SAPIENZA
UNIVERSITÀ DI ROMA

Space Invaders Ultimate

Interactive graphics course - Final project

Federico Alfano 1262220 - Luca Vargiu 1422135

December 11, 2018

1 Introduction

The project is a tribute to the to the popular '80s game: **Space Invaders** giving it our personal interpretation, in a 3D remake. At an early stage we made some design choices. The first one is the framework, Three.js in this case it seemed to fit perfectly our needs as well as being highly documented.
With the aim of addressing the difficulties of teamwork and to be open to further additions we have decided to use the MVC architectural pattern and other ones that will be described in this report taking advantage of the great work published on the blog hecodes.
The game will have four different modes:

1. Classic Arcade: *Is the one similar to the original one;*
2. Classic Custom: *Is similar to the previous one, but instead of levels you can choose the game parameters;*
3. Futuristic Arcade: *It has the same dynamics of the classic mode, but the enemies have a more "realistic" movements;*
4. Futuristic Custom: *Is similar to the previous one, but instead of levels you can choose the game parameters.*

The *"Arcade"* game modes involve the player in a session of 5 waves with escalating difficulties, while the *"Custom"* ones regard uniquely a user-defined wave. But before starting dive into the project in a more technical way, let's see how the main page will appear.

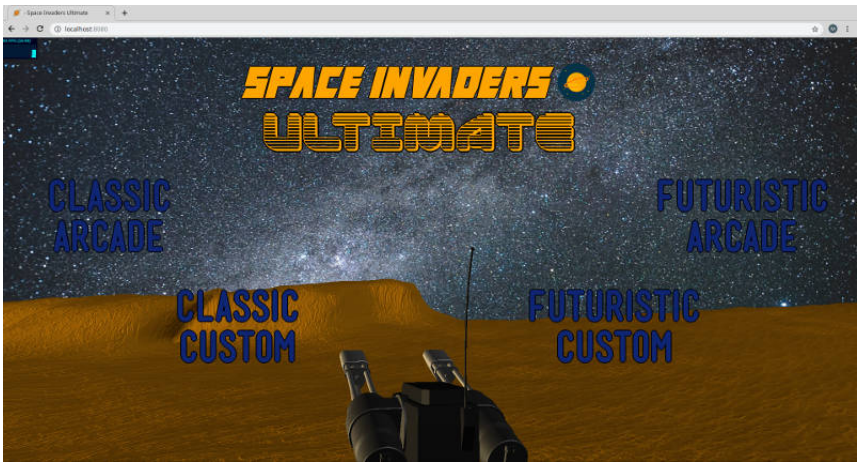

Figure 1: The Animated HomePage of the final result

1.1 MVC

As we said in the introduction we think that the teamwork and the complexity of the project needs a more structured approach, and we found valid guidelines into the *hecodes* blog [1]. This section will not provide a useless repetition of the cited article, conversely we'll focus on the differences. Due to the complexity of the project if compared to the example we had to make some different design choices, mainly the worhy of note are two:

- The TurretMediator doesn't directly contain the object3D related to the turret but a container that contains the turret itself plus the bullet, this hierarchy is done just to make the bullet's movement easier to generate.
- The second difference is about the controller, into the example we have just one controller related with the MainView, in our implementation we had too many function and we created many more controllers, everyone associated with his own mediator. We also decide that Every controller is in charge of every logic operation such as the computation of the next position, even if in the article it has just to react to user inputs.

