CPS3231 Computer Graphics
Lecturers: Dr. Sandro Spina and Dr. Keith Bugeja

Report about the classic Arcade Game Asteroid

Autumn Semester 2023

Massimiliano Nunzio Vella
Rankhofstrasse 13
6006 Luzern

15.02.2024
Number of words: 6’335

Table of Content

Acknowledgments:

Firstly, I want to acknowledge the support of my supervisors, namely Dr. Sandro Spina and Dr. Keith Bugeja, which have guided me throughout the CPS3231 Computer Graphics course. They have provided me with the course material, as well as supported me throughout the course to make this project possible.

Introduction

The following report will give a detailed analysis on my code for the old arcade game ‘Asteroids’ in a more modern context. Asteroids is a space-shooter with the main goal of survival and achieving the highest score of each player that played the game. The player will have to navigate a spaceship and shoot two types of enemy classes: asteroids and saucers. Asteroids come in three different forms: large, medium, and small. The saucers are enemy spaceships that target the players spaceship and try to shoot it with their lasers. Saucers also come in the sizes small and large. Hence, the gameplay appears to be endless, and the game can only be ended when leaving the game or losing all the life points.

The aim of this project is to apply a new aesthetic to this retro arcade game by using modern features, like textures, lights, etc., while staying true to the core principles of the game.

Task 1 – Game Logic (35%)

The game logic is the core of the entire game, as it supplies the means to play the game. The key aspects of this game are the controls, spaceship properties, asteroid properties, and saucer properties. Another key aspect is the game progression.

Player Controls

The player controls revolve around rotating the spaceship and moving it forward. To achieve this behaviour, I have used key listeners in combination with a class called ‘Spaceship’ that properly interprets the keys pressed by the player. Firstly, I have set up the key listeners in the script.js file in the following way:

```
//track key inputs and make array out of it
const keys = {}
window.addEventListener('keydown', (e) => {
    keys[e.code] = true
})

window.addEventListener('keyup', (e) => {
    keys[e.code] = false
})
```

The pressed keys are then stored in an array called keys, which is then further used to translate the keys into the necessary movement. Then, I have implemented the class ‘Spaceship’ with distinct qualities:

```
//class for spaceship
export class Spaceship {
    constructor (x, y, angle, speed) {
        this.x = x
        this.y = y
        this.angle = angle
        this.speed = speed
        this.hyperDriveCD = 0
        this.lives = 3
        this.lasers = []

        this.lasersCountID = 0  //use this as ID for the lasers. Just increment by
one any time you create a laser object.
    }
```

The Spaceship class takes x, y, angle, and speed as an argument, so that the spaceship can be properly exist, which will then be steered through the key listeners.

The movement of the spaceship was defined through a WASD pattern and implemented within in the move function. Furthermore, I have implemented a constraint that the spaceship cannot move when the gameState is the ‘GameOverMode’.

```
move (keys) {
if (gameState !== 'GameOverMode') {
        const rotationSpeed = 0.05
        if (keys['KeyW']) {
            const angle = this.angle
            const deltaX = -this.speed * Math.sin(angle)
            const deltaY = this.speed * Math.cos(angle)

            this.x += deltaX
            this.y += deltaY
            // wrap around the screen
            if (this.y > verticalScreenLimit || this.y < verticalScreenLimit * -1)
{
                this.y *= -1
            }
            if (this.x > horizontalScreenLimit || this.x < horizontalScreenLimit *
-1) {
                this.x *= -1
            }
        }
        //hyperdrive implementation
        if (keys['KeyS']) {
            if (this.hyperDriveCD === 0) {
                placeRandomly(this)
                this.hyperDriveCD = 30
                this.countdownCooldown()
            }
        }
        if (keys['KeyA']) {
            this.angle += rotationSpeed
        }
        if (keys['KeyD']) {
            this.angle -= rotationSpeed
        }
       //rest of the code
```

The W key was used as the forward option for the ship. The forward movement was defined by using the angle in combination with the speed to define the deltaX and deltaY variable, which then were added to the current x and y position of the spaceship. Through the addition of the delta variables a smooth movement of the spaceship is supported. Furthermore, when the spaceship would go over the screen limits, the spaceship wraps around the screen and appears on the other side of it through the invers of the x or y coordinate. The A and D key were used to change the angle of the spaceship, which then results in the turning of the spaceship. A fixed rotationSpeed variable was used to clearly define the speed in which the spaceship can turn on either side.

