Desert Dash

Cacciarini Gianluca 1871982

September 2023

Contents

1 Introduction

Desert Dash takes inspiration from a similar game called Temple Run and is an exciting browser-based 3D infinite runner game. Developed using powerful graphic libraries, Three.js and Tween.js. They provided the essential tools and features to transform my concept into a fully realized game.
Set in a vast desert landscape, Desert Dash places players behind the wheel of a car on a endless road. The primary objective is to cover the longest distance possible while avoiding obstacles.
During the development of this project, I encountered several challenges that helped me in insights into the complexities that can arise when creating games of this nature. These experiences made me more aware of the factors that need to be considered when working on such projects. These lessons learned during the development process have contributed significantly to my understanding of game development and its unique demands.

1.1 Commands

The car is controllable through the use of keys: **Left Arrow** and **Right Arrow**.

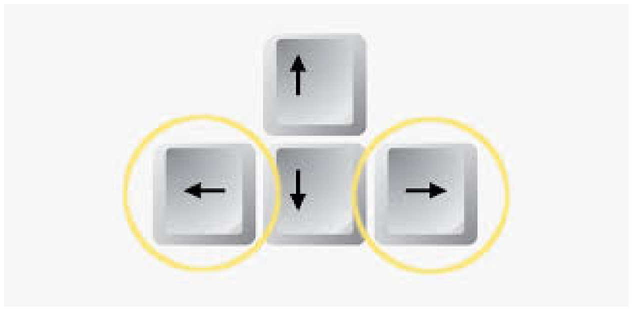


Figure 1: Commands

2 Models

In the context of the game, I used a variety of 3D models to create the environment, obstacles and the main vehicle. I obtained these models from Sketchfab, a platform for sharing 3D content. To smoothly integrate these models into the game, I employed GLTFLoader, a library that simplifies the process of loading models directly into the gaming environment.
In this section, I will showcase the models used in the game, all of which have been properly scaled to fit within the scene.

2.1 Main car

Main car model

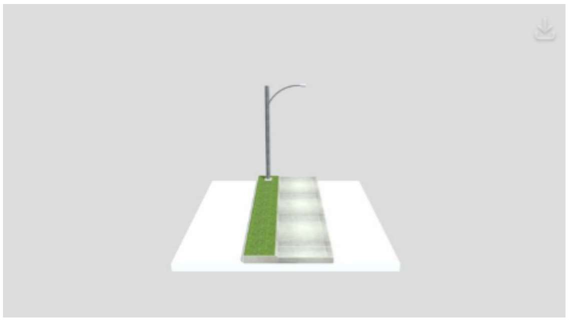
Wheel model

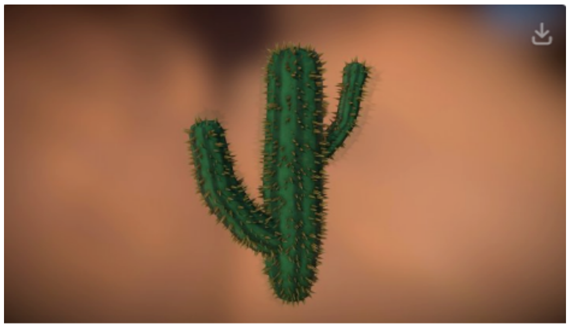
While examining the imported car model, I noticed an issue in its hierarchical structure where the wheels rotated around the center point between all four wheels. Consequently, when I tried to implement wheel rotation animation, they rotated around the center point axis, which was not the desired result. To address this problem, I introduced an additional wheel model and duplicated it four times to replace the original wheels in the main model. These new wheels were correctly positioned to align with the car's axis, allowing each of them to rotate independently along the X-axis. This adjustment finally allowed for the desired wheel animations.

2.2 Ambient

In creating the game environment, I used two types of models:

- **Sidewalk:** I employed a sidewalk model and replicated it several times to achieve the effect of a continuous roadside.
- **Cacti:** I used cactus models, which I cloned, instantiated, and randomly placed throughout the desert.


