animation of each sphere present in the game.
Two animation objects, animationSpherePosition and animationSphereRotation, are created. These animations will manipulate the position and rotation of the sphere, respectively.
The animationSphereRotation is configured to change the rotation of the sphere gradually. It specifies keyframes at different frames to create a smooth rotation effect. The sphere rotates by 360 degrees (2π) over the course of 120 frames.
Similarly, the animationSpherePosition is configured to create an up-and-down motion of the sphere. It moves the sphere along the Y-axis to give the appearance of it going up and down. It also uses keyframes to control the motion.
An event listener is added to the onAnimationGroupEndObservable of the animationSphere. When the animation group ends (i.e., the sphere completes its animation), it restarts the animation group. The sphere moves up and down while rotating around its vertical axis.
Now we report some keyframes of the animation:

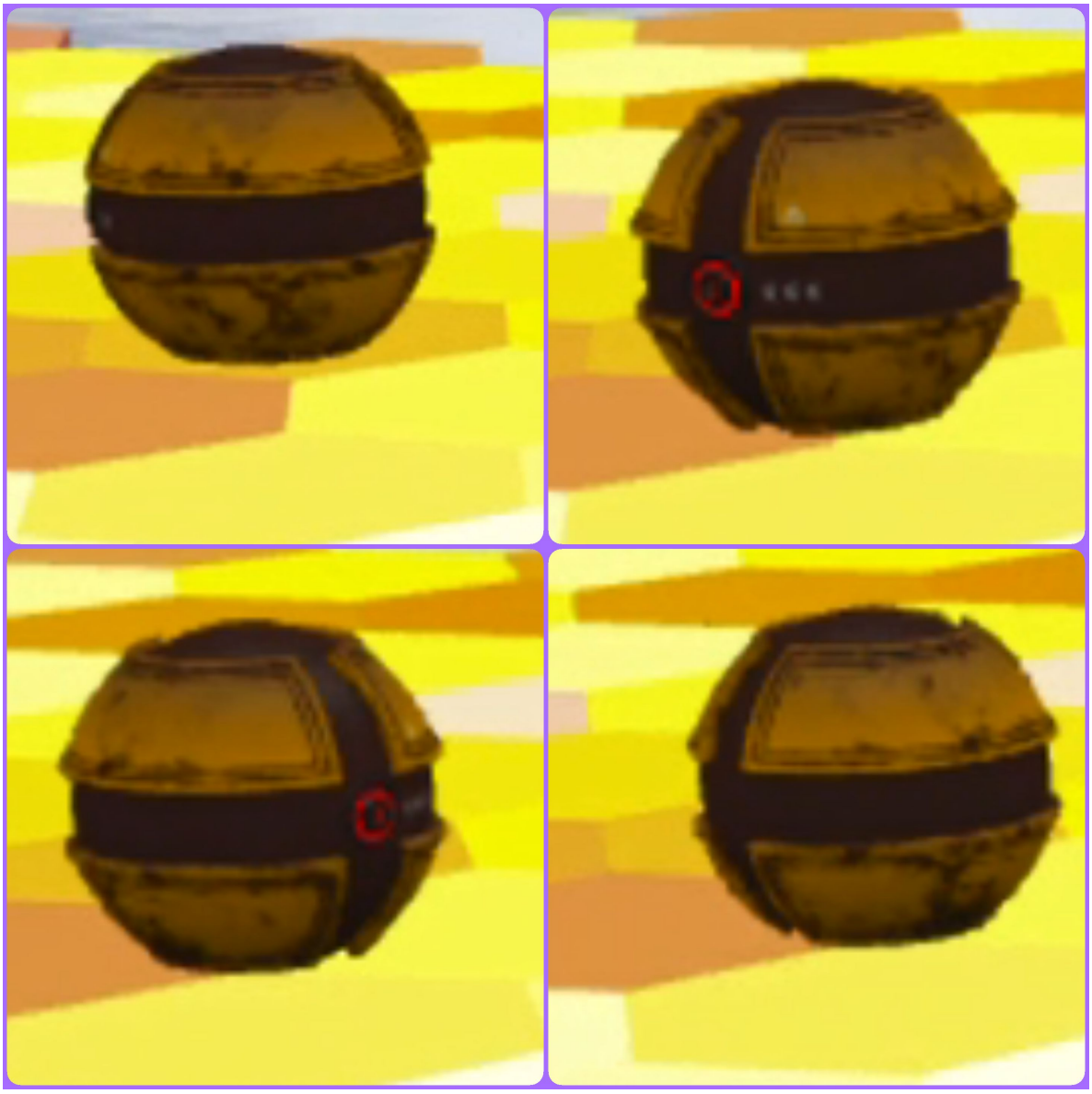


Figure 22: Animation of Spheres

6 Physics engine

Physics Settings and Engine In our Babylon.js project, we've incorporated a physics engine to simulate realistic interactions and movements of objects. Specifically, we've used the Oimo.js physics engine through the BABYLON.OimoJSPlugin.

Oimo Physics Engine is a 3D rigid-body physics engine commonly used in game development and other applications that require realistic physics simulations. It is designed to simulate the behavior of objects in a virtual 3D world, including collision detection, rigid body dynamics, and more.

```
1 //PHYSICS SETTINGS
2 const physicsPlugin = new BABYLON.OimoJSPlugin(); // Use the Oimo.js physics
      engine
3 const g = 9.81;
4
5 ...
```

```
6
7 // ENABLE PHYSICS
8 scene.enablePhysics(new BABYLON.Vector3(0, -g, 0), physicsPlugin);
```

Listing 12: Physics Settings

We create an instance of the Oimo.js physics plugin and configure the gravitational constant (g) to simulate Earth's gravity at approximately 9.81 m/s^2.
We enable physics in our scene using scene.enablePhysics. This call specifies the gravitational force acting in the negative Y-direction (upwards) and associates the Oimo.js plugin with scene.
Now we discuss how we handle physics for the player character. we have code that updates the player's linear velocity, applies random forces, and defines the physics impostor for the player's mesh:

```
1 // Handle Player Physics
2 const currentVelocity = player.physicsImpostor.getLinearVelocity().clone();
3
4 // Modify player's linear velocity on collison with hexagon
5 player.physicsImpostor.setLinearVelocity(new BABYLON.Vector3(0, Math.ceil(Math.abs
      (currentVelocity.y)) * 0.7, 0));
6
7 ...
8
9 // Modify player's linear velocity on collison with Bomb Sphere
10 player.physicsImpostor.setLinearVelocity(new BABYLON.Vector3(Math.random() * 40 -
      20, Math.ceil(Math.abs(currentVelocity.y)) * 12, Math.random() * 40 - 20));
11
12 ...
13 // Create a physics impostor for the player
14 player.physicsImpostor = new BABYLON.PhysicsImpostor(player, BABYLON.
      PhysicsImpostor.BoxImpostor, { mass: 1, restitution: 0, friction: 0 }, scene);
```

Listing 13: Physics uses

We have created a physics impostor for the player mesh using BABYLON.PhysicsImpostor. This impostor represents the physical properties of the player, including its mass, restitution (bounciness), and friction. It's essential for handling collisions and interactions with other objects in the scene.

6.1 Collision Boxes for Hexagons and Players

Collision implementation is a fundamental aspect of creating a dynamic and interactive 3D game or application using the Babylon.js framework and the Oimo Physics Engine. The primary goal of collision detection and response is to simulate realistic interactions between different objects in the virtual world. This process enhances the gaming experience by allowing objects, like the player character and hexagonal platforms, to respond appropriately when they come into contact.
The collision boxes serve as simplified representations of the actual objects and are essential for efficient collision detection. For example, the player's collision box (playerCollisionBox) is created to envelop the player character, while collision boxes for hexagonal platforms (hexagonBox) are generated based on their bounding information.
Collision detection is then performed within the scene.registerBeforeRender function. This function is executed on each frame update, making it the ideal location to check for collisions continuously. The code employs the intersectsMesh method, a built-in Babylon.js function, to determine if two mesh objects intersect. In this case, it checks if the player's collision box intersects with any of the hexagonal platform collision boxes.