Furthermore, a hyperspace jump feature was implemented in form of the S key.

```
//previous code
        if (keys['KeyS']) {
            if (this.hyperDriveCD === 0) {
                placeRandomly(this)
                this.hyperDriveCD = 30
                this.countdownCooldown()
            }
        }
//following movement code
   countdownCooldown () {
        // Check if the hyperDriveCD is greater than 0
```

```
        if (this.hyperDriveCD > 0) {
            // Create an interval that decreases the hyperDriveCD every second
            const intervalId = setInterval(function () {
                this.hyperDriveCD--

                // Check if the countdown has reached zero
                if (this.hyperDriveCD === 0) {
                    // Stop the interval when the countdown reaches zero
                    clearInterval(intervalId)
                    console.log('Hyperdrive is ready!')
                }
            }, 1000) // Interval set to 1000 milliseconds (1 second)
        }
    }
}
```

The S key triggers the hyperspace jump feature that aims to randomly place the spaceship within the bounds of the screen. Furthermore, there is a set cooldown on the jump of 30 seconds that is monitored by the countdownCooldown function that decrements the hyperdrive periodically. The placeRandomly function is located within the placeRandomly.js file and it works as follows:

```
export const placeRandomly = function (object) {
    //Math.floor(Math.random() * (max - min + 1) + min)
    const horizontalMax = horizontalScreenLimit
    const horizontalMin = horizontalScreenLimit * -1
    const randomNrHorizontal = Math.floor(Math.random() * (horizontalMax -
horizontalMin + 1) + horizontalMin)
    const verticalMax = verticalScreenLimit
    const verticalMin = verticalScreenLimit * -1
    const randomNrVertical = Math.floor(Math.random() * (verticalMax - verticalMin
+ 1) + verticalMin)

    object.x = randomNrHorizontal
    object.y = randomNrVertical
    return object
}
```

This function takes in an object and places it within the limits of the screen. I defines the maximum and minimum for the horizontal and vertical space of the screen and defines random numbers within these limits. These numbers then overwrite the object's x and y positions, and the function then returns the object. Through this method a clearly regulated hyperspace jump is possible without getting out of bounds of the screen limits.

Lastly, for the controls of the spaceship, the shooting of the lasers must be possible. This was achieved through pressing the SPACE key.

```
if (keys['Space']) {
            if (isReady) {
                isReady = false
                console.log('Laser Count ID = ' + this.lasersCountID)
                this.lasers.push(new Laser(this.x, this.y, this.angle, 'laser' +
this.lasersCountID))
                this.lasersCountID++
                console.log('Added new Laser. Laser Count ID = ' +
this.lasersCountID)
                setTimeout(() => {
                    isReady = true
                }, 500)
```

```
            }
        }
//File: Laser.js
export class Laser {
    constructor (x, y, angle, name) {
        this.x = x
        this.y = y
        this.angle = angle
        this.speed = 0.1
        this.radius = 0.3

        this.name = name      //name of the laser .... use this to remove from
scene graph
        this.spawned = false  //use this to determine if object is attached to
scene graph
        this.hasHit = false
    }
```

The shooting of the lasers is bound to the isReady variable that is set to true to begin with. If this variable has the value true a new laser is pushed to the spaceship.laser array. The laser is defined through the Laser class that uses the spaceship as its origin, as it uses its x, y position, and angle. The further logic regarding the lasers will be shown within the rendering aspect of the asteroids game. Each laser has a laserCountID that is later used during the rendering of the lasers. Furthermore, the value of the isReady variable is then set to false. After 0.5 seconds the isReady variable is then again set to true, so the next laser can be shot by the spaceship. I have implemented the setTimeout method, so the player cannot spam laser shots and randomly hit everything in their way but rather must shoot methodically, which then makes the game a little bit harder and more enjoyable.

Enemy Classes

The enemy classes are the main competition within the game loop for the player. Throughout the game more enemies spawn that must be defeated, so that the player can reach a greater high score. Each enemy class has its distinct features that will be explored in the following passages.

Asteroids

Within the script.js file, where the main and render loop are placed, multiple global variables have been instantiated that store relevant information which are used by the asteroid file but also the script file for further manipulation. These are the respective global variables:

```
let largeAsteroids = []
let mediumAsteroids = []
let smallAsteroids = []
let amountToSpawn = 4 //indicates how many asteroids to spawn
```

I have created three different arrays for each kind of asteroid that can be created. These arrays are used to further manipulate its values and control interactions like collisions and hit

detections. Furthermore, the variable amountToSpawn is set to 4 so that the requirement of each wave having n + 4 numbers of asteroids can be applied more easily. This spawning behaviour has been established through the initNewWave function:

```
function initNewWave () {
    if (checkArrayLengths()) {
        waveNr++
        amountToSpawn++
        spawnAsteroids(amountToSpawn, largeAsteroids, mediumAsteroids,
smallAsteroids)
    }
}
```

Here the amountToSpawn variable is increased as each wave is destroyed, thus, following the n + 4 spawning requirements.

However, the asteroids and its properties are defined in the Asteroids.js file. Firstly, a superclass called Asteroid has been defined, as it implements all the properties each asteroid subclass uses.

```
class Asteroid {
    constructor () {
        this.x = getRandomNrHorizontal()
        this.y = getRandomNrVertical()
        this.spawned = false
        this.isHit = false //shows if the asteroid is hit --> easier to remove then
from the array and scene graph
        asteroidCountID++
    }
}
```

Yet again, the x and y coordinates are assigned through a method that assigns a random position within the screen limits. Also, the variables spawned and isHit are set to false, as they are used for future rendering of the asteroids and hit detection.

