Final project: Planet Escape

Interactive Graphics 2018/2019 - Sapienza University

Leonardo Sarra

September 22, 2019

Contents

1 Introduction

Figure 1: A screenshot from the game with bonuses and some enemies on screen

Planet Escape is a game developed as the final project of the Interactive Graphics course held at Sapienza University.
Planet Escape is an endless runner type of game with a low-poly style.
The goal is to dodge/destroy the enemies that you, as a spaceship, will encounter along the way and to go as far as possible to establish your record.
At the same time, you will have to manage your spaceship's energy to prevent it from going dry and so losing.

1.1 Specifications

This project during its development had to satisfy some requirements and so it ended up with the following specific:

- Multiple models (8 of which are complex hierarchical models).
- 3 different types of lights.
- Multiple types of textures (color, procedural, image and specular).
- Animations for each model of the game, hierarchical models comes with animations that exploits their structure.

- User interactions: Audio management, Game state/difficulty customization and scoreboard.

In the following chapters I will explain how the requirements were meet and I will analyze the chosen implementation for those features.

1.2 Technologies and tools

The project was developed by using the WebGL Javascript APIs, which are commonly used to develop 2D/3D graphics that are expected to run in a web-browser environment without the need for any additional plugins or browser extensions.
More specifically Planet Escape uses:

- **Three.js (r108)**, a wrapper for the WebGL APIs which simplifies the interactions with the low-level APIs while at the same time offering a wide range of extra features to support game-development.
- **OBJLoader and MTLLoader**, two small libraries built on Three.js which can be used to load OBJ model files and the relative MTL material file.
- **GLTFLoader**, a library built on Three.js which can be used to load the GLTF/GLB format which can store models, materials, textures, and animations in the same file.
- **Dat.GUI**, a library which provides an overlay that can be used by the user to control the underlying graphical application.
- **Stats**, another library of the Three.js framework which is used to analyze at glance the performance of GPU-powered graphical applications.
- **TweenMax**, a high-performance animation library that supports all major browsers.

Regarding the tools, during the development I used **Blender** extensively to do the various hierarchical models for the enemies and player vehicles, those models were later exported in a GLTF file ready to be imported into the game. **Krita** was also used to modify the texture images.

2 Game elements

In this chapter, I will present all the elements that are part of the game, the ideas behind those elements and how they impacted the game experience. I will not go too deep into the implementation details which will be presented later on.

2.1 Scene: Terrain, sky, lights, fog, and camera

The game takes place in a scene which consists of many elements which have no real impact on the game's logic but that help to provide a good looking game.
The first and probably most evident element is the terrain, which is represented as a 3D flat-shaded orangish cylinder, that specific shading technique was chosen to achieve a low-poly style of this mesh.
When the flat shading is used, each vertex of each triangle of the polygon mesh is chosen as the key vertex for that triangle and then the illumination equation is performed to calculate the illumination value for that vertex, and the entire triangle is invested with a copy of that value, the ending result is an approximation of the illumination that could be achieved using different and more realistic interpolation techniques like Phong and Gouraud's ones.

Considering that a standard cylinder would have looked a bit dull in the scene, I opted to dynamically add new vertices and slightly change the position of all the vertices in order to simulate a jagged terrain.
This cylinder will be seen from the side using a perspective camera which is placed in a way that only half of the cylinder can be seen, this was necessary to sell the illusion that the cylinder is indeed some sort of terrain.
The scene also contains a white fog which helps in hiding what is on the horizon like the end of the terrain, the fog will also influence the look of other elements of the scene, like the clouds, based on their distance from the camera.
The clouds that populates the scene have a dodecahedron shape and a white texture color, they are not flat-shaded (uses Phong shading instead) nor have a specular map, this was intended to prevent them from popping up too much in the scene which could have distracted the user from seeing other elements of the game that have an impact on the gameplay (e.g: bonus and enemies).
The clouds are shattered around the cylinder at a different distance from the

camera and sometimes can have multiple dodecahedron objects as children (up to 4 objects) to form a bigger cloud.
The scene also has 3 lights:

- An hemisphere light, a light source positioned directly above the scene, with color fading from the sky color to the ground color and that cannot cast shadows.
  The color that opted for is not exactly the sky color but a lighter blue (Alice Blue), If I ended up using a standard blue the scene would have looked too "cold".

- A direction light, a light that gets emitted in a specific direction.
  This light will behave as though it is infinitely far away and the rays produced from it are all parallel. It is used in the game to simulate the light coming from an off-screen sun.

- Ambient light, a light that globally illuminates all objects in the scene equally.
  It cannot be used to cast shadows as it does not have a direction.
  For the game, I opted to use a pink tonality for this light to make the screen looks "warmer". Note that the intensity of this light will increase as time passes.

Figure 2: Scene background texture

The scene will also have a procedural texture as its background, the texture consists of a progressive transition (linear gradient) between two shades of blue across a line.

2.1.1 Animations

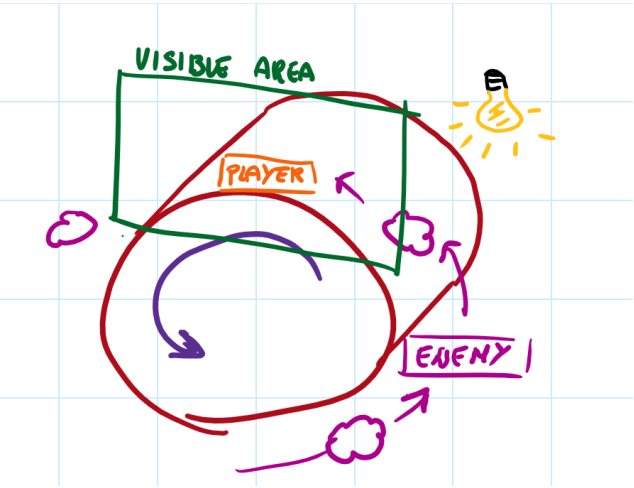


Figure 3: Doodle of the scene with its basic animations

In order to sell the illusion of movement to the player, different rotations animations were implemented.
The terrain, for example, is subject to two different transformations, the first one is a constant rotation on itself and can be spotted right away, the second one, instead, is very subtle and consist in changing the conformation of the terrain when a player reaches a new level (vertices' positions are randomly changed).
The clouds also come with two transformations, one is a translation around the cylinder and the other one is a rotation on the side nearer to the camera. The animation speed is influenced by the number of object children a cloud has.

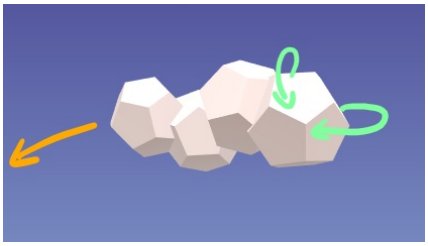
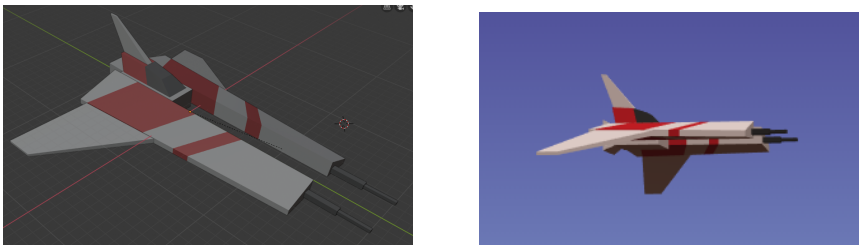


Figure 4: Cloud animation drew in

2.2 Player vehicles

The player during the game has the ability to control his vehicle and move its model across the playable area, the ship by default gets spawned in the middle of the screen.
For the game four different vehicles were developed, those vehicles comes with complex hierarchical models and use texture images/color and different material properties (roughness, sheen tint, and no shininess) with the objective to create a model that doesn't obstruct distract the player while its playing.
Note that each vehicle is different concerning movement, energy consumption, and abilities pool, this could make a specific spaceship more or less hard to play depending on the player's playstyle and skills.

2.2.1 Spaceship 1 (Standard vehicle)



(a) Spaceship1 in Blender **(b)** Spaceship1 in-game

Figure 5: Spaceship1 model

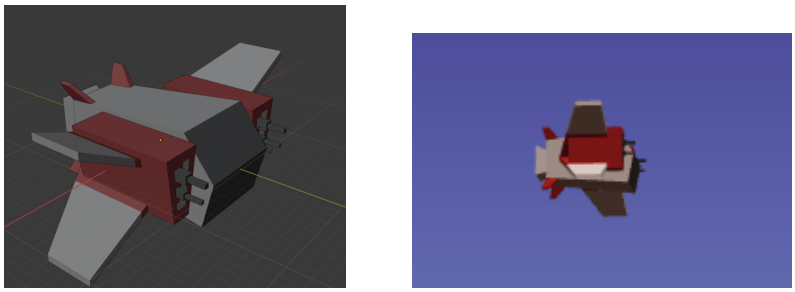
By default, the player will use the spaceship n.1 which is the standard vehicle, a well-balanced one with good maneuverability and energy consumption, not too fast nor too slow.
The vehicle is made out of 9 parts which composes its hierarchical model, the relations are as follow:

.

```
mainHull
├── engine
├── topHull
├── wingLeft
│   ├── wingFarLeft
│   └── baseCannonLeft
│       └── cannonLeft
└── wingRight
    ├── wingFarRight
    └── baseCannonRight
        └── cannonRight
```

The standard vehicle uses 3 texture color (red, gray, black) over its elements, those same color will be also used for Spaceship n.2 and n.3.