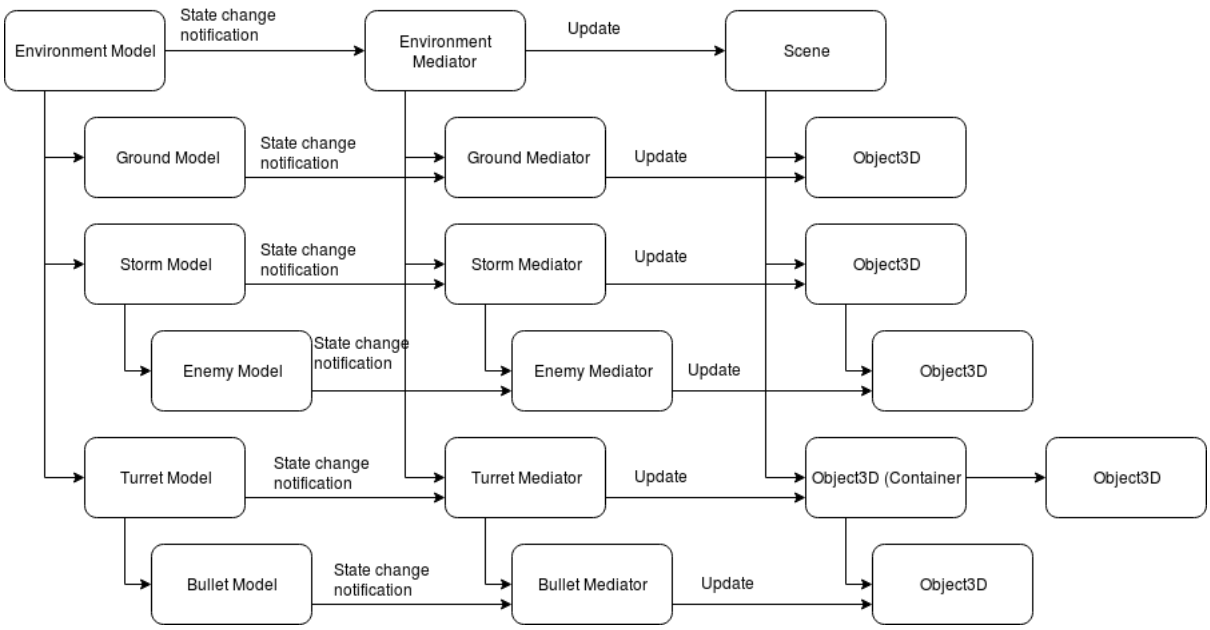

Figure 2: The structure of the software

2 Models and Hierarchy

In this chapter we are going to talk about the models and their hierarchy. As the architecture states, every model has to notify changes to his mediator that propagates on his object3D the modifications. The work needs those main models:

- Environment
- Ground
- Turret with the bullets
- Storm which contains a number of enemies

A good source where to get the models and the textures is the website *clara.io* where we got the Ground, the turret and the enemy in the *collada* format, loading them with the colladaLoader.
So let see an overview of our chosen organization of the project. The Environ-

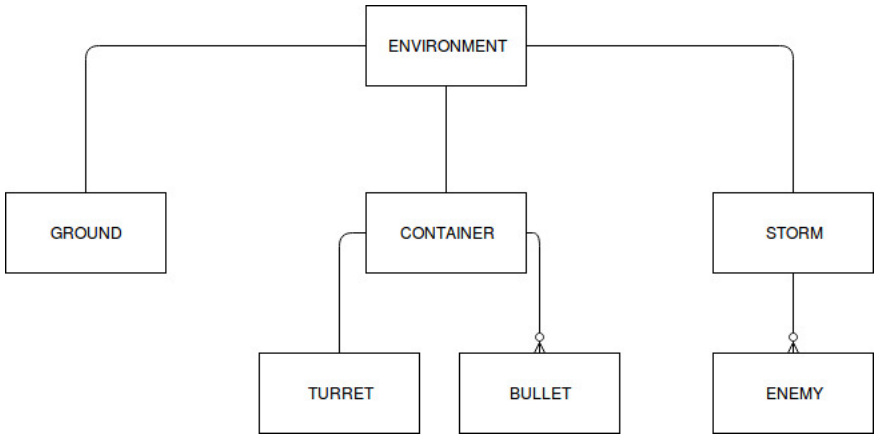

Figure 3: An ER style diagram of the objects hierarchy

ment is a big container, and the Ground is a standalone component, they are not interesting as the other two.

2.0.1 Turret/Bullet

They are children of the same container because (as we'll see into the animation section) the bullet's animation starts from the turret's position but must not be influenced by its movement.
The turret is probably the most complex model into the project. The model downloaded from *clara.io* was a single block, so we needed to separate it into three parts:

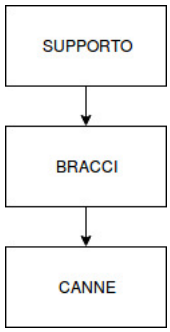

Figure 4: A simple representation of the turret parts in a diagram.