Upon detecting a collision, the code triggers various actions. These actions include modifying object velocities, playing sounds, and updating game variables. The collision response is crucial for creating the illusion of physics in the virtual world. For instance, when the player collides with a hexagonal platform, their vertical velocity may be adjusted to simulate a bounce or impact effect.
Furthermore, our code maintains state information for each hexagonal platform, such as whether it has already been collided with and its remaining "life." If a hexagon has been collided with, its state is updated, and if its life reaches zero, the hexagon is disposed of. This element of the code showcases how collision detection can influence the game's progression and mechanics.

```
var playerCollisionBox = BABYLON.MeshBuilder.CreateBox("playerCollisionBox", {
    width: playerCollisionBoxDimensions.x, height: playerCollisionBoxDimensions.y,
     depth: playerCollisionBoxDimensions.z }, scene);
playerCollisionBox.parent = player;
playerCollisionBox.position.copyFrom(playerCollisionBoxPosition);
playerCollisionBox.isVisible = false;

...

 scene.registerBeforeRender(function () {
        ...

        for (var i = 0; i < hexagonsMap.length; i++) {

            ...

            if (!hexagonReal.isDisposed() && playerCollisionBox.intersectsMesh(
    hexagonBox, true)) {
                ...
            }
        ...
        }
    }
);

...

//HEXAGON COLLISION BOX
function create_collision_box(hexagon, scene, name, platformLevel) {
    var boundingInfo = hexagon.getBoundingInfo();
    var renderingPosition = boundingInfo.boundingBox.centerWorld;
    var hexagonCollisionBox = BABYLON.MeshBuilder.CreateDisc(name, { radius: 1,
    tessellation: 6 }, scene);
    hexagonCollisionBox.rotation.x = Math.PI / 2;
    hexagonCollisionBox.position = renderingPosition;
    hexagonCollisionBox.isVisible = false;

    ...

    hexagonsMap.push([hexagon, hexagonCollisionBox, false, 60, sphere, type]);

}
```

Listing 14: Collision detection

7 Camera Movement

The game's main camera is initialized in a specific function, namely **configure_camera**. The function creates a FollowCamera named "FollowCam" at a specific position in the scene. This camera type is designed to follow a target, which in our case is the chest of the player. In this way, the view is always along the direction in front of the player's body. In addition, we added some settings to the chamber, in particular:

- **heightOffset = 2**: sets the height offset of the camera to 2 units above the target
- **cameraAcceleration = .02**: sets the camera's acceleration when following the target
- **maxCameraSpeed = 1**: sets the maximum speed at which the camera can move.
- **attachControl(canvas, true)**: attaches camera controls to a canvas element, allowing the user to interactively control the camera (e.g., zoom, rotate) using the mouse or touch input.

```
//CONFIGURATION OF THE CAMERA
function configure_camera(scene) {
    var fov = localStorage.getItem("camera_fov") || 120;
    // Parameters: name, position, scene
    let camera = new BABYLON.FollowCamera("FollowCam", new BABYLON.Vector3(90,
    100, 0), scene);
    camera.heightOffset = 2;
    camera.rotationOffset = 180;
    camera.cameraAcceleration = .02;
    camera.maxCameraSpeed = 1;
    camera.attachControl(canvas, true);
    return camera;
}
```

Listing 15: Camera configuration

8 Light Configuration

For the light configuration, we used a hemispherical light created with the following code:

```
var light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(0, 1, 0),
    scene);

light.intensity = 0.5;
```

Listing 16: Light configuration

Using this function in Babylon js we create a new Hemispheric Light object and initialise it with a name passed as the first parameter, the second parameter represents a vector specifying the direction of the light and finally the scene in which the light will be added. Hemispheric lights are present in both scenes and provide ambient lighting, illuminating objects from the specified direction. Moreover, we added a parameter to the setup of the light:

