Interactive Graphics - Final Project

Marco Lo Pinto, 1746911 Pasquale Silvestri, 1749274

September 8, 2021

The project developed as the final project for the Interactive Graphics class is a first person shooter game. You are a soldier in a virtual disaster simulator, your objective is to survive the attacks of rogue robots and reach the computer needed to end the simulation.
You can explore the rooms, shoot robots and exploding barrels, open doors to move between rooms, turn on/off lights, climb stairs and pick up power-ups. The menu presents two main options:

- Play: To play the main level and test all the functionalities of the game
- Dojo: To test the animations of the hierarchical model of the robot

The environment chosen to develop the project is Three.js, bundled using Webpack. Tween.js has been used to animate the objects and a custom made physics engine has been implemented to overcome some performance and functionalities limitations of mainstream engines like Cannon.js or Physijs.

Object Oriented Programming Syntax in JavaScript (ES6) has been used to achieve modularity and reuse of code through inheritance. It has been developed and tested on Google Chrome (version 90) and it has been tested on Edge (version 90) and Firefox (version 86). It is important to create a local server in order to bypass browser restrictions on CORS policies when serving locally.

The project folder structure is organized as follows:

- **dist/**: In this folder there is the compiled version of the source code used in production.
- **documentation/**: Contains this documentation
- **src/**: This directory contains the source files needed for application development (the raw, non-minified code) and the game resources like textures and models. It is structured as follows:
  - **asset/**: This folder stores all the models and textures loaded in the game.
  - **components/**: This is the heart of the app; it contains all the files used to implement the game logic.
  - **app.js**: The entry file for Webpack (i.e. the main file of the project).
  - **index.html**: The HTML page of the project.
  - **style.css**: The Cascading Style Sheets for the elements in the page.

- **package.json**: This file stores important metadata about the project and also defines functional attributes of this project that npm uses to install dependencies, run scripts, and identify the entry point to the package.
- **README.md**: Contains useful information about this project.
- **webpack.config.js**: Webpack configuration file.

The core of the game is the app.js file. It starts by loading all the needed resources asynchronously. Once all the resources are loaded, the menu is shown, the chosen level populated and the rendering loop is stared. In the rendering loop, beside the actual rendering of the scene, every object's update method is called with the correct delta time and the physics world is stepped forward.

How to Play

Control the movement of the player using the W,A,S,D and SPACEBAR keys to move respectively forward, left, backward, right and to jump. Use the mouse to look around and the left or right buttons to shoot. When close to an interactive object (Doors, lamp lights and computer) a message suggesting to press E will appear, pressing E will trigger the interaction (open/close of the door, turn on/off of the light, end the simulation). Walk over the power-up cubes to grab them. The 3 top colored bars indicate the level of health, shield and ammo remaining. Shoot the robots to kill them, shoot or touch the barrels to make them explode; beware that if you find yourself too close to an exploding barrel, you'll be hit by the explosion. In the dojo is possible to trigger the robot animation using the buttons from 1 to 5.

Custom Physics Engine

The physics engine used in the game is a custom physics engine developed by us after encountering different problems with the integration of the most common physics engines into our modern webpack workflow, and also due to the bad performance resulting from using such engines. The custom engine we developed is limited to the functionalities needed by the project, but delivers better performance and is built using a modern approach to web development, making its integration in the project much easier. The engine is composed of multiple classes handling different aspects of the physical interaction.

Movement Engine

The movement engine class handles the kinematics equations needed to simulate a natural movement. Information such as acceleration, linear velocity, gravity and position displacement per frame are kept in the object's instance of this class, and the evolution of those values, given internal and external factors, is computed step by step in the animation loop, using the delta time between frames as the time reference. Every object added to the physics world has an instance of this class.

Physics Shape

The physics shape class holds all the information about the geometry of physics object. It takes a Three.js buffer geometry, extracts its vertices, computes its local center and bounding box. For every three vertices, it computes a Face, with its center point, normal and plane equation. The class also has 4 convenience methods that transform vertices, normals, faces and center from their local coordinate system to the world coordinate system using the three.js object's transformation matrix.

Bounding Physics Shape

The bounding physics shape class extends the physics shape class to handle simpler bounding geometries for faster raycasting and preliminary collision calculations. The difference with a normal physics shape is that it can only model cubic geometries or planes and the computed faces are made of 4 vertices instead of 3.