Subdivision of the turret's parts required the use of the Blender 3D modeling software in order to obtain the parts previously described in the tridimensional model used in the project. In the following image you can see the result obtained by the hierarchical subdivision of the model in the application: There is not so

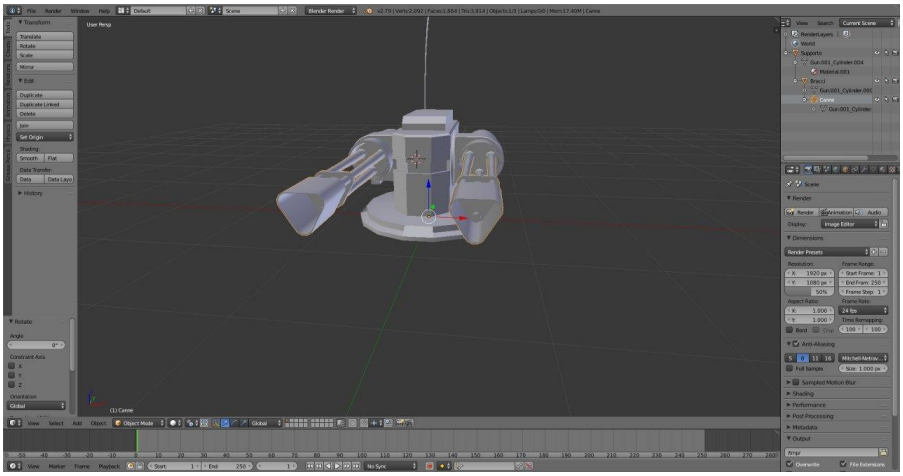

Figure 5: Turret 3D model hierarchical subdivision in Blender.

much to say about the bullet, it is a very simple sphere, with a yellow basic Mesh Material. It's object3D is encapsulated into the BulletMediator that is child of the TurretMediator.

2.0.2 Storm

The *Storm* is the enemies' container into the classic mode, it's just a transparent inclined plane containing all the enemies, it allows us to control all the enemies movements simply using the x and y axes without worrying about the z one. As a final result a clear descending movement directed towards the player is obtained, correcting also the enemy invaders' inclinations.

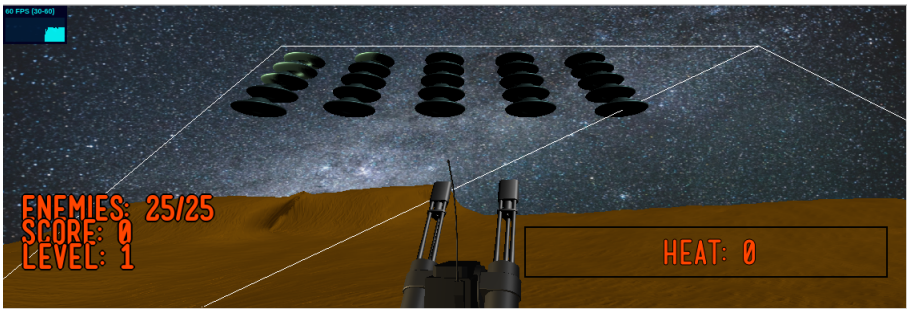

Figure 6: How the plane would appear

2.0.3 TPStorm

The *TPStorm (TwoPointStorm)* is an extension of the previously described Storm where the movements of the enemies are handled in a more freely way.
The number of enemies loaded in scene depends on how many enemies will be displayed on the screen during wave, and is defined and can be adjusted in the model by acting on one of its parameters. When the storm is build up, enemies are positioned in equal number on two different spawning points that are located over the camera's visible way.
During a wave, at temporized intervals an enemy is selected and starts to move from its spawning point entering in the player's visible area, making displacements under a set of random positions of fixed length, defined under boundaries that involve all the axes. In other terms the random positions are selected in a parallelepiped enclosed in the player's field of vision.
After all random positions have been reached, the enemy is guided to reach a final position randomly chosen among six predefined available.