- **intensity = 0.5** : sets the intensity of the light to 0.5

9 Musics

Music and sound effects play a crucial role in enhancing the gaming experience in Babylon.js, as they help create an immersive and engaging atmosphere for players. Here's how we setup in the game:

```
musics: {
    soundtrack: {
        Url: "assets/static/sounds/soundtrack.mp3"
    },
    endgame: {
        Url: "assets/static/sounds/endgame.mp3"
    },
    startgame: {
        Url: "assets/static/sounds/startgame.mp3"
    },
    sphere1: {
        Url: "assets/static/sounds/sphere1.mp3"
    },
    sphere2: {
        Url: "assets/static/sounds/sphere2.mp3"
    }
}

...

// MUSIC IN BACKGROUND
var soundtrack = new BABYLON.Sound("soundtrack", Assets.musics.soundtrack.Url,
    scene, null, {
    loop: true,
    autoplay: true
});

var sphere1Sound = new BABYLON.Sound("endgame", Assets.musics.sphere1.Url, scene2,
     null, {
    loop: false,
    autoplay: false
});


function endSphereSound1() {
    sphere1Sound.stop();
}

...

// Create a sound for the countdown
var countdownMusic = new BABYLON.Sound("countdownMusic", Assets.musics.startgame.
    Url, scene, null, {
    loop: false,
    autoplay: true
});
```

Listing 17: Music configuration

Background Music (soundtrack) initializes a background soundtrack using Babylon.js's BABYLON.Sound class. The loop: true option ensures that the soundtrack plays continuously, creating a consistent musical backdrop for the game. The autoplay: true option starts playing the soundtrack automatically when the scene is loaded.
Sphere Sounds (sphere1Sound and sphere2Sound) are specific sound effects associated with interaction of player with spheres. The code initializes two sphere sounds. sphere1Sound is

set to loop false, which means it will play once without repeating.
The importance of sound in a game, especially in a 3D environment like Babylon.js, cannot be overstated. Background music helps set the mood and immerse players in the game's world. It can convey emotions, enhance tension, or provide relaxation, depending on the game's context.
Sound effects like sphere sounds provide crucial feedback to players. They can indicate good actions, danger, or the completion of tasks, enhancing the player's understanding of the game world and their progress within it. In our case the Bomb sphere play on interaction with player an explosion sound that is coherent with his effect. Also Protection sphere has a bubble sound that is coherent with the bubble that appear after on player.
In the endgame we changed background music for help player to understand that the game is finished.
We used a countdownMusic sound that likely represents the countdown sound effect, such as the "Ready, Set, Go!" in Mario Kart. It's set to play once (loop: false) and starts playing automatically when the scene is loaded (autoplay: true). This sound prepare players for the start of the game. It enhances the overall gaming experience and adds an element of anticipation and excitement.

10 Settings

On the settings page, you can change the game options as you wish. Of course, it is possible to change the options during the game, so the game restarts with the new options set up. Game settings are related to the player name (if survival mode) or the names of the two players (if 1 Vs 1 mode), difficulty, game maps, motion blur, texture platform, and volume. There are also three buttons in the game settings: 'Save', which allows the player to save the settings set, 'Reset', which allows the player to set the default settings, and finally 'Exit' to return to the home page.

