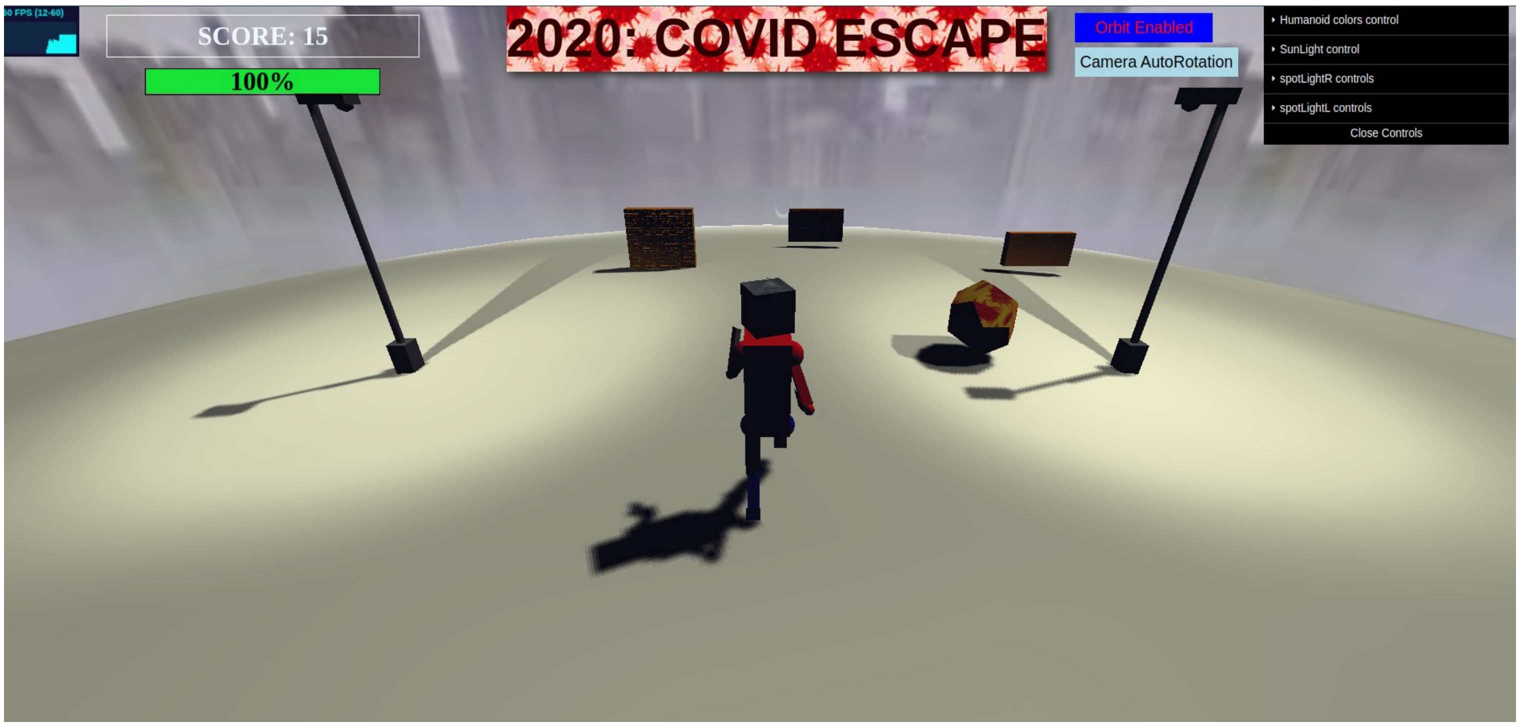
IG PROJECT: 2020 COVID ESCAPE

Marussi Diego – 1962008

1 – Introduction

This project consist in the development of a game called "2020: Covid Escape".
This game is basically an "Endless Run" where you have to avoid the wall obstacles or worst, the covid! And score as many points as possible as you progress through the race.

You can play or interact with the scene in many ways, moving the character with the keyboard, changing some properties of objects in the scene and take a look around with the camera movements/rotation.



1.1 – Project Requirements:

Hierarchical model:

In this scene we have two principal hierarchical models, the *Humanoid* and the *Street Lamp* .
The most complex one is the humanoid models compose by 22 body parts that represent a kind of human.
Then you can find several street lamps right and left in the scene composed by 4 parts, but not associated to any animations.

Lights and textures:

In the scene we can find three different types of lights:

- one Ambient Light,
- one Directional Lights,
- two Spot Lights.

Different textures were used, like normal texture (normal map) for the street/ground, for the body of the Humanoid, for the Street lamp, for the wall. While image texture for the head of the humanoid and the virus objects.

User interaction:

There are several types of user interaction with the object and the scene.

- Keyboard movements to move the humanoid left and right to avoid the obstacle.
- Button to perform different action with the camera, using Orbit control to rotate/move the camera around through the mouse, or make the camera autoRotating around the scene or even choose to play in 3° or 1° person (moving the camera behind or in front of the humanoid and attach the camera to it).
- The GUI controls that permits you to change properties of some object, like the position, the color, and other.

Animations:

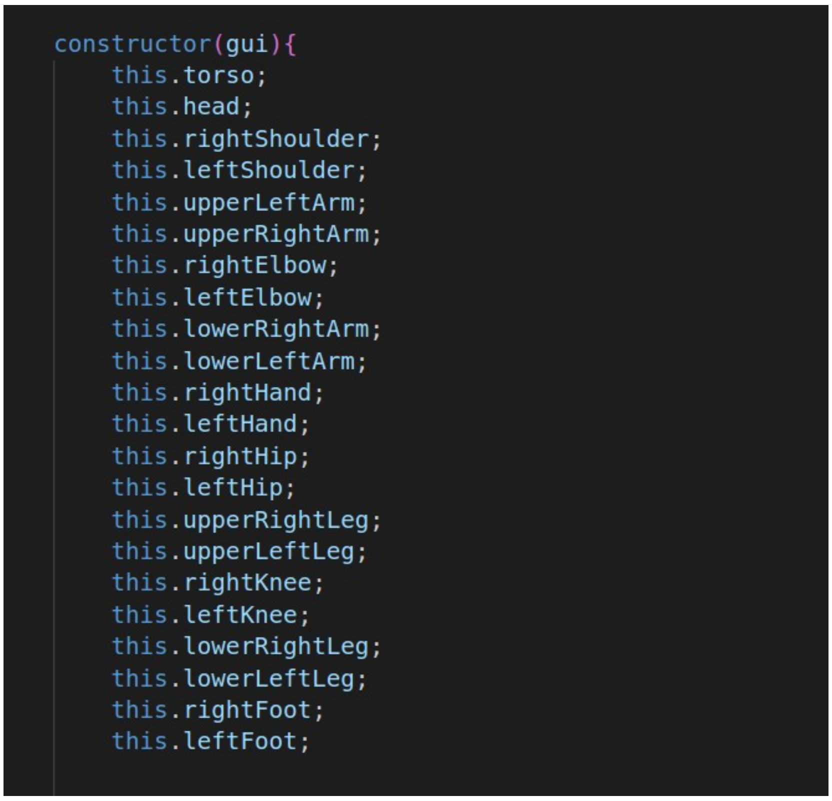
In this game we have a several animations for different objects, not imported but developed using javascript or the TweenJs library. We have left out several animation in order not to make the rendering of the game too heavy.
You can find the running animation of the humanoid (the most complex), of the head of the humanoid floating, or the one of the sun light, and also the rotation of the ground/street to have the perception of movement of objects.