2.2.2 Spaceship 2



(a) Spaceship2 in Blender **(b)** Spaceship2 in-game

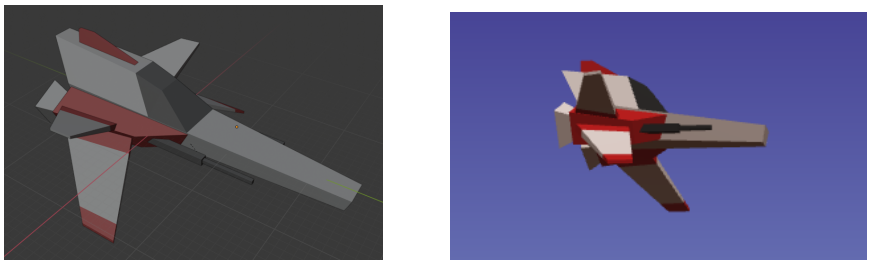
Figure 6: Spaceship2 model

The second player vehicle is smaller than the default one but is also way slower, to compensate that the vehicle is also less energy-intensive.
This spaceship will consume less energy over-time and its shield ability will have a reduced cost.
The model is composed by:

.

```
mainHull
├── boxLeft
│   ├── baseLowerCannonLeft
│   │   └── lowerCannonLeft
│   └── baseUpperCannonLeft
│       └── upperCannonLeft
├── boxRight
│   ├── baseLowerCannonRight
│   │   └── lowerCannonRight
│   └── baseUpperCannonRight
│       └── upperCannonRight
├── engine
├── tailLowerLeft
├── tailLowerRight
├── tailUpperLeft
├── tailUpperRight
├── wingLowerLeft
├── wingLowerRight
├── wingUpperLeft
└── wingUpperRight
```

2.2.3 Spaceship 3

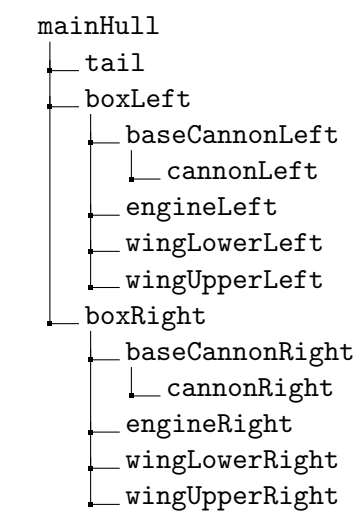


(a) Spaceship3 in Blender **(b)** Spaceship3 in-game

Figure 7: Spaceship3 model

The third spaceship is the fastest one but also the one with the highest energy consumption over-time.
This spaceship is supposed to be used by high skilled players who are able to gather energy no matter how chaotic the situation is.
This spaceship's hierarchical model is composed by the following elements:

.



2.2.4 Bonus spaceship: TARDIS

Up to this point I used color texture with different shading technique and material properties but I never used an image or specular texture.
In order to fully satisfy all requirements, I added an extra vehicle which doesn't follow the low-poly guidelines because of its image texture and its model with a high number of polygons (3700+ vertices).
The vehicle in question is the TARDIS, the spaceship from Doctor Who, a British science-fiction television program produced by the BBC since 1963.
The TARDIS is a simple blue police box, without cannons nor wings like the previous spaceships.
I represented the TARDIS by applying three textures on the model: an image texture for its body, an image and a specular texture for its windows material.

(a) Body texture

(b) Texture for window material

(c) Specular texture for window material

Figure 8: TARDIS's textures

The hierarchical model of the TARDIS is composed by the following objects:

- base
  - doorMain
  - door1
  - door2
  - door3
  - top

Note that the TARDIS is lacking any sort of offensive weapon and so will not be able to fire at the enemies ship, to balance the difficulty I opted to reduce drastically the energy cost over-time while still provide good maneuverability to the ship. The materials used are very similar to the ones of the other ships, they differ only by the specular property which is present exclusively on the TARDIS model.

(a) TARDIS in the Doctor Who TV-Series

(b) TARDIS in Blender

(c) TARDIS in-game (with textures)

Figure 9: TARDIS comparison

2.2.5 Animations

Each vehicle has its animation for translating over the x and y-axis.
The vehicles will always follow the mouse cursor even if they will take some time to adjust to the new position (time required depends on the vehicle), moreover if there is a translation over the y-axis the vehicle will also slight rotate while it arrives at the new y value, rotation that will disappear soon after it reaches the new height. This rotation provides a feeling that each vehicle has its weight that they behave accordingly to what the player would expect.
The TARDIS is a special vehicle and it also comes with a special idle anima-

tion which consists of an ever-running spinning animation which simulates the one that is seen in the British TV series.
The other vehicles will have, instead, another exclusive animation which comes in the form of smoke, while the ship is going it will produce small clouds of smoke behind its engine.
These clouds of smokes have an icosahedron shape are flat-shaded, comes with a white specular color and have a gray texture color.

Figure 10: Smoke in-game

The smoke comes is animated by putting together 3 different animations:

- Spawn animation: the smoke spawns being invisible and will become more visible as the time passes (Fade in).
- Translation animation: the smoke spawns near the engine but will soon start to translate over the x-axis for a specific distance and the y-axis for a random quantity.
- Rotation animation: while the smoke translates it will also rotate on itself

Note that the smoke frequency and translation of the smoke will increase in relation to the level that the player is in.
In case the game over conditions is met the vehicle will fall to the terrain and will rotate on one of its sides, the smoke will have a black tint and will also increase its frequency and will have a bigger model.

Figure 11: Falling vehicle

When the ship reaches the terrain it will then explode in 5 smaller red fragments (with tetrahedron shape, flat-shaded and with a white specular color) that go in random directions and that will disappear soon after.

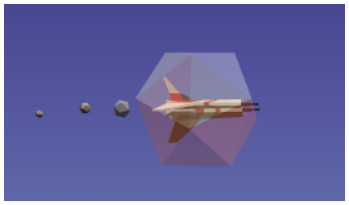
2.3 Player abilities

For the game, I developed two abilities which can be used by the player.
The first one is a shield that when is used enables the user to prevent the ship from exploding when it comes in contact with an enemy.
The activation of the shield cost 20% of the current energy of the ship and will cost extra energy while it's active, once the shield explodes or is disabled it cannot be activated again.
The shield model has an icosahedron shape (flat-shaded and with a white specular color) with a pink color and it "encompasses" and follow the ship's model.
The other ability is the fire one and can be used to destroy enemy ships, each shot will cost 5% of the maximum energy level.
The model of the shot is a longer-than-high box with a green color (flat-shaded and white specular color) and is spawned approximately from the cannons of the player ship and when the shot hits an enemy both the enemy and the shot will disappear from the scene.
Note that the projectiles will spawn from the spaceship's cannons so the number of fired projectiles is equal to the number of cannons that are equipped by the vehicle, this also means that the TARDIS cannot use that ability because it doesn't come equipped with any cannons.

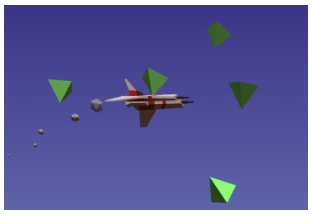
2.3.1 Animations

Both the shield and the shots come with their own animations.
The shield has 4 animations:

- Activation/Cancellation animation: The shield will fade in or out depending on the action
- Idle animation: The shield rotates on itself while active.
- Movement animation: when the vehicle is moving the shield will simulate the rotation and movement of the vehicle.
- Explosion animation: when the shield comes into contact with an enemy, the shield will explode in 5 green tetrahedron fragments (with the same material properties of the shield) that will go in random directions and progressively reduce their size as the time passes.



(a) Shield active



(b) Shield exploding

Regarding the shots, they have 3 animations:

- Spawn animation: the fade-in very quickly starting from the cannon of the ship
- Movement animation: a basic translation over the x-axis
- Explosion animation: when the shot hits an enemy, the enemy will explode in 5 green fragments like the shield explosion animation and both the enemy and projectile will be removed from the scene.

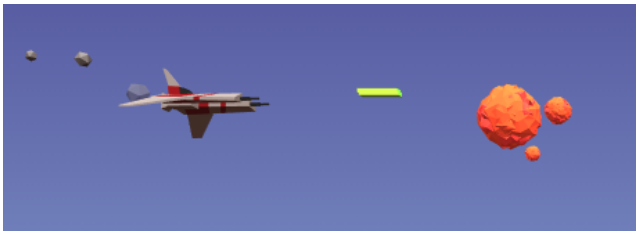


Figure 12: Firing a projectile

2.4 Enemies

During the game, the player will have to be careful to the enemies which may come in different shapes, dimensions and with different speeds, not dodging an enemy will result in a game-over (assuming the shield is not active).
The enemies are divided into 2 two groups: ground enemies and aerial enemies.
The ground enemies are the ones that spawn in the lower half of the playable area and the aerial one are the ones that cover the rest.
The algorithm to choose the next enemy works by generating a random distance from the terrain and based on that it will choose the enemy by picking one of the two enemies of the groups to which the distance belongs.
There are a total of 4 enemies models, each one has a hierarchical model and is subject to animations that exploit its structure.