Face

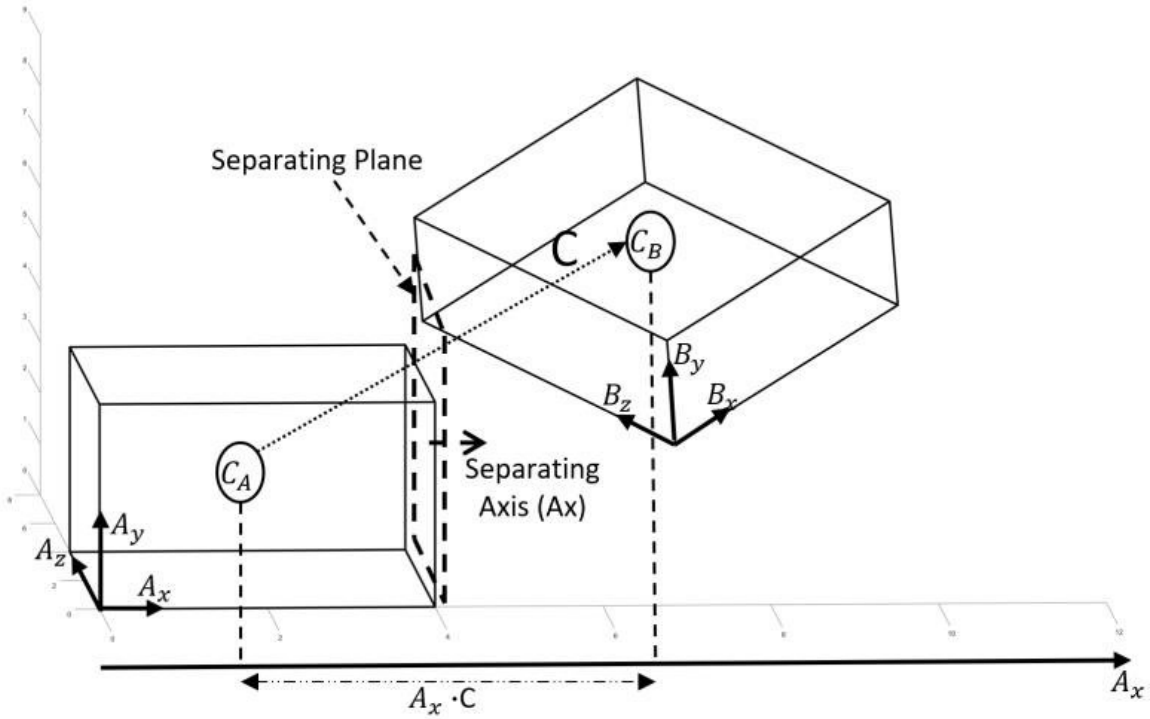
The face class hold 3 or 4 vertices and uses them to compute the center point of the face, the normal and the residing plane's equation. It also has a method to apply the world transformation matrix.

Physics Object

This class represents an object in the physics world. It holds the collision shape (an instance of Physics Shape), the collision bounding shape (an instance of Bounding Physics Shape), the graphics mesh that, beside all the webGL and Three.js graphics properties, also carries the

transformation matrix needed to move all the normals, midpoints and vertices, from their local coordinate frame of reference to world coordinates. This object also holds the custom onCollision response callback.

SAT



The SAT class encodes the separating axis theorem, a three dimensional implementation of the more general hyperplane separation theorem, a theorem about disjoint convex sets in n-dimensional Euclidean space. Given two disjoint sets, if both these sets are closed and at least one of them is compact, then there is a hyperplane in between them and possibly even two parallel hyperplanes in between them separated by a gap. An axis which is orthogonal to a separating hyperplane is a separating axis, because the orthogonal projections of the convex bodies onto the axis are disjoint.
In more formal terms:

> Let A and B be two disjoint nonempty closed convex subsets of $\mathbb{R}^n$, one of which is compact. Then there exist a nonzero vector $\vec{v} \in \mathbb{R}^n$ and two real numbers c_1 and c_2 with $c_1 < c_2$ such that:
>
> $$\langle x, \vec{v} \rangle > c_2 \quad \text{and} \quad \langle y, \vec{v} \rangle < c_1 \qquad \forall \quad x \in A \quad \text{and} \quad y \in B$$

A simple metaphor to explain such theorem is the shadows of 2 objects over 3 orthogonal surrounding walls. If all the shadows on each wall overlap, the two shapes collide. If at least on pair of shadows don't overlap, the two shapes don't collide, and the normal of the wall on which they don't overlap is the separating axis.

In the engine, using the faces of the physics shapes of each pair of objects, the vertices are projected on every normal of each face of the two objects, computing the normal's intervals. For each normal, the covering intervals are checked for overlap, if at least one pair of intervals don't overlap, no collision is returned, otherwise, the smallest overlap and relative normal are returned. The smallest overlap value is the penetration of one shape into the other, and is returned with the separating axis to correct any interpenetration happening between the objects.

Raycaster

The raycaster is the component of the physics engine used to check which objects are in a certain direction with a certain distance from an origin point. As the name implies, it consists in casting a ray from an origin point in a certain direction for a certain max travel distance. In the game is used by both the player and the robots to shoot each other.
The casting is done against the collision shapes added to the physics world and is divided into multiple phases done for each face of each object in the world. Each phase is increasingly more computational intensive but also, in each phase, most vertices are discarded. The first phase consists in checking the smallest distance of the origin point from the plane containing the current face. If this distance is greater than the specified max distance, for sure a ray intersection in whatever direction will be at least as distant, more commonly further away. This discards all the faces and objects further than the required max travel distance of the ray. The second phase computes the intersection of the line of the ray and the plane containing the face, returning, if there is an actual intersection, the intersection point and distance from the origin. The final phase is needed to check if the intersection point is within the face itself or if is outside the triangle. Phase one is just the connecting vector between the origin and a point on the plain dot multiplied by the plane normal.
In phase two the reasoning is this:
a plane having normal $\vec{n}$ and p_0 as a point on it, can be expressed as the set of points p for which:

$$(p - P_0) \bullet \vec{n}$$

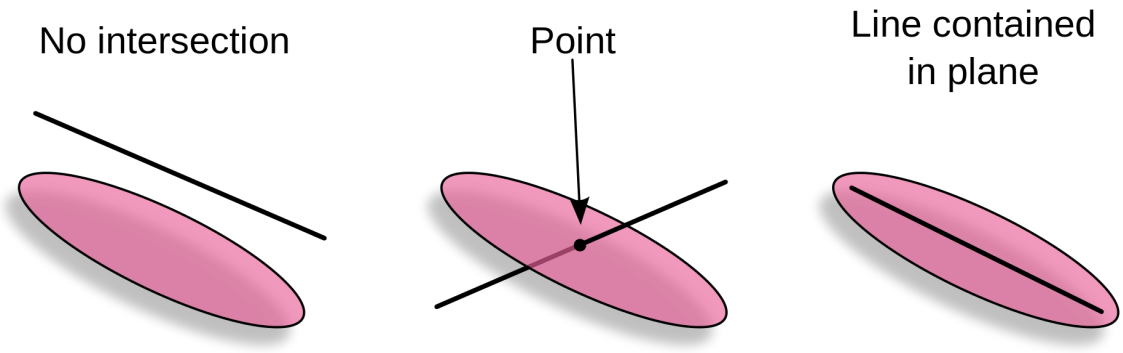
and a line with $\vec{l}$ as direction and l_0 as a point on it can be expressed as the set of points p for which:

$$p = l_0 + \vec{l}d$$

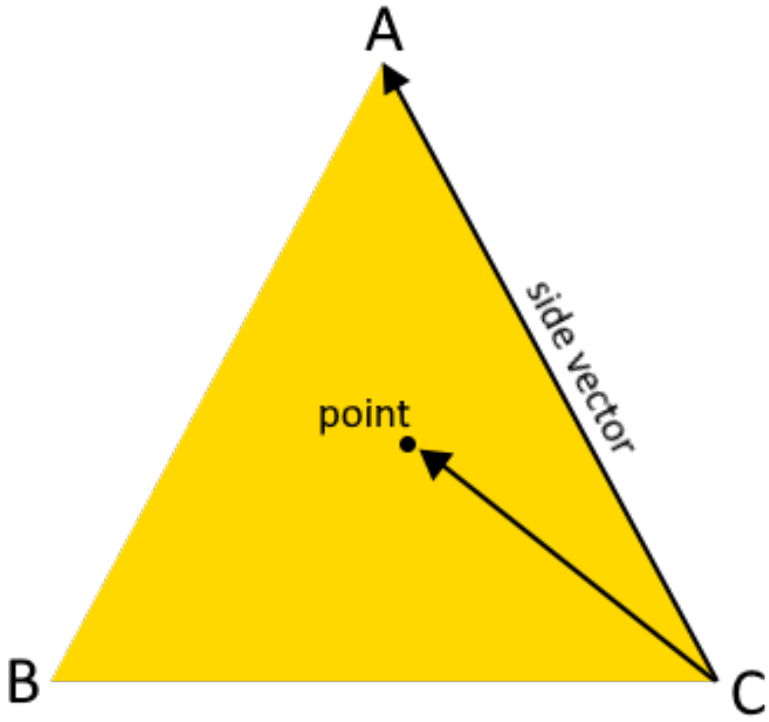
where d is a real number coinciding with the distance on the line of the point p from the point l_0 if $\vec{l}$ is a unitary vector.
Substituting the equation for the line into the equation for the plane and solving for d, gives

$$d = \frac{(p_0 - l_0) \bullet \vec{n}}{\vec{l} \cdot \vec{n}}$$

If both the $denominator = 0$ and the $numerator = 0 \implies$ the entire line is contained in the plane
If the $denominator = 0$ but the $numerator \neq 0 \implies$ there is no intersection and the line and the plane are parallel.
Otherwise, there is intersection and d is the parameter needed to find the intersection point in the previous equation, while also being the distance on the line of this intersection point from l_0.



The last phase is implemented using a very simple intuition. Given a counterclockwise set of coplanar vertices, each vertex v_i makes with its predecessor v_{i-1} a side of the face (side $\vec{s} = v_i - v_{i-1}$). Given a coplanar point p, the vector $\vec{k} = p - v_{i-1}$ is on the left of the face's side if the cross product $\vec{s} \times \vec{k}$ is negative. If this logic is repeated for each vertex composing the face and the cross product is always negative, it means that the point is inside the face.



The raycasting is done looping on all the faces of all the objects in the physics world, given an origin, a direction and a max distance travel. Can be stopped at the first object hit, whatever it is, and the intersections can be ordered in ascending o descending order. In the game ascending order is used to get the closest hit object.

Physics World

This class is the heart of the physics engine. It hold 2 lists of objects. One for dynamic objects, objects that are meant to be moving and colliding with other objects. One for static objects, object considered immovable and for which the kinematic equations are not even calculated. For every frame a call to the step() function is done, this is where the kinematic is updated for every object and every pair of objects is checked for collisions using SAT. If there is a collision between two objects, a callback on the onCollision methods of the objects is done, passing the intersection point, the penetration between the objects, the other object the owner of the method is colliding with, and the delta time for this frame. In this callback every object implements its own collision resolution strategy. Static objects like walls, floor, stairs, doors and floor lights, simply repel the other colliding object. More complex objects use this callback to do more interesting things, for example, the gas barrels explode and inflict damage to the surrounding objects, the power-ups increase health, shield or ammo of the player, and so on.

