INTERACTIVE GRAPHICS

Final Project

Students:

Bracalello Gianmarco - Matricola: 1551766
Brandi Jessica - Matricola: 1608439

Description of the environment used

This is the project presentation of the final project of the exam of Interactive Graphics of master's degree in Engineering in Computer Science of:

- Gianmarco Bracalello
- Jessica Brandi

The name of our project is "Tesseract Hunt", since it is a run of Iron Man (the main character of our game) on a street with obstacles (car of different types), but also with rewards (Tesseracts and Arc Reactors).

Our game is composed by 11 levels, in each of which the speed, and so also the difficulty, increases.

The goal of the game is to avoid the obstacles and to reach the highest score possible.

For our project, instead of using the basic WebGL, we decided to employ Three.js which is a cross-browser JavaScript library and Application Programming Interface used to create and display animated 3D computer graphics in a web browser. It is based on WebGL, but it provides further additional facilities dealing with all implementation aspects and making easier to build a more complex application.

WORLD:

In order to set up our environment we used the Three.js Scene, which is where we place objects, lights and cameras, since to be able to display anything with Three.js we need three things:

- Scene
- Camera
- Renderer

We used a perspective **camera** with 3 attributes. The first attribute is the field of view, which is the extent of the scene that is seen on the display at any given moment. The second one is the aspect ratio (width of the element divided by the height). The other two attributes are the near and far clipping plane, so objects further away from the camera than the value of far or closer than near won't be rendered.

```
//Scene and Camera
var scene = new THREE.Scene;
var camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);
camera.position.z = -5;
camera.lookAt(new THREE.Vector3(0, 0, 0));
```

We created also a **renderer** to display the scene with the animate loop.

```
//Renderer
var renderer = new THREE.WebGLRenderer();
renderer.setClearColor(0x404040, 1);
renderer.setSize( window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

The **animate loop** will create a loop that causes the renderer to draw the scene every time the screen is refreshed.

```
animate();

function animate(){
    requestAnimationFrame(animate);
    game.update();
    TWEEN.update();
    renderer.render(scene, camera);
}
```

We also used the Three.js library to represent the fundamental geometry of our game: the **street**.

We created a cylinder geometry with a street texture in order to build the street, which is the ground on which Iron Man runs towards the obstacles and the rewards, that also are on this ground.

We adjusted the position of this street and we also rotate it 90 degrees on the z axis to put it horizontally.

```
var geometry = new THREE.CylinderGeometry(this.game.streetRadius, this.game.streetRadius, this.game.streetLength, 64);
var material = new THREE.MeshBasicMaterial({map: this.worldTexture});
var street = new THREE.Mesh(geometry, material);
street.name = 'Street';
this.worldMesh.add(street)
```

We loaded a **background** texture for our scene:

```
// Load background texture
var loader = new THREE.TextureLoader();
loader.load('textures/background.jpeg', function(texture){
    scene.background = texture;
});
```

CHARACTERS:
For the **characters** we used pre-built 3D models, we imported and loaded them in our scene with a glTF extension (Graphics Language Transmission Format).
We load these objects into our project with asynchronous functions like the one below, that makes the object a 3D object that we can animate and add to our scene.

```
async load(scene, callback){
    var loader = new GLTFLoader();
    loader.load('./models/iron_man_mark_46/scene.gltf',
```

Each mesh has its own dimension, position and rotation and in order to make the game a third-person game we adjusted them after the loading.

```
this.mesh.name = "IronMan";
this.mesh.scale.set(.06, .06, .06);
this.mesh.position.y = 6;
this.mesh.position.z = 4.6;
this.mesh.rotateY(Math.PI/2);
```

All the characters are simple meshes, except for **Iron Man**, the main character, which is a **skinned mesh**, so a mesh with a skeleton, with all the characteristics of the other meshes, plus a hierarchical set of interconnected parts called bones, which collectively form the skeleton, a virtual armature used to animate the mesh as a real human.
Each bone of the character is an Object3D with its own position and orientation. Below we can see few examples of retrieving of the body and of the spine of our character.