2.4.1 Airplane

The airplane is one of the ground enemies, it is faster than the other enemies in the same group (pterodactyl) and comes with a mid-sized model.
It is not imported through the usage of the loaders but entirely made using Three.js APIs. Its hierarchical model is composed as such:

.

```
cabin
├── engine
│   └── propeller
│       ├── blade1
│       └── blade2
├── wingleft
├── wingright
├── propeller
├── wheelAxisLeft
│   └── wheelProtectionLeft
│       └── wheelLeft
├── wheelAxisRight
│   └── wheelProtectionRight
│       └── wheelRight
└── suspension
    └── wheelBehind
```

Figure 13: Airplane in-game with propeller animation doodled in

The colors chosen for this model are red, white and brown and its materials come with the same property that were shown before and I think are good when dealing with low-poly styles (flat-shaded, white specular color).

2.4.2 Pterodactyl

The pterodactyl is the second and last enemies that belong to the ground group. Its model is a hierarchical one and it was made using Blender, the models have skewed pivots which were used in order to improve the animations that were implemented using Three.js.
The pterodactyl as an enemy is both mid-sized and big-sized at the same time because when it opens its wings it will temporarily occupy more space on the player screen.
The model is composed of these elements:

.

```
torso
├── neck
│   └── head
├── legLeft
│   └── footLeft
├── legRight
│   └── footRight
├── wingLeft
│   └── handLeft
└── wingRight
    └── handRight
```

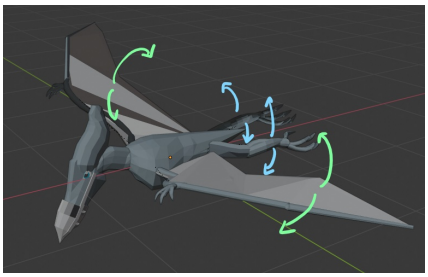


Figure 14: Pterodactyl in Blender with animations doodled in

The pterodactyl has, like all the other models, a low-poly look.
For the body 2 types of gray were used while head also uses black and blue (for the eyes).

2.4.3 Asteroid

The asteroid is one of the enemies that belongs to the aerial group.
It is a mid-sized enemy and it also the enemy in the aerial group with the highest translation speed.
Its model was made in Blender and consist of 1 big rock with 3 smaller rocks that orbit around.
Therefore the hierarchical model is composed of few elements:
.

```
mainAsteroid
├── miniAsteroid1
├── miniAsteroid2
└── miniAsteroid3
```

Each of the mini-asteroid has its pivots put inside main asteroid, by doing so they will easily rotate with different orbits depending on their starting position.

In order to make this enemy more interesting I decided to make it have a dynamic model, this means that the objects of which the layout is made will change over time.
Differents layouts are achieved by removing one or more mini asteroids from the model, there are 4 possible outcomes, each with 25% possibility of occurring, for the model:

- Layout1: All the mini asteroids are visible.
- Layout2: The miniAsteroid1 object is removed.
- Layout3: The miniAsteroid1 and miniAsteroid2 are removed.
- Layout4: The miniAsteroid2 object is removed.

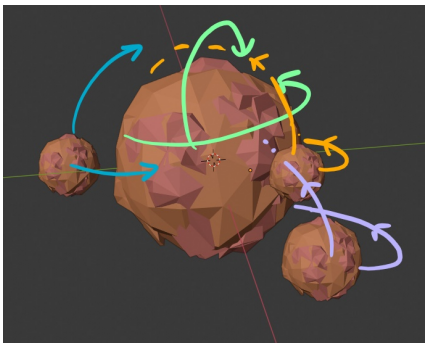


Figure 15: Asteroid in Blender with animations doodled in (and all the mini asteroids visible)

As always the model uses a low-poly style that I adopted but this time it comes with warmer colors (red and orange).

2.4.4 Satellite

The satellite is the last enemies, it belongs to the aerial group, its way larger than the other enemies if its solar panels are considered and it also comes with a slow translation speed.
The satellite's model was done in Blender and comes with custom pivots for some of his parts (parabola and hull).
The model is hierarchical and is composed of:

.

```
topPart
├── middlePart
├── bottomPart
├── wing1
├── wing2
└── lowerParabola
    └── parabola
```

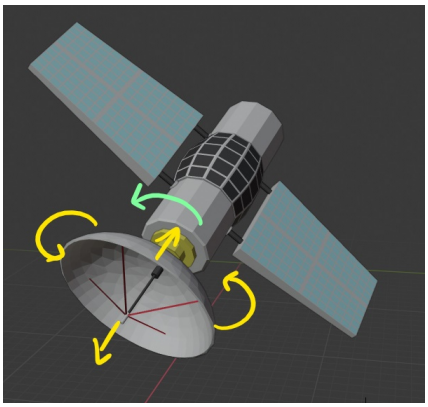


Figure 16: Satellite in Blender with animations doodled in

Like the rest, this model also follows a low-poly style.
This model, together with the pterodactyl, is one of the most challenging for the player because the extremities of the satellite (wing1 and wing2) will occupy a discrete amount of space, distracting the player and reducing the space of maneuver for the ship.
To make things harder for the player, while still making the game less repetitive, I opted to make the satellite spawn in 3 different positions which are intended to create problems to the player when the wings and the parabola animations are running.

2.4.5 Animations

The enemies share some animations regarding movement and their explosion but each one them also comes with unique animations that exploit its hierarchical structure.
Every enemy is spawned below the terrain and is subject to a translation on the x and y axes which will make it rotates around the cylinder to reach the player's vehicle, once an enemy makes at least one ride around the cylinder

it is eligible to be removed from the game and its resources being reused for a new enemy, based on the level the enemy could persist for multiple rides around the terrain.
When enemies are hit by the player vehicle they will explode in 15 red tetrahedron fragments that will go in random directions and slowly reduce their dimension and then disappear.
Regarding the exclusive animations we have the one of the airplane enemy which is by far the most basic one and consist in the propeller rotating, the rotation of the propeller will make rotate also its children (blade1 and blade2).
The pterodactyl enemy instead has animations for its wings and legs, thanks to these animations the pterodactyl will constantly flap its wings (different speeds are used depending on the movement of the wings) and its legs will oscillate to simulate that it's flying.
Considering the structure of the pterodactyl those animations will also influence the position of its feet and hands.
The asteroid uses animations for each of the objects of its structure, the main asteroid will randomly rotate on itself while its mini asteroid will randomly rotate on the main asteroid with random speed and creating differents orbits depending on the mini asteroid that is animated.
Last but not least we have the satellite which will have two animations, the first one covers the parabola which will constantly move and change its alignment and the other ones is for the "topPart" which will rotate on its skewed pivot, this will make the wings, the middle and bottom part rotate and at the same time the animation will slightly move the satellite in the space.

2.5 Bonuses

The bonuses are represented using an animated octahedron, they follow the guidelines for a low-poly look so they have bright colors, are flat shaded and comes with a white specular color which is used to exalt the light on them.
There are 3 types of bonuses, each one with its colors:

- Standard blue bonuses, which restores about 5% of the player's vehicle energy.
- Uncommon green bonuses, which restores 10% of the energy.

- Rare gold bonuses, which restores the energy of the spaceship to 100%.

Figure 17: A bonus for every type

There are some rules behind the spawn of these bonuses; first of all the bonuses are spawned in batches after the player travels for a fixed amount of distance, each bonus of the batch is slightly skewed (both in x and y-axis) in relation to the previous one.
Each bonus has a different probability of spawning there is a 1% probability of getting the gold one, 10% for the green one and 89% for the standard one.
On top of that, there is an extra rule, if a gold bonus is spawned then all the bonuses of the same batch must be of the standard type, by doing so I'm trying to prevent the game from giving too many energy in the same batch which could be exploited at the player advantage.

2.5.1 Animations

The bonus is always spawned below the cylinder that represents the terrain, with a random distance from it, and it will be subjected to two transformations, a translation along the x and y-axis which will make it rotate around the cylinder on top of that it will randomly rotate on itself (in a similar way to the clouds)
In case the player hit a bonus, it will be applied and the bonus will explode in 5 smaller fragments that go in random directions and that will disappear after a few seconds.
Like the enemies, its explosion fragments have a tetrahedron shape, are flat shaded and come a white specular color to increase the visibility. Note that the explosion animation uses the same color of the bonus that is hit, thanks to that the user has a visual cue to understand if he obtained a particular bonus or not even when the game becomes more chaotic and fast.

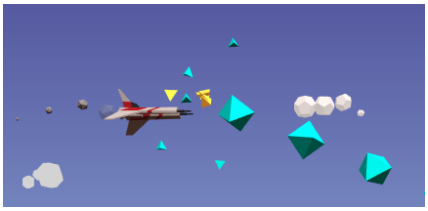


Figure 18: Bonuses exploding

When a bonus does a rotation around the cylinder and it is not visible it is eligible for being disposed and its resources being reused, as for the enemies it may takes multiple rides around the terrain, depending on the level of the game, for it to being actually disposed.

2.6 User interface

The user interface of the project is divided into two part.
The first one is the header in which all sort of useful info will appear, like level, distance traveled, energy remaining and the shield cooldown, there is also a small instruction section in which a summary of the rules of the game is shown.

