Interactive Graphics - Cocomero Battle

Stefano Foti - 1914737

July 17, 2022

1 Introduction

This document has been produced together with the mini game "Cocomero Battle". The objective of the game is very simple: drive the helicopter in the sky and broke as many watermelons as possible throwing bullets, until there is health on the bar on the top right corner. The health bar starts with a value of 100.

Every time you throw a bullet and hit a watermelon, the counter of broken watermelons increases by 1. Every time you miss a watermelon, the health bar decreases by 2; every time you crash a watermelon with the helicopter, the health decreases by 7, but also the counter of broken watermelons increases by 1.

In any moment, on the top left corner you can see the current amount of broken watermelons, the latest counter value, and whether the current hit value is a record or not.

Figure 1: Helicopter. The color is customizable. The helicopter angle depends on the target position.

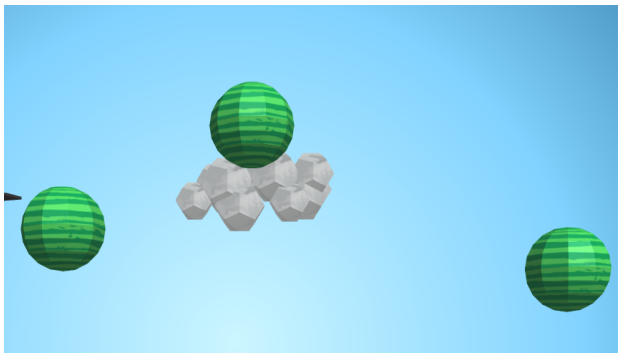


Figure 2: Watermelons. The size can be modified at startup. A texture is applied to get the watermelon appearance

2 Customization at startup

At application startup, an overlay will show a customization form. The customization form is very simple, and mostly made in HTML5 with bootstrap.

In this section, you can customize the following:

- Controls - You can choose wheter you prefer to control the helicopter target position with mouse pointer or WASD keyboard controls. In both cases you can throw bullets by left clicking or hitting the space key.
- Helicopter color - You can customize the helicopter color between green, orange and yellow.
- Watermelon size - You can set the size of watermelons. Depending on the size, of course also the radius used to understand if the bullet hit or not the watermelon will change.
- Speed - While you move the mouse, the helicopter will not get directly on the position. The position of the pointer will be used as a target position; the helicopter will reach the target position depending on the speed, that can be customized. The higher the speed, the lower the time to reach the target position. While reaching the target position the helicopter will slightly rotate on the three axis, simulating the behaviour of a flying helicopter.

- Difficulty - This parameter changes the amount of watermelons coming from the right side. The higher the amount, the harder the game.
- Music - To enable/disable music.
- Sounds - To enable/disable helicopter and crash sounds.
- Animations - To enable/disable animations

Figure 3: Home screen, with all available settings.

3 Architecture

The architecture of the project is very simple, and totally javascript based. I used javascript classes for all of the items (real or abstract) in the project, dividing it in several javascript files, mostly corresponding to one class. There are several other js files, a main one where the execution starts, a utils one with some common stuff and so on. I know that such an approach typically is not recommendable with javascript for performance reasons (I also included a really simple readiness strategy, to avoid loading issues), but I hope the code is much more clear and readable in this way. In any case it is possible pack and minify it. All of the custom js produced files are under the src directory. The libs directory contains all the downloaded js libraries, while the assets directory contains textures, sounds and so on. The root directory mostly keeps a stylesheet file and the index.html.

All of the graphics uses the three.js library, while the animations uses tween.js. There is no phisics engine used. I was evaluating the possibility to add it; the project as is runs smoothly on most devices, but I already observed some lags while tuning the animations. Adding a phisics engine could make it heavier, and might not necessarily be a good idea seen the topic. Anyway I think it is somehow realistic.