Sidewalk model


Cactus model

2.3 Obstacles

There are three types of obstacles along the road:

Obstacle 1

Obstacle 2

Obstacle 3

3 Textures & Lights

3.1 Textures

Within my project, I employed various textures:

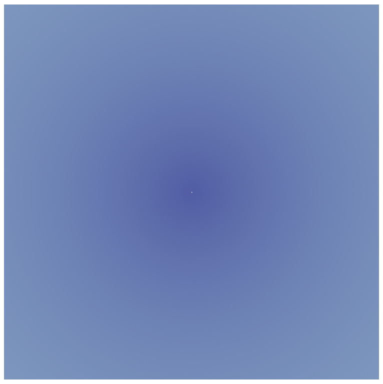
1. **Road Texture:** I used a specific texture for the road, which was then repeated using the ‘texture.repeat.set(x, y)‘ function, where ’x’ represents the number of repetitions horizontally and ’y’ represents the number of repetitions vertically. This texture was applied to the materials of the different planes that compose the road.
2. **Desert Texture:** Another texture was used for the sandy terrain alongside the road to create the desert appearance.
3. **Skybox Texture:** I used six different images as textures to create a skybox, which I will describe in the following paragraph.

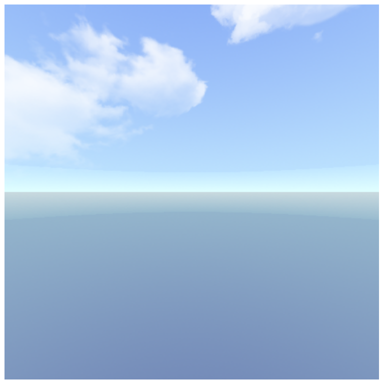
These textures were crucial for the visual rendering of the project.

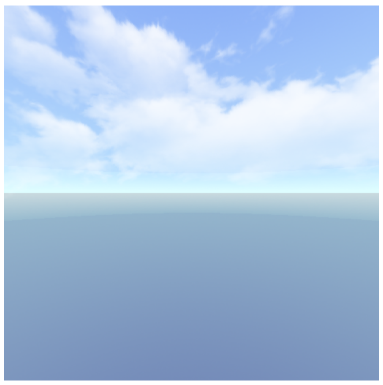
3.1.1 Skybox

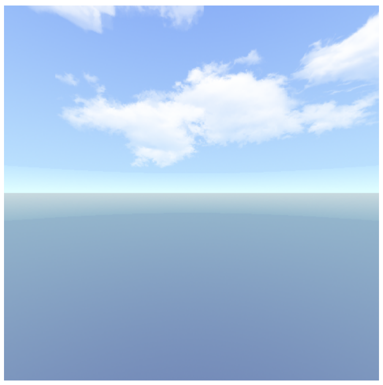
I used a *Skybox* for the environment in my project. To achieve this, I used the Three.js **‘TextureLoader‘** library to load six different images. These images represent the faces of a virtual box and were sourced from the following files:

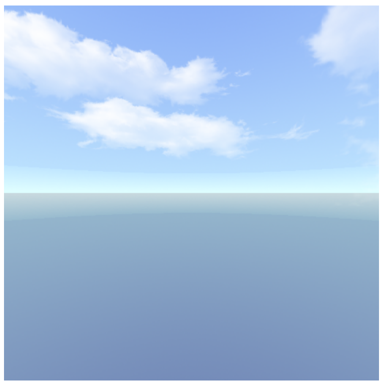
Top face


Bottom face


Front face


Left face


Right face


Back face

Then I used these images to create materials with **‘MeshLambertMaterial‘** and applied them to the backside of a box geometry that is placed in such I way that I'm inside it with the camera. This brought the Skybox to life, contributing to the illusion of a three-dimensional background. Finally, I added this Skybox to my scene, allowing players to immerse themselves in the game.

3.2 Lights

Regarding the *Lights* I used:

- **Ambient light:** which is shining from every direction, I used it to have a base color for our objects.
- **Directional light:** is defined by the position of these light rays, from all the parallel rays we define one, this specific light ray will shine from the position we define (*dirLight.position.set(0, 100, -10)*) to the 0, 0, 0 coordinate. The rest will be in parallel. By setting the position of the light we determine which side of our objects will be brightest and which ones stay in the dark.

4 Animations

I've added smooth animations made possible through the powerful Tween.js library, allowing for fluid movement and transitions that make the game more enjoyable.