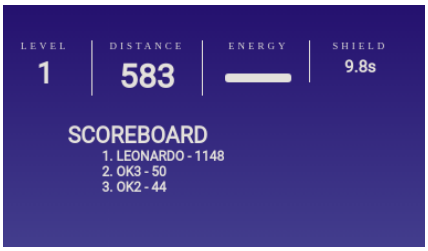


Figure 19: HTML Header with scoreboard visible

Below the header there is the playing area, here the player vehicle is able to move freely and all the enemies and bonuses will be shown in these boundaries, in the playing area there is also the “game has ended” message and the scoreboard which can only appear when the game-over conditions are met (vehicle destroyed or energy depleted) and will disappear when the game is restarted.
On top of these two parts, there is an overlay (made using the dat.gui library) which is supposed to be used by the player to customize his game experience (by influencing the game behavior), if needed this dynamic UI can be collapsed or resized to not obstruct the vision of the coming enemies.

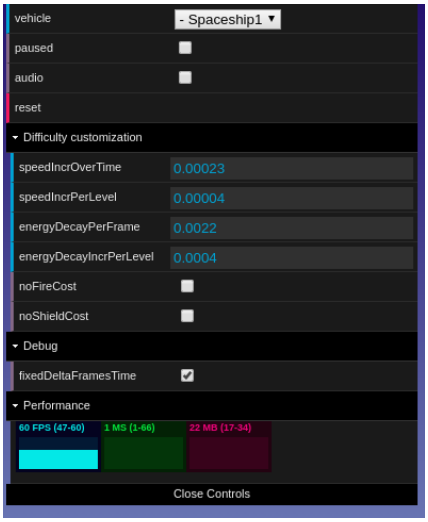


Figure 20: Overlay

2.7 User input and interactions

The user can control its player character through the usage of keyboard and mouse. The mouse is used for the sole purpose of moving the character while the keyboard has 4 main purposes:

- **"A"** key, is the "Attack" button. When this button its pressed the ship will fire a laser on a straight line, if the laser hit an enemy it will explode and the shot will cease to exist.
  Firing a shot will cost the player 5% of the maximum spaceship energy level.

- **"S"** key, is the "Shield" button. If this button is used a shield around the spaceship will be deployed, this shield will protect the player from one hit with an enemy.
  Note that the shield comes with an activation cost which 20 % of the current energy level, and while it is active the energy will deplete faster.
  If the user wants to deactivate the shield it can do so by pressing the "S" key again.
  After the shield is disabled or destroyed it will go offline for 10 seconds and can't be brought online again in this timeframe, this was done to prevent the players from abusing the mechanics.

- If the game has ended the **"S"** key can be used to add your name to

the scoreboard, by doing so you will save your record and you will be able to compare in a future playthrough.

- If the game has ended the **"R"** key can be used to restart the game and play again.

The player can also interact with how the game works by using the overlay UI on the right which will enable him to:

- Choose its spaceship (from a pool of 4), changing the spaceship will influence the difficulty and the gameplay because, as already shown, each spaceship has its own 'quirks'.
- Manage the background audio track which is disabled by default.
- Pause or reset the game.
- Customize the difficulty by changing the values of the speed increment over time, the speed increment per level, the decay of the energy per frame, the bonus decay energy per level.
  Eventually the player can also disable the activation cost of the energy shield or the cost of firing a shot to destroy an enemy.
- Enable/disable dynamic animations that depends on the delta time between rendered frames (disabled by default).
  Enabling dynamic animations ensures a smooth game even in case the client's performance fluctuates, on another end animations' quality may drop drastically on low-end machines.
  Instead, if the animations don't depend on the delta time they will be extremely smooth but the game may be subject to slowdowns on less powerful machines.
- Check game performance (FPS, memory, and latency).

Note that if the user changes spaceship mid-game or disable the energy costs of the shield/projectiles the game will automatically reset its score to prevent any tampering with the scoreboard.

3 Implementation

For the implementation of the game I wanted to adopt a modular approach that used an object-oriented paradigm.
This choice can be seen right away by the fact that every element of the game is implemented in its own .js file and is declared as an object (the only exception being the init and scoreboard .js files) and so can be easily replaced with something different as long as the APIs remain the same.
In order to work the HTML page is supposed to include every js file game-related, otherwise the game will crash during the play-through or not boot correctly.
When developing the project I also wanted to provide an implementation that is good from the point of view of performance, to reach this goal I intensively used holders which enabled me to reuse resources (models and particles) instead of instantiating a new object.
Last but not least I wanted to reduce as much as possible the loading times of the game and to do so I introduced the "EnemyMeshStorage", an object which loads, in background, the models and texture of the enemies and works like a cache after the resources are loaded.
While the resources are not ready I make the users play a simplified version of the game which doesn't contain the enemies or bonuses, once the resources are loaded in the background the real game will begin and hopefully the player will not notice this small delay.
In this chapter, I will present the solutions that were adopted for the problems that I have mentioned and the most relevant parts of the game which were seen in the previous chapter.

3.1 Game init

The game starts by initializing all the data structures needed for its correct execution and by starting the WebGL render loop. After the webpage has been loaded the following init function is triggered (onLoad event):

```
window.addEventListener('load', init, false);
function init() {
    initHTMLUI();
    initMeshStorage();
    resetGame(); // Init data structures or reset them to standard values
    initDatGUI();
    initScene();
    initLights();
    initTerrain();
    initSky();
```

```
    initParticlesHolder();
    initBonusesHolder();
    initProjectilesHolder();
    initEnemiesHolder();
    setupPlayerInputListener();
    htmlUI.loadingMessage.style.display = 'none'; // Hide loading message
    loop(); // starts render loop
}
```

Most of these methods are used to instantiate objects that are later used in the render loop except for "*resetGame*()" which instantiates the "game" data structure whose values heavily influences the game logic and covers all most of the mechanics of the game.

```
function resetGame() {
    if (game == undefined) {
        game = {};
    }
    if (typeof (game.vehicle) != 'undefined') scene.remove(game.vehicle);
    game.hasShield = false;
    game.energyDecayPerFrame = 0.0022;
    game.energyDecayIncrPerLevel = .000040;
    game.level = 1;
    game.energy = 100;
    game.firstMeshesSpawned = false;
    game.vehicle = undefined;
    game.targetBaseSpeed = .00035;
    game.vehicleInitialSpeed = .00035;
    game.baseHeigth = 100;
    game.speedIncrement = .0000027;
    game.levelSpeedIncrement = .000040;
    game.bonusLastSpawn = 0;
    game.enemiesLastSpawn = 0;
    game.speedLastUpdate = 0;
    // More declarations that I'm skipping to keep this short
    game.vehicleAdjustmentPositionSpeed = 0.005;
    game.vehicleAdjustmentRotationSpeedZ = 0.0008;
    game.vehicleAdjustmentRotationSpeedX = 0.00001;
    game.discountEnergyCost = 1;
    htmlUI.level.innerHTML = 1;
    htmlUI.distance.innerHTML == 0;
    htmlUI.energy.style.right = (100 - game.energy) + "%";
    htmlUI.energy.style.backgroundColor = (game.energy < 50) ? "#f25346" : "#68c3c0";
    htmlUI.shield.innerHTML = "Ready";
    createVehicle(vehicleType);
    if (typeof (sound) != 'undefined' && audioStarted) {
        sound.stop();
        if (options.audio) sound.play();
    }
    if (typeof (ambientLight) != 'undefined') ambientLight.intensity = 0.4;
}
```

As can be seen, "*resetGame*()" can both instantiate the "game" object or reset it to its default values which covers almost everything related to the game, it also takes care of restarting the audio source if available or updating

the game HTML UI and restoring the intensity of the lights to the default value.

3.2 Audio management

The game audio is activated through the dat.gui overlay from which the user could use a checkbox to easily enable or disable it, this will trigger an *onChange* javascript event and its listener that will take care of applying the new setting.
Under the hood the audio was implemented using the Three.js APIs

```
function initDatGUI() {
    ...
    gui.add(options, "audio").onChange(function () {
        if (options.audio) {
            audioListener = new THREE.AudioListener();
            camera.add(audioListener);
            sound = new THREE.Audio(audioListener);
            startAudio();
        }
        else {
            if (typeof (sound) != 'undefined') sound.stop();
            audioStarted = false;
        }
    });
    ...
}
function startAudio() {
        var audioLoader = new THREE.AudioLoader();
        console.log("starting audio...");
        audioLoader.load('music/background.ogg', function (buffer) {
            sound.setBuffer(buffer);
            sound.setLoop(true);
            sound.setVolume(1);
            sound.play();
            audioStarted = true;
            console.log("audio started");
        });
}
```

Through the Three.js's *Audio*, *AudioListener* and *AudioLoader* objects I can create a source, attach it to the camera and load an OGG audio file, I also decleare some properties that will change its reproduction (like volume and loop flag).
The audio that is loaded will continue playing until the game run ends or the user disables it again.
The audio is opt-in to comply with the new guidelines of the major web-browsers which forces web pages to not have an audio track starts automatically unless the users have given permission through some kind of input.

3.3 Player input management

```
function setupPlayerInputListener() {
    document.addEventListener('mousemove', handleMouseMove, false);
    document.addEventListener('keyup', (e) => {
        if (e.which == 83 && game.showReplay) {
            addPlayerScore();
        } else if (e.which == 83 && !game.showReplay) {
            if (!game.hasShield) addShield();
            else disableShieldImmunity(false);
        } else if (e.which == 65 && !game.gameOver
                    && game.vehicle != undefined && vehicleType != 3) {
            fireShot();
        } else if (e.which == 82 && game.showReplay) resetGame();
    });
}

var mousePos = { x: 0, y: 0 };
function handleMouseMove(event) {
    var tx = -1 + (event.clientX / WIDTH) * 2;
    var ty = 1 - (event.clientY / HEIGHT) * 2;
    mousePos = { x: tx, y: ty };

}
```

The player gives input to the game through 2 devices, the keyboard which triggers the "keyup" event and the mouse which triggers "mousemove" event.
Depending on the key pressed and moment it got pressed the listener could end up adding/removing a shield to/from the vehicle, it could start the game again, fire a projectile or open the form to add your name to the scoreboard.
The mousemove event will trigger, instead, a function that will keep the mousePos data structure updated with the cursor position in a way that is not dependent to the screen size, this info will be later used in the render loop to move the spaceship accordingly.

