Project

Web Browser used to test the project: Google Chrome.

Libraries and imported models

For the development of the project it has been used the:

- Three.js library imported using the cdn

Additionally we imported other Three.js functionalities using the cdn:

- Orbit Controls to implement the movement of the camera along the scene
- Water shader to implement the water inside the lake in the scene
- GLTFLoader() and FBXLoader() to import the 3D models
- GUI to implement the gui used to interact with the elements in the scene

The imported libraries and functionalities can be found at the beginning of the file ‘project.js’.

```
import * as THREE from "https://threejs.org/build/three.module.js";
import { OrbitControls } from "https://threejs.org/examples/jsm/controls/OrbitControls.js";
import { GLTFLoader } from "https://threejs.org/examples/jsm/loaders/GLTFLoader.js";

import { Water } from  "https://threejs.org/examples/jsm/objects/Water.js";
import { FBXLoader } from "https://threejs.org/examples/jsm/loaders/FBXLoader.js";
import { GUI } from 'https://cdn.jsdelivr.net/npm/three@0.121.1/examples/jsm/libs/dat.gui.module.js';
```

The 3D models used in the project are:

- the lighthouse
- the seagull
- the chicken
- the human

The first three models were downloaded from the ‘sketchfab.com’ website and they are gltf models; credits of these models are inside the file ‘CreditsFor3DModels.txt’. The human model was downloaded from the ‘mixamo.com’ website and it is a fbx model.

Initialization of the scene

At the beginning of the init() function, we initialize:

- the scene
- the camera which uses the perspective projection. Additionally we define the orbitControls to allow the camera to orbit around the scene.
- the renderer. For the renderer we set up the clear Color, the size of the canvas and the shadow maps in the scene by enabling them, and then we specify its property set to PCFSoftShadowMap to produce better soft shadows.
- the GUI where we define the folders which will contain different options to interact with the elements in the scene.
- the lights: we define an ambient light, and a directional light whose property to cast shadow is set to true; in this way the light will cast dynamic shadows. The directional light will move repeatedly in the scene along a circumference and this is done in the animate() function.

Elements in the scene

Ground

The ground of the entire scene is realized as a Cube. It is created by defining a BoxGeometry; its appearance is defined in the MeshStandardMaterial() where we apply the textures to give to the ground the appearance of a lawn. More specifically for the ground we specify the color map, the normal map, the displacementMap with its bias to give the object a sense of depth and the roughnessMap with its roughness to alter the roughness of the object. Finally, since the ground is the place where all the objects of the scene are located, we specify that the ground will receive the shadow casted by other objects, but it does not cast shadows.

```
const colorTextGrass = textureLoader.load('./Textures/grassField/Stylized_Grass_003_basecolor.jpg');
const bumpMapGrass = textureLoader.load('./Textures/grassField/Stylized_Grass_003_normal.jpg');
const displacementMapGrass = textureLoader.load('./Textures/grassField/Stylized_Grass_003_height.png');
const roughMapGrass = textureLoader.load('./Textures/grassField/Stylized_Grass_003_roughness.jpg');

const geometry = new THREE.BoxGeometry( 300, 20, 300);
const material = new THREE.MeshStandardMaterial( {
    map:colorTextGrass,
    normalMap:bumpMapGrass,
    displacementMap:  displacementMapGrass,
    displacementBias: -0.9,
    roughnessMap: roughMapGrass,
    roughness: 0.5,
} );
const cube = new THREE.Mesh( geometry, material );

cube.castShadow = false;
cube.receiveShadow = true;

cube.position.y = -10;
scene.add( cube );
```

Clouds

Clouds are created through a specific function called cloudGen(). In the function we create a custom geometry for the cloud by using the Group() function; the custom geometry, which is the CloudGroup, is composed of three spheres; the appearance of each sphere is defined in the MeshLambertMaterial() where we set the color of the objects. MeshLambertMaterial() has been used to simulate an object with no shiny surface. Since clouds are the higher elements in the scene, they cast shadows on the ground and on the other elements on the ground, but it does not receive any shadow. Each sphere is added to the cloudGroup and the cloudGroup is then returned once the function is invoked.

```
function cloudGen(){

    const cloudGroup = new THREE.Group();
    const cloudColor = new THREE.MeshLambertMaterial({
        color:'white',
    });

    const sphere1 =  new THREE.SphereGeometry(3, 14, 16);
    const sphereCloud1 = new THREE.Mesh(sphere1, cloudColor);
    sphereCloud1.castShadow = true;
    sphereCloud1.position.set(-2,30,0);
    cloudGroup.add(sphereCloud1);

    const sphere2 = new THREE.SphereGeometry(3, 14, 16);
    const sphereCloud2 =  new THREE.Mesh(sphere2, cloudColor);
    sphereCloud2.castShadow = true;
    sphereCloud2.position.set(2,30,0);
    cloudGroup.add(sphereCloud2);

    const sphere3 = new THREE.SphereGeometry(3.5, 14, 16);
    const sphereCloud3 =  new THREE.Mesh(sphere3, cloudColor);
    sphereCloud3.castShadow = true;
    sphereCloud3.position.set(0,30,0);
    cloudGroup.add(sphereCloud3);

    return cloudGroup;

}
```