This superclass is then inherited by these subclasses:

```
export class LargeAsteroid extends Asteroid {
    constructor () {
        super()
        this.type = 'large'
        this.angle = 3
        this.radius = 1.5
        this.speed = 0.005
        this.name = 'largeAsteroid' + asteroidCountID
        this.value = 20
        asteroidCountID++
    }
}
export class MediumAsteroid extends Asteroid {
    constructor () {
        super ()
        this.type = 'medium'
        this.angle = 2
        this.radius = 1.2
        this.speed = 0.008
        this.name = 'mediumAsteroid' + asteroidCountID
        this.value = 50
        asteroidCountID++
    }
}
export class SmallAsteroid extends Asteroid {
    constructor () {
        super ()
        this.type = 'small'
        this.angle = 1
        this.radius = 0.6
        this.speed = 0.015
        this.name = 'smallAsteroid' + asteroidCountID
        this.value = 100

        asteroidCountID++
    }
}
```

The subclasses then extend the superclass in the following way. They define a type for each asteroid, a distinct angle, radius, speed, and value. Furthermore, they have a distinct name with an ID that gets incremented each time they are spawned, which is then used to remove it from the scene graph, as well as the array.

Furthermore, the spawning and movement are controlled through functions.

```
export const spawnAsteroids = function (amount, largeAsteroids, mediumAsteroids,
smallAsteroids, scene) {
    for (let i = 0; i < amount; i++) {
        if (i % 3 === 0) {
            largeAsteroids.push(new LargeAsteroid())
        } else if (i % 3 === 1) {
            mediumAsteroids.push(new MediumAsteroid())
        } else {
            smallAsteroids.push(new SmallAsteroid())
        }
    }
}
export const moveAsteroids = function (asteroids) {
    for (let i = 0; i < asteroids.length; i++) {
        const angle = asteroids[i].angle
        const deltaX = asteroids[i].speed * Math.sin(angle)
        const deltaY = -asteroids[i].speed * Math.cos(angle)
```

```
        asteroids[i].x += deltaX * Math.cos(angle)
        asteroids[i].y += deltaY * Math.sin(angle)
        //invert the position so it wraps around the screen
        if (asteroids[i].y > verticalScreenLimit || asteroids[i].y <
verticalScreenLimit * -1) {
            asteroids[i].y *= -1
        }
        if (asteroids[i].x > horizontalScreenLimit || asteroids[i].x <
horizontalScreenLimit * -1) {
            asteroids[i].x *= -1
        }
    }
}
```

In the spawnAsteroids function, a for loop is used to push multiple asteroids until the declared amount is fulfilled. Furthermore, the modulo of i % 3 is used to define which kind of asteroid should be spawned. The moveAsteroid function iterates over the asteroid array and uses the same logic for a smooth movement for the asteroids. This method is then used when rendering the asteroids. Another key feature regarding the asteroids is the splitting into two smaller kinds of asteroids, when large and medium ones are hit. In the main function of the script.js file in lines 529 to 553, I have implemented the following function:

```
const handleLaserHitDetection = function (laser, targets) {
            for (let target of targets) {
                target.isHit = laserHit(laser, target)
                if (target.isHit) {
                    const targetIndex = targets.indexOf(target)
                    targets.splice(targetIndex, 1)

                    // If the target is a large or medium asteroid, split it into
two smaller ones
                    if (target instanceof LargeAsteroid) {
                        mediumAsteroids.push(new MediumAsteroid())
                        mediumAsteroids.push(new MediumAsteroid())
                    } else if ( target instanceof MediumAsteroid) {
                        smallAsteroids.push(new SmallAsteroid())
                        smallAsteroids.push(new SmallAsteroid())
                    }

                    const targetNode = scene.findNode(target.name)
                    if (targetNode) {
                        scene.removeNode(targetNode.name)
                    } else {
                        console.error('node not found')
                    }
                }
            }
        }
```

This function uses the laser that is shot and compares it to the target's location. When the target is hit the target is removed from its array. After the removal the target tested in a way where it is either a LargeAsteroid, MediumAsteroid, or none of these two. If it is one of the two asteroid kinds, two new asteroids are put into the array and then spawned within in the render loop.

