Interactive Graphics Report

PacTeam
Edoardo Piroli 1711234 - Luca Faraoni 1667345

22nd September 2019

Contents

1 — Introduction

For our final project, we have decided to build a 3D version of the traditional Pacman game using the following libraries: Three.js, threex.domevents.js, and OBJLoader.js.

1.1 How to play

The game presents itself with the camera placed and oriented as in a first person game. The goal is to eat all the balls in the maze, avoiding to be eaten by the ghosts. It is possible to move in each direction and to rotate the camera. The commands are the followings:

- 'W': move forward;
- 'A': move to the left;
- 'S': move backward;
- 'D': move to the right;
- $\leftarrow$: rotate the camera to the left;
- $\rightarrow$: rotate the camera to the right.

There are some bigger balls, the super-balls, which, if eaten, grant Pacman the ability to eat the ghosts for a certain amount of time. Both balls or ghosts can be eaten exclusively if they are in front of Pacman as that's where its mouth is located. Eating ghosts or balls increases the score based on the chosen difficulty. When touched by a ghost, Pacman dies and the game ends.

Before the start of the game, it is possible to change the difficulty in the settings menu, located at the top left corner. There are three different difficulty levels that change some game parameters, according to the following table.

Table 1.1: Difficulty levels parameters

Difficulty level	**N. of ghosts**	**Ghosts' spawn time**	**Super-ball time**	**Balls points**	**Ghosts points**
Easy	4	15s	20s	2	20
Medium	6	10s	15s	5	50
Hard	8	8s	10s	10	100

1.2 Browsers tested

All the aspects of the project have been tested on Google Chrome.

2 — Models

2.1 Maze

The maze is composed by a floor, many cubes representing the walls and also balls and super-balls. In order to build it as the original Pacman maze, we have used a matrix where each cell's value represents a different object, 1 for walls, 2 for balls, 3 for super-balls.

- *Floor*: the floor is a plane declared as follows

```
var floor = new THREE.Mesh(
    new THREE.PlaneGeometry(205, 211, 10, 10),
    new THREE.MeshPhongMaterial({map: textureFloor})
);

floor.rotation.x = Math.PI / 2;
floor.position.x = 100;
floor.position.z = -103;
```

- *Walls*: walls are formed by a unitary cube which is defined as follows

```
var unique_cube = new THREE.BoxBufferGeometry(5, 50, 5);
var unique_cube_material = new THREE.MeshPhongMaterial({map: textureWall};

//We use the same geometry and material to optimize rendering
var cube = new THREE.Mesh(
    unique_cube,
    unique_cube_material
);
```

  Each cube has its own position based on the cell it belongs to. Moreover there are 2 special cubes, which are defined as follows

```
var cube1 = new THREE.Mesh(
    new THREE.BoxBufferGeometry(5, 50, 5),
    new THREE.MeshPhongMaterial({map: textureTeleport})
);
var cube2 = new THREE.Mesh(
    new THREE.BoxBufferGeometry(5, 50, 5),
    new THREE.MeshPhongMaterial({map: textureTeleport})
);

cube1.position.set(0, 25, -110);
cube2.position.set(200, 25, -110);
```

  Such cubes, as described later, are used to create a portal as in the original game.

- *Balls*: balls are defined as follows

```
var unique_ball = new THREE.SphereBufferGeometry(0.75, 16, 16);
var unique_ball_material = new THREE.MeshPhongMaterial({color: 0xffff00};

//We use the same geometry and material to optimize rendering
var ball = new THREE.Mesh(
    unique_ball,
    unique_ball_material
);
```

  Each ball has its own position based on the cell it belongs to.

- *Super-balls*: super-balls are defined as follows

```
var unique_super_ball = new THREE.SphereBufferGeometry(1.5, 16, 16);
var unique_super_ball_material = new THREE.MeshPhongMaterial({color: 0xffd700};

//We use the same geometry and material to optimize rendering
var ball = new THREE.Mesh(
    unique_super_ball,
    unique_super_ball_material
);
```

  There are 4 super-balls positioned as in the original pacman game.