```
bones() {
    // retrieve bones
    this.body = this.mesh.children[0].children[0];
    this.spine = this.mesh.children[0].children[0].children[0].children[0].children[0].children[2].children[0].children[2].skeleton.bones[7];
```

LIGHTS:
We added three lights to our scene in the index.html file:

- An ambient light
- A directional light
- A point light

The ambient light is a white light that globally illuminates all the objects in the scene equally. This light cannot be used to cast shadows as it does not have a direction.

```
var ambLight = new THREE.AmbientLight(0xffffff, 1);
scene.add(ambLight);
```

The directional light is 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. We use this light to simulate daylight; the sun is far enough away that its position can be considered to be infinite, and all light rays coming from it are parallel.

```
var light = new THREE.DirectionalLight(0xffffff, 1);
light.position.set(0,5,0);
scene.add(light);
```

The point light is a light that gets emitted from a single point in all directions. We put this light in the Tesseract to simulate the emission of the light.

```
var pntLight = new THREE.PointLight(0xffffff, 2);
pntLight.position.set(0,0,0);
scene.add(pntLight);
```

STYLE:

The game is composed of two scenes: the initial scene and the playing scene. The first scene is where the player is introduced to the game and is based on the stylesheet present in game.css where, in collaboration with the HTML tags, the style of the scene is built.

There are divs with the name of the game and the instructions, so that the player knows how to play the game and the instructions to start to play.

In this scene is also implemented a listener that allows the player to go to the playing scene by pressing “Enter”. Here all the models are loaded to build the environment where the actual game will start.

In the playing scene the div with the level, distance and energy is added, by using the style visibility tool. All the divs regarding the progress in the game are collected inside a wrapper to show them together when the player starts to play.

Libraries, tools and models

LIBRARIES:

The main goal of this project is to animate a hierarchical model in order to appreciate the power of a library such as Three.js in rendering a complex 3D model.

The library Three.js allows to add 3D models to a web page without the overhead of the code lines needed to do the same in WebGL; furthermore WebGL renders only basic elements such as points, lines and triangles.

Once we loaded the library we had to animate the loaded models and, to do that, we used another library called Tween.js.

Tween.js allows us to have a smoother movement by taking in input its starting and ending position and duration, applying an interpolation function.

MODELS:

The models we loaded in the scene are divided in two main categories:

- Obstacles
- Main character

They are represented by a mesh that we add to the scene.

The obstacles are all elements that can be found in a city environment and so we have a Car, a Police Car and a Taxi. Added to the scene there are also two more models representing the bonus that the player can get to increase its energy: the Tesseract and the Reactor.

All models are loaded from a glTF file downloaded from Sketchfab by using a gltf.scene file that can be referenced in the code and then are adjusted to have a position that is coherent with the scene in which the game is played.

Technical aspects of the project

RANDOM GENERATOR

As soon as the elements of the initial scene are removed, the *play()* function starts to spawn the elements that then will form the playing scene.
The spawning function takes as input the instances of the obstacle we want to generate and calls a random generator that spawns the obstacles in the scene.
We divide the circumference in *n* parts and through a switch-case clone the mesh and set up the position on the street.
Finally we add the obstacle to the mesh, eventually move it and add to the collidableArray.

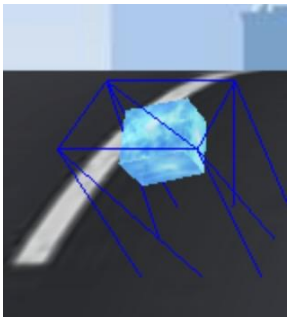
COLLISION DETECTION

In order to check if Iron Man hits something in the world we developed a collision detection system based on the *BoundingBox* intersection.
We created a *BoxGeometry* around each object in the scene and we computed the corresponding *BoundingBox* as we can see below.