3.4 Scene init

In the function "*init*()" that was presented before, multiple init functions were called, some of them handles the elements of the scene that we presented in the previous chapter. For example, the camera, the fog and the render are instantiated in the "*initScene*()" function.

```
function initScene() {
    HEIGHT = window.innerHeight;
    WIDTH = window.innerWidth;
    scene = new THREE.Scene();
    scene.fog = new THREE.Fog(Colors.gray, 100, 950);
```

```
    aspectRatio = WIDTH / HEIGHT;
    fieldOfView = 60;
    nearPlane = 1;
    farPlane = 10000;
    camera = new THREE.PerspectiveCamera(
        fieldOfView,
        aspectRatio,
        nearPlane,
        farPlane
    );
    camera.position.x = 0;
    camera.position.z = 200;
    camera.position.y = 150;
    renderer = new THREE.WebGLRenderer({
        alpha: true,
        antialias: true
    });
    renderer.setSize(WIDTH, HEIGHT);
    renderer.shadowMap.enabled = true;
    container = document.getElementById('world');
    container.appendChild(renderer.domElement);
    window.addEventListener('resize', handleWindowResize, false);
}

function handleWindowResize() {
    HEIGHT = window.innerHeight;
    WIDTH = window.innerWidth;
    renderer.setSize(WIDTH, HEIGHT);
    camera.aspect = WIDTH / HEIGHT;
    camera.updateProjectionMatrix();
}
```

To do so we use the *Three.Scene*, *Three.Fog* and the *Three.PerspectiveCamera* objects.
The render uses by default a transparent background.
By doing so I can use a CSS style-sheet to make the procedural texture with the linear gradient function and use it as the game background, here is a snippet of CSS code that covers the creation of the texture:

```
.game-holder {
position: absolute;
width: 100%;
height: 100%;
background: -webkit-linear-gradient(#210f72, rgb(173, 210, 245));
background: linear-gradient(#210f72, rgb(173, 210, 245) );
}
```

The fog is added directly to the *Three.Scene* object and is declared by passing its color and two floats that are used to express at which distance the fog is supposed to be applied and the distance after which it can be ignored.
The perspective camera is declared like in WebGL by passing the fov, the aspect ratio, the near plane, and the far plane.
The method also takes care of adding a listener to update the camera aspect and the dimensions at the render level in case the browser window is resized

by the user.

After the camera and the render are all set up, the main init function will take care of the lights which are initialized using the various constructors provided by Three.js which are specific for the light type (hemisphere, direction, and ambient light).

```
function initLights() {
    shadowLight = new THREE.DirectionalLight(Colors.white, 1);
    shadowLight.position.set(160, 340, 350);
    shadowLight.castShadow = true;
    shadowLight.shadow.camera.left = -SHADOW_CAMERA_POS;
    shadowLight.shadow.camera.right = SHADOW_CAMERA_POS;
    shadowLight.shadow.camera.top = SHADOW_CAMERA_POS;
    shadowLight.shadow.camera.bottom = -SHADOW_CAMERA_POS;
    shadowLight.shadow.camera.near = NEAR_PLANE;
    shadowLight.shadow.camera.far = FAR_PLANE/10;
    shadowLight.shadow.mapSize.width = SHADOW_MAP_SIZE;
    shadowLight.shadow.mapSize.height = SHADOW_MAP_SIZE;
    scene.add(shadowLight);
    hemisphereLight = new THREE.HemisphereLight(Colors.aliceBlue, 0x000000, 1)
    ambientLight = new THREE.AmbientLight(Colors.pink, .4);
    scene.add(hemisphereLight);
    scene.add(ambientLight);
}
```

Note that the shadow camera used to generate the depth map of the scene is manually configured, together with the shadow resolution.
Now the only two remaining elements that are waiting to be configured through their init functions are the sky and the terrain.

```
function initTerrain() {
    terrain = new Terrain();
    terrain.mesh.position.y = -550;
    scene.add(terrain.mesh);
}

function initSky() {
    sky = new Sky(game);
    sky.mesh.position.y = -600;
    scene.add(sky.mesh);
}
```

The terrain and sky objects comes with their mesh variable which is nothing more than a Three.Object3D that contains multiple related elements (cylinder for the terrain and clouds for sky).
Let's analyze how terrain is created in the Terrain object.

```
Terrain = function () {
    var geom = new THREE.CylinderGeometry(600, 600, 800, 40, 10);
    geom.applyMatrix(new THREE.Matrix4().makeRotationX(-Math.PI / 2));
    geom.mergeVertices();
```

```
    var l = geom.vertices.length;
    this.verts = [];
    for (var i = 0; i < l; i++) {
        var v = geom.vertices[i];
        this.verts.push({
            y: v.y,
            x: v.x,
            z: v.y,
            ang: Math.random() * Math.PI * 2,
            amp: 5 + Math.random() * 15,
            speed: 0.016 + Math.random() * 0.032
        });
    };
    var mat = new THREE.MeshPhongMaterial({
        color: Colors.brown,
        transparent: false,
        opacity: .9,
        flatShading: true,
    });

    this.mesh = new THREE.Mesh(geom, mat);
    this.mesh.receiveShadow = true;
    this.moveVerts()
}

Terrain.prototype.moveVerts = function () {
    // function to find a new position for the vertices
}
```

The terrain is created by using the $THREE.CylinderGeometry$, rotating the cylinder, adding extra vertices to the geometry and then adding a $THREE.MeshPhongMaterial$ with colors and other properties.
The Sky follows a very similar approach but instead of instantiating a cylinder it instantiates multiple clouds (Cloud object) which uses $THREE.DodecahedronGeometry$ instead.

3.5 Resource loading

The resource loading takes places through the usage of the $OBJLoader$, $MTLLoader$ and the $GLTFLoader$.
The first two are used to load the 3d model saved as a .obj file and the materials saved as .mtl eventually they will also load image textures and specular maps (like the textures used for the TARDIS).
The $GLTFLoader$, instead, loads .gltf and .glb files which may contain in a single package model, materials and texture (eventually even built-in animations, not used in this project).
These two libraries were heavily used to load the player vehicles like in this snippet:

```
function createVehicle(vehicleType) {
    var file = null;
```

```
    if (vehicleType == 0) {
        file = "spaceship1";
        game.vehicleAdjustmentPositionSpeed = 0.004;
        game.vehicleAdjustmentRotationSpeedZ = 0.0008;
        game.vehicleAdjustmentRotationSpeedX = 0.00001;
        game.discountEnergyCost = 1;
    }
    else if (vehicleType == 1) {
        file = "spaceship2";
        ... // setup vehicle specific speeds and abilities costs
    }
    else if (vehicleType == 2) {
        file = "spaceship3";
        ... // setup vehicle specific speeds and abilities costs
    }
    else if (vehicleType == 3) {
        file = "TARDIS";
        ... // setup vehicle specific speeds and abilities costs
    }
    if (vehicleType == 3) { // TARDIS
        var mtlLoader = new THREE.MTLLoader();
        mtlLoader.load('models/' + file + '.mtl', function (materials) {
            materials.preload();
            var loader = new THREE.OBJLoader();
            loader.setMaterials(materials);
            loader.load(
                'models/' + file + '.obj',
                function (object) {
                    game.vehicle = object;
                    scene.add(object);
                    ... // set rotation, scale
                    scene.add(game.vehicle);
                }, ...
            );
        );
    } else { // SPACESHIP1,2,3
        var gltfLoader = new THREE.GLTFLoader(THREE.DefaultLoadingManager);
        gltfLoader.load('models/' + file + '.glb',
            function (gltf) {
                game.vehicle = gltf.scene;
                scene.add(gltf.scene);
                ... // set rotation, scale
                .. // traverse children and castShadow property to true
            },
            ...
            );
    }
}
```

A similar approach is used for the enemies, except that the model is cached through the EnemyMeshStorage layer. This particular object handles the loading and provides APIs for creating a new instance of a mesh without reloading it again.

```
EnemyMeshStorage = function () { }

EnemyMeshStorage.prototype.load = function () {
```

```
    var storage = this;
    storage.airplane = new AirPlane();
    storage.airplane = this.airplane.mesh;
    var gltfLoader = new THREE.GLTFLoader(THREE.DefaultLoadingManager);
    loadAsyncModel(gltfLoader, 'models/pterodactyl.glb');
    loadAsyncModel(gltfLoader, 'models/asteroid.glb')
    loadAsyncModel(gltfLoader, 'models/satellite.glb')
    storage.airplane = new AirPlane();
    storage.airplane = this.airplane.mesh;
}

EnemyMeshStorage.prototype.getAsteroidMesh = function () {
    if (!this.isReady()) return;
    return this.asteroid.clone();
}

... // More getters

//Are all models loaded?
EnemyMeshStorage.prototype.isReady = function () {
    if (this.airplane != undefined && this.asteroid != undefined
    && this.satelite != undefined && this.pterodactyl != undefined) return true;
    else return false;
}
```