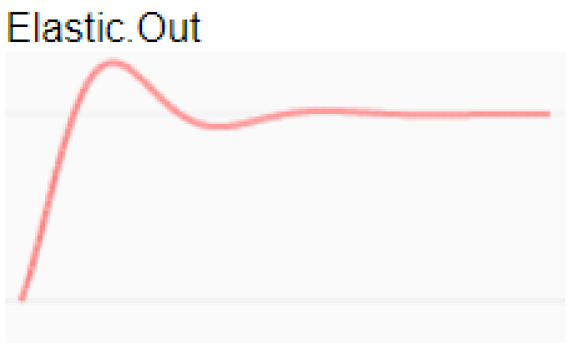
4.1 Car movement

I have implemented a key animation that allows the dynamic movement of the car both right and left in the game context. This animation was done through the following code:

```
_carRotation(dir) {
    ...
    const tween1 = new TWEEN.Tween({
      car: initialRotation,
      posX: initialPosition
    })
      .to({
        car: initialRotation + rotY,
        posX: initialPosition + addX
      }, 250)
      .onUpdate((element) => {
        this.mainCar.rotation.y = element.car;
        this.mainCar.position.x = element.posX;
      })
      .easing(TWEEN.Easing.Linear.None)
      .start()
      .onComplete(() => {
        const tween2 = new TWEEN.Tween({
          car: this.mainCar.rotation.y,
        })
          .to({
            car: initialRotation,
          }, 250)
          .onUpdate((element) => {
            this.mainCar.rotation.y = element.car;
          })
          .easing(TWEEN.Easing.Elastic.Out)
          .start()
          .onComplete(() => {
            this.animationInProgress = false;
          });
      });
  }
```

this code manages the movement of the car to the right and left, changing its rotation and position according to the direction chosen provided as function input.

To achieve the visual effect characteristic of a car that changes direction quickly, like a drift effect, I used a function of easing provided by the Tween.js library. **‘Elastic.Out‘** indicates a type of easing that creates a spring-back effect at the end of the animation.



Elastic.Out function

4.2 Wheels movement

Regarding the movement of the wheels, as I already said in the section devoted to the model of the main car, I have added four wheels to the main car in such a way that each wheel could independently rotate along the x-axis, the code is the following:

```
_wheelsRotation() {
    //*** Main car wheels movement ***
    this.LBWheel.rotation.x += 0.05;
    this.RBWheel.rotation.x += 0.05;
    this.LFWheel.rotation.x += 0.05;
    this.RFWheel.rotation.x += 0.05;

    //*** Obstacles wheels movement ***
    this.obstaclesGroup.children.forEach(car =>{
      if(car.getObjectByName('FLWheel')){
        car.getObjectByName('FLWheel').rotation.x -= 0.05;
      }
      if(car.getObjectByName('RLWheel')){
        car.getObjectByName('RLWheel').rotation.x -= 0.05;
      }
```

```
        if(car.getObjectByName('FRWheel')){
          car.getObjectByName('FRWheel').rotation.x -= 0.05;
        }
        if(car.getObjectByName('RRWheel')){
          car.getObjectByName('RRWheel').rotation.x -= 0.05;
        }
      })
    }
```

4.3 Collision animation

Finally I also implemented an animation that is activated whenever the main car collides with an obstacle, the effect of such animation is similar to that of some arcade games of the past in which the character appears and disappears quickly after after suffering damage.

```
...
this.materials.forEach((material) => {
    material.transparent = true;

    const opacityStart = material.opacity;
    const opacityEnd = 0.1;

    const state = { opacity: opacityStart };

    new TWEEN.Tween(state)
      .to({ opacity: opacityEnd }, 200)
      .easing(TWEEN.Easing.Quadratic.InOut)
      .onUpdate(() => {
        material.opacity = state.opacity;
      })
      .yoyo(true)
      .repeat(7)
      .start()
      .onComplete(() => {
        this.vulnerable = true;
      });
});
```

I achieved the desired effect by playing with the opacity of the materials that make up the main car. I reach the final opacity **opacityEnd** and return to the initial opacity through the function **'.yoyo(true)'** and I repeat such animation seven times **'.repeat(7)'**.

5 Game logic

5.1 Endless road generation

One of the main challenges in creating an infinte runner game was to establish a game environment that gives the illusion of being endless. The solution I employed to achieve this sensation involved keeping the main car static while allowing the surrounding environment to move. In essence, it appears as though the car is in motion, but in reality, it's the environment around it that's shifting.
To achieve this effect, I generated four initial **PlaneGeometry**, that make up the road, positioned one after the other along the negative Z-axis. By using the **'_updateRoad'** function, I moved each of these four planes along the positive Z-axis at a speed matching the desired car movement in the game. Whenever one of these planes crosses a predetermined point along the Z-axis, which corresponds exactly to its length, I atuomatically reposition it as the new last plane among the four. In this way, these four planes are continually shifted from the beginning to the end, creating the illusion of an endless road within the game.

