***ZOMBIE SURVIVAL***

***Interactive Graphics Project***

***A circuit board

Description automatically generated***

***Nicolò Mantovani,1650269***

***Gabriele Nicosanti,1599265***

**1. Introduction**

The aim of the project was to create a 3D game through a library named ‘*three.js’*, one of the most popular JavaScript framework for displaying 3D content on the web.

We developed a first person shooter game whose rules are simple: the player is surrounded by a fixed number of zombies and, in order to save its own life and win the game, he must kill them all. If the player is touched by a zombie at least one time then the game is ended, otherwise the player can freely run within the map bounds aiming to shoot the zombies.

We have arranged four levels of difficulty (easy, medium, hard and madness), each in a separate JavaScript file, to let the player have a wider game experience and a brief tutorial that shows how to play the game.

**2. Three.js**

Three.js is a JavaScript library and an API (Application Programming Interface) that is used to create and display animated 3D computer graphics in a web browser without relying on proprietary browser plugin: this is possible due to the advent of *WebGL.*

Some of the main features of this API are:

* Scenes: add and remove objects at run-time;
* Cameras: perspective, orthographic, cube, array and stereo;
* Geometry: planes, cubes, spheres, torus;
* Objects: meshes, particles and more;
* Materials: basic, Lambert, Phong, smooth shading, texture and more;
* Lights: ambient, directional, point, spot and shadows;
* Data loaders: binary, image, JSON and scene.

**3. Scene, environment and camera**

Through the method *‘Scene()’* it is possible to create the scene, which allows to set up what and where is to be rendered: objects, lights and cameras.

We have placed a texture, as background, that represents a red/orange sky and a red fog inside the scene in order to recreate the post-apocalyptic atmosphere. The player holds a weapon, whose model has been imported **(FROM?)**, and the environment in which he can move is composed of buildings, sidewalks, lamps and the ground. Each of them has been first imported from **(repo name?)** and then some of them have been modified by us as follows:

**GLI DICIAMO COME SONO FATTI O GLI METTIAMO LE PICCOLE MODIFICHE APPORTATE DA NOI?**

* Sidewalks: re-scaled (and changed the color (?), if we want) and added for each a bounding box to improve the collision management;
* Lamps: re-scaled
* Buildings:

The camera always looks ahead in front of the player and we chose to use the ‘*PerspectiveCamera’* since it mimics the way the human eye sees and hence matches perfectly the first person shooter games.

**4. Lights and materials**

The basic idea was to place a light of type ‘*PointLight’* (a light scattered from a point towards all directions) for each lamp. But since the number of lamps was 1600, this huge number of lights would be too heavy for the project. We therefore decided to follow another way: let lamps turn on around the player in a fixed radius and turn off the ones that were already turn on as soon as the player moved. Unfortunately, this way also turned out to be computationally heavy and we hence adopted just one light that worked the same way as the sun: *‘HemisphereLight’*, a light source positioned directly above the scene with color fading from the sky color to the ground color.

Since the *‘MeshBasicMaterial’* is not affected by lights, for most of the object present in the scene we used the *‘MeshLambertMaterial’*, a material for non-shiny surfaces without specular highlights. The shading is calculated using the *‘Gourad’* shading model, which calculates shading per vertex and interpolates the results over the polygon’s faces. Another important reason is the performance: due to the simplicity of the reflectance and illumination models, performances are greater than *‘MeshPhongMaterial’, ‘MeshStandardMaterial’* and *’MeshPhysicalMaterial’* at the cost of some graphical accuracy.

**5. Collision**

One of the most difficult things to implement, along with the zombies, were the collisions. In a first phase of the project we used the *‘Raycaster’*, a class mainly used for mouse picking (working out what objects in the 3D space the mouse is over) amongst other things. The *raycaster* simply provide a ray casted from an origin towards a direction and check if the ray hits something: if so, an array of sorted objects (distance metric) is returned and actions can be taken. We hence used an array of 16 rays casted towards the cardinal direction: North, East, West, South, North-East etc.. In the render loop this method was computationally heavy and therefore we adopted another way: bounding boxes all over the object present in the scene. Indeed, *three.js* provides a boolean method named *‘intersectsBox’* that check if two boxes intersect themselves. Collision are done as follows:

* Player and buildings: since the sidewalks were part of a unique geometry, we took each sidewalk separately in order to have a different bounding box for each sidewalk. Considering that the method *‘intersectsBox’* works only between two bounding boxes, all around the player feet we placed a transparent box that always follows the camera to allow the function to check collisions;
* Player and map bounds: to deny the player from leaving the map we decided to put four invisible walls;
* Bullets and zombies: since the camera is fixed and it is not allowed to move it up and down, the collision between bullets and zombies can only occurs at the height of the zombie’s head (for the orange zombies). **GABBO LA COLLISIONE CON GLI ALTRI 2 TIPI DI ZOMBIE**

**6. Zombies as hierarchical models**

**Gabbo parlaci tu degli zombie che sai come li hai fatti. La parte che parla del modello dell’arma la vogliamo mettere o non serve?**

**7.Textures o no (?)**