2.2 Pacman

Pacman is a 3D model imported as follows

```
var pacman_model;
var loader = new THREE.OBJJLoader();
loader.load(
    '3DModels/pacman.obj',
    (object) => {
        object.rotation.y = Math.PI * 50 / 126;
        object.rotation.z = 0.15;
        object.travers((child) => {
            if (child instanceof THREE.Mesh) {
                child.material.color.setHex(0xffff00);
            }
        });
        pacman_model = object;
        pacman_model.position.set(110, 6, -10);
    };
}
```

2.3 Ghosts

As for the ghosts, we load just one model and for every new ghost we apply to it one of four different colors chosen randomly: blue(0x00ffff), red(0xce3025), pink(0xffb8ff), and orange(0xffb852) as in the original game.

The ghosts' model is loaded as follows:

```
var ghost_model;

function loadGhost(loader) {
    loader.load(
        '3DModels/pacman_ghost.obj',

        (object) => {
            object.scale.set(4, 3, 4);
            ghost_model = object;
        }
    );
}

var loader = new THREE.OBJLoader();
loadghost(loader);
```

Finally every ghost model has 4 spherical objects associated to it as its children, these objects, leveraging the hierarchical structure of the model, move with the associated ghost and rotate, while oscillating up and down, around it.

Ghosts are automatically added to the environment via the following function:

```
function spawn() {
    if (n_ghosts < GHOSTS_MAX_NUMBER[difficulty_level]) {
        // Class described later
        var ghost = new Ghost();
        scene.add(ghost.ghost);
        scene.add(ghost.cube);
```

```
        // Arrays used to handle ghosts
        ghosts.push(ghost);
        ghosts_objects.push(ghost.cube);

        // Keeps track of the number of ghosts actually deployed
        n_ghosts++;
    }
}
```

The Ghost class is defined in the ghost.js file, it is too long to be reported but here is its constructor:

```
const POSSIBLE_GHOSTS_COLOR = [0xce3025, 0xffb852, 0x00ffff, 0xffb8ff];
const POSSIBLE_GHOSTS_POSITIONS = [new THREE.Vector3(70, 3, -130), new THREE.Vector3(130, 3,
    -130), new THREE.Vector3(70, 3, -90), new THREE.Vector3(130, 3, -90)];

constructor() {
       // It clones the pre-loaded model and assigns a color to it
        this.ghost = ghost_model.clone();
        var chosen_color = POSSIBLE_GHOSTS_COLOR[Math.floor(Math.random() *
            POSSIBLE_GHOSTS_COLOR.length)];

        this.ghost.traverse((child) => {
            if (child instanceof THREE.Mesh) {
                child.material = child.material.clone();
                child.material.color.setHex(chosen_color);
            }
        });

        this.actual_direction = 'down';

        // Transparent cube used to handle ghosts' collisions described later on
        this.cube = new THREE.Mesh(
            new THREE.BoxBufferGeometry(6, 6, 6),
            new THREE.MeshPhongMaterial()
        );
        this.cube.material.opacity = 0;
        this.cube.material.transparent = true;

        // Spheres associated to the ghost
        var sphere1 = new THREE.Mesh(
            new THREE.SphereBufferGeometry(0.1, 32, 32),
            new THREE.MeshPhongMaterial()
        );

        var sphere2 = new THREE.Mesh(
            new THREE.SphereBufferGeometry(0.1, 32, 32),
            new THREE.MeshPhongMaterial()
        );

        var sphere3 = new THREE.Mesh(
            new THREE.SphereBufferGeometry(0.1, 32, 32),
            new THREE.MeshPhongMaterial()
        );

        var sphere4 = new THREE.Mesh(
            new THREE.SphereBufferGeometry(0.1, 32, 32),
            new THREE.MeshPhongMaterial()
        );

        sphere1.position.x += this.radius;
        sphere2.position.x += this.radius;
        sphere3.position.x -= this.radius;
        sphere4.position.x -= this.radius;

        sphere1.position.z += this.radius;
        sphere2.position.z -= this.radius;
        sphere3.position.z += this.radius;
        sphere4.position.z -= this.radius;

        sphere1.name = 'sphere1';
        sphere2.name = 'sphere2';
        sphere3.name = 'sphere3';
        sphere4.name = 'sphere4';
```

```
        this.ghost.add(sphere1);
        this.ghost.add(sphere2);
        this.ghost.add(sphere3);
        this.ghost.add(sphere4);

        this.ghost.traverse((child) => {
            if (child instanceof THREE.Mesh) {
                child.material.color.setHex(chosen_color);
            }
        });

        // Attributes used to handle the movements of the spheres as described later on
        this.sphere_speed = Math.PI/80;
        this.current_angle = 0;
        this.radius = 1.5;

        this.y_speed = Math.PI/150;
        this.current_y_angle = 0;
        this.y_radius = 0.628;

       // Also the position of the ghost is randomly picked from a set of predefined options.
        var position = POSSIBLE_GHOSTS_POSITIONS[Math.floor(Math.random() *
            POSSIBLE_GHOSTS_POSITIONS.length)];
        this.ghost.position.set(position.x, position.y, position.z);
        this.cube.position.set(position.x, position.y, position.z);
    }
```

2.4 Play

In the homepage there is an object representing the text "PLAY". This object is clickable and makes the game start.