Note that the airplane model is not coming from the GLTF loader because it is entirely made in Javascript through the usage of multiple $THREE.BoxGeometry$ which are linked to compose its hierarchical model.
As for the Airplane model, not all the 3D objects in the game are coming straight out from a GLTF or OBJ file, take for example the projectiles, the particles, the bonuses, and the clouds.
Those 3D objects are instantiated in a synchronous way when the Particle/Bonus/Projectile/Cloud constructor is called through the use of different $THREE.Geometry$ and $THREE.Material$ objects each one representing the properties and shapes that were discussed in the previous chapter.

```
Bonus = function () {
    var geom = new THREE.OctahedronGeometry(5, 0);
    var mat = new THREE.MeshPhongMaterial({
        color: 0x009999,
        shininess: 0,
        specular: 0xffffff,
        flatShading: true
    });
    this.mesh = new THREE.Mesh(geom, mat);
    this.mesh.castShadow = true;
    this.angle = 0;
    this.dist = 0;
}
...

Cloud = function () {
  this.mesh = new THREE.Object3D();
  this.mesh.name = "cloud";
```

```
  var geom = new THREE.DodecahedronGeometry(20, 0);
  var mat = new THREE.MeshPhongMaterial({
    color: Colors.white,
  });
  var nChildren = 3 + Math.floor(Math.random() * 3);
  for (var i = 0; i < nChildren; i++) {
    var mesh = new THREE.Mesh(geom.clone(), mat);
    ... // set mesh position
    var scaleFactor = .1 + Math.random() * .8;
    mesh.scale.set(scaleFactor, scaleFactor, scaleFactor);
    mesh.castShadow = true;
    mesh.receiveShadow = true;
    this.mesh.add(mesh);
  }
}
...

Projectile = function (holder) {
    var geom, mat;
    this.holder = holder;
    var geom = new THREE.BoxGeometry(7, 0.75, 2);
    var mat = new THREE.MeshPhongMaterial({
        color: 0x7CFC00,
        shininess: 0,
        specular: 0xffffff,
        flatShading: true,
        opacity: 0.0,
        transparent: true,
    });
    this.mesh = new THREE.Mesh(geom, mat);
}

...

Particle = function (isSmoke, holder) {
  var geom, mat;
  this.holder = holder;
  if (isSmoke) {
    geom = new THREE.IcosahedronGeometry(3, 0);
    mat = new THREE.MeshPhongMaterial({
      color: 0x009999,
      shininess: 0,
      specular: 0xffffff,
      flatShading: THREE.FlatShading,
      opacity: 0.0,
      transparent: true,
    });


  } else {
    geom = new THREE.TetrahedronGeometry(3, 0);
    mat = new THREE.MeshPhongMaterial({
      color: 0x009999,
      shininess: 0,
      specular: 0xffffff,
      flatShading: THREE.FlatShading,
      transparent: false,
    });
  }
```

```
  this.mesh = new THREE.Mesh(geom, mat);
  if (isSmoke) {
    this.mesh.castShadow = true;
    this.mesh.receiveShadow = false;
  }
}
```

3.6 Management of projectiles, particles, enemies and bonuses using holders

The projects uses multiple 3D objects which are required to be in the scene for little time and that sometimes are constantly recreated.
For this reason, the project uses the concept of a holder which provides a transparent layer to the developer to request the objects that he wants to add to the scene.
The holder is instantiated with a default number of objects that are put inside an availability pool, when a request comes if there is any available object in the pool the holder will retrieve it, otherwise a new object will be created.
After an object is used it gets re-added to the pool containing the available objects and so the resources will be recycled.
This approach provides a good performance boost, in particular on low-end machines, after the pool got stabilized to a specific length.
The project uses a $ProjectilesHolder$, a $ParticlesHolder$, an $EnemiesHolder$ and a $BonusHolder$.
Let's take as an example the $ProjectilesHolder$

```
ProjectilesHolder = function (nProjectiles) {
    this.mesh = new THREE.Object3D();
    this.projectilesPool = [];
    this.activeProjectiles = [];
    for (var i = 0; i < nProjectiles; i++) {
        var projectile = new Projectile(this);
        this.projectilesPool.push(projectile);
    }
}

ProjectilesHolder.prototype.spawnParticles = function (pos) {

    if (this.projectilesPool.length) {
        projectile = this.projectilesPool.pop();
        projectile.mesh.material.opacity = 0;
    } else {
        projectile = new Projectile(this);
    }
    this.mesh.add(projectile.mesh);
    projectile.mesh.position.y = pos.y;
    projectile.mesh.position.x = pos.x;
    projectile.fire(pos);
}
```

```
ProjectilesHolder.prototype.checkCollisions = function (pos) {
    /** Code to check if there is a collision
    over the elements in the activePoolProjectiles **/
}
```

This particular holder comes with its custom constructor that creates the related data structures to hold the Projectile objects and provides an API "*spawnProjectiles*()" which can be used to add new projectiles to the scene (the argument is the spawn position) and a "*checkCollisions*()" API which is used to check if a projectile hit something at a particular position.
This holder (like the others) instantiate a mesh which will contains all the visible elements that it is holding, similar to what we have seen in the Sky object, this *Object*3*D* mesh object is added to the scene in the init of the game, after that it will never be removed.
The other holders are very similar but usually comes with small differences like the type of objects that are held (*Particle*, *Enemy*, and *Bonus*) and the APIs which don't always have a "*checkCollisions*()", sometimes an "*animate*()" method is present instead (in case of *EnemiesHolder* and *BonusesHolder*).
The *ParticleHolder* is a special case because it holds two types of particles (Smoke and exploding fragments) each with its own pool in order to optimize further the performance considering that both smoke and exploding fragments are extensively used within the game.

3.7 Animations and collisions

The animations of the game are handled in two ways, the first is by handling them frame by frame using javascript code and Three.js APIs, the second one is through the use of the TweenMax library.
The first approach is used for the vehicles' movements, the enemies, the shield, the bonuses, the clouds, and terrain animations while the second is used primarily for the projectiles, the smoke, and the explosion effects.
Let's start with analyzing the animations done in Javascript, in particular, the ones for the vehicles.

```
function updateVehicle() {
    if (game.gameOver || game.vehicle == undefined) return;
    var maxY = 250;
    if (vehicleType == 3) maxY = 242;
    var targetY = normalize(mousePos.y, -.75, .75, 75, maxY);
    var targetX = normalize(mousePos.x, -1, 1, -150, 150);
    game.vehicle.position.y += (targetY - game.vehicle.position.y)
        * game.vehicleAdjustmentPositionSpeed * deltaTime;
    game.vehicle.position.x += (targetX - game.vehicle.position.x)
```

```
        * game.vehicleAdjustmentPositionSpeed * deltaTime;
    game.vehicle.rotation.z = (targetY - game.vehicle.position.y)
        * game.vehicleAdjustmentRotationSpeedZ * deltaTime;
    if (vehicleType != 3) game.vehicle.rotation.x = (game.vehicle.position.y - targetY)
        * game.vehicleAdjustmentRotationSpeedX * deltaTime;
    if (typeof (game.bubble) != 'undefined') {
        game.bubble.position.y = game.vehicle.position.y;
        if (vehicleType == 3) game.bubble.position.y = game.bubble.position.y + 8;
        game.bubble.position.x = game.vehicle.position.x;
        game.bubble.rotation.z = game.vehicle.rotation.z;
        game.bubble.rotation.x = game.vehicle.rotation.x;
    }
}
```

The vehicle animations are implemented in the "*updateVehicle*()" method which normalize the mouse input over the playable area and then apply a translation and a rotation based on the adjustment speed of the vehicle type that the player is using, on top of that the "bubble" object (the shield) is rotated and translated to mimic the movements of the ships that it is protecting.
The shield has also an extra set of animations which are defined in the "*handleShield*()" method

```
function handleShield(deltaTime) {
    if (game.bubble == undefined) return;
    if (game.shieldCooldown != 0 && !game.gameOver) {
        game.shieldCooldown = game.shieldCooldown - deltaTime;
        if (game.shieldCooldown < 0) game.shieldCooldown = 0;
    }

    game.bubble.rotation.z += .002 * deltaTime;
    game.bubble.rotation.y += .002 * deltaTime;
    if (game.hasShield == true && game.bubble.material.opacity < 0.3) {
        game.bubble.material.opacity += 0.01;
        game.bubble.material.visible = true;
    } else if (game.hasShield == false && game.bubble.material.opacity > 0) {
        game.bubble.material.opacity -= 0.01;
    }
    if (game.bubble.material.opacity < 0) {
        game.bubble.material.opacity = 0;
        game.bubble.material.visible = false;
    }
}
```

This function will handle the cooldown of the shield, its rotation over the z and y axis and the fade in/out effect that is applied when the shield is added/removed to/from the vehicle.
Note that some of the animations that we have seen depend on the *deltaTime* variable which represents the time that has passed between the frames, thanks to that the behavior of the animations dynamically change to adapt to the client's performance.
Now let's consider the animations of the sky and the terrain, which follow the same approach that was just shown.

