Spaceship Run

Report of the final project of
INTERACTIVE GRAPHICS

Submitted by:
Anton Volkov **Simone De Angelis**

DEPT. OF INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE
LA SAPIENZA, UNIVERSITY OF ROME
Rome
September 2022

Contents

Chapter 1

Introduction

The idea behind the creation of the game is pretty simple: a spaceship needs to reach the end of a path composed by some rings placed in the void space while avoiding swarms of asteroids (see image 1.1).

The development process has followed this idea and was centered around the three main elements of the game just described:

- Spaceship: movement and controls
- Meteorites: movement and collisions
- Rings: Path definition

The structure of this report, after the description of the environment and libraries used, will follow the same flow of development of the game: we will describe how the spaceship and its movements have been modelled, as well as how the application gets the input from the user. After that we're going to describe the asteroids implementation, both in terms of movement

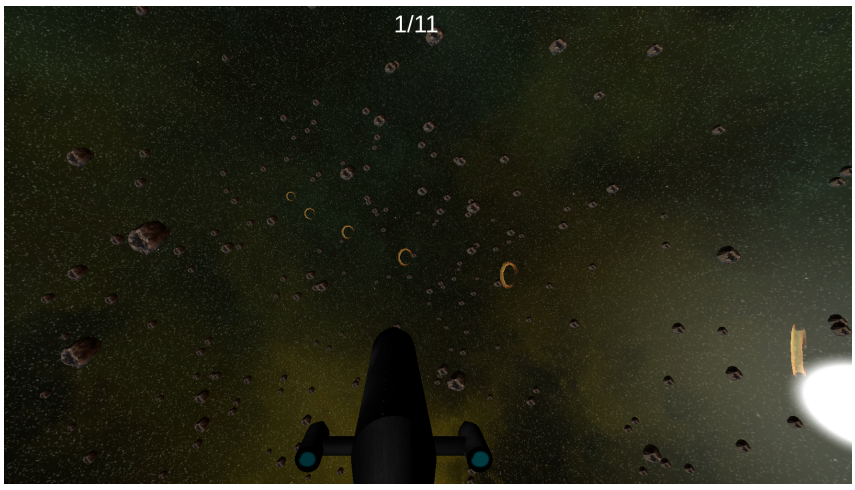

Figure 1.1: Game view

and instantiation to save computational usage. Moreover, we're going to describe the collision system and path creation, and how the game ensures that the user pass through each ring placed in the map.

1.1 Controls

The user can control the movements of the spaceship both using keyboard or mouse inputs. Using the drag controls one can move the ship just by pressing the **w** key that will start the movement in the z direction of the ship, while orienting the ship by holding the right button of the mouse and moving the cursor. Key commands:

- $W\ A\ S\ D$ movement along z and x directions.
- $R\ F$ movement along y direction
- $Q\ E$ rotation counter and clockwise around z.

Chapter 2

Environment and code structure

This chapter will describe all the environments and libraries used to develope the project.

2.1 Three JS

Three.js is a JavaScript-based WebGL engine that can run GPU-powered games and other graphics-powered apps straight from the browser. It is the main environment used for the realization of the project. All the features present in the game are included in the library:

- intersection method to check collision between boxes
- fly controls used for getting input for the ship
- matrix transformation to place and move objects in the scene
- lights
- loaders to load 3D models and textures

Our SpaceShip Run game does not make use of an external physic engine, since the only physic interaction that we needed to implement was collision. This choice made the game less heavier at the cost of implementing our own collision system. In the next chapters we will talk about the implementations of the items listed above.

2.2 Blender

In order to have the ship such that the components that have to be animated are in a hierarchical structure, the ship has been modeled and textured by Anton Volkov in blender, a 3d modeling software.

2.3 Code structure

In this section we are going to go over all the files composing the game, focusing on their functioning inside the application. We will skip the *InstancedMeshGroup.js* , *ship.js* and *levelGeneration* file that will be the topic of the next chapters. The code files can be seen in image 2.1.

2.3.1 Game.html and game.css

In this file are coded the 3 different pages that compose the game together with the one drawn by the renderer of Three js. The inital screen is the initial page that briefly describes the goal of the game and asks the user to choose between 3 levels of difficulty that will influence the number and velocity of the asteroids. ”Easy” loads 200 asteroids, ”Medium” loads 400 asteroids and increments the velocity by 50% and ”Hard” loads 600 asteroids and doubles the velocity with respect to the easy mode. The other 2 pages appears when an intersection with an asteroid happens i.e game over or when the player wins the game by passing through each ring.