3 — Lights and Textures

3.1 Textures

In order to load the textures we have used THREE.TextureLoader(); There are different textures for the following objects:

- **Floor**: For the floor we have used a sandy dark texture;
- **Walls**: For the walls we have used a grass-like texture;
- **Portals**: For the portals we have used an image representing a spiral as texture.

3.2 Lights

As for the lights we have chosen to use a very soft ambiental light, with color 0xffffff and intensity 0.3. In the homepage, in order to better display the maze's structure we have added a strong directional light, with same color as the ambiental one and intensity 0.8 which is removed as the game starts(upon a click on the PLAY object). Furthermore we have added a spotlight which moves following pacman, as if he was carrying a torch. The lights are defined as follows

```
var ambientLight = new THREE.AmbientLight(0xffffff, 0.3);
var dirLight = new THREE.DirectionalLight(0xffffff, 0.8)
dirLight.position.set(100, 80, -50);
var spotLight = new THREE.SpotLight(0xffffff, 0.8, 200, Math.PI/4, 1, 2);
```

In order to handle the change in orientation of the spotlight, we have added, as its target an invisible object in front of Pacman.

4 — Animations

The only moving objects are ghosts and Pacman. Pacman is moved by the user through the commands we have described in the introduction. As for the ghosts they are moved autonomously around the maze, although they differ from the ghosts of the original game as they don't follow any pattern but move randomly. In particular ghosts may change directions whenever they find themselves in a cross-road.

4.1 Collisions

In order to handle Pacman's interactions with the environment we have used raycasting. We have defined a raycaster as

```
var raycaster = new THREE.Raycaster();
// Setting the far property to 3 because we are interested only in close collisions.
raycaster.far = 3;
```

In particular at every rendering we cast 8 different rays starting from Pacman's center in the 8 main cardinal-directions. For example here is the code to cast a ray straight in front of pacman

```
raycaster.set(
    pacman.position, new THREE.Vector3(
        Math.sin(-camera.rotation.y),
        0,
        -Math.cos(-camera.rotation.y)
    )
);
```

For every movement-command('W', 'A', 'S', 'D', $\rightarrow$, $\leftarrow$) we use the following directions to cast the rays:

- '**W**': We use the North, North-East and North-West directions;
- '**A**': We use the West, South-West and North-West directions;
- '**S**': We use the South, South-East and South-West directions;
- '**D**': We use the East, South-East and North-East directions;
- $\rightarrow$: We use the East and North-East directions;
- $\leftarrow$: We use the West and North-West directions.

Thus, for each collision, if the object is a:

- *wall*: we skip the command which yielded the collision;
- *portal*: Pacman is teleported to the opposite portal;
- *ball*: Pacman eats it(the ball is removed from the scene) and the score is increased. If a super-ball is detected Pacman also gains the super-ball power. Whenever a ball is eaten, if it is the last one, the game ends and Pacman has won.
- *ghost*: if Pacman has the super-ball power, it eats the ghost (which is removed from the scene), and the score is increased. Otherwise, Pacman dies and the game is over. To check for ghosts collisions, since collision with 3D imported objects can be quite computationally-expensive, we made a trick: each ghost carries an invisible cube which moves with it and we look for collisions using these cubes.

4.2 Ghosts Movement

4.2.1 Ghost Movement

To handle ghosts movement we have used the maze matrix. In particular, when in a new cell, the ghost checks for the available directions looking cells around itself. If there is a cross, where there are at least three possible directions, it randomly picks one and rotates. If not, the ghost keeps moving forward.