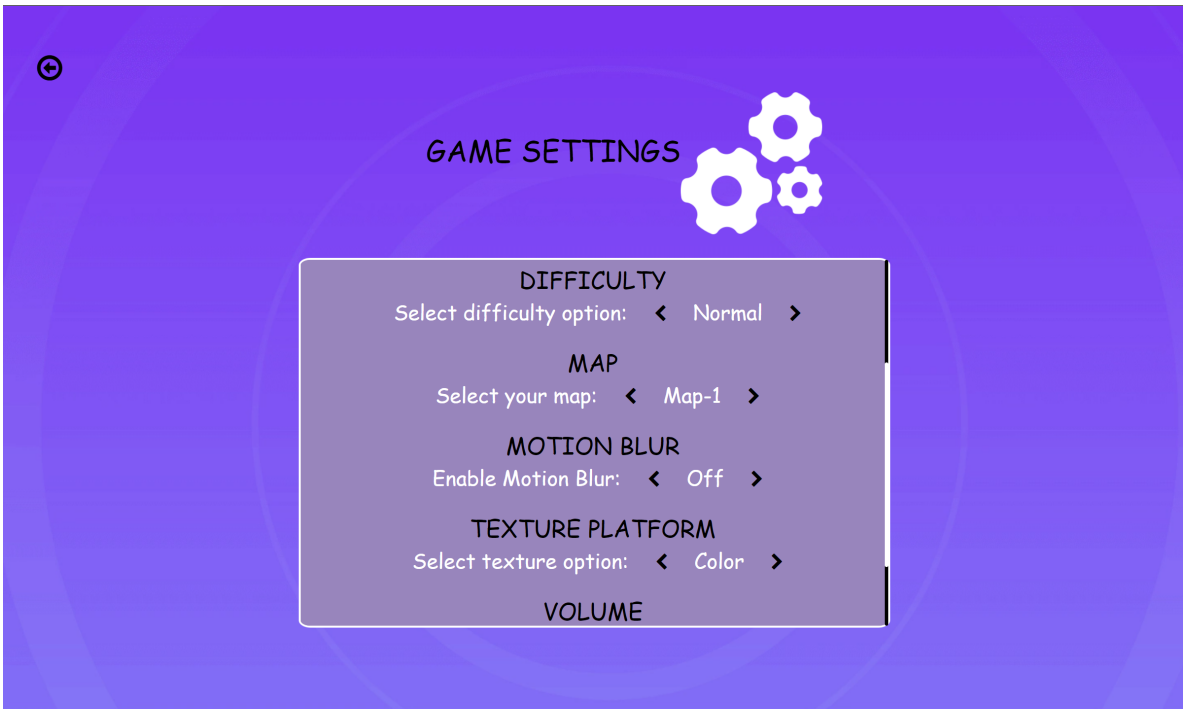


Figure 23: Settings

10.1 Player Names

To set the player's name, we took the name entered by the player on the setting page of the game and displayed it during the game. When the mode is 1Vs1, i.e. the names entered are two, we took them and displayed them in the game. To do this, we use local storage so that we can set and get the names. This feature allows the player to have a customised game experience.

Figure 24: Change player names

10.2 Difficulty

It is possible to switch on three levels of difficulty: Easy, Normal (default), and Advanced. Moving from one level to another results in a different duration of the hexagon collapse, a different duration of the bubble protection sphere and a different jump power range of the bomb jump sphere (even if the jump power is random for a specified range). The function to configure the difficulty is the following:

```
//CONFIGURATION DIFFICULTY
function configure_difficulty(diff){
    if(diff == "Easy"){
        lifeHexagon = 90;
        invicibilityTime = 10000;
        range_x_z_bomb = 40;
    }
    else if(diff == "Normal"){
        lifeHexagon = 60;
        invicibilityTime = 5000;
        range_x_z_bomb = 300;
    }
    else if(diff == "Advanced"){
        lifeHexagon=30;
        invicibilityTime = 1000
        range_x_z_bomb = 700;
```

```
    }
    else{
        lifeHexagon = 60;
        invicibilityTime = 5000;
        range_x_z_bomb = 300;
    }
}
```

Listing 18: Difficulty configuration

10.3 Game Maps

For the game maps, it is possible to switch between five maps, so that the player can play in different scenarios. The code to set up different game maps is described below:

```
//SKYBOX CONFIGURATION
function configure_skybox_material(scene, skybox) {
    var maps = localStorage.getItem('maps');
    var skyboxMaterial = new BABYLON.StandardMaterial("skyBox", scene);
    if (maps == "Map-1") {
        skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(Assets.textures
    .sky.Url, scene);
    } else if (maps == "Map-2") {
        skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(Assets.textures
    .sky1.Url, scene);
    } else if (maps == "Map-3") {
        skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(Assets.textures
    .sky2.Url, scene);
    } else if (maps == "Map-4") {
        skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(Assets.textures
    .sky3.Url, scene);
    } else if (maps == "Map-5") {
        skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(Assets.textures
    .sky4.Url, scene);
    }
    skyboxMaterial.backFaceCulling = false;
    skyboxMaterial.reflectionTexture.coordinatesMode = BABYLON.Texture.SKYBOX_MODE
    ;
    skyboxMaterial.diffuseColor = new BABYLON.Color3(0, 0, 0);
    skyboxMaterial.specularColor = new BABYLON.Color3(0, 0, 0);
    skyboxMaterial.disableLighting = true;
    skybox.material = skyboxMaterial;
    return skyboxMaterial;
}
```