2 – Environment

2.1 – Objects and 3Dmodels

2.1.1 – Human Hierarchical structure object

The hierarchical model has been created uniquely with threeJs, joining different mashing in a parent-child relation. The models structure is very similar to an human structure, simplified just with the structure useful for the running motion. It is added to the scene at the level of the ground.
This is a look at the constructor of the humanoid class used to develop the model by listing all its parts:



```
constructor(gui){
    this.torso;
    this.head;
    this.rightShoulder;
    this.leftShoulder;
    this.upperLeftArm;
    this.upperRightArm;
    this.rightElbow;
    this.leftElbow;
    this.lowerRightArm;
    this.lowerLeftArm;
    this.rightHand;
    this.leftHand;
    this.rightHip;
    this.leftHip;
    this.upperRightLeg;
    this.upperLeftLeg;
    this.rightKnee;
    this.leftKnee;
    this.lowerRightLeg;
    this.lowerLeftLeg;
    this.rightFoot;
    this.leftFoot;
```

As you can see the models is subdivided starting with the ‘torso’, the main part of the body which is added to the scene. Then going down we add the bod part connected as the human body, like the ‘shoulder’ connect to the ‘torso’ and the ‘upper arm’, or moving to the leg, the ‘upper leg’ is connected to the ‘hip’ and down to the ‘knee’.

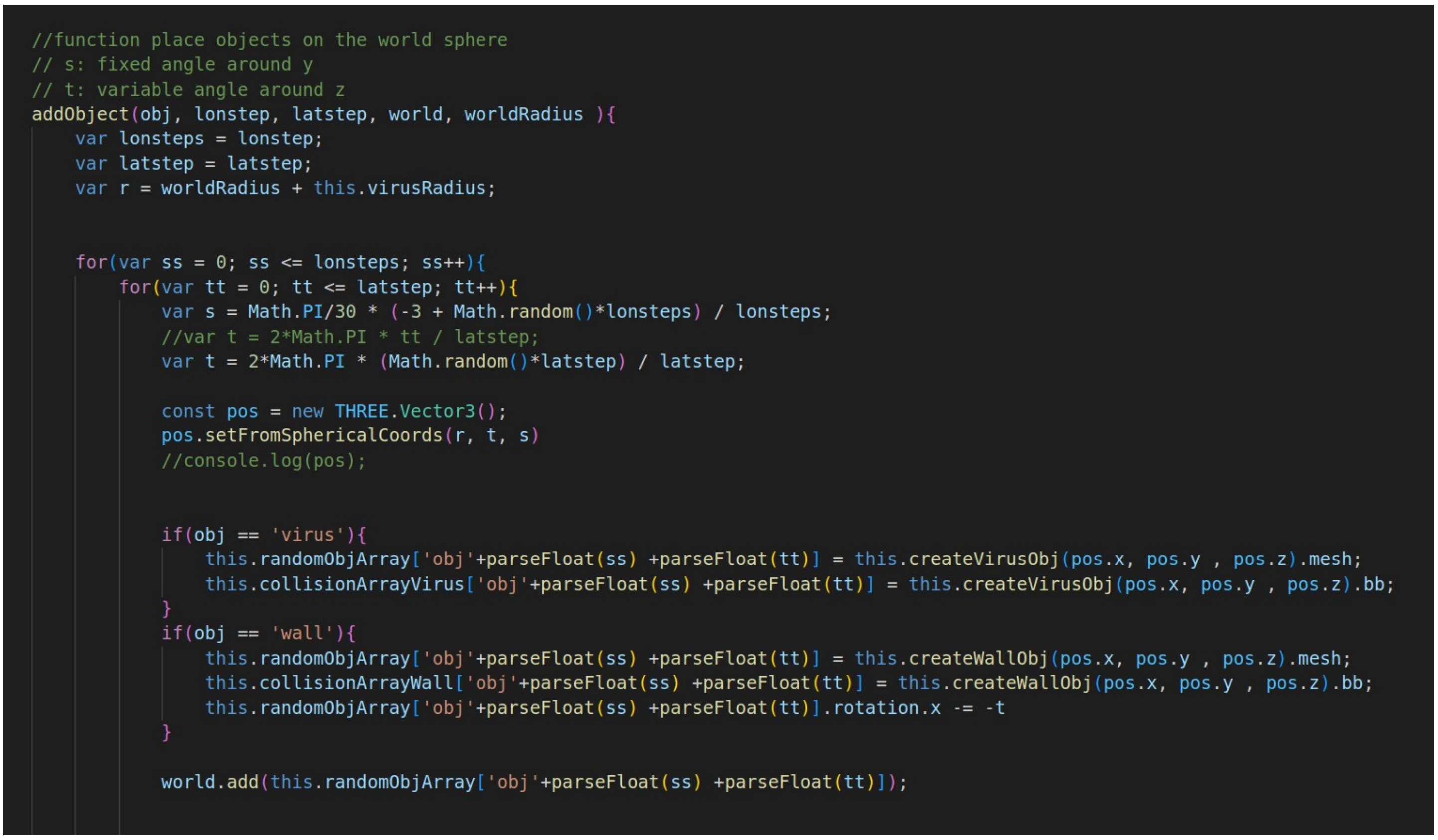
The Three geometry used are Box and Sphere with different dimensions.
An important thing is that like the human body the movements will be exploit by the rotational joint of the body, like the shoulder, the elbow, the hip and the knee.

2.1.2 – Environment Objects

We have different objects into the scene where we have the main object that is the world/street that is added to the scene. Instead all the other object are attached to the ground as child objects to undergo the same transformations.
The objects are defined in different classes and initialized in the main.js as class objects.

The difficulty is occurred when I had to place the object on a sphere object in a restrict range.
This was possible using a function to place object on the world/street called “addObjects” or “addStreetLamps” that through the “setFromSphericalCoords” function and setting in the right way the polar parameters allowed us to place the Objects exactly on the surface of the Ground sphere. Then I have to correct the orientation of the object placed rotating by the same angle (‘t’) used to place it on the polar coordinate for each object.
All of this through a For cycle to iterate the creation and addition of the Objects.