There are several other libraries and included stuff in the index, let's see all of them:

- three.js - Core graphics library
- tween.js - Library used to animate explosions, helicopter wake, exhaust gas;
- howler.js - Library used to play music and sounds during the game;
- Web commons
  - Bootstrap - Framework used to show health and stats during the game and the controls panel at startup/rematch;
  - jQuery - To update the DOM elements and read selected values;
  - Google fonts

4 Helicopter

The helicopter is a hierarchical model made of common 3D shapes. The core of the helicopter is the cabin. To the cabin I attached windows, tail, basement, helix and so on. The main helix and the second one rotates; depending on the speed of the helicopter, the speed of helix changes. Depending on the distance between the helicopter and target position, the helicopter itself rotates of a certain angle.

This is the hierarchical model of the helicopter; it's not the only hierarchical model in the project, but for sure is the most complex of the lot.

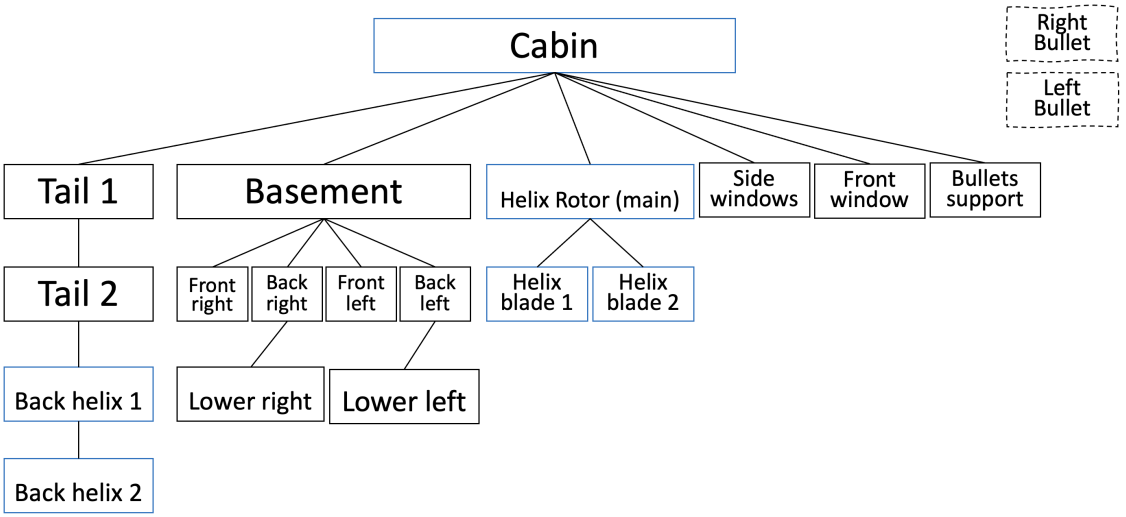


Figure 4: Hierarchical model of the helicopter. Blue bordered items are animated (main helix and tail helix rotates, while the whole helicopter changes orientation depending on the delta between current and target position. Bullets can be added or not to the helicopter mesh, and that's why they have a dotted border in the figure.

5 Bullets

By pressing space or left clicking, the helicopter will throw a bullet (if any available). Each bullet is again a hierarchical model, but I'll not dive into details that much.

Until a bullet is launched, it's included into the main helicopter mesh. When you launch it, it gets removed from the helicopter mesh, and a new one is added to the scene; at every scene update, the bullets keeps increasing its x value. At the same time, right before the bullet a sort of bullet wake will appear, until the bullet hits a watermelon or gets out of the screen. As soon as one of those conditions is satisfied, the bullet is attached again to the helicopter mesh, the "thrown" flag is set to false and the bullet is ready to be launched again.

```
launch(i) {
    bullets[i].m.position.x = helicopter.m.position.x;
    bullets[i].m.position.y = helicopter.m.position.y - 2.5;
    i%2 == 0 && (bullets[i].m.position.z = helicopter.m.position.z + 10);
    i%2 == 1 && (bullets[i].m.position.z = helicopter.m.position.z - 10);
    bullets[i].thrown = true;
    bullets[i].boom = false;
    bullets[i].pos = i;
    scene.add(bullets[i].m);
    i%2 == 0 && helicopter.m.remove(helicopter.bulletL);
    i%2 == 1 && helicopter.m.remove(helicopter.bulletR);
}
```