In the init(), the function cloudGen is called three times to generate three clouds which are then added to the scene.

Trees: pines and round trees

Pines and round trees are generated respectively by the functions:

- treePine(): we define a new group for the custom geometry of the pines. The geometry is composed by a trunk modeled as a cylinder, and three cones which represent the upper part of the pine. Starting from the higher cone, each cone has increasing radius. To simulate the surface of the wood, we specified the appearance of the cylinder through the MeshStandardMaterial() defining the color map and the normal map, applying the wood texture. As regards the appearance of the upper part of the tree, we specified only the color of the cones. Then, we specify that each element will cast shadows, since the pine will cast shadow on the ground. Finally, each geometry is added to the pineGroup which is returned by the function. The three pines are generated by the three invocations of the function treePine(), their position is set in the init() and they are added to the scene.

- treeRound(): we define a new group called treeRoundGroup() for the round trees. The group will be composed of icosahedrons, scaled to obtain three wide crowns of the tree, the trunk and two small branches. Since we wanted to create low poly round trees, we specified the appearance of each component of the tree through the MeshLambertMaterial, since it can simulate surfaces like wood; more specifically, we simply define the color of the crown and the color of the trunk of the tree. Finally, for each element of the group we specify that it will cast shadows, since the entire tree will cast the shadow on the ground. The round tree is returned by the function treeRound() which is called in the init() three times to define a small group of trees located to the left side of the scene.

Street and sand road

In the scene we have two possible street:

- streetGen(): it generates the street, where the car is located. The function defines a boxGeometry, a rectangular cuboid, which will be the street. For the street we define its appearance through the meshStandardMaterial(), where we apply on the object color map, normal map, displacement map with its bias to give the object a sense of depth and the roughnessMap with its roughness to alter the roughness of the object. Since the street will cast shadows and will receive the shadows projected by the other objects located above it, we set to true the property castShadow and receiveShadow. The function streetGen() is called in the init() function and it will return the created street, which is then added to the scene.
- sandRoadGen(): it generates the sand road where the human figure is located. The function is similar to the streetGen(), however the applied textures change, since this time we apply specific textures to simulate the appearance of a sand road.

Fences

We have two possible fences, the metal fences located near the street and the wooden fences located near the round trees:

- fenceGen(): the function generates the metal fence. The metal fence is composed of an horizontal bar and four vertical bars: two of them located at the beginning and the end of the fence, two of them located in the central part of the fence. We define a group for the metal fence, composed by the previous described element; each element composing the fence is a cylinder. Since the fence surface is supposed to be metallic, in the meshStandardMaterial() we specify the metalnessMap and the metalness fields, to alter the metalness of the material. Finally, for each element of the fence we specify that it will receive and cast shadows.
- woodFenceGen(): the function generates the single fences composing the entire wooden fence, inside which is located the chicken. The wooden fence is composed of two vertical bars and two horizontal bars. Following the same logic of the metal fence, we define a group for the custom geometry of the wooden fence, and we apply on each element the textures in the MeshStandardMaterial() to simulate a wooden surface. Then in the init() function we call the function to create the six fence parts that compose the entire wooden fence and we set their position in order to create a cage.

Circle of stones

To generate the circle of stones near the three pines, we define the function circleStone(). In the function, we define each stone as a dodecahedron, and since we want to simulate a low poly stone surface, we use the MeshLambertMaterial(), where we specify only the color of the stones. The circle of stones is composed by fifteen stones disposed along a circle; so, we firstly define an array of stones where we add each stone which is created in the for cycle; for each stone:

- we set its position along the circumference
- we translate the stone in such a way they are near the three pines
- we specify that each stone will cast shadow on the ground
- we scale its dimension according to a tossed value. More specifically, to randomize the dimension of each stone, we firstly define a variable toss which can be equal to zero or one. Then, according to the tossed value, we scale the stone, modifying its height. Finally we add the stones to the scene. Each time we refresh the page, the heights of the stones will change.