Saucers

Again, multiple global variables for the saucers have been placed in the script.js file. These again store either the different saucer kinds or the lasers of these saucers.

```
let largeSaucers = []
let largeSaucerLasers = []
let smallSaucers = []
let smallSaucerLasers = []
```

For the two kinds of saucers, I have again set up a superclass called Saucer that supplies all necessary properties that are shared between the saucers.

```
class Saucer {
    constructor () {
        this.x = getRandomNrHorizontal()
        this.y = getRandomNrVertical()
        this.isHit = false //shows if the asteroid is hit --> easier to remove then
from the array and scene graph --> when spawned also false
    }
}
```

The subclasses then again extend the superclass. They use the variable type to declare the differences between the two saucers. Furthermore, a distinct angle, radius, speed, and value is again assigned.

```
export class largeSaucer  extends Saucer {
    constructor () {
        super()
        this.type = 'large'
        this.speed = 0.05
        this.radius = 0.8
        this.name = 'largeSaucer' + saucerCountID
        this.value = 200

        saucerCountID++
    }
}

export class smallSaucer extends Saucer {
    constructor () {
        super()
        this.type = 'small'
        this.speed = 0.08
        this.radius = 0.4
        this.name = 'smallSaucer' + saucerCountID
        this.value = 1000

        saucerCountID++
    }
}
```

Again, the spawning and movement are defined within functions, which results in a more modular code.

```
export const spawnSaucers = (largeSaucers, smallSaucers, highscore) => {
    const randomNumber = Math.random()

    if (randomNumber > 0.75) {
        if (highscore < 10000) {
            const saucer = new largeSaucer()
            largeSaucers.push(saucer)
        } else {
            const saucer = new smallSaucer()
            smallSaucers.push(saucer)
        }
    }
}
```

The spawning of the saucers requires that before reaching 10’000 points in the high score, only large saucers are spawned if a random number between 0 and 1 is greater than 0.75. If the high score is bigger than 10’000 only small saucers will be spawned.

The movement of these saucers is just a simple lateral movement. However, it is supplemented by other helper methods. One that stops the saucers from going over the screen limits and one that initiates the shooting of their lasers.

```
export const moveSaucers = function (saucer, spaceship, saucersLasers) {
    changeOrientation(saucer)
    saucer.x += saucer.speed
    if (saucer instanceof largeSaucer) {
        saucersShoot(saucer, saucersLasers)
    } else {
        smallSaucersShoot(saucer, spaceship, saucersLasers)
    }
}
```

The change in orientation is simple, as when it reaches the screen limits the invers is applied to the x-coordinate. The shooting, however, is dependent on the saucer and if it is either a large or a small saucer. Thus, when it is a large saucer the method saucerShoot is triggered, while when it is a small saucer is the smallSaucersShoot method is triggered.

```
export const saucersShoot = function (saucer, saucersLasers) {
    // Check if the saucer should shoot
    if (shouldShoot()) {
        // Choose a random location on the map for the laser
        const targetX = Math.random() * horizontalScreenLimit * 2 -
horizontalScreenLimit
        const targetY = Math.random() * horizontalScreenLimit * 2 -
horizontalScreenLimit

        // Calculate angle towards the target
        const angle = Math.atan2(targetY - saucer.y, targetX - saucer.x)

        // Create and add a new laser to the global array
        const newLaser = new Laser(saucer.x, saucer.y, angle, 'EvilLaser' +
laserCountID)
        saucersLasers.push(newLaser)

        laserCountID++
    }
}
```

```
export const smallSaucersShoot = function (saucer, spaceship, saucersLasers) {
    // Check if the saucer should shoot
    if (shouldShoot()) {
        // Calculate angle towards the spaceship
        const angle = Math.atan2(spaceship.y - saucer.y, spaceship.x - saucer.x)

        // Create and add a new laser to the global array
        const laser = new Laser(saucer.x, saucer.y, angle, 'EvilLaser' +
laserCountID)
        saucersLasers.push(laser)

        laserCountID++
    }
}
```

The saucersShoot function defines the angle of the laser as random, while the smallSaucersShoot function aims at the spaceship. The Math.atan2 function is used to define an angle from the x axis to a point. Here, the two points that define the angle are calculated through the spaceship and the saucer that is supposed to shoot. Another key aspect is the shouldShoot function that is used to check whether a saucer is supposed to shoot. It was required that every two seconds a fair Bernoulli trial takes place so that a laser is shot.

```
export const shouldShoot = function () {
    // Check if it's been at least 2 seconds since the last shoot
    if (Date.now() - lastShootTime > 2000) {
        lastShootTime = Date.now() // Update the last shoot time
        const randomNumber = Math.random()
        if (randomNumber < BERNOULLI_CONSTANT) {
            return true
        }
    }
    return false
}
```

However, I have encountered an issue with the tracking of the spaceship, which results in the saucers not being able to shoot at the spaceship directly. Thus, the lasers shoot at random again, therefore, this function does not work as intended. Furthermore, when spawning large and small saucers and having them shoot their lasers, only one of each kind shoots. For example, only one of the large saucers' shoots, while the others do not. Therefore, the shooting needs improvement, which was not possible in this time frame.

Collision Detection and Response

Collision Detection

To increase the modularity of my code, I have decided to place the collision detection methods into the collisionDetection.js file. In that file, I have defined two functions: checkCollisionTarget and laserHit.