Figure 5: Function invoked to launch a bullet. Sets the thrown flag, resets the boom flag (true if the bullet took a watermelon) and removes the related bullet from the helicopter.

```
move(i) {
    let bullet = bullets[i];
    if(bullet.thrown){
        bullet.m.position.x += bullet.speed;
        let position = bullet.m.position.clone();
        position.x -= 10;
        effectsOn && new EffectsUtils().init(position, EFFECTS.BULLET_LAUNCH);
        if (bullet.m.position.x > 250) {
            scene.remove(bullet.m);
            i%2 == 0 && helicopter.m.add(helicopter.bulletL);
            i%2 == 1 && helicopter.m.add(helicopter.bulletR);
            bullet.thrown = false;
        }
    }
}
```

Figure 6: Function invoked in the render loop, to move a bullet once thrown. When the bullet x gets too high, the bullet is added back to the helicopter.

6 Watermelons

Each watermelon is built by calling the constructor of the class, and then added to the array holding all the active objects. The higher the difficulty, the more the calls to Melon() constructor. The Melons class manages the initialization and restoring to the desired amount.

```
class Meloon {
    constructor() {
        const g = new THREE.SphereGeometry(obstacleSize, obstacleSize, obstacleSize);
        const texture = new THREE.TextureLoader().load( "assets/texture_wm.jpg" );
        texture.wrapS = THREE.RepeatWrapping;
        texture.wrapT = THREE.RepeatWrapping;
        texture.repeat.set( 1, 1 );
        const mat = new THREE.MeshPhongMaterial({
            map: texture,
            flatShading: true
        });
        this.m = new THREE.Mesh(g, mat);

    }
}
```

Figure 7: Meloon class. The constructor initializes a single watermelon.

```
class Meloons {
    constructor() {
        this.m = new THREE.Object3D();
    }

    initMeloons() {
        for (let i = 0; i < difficulty; i++) {
            if(meloons.length >= difficulty) {
                break;
            }
            let meloon = new Meloon();
            meloon.m.position.y = 50 + getRandomInt(0, 80);
            meloon.m.position.x = 200 + getRandomInt(0, 60);
            this.m.add(meloon.m);
            meloons.push(meloon);
        }
    }
```

Figure 8: Part of the meloons class. The initMeloons() initializes/restores the watermelons.

The meloons class, in the homonym file, has the ownership of managing the moviment of watermelons (collapsed lines), and detecting both bullets hit and collisions with the helicopter. The hits are computed as below, depending on the delta between current bullet and current watermelon. In case of hit, an explosion animation will start (if animations enabled) using the custom class EffectUtils, the bullet and the melon gets removed from the scene, the thrown flag gets a reset and the bullet is added again to the helicopter in proper position. The hit counter will increase by one, using the game.newHit by the game class. Also a sound will be played. In case of crash, indeed, a different explosion animation will start (with the same color of the helicopter), and the health will decrease by the HEALTH.CRASH constant. Also in this case, a crash sound will be played by the game class and the hit counter will increase by one. In case of miss, the melon will be removed from the scene and health reduced by the constant below.

```
for(let i = 0; i < meloons.length; i++) {
    let meloon = meloons[i];

    if(!meloon.m.isMoving){···
    }
    bullets.filter(b => b.thrown).forEach(bullet => {
        let diff = bullet.m.position.clone().sub(meloon.m.position.clone());
        let dBullet = diff.length();
        if (dBullet < 2*obstacleSize) {
            effectsOn && new EffectsUtils().init(meloons[i].m.position, EFFECTS.EXPLOSION);
            this.m.remove(meloons[i].m);
            meloons.splice(i, 1);
            scene.remove(bullet.m);
            bullet.thrown = false;
            bullet.pos%2 == 0 && helicopter.m.add(helicopter.bulletL);
            bullet.pos%2 == 1 && helicopter.m.add(helicopter.bulletR);
            game.newHit();
        }
    })
```