Player

Player

The Player class represents the user in the game, it handles all of its inputs and it's an Object3D, parent of the camera and of a rifle model. It contains all of the player related information, which are:

- height
- center position
- feet position
- current health
- current shield
- current ammo
- max health
- max shield
- max ammo
- running speed
- jumping speed

It also holds a reference to the camera, the movement engine for the physics simulation, the collision geometry (a cylinder), the hud, the rifle model, its animation and shooting particle effect and an input control system that, by being an event dispatcher, replays parsed inputs to the rest of the game.
The camera is placed at the specified height from the feet and is rotated by the control system,

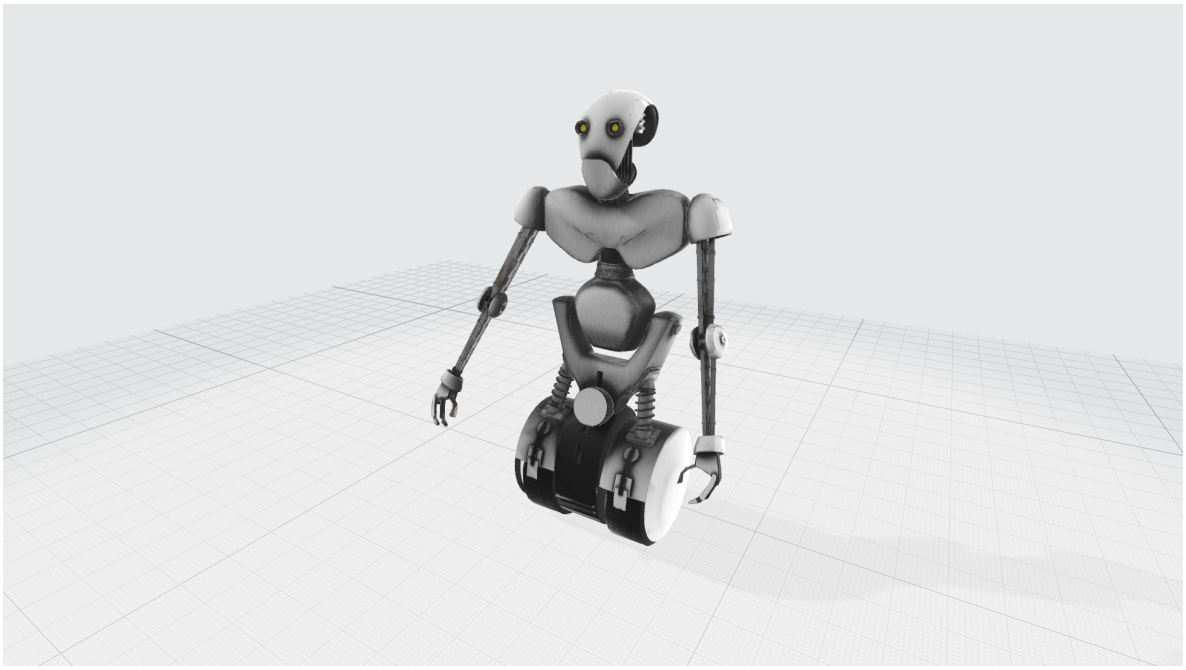
following the mouse movement.
The rifle is loaded asynchronously and added as a child of camera, so that is always visible in the foreground, following the head movement. The shooting animation is triggered by the input control when either the left or right mouse buttons are pressed and the shoot method is called. Is a recoil animation of the rifle. The animation is followed by a small explosion at the tip of the rifle, made using the custom particle system.
The HUD is used to show the current health, shield and ammo remaining, with 3 colored bars at the top left of the screen. A callback to the three bars' setPercentage method is set so that whenever the player's stats change, also the bars change.
The control system is an event dispatcher instance with specific methods bound to the mouse click, mouse movement and key press and release. When the left mouse button is clicked for the first time a request for the pointer lock is initiated. The Pointer Lock API provides an input methods based on the raw movement of the mouse over time (delta displacement of the pointer coordinates on the screen), locks the target of mouse events to a single element (in this case the webGL canvas), removes the cursor from view and eliminates limits on how far mouse movement can go in a single direction. When the pointer lock is acquired, a lock event is dispatched, the HUD listens for it and makes a cross-hair appear where the rifle is aiming. When the "esc" button is pressed, the pointer is unlocked and the unlock event is dispatched by the player. The Pause menu listens for it and pauses the render loop while showing a context menu. The W,A,S,D and SPACEBAR keydown and keyup events are listened to handle a set of flags indicating the requested type of movement for the player, like moving forward, moving sideways or jumping. Those flags are then interpreted once per frame by the update method. When the mouse is moved and the pointer is in locked state, the camera is rotated following the mouse movement.
When pressing a mouse button, the shoot method is called. In this method a ray is casted from the tip of the rifle, directed where the player is looking at. The closest object that is encountered, if is a "shootable" object (robots and barrels), gets shoot. The reaction to the hit is handled by the object being hit. Instead, when is the payer that gets hit by a robot shooting, or a barrel explodes too close to the player, the dealDamage method is called, computing how much of the shield and health has to be depleted. If the health reaches zero, a death animation of the player falling on the ground is played, and the death menu is shown.
Once per frame the update method gets called, the movement flags set by the keyboard inputs are checked and, using the transformation matrix of the camera, the movement velocity vector is computed and passed to the movement engine. The movement engine will calculate the displacement, that gets subsequently added to the current player's position. Also the update method is the place where the death animation, rifle animation and particle effect are updated.

HUD

The HUD (Heads Up Display) is the superimposed interface that show vital information and suggestions to the player. It can be used to create as many progress bars as needed at the top left of the screen. Is also handles the cross-hair at the center of the screen, and shows or hides it depending on the pointer lock state returned by the player's control system. The HUD is also responsible of the caption, the white text at the bottom of the screen, used by interactive objects to describe the possible action. The caption has an owner set whenever the text changes, and, unless the owner changes, only the owner can decide when to hide the caption.