Mountains

The function mountainGen() generates the mountains in the scene. As for the circle of stone, we define each stone as a dodecahedron, and to give to them the low poly appearance we just specify its color with the MeshLambertMaterial(). The mountains are composed of thirty mountains; so we define an array which will contain each mountain. Then in the for cycle:

- we create the mountain
- we set its position and we translate it. This time the x-position of the mountain will be a random value in range [20, 70], while the z-position of the mountain will be a random value in range [0,120]. In this way, the group of mountains will appear in the right part of the scene, and it will be positioned in such a way that it will not cover the surrounding elements.
- we scale the mountain's dimensions according to a variable toss. We toss a value between 0 and 1, and we randomly generate, from a certain range of values, the height and the width of each mountain. Finally, according to the tossed value, we scale each mountain, modifying its width and height. In this way we create a group of mountains with different aspects, according to their dimensions. Each time we refresh the page, the configuration of the mountains will change.

Lamps

The function createLamp() creates each single lamp; it is called in the init() function to create three lamps that are added to the scene and are located near the street. Inside the function we create a group, called groupLamp, for the custom geometry of the lamp; the group is composed of a vertical pole, an horizontal pole, an additional smaller vertical pole which is the one attached to the spherical lamp, and the spherical lamp. Each pole is a cylinder and the smaller pole is an icosahedron. To simulate the wooden surface of the three poles, in the MeshStandardMaterial, we apply the textures on each of them, specifying the color map, normal map, displacement map with its bias and the roughness map with its roughness.
As regards the lamp, we firstly define its geometry which is a sphere; then we define the pointlight with its parameters: color, intensity and distance; we set its property castShadow to

true to calculate the shadow for the pointLight. Then, we define the lamp object appearance, so in the MeshStandardMaterial() we set its emissive property, the emissive intensity set to 1, and the color of the sphere. Finally we add the created spherical lamp to the light, then we add them to the groupLamp.

Car

The car is created in the function createCar(). We define a group to create the custom geometry of the car. The group is composed of:

- car cabin: it is a cube, whose appearance is modeled in the MeshLambertMaterial. More specifically, we specify for the top and bottom face of the cube the white color; on the front and back faces of the cabin we apply a custom texture to simulate the windscreen of the car, while on the lateral side of the cabin we apply a custom texture to model the car windows. Since the car cabin casts shadows we set its property castShadow to true.
- car body: it is a red cube, whose appearance is modeled through the MeshLambertMaterial where we set its color. Since the car body casts a shadow on the ground, we set its property castShadow to true.
- car pipe: it is located behind the car and it is a gray cylinder. Like the car body, we set its color and its castShadow property.
- tires: since the car has four tires, we firstly define the tires geometry which are cylinders and an array ‘tires’ which will contain the tires created in the for cycle. In the for cycle, we create a tire, and we apply the textures on it; more specifically, we apply a color texture on the lateral sides of the tire to simulate the wheel rim, while in the central part of the tire we just set the color to black to model the tire. After adding all the created tires to the array, we access the single elements/tires to set their position in such a way each tire appears as attached to the car. Finally we return the car group once the function is invoked in the init().

Car textures

We create textures specific for the front and the side of the car, in order to do that we use the HTML canvas to draw shapes that will simulate the car windows. So in the function frontTextureCar() we firstly create the canvas element that will be turned into a texture. We define the width and the height of the canvas, so its resolution, and the 2D drawing context to execute the drawing commands. In the next lines we fill the canvas with a white rectangle so we set the fill style to be white. We fill a rectangle by setting its top-left position and its size in the fillRect() function. Then we fill a second rectangle with the gray color, it starts at the coordinate (8,8) and it will not fill the canvas, so it paints the car windscreen. Finally we turn the canvas elements into a Three.js texture and we return the texture.
In a similar way we define the function sideTextureCar() to define the texture for the side of the car. The function is the same as frontTextureCar() however it adds a third rectangle to be filled with gray color and modifies the coordinates of the two gray rectangles in order to have two car windows. The textures returned by the two functions are then mapped on the cabin of the car.

Car animation

For the car animation we define the set of car parameters in the paramCar variable which contains the variables that will be modified through the GUI. In the set of car parameters, we define:

- a switch to start and stop the animation
- a function to reset the animation of the car; the function positions the car at its initial position, setting the switch to false, so the car will not move; additionally the function sets the speed of the car and the speed of the wheels to their initial value.
- The value of the car and wheel speed, which are global values stored in the respective variables (speed, speedWheels).