2.3.2 gameMaster.js

In the previous subsection 2.3.1 we have talked about the 3 different screens that can appear during the game. The gameMaster class holds the methods that gets called inside the *game.js* file to show each screen instead of another one. This is done using a function *toggleScreen* that takes as input the id of a screen and a boolean value to turn on or off the screen associated to the id just passed as input.

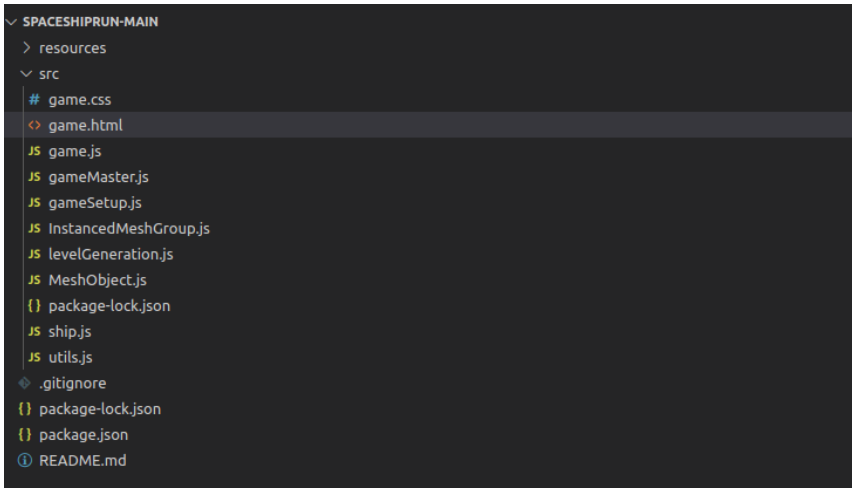

Figure 2.1: Code structure

2.3.3 MeshObject.js

This class inherits the Object3D class of Three js and adds 2 methods: one to directly load GLTF models inside the class, and another one to create a bounding box to the object defined in the class. This last method takes as input the dimension of the box, instead of automatically computing it, since one could load a complex model with several meshes. We will talk about this problem and its workaround more in detail when dealing with the class that adds and move meteorites in the scene, *InstancedMeshGroup.js*.

2.3.4 gameSetup.js

This file holds the function that sets up the scene. This function works asynchronously, since the load of the 3D model of the meteorites is done using the Three js GLTF loader that loads asynchronously the model. Inside this function each object composing the game gets added to the scene:

- Ship
- Swarm of meteorites
- Rings
- Camera
- Skybox
- Lights

A skybox gets loaded thanks to the TextureCubeLoader of Three js and added as a background to mimic depth and appearance of the space. Two source of lights are added to the scene: a directional light and a point light positioned where the sun of the skybox is placed i.e at infinity (the infinity value for x position of the point light is set to a high value) in the x direction.

2.3.5 game.js

This file holds the game class that wraps up all the files mentioned in the previous subsections. It also holds the animation loop in which all the methods that move the objects in the scene are called, as well as a *init* method that sets up the scene.

Chapter 3

The Ship

3.1 The Ship model

The ship has been modeled in Blender, and is composed of 3 main parts:

- The fuselage: main body of the ship, where the cockpit is present. It's composed of a cilinder, a cone for the nose and a sphere scaled up on an axis for the cockpit.
- The directional thrusters axis: an axis where the directional thrusters are connected to and can swivel around. It's only a thin cylinder
- The directional thrusters: thrusters that are animated in the code to provide a visual explanation on how the ship turns.

3.2 Ship's animation

The ship uses the FlyControls module from Three.js for the movement controls. For the animation every frame the ship's method vectorThrust, which checks the last frame forward direction, in the ship's space, against the current ship's direction, which is always the negative z direction. We then subtract the old direction from the current one, getting so the direction the ship is tilting towards in the ship's space, and by using the x and y coordinates of the resulting vector we roll the ship and tilt the directional thrusters to animate the ships as it's actually using those thrusters to change its direction.