5.2 Scene generation

Similar to how the road is infinitely generated, the desert terrain on its sides, as well as the sidewalk, follow the same logic. The desert terrain also consists of four planes, and whenever one of these planes crosses a predetermined point along the Z-axis, which corresponds exactly to its length, they are automatically repositioned at the end of the four.
As for the sidewalk along the road's edge, I created multiple instances of the imported model and placed them one after another to achieve the continuous sidewalk effect. To have the sidewalk on the other side of the road, I had to apply scaling with a value of -1 on the X-axis. The function that updates the position of the sidewalk is as follows:

```
_updateSidewalker() {
  this.sideWalkerGroup.position.z += this.speedZ;
  this.sideWalkerGroup.children.forEach(object => {
    const childZPos = object.position.z +
        this.obstaclesGroup.position.z;
    if (object instanceof THREE.Object3D) {
      if(childZPos > 5 + this.size.z){
        object.position.z = -this.sideWalkerGroup.position.z +
            this.lastSideWalkerZ;
      }
    }
  });
}
```

SidewalkerGroup contains all the instances of the model sidewalk and is updated by a value equal to the speed along the Z-axis (*this.speedZ*). For each of the instances contained in the Group is calculated the *childZPos* which is the world position of the object, more precisely the position it has inside its parent relative space, and if such value is greater than the size of $5 + this.size.z$, where $this.size.z$ is the z-length of the Sidewalk model, we reset the object as the last sidewalk in the Group.

5.3 Obstacle spawn

As for how obstacle spawning works, I defined an array containing possible positions along the X-axis where the various obstacles can spawn. This array determines the horizontal positions for the obstacles.
Regarding the Z-axis, where obstacles are spawned, I implemented an auxiliary function that calculates an offset value between two input parameters. This offset value must be a multiple of another specified input value. In my case, obstacles can only be generated at position that are multiples of the Z-dimension of the largest obstacle, denoted as **'this.sizeObs2.z'**. This approach prevents obstacles from overlapping or partially overlapping.
Once the offset is calculated, it is used to determine the actual position of the obstacle at a distance calculated as **'- (200 + offset)'**. This ensures that obstacles are always spawned at a distance that remains otside the camera's perspective far plane.

```
_setupObstaclePlane(obj, refZPos = 0) {
  obj.position.x = this._pickRandom(this.obstacleCarPosX);
  const offset = this._getMultipleInRange(this.sizeObs2.z, 0, 200);
  obj.position.z = refZPos - 200 - offset;
}
```

Obstacles, like the rest of the scene, also move along the Z-axis. When obstacles move out of the camera's field of view, they are repositioned outside the camera's far plane. This avoids the need to deallocate and recreate the same object.
The function responsible for this is as follows:

```
  // update obstacles
_updateObstacles() {
  // obstacles movement
  this.obstaclesGroup.position.z += this.speedZ;

  this.obstaclesGroup.children.forEach(obstacle =>{
    const obstacleZPos = obstacle.position.z +
        this.obstaclesGroup.position.z;
    if (obstacle instanceof THREE.Object3D) {
      if(obstacleZPos > 2){
```

```
                this._setupObstaclePlane(obstacle,
                    -this.obstaclesGroup.position.z);
              }
            }
          });
        }
```

5.4 Collision

Collision management has been implemented using the **'THREE.Box3.setFromObject'** method provided by the Three.js library. This method allows us to obtain a **'Box3'** object that defines a bounding box completely encompassing the specified input object. In our case, we create two bounding boxes for each child of the **'obstacleGroup'**: one for the main car and one for the obstacle under analysis belonging to the group. Once these bounding boxes are created, we use the **'intersectsBox'** function to determine whether the two boxes collide or not.

```
    _checkCollisions() {
        this.obstaclesGroup.children.forEach(obstacle =>{

        let box1 = new THREE.Box3().setFromObject(obstacle);

        let box2 = new THREE.Box3().setFromObject(this.mainCar);

        const isIntersecting = box1.intersectsBox(box2);
        ...
```