Enemies

Robot



Robot is the enemy in the game and its main purpose is to detect and kill the player.
The class has various parameters that are created and initialized when the class is instantiated, the most important and relevant are:

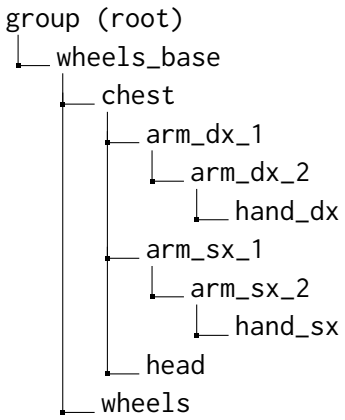
- **group**: It contains the loaded .glb hierarchical model.
- **group.geometry**: The geometry that will be used for the PhysicsWorld class; in this case it's a cylinder.
- **group.movementEngine**: An instance of the MovementEngine class for the robot.
- **playerToFollow**: The entity that the robot needs to follow and attack: if null, it doesn't follow anyone.
- **_tweenAnimations**: An instance of the Tween.js group: this is necessary if the project has multiple animations of different objects. This instance coordinates all the animations of the robot. There are six different animations, which are:
  - **alert**: Invoked when the entity that the robot needs to follow enters in the maximum eye radius distance.
  - **shoot pose**: Invoked when the entity is near enough to aim and shoot to the entity to follow (the shooting distance).
  - **shooting**: Invoked when the robot shoots to the entity.
  - **shoot pose reverse**: Invoked when the robot is not near enough to shoot the player.

  - **death**: Invoked when the robot dies.
  - **idle**: Invoked when the robot can't see the entity: it follows a random path by rotating and walking.

  To simplify the creation of the animations, the class has a method called createGeneralAnimation() which wraps the logic of Tween.js so to create all the necessary animations and making them able to interpolate to each other smoothly.
  To start a new animation, it is called the method startAnimation(), which stops all the current animation in the robot and starts the new one passed as parameter.

- **_shootExplosion**: ParticleSystem instance for dealing with the shooting explosion in the right hand of the robot.
- **_explosionParticles**: ParticleSystem instance for dealing with the explosion effect of the robot when dies.
- **_idleMovement**: Useful informations for the idle movement animation.
- **isFollowing**: Boolean value to check if the robot is following the entity.
- **isInShooting**: Boolean value to check if the robot is preparing to shoot the entity.
- **isShootingProjectiles**: Boolean value to check if the robot is shooting the entity.
- **isDead**: Boolean value to check if the robot is dead.
- **shootingCooldownMax**: The interval in seconds between one shot and another.
- **shootingCooldown**: The current interval in seconds between one shot and another.
- **_shootingProbability**: The probability (between 0 and 1) that the shot was successful (and not a miss).
- **angryDurationMax**: The duration in seconds that the robot is in angry state.
- **angryDuration**: When the robot is shot, angryDuration becomes equal to angryDurationMax. While the value is greater than zero, the robot can follow the entity for the amount of time equal to angryDurationMax seconds, no matter of the distance between them. When the value is equal or less than zero, the robot exits the angry state.
- **health**: The health of the robot.
- **velocity**: The speed of the robot.
- **rotationVelocity**: The rotation speed of the robot.
- **shootingDistanceMax**: Maximum distance that the robot can shoot the entity.
- **eyeRadiusDistanceMin**: Maximum distance that the robot can see the entity (if not in angry state and is not following the player).
- **eyeRadiusDistanceMax**: Maximum distance that the robot can see the entity (if not in angry state and is already following the player).

The group is organised as follows:



The logic of the robot is mainly in the update() method, which is called at every step in the LevelCreator class: it updates the animations, the particle effects of the robot (shooting and exploding) and its state in relation with the entity to follow.

When the robot is near the player (i.e. when the radial distance between the robot and the entity to follow is equal or less than shootingDistanceMax), the shoot method is called. When executed, if the shooting cooldown is equal or less than zero, the shooting animation is called and a ray is casted from the tip of the right hand's robot, directed towards the cited entity. The closest object that is encountered, if is a "shootable" object (player and barrels but not robots, there is no friendly fire between them), gets shoot. The reaction to the hit is handled by the object being hit. Instead, when is the robot that gets hit by a player shooting, or a barrel explodes too close the robot, the dealDamage method is called, computing how much of the health has to be depleted. If the health hits zero, the death animation of the robot falling on the ground is played, and then the robot explodes, removing itself from the scene.

Test Robot

Test Robot is a version of the Robot class that is used to create, test and debug the Tween.js animations that will be used in the final version of Robot. It can be found on the Dojo level to check more clearly the final animations achieved by the Robot.

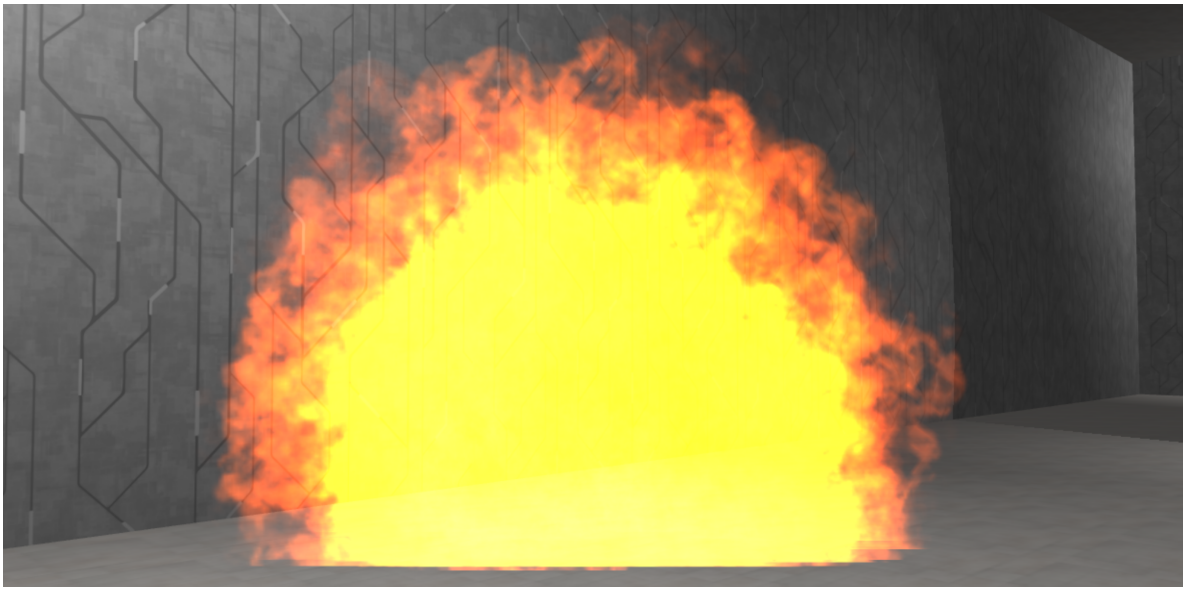
By pressing one of these keys on the keyboard, a respective animation will play:

- **1**: Alert
- **2**: Shoot pose
- **3**: Shooting
- **4**: Shoot pose (reverse)
- **5**: Death

All animations can be interpolated with each other smoothly in mid execution.

Environment

Particle System



The ParticleSystem class extends the Three.js class Points to create powerful particle effects using custom vertex and fragment shaders. To create an effect, it is necessary to pass some parameters when instantiating the class, which are:

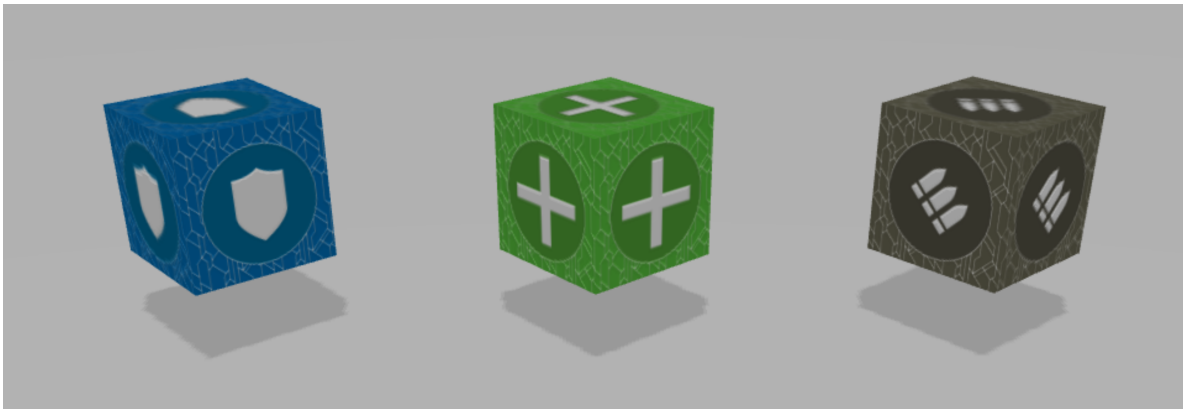
- **scene**: The scene to add the particle effect.
- **camera**: The camera object: it is necessary for sorting the particles so to make them appear in order of distance.
- **duration** (optional): The duration in seconds of the animation: if null, the duration is infinite.
- **particleImage** (optional): A texture for the particle effect: if null, the default one is used.
- **onFinish** (optional): A callback function invoked when the effect is finished.

There are various methods that can be used to obtain different visual effects, which are:

- **setGeneralPosition(x,y,z)**: Sets the general position of the particle effect.
- **setGeneralRadius(x,y,z)**: Sets the distance range in which a particle can be created relative to the general position.
- **setParticleSize(particleSize)**: Sets the possible size of the particle (it varys a little around this value).
- **setGeneralLife(generalLife)**: Sets the possible life of the particle (it varys a little around this value).
- **setGeneralVelocity(x,y,z)**: Sets the direction vector in which the particles needs to move (useful to create comets or blasts).

- **setNumberOfParticles(numberOfParticles)**: Sets the indicative number of particles that needs to be present each frame. Default is 75.

Pickups



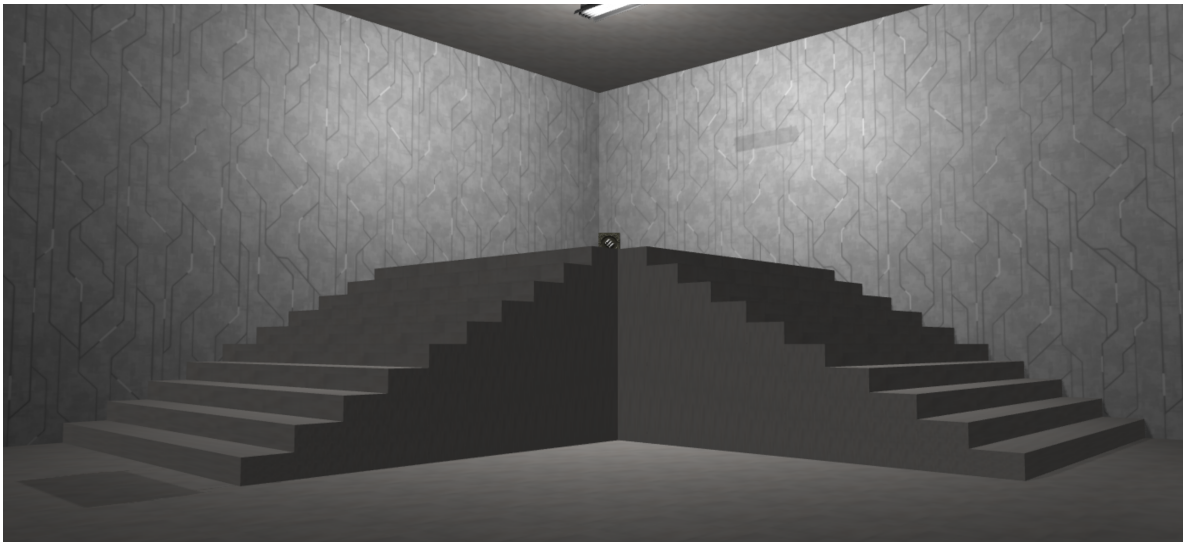
Pickup is a class that extends the Three.js Mesh class. In the game it appears as a floating cube with texture that visually describes its functions and it has a rotation animation.
There are three types of pickups, each of them extends the Pickup class, does a specific function once picked-up by the player and has a particular texture: PickupHealth, PickupAmmo and PickupShield. The former increments the life, the second increments the number of ammunition of the weapon and the latter increments the shield. All of them have also a bump map.