Chapter 4

Instantiated asteroids

One of the major problems that we needed to face when developing the game was how to make the game run smoothly while still drawing a huge amount of vertices for the asteroids. Fortunately, THREE js has a class called *InstancedMesh* that one can use when it's needed to render a large number of objects with the same geometry and material but with different world transformations. However, using this class instead of the standard one introduced various problems in terms of implementation of collision and 3D model complicated by the fact that the documentation lacks of explanations for the latter cases. In this chapter we will talk about our class *InstancedMeshGroup* that is a incomplete attempt to wrap up the instanced mesh class of THREE js with some methods to load instantiated 3D models, place and move them around the scene and create bounding box around them.

4.1 InstancedMeshGroup Class

The InstancedMeshGroup Class place meshes in the scene with respect to randomly sampled positions inside a cube with a user defined side and center in the scene. Once each position of each mesh is sampled (line 7-10 in 4.1) a traverse function loops over each child of the 3D model passed as input to this class (promise at line 1) and once a mesh is found, the method creates a instanced mesh using the child geometry and material, scaling and positioning each geometry at the previously sampled positions (line 17-24). Finally, the instanced mesh is added to the instanced mesh group.

```
1 loadMeshAsCube(count=1 , promise, CubeOrigin = [0,0,0] , CubeSide = 10,
     scale, parent=this)
2 {
3     this.CubeSide = CubeSide
4     this.CubeOrigin = CubeOrigin
5     parent.count = count
6
```

```
7     for (let i=0; i < parent.count; i++)
8     {
9         parent.meshPosition.push(sampleCubePosition(CubeSide, CubeOrigin
              ))
10    }
11    promise.scene.traverse
12    (function(child)
13        {
14        if (child.isMesh)
15        {
16        child.material.side = THREE.DoubleSide // An attempt to fix the
              hole in the meteorites
17        var instancedMesh = new THREE.InstancedMesh( child.geometry,
              child.material, count );
18        var matrix = new THREE.Matrix4()
19        for (let i=0; i < count; i++)
20            {
21            matrix.makeScale(scale, scale, scale)
22            matrix.setPosition(parent.meshPosition[i][0], parent.
                  meshPosition[i][1], parent.meshPosition[i][2])
23            instancedMesh.setMatrixAt( i, matrix );
24            }
25        parent.add(instancedMesh)
26        instancedMesh.instanceMatrix.needsUpdate = true;
27        instancedMesh.matrixWorld
28        }
29        }
30    )
31    }
```

Listing 4.1: Mesh loading

Two other methods of the class, *moveAlongAxis* and *rotateOnAxis*, allow each instantiated mesh to move and rotate along a specific axis. THREE js already has methods to move and rotate 3D objects but does not provide similar functions for the case of instantiated objects. By using this class, it's possible to load any 3D GLTF or GLB model, place and move them in the scene. Moreover, if only a single mesh compose the 3D model loaded, the class can also associate to each geometry an update-able bounding box as we will see in chapter 6.

4.2 Respawn

Since we wanted the asteroids to constantly move against the ship, the method *moveAlongAxis* takes as optional inputs a respawning and limit position. For each geometry of the instantiated

mesh the method check if, along the chosen axis, the position has exceeded the limit parameter. If this happens, each geometry position along the chosen axis gets reset to the respawning parameter passed as input.

```
1 this.meteorites.moveAlongAxis('z', this.scene.userData.
    meteorites_velocity, this.ship.position.z -500, this.ship.position.z
    +100)
2 this.meteorites.rotateOnAxis('z', .01)
```

Listing 4.2: moveAlongAxis inside the animation loop

We can see from 4.2 that the respawn and limit position depends on the ship z position. This implementation is sufficient to achieve the wanted behaviour of the asteroids, since, as we're going to see in detail in chapter 5, the path is mostly linear. However, one can notice that by moving the ship for a while along a direction different from z, the meteorites spawn always at the same x and y position.

Chapter 5

Ring path

In this chapter we will describe how the rings are generated randomly in the map, and how the game ensures that the player pass through each ring. The *levelGeneration.js* file holds the class that generates the random levels of the game. Each ring is a MeshObject (see 2.3.3) with an associated mesh composed by the geometry of a torus and a MeshBasicMaterial (not affected by lights in the scene) to which a texture of an explosion is applied (see image 1.1).

5.1 Rings generation

To generate the path we first calculate the number of checkpoints. Then for the path we start from a checkpoint and use a *direction* vector to keep track of the direction of the track. For each checkpoint, we find a new direction that has a 10° difference from *direction* and place it at a defined distance from the previous checkpoint, in our case 100 units away, then we set *direction* to the new found direction and repeat until all checkpoints are placed. This process ensures that the path created does not deviate too much from the z direction, remaining mostly linear.