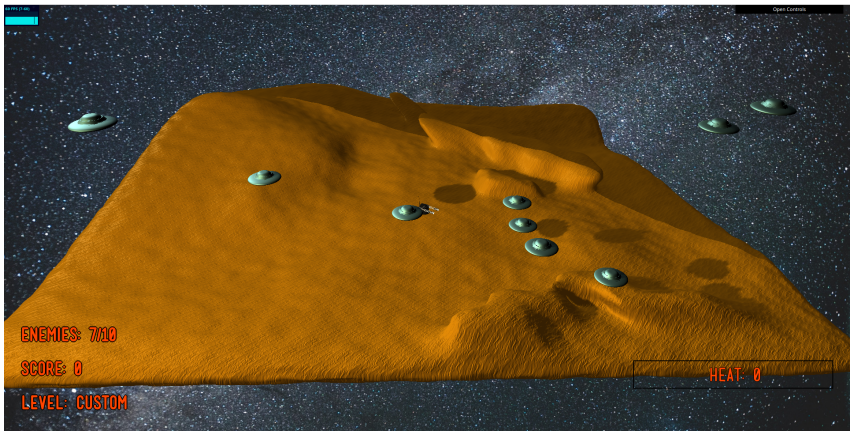

Figure 7: *TPStorm* appearance obtained by moving the camera.

3 Animations and Lights

Following the MVC pattern, animations' and lights' creation and management takes place in the *View/ViewMediator* and *Control* classes related to the respective *Model*.
By the following subsections there will be described with more detail the code, where necessary, and the execution related to the most important components of the project.

3.1 Turret

The turret is the most complex object and obviously their parts are animated separately:

- **Supporto:** it is the parent of all components, in particular it rotates along the y-axis following the mouse x-coordinates.
- **Bracci:** they rotate on the x-axis following the mouse y-coordinates.
- **Canne:** their animation is a translation on z-axis, it also has a color animation indicating the overheating.

```
if (this.mouse_moved) {
    this.supporto.rotation.y = -this.element.mouse.x;
    this.bracci.rotation.x = Math.min(0, -this.element.mouse.y);
    this.mouse_moved = false;

}
```

Figure 8: The code in charge of animate *Supporto* and *Bracci*

3.2 Bullet

The TurretMediator builds also a Plane far away in front of the camera that someway indicates the Far into the frustum, covering all the background. On mouse down event for every created bullet, the system calculates the intersection point with mouse coordinates and the plane in 3D space and passes it to the BulletMediator throught the related model. Once the BulletMediator launches his makeObject3D method also builds two lines starting from the turret and going to the given point. On each iteration of the render frame the bullet translate along the line for a bigger step depending on the bullet mediator's speed with the THREEJS method *at*.

3.3 Storms implementations

As described before, both the Storms share a common behaviour, related on how the two game modes keep active the interactions between the enemies, the

environment and the player's actions.
As we can inspect from the code of both the Controller classes regarding the *onGame* function that is used to execute the current frame animation, both the game modes can be subdivided in the following parts:

- *If statement* that checks whether a game-over status hasn't been already reached or the game isn't in pause, that also leads to the following actions:
  - deployment, check and movement of the enemies under the rules of the actual Storm;
  - check whether a winning or a losing condition has been reached;
  - synchronize the *UI* (User Interface) with the obtained data;
- *If statement* that checks if a reset have to be instantiated, making all enemies to return to their initial state and acting on the content of the UI.

Figure 9: TPStorm render calculation function, from TpsController.js file.

There are also similar patterns of implementation regarding the type of mode that would be executed:

- *Arcade mode*: the game can't reach the Win condition unless the final level has been beaten, otherwise forcing a soft-reset (not touching the score) at each wave;
- *Custom mode*: each Storm invokes the creation of a *Gui* that allows the interaction of the player with the gaming scene acting on some parameters that subordinate the movements and the appearance of the enemies and also the light of the environment.

In every mode the player wins only if all the enemies have been destroyed before a lose condition has been verified.
In the following subsections there will be described the peculiarities of each mode in the management of the invaders.

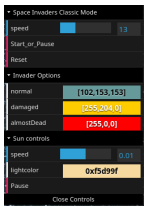

Figure 10: Storm GUI.