A look in the “addObjects” function:



```
//function place objects on the world sphere
// s: fixed angle around y
// t: variable angle around z
addObject(obj, lonstep, latstep, world, worldRadius ){
    var lonsteps = lonstep;
    var latstep = latstep;
    var r = worldRadius + this.virusRadius;


    for(var ss = 0; ss <= lonsteps; ss++){
        for(var tt = 0; tt <= latstep; tt++){
            var s = Math.PI/30 * (-3 + Math.random()*lonsteps) / lonsteps;
            //var t = 2*Math.PI * tt / latstep;
            var t = 2*Math.PI * (Math.random()*latstep) / latstep;

            const pos = new THREE.Vector3();
            pos.setFromSphericalCoords(r, t, s)
            //console.log(pos);


            if(obj == 'virus'){
                this.randomObjArray['obj'+parseFloat(ss) +parseFloat(tt)] = this.createVirusObj(pos.x, pos.y , pos.z).mesh;
                this.collisionArrayVirus['obj'+parseFloat(ss) +parseFloat(tt)] = this.createVirusObj(pos.x, pos.y , pos.z).bb;
            }
            if(obj == 'wall'){
                this.randomObjArray['obj'+parseFloat(ss) +parseFloat(tt)] = this.createWallObj(pos.x, pos.y , pos.z).mesh;
                this.collisionArrayWall['obj'+parseFloat(ss) +parseFloat(tt)] = this.createWallObj(pos.x, pos.y , pos.z).bb;
                this.randomObjArray['obj'+parseFloat(ss) +parseFloat(tt)].rotation.x -= -t
            }

            world.add(this.randomObjArray['obj'+parseFloat(ss) +parseFloat(tt)]);
```

World/Street
The Street or Ground where the run take place is a Three.Sphere object of a large radius r=200 to seems a straight ground without spherical curvature.
It has a grey/yellow color and a “normal map” similar to a ground or a street.

Walls
Represent the basic obstacle of the game and we find it more frequently in the scene. It is a Three.Box geometry with different measures using the function Math.Random().
It has a normal map representing the bricks of a wall and the color

Virus
The virus object represent the covid virus that the character has to avoid together with the wall. This is more dangerous than the wall. It is a Three.Dodecahedron geometry with a texture of virus.

Street Lamps
This is another hierarchical structure, but more simpler than the humanoid.
It is composed by four parts: three Three.Box geometries and one Three.Sphere geometry. The firsts three one represent the base, the pole and the masklamp of the street lamp while the sphere represent the bulb of the light of the lamp. It has a grey color with a detailed normal map.

2.1.3 – Glb 3Dmodels

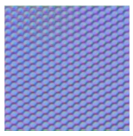
We tried to use imported glb or gltf objects but they are slightly heavy for the rendering and to manipulate. We tried the “covid models” and the “surgical mask models” (that you still can find in the scene).

2.2 – Material and Textures

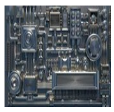
Starting with the materials we used mainly the Three.MeshStandardMaterial that represent the objects more similar to the reality reacting to the effect of the lights and having multiple properties for the material aspect. The loading of the texture is done through a function of the class Utils called “getTexture” that using the *textureLoader* function permits to load textures on the object materials.

All the objects has the ability to cast the shadows but one, i.e. the world sphere geometry that instead can receive the shadow, enabled that property.

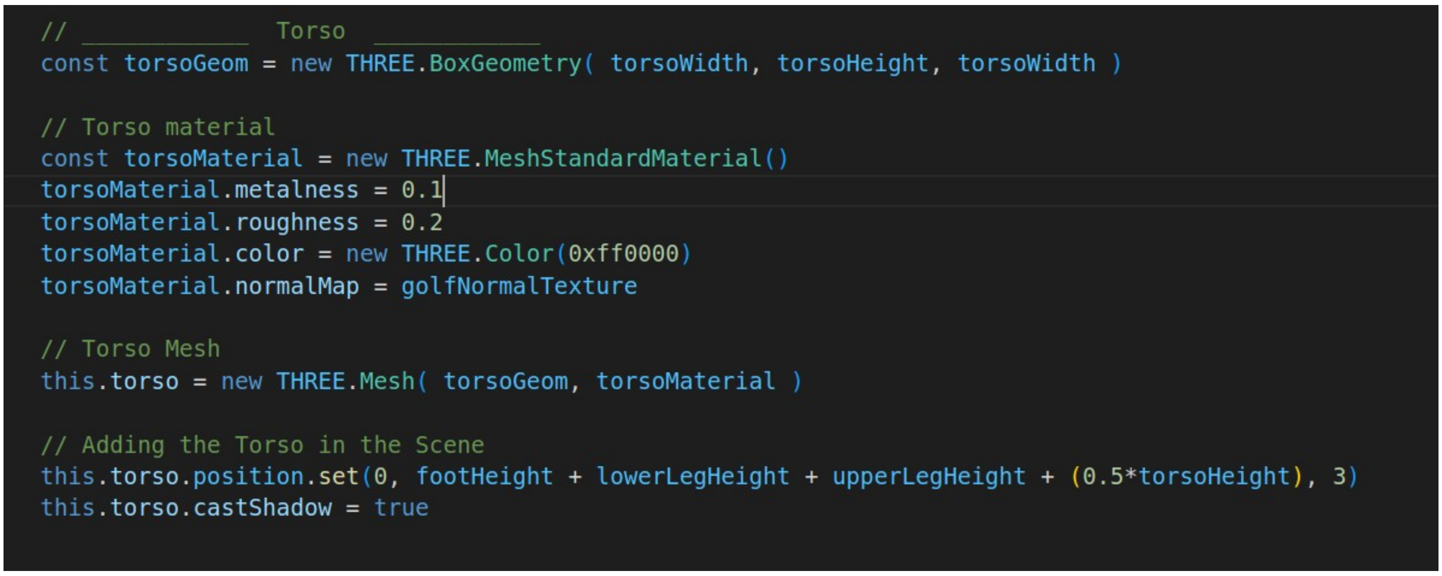
The humanoid has a normal map similar to the one of a golf ball (“golfNormalMap.png”), a non metallic but smooth surface with the “metallic” and “roughness” parameters for the material. It has a red color for the upper-part of the body and a blue color for the lower-part, but the color can be changed.



For the head of the humanoid we used a MultiMaterial vector to associate a different textures (“roboticsTextures.jpg” and just one “smile1.jpg”) to each face of the cube-head, keeping the same properties of the Standard material.

