Interactive Graphics – Final project

“Minigame.io”

Emanuele Musumeci, 1653885

The presented project implements a simple videogame featuring a series of “mini-games” (easy levels with a specific objective), designed modularly to allow an easy addition of new levels. It is implemented using the *Babylon.js* framework and *Cannon.js* as a physics engine (already integrated in Babylon.js). Currently two mini-games are available: one featuring a shooting range with a pistol, that requires shooting targets, the other one featuring a basketball and a basket, that requires scoring some free-throws. After a first round where all mini-games are played in sequence, mini-games will be chosen randomly and randomized (each mini-game provides its own randomization features). The player is implemented as a hierarchical model of a “drawing mannequin” with animations for walking (forward, left, right, back), jumping, falling and being idle. Possible actions triggered by user input are: moving the player, picking up/dropping objects (not all objects are pickable), some of which can shoot projectiles (currently only a pistol can do so but the game provides a modular way of creating a “shooter” object). A simple but complete “menu-based” GUI is provided using the *Babylon,js* GUI libraries.

1. Framework overview

The project builds on the Babylon.js libraries and is aimed at preserving their modularity by creating an architecture based on several modules (using the primitives from Babylon.js):

- A **Game Engine**, acting as a controller for the basic “flow” of the game. It provides functions to also create/remove and manage cameras, it manages user inputs and provides an easy way to add/remove shadows to objects. It also holds the reference of the **current scene** being rendered.
- Babylon.js provides an integration with various Physics engines, among which Cannon.js was chosen for its richness of features. A simple interface to the Physics Engine was implemented to provide easy access to physical features, providing also helpers for the creation of colliders and Physical impostors (physical bodies).
- A **Level Manager** that creates and manages levels. The