Figure 9: How an hit is managed

```
    let heDiff = hPos.sub(ePos).length();

    if (heDiff < 2*obstacleSize) {
        // meloon crash
        effectsOn && new EffectsUtils().init(meloon.m.position, EFFECTS.HELI_CRASH);
        this.removeMeloon(i);
        health.reduceHealth(HEALTH.CRASH);
        game.newHit();
    } else if (meloon.m.position.x < -250) {
        // melon out
        health.reduceHealth(HEALTH.MISS);
        this.removeMeloon(i);
    }
}
```

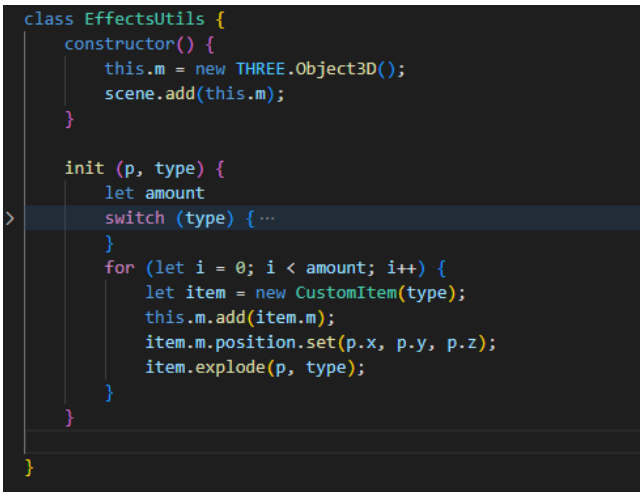
Figure 10: How crash and miss are managed

7 Animations

All animations are managed by the EffectsUtils class, as below. The class inits a certain amount of "CustomItems", that are basic 2D/3D shapes, moved with the library tween.js. The constructor inits the a basic 3D object. In the init() function, invoked to start an animation, I create as many CustomItems as tuned depending on the event type. The constructor of every CustomItem triggers the creation of a simple 2D/3D shape.

```
class CustomItem {
    constructor(type) {
        let g;
        switch (type) {
            case EFFECTS.EXPLOSION:
                g = new THREE.IcosahedronGeometry(10, 0);
                break;
            case EFFECTS.BULLET_LAUNCH:
                g =  new THREE.PlaneGeometry( 5, 3 )
                break;
            case EFFECTS.HELI_MOVING:
                g =  new THREE.PlaneGeometry(4, 2)
                break;
            case EFFECTS.HELI_CRASH:
                g =  new THREE.IcosahedronGeometry(10, 0)
                break;
            default:
                return;
        }
        let mat = new THREE.MeshPhongMaterial({
            specular: 0xffffff,
            flatShading: true
        });
        this.m = new THREE.Mesh(g, mat);
    }
```

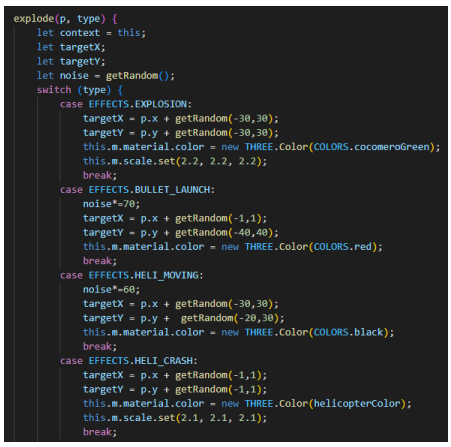
Figure 11: CustomItem class



```
class EffectsUtils {
    constructor() {
        this.m = new THREE.Object3D();
        scene.add(this.m);
    }

    init (p, type) {
        let amount
        switch (type) {···
        }
        for (let i = 0; i < amount; i++) {
            let item = new CustomItem(type);
            this.m.add(item.m);
            item.m.position.set(p.x, p.y, p.z);
            item.explode(p, type);
        }
    }

}
```