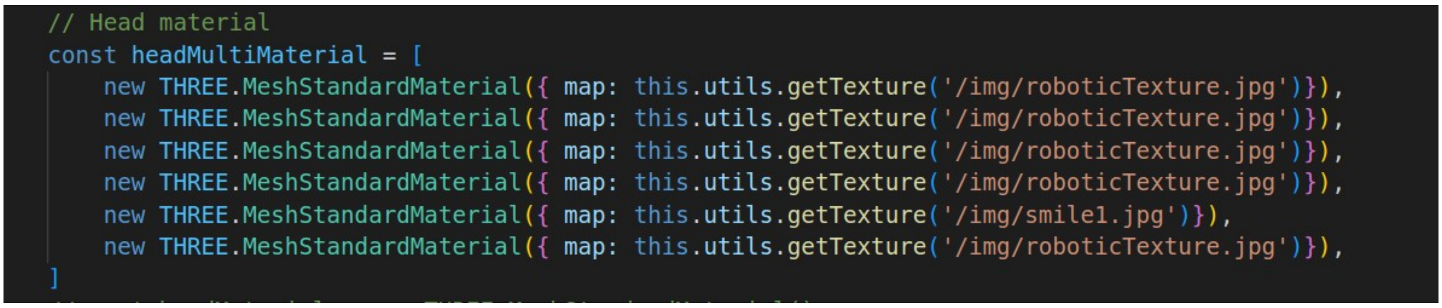


This an example of the torso mesh with its material properties:



```
// ____________    Torso   ____________
const torsoGeom = new THREE.BoxGeometry( torsoWidth, torsoHeight, torsoWidth )

// Torso material
const torsoMaterial = new THREE.MeshStandardMaterial()
torsoMaterial.metalness = 0.1
torsoMaterial.roughness = 0.2
torsoMaterial.color = new THREE.Color(0xff0000)
torsoMaterial.normalMap = golfNormalTexture

// Torso Mesh
this.torso = new THREE.Mesh( torsoGeom, torsoMaterial )

// Adding the Torso in the Scene
this.torso.position.set(0, footHeight + lowerLegHeight + upperLegHeight + (0.5*torsoHeight), 3)
this.torso.castShadow = true
```

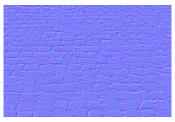
the Head multi-Material:



```
// Head material
const headMultiMaterial = [
    new THREE.MeshStandardMaterial({ map: this.utils.getTexture('/img/roboticTexture.jpg')}),
    new THREE.MeshStandardMaterial({ map: this.utils.getTexture('/img/roboticTexture.jpg')}),
    new THREE.MeshStandardMaterial({ map: this.utils.getTexture('/img/roboticTexture.jpg')}),
    new THREE.MeshStandardMaterial({ map: this.utils.getTexture('/img/roboticTexture.jpg')}),
    new THREE.MeshStandardMaterial({ map: this.utils.getTexture('/img/smile1.jpg')}),
    new THREE.MeshStandardMaterial({ map: this.utils.getTexture('/img/roboticTexture.jpg')}),
]
```

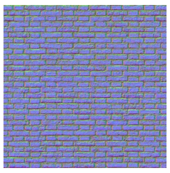
We have choose a cubic background that is all around the scene utilizing the c*ubeTextureLoader* function to load the six texture for the background.

The World/Street has a standard material with a normal texture (“streetNormal.png”) representing a road/ground texture with a grey color.



The Virus always a standard Material with a texture of red viruses (“virusTex.jpg”) and a yellow background color. Its material has an high roughness value parameter because its rough surface.

The wall as mentioned before as a Brick texture associated with an orange color, an high roughness value parameter and no metallic.



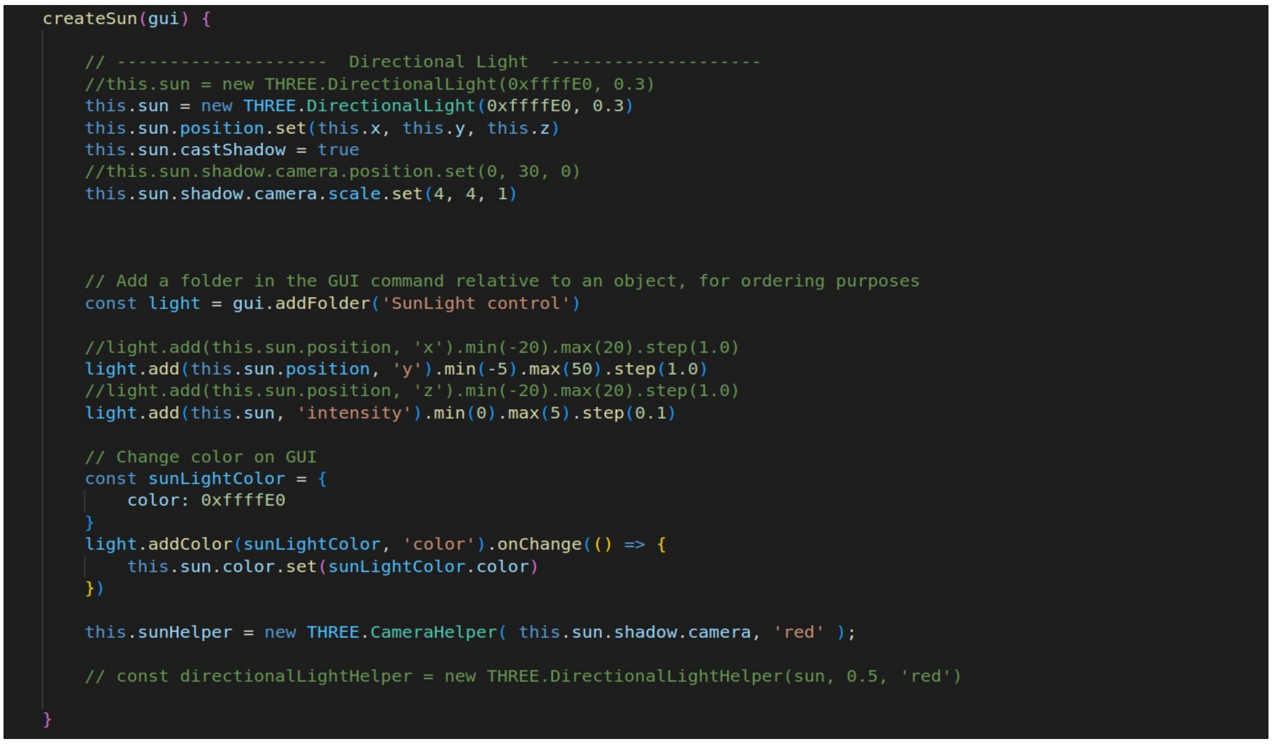
The StreetLamps has instead an high metallic parameter, and a normal map associated with a grey color. But the bulb has different parameter due to its different composition.

A particular detail goes that we have create normal map to associate with the Standard Material component of the meshes using an online toolbox that permits to transform an image into a png normal map to use in the ThreeJS objects called: *“NormalMap – Online”*.

2.3 – Lights

2.3.1 – Sun Light

The sun light is a Three.DirectionalLight, i.e. a light that gets emitted in a specific direction, in fact this light will behave as though it is infinitely far away and the rays produced from it are all parallel (like the sun).