Listing 19: Game maps configuration

10.4 Motion Blur

It is possible to activate and deactivate the motion blur from the game settings. The function used to configure the motion blur is the following:

```
//CONFIGURATION OF MOTION BLUR
function configure_motion_blur(scene, camera) {

    //CHECK MOTION BLUR
    var mbValue = localStorage.getItem("motion_blur") || "Off";
    console.log(mbValue);

    if (mbValue == "On") {
```

```
        var motionBlur = new BABYLON.MotionBlurPostProcess(
            'motionBlur', // name
            scene, // scene
            1.0, // motion blur strength
            camera // camera
        );

        motionBlur.motionStrength = 2;
        motionBlur.motionBlurSamples = 16;

    }
}
```

Listing 20: Motion Blur configuration

10.5 Texture Platform

In the game settings, it is also possible to change the texture of the platform between: "Random Color", which represents the texture applied by default, "Wood", which applies a wood-like texture to the platform, and finally "Stone", which applies a stone-like texture to the platform. The function to set up this feature is described below:

```
//FUNCTION TO CONFIGURE THE TEXTURE PLATFORM
function configure_texture_platform(texture){

    if(texture == "Wood"){
        model = Assets.models.hexagon1.Url;
        modelName = "hexagon1.glb";
        modelPlatform = Assets.models.platform1.Url ;
        modelPlatformName = "platform1.glb";

    }
    else if(texture == "Stone"){
        model = Assets.models.hexagon2.Url;
        modelName = "hexagon2.glb";
        modelPlatform = Assets.models.platform2.Url ;
        modelPlatformName = "platform2.glb";
    }
    else{
        model = Assets.models.hexagon.Url;
        modelName = "hexagon.glb";
        modelPlatform = Assets.models.platform.Url ;
        modelPlatformName = "platform.glb";
    }
}
```

Listing 21: Texture Platform configuration

10.6 Volume

In the game settings, you can set the game volume to your liking by changing the volume slider. Below is the code for setting the volume during the game:

```
    //VOLUME
    var volume = localStorage.getItem('volume');
    BABYLON.Engine.audioEngine.setGlobalVolume(volume / 100);
```

Listing 22: Volume configuration

11 Multiplayer

One of the most important features we have implemented in our project is the chance to play multiplayer games, specifically to be able to play a game with **two real players**, in 1v1 mode. The first player will play through the **WASD** keys, while the second player with the keys ↑←↓→.
In order for both players to have a dedicated view, we split the screen into two equal parts. The left part corresponds to the first player and he will have his own camera, while the right part corresponds to the second player and he too will have another dedicated camera. Both cameras are FollowCameras and are locked on the chest of the first and second player, respectively. This is done by the following code:

```
    const cameraFirst = configure_camera(scene, "First");

    const cameraSecond = configure_camera(scene, "Second");

    scene.activeCameras.push(cameraFirst);
    scene.activeCameras.push(cameraSecond);

    // Create two viewports
    var viewport1 = new BABYLON.Viewport(0, 0, 0.5, 1);
    var viewport2 = new BABYLON.Viewport(0.5, 0, 0.5, 1);

    // Set the viewports on the cameras
    cameraFirst.viewport = viewport1;
    cameraSecond.viewport = viewport2;

    cameraFirst.lockedTarget = playerScene["transformNodes"][5]; //chest player 1
    cameraSecond.lockedTarget = playerSceneTwo["meshes"][0]._scene.transformNodes
    [45]; //chest player 2
```

Listing 23: 1v1 cameras configuration

Obviously, we mapped player two's commands to movement and jump actions as in player one. In this way, both players can perform the same type of actions, animations and movement.

```
        if (keyStatus[87] || keyStatus[83]) { //press W or S
            move_player(cameraFirst, player);
            if (!jumping) {
                animationGroupW.start();
            }
        }
        if (keyStatus[68]) { //press D
            move_player(cameraFirst, player);
            if (!jumping) {
                animationGroupW.start();
            }
        }
        if (keyStatus[65]) { //press A
            move_player(cameraFirst, player);
            if (!jumping) {
                animationGroupW.start();
            }
        }
        if (keyStatus[38] || keyStatus[40]) { //press ArrowUp or ArrowDown
            move_player(cameraSecond, playerTwo);
            if (!jumpingTwo) {
                animationGroupWTwo.start();
            }
        }
```

```
        if (keyStatus[39]) { //press ArrowRight
            move_player(cameraSecond, playerTwo);
            if (!jumpingTwo) {
                animationGroupWTwo.start();
            }
        }
        if (keyStatus[37]) { //press ArrowLeft
            move_player(cameraSecond, playerTwo);
            if (!jumpingTwo) {
                animationGroupWTwo.start();
            }
        }
        if (keyStatus[32]) { //press spacebar
            jump(cameraFirst, player);
        }
        if (keyStatus[16]) { //press ShiftRight
            jump(cameraSecond, playerTwo);
        }
```