```
loader.load('./models/iron_man_mark_46/scene.gltf', object => {
    this.mesh = new THREE.Mesh(
        new THREE.BoxGeometry(50, 150, 55),
        new THREE.MeshBasicMaterial({opacity: 0.0, transparent: true})
    ).add(object.scene);

    this.mesh.geometry.computeBoundingBox();
```

Here we have some examples of the boxes around the objects:


We used the function *intersectsBox()* to check if the collision between Iron Man and an obstacle happened.

If the Iron Man BoundingBox intersects the BoundingBox of an obstacle (Car, Police Car and Taxi) the energy of the player is decremented.

```
if (ironmanBB.intersectsBox(obstacleBB)){

    var obstacle = this.game.collidableArray[i].children[0].name;

    switch (obstacle) {
        case 'Car':
            this.removeEnergy(this.game.carValue);
            break;
```

While if the intersected object is a Tesseract or a Reactor the energy is incremented.

```
case 'Tesseract':
    this.addEnergy(this.game.tesseractValue);
    break;
```

We have two functions to modify the energy of the player:

- **addEnergy()**, in which we add to the energy of the player a different value for each reward object (Tesseract or Reactor), by modifying its value on the HTML.
- **removeEnergy()**, in which we subtract to the energy a different value for each obstacle object (Car, Police Car, Taxi), by modifying the value on the HTML. In this function we also check if the energy of the player is 0 and, in that case, we make the 'Game Over' div visible and, after a while, we reload the game.

RENDERING LOOP

To keep the game running, in the rendering loop, we called a function update() that updates all the features of the game.
Here, at each level, we increase the rotation of the street in order to also increase the difficulty of our game and make it even more interactive.
It is in this function that we keep the game alive.

Animation

An important part of the project is the animation engine to build the best playing experience possible.

WORLD:

To animate the world we choose to move the world around the main character keeping the orientation always the same, by doing this we have created the illusion that the character is running.

To animate the whole world we created a mesh representing the street and we added each obstacle to the street so that the rotation of the street would be inherited by each one of them.

To make the game more challenging we created a function *update()* in which we increase the speed of rotation at each level, without using a Tween.js

MODEL:

The animation for the model is implemented for the Tesseract and Iron Man. For what concerns the Tesseract, a simple rotation is implemented using Tween.js to make it smooth. The rotation is implemented in the *move()* function around two axes for both the Tesseract in the initial scene and the playing scene.

The animation of the player (Iron Man) is a bit more complex, since it is a hierarchical model. We needed to retrieve the skeleton and modify it as we wanted by applying one or more Tweens to each bone.

To move the model as humanly as possible we needed that the Tweens applied to the bones started together so we built a Tween.Group() and added each Tween of the main character to it. Then, in a separate function, we fetch all the Tween and start them.

The functions we used to move the character are:

- *run()*: in this function there are all the Tweens regarding the skeleton of the character and each Tween is chained to form a loop to allow the infinite repetition. At the end of the creation of the Tweens is called the function startRunningTweens(), that actually starts the group of Tweens.
- *moveLeft()*: in this function there is just one Tween that translates the character towards the left allowing the player to collect the rewards and avoid the obstacles.
- *moveRight()*: in this function there is just one Tween that translates the character towards the right allowing the player to collect the rewards and avoid the obstacles.
- *jump()*: in this function we used two Tweens to implement the two parts of jumping, then we chained the Tweens and added them to the jumpTweens group that is then started in the function startJumpTweens().

User Manual

- ➔ As soon as the initial page is loaded, after waiting a while to load the models, the game is ready to start by pressing Enter.
- ➔ The game starts with the camera pointing to Iron Man running on the street. The game can begin, collecting rewards and avoiding obstacles. If the character hits the obstacles the energy decreases and if it hits the rewards the energy increases.
- ➔ At every level the rotation speed, and hence the difficulty, increases.

- ➔ It is possible to avoid obstacles and collect rewards by moving left pressing **A** key and moving right pressing **D**, it is also possible to jump by pressing the **SpaceBar**.
- ➔ The goal of the game is to run for as long as possible, increasing the energy without hitting obstacles and losing all the energy.