The checkCollisionTarget function uses the spaceship and a target object as the arguments and returns a Boolean if a collision has taken place or not.

```
export const checkCollisionTarget = function (spaceship, target) {
    let collided = false

    const targetRadius = target.radius
    const targetX = target.x
    const targetY = target.y

    const distance = Math.sqrt(Math.pow(spaceship.x - targetX, 2) +
Math.pow(spaceship.y - targetY, 2))
    if (distance <= targetRadius) {
        collided = true
    }
    return collided
}
```

In this function the distance is calculated between the target and the spaceship. If the distance is smaller than the target radius, which all enemy classes have as a property, then the Boolean collided is set to true and returned. This function is then used in the script.js file to handle the collisions.

The laserHit function also uses two arguments and returns a Boolean to express if a target has been hit. The arguments here are a laser object and again an abstract.

```
export const laserHit = function (laser, object) {
    let isHit = false

    const objectRadius = object.radius
    const objectX = object.x
    const objectY = object.y

    const distance = Math.sqrt(Math.pow(laser.x - objectX, 2) + Math.pow(laser.y -
objectY, 2))
    if (distance <= objectRadius) {
        isHit = true
    }
    return isHit
}
```

Here again the distance between the laser and the object is calculated and but this time it is compared to the object radius. This function is, again, later used in the script.js file, so the hit detection can be handled appropriately.

Collision Response

The collisions of the spaceship are handled through the function 'handleCollisionTarget' function that is defined in the script.js file on line 179. The function is written as the following:

```
const handleCollisionTarget = function (spaceship, targets) {
    for (let i = 0; i < targets.length; i++) {
        const target = targets[i]
        if (checkCollisionTarget(spaceship, target)) {
            spaceshipRespawn(spaceship)
        }
    }
}
//Spaceship.js file
export const spaceshipRespawn = function (spaceship) {
    spaceship.x = 0
    spaceship.y = 0
    spaceship.lives--
}
```

This function uses the spaceship and the target array as arguments and uses the previously discussed checkCollisionTarget function to trigger the spaceshipRespawn function that resets the spaceships coordinates, while also decrementing the spaceship lives. The handleCollisionTarget function is then called in the spaceshipNode.animationCallback function in line 470 to 474.

Furthermore, the hit detection between the lasers that are shot by the spaceship and the possible targets is also resolved within an animationCallback function. This time, however, it is nested within the spaceshipNode.animationCallback, which loops over the spaceship.lasers array and uses the handleLaserHitDetection function:

```
const handleLaserHitDetection = function (laser, targets) {
                            for (let target of targets) {
                                target.isHit = laserHit(laser, target)
                                if (target.isHit) {
                                    highscore = addToHighscore (highscore, target)
                                    const targetIndex = targets.indexOf(target)
                                    targets.splice(targetIndex, 1)

                                    // If the target is a large or medium asteroid,
split it into two smaller ones
                                    if (target instanceof LargeAsteroid) {
                                        mediumAsteroids.push(new MediumAsteroid())
                                        mediumAsteroids.push(new MediumAsteroid())
                                    } else if ( target instanceof MediumAsteroid) {
                                        smallAsteroids.push(new SmallAsteroid())
                                        smallAsteroids.push(new SmallAsteroid())
                                    }
                                    const targetNode = scene.findNode(target.name)
                                    if (targetNode) {
                                        scene.removeNode(targetNode.name)
                                    } else {
                                        console.error('node not found')
                                    }
                                }
                            }
                        }
```

Firstly, this function loops over the target array and then checks if the target was hit or not through the laserHit function. If the target was hit the highscore is increased through a function that uses the targets value and adds it to the highscore. Then later on it is checked if the target is a large or small asteroid. This behaviour has been previously described in the report. These two functions are also used for the behaviour of the lasers of the saucer objects.

```
const handleShootingOfSaucers = function () {
        //method to handle laser hitting
        const handleLaserHitDetection = function (laser, targets) {
            for (let target of targets) {
                target.isHit = laserHit(laser, target)
                if (target.isHit) {
                    const targetIndex = targets.indexOf(target)
                    targets.splice(targetIndex, 1)

                    // If the target is a large or medium asteroid, split it into
two smaller ones
                    if (target instanceof LargeAsteroid) {
                        mediumAsteroids.push(new MediumAsteroid())
                        mediumAsteroids.push(new MediumAsteroid())
                    } else if ( target instanceof MediumAsteroid) {
                        smallAsteroids.push(new SmallAsteroid())
                        smallAsteroids.push(new SmallAsteroid())
                    }

                    const targetNode = scene.findNode(target.name)
                    if (targetNode) {
                        scene.removeNode(targetNode.name)
                    } else {
                        console.error('node not found')
                    }
                }
            }
        }
```

Within this handleShootingOfSaucers function the handleLaserHitDetection was newly defined as the highscore should not be added when hitting an asteroid. Another issue that was encountered, however, is that only one of each kind of saucer is able to shoot lasers. Sadly, I was not able to find the solution to that problem, as each saucer should be able to shoot, which was defined in the movement function of the saucers:

```
export const moveSaucers = function (saucer, spaceship, saucersLasers) {
    changeOrientation(saucer)
    saucer.x += saucer.speed
    if (saucer instanceof largeSaucer) {
        saucersShoot(saucer, saucersLasers)
    } else {
        smallSaucersShoot(saucer, spaceship, saucersLasers)
    }
}
```

Scoring and Life System

The scoring system uses the values of each enemy object and adds it to the high Score. The implementation of the values can be seen in the constructors of the enemy classes, which was previously described.