Gas Cylinder

The class GasCylinder extends the Three.js Mesh class. Its purpose is to explode and do radial damage to objects that have life (i.e. objects that have a dealDamage() method). It has two textures: one for the top and the bottom of the cylinder and one for the lateral part; all of the three surfaces have a bump map so to show some irregularities.
When the cylinder is touched by a dynamic object or gets shot by the player or a robot, it calls

the explode() method that removes from the scene the barrel, creates a ParticleSystem instance which represents an explosion and calls the dealDamage() of each object that was near the cylinder.

Stair



Staircase is a class that extends the Three.js Mesh class. It is useful for dynamic objects (like Player or Robot) so that they can move from one height to another.
Unlike other objects in the world, its mesh geometry is not the same as its collision geometry, which is a custom created Three.js' BufferGeometry named HalfCubeGeometry (it is located in the Tools folder, in CustomGeometries.js). This was done for two reasons: to simplify the collision logic and to allow dynamic objects to climb without hitting the steps.

Door

Door is a class that adds an interactive door to the world. Is a .glb hierarchical model that organised as follows:

```
group (root)
├── door_r
└── door_l
```

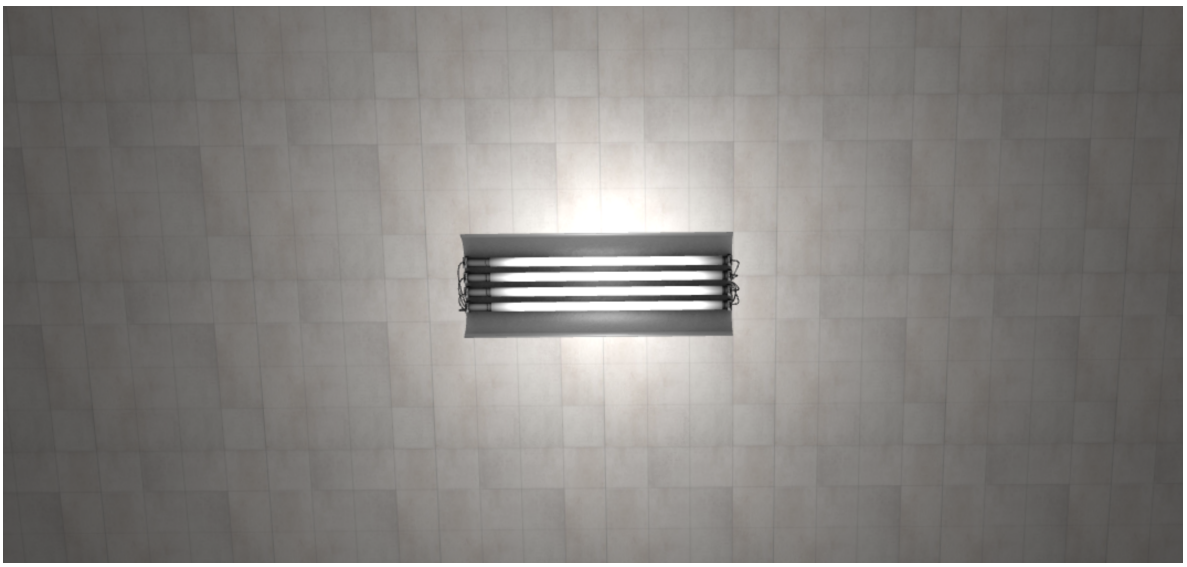
The object has two animations to open and close the door and to prevent dynamic objects to pierce through it when is closed, the two children of the model have a collision geometry attached to them and used in the PhysicsWorld.

If the player is near the object, by pressing the ’E’ key on the keyboard the door will open or close.
The Door model was created by us using Blender.

Floor Light

FloorLight is a class that adds to the world a light source to the ground. It is a complex .glb model that has an emissive map, which is useful to visually understand when the light is on and off. This object instantiates also a Three.js’ Spotlight class, which is a light that gets emitted from a single point in one direction, along a cone that increases in size the further from the light it gets.
If the player is near the object, by pressing the ’E’ key on the keyboard the light and the emissive map intensity is changed so to produce as result the activation or deactivation of the light source.