Here is the relative code to check for possible directions and for the ghosts movement.

```
moveGhost() {

    if (-this.ghost.position.z % 10 == 0 && this.ghost.position.x % 10 == 0) {
            this.ghost.rotation.y = this.get_rotation(-this.ghost.position.z/5, this.ghost.
                position.x/5);
    }

    this.ghost.position.x += 0.25*Math.sin(this.ghost.rotation.y);
    this.ghost.position.z += 0.25*Math.cos(this.ghost.rotation.y);

    this.cube.position.set(this.ghost.position.x, this.ghost.position.y, this.ghost.position.z
        );

    .
    .
    .
}

get_rotation(i, j) {
    var directions = [];
    if (maze[i+2][j] != 1 && i != 34) {
        directions.push('up');
    }

    if (maze[i-2][j] != 1) {
        directions.push('down');
    }

    if (maze[i][j-2] != 1 && i!=22) {
        directions.push('left');
    }

    if (maze[i][j+2] != 1 && i!=22) {
        directions.push('right');
    }

    if (directions.length == 2 && directions.includes(this.actual_direction)) return this.
        ghost.rotation.y;

    switch (directions[Math.floor(Math.random() * directions.length)]) {
        case 'up':
            this.actual_direction = 'up';
            return Math.PI;
        case 'down':
            this.actual_direction = 'down';
            return 0;
        case 'left':
            this.actual_direction = 'left';
            return -Math.PI/2;
        case 'right':
            this.actual_direction = 'right';
            return Math.PI/2;
    }
}
```

4.2.2 Sphere Movement

At every rendering the following instructions, as part of a method of the Ghost class, are executed in order to handle the movement of the balls for every deployed ghost.

```
this.current_angle += this.sphere_speed;
this.current_y_angle += this.y_speed;
```

```
for (var i=0; i < this.ghost.children.length; i++) {
    var obj = this.ghost.children[i];
    if (obj.name == 'sphere1') {
        obj.position.x = this.radius * Math.sin(this.current_angle);
        obj.position.z = this.radius * Math.cos(this.current_angle);
        obj.position.y = this.y_radius * Math.sin(this.current_y_angle);
    }
    else if (obj.name == 'sphere2') {
        obj.position.x = this.radius * Math.cos(this.current_angle);
        obj.position.z = -this.radius * Math.sin(this.current_angle);
        obj.position.y = -this.y_radius * Math.sin(this.current_y_angle);
    }
    else if (obj.name == 'sphere3') {
        obj.position.x = -this.radius * Math.cos(this.current_angle);
        obj.position.z = this.radius * Math.sin(this.current_angle);
        obj.position.y = -this.y_radius * Math.sin(this.current_y_angle);
    }
    else if (obj.name == 'sphere4') {
        obj.position.x = -this.radius * Math.sin(this.current_angle);
        obj.position.z = -this.radius * Math.cos(this.current_angle);
        obj.position.y = this.y_radius * Math.sin(this.current_y_angle);
    }
}
```

5 — Extra Stuff

5.1 Minimap

In order to improve the overall game experience, we have added a minimap in the top right corner. This minimap is represented in a second viewport, which displays a second orthographic camera point of view. The second camera is instantiated as follows

```
var cameraOrtho = new THREE.OrthographicCamera(-2.5, 202.5, 207.5, -2.5, -1000, 1000);
cameraOrtho.up = new THREE.Vector3(0, 0, -1);
cameraOrtho.lookAt(new THREE.Vector3(0, -1, 0));
```

In order to set up the minimap's viewport we have used the following instructions:

```
var insetHeight = window.innerHeight/3, insetWidth = window.innerWidth/3;
renderer.setViewport(0, 0, window.innerWidth, window.innerHeight);
renderer.render(scene, camera);
renderer.clearDepth();
renderer.setScissorTest(true);
renderer.setScissor(window.innerWidth-insetWidth-2, window.innerHeight - insetHeight,
    insetWidth-2, insetHeight);
renderer.setViewport(window.innerWidth-insetWidth-2, window.innerHeight - insetHeight,
    insetWidth-2, insetHeight);
renderer.render(scene, cameraOrtho);
renderer.setScissorTest(false);
requestAnimationFrame(animate);
```

5.2 Audio

We have set up a variety of different audio tracks for the different interactions with the environment. In the homepage it's possible to pause the main soundtrack and in the settings menu there are a volume slider, to adjust the volume accordingly, and a mute icon to disable every sound. The soundtrack only starts after hitting the PLAY object.

5.3 Manager

We have used a loading manager(new THREE.LoadingManager()) in order to first load everything, while displaying a loading bar, and only then set up the homepage.