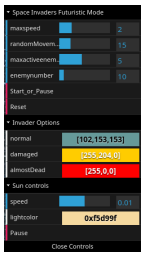

Figure 11: TPStorm GUI.

3.3.1 Storm

In the *Classic mode* the enemies' group have to bounce from one corner of the screen remaining always visible to the player. Whenever a corner is trespassed by a spaceship, the direction of the horizontal movement is changed and all the enemies are pushed down by a certain amount of space. If, during the game execution, the enemies' number decreases their speed increases, making harder the elimination of the remaining spaceships.
When an enemy is destroyed it is put in an hidden position under the map in order to avoid a possible interference with the turret's shoots. It will return to its starting point only if a reset is called.
The player loses when an enemy reaches a fixed y that coincides with the ground in its position.

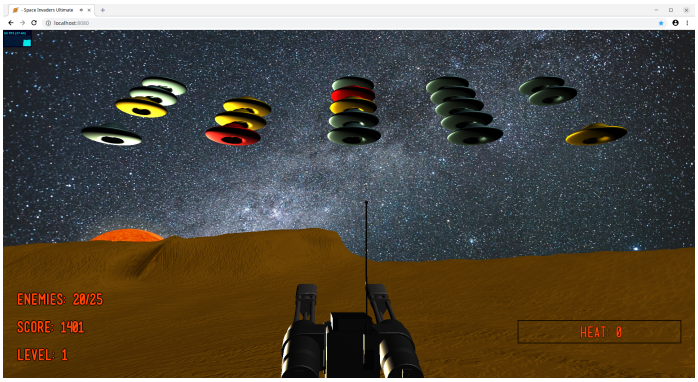

Figure 12: Storm during a wave.

3.3.2 TPStorm

In the *Futuristic mode*, as said before, each enemy is independent from another and starting from one of the two fixed spawning points it has to reach a random ending-point passing through a fixed number of random intermediate points.

During all the execution the spaceship speed remains constant but results higher than the Classical's counterpart. Moreover in this mode the behaviour of the spaceships is perceived as less predictable by the player, so making higher the grade of challenge, especially when a lot of enemies are on the screen.
When an enemy is destroyed it is restored to its spawning-point and a new random ending-point is assigned to it. If another enemy is needed in order to complete the current wave then it is put in a ready-queue that allows a future *respawn* as a new enemy.
As it can be easily assumed, the player loses when an enemy reaches its own assigned ending-point.

Figure 13: TPStorm during a wave.

3.3.3 Enemy

The enemy is a single block model, so the animation is not as complicate as the turret one, anyway would be interesting to better know how it moves. First let define the movement it does:

1. Rolling
2. Explosion
3. Movement into the space

The third one depends only to the storm, so in the next two paragraphs we'll focus only on the first two ones.

Rolling :
After an hit the enemy rolls on the z-axis, following the rule of the **Under-damped Oscillator**

$$x = e^{-\gamma t}\alpha cos[\omega_1 t - \alpha] \tag{1}$$

The implementation differs from the rule just for a t value that activates the roll only when it's less than the number of phases defined

```
if (this.rollingAnimation.isRolling) {//Rolling animation
    let temp_degree = (this.rollingAnimation.currentDegree+this.rollingAnimation.degreeSpeed)%360;
    if(temp_degree<this.rollingAnimation.currentDegree){
        ++this.rollingAnimation.t < this.rollingAnimation.phases?
        this.rollingAnimation.isRolling=true:(this.rollingAnimation.isRolling=false, this.rollingAnimation.t=0);
    }
    this.rollingAnimation.currentDegree = temp_degree;
    this.object3D.rotation.z = THREE.Math.degToRad(this.rollingAnimation.maxFirstOscillationDegrees*
            Math.exp(-this.rollingAnimation.t/this.rollingAnimation.phases)*
            Math.sin(THREE.Math.degToRad(this.rollingAnimation.currentDegree))); // Moto armonico smorzato
}
```

Figure 14: Implementation of Underdamped Oscillator's equation.