Top Light

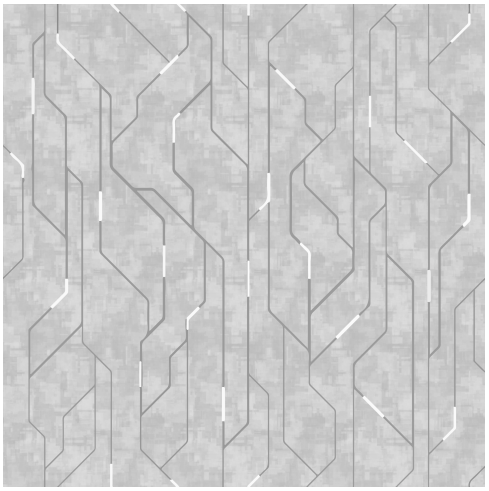


TopLight is a class that adds to the world a light source to the ceiling. It is a complex .glb model that has a normal map, a metalness map and an emissive map; the latter is useful to visually understand that the light is on. This object instantiates also a Three.js' PointLight class, which is a light that gets emitted from a single point in all directions.

Control Panel

ControlPanel is the class handling the model and interaction of the computer, placed at the end of the game, needed to shutdown the simulation, that is the objective of the game. It is a complex .glb model that the player can interact with by pressing the 'E' key on the keyboard, resulting in winning the game.

Wall



Wall extends the Three.js Mesh class. In the game it is used to prevent dynamic objects to access certain areas and to divide the level in rooms.
It has different textures that can be used to change the level design (see the difference between the classic level and the dojo).

Floor

Floor extends the Three.js Mesh class. In the game it is used as both floor and ceiling, depending if it is positioned above or below player.
Its main purpose is to prevent the objects that are affected by gravity to fall indefinitely.

Tools

Level Creator

LevelCreator one of the main classes of the project that aims to simplify the creation of the levels. It has various methods to create each possible object that can be seen in the world. It has also methods to create the player, the point light that acts as a sun, the ambient light and the fog. It has also a function called clearLevel() to remove from the scene and from the PhysicsWorld all the objects.

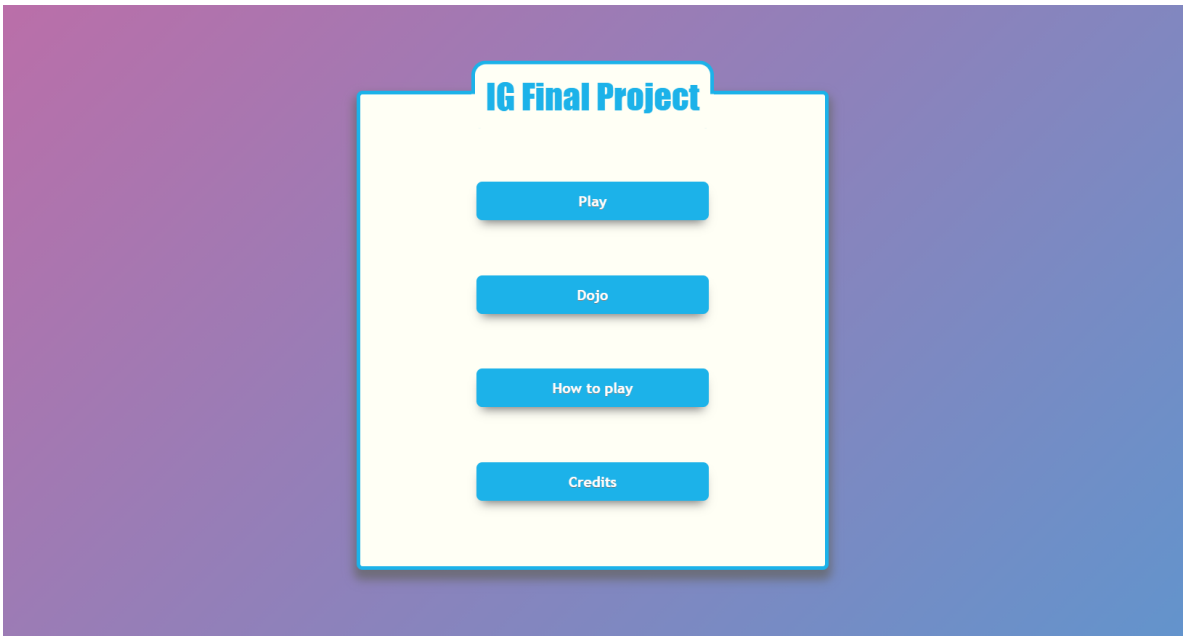
General Loading Manager

The General Loading Manager extends the Three.js' Loading Manager to increase its functionalities. It can handle multiple callback for a single resource or multiple resources compounded. It delivers a simpler, integrated way to load resources and to know when all of the asynchronous loadings are finished. In the project it is used to load all the textures and models before actually launching the game. When the game is started for the first time and the resources are being downloaded and parsed, a loading animation is shown, waiting for this class to notify the end of the loading process.

Custom Geometries

CustomGeometries stores various custom-made Three.js' BufferGeometries. It is mainly used to create simple geometries for the collision detection for the PhysicsWorld.

Menus



The menus are used to navigate in the game levels. All the menus in the game use the same structure, while being customized for their specific use. In general all the menus will contain an "How To Play" button, whit a simple explanation of the game's buttons and dynamics, a "Credits" button, and a Main Menu button to move back to the first menu.

Main menu

The main menu, is the first menu encountered. It allows to choose between the fully fledged game level, or the dojo.

Pause menu

The pause menu appears when the user exits the pointer lock state, generally by clicking on the "esc" key. When this menu is showing the game loop doesn't progress and the user can choose to go back to the game, go to the main menu or read again how to play or the credits.

Death menu

This menu appears when you succumb to the robots and fall to the ground. It lets you go back to the main menu and try again.

Win menu

This menu effectively states the winning of the game. It appears when the player reaches the end goal of the game, the computer to exit the simulation.