Through the GUI, we can modify these variables, interacting with the car. To animate the car we defined the function animateCar() that is called in the animate() function if the switch for the animation is set to true. The function animates the car wheels and moves the car along the street; once the car reaches the end of the scene, it respawns at its initial position. In the animateCar() function, we access the wheels of the car which are the children of the car group.Then we rotate the wheels according to their speed that can be modified through the GUI. To model the fact that the car will respawn at its initial position, we check if the car position is greater than the end of the scene/street; if so, we restore the car's initial position, otherwise we decrease the car position according to its speed moving it over the street.

Lighthouse

The lighthouse in the scene is a 3D model which is loaded through the GLTFLoader(). Once the model is loaded we set its position and we scale it in such a way that it is located in the left side of the scene and its dimensions fit the ones of the other elements in the scene. We decided to model the real behavior of the lighthouse; to do so, we defined a pointlight that will simulate the light emitted by the lighthouse. Since we need to model the rotating light in the lighthouse, we define a black cube that will rotate around the pointlight position. To let the cube rotate around the light position, we define a pivot as a group; then we add the pivot to the point light so it will be the light parent and it will be positioned on the light position. Then we define the black cube and it will be added to the pivot, so the cube will be a child of the pivot. Finally we set the position of the cube with respect to the pivot point and in such a way it is as near as possible to the light.
At this point we set the rotation of the cube along the y-axis and around the pivot in the animate() function by rotating the pivot point; the rotation of the pivot point along the y-axis will make the black cube rotate around the pivot itself.

Seagull

The seagull is a 3D model loaded through the GLTFLoader(). Once it is loaded, we set its position and dimensions in order to let it be positioned in the air, over the scene, and between the clouds; then we store in specific variables the names of the wings of the seagull that will be animated.
The function to animate the seagull is called birdAnimation() and it is called in the animate() function. The birdAnimation() function animates the seagull wings to model the bird’s flight. To do so we first verify that each variable associated with the seagull is initialized. We want the left wing and the right wing to move, respectively, in a specific range [-40 degrees, 40

degrees] and [-220 degrees, -140 degrees]. To do so we check, for the left wing, if the wing rotation around the z-axis is less than 40 degrees and the invert variable is equal to zero; if so we increase the wing angle by the speed of the wing, which is 3 degrees. If the wing angle is greater than 40 degrees, we invert the rotation and we decrease the wing angle by the speed of the wing. If the wing angle is less than -40 degrees we invert the rotation of the wing. The same procedure is done for the right wing.
Additionally there is the possibility to move the seagull with the directional arrows on the keyboard. So, in the birdAnimation() function, there will be a section where we limit the seagull's movements, to avoid the seagull from leaving the scene.

Chicken

The chicken in the scene is a 3D model loaded through the GLTFLoader() function. As for the seagull model, we set its position and dimensions and we set the variables associated with the bones of the chicken in order to animate the chicken. For the chicken we have three possible animations: walk animation where the chicken walks, wings animation where the chicken moves its wings, eat animation where the chicken moves its head as if it’s eating from the ground.
In the walkingChicken() function we implement the walking animation; we want the left and right leg of the chicken to move, respectively, in range [30 degrees, -30 degrees] and [-30 degrees, 30 degrees]. To do so we check, for the left leg, if the leg rotation around the z-axis is less than 30 degrees and the invert variable is equal to zero; if so we increase the leg angle by the speed of the chicken leg, which is 1 degrees. If the leg angle is greater than 30 degrees, we invert the rotation and we decrease the angle by the speed of the leg. If the leg angle is less than -30 degrees we invert the rotation of the leg. The animation of the right leg is the same, however the leg will move first backwards and then forwards.
During the walking animation the chicken will move forward along the z-axis. Since it can move only in the range of the cage, if its position has reached one of the ends of the cage, the chicken will rotate along the y-axis by 90 or -90 degrees and it will turn back.
In the eatingChick() function we implement the eating animation. During the animation the head and the body of the chicken will rotate along the z-axis in range between the initial and final position. Following the same logic of the walkinChicken() function, if the head is less than the final position and the invert variable is 0, we increase the angle by 1 degree. Once the head is bent over, so its angle is greater than the head's final position, we invert the rotation of the head and we decrease its angle by 1 degree. If the head’s angle is less than the initial position, which means that the head is relieved, we invert again the rotation.
In the flyingChick() function we animate the wings of the chicken; the animation consists in rotating the left and right wings of the chicken along the y-axis in range [50 degrees, 140 degrees] and in range[-140 degrees, -50 degrees]. So for the left wing we check if its angle is less than 140 degrees and the invert variable is 0. If so, we increase the angle of the wing by the speed wing value, so by 2 degrees. If the wing’s angle is greater than 140 degrees we invert the rotation and we decrease it by 2 degrees until its value is less than 50 degrees; at this point we invert again the rotation. The same goes for the right wing.