Explosion After the third hit the enemy explodes. The animation works on a second object that is hidden and overlapped to the living enemy. That object is a configurable set of THREE.Points that is created by the function *explodeAnimation* which alsco creates an array of random directions (one per point) representing a random generated value between *this.movementSpeed / 2* and *-this.movementSpeed / 2* . During the explosion the rendering calls the function *updateExplosion* increases/decreases the position on every axis by the generated direction.

```
updateExplosion(times=1) {
    var pCount = this.totalObjects;
    while (pCount--) {
        var particle = this.explosion.geometry.vertices[pCount];
        particle.y += this.dirs[pCount].y*times;
        particle.x += this.dirs[pCount].x*times;
        particle.z += this.dirs[pCount].z*times;
    }
    this.explosion.geometry.verticesNeedUpdate = true;
}
explodeAnimation(x, y){
    var geometry = new THREE.Geometry();
    for (var i = 0; i < this.totalObjects; i++)
    {
        var vertex = new THREE.Vector3();
        vertex.x = x;
        vertex.y = y;
        vertex.z = 0;
        geometry.vertices.push(vertex);
        this.dirs.push({x: (Math.random() * this.movementSpeed) - (this.movementSpeed / 2),
            y: (Math.random() * this.movementSpeed) - (this.movementSpeed / 2),
            z: (Math.random() * this.movementSpeed) - (this.movementSpeed / 2)});
    }
    var material = new THREE.PointsMaterial({size: this.objectSize, color: 0xFF0000});
    var particles = new THREE.Points(geometry, material);
    this.object = particles;
    this.status = true;
    return this.object;
}
```

Figure 15: The code responsible for the explosion animation.

3.4 Lights and Shading

In the scene there have been introduced two sources of light:

- *AmbientLight* that allows the presence of a minimum illumination in the scene also withouth a direct source of light;
- *PointLight* that is the main source of light that propagates in every direction and it's also responsible for the shadow casting and the shading effects on the objects.

In order to obtain a very dynamic effect that also changes during the course of game and the various waves the PointLight has been put as a child of a *Sun* object (a textured sphere) that periodically orbits around the ground.
The source of light shares the same color of the moving star. During the execution of the various waves, it has been implemented that also the Sun changes its color according to the number of the current level. This allows the perception of different atmospheres during the continuation of the game, making each intermediate level even more different.
These types of variations are also possible and let free to the player in the Custom mode GUI.
Following to the Threejs documentation[**?**]threejsdoc), the enabling of shadows to the objects of the scene takes only few lines of code, acting only on the *THREE.renderer* object, on the *light-source* and on the involved *meshes*:

```
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.PCFSoftShadowMap;
```

Figure 16: Renderer settings.

```
light.castShadow = true;
light.shadow.mapSize.width = 512;
light.shadow.mapSize.height = 512;
light.shadow.camera.near = this.element.properties.size;
light.shadow.camera.far = 550 + this.element.properties.distanceunits;
```

Figure 17: Light settings.

```
container.castShadow = true;
container.receiveShadow = true;
```

Figure 18: Object settings.

4 Conclusions

As a final result of the work described by this document, our group comes out of the experience brought by the project with a very positive attitude and having expanded and improved their personal knowledge with respect to the languages and the programming methods used.
In particular the application of general patterns to a complex practical context gave us the ability to setting up in the best way all the individual assigned tasks that led us to compose the final product without overlapping each other during the work. Moreover, assuming this programming approach applied to Threejs/WebGL, we obtained a very good result in the performances of the running application instead of using a naive implementation, without suffering from relevant drops of *FPS*(*Frames Per Second*) or wastes of memory despite having the animations of different indipendent models on the screen.
The application runs smoothly of most type of GPUs, but increasing the number of spaceship that are animated in the scene to a very high value makes the scene slowing down also using very simple models, suggesting that very complex animations and/or a lot of simultaneus operations need a better hardware to make the software run with acceptable performances. We also registered under the execution on machines mounting as operating systems different Linux distributions a radical FPS drop in the Firefox browser as a possible result of a bad behaviour in the usage of the system's GPU drivers. Similar issues haven't been found in another type of main browsers as majority of tests have been taken under Chrome/Chromium browser.

References

[1] MVC Pattern For Building Three.js Applications.

[2] Three.js official documentation *https://threejs.org/docs/*

http://hecodes.com/2016/07/using-mvc-pattern-for-building-complex-three-js-applications/