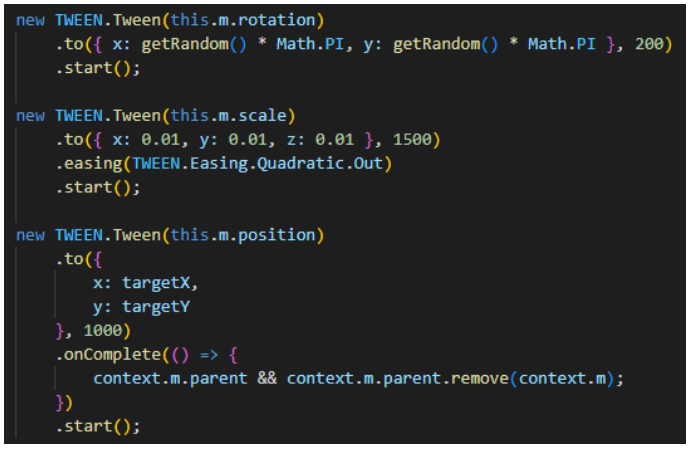
Figure 12: EffectsUtils class

For each item, I invoke its explode() function, with some slight difference depending on the event type, asking tween.js to gently change parameters like scale and position (wrt the starting one).



```
explode(p, type) {
    let context = this;
    let targetX;
    let targetY;
    let noise = getRandom();
    switch (type) {
        case EFFECTS.EXPLOSION:
            targetX = p.x + getRandom(-30,30);
            targetY = p.y + getRandom(-30,30);
            this.m.material.color = new THREE.Color(COLORS.cocomeroGreen);
            this.m.scale.set(2.2, 2.2, 2.2);
            break;
        case EFFECTS.BULLET_LAUNCH:
            noise*=70;
            targetX = p.x + getRandom(-1,1);
            targetY = p.y + getRandom(-40,40);
            this.m.material.color = new THREE.Color(COLORS.red);
            break;
        case EFFECTS.HELI_MOVING:
            noise*=60;
            targetX = p.x + getRandom(-30,30);
            targetY = p.y +  getRandom(-20,30);
            this.m.material.color = new THREE.Color(COLORS.black);
            break;
        case EFFECTS.HELI_CRASH:
            targetX = p.x + getRandom(-1,1);
            targetY = p.y + getRandom(-1,1);
            this.m.material.color = new THREE.Color(helicopterColor);
            this.m.scale.set(2.1, 2.1, 2.1);
            break;
```

Figure 13: explode() function



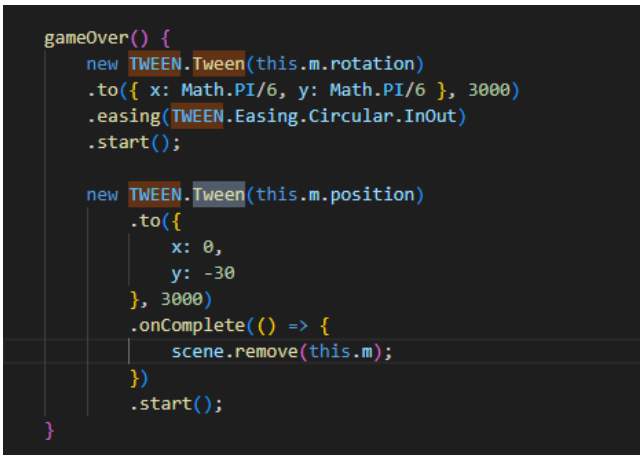
```
new TWEEN.Tween(this.m.rotation)
    .to({ x: getRandom() * Math.PI, y: getRandom() * Math.PI }, 200)
    .start();

new TWEEN.Tween(this.m.scale)
    .to({ x: 0.01, y: 0.01, z: 0.01 }, 1500)
    .easing(TWEEN.Easing.Quadratic.Out)
    .start();

new TWEEN.Tween(this.m.position)
    .to({
        x: targetX,
        y: targetY
    }, 1000)
    .onComplete(() => {
        context.m.parent && context.m.parent.remove(context.m);
    })
    .start();
```