Lake

The small lake inside the circle of stones is created through the water shader (Water.js). To generate the water we call the buildWater() function in which we define the geometry of the water, a circle; then we define texture properties and the texture to be applied. Then we set its position.

```
function buildWater() {
    const waterGeometry = new THREE.CircleGeometry(30, 30);
     water = new Water(
      waterGeometry,
      {
        textureWidth: 512,
        textureHeight: 512,
        waterNormals: new THREE.TextureLoader().load('./Textures/water/Water_002_NORM.jpg', function ( texture ) {
          texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
        }),
        alpha: 1.0,
      }
    );
    water.rotation.x =- Math.PI / 2;
    water.position.x = 100;
    water.position.y = 1;
    water.position.z = -80;

    return water;
  }
```

The water animation is set in the update() function, where we redraw the scene every time the screen is refreshed (typically around 60 times per second).

```
function update() {

    // Animate water
    water.material.uniforms.time.value += 0.5 / 60.0;

}
```

Human figure

The human figure is a 3D model loaded through the FBXLoader(). As for the other models we set its position and dimensions, we define the variables associated with the skeletal parts of the human's body. For the human body we have two functions: animateModelIdle() and walkAnimation().
In the animateModelIdle() function we set the initial position of the model, we set the angles' values in such a way that the legs are stopped, and the arms are bent.
In the walkAnimation() function we model the human's walk. Following the same logic of the previous animations, we start by moving the human's right and left arms, respectively, in range [-20 degrees, 20 degrees] and [20 degrees, -20 degrees]. To do so, for the right arm, we check if its angle is greater than -20 degrees and the invert variable is equal to zero; if so, we decrease its value by 0.05 degrees. If the right arm's angle is less than -20 degrees we invert the rotation along the x-axis and we increase the angle until it is greater than 20 degrees; at that point we invert again the rotation. The same goes for the left arm.

Following the same procedure we move the torso of the model in a range [10 degrees, -10 degrees].
For the animation of the human's legs, we rotate the left and right legs, respectively, in range [-20 degrees, 20 degrees] and [20 degrees, -30 degrees], but at the same time we also move the left and right calfs. To do so, for the left leg, we firstly check if the leg's angle is greater than -20 degrees and the invert variable for the leg is zero; if so, we decrease the angle value by 0.05 degrees. So in the animation, the human figure will raise forward her leg, and during this movement we start moving the left calf by bending it and then stretching it. To do so, we check if the invert variable for the left calf is zero, and, if so, we rotate the calf forward by increasing its angle by 0.05 degrees. Once the calf's angle has reached 20 degrees, we invert its rotation and we stretch the calf. After inverting it, we additionally invert the entire left leg rotation in order to move the leg backward. To do that, we increase the leg's angles by 0.05 degrees until it reaches 20 degrees; at that point we revert the rotation.
To summarize, during the animation of the legs we have the following steps:

- Move forward the entire leg and during this movement, the calf bends and then stretches.
- Move backward the entire leg.

The same method is applied to animate the right leg, however the right leg will do the opposite movements.
Finally, during the walking animation, we increase the human's position along the z-axis by increasing her hips position. In case the human reaches the end of the sand road, we set her position to the initial one.

User manual

As a user there is the possibility to:

- Move the camera around the scene through the left button of the mouse, to move the camera position in the scene through the right button of the mouse, to zoom the camera with the scroll wheel of the mouse.
- Move the camera position along the x, y, z, through the GUI displayed on the screen.
- Move the seagull using the directional arrows of the keyboard, each arrow will move the seagull forward, backward, to the left and to the right.
- Start/stop the animation of the human model with the switch called 'animationSwitch'.
- Start/stop the animation of the car, to reset the animation which will position the car at its initial position and set the switch animation to false; to modify the speed of the car and the speed of its wheels.
- Start/stop the three possible animations for the chicken, having the possibility to combine them together.