Listing 24: 1v1 player key mapping

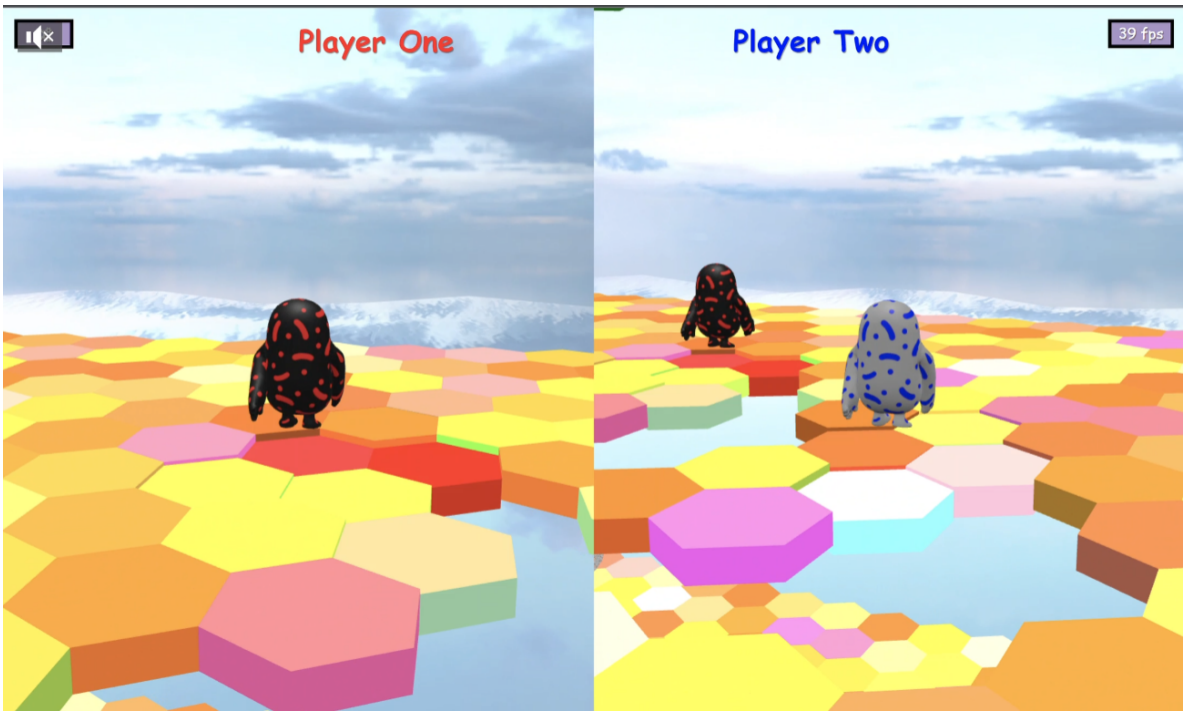


Figure 25: 1v1 Multiplayer game

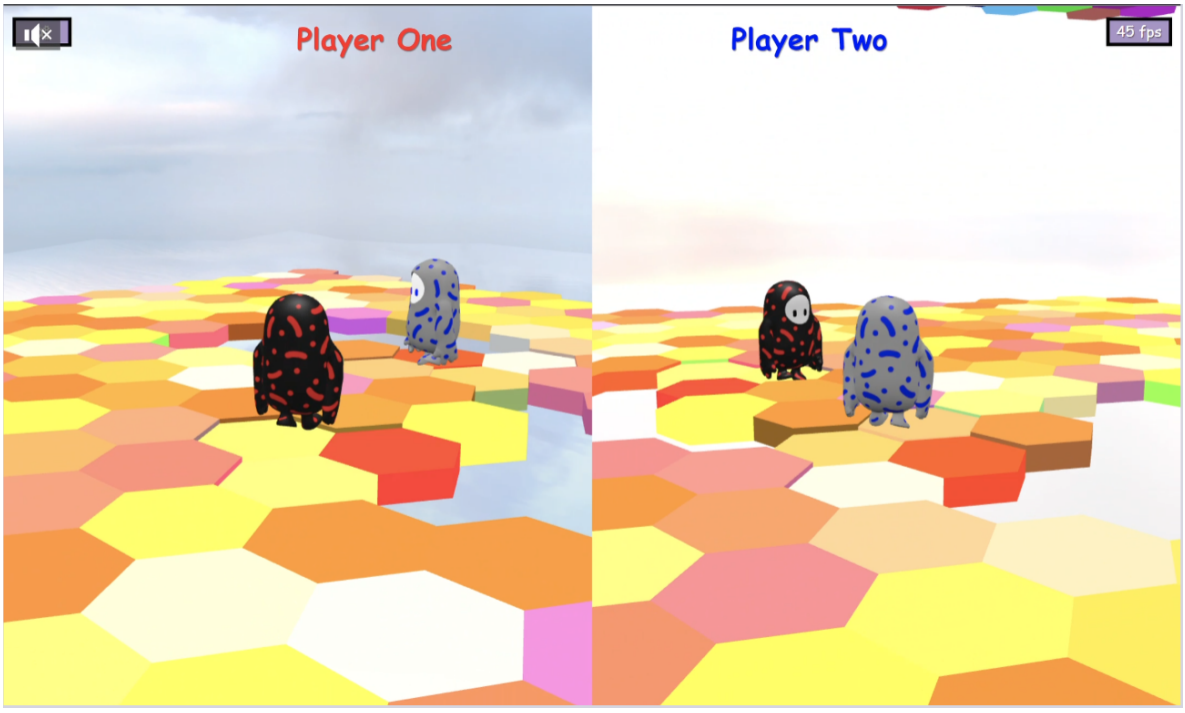


Figure 26: 1v1 Multiplayer game

12 Manual of game

The manual provides a description of the two game modes and the commands for playing in both modes. It is possible to see the manual also in the tutorial page of the game.

12.1 Survival Mode

In the survival mode, the player begins the round by floating above spaced-out hexagonal platforms. After a countdown, the player is dropped onto their own platform. When a platform is touched, it starts blinking and immediately disappears. Below several levels of hexagons is a water level that ends the game. When the player touches the water, he is disqualified. Above each hexagonal platform, there are two types of spheres: the bomb jump sphere which, if touched, explodes and makes the player jump, the bouble protection sphere which, if touched, encloses the player in a bubble that prevents the platform from disappearing. Try to survive as long as possible! Do your best!

12.2 Survival Commands

Figure 27: Survival Commands

12.3 1 Vs 1 Multiplayer

In the 1 Vs 1 mode, Players begin the round by floating on spaced-out hexagonal platforms. After a countdown, the two players are dropped onto their own platform. When a platform is touched, it starts blinking and immediately disappears. Below several levels of hexagons is a water level that ends the game. When a player touches the water, he is disqualified. Above each hexagonal platforms, there are two types of spheres: the bomb jump sphere which, if touched, explodes and makes the player jump, and the bouble protection sphere which, if touched, encloses the player in a bubble that prevents the platform from disappearing. Try to survive longer than your opponent and may the best man win!

12.4 1 Vs 1 Commands

Figure 28: 1Vs1 Commands Multiplayer

13 Final comments

This final project of the interactive graphics course entitled Fall-Guys helped us to improve our knowledge about the technologies used and to improve our ability to work as a team. The visual aesthetics of our Fall Guys project are undoubtedly eye-catching. The vibrant colors, playful character designs, and well-crafted environments capture the essence of the game. The diverse and challenging levels created add depth and excitement to the game. The incorporation of multiplayer functionality adds a competitive edge to the game, aligning it with the spirit of Fall Guys. It's an excellent feature that enhances the overall gameplay experience. The sound effects and music in the game contribute significantly to the immersive experience. They create an atmosphere that complements the gameplay and adds to the fun.

Figure 29: Final Comments