For the sky object and its clouds we use the "*moveClouds*()" method which rotates the Sky mesh while still triggering the animation of the single clouds by calling the rotate method on each of them.

```
Sky.prototype.moveClouds = function () {
    for (var i = 0; i < this.nClouds; i++) {
        var cloud = this.clouds[i];
        cloud.rotate();
    }
    this.mesh.rotation.z += this.game.baseSpeed * this.game.deltaTime;
}

Cloud.prototype.rotate = function () {
  var len = this.mesh.children.length;
  // Rotate each children of the cloud
  for (var i = 0; i < len; i++) {
    var childrenMesh = this.mesh.children[i];
    var animationBoost = (i+1)*0.5;
    childrenMesh.rotation.z += Math.random() * .0055 * animationBoost;
    childrenMesh.rotation.y += Math.random() * .00225 * animationBoost;
  }
}
```

The terrain instead can be easily rotated by using the following code in the render loop.

```
terrain.mesh.rotation.z += game.baseSpeed * deltaTime;
```

Bonuses animations work in a similar way to what was seen for the sky and so I will not analyze it.
The most complex animations by far are the ones used for the enemies, this is due to the fact that those animations exploit enemies' complex hierarchical models.
The enemies are animated in the render loop through the call of the "*animate*()" API of the EnemiesHolder

```
EnemiesHolder.prototype.animateEnemies = function () {
  for (var i = 0; i < this.enemiesInUse.length; i++) {
    var enemy = this.enemiesInUse[i];
    // Rotation around the terrain
    var speedBonus = (enemy.type == 0 || enemy.type == 1) ? 0.05 : 0.0;
    enemy.angle += this.game.baseSpeed * this.game.deltaTime * (0.6 + speedBonus);
    if (enemy.angle > Math.PI * 2) enemy.angle -= Math.PI * 2;
    enemy.mesh.position.y = -650 + Math.sin(enemy.angle) * enemy.distance;
    enemy.mesh.position.x = Math.cos(enemy.angle) * enemy.distance;
    if (enemy.type == 0) { // Airplane animation
      enemy.propeller.rotation.x += .7 * deltaTime;
    } else if (enemy.type == 1) { // Asteroid animations
      enemy.mesh.rotation.z += .0010 * game.deltaTime;
      enemy.mesh.rotation.y += .0010 * game.deltaTime;
      enemy.miniAsteroid1.rotation.y += .0015 * game.deltaTime;
      enemy.miniAsteroid1.rotation.x += .0015 * game.deltaTime;
      enemy.miniAsteroid2.rotation.y += .0025 * game.deltaTime;
      enemy.miniAsteroid2.rotation.x += .0025 * game.deltaTime;
```

```
      enemy.miniAsteroid3.rotation.y += .0015 * game.deltaTime;
      enemy.miniAsteroid3.rotation.x += .0015 * game.deltaTime;
    } else if (enemy.type == 2) { // Satellite animations
      enemy.sateliteDisc.rotation.x += 0.05;
      enemy.sateliteTopPart.rotation.x += 0.025;
    } else if (enemy.type == 3) { // Pterodactyl animations
      if (enemy.wingLeft.rotation.x > 1) enemy.decreaseWings = true
      else if (enemy.wingLeft.rotation.x < -0.75) enemy.decreaseWings = false;
      if (enemy.legLeft.rotation.y > 0.2) enemy.decreaseLegLeft = true;
      else if (enemy.legLeft.rotation.y < -0.2) enemy.decreaseLegLeft = false;
      if (enemy.legRight.rotation.y > 0.2) enemy.decreaseLegRight = true;
      else if (enemy.legRight.rotation.y < -0.2) enemy.decreaseLegRight = false;
      if (enemy.decreaseWings) {
        enemy.wingLeft.rotation.x -= 0.065;
      }
      else {
        enemy.wingLeft.rotation.x += 0.05;
      }
      if (!enemy.decreaseWings) {
        enemy.wingRight.rotation.x -= 0.05;
      } else enemy.wingRight.rotation.x += 0.065;
      var legDiff = Math.random() * 0.010;
      if (enemy.decreaseLegLeft) enemy.legLeft.rotation.y -= legDiff;
      else enemy.legLeft.rotation.y += legDiff;

      if (enemy.decreaseLegRight) enemy.legRight.rotation.y -= legDiff;
      else enemy.legRight.rotation.y += legDiff;
    }
    // ... code to handle projectiles hit, explosion and removal of the enemy
  }
}
```

All those animations assume that the references to the various part of the model are already set (it is done at spawning time) and go on animating each one of them through rotations on different axis, sometimes taking advantage of the already set rotation pivots.
Note that the animations for the pterodactyl are the most complex because they use the concept of states that the others simply don't have.

Regarding the animations done using TweenMax, those include animations that weren't supposed to be looped and for which I know at spawn time where and when they should end. This approach was extensively used for the particles

```
Particle.prototype.explode = function (isSmoke, speedFactor, pos, color, scale) {
  var _this = this;
  var _p = this.mesh.parent;
  this.mesh.material.color = new THREE.Color(color);
  this.mesh.material.needsUpdate = true;
  this.mesh.scale.set(scale, scale, scale);
  var targetX = pos.x + (-1 + Math.random() * 2) * 50;
  var targetY = pos.y + (-1 + Math.random() * 2) * 50;
  var ease = Power2.easeOut;
  var speed = .6 + Math.random() * .2;
  // choose targetX and targetY depending on speedFactor and random
```

```
  if (isSmoke) {
    if (speedFactor > 23) speedFactor = 23
    targetX = pos.x - 35 - speedFactor;
    targetY = pos.y + (-1 + Math.random() * 2) * 3;
    ease = Power0.easeOut;
    rotationSpeedFactor = 3
    TweenMax.to(this.mesh.material, speed * 0.3, { opacity: 1 });
  }
  else rotationSpeedFactor = 12

  TweenMax.to(this.mesh.rotation, speed, { x: Math.random() * rotationSpeedFactor,
                                           y: Math.random() * rotationSpeedFactor });
  TweenMax.to(this.mesh.scale, speed, { x: .1, y: .1, z: .1 });
  TweenMax.to(this.mesh.position, speed, {
    x: targetX, y: targetY, delay: Math.random() * .1, ease: ease, onComplete: function () {
      if (_p) _p.remove(_this.mesh);
      _this.mesh.scale.set(1, 1, 1);
      if (isSmoke) _this.holder.smokePool.unshift(_this);
      else _this.holder.particlesPool.unshift(_this);
    }
  });
}
```

From this snippet of code you can see that based on the type of particles (smoke/not smoke) I decide how the animations should perform and even add some extra ones like the fade in effect which is exclusive to the smoke.
TweenMax's "*to*()" API works by passing the object that I want to manipulate, the duration or speed of the manipulation and the ending result that I'm expecting, eventually I can even give the ease which will influence how I reach the desired values.
This type of animations are used for the opacity but also for the rotation of the particles, their translation over the axis and the manipulation of the mesh dimension.
By using the *onComplete* function I'm able to execute a block of code after the tween has completed, in this case when the translation animation completes I will remove the particle from the scene and from the pool of the active particle in the holder just to add it back to the available pool.
Something very similar is done for animating the projectiles, in this case two tweens were used, one to handle the opacity and the other to make the translation and on its completion remove the element from the scene.
While they are very similar to what is seen in the *ParticleHolder*, those tweens may be terminated prematurely if a projectile hits an enemy.

```
ProjectilesHolder.prototype.checkCollisions = function (pos) {
    var ret = false;
    for (var i = 0; i < this.activeProjectiles.length; i++) {
        var projectile = this.activeProjectiles[i];
        var distance = projectile.mesh.position.distanceTo(pos);
        if (distance < 18) {
            projectile.tween1.kill(); // opacity animation killed
            projectile.tween2.kill(); // translation animation killed
            this.projectilesPool.unshift(this.activeProjectiles.splice(i, 1)[0]);
```

```
            this.mesh.remove(projectile.mesh);
            i--;
            ret = true;
        }
    }
    return ret;
}
```

This logic that determines the animation interruption is expressed in the "*checkCollision*()" API of the *ProjectileHolder* where if a projectile and an enemy are too close a collision is detected, which will lead to the tweens of the colliding projectiles getting killed and the meshes getting removed from the scene and recycled for later use.
Similar collisions checks are used both in the *BonusHolder* and the *EnemiesHolder* but with different results.
In case of the bonuses a collision will trigger an energy increase while in case of the *EnemiesHolder* it will lead to game over or the destruction of the shield, in both the holders if a collision is detected some kind of particles will be spawn through the *ParticleHolder* to provide a visual feedback to the user.

```
EnemiesHolder.prototype.checkCollisions = function () {
  if (game.vehicle == undefined) return;
  for (var i = 0; i < this.enemiesInUse.length; i++) {
    // Is the enemy hit by a projectile?
    if (this.projectilesHolder.checkCollisions(enemy.mesh.position)) {
      this.particlesHolder.spawnParticles(false, 0, enemy.mesh.position,
                                          5, Colors.green, 3);
      this.enemiesPool.unshift(this.enemiesInUse.splice(i, 1)[0]);
      this.mesh.remove(enemy.mesh);
      i--;
      continue;
    }
    // Is the enemy colliding the player's vehicle?
    var d = game.vehicle.position.distanceTo(enemy.mesh.position);
    if (d < 15) {
      this.enemiesPool.unshift(this.enemiesInUse.splice(i, 1)[0]);
      i--;
      this.mesh.remove(enemy.mesh);
      if (this.game.hasShield) {
        this.particlesHolder.spawnParticles(false, 0, game.vehicle.position.clone(),
                                            5, Colors.green, 3);
        disableShieldImmunity(true);
      }
      else {
        this.particlesHolder.spawnParticles(false, 0, enemy.mesh.position.clone(),
                                            15, Colors.red, 3);
        game.gameOver = true;
      }
    } else if (enemy.angle > Math.PI) {
      this.enemiesPool.unshift(this.enemiesInUse.splice(i, 1)[0]);
      this.mesh.remove(enemy.mesh);
      i--;
    }
  }
```

```
}


BonusHolder.prototype.checkCollisions = function () {
    if (this.game.vehicle == undefined) return;
    for (var i = 0; i < this.bonusesInUse.length; i++) {
        var distance = game.vehicle.position.distanceTo(bonus.mesh.position);
        if (distance < 15) {
            this.bonusesPool.unshift(this.bonusesInUse.splice(i, 1)[0]);
            this.mesh.remove(bonus.mesh);
            var energyBonus = 5;
            if (bonus.type == 1) energyBonus = energyBonus * 2;
            else if (bonus.type == 0) energyBonus = 100;
            if (!this.game.gameOver) this.game.energy += energyBonus;
            if (this.game.energy > 100) this.game.energy = 100;
            this.particlesHolder.spawnParticles(false, 0, bonus.mesh.position.clone(), 5, bo
            i--;
        } else if (bonus.angle > Math.PI || bonus.mesh.position.x < -600) {
            this.bonusesPool.unshift(this.bonusesInUse.splice(i, 1)[0]);
            this.mesh.remove(bonus.mesh);
            i--;
        }
    }
}
```