The life system is controlled by the spaceship.lives variable, which is set to 3 within the constructor. However, these can be decremented when hit by an object or a laser in the spaceshipRespawn function. Furthermore, when restarting the game the value is reassigned through the global constant INITIAL_LIVES in the script.js file in line 27. Furthermore, when reaching 10k points as a score the player gains another life. This behaviour is defined within the render function:

```
//if the extra life is possible to have and highscore high enough then add the
extra life
        if (possibleExtraLife && highscore > 10000) {
            spaceship.lives++
            possibleExtraLife = false
        }
```

This if-statement checks if the player can gain an extra life, which is defined by the possibleExtraLife variable in script.js file in line 32, in combination with the high score being over 10k. Then the live counter is incremented, and the flag is set to false, so the player cannot a new life anymore.

Game Progression

The game progression is controlled by the function initNewWave. The goal is to progress through the game if all present asteroids and saucers are removed. However, I have decided to progress throughout the waves by just checking the asteroid arrays. The reason for this is that saucers can be spawned anytime in ten second timespan. The initNewWave function, thus, looks like this:

```
function checkArrayLengths () {
    return largeAsteroids.length === 0 && mediumAsteroids.length === 0 &&
smallAsteroids.length === 0
}

function initNewWave () {
    if (checkArrayLengths()) {
        waveNr++
        amountToSpawn++
        spawnAsteroids(amountToSpawn, largeAsteroids, mediumAsteroids,
smallAsteroids)
    }
}
```

The initNewWave function uses the checkArrayLengths function to check if the arrays are empty. If the arrays are empty the value true is returned which triggers the initNewWave to increase the waveNr and amountToSpawn variables. The waveNr variable is used to portray

the current wave state of the game. The amountToSpawn is increased by one as the base value is 4, which results in the requested number of asteroids to spawn n + 4.

The decision about which saucer kind to spawn is implemented in the spawnSaucers function. The behaviour has been previously explained. To summarize, if the high score is below 10k then only large saucers are spawned and ever number above results in small saucers being called. The abortion of the game is controlled by the players life's being reduced to zero. The updateGameState function uses the following if statement to change the game state into the game over mode, so that the game can be aborted:

```
const updateGameState = function (spaceship, scene) {
// previous code
if (spaceship.lives <= 0) {
            gameState = 'GameOverMode'
        }
// following code
}
```

Rendering (45%)

To structure the rendering process, I have used the framework given to the students by Dr. Spina and Dr. Bugeja. The framework uses and interplay between the scene.js, woggle.js, model.js, matrix.js, material.js, light.js, index.html, and script.js file. At first, the main function in the script.js file instantiates the canvas and the webgl context. Then a new scene is instantiated which saves all the following steps to create a scene graph.

At first, I created multiple geometrical shapes for each object in the scene graph that has to be portrayed. Firstly, I created a quad that wraps over the entire screen and is used to create the space like background of game. Secondly, three different geometries were created for the different kinds of asteroids. Each of the geometries are based on the deformedSphere function. The deformedSphere function is placed in the CreateGeometry.js file in line 101 to 164. At first it creates a sphere but also defines a deformation, which is then applied to the spherical coordinates. Through that the asteroid geometry vertices resemble a real life asteroid instead of just being spherical. Then, the spaceship geometry was defined, which is a triangle, that will be manoeuvred by the player. Lastly, two other sphere geometries were set up for the two kinds of saucers. These geometries were then assigned to the different models for each object that they are representing.

Secondly, the light source was defined. The light source chosen was a directional light and its position was set to the star emitting light in the background picture. By binding it to the same coordinates as the ones in the background picture, the simulation of it emitting light can be achieved.

Thirdly, different kinds of materials were created. These materials act as textures and use pictures from the pictures folder to create textures for the objects. These .png files give a visually pleasing aesthetic to the objects on the screen that are rendered. Therefore, the asteroid, saucer, spaceship and background objects seem more realistic. After defining the materials, they are bound to the model of each object.

Lastly, I created multiple nodes to create the scene graph. At first, I limited myself to the creation of Group nodes and a limited amount of model nodes.

```
let lightNode = scene.addNode(scene.root, light, 'lightNode', Node.NODE_TYPE.LIGHT)
    //implement the starry background --> needs to be first so the other stuff
renders on top of it!!
    let backgroundNode = scene.addNode(lightNode, backgroundModel,
'backgroundNode', Node.NODE_TYPE.MODEL)
    let asteroidNode = scene.addNode(lightNode, null, 'asteroidNode',
Node.NODE_TYPE.GROUP)
    let spaceshipNode = scene.addNode(lightNode, spaceshipModel, 'spaceshipNode',
Node.NODE_TYPE.MODEL)
```

```
    let lasersNode = scene.addNode(lightNode, null, 'lasersNode',
Node.NODE_TYPE.GROUP)
    let evilLasersNode = scene.addNode(lightNode, null, 'evilLasersNode',
Node.NODE_TYPE.GROUP)
    let saucerNode = scene.addNode(lightNode, null, 'saucerNode',
Node.NODE_TYPE.GROUP)
```

The model nodes are limited to the spaceship and the background node, as they can be placed without having other nodes getting attached to it. The asteroidNode, laserNode, evilLaserNode, and saucerNode are group nodes, so that in the following code other nodes can be attached to them.