Figure 14: Tween usage

Tween.js is used also in the meloon class to move watermelons or at the end of the game, to simulate the helicopter falling down.

```
        meloon.m.isMoving = true;
        let oldX = meloon.m.position.x;
        let oldY = meloon.m.position.y;
        var cx = oldX - getRandomInt(20*difficulty/2,60*difficulty/2);
        var cy;
        if(oldY < 100) {
            cy = getRandomInt(oldY, 200)
        } else {
            cy = getRandomInt(0, oldY)
        }
        new TWEEN.Tween(meloon.m.position)
        .to({x:cx, y:cy}, 2000)
        .onComplete(() => {
            meloon.m.isMoving = false;
        })
        .start()
```

Figure 15: Tween usage for meloon moving.



```
gameOver() {
    new TWEEN.Tween(this.m.rotation)
    .to({ x: Math.PI/6, y: Math.PI/6 }, 3000)
    .easing(TWEEN.Easing.Circular.InOut)
    .start();

    new TWEEN.Tween(this.m.position)
        .to({
            x: 0,
            y: -30
        }, 3000)
        .onComplete(() => {
            scene.remove(this.m);
        })
        .start();
}
```

Figure 16: Tween usage in the gameOver function of helicopter class

8 Sky and clouds

The light blue gradient background is added to the page in the custom css file. The sky class has the ownership of managing the clouds. The clouds have no real function; in my original idea the helicopter had to avoid not only watermelons but also cloud, but then I just keep them there. Another texture is applied to clouds.

9 Health

Once the health ends, the game is over. The Health class manages the health, exposing the reduceHealth and restore functions. The first one reduces of the amount in input the health value, and ends the game when the health is over. The second one resets the health to 100, in case of new play. Both functions in the end call the updateProgressBar() function, that uses jQuery to update the progress bar in the DOM. The progress bar changes the color depending on the remaning health amount, replacing preposed bootstrap classes. The progress bar on the right top corner in fact is a bootstrap component.

10 Game

The game class manages the game status. Starts the game with the start() function, exposes the newHit() function already discussed, manage sounds, some DOM items and so on. At the end of the game (and so when health gets 0) the end() function of game class is called. The game goes to full screen at this stage.

```
start() {
    openFullscreen();
    $("body").addClass("playing")
    if(soundsOn) {···
    } else {···
    }

    if(musicOn) {···
    } else {···
    }

    this.started = true;
    $("#controls").hide();
    health.restore();
    this.restart();
    scene.add(helicopter.m);
}
```

Figure 17: Tween usage for meloon moving.

```
end() {
    $("body").removeClass("playing")
    this.started = false;
    helicopter.gameOver();
    scene.remove(bullets[0].m);
    scene.remove(bullets[1].m);
    meloonsObj.removeMeloons();
    helicopterSound && helicopterSound.stop();
    musicSound && musicSound.stop();
    $("#controls").show();
}
```

Figure 18: Tween usage in the gameOver function of helicopter class

11 Main

The main.js file starts the execution of the game. The first function called is the init() function, that will perform initialization stuff and enter the render loop. The init function is called just at startup, then in case of rematch it's not recalled from scratch.

```
function init() {
    initScene();
    initCamera();
    initRenderer();
    initListeners();
    initControls();
    initLights();
    initMeloons();
    initHealth();
    initClouds();
    initBullets();
    initGame();
    loop();
}
```

Figure 19: Starting point

```
function loop() {
    sky.moveClouds();
    if (game.started) {
        updatePosition();
        helicopter.update();
        bullets[0].thrown && bullets[0].move(0);
        bullets[1].thrown && bullets[1].move(1);
        meloonsObj.initMeloons();
        meloonsObj.move();
    }
    TWEEN.update();

    webGlRenderer.render(scene, camera);
    requestAnimationFrame(loop);
}
```

Figure 20: Render loop

12 Sounds

Sounds have been added with the howler.js library, as follows.

```
if(soundsOn) {
    helicopterSound = new Howl({
        src: ['assets/helicopter.mp3'],
        loop: true,
        volume: 0.6,
    });
    this.explSound = new Howl({
        src: ['assets/explosion.wav'],
        volume: 0.8,
    });
    helicopterSound.play();
} else {
    helicopterSound && helicopterSound.stop();
}
```

Figure 21: Sounds included with howler.js library; exaple of helicopter sound (reproduced in loop) and explosion sound, instantiated but started when required.