3.8 User interactions

In additions to the various inputs that the player can give to control its spaceship, the project comes with some extra interactions that let the user influence how the game is played.
Those interactions, as seen in the previous chapters, deals with changing the spaceship, managing the scoreboard, pausing/resuming/resetting the game, customizing the difficulty and monitoring of game performance.
All those interactions (with the exception of the scoreboard) happens through the dat.gui overlay.

```
function initDatGUI() {
    ... // setting up default values for dat.gui
    stats = new Stats();
    gui = new dat.GUI({ width: 280 });
    // Populate main section
    [].forEach.call(stats.domElement.children, (child) => (child.style.display = ''));
    gui.add(options, 'vehicle', { '- Spaceship1': 0, '- Spaceship2': 1,
                                  '- Spaceship3': 2, '- TARDIS': 3 });
    gui.add(options, 'paused');
    gui.add(options, "audio").onChange(function () {
        if (options.audio) {
            audioListener = new THREE.AudioListener();
            camera.add(audioListener);
            sound = new THREE.Audio(audioListener);
            startAudio();
        }
        else {
            if (typeof (sound) != 'undefined') sound.stop();
```

```
            audioStarted = false;
        }
    });
    gui.add(options, 'reset');
    // Create and populate difficulty section
    var difficulty = gui.addFolder("Difficulty customization");
    difficulty.add(options, 'speedIncrOverTime');
    difficulty.add(options, 'speedIncrPerLevel');
    difficulty.add(options, 'energyDecayPerFrame')
    difficulty.add(options, 'energyDecayIncrPerLevel');
    difficulty.add(options, 'noFireCost').onChange(function () {
        removeShield();
        resetGame();
    });
    difficulty.add(options, 'noShieldCost').onChange(function () {
        removeShield();
        resetGame();
    });
    // Create and populate debug section
    var debug = gui.addFolder("Debug");
    debug.add(options, 'fixedDeltaFramesTime');
    // Create and populate performance section
    var perfFolder = gui.addFolder("Performance");
    var perfLi = document.createElement("li");
    stats.domElement.height = '48px';
    stats.domElement.style.position = "static";
    perfLi.appendChild(stats.dom);
    perfLi.classList.add("gui-stats");
    perfFolder.__ul.appendChild(perfLi);
    perfFolder.domElement.getElementsByClassName("title")[0].addEventListener("click", funct
        if (perfLi.style.height != "auto") perfLi.style.height = "auto";
        else perfLi.style.height = "";
    });
}
```

The dat.gui attaches itself to an "options" data structure whose value will be used inside the game loop.
When a change is done thorugh the overlay, the data structure will change and an *onChange* event will be triggered, depending on what got changed the listener will be executed and the game run may get reset.

I will not explore the HTML elements of the UI because the code only interacts with them by linking some variable to the DOM elements and depending on the game state change their content or visibility.

More interesting is the scoreboard implementation, which relies on the browser *LocalStorage*.
This means that every time the user saves a score it will be maintained, in a JSON format, even if the game page is closed.
By default the scoreboard implementation can save only the last 8 scores but that limit can be easily increased as long as the *LocalStorage* doesn't get full.
Then when there is the need to draw the scoreboard (i.e: when the game

over condition is met) I can just loop through the JSON array and by accessing the various attributes of the JSON object I will populate the DOM. Here follows a small snippet of the scoreboard that shows how a score is added to the *LocalStorage* in a decreasing order on the distance attribute.

```
function addScore(name, distance) {
    if (name === "") return;
    initLocalStorage();
    var u = JSON.parse(localStorage.scoreboard);
    var o = {
        name: name,
        distance: distance,
    };
    var ins_pos = undefined;
    for (var i = 0; i < u.length; i++) {
        if (u[i].name == undefined) {
            u.splice(i, 1);
            i--;
            continue;
        }
        if (u[i].name == name && u[i].distance == distance) return;
        if (u[i].distance <= distance) {
            ins_pos = i;
            break;
        }
    }
    if (ins_pos != undefined) u.insert(ins_pos, o);
    if (u.length == 0) u.push(o);
    else if (ins_pos == undefined) u.insert(u.length,o);
    if (u.length > 8) u.pop();
    localStorage.scoreboard = JSON.stringify(u);
}
```

3.9 Game loop

Now that I presented all the major components of the game we can see what the game loop actually does in order to understand how does it takes advantage of the different components of the game.
The operations that takes inside the loop are the following:

- Computation of the delta time between frames by using Javascript's Date APIs.
- Apply difficulty changes done by the user through the overlay UI, the changes are applied by modifying the game data structure.
- Compute distance traveled, base rotation speed based on the difficulty level.

- Spawn bonus coins and enemies if last spawned happened too long ago.
- Increase speed based on distance and level.
- Update vehicle position and shield based on input
- Rotate terrain an increase ambient light based on
- Apply energy decay
- Spawn smoke if needed at the back of the spaceship
- Show scoreboard, update game speed and animate vehicle in case of game over.
- Animate sky, terrain, bonuses, enemies and check collisions
- Update HTML UI to show shield cooldown, difficulty, and energy level.
- Render the scene.

Based on the game status the number of tasks of the game loop may change, for example in case the game is paused all those tasks except rendering are ignored and in case the game over condition is met there will be no difficulty/distance increases or enemies/bonuses spawn.

4 Sources

For the development of this game I learned about Three.js through the Three.js's discourse forum and documentation (for geometries, material properties, lights, and loaders) but I also learned a lot from the Karim Maaloul tutorials and snippets on CodePen which provided insight on Three.js APIs, on recycling of resources, and how to setup a basic scene.
During the development different Blender forums helped me at understanding how to create, transform, split/merge and setting up relations for the 3d objects in Blender.
I also used Google Poly for the design of the models of the enemies and the player spaceships, considering that Google Poly doesn't provide hierarchical models I ended up redoing the same model in Blender and adjusting it to be compliant with the required specifications.
Follows the references to the resources that I used during the development.

References

[1] Karim Maaloul's snippets
`https://codepen.io/Yakudoo`

[2] Exploring Animation And Interaction Techniques With WebGL, Karim Maaloul
`https://www.smashingmagazine.com/2017/09/animation-interaction-techniques-WebGL/`

[3] Animating a basic 3D scene with Three.js, Karim Maaloul
`https://tympanus.net/codrops/2016/04/26/the-aviator-animating-basic-3d-scene-threejs/`

[4] Splitting objects in Blender, Blender forum
`https://blenderartists.org/t/how-can-i-split-one-object-into-two/357974/4`

[5] Merge objects in Blender, Blender forum
`https://all3dp.com/2/blender-how-to-merge-objects/`

[6] Change pivots or local origin of an object, Blender StackExchange
`https://blender.stackexchange.com/questions/1291/change-pivot-or-local-origin-of-an-object`

[7] Shading, Wikipedia
https://en.wikipedia.org/wiki/Shading

[8] Three.JS documentation
https://threejs.org/docs/#api/en/core/Geometry
https://threejs.org/docs/#api/en/objects/Mesh
https://threejs.org/docs/#api/en/constants/Materials
https://threejs.org/docs/#api/en/lights/Light
https://threejs.org/docs/#api/en/loaders/Loader

[9] Spaceship1,2,3 design, Liz Reddington
https://www.artstation.com/artwork/Z12EG

[10] Satellite design, Google Poly team
https://poly.google.com/view/2i9NPgZ-ALP

[11] TARDIS design and texture, Neil Nathanson
https://poly.google.com/view/8kfkpUr245T

[12] Low-poly Asteroid design, Jarlan Perez
https://poly.google.com/view/8sNKYRTUFAe

[13] Low-poly Pterodactyl design, Hoai Nguyen
https://poly.google.com/view/1tuiNTr4-MX

[14] Low-poly airplane design, Karim Maaloul
https://tympanus.net/codrops/2016/04/26/
the-aviator-animating-basic-3d-scene-threejs/