For the asteroid nodes of each different kind, multiple loops were created to set up the different nodes for each asteroid. These loops go through each element of the asteroid arrays and create a node for them. These nodes are then attached to the corresponding group node. Then these nodes have an animationCallback and visualize the movement of each movement function that affects the objects.

The spaceship node seems particularly interesting, as the shot lasers are also rendered within the animationCallback of the spaceship. However, the lasers could also exit the screen, which lead to performance issues after spawning multiple lasers. This problem has been solved through filtering the laser array and removing the nodes that correspond the the laser off screen.

```
//create new array that has all the lasers that are off screen and store them
            let offScreenLasers = spaceship.lasers.filter((laser, i) =>
!isOnScreen(spaceship.lasers[i]))

            //manipulate the lasers array that has all the lasers that are on
screen
            spaceship.lasers = spaceship.lasers.filter((laser, i) =>
isOnScreen(spaceship.lasers[i]))

            offScreenLasers.forEach((node) => { scene.removeNode(node.name) })
```

By filtering the laser array and deleting the nodes, only the needed on-screen lasers were visible and computational power is reserved for the necessary objects that have to be rendered. It is to note that the creation of this offScreenLaser array is repeated multiple times as the animationCallback is called by the render function. Therefore, the array is frequently filtered so that the lasers off screen are constantly filtered out.

The saucers are defined within the spawnAndMoveSaucers function. This function is repeated function every ten seconds, which is achieved through the setInterval function.

```
function spawnAndMoveSaucers () {
        let crtNumberLargeSaucers = largeSaucers.length
        let crtNumberSmallSaucers = smallSaucers.length

        spawnSaucers(largeSaucers, smallSaucers, highscore)

        // Add nodes and animation callbacks for large saucers
        for (let i = crtNumberLargeSaucers; i < largeSaucers.length; i++) {
            let largeSaucersNode = scene.addNode(saucerNode, largeSaucersModel,
largeSaucers[i].name, Node.NODE_TYPE.MODEL)
            largeSaucersNode.animationCallback =
createSaucerAnimationCallback(largeSaucers[i], largeSaucersNode, largeSaucerLasers)
        }

        // Add nodes and animation callbacks for small saucers
        for (let i = crtNumberSmallSaucers; i < smallSaucers.length; i++) {
            let smallSaucersNode = scene.addNode(saucerNode, smallSaucerModel,
smallSaucers[i].name, Node.NODE_TYPE.MODEL)
            smallSaucersNode.animationCallback =
createSaucerAnimationCallback(smallSaucers[i], smallSaucersNode, smallSaucerLasers)
        }
    }
    // Run the function initially
    spawnAndMoveSaucers()

    // Set interval to run the function every 10 seconds
    setInterval(spawnAndMoveSaucers, 10000)
```

As it is constantly repeated, I have encountered the problem that the animation for the saucers sped up after each repetition of the spawnAndMoveSaucer function. This was caused by the saucer nodes repeating the moving function. To solve this issue, I have decided to use a control number as a parameter for the for-loops. The control variable changes constantly, as the array length also changes constantly. Through assigning the length of the arrays as the i parameter, the saucers that already have a node and their movement defined are not included in the future callbacks.

Sadly, I was not able to include particle effects to my gameplay due to time restriction within the exam phase at the university. Additionally, I was not able to include a classical render mode that renders the outlines and strips the game of its textures and colours, due to the same reasons previously mentioned.

User Interface (UI) (20%)