5.2 Track definition

In order to ensure that the ship pass through each ring in the map, to each ring is associated a bounding box that gets computed using the second method *createBoundingBox* of our class MeshObject, as already discussed in 2.3.3. As can be seen in 1.1, we added a counter at the top of the canvas that updates itself whenever a intersection with the ship is detected. The game ends only when this counter reaches the its maximum value.

Chapter 6

Collision system

The collision gets checked inside the game loop defined in the *game.js* file. There are differences on how we created and updated the bounding boxes associated to the objects in the scene, but in the end we always used the *intersectBox* method of the class **Box3** to check collisions between boxes.

```
1  // Collision detection stuff
2  this.ship.updateBoundingBox(this.ship.matrixWorld)
3  this.meteorites.intersectObject(this.gameMaster, this.ship.helperBox.box
       )
4  for (let mesh of this.checkpoints.children)
5  {
6      if (this.ship.helperBox.box.intersectsBox(mesh.helperBox.box))
7      {
8          this.gameMaster.updateCPCount()
9          mesh.helperBox.box.makeEmpty()
10     }
11 }
```

Listing 6.1: moveAlongAxis inside the animation loop

Line 3 at 6.1 check the collision between meteorites and the ship. Line 4-11 instead check the collision between the bounding box placed inside the rings and the ship. In the next sections we're going to talk about how we approached the creation and update of the bounding boxes for the ship and the meteorites.

6.1 Ship's bounding box

The bounding box the ship is created using the method inside the custom Ship class. This method takes as input the dimension of the box and place the bounding box where the ship is positioned using a THREE js method of the Box3 class. The bounding box is then updated in

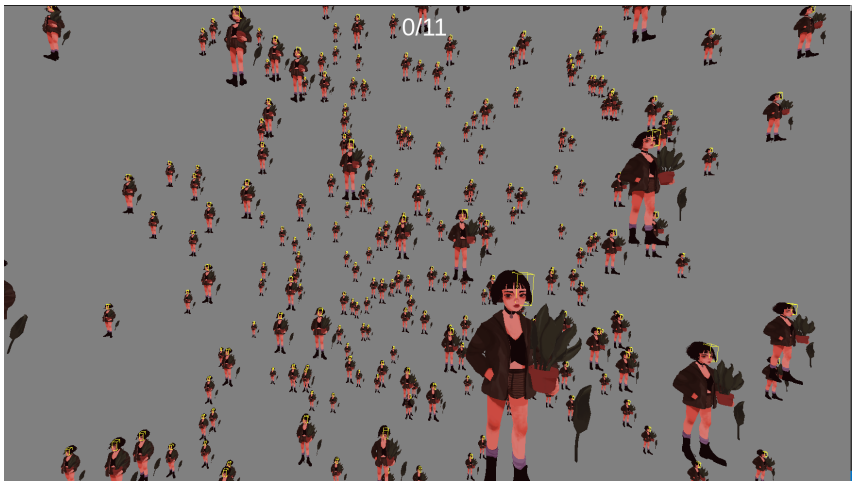

Figure 6.1: Wrong bounding box computation

line 2 at Listing 6.1 by applying the world matrix of the ship to the geometry of the box.

6.2 Asteroid's bounding boxes

The development of the bounding boxes of the asteroids has followed a different approach with respect to the other bounding boxes present in the scene. The InstancedMeshGroup class provides two other methods to create the bounding boxes of the instantiated mesh and to intersect another bounding box with the ones created. The method to create bounding box will return a warning if the loaded model is composed of multiple meshes. This is because the geometries of the bounding boxes are computed using the property *boundingBox* of mesh's geometries. If a model is composed of multiple meshes, this computation will be wrong, as can be seen in image 6.1, where a nice model of Matilda from the movie Leon The Professional is shown. The model used for the asteroids has only one mesh and thus the bounding boxes get correctly created and updated inside the *moveAlongAxis* method called in the game loop (see listing 4.2) similarly to how we've seen earlier for the ship case. Finally, at line 2 in Listing 6.1 the method *intersectObject* of the custom class is called and loops over each bounding box, calling the *intersectBox* function of THREE js to detect possible collisions.

Chapter 7

Sources

Meteorites model
Metal texture
Skybox
Matilda model