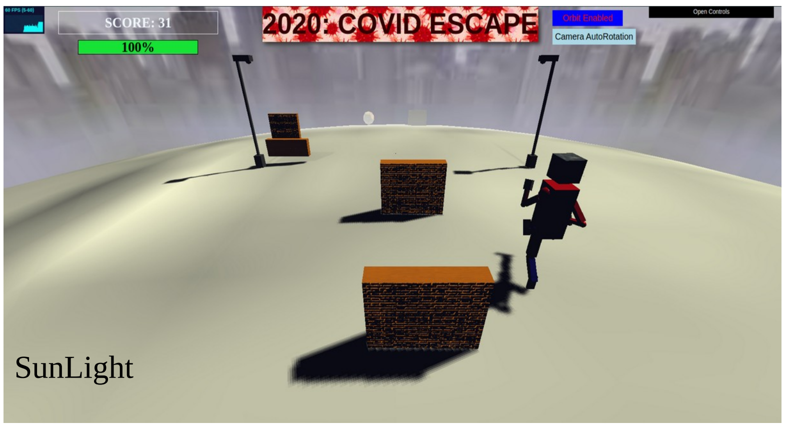


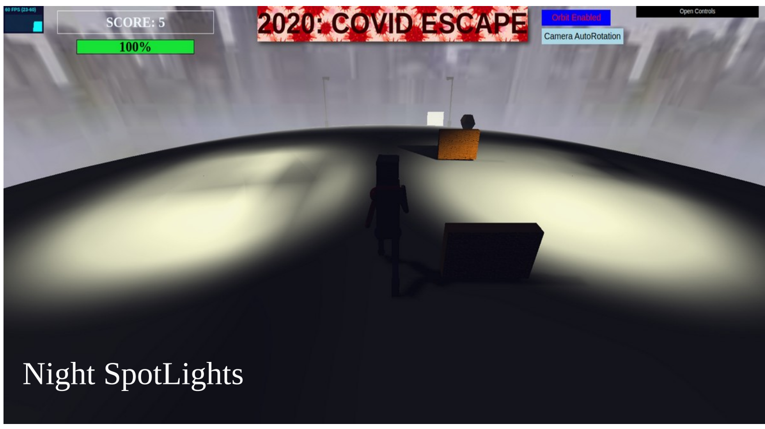
```
createSun(gui) {

    // --------------------  Directional Light  --------------------
    //this.sun = new THREE.DirectionalLight(0xffffE0, 0.3)
    this.sun = new THREE.DirectionalLight(0xffffE0, 0.3)
    this.sun.position.set(this.x, this.y, this.z)
    this.sun.castShadow = true
    //this.sun.shadow.camera.position.set(0, 30, 0)
    this.sun.shadow.camera.scale.set(4, 4, 1)



    // Add a folder in the GUI command relative to an object, for ordering purposes
    const light = gui.addFolder('SunLight control')

    //light.add(this.sun.position, 'x').min(-20).max(20).step(1.0)
    light.add(this.sun.position, 'y').min(-5).max(50).step(1.0)
    //light.add(this.sun.position, 'z').min(-20).max(20).step(1.0)
    light.add(this.sun, 'intensity').min(0).max(5).step(0.1)

    // Change color on GUI
    const sunLightColor = {
        color: 0xffffE0
    }
    light.addColor(sunLightColor, 'color').onChange(() => {
        this.sun.color.set(sunLightColor.color)
    })

    this.sunHelper = new THREE.CameraHelper( this.sun.shadow.camera, 'red' );

    // const directionalLightHelper = new THREE.DirectionalLightHelper(sun, 0.5, 'red')

}
```

As you can see in the code a position of the lights has been chosen, and the ability to cast shadows enabled. So in this way the object can project their shadows on an object in which was settled the ability to receive shadow, in this case, out world sphere geometry.
Moreover I added a gui control on this light for some properties like intensity and color.

2.3.2 – Street Lights

The street Lights are none other than two simple spot lights fixed in the scene to permit to see the 3dObjects when the sun is down and so having some light in the scene.
The first idea was to attach a spotLight to each bulb of the street Lamps, but this idea was too computationally expensive, to add all that lights (like 30s) and calculate all the shadows and the variation on the color in the fragment shader, at a lower level.
So, the code is pretty similar of the one of the SunLight with the gui controls too.
But in this case I added the properties “penumbra” to have a more imperfect light on the ground.





3 – Animations

The animations are mostly developed with the TweenJs library to perform some action, changes in relation to the time. All the animation are linked by the ‘*velDifficulty*’ variable that represent the difficulty of the game as well as the velocity of the animation/transformation.

3.1 – Humanoid Running Animation

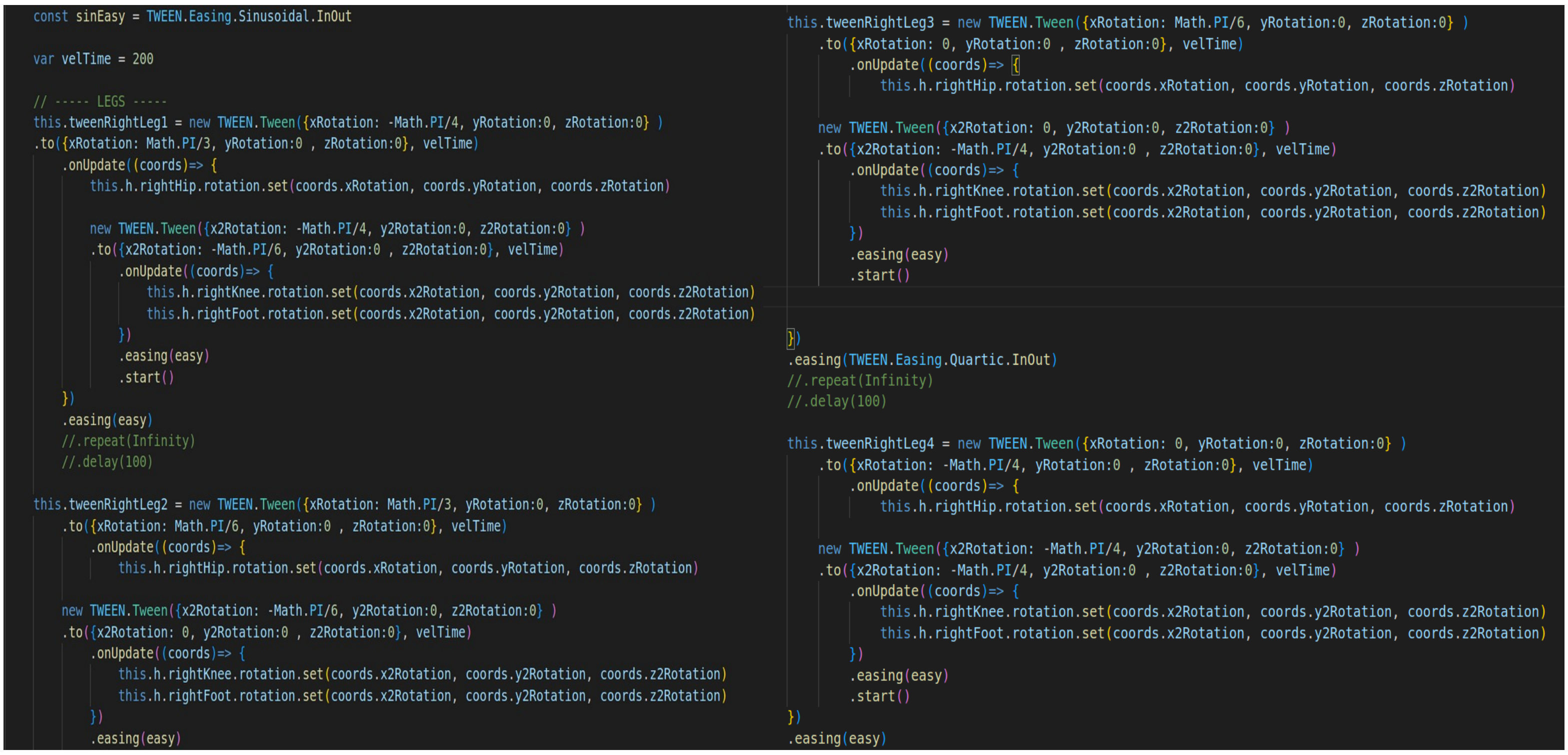
The complex animation was done in the case of the running animation of the humanoid character.
I used 4 different time step Tweens to develop the Running animation (and amateur drawings to see the evolution of the movements) where at each time step different rotations occurs to specific parts of the body. The body parts involved in the rotation are:

- The right and left Hips: angles[-PI/4, PI/3, PI/6, 0];
- The right and left Knees: [-PI/4, -PI/6, 0, -PI/6];
- The right and left Shoulders: angles [0, -PI/4, 0, PI/4] (while the elbows keep an angle of 120° degree).

Obviously the right and left leg as for the arms are in different time step so different angle to represent the animations

The angles at each time step for the hip and the knee, or for the shoulder in the case of the arm were developed slowly and experimenting from time to time to see the best transformations to represent the running movements.

Here an example of the Right Leg tween animation:



```javascript
const sinEasy = TWEEN.Easing.Sinusoidal.InOut

var velTime = 200

// ----- LEGS -----
this.tweenRightLeg1 = new TWEEN.Tween({xRotation: -Math.PI/4, yRotation:0, zRotation:0} )
.to({xRotation: Math.PI/3, yRotation:0 , zRotation:0}, velTime)
    .onUpdate((coords)=> {
        this.h.rightHip.rotation.set(coords.xRotation, coords.yRotation, coords.zRotation)

        new TWEEN.Tween({x2Rotation: -Math.PI/4, y2Rotation:0, z2Rotation:0} )
        .to({x2Rotation: -Math.PI/6, y2Rotation:0 , z2Rotation:0}, velTime)
            .onUpdate((coords)=> {
                this.h.rightKnee.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
                this.h.rightFoot.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
            })
            .easing(easy)
            .start()
    })
    .easing(easy)
    //.repeat(Infinity)
    //.delay(100)

this.tweenRightLeg2 = new TWEEN.Tween({xRotation: Math.PI/3, yRotation:0, zRotation:0} )
    .to({xRotation: Math.PI/6, yRotation:0 , zRotation:0}, velTime)
        .onUpdate((coords)=> {
            this.h.rightHip.rotation.set(coords.xRotation, coords.yRotation, coords.zRotation)

    new TWEEN.Tween({x2Rotation: -Math.PI/6, y2Rotation:0, z2Rotation:0} )
    .to({x2Rotation: 0, y2Rotation:0 , z2Rotation:0}, velTime)
        .onUpdate((coords)=> {
            this.h.rightKnee.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
            this.h.rightFoot.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
        })
        .easing(easy)
```

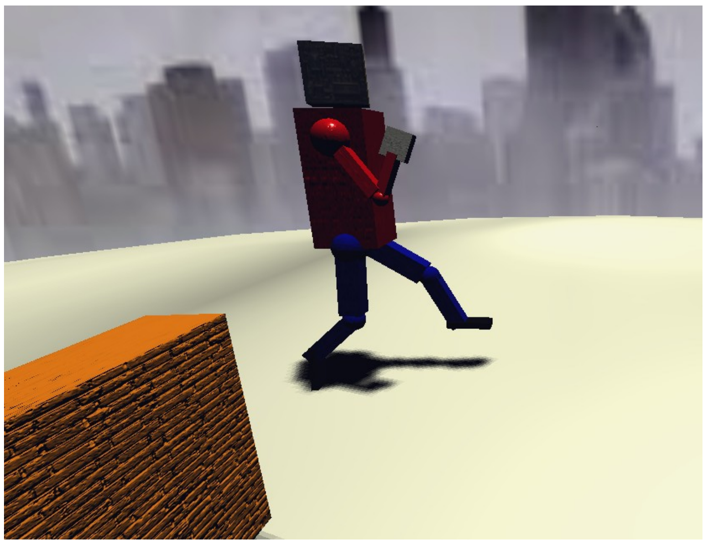
```javascript
this.tweenRightLeg3 = new TWEEN.Tween({xRotation: Math.PI/6, yRotation:0, zRotation:0} )
    .to({xRotation: 0, yRotation:0 , zRotation:0}, velTime)
        .onUpdate((coords)=> {
            this.h.rightHip.rotation.set(coords.xRotation, coords.yRotation, coords.zRotation)

    new TWEEN.Tween({x2Rotation: 0, y2Rotation:0, z2Rotation:0} )
    .to({x2Rotation: -Math.PI/4, y2Rotation:0 , z2Rotation:0}, velTime)
        .onUpdate((coords)=> {
            this.h.rightKnee.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
            this.h.rightFoot.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
        })
        .easing(easy)
        .start()


})
.easing(TWEEN.Easing.Quartic.InOut)
//.repeat(Infinity)
//.delay(100)

this.tweenRightLeg4 = new TWEEN.Tween({xRotation: 0, yRotation:0, zRotation:0} )
    .to({xRotation: -Math.PI/4, yRotation:0 , zRotation:0}, velTime)
        .onUpdate((coords)=> {
            this.h.rightHip.rotation.set(coords.xRotation, coords.yRotation, coords.zRotation)

    new TWEEN.Tween({x2Rotation: -Math.PI/4, y2Rotation:0, z2Rotation:0} )
    .to({x2Rotation: -Math.PI/4, y2Rotation:0 , z2Rotation:0}, velTime)
        .onUpdate((coords)=> {
            this.h.rightKnee.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
            this.h.rightFoot.rotation.set(coords.x2Rotation, coords.y2Rotation, coords.z2Rotation)
        })
        .easing(easy)
        .start()
})
.easing(easy)
```

In this code you can see how at each tween is linked another tween that connect the movements of the rightHip with that of the right knee. The same is valid for the left Leg.

In the same way work also the Arms tweens.

To connect the various time step I used the “*chain*” function of the tweens library that permits to link one tween to another one and, if desired, create a loop connecting the last tween with the first one.

A view of two step of the running animation:



3.2 – Sun Animation

The Sun Animation is also developed with the Tween library. This animation is used to show the time change in a day, from day to night and viceversa changing the intensity of the light and the position.

So, here there are two tween, separated because they have different duration of execution, one for the intensity of the light of the sun and one for the position of the sun.
The first tween lasts 10 second and goes from a light intensity of 0.05 to an intensity of 1.0 and viceversa using the Yoyo tween function that permits to play an animation go back and forth.
The same Yoyo function is used in the position animation that change the position of the sun only on the x and z-axis from the position x = -20, z=3, to the position x = 20, z= -3 (the y position is always = 20), but this animation lasts 20 second, because is valid both for the day and night (twice the one of the intensity).
So the you see the light of the sun change and also the shadow of the objects in the scene representing the passing of the day.

3.2 – SpotLight Animation

There is also the spotLight animations that is equal to the one of the sun changing the intensity, with the same duration, but with the intensity reversed because the street Lamps work only at night.

→ *(See the images in the paragraph 2.3: Lights)*

3.3 – World and objects Animations

The world animation is an important one because it is a simple means to make it seem that everything is moving and that the race goes on.
It is simply developed in the *animate()* function, where also the tween function are called, to set a rotation of the world sphere geometry along the x axis in relation with the time.

4 – User Interactions

Keyboard moves interaction

For the Keyboard interaction I use the ‘*document.addEventListeners(‘keydown’, onDocumentKeyDown, false)*’ listener and the function *onDocumentKeyDown(event)* taking the pressing keyboard event to create the movements interaction for the users.
In this way the users can:

- pressing the “A” key move the humanoid on the left;
- pressing the “D” key move the humanoid on the right;

I set a limit in the scene on the x-axis (right and left movements) from -10 to 10 that is the area that can be utilize by the user to move the humanoid and avoid the obstacles (this lasts can be also outside this limit).

keyboard camera interaction

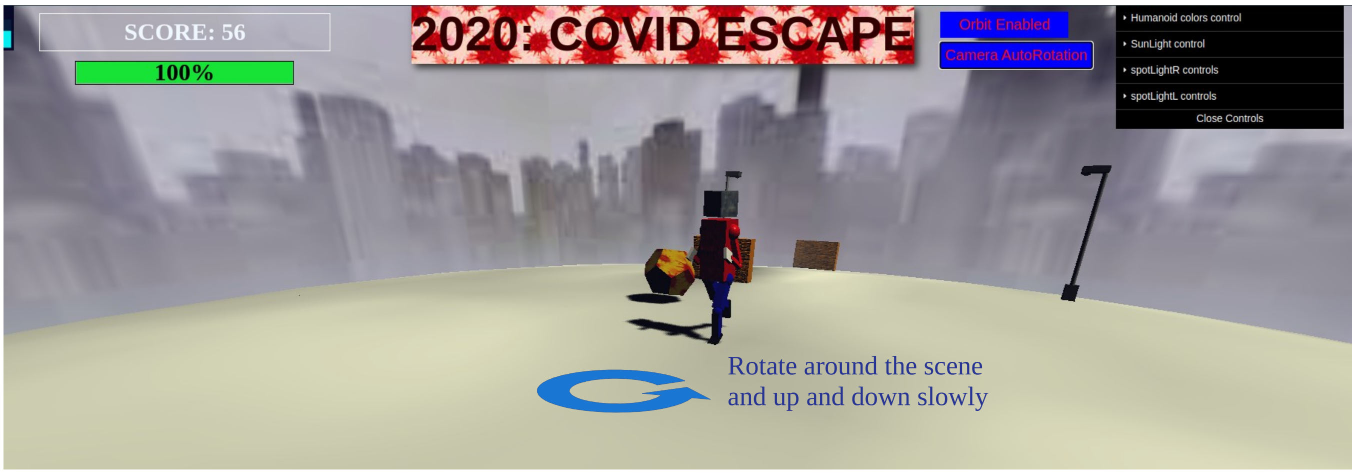
This use the same function and listener to create the interactions between the keyboard and the scene.

- (Only when the “Orbit enabled” button is disabled)
  pressing the “O” key you can change the camera position from behind the humanoid (in third person) to in front of the humanoid (in first person). Also changing in first person I attach the camera to the body so the camera position follow the displacement of the humanoid along the x-axis to have the perception to play in first person. I have disabled the running rotation of the torso because this affect this first-person camera experience;

Orbits Buttons

I have placed two different button on the canvas to permit to the user different action on the scene.

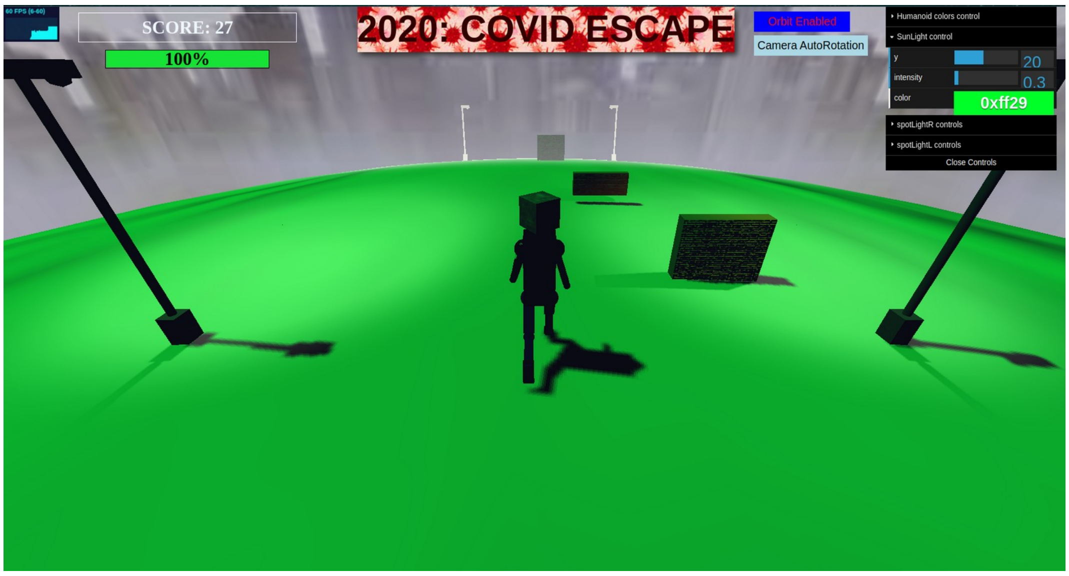
- Orbit enabled button:
  when it is enabled the user can use the orbit controls library to orbit the camera around the a target (in this case the center of the scene where the animations are performed) controlling it clicking the left key of the mouse and moving around the scene.
  When disabled you can use the keyboard camera interactions;
- Camera AutoRotation: (only when the “orbit enabled” button is enabled)
  when it is enabled this perform an orbit continuous rotation around a target (camera Center in a center position defined as an object) and moving also up and down slowly, to have a complete view around the scene.



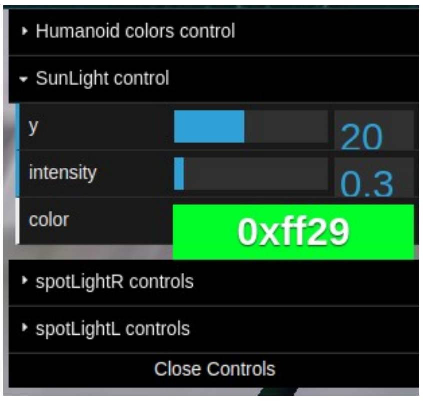
Gui properties changing

Using the *GUI* class from *dat.gui* library of ThreeJs you can actuate some changes in objects on the scene. In this game I created four different folders for the objects to which you can actuate some changes:

- SunLight control:
  you can change the y-position of the sun directional light,
  then you can change the intensity of the light and its color.
- SpotLightR and SpotLightL control:
  you can change the position on all axis (x, y, z),
  then you can change the intensity of the light, the penumbra property and its color.
- Humanoid colors control:
  you can change the color of the upper body of the humanoid, starting from the red chosen color,
  you can change the color of the lower body of the humanoid, starting from the blue chosen color.



y 20
intensity 0.3
color 0xff29



y 20
intensity 0.3
color 0xff29

5 – Game values: Score and Health

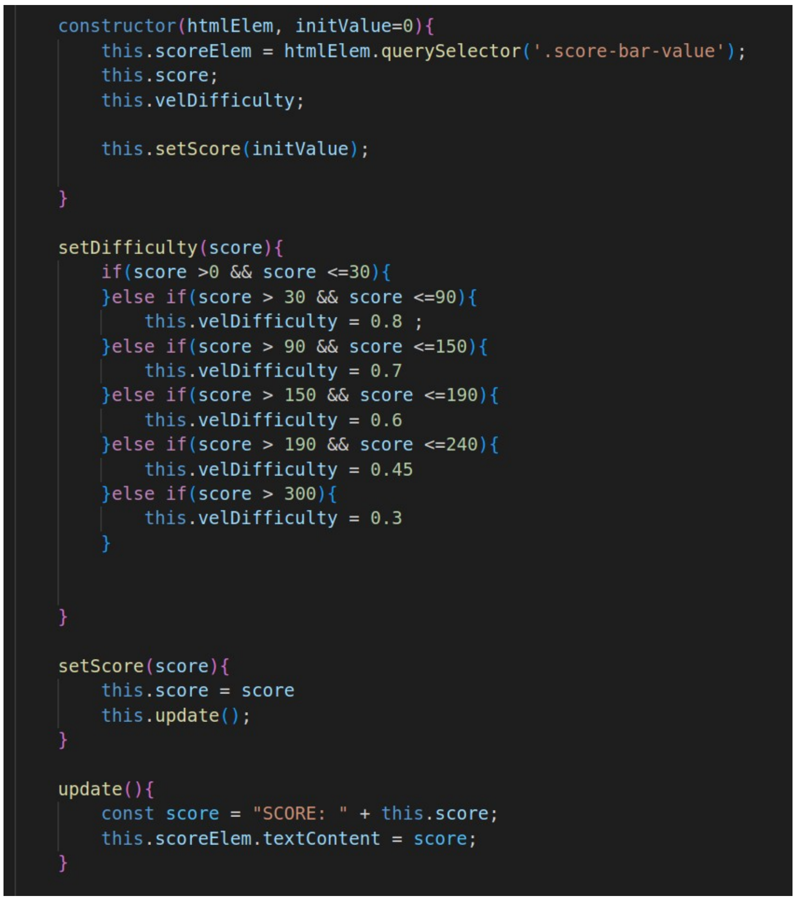
5.0 – ! (Implementation Issue): Collision detection to update the game values

I have encountered problems in implementing collision detection. This is an important part for this type of project because is the “active” one, which allows you to really make the game happen.
I have tried different solution. The first one is using the RayCaster class to design the intersection between the object in the scene (in particularly between the humanoid vs walls/virus).
I create Three.Box3 objects to build bounding boxes around the box object or a Bounding sphere around the virus objects. For the Humanoid I create a bounding box around the torso Geometry and getting the vertices of the torso with threeJs function, but for some reason it doesn’t works. I tried different changes but it doesn’t work.
Then I tried also a simple solution making a distance difference between the object position in the scene that are below a certain threshold, but also in this case I can’t get the position and compare them.

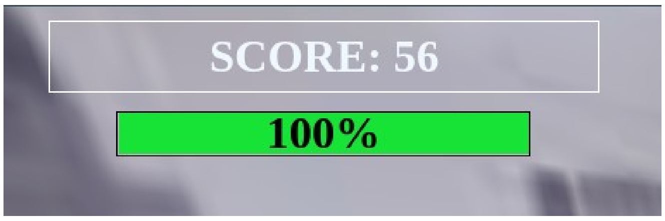
At the end I was unable to solve this problem before the delivery, despite the effort.

5.1 – Score

The score is the value that defined how far have you progressed in the game.
It is referred to the time, each second permits you to gain 1 point.
At some step then the difficulty is increased adding velocity to the game and so increasing the difficulty to avoid the objects. The “velDifficulty” start at 1.0 and slowly decrease at each score(time) step.
The javascripts functions work on a html-css object that represent the score graphically in the window.



```
constructor(htmlElem, initValue=0){
    this.scoreElem = htmlElem.querySelector('.score-bar-value');
    this.score;
    this.velDifficulty;

    this.setScore(initValue);

}

setDifficulty(score){
    if(score >0 && score <=30){
    }else if(score > 30 && score <=90){
        this.velDifficulty = 0.8 ;
    }else if(score > 90 && score <=150){
        this.velDifficulty = 0.7
    }else if(score > 150 && score <=190){
        this.velDifficulty = 0.6
    }else if(score > 190 && score <=240){
        this.velDifficulty = 0.45
    }else if(score > 300){
        this.velDifficulty = 0.3
    }


}

setScore(score){
    this.score = score
    this.update();
}

update(){
    const score = "SCORE: " + this.score;
    this.scoreElem.textContent = score;
}
```



5.2 – Health

The healthBar represent the health of the Humanoid. When it reaches the value 0 it is GameOver.
Also the healthBar is an html.css object and the javascripts function work on it, in particularly on two component: the width of the green color part of the bar and the value in the bar representing the health.

When the user hit an object the Humanoid loses health (lowering the green bar and changing the value):

- when hit a wall it loses 25% of the life,
- when hit a virus (take covid) loses the 50% of life.

This part is unused because the problem of the collision detection

6 – Libraries

The libraries used are mainly the ThreeJS library.

- All the main Three Classes (all the principal class and functions) from “three”;
- The OrbitControls from “OrbitControls.js”;
- GLTFLoader from “GLTFLoader.js”;
- GUI from “dat.gui”;
- TWEEN from “tween.js” , the library used for almost all the animation in the game.

- The normalMap-Online toolbox, to create normal map .png from .jpg images

Then the other import are from the my own file having the classes defined from me to build function and create object for the game.