The goal of my UI was to create a game state machine that makes it simple for the potential player to navigate between the game states. The game state machine encompasses of the following three game states: Attract Mode, Gameplay Mode, and Game Over Mode.

```
//-------------------------------------------------------------------------------//
// variables for the UI
//-------------------------------------------------------------------------------//
let gameState = 'AttractMode' // starts with Attract, as it always does this to
begin with --> gets changed to transitin between game states
const startButton = document.querySelector('.start-button')
const gameTitle = document.querySelector('.game-title')
const highScores = document.querySelector('.high-scores')
const instructions = document.querySelector('.instructions')
const controls = document.querySelector('.controls')
const gameplayContainer = document.querySelector('.gameplay-container')
// const lives = document.querySelector('.lives')
// const score = document.querySelector('.score')
const gameOverContainer = document.querySelector('.gameOver-container')
const initialsInput = document.getElementById('initials-input')
const submitInitialsButton = document.getElementById('submit-initials-button')
const returnButton = document.getElementById('return-button')

gameplayContainer.classList.add('hidden')
gameOverContainer.classList.add('hidden')

//-------------------------------------------------------------------------------//
// UI implementation
//-------------------------------------------------------------------------------//
const updateGameState = function (spaceship, scene) {
    if (gameState === 'AttractMode') {
        startButton.addEventListener('click', () => {
            startButton.classList.add('hidden')
            gameTitle.classList.add('hidden')
            highScores.classList.add('hidden')
            instructions.classList.add('hidden')
            controls.classList.add('hidden')
            gameState = 'GameplayMode'
        })
        // Display game title
        displayTitle('Asteroids')

        // Display high scores
        displayHighScores(highscores)

        // Display instructions on how to start the game
        displayInstructions('Press the start button to start your game :)')

        // Ensure all controls are well communicated to the player through the UI
        displayControls()
    } else if (gameState === 'GameplayMode') {
        // Display lives and score during gameplay
        gameplayContainer.classList.remove('hidden')

        displayLives(spaceship.lives)
        displayScore(highscore)
        displayWaveNr(waveNr)
```

```
        if (spaceship.lives <= 0) {
            gameState = 'GameOverMode'
        }
    } else if (gameState === 'GameOverMode') {
        // Display game over screen
        gameOverContainer.classList.remove('hidden')

        // Handle high score input
        submitInitialsButton.addEventListener('click', () => {
            // Get  input value and strip whitespace
            const initials = initialsInput.value.trim()
            // Check if initials is not empty and not already in highscores array
            //--> this is supposed to not let the same initial be set one after the
other
            if (initials && !highscores.includes(initials)) {
                // Add to highscores array
                highscores.push(initials)
            }
        })
        // Handle return to Attract Mode or restart
        returnButton.addEventListener('click', () => {
             // Reset any other necessary variables or game state
            gameOverContainer.classList.add('hidden')
            gameState = 'AttractMode'
            spaceship.lives = INITIAL_LIVES // Reset spaceship lives to initial
value
            highscore = 0 // Reset the highscore
            waveNr = 0 // Reset the wave number or any other game progress
indicators
            amountToSpawn = 4    // reset to base amount of asteroids to spawn

            // Hide gameplay container if it was displayed
            gameplayContainer.classList.add('hidden')

            // display attract mode elements
            startButton.classList.remove('hidden')
            gameTitle.classList.remove('hidden')
            highScores.classList.remove('hidden')
            instructions.classList.remove('hidden')
            controls.classList.remove('hidden')

            //reset the gameplay arrays
            removeAllFromArrayAndScene(largeAsteroids, scene)
            removeAllFromArrayAndScene(mediumAsteroids, scene)
            removeAllFromArrayAndScene(smallAsteroids, scene)
            removeAllFromArrayAndScene(largeSaucers, scene)
            removeAllFromArrayAndScene(smallSaucers, scene)
        })
    }
}
```

Firstly, I set up multiple variables that define the game state and variables that acces the html elements that are used to portray the game state. If the game state is set to Attract Mode, the start button and other elements are visible on the screen, and some can be interacted with by the player. Furthermore, different functions are used to show the Attract Mode elements, which are located in the AttractMode.js file. Then it was defined that at the event of the start button being clicked the game state switches to gameplay mode and hides all the Attract Mode elements. In the Gameplay Mode the previously hidden Gameplay Mode elements remove the hidden property and are shown on the screen. These elements display the lives,

the current score of the player and the wave number which is currently played by the player. Then the condition that if the lives reach 0 the Game Over Mode is applied as a state. Within that state the previously hidden Game Over elements were hidden are shown. The gameplay visuals, however, are still visible but the player is not able to give button inputs. This has been defined in the movement function of the spaceship, as it checks for the game state before interpreting the button inputs. This happens in line 22 of the Spaceship.js file. Then, the event listeners are added to the needed buttons to restart the game or to submit the players initials in the high score list. The submit button works as the following: It take the initials typed in by the player and trims for white spaces so that no repetition of the initials can be put into the high scores array. The repetition is avoided through the if statement that checks if there is any kind of input by the player in combination with the parameter the initials are not already part of the high score array. If this is condition is met, then the initials are inserted into the array. The return button hides all the Game Over Mode elements and resets all the arrays and variables that are used to start the game. Furthermore, the Attract Mode elements, which were hidden are then showed again on screen. Therefore, the player returns to the Attract Mode.

Conclusion and reflection on the project

Throughout this project, I have used a scene graph to handle the rendering to visualize the previously employed game logic. Through this interplay the asteroid game is playable through a webgl context. Furthermore, through the implementation of a simple UI that has distinct features, the game is easier to grasp for the potential player and the game is clearly structured.

Sadly, I was not able to integrate every requirement of the project due to external deadlines of my studies. However, the core principles and certain extras were applied to the gameplay, as a modern version of the classic video game called asteroids was realized.

Overall, the project presents on a modern take on a classic video game by using models, textures, and a light source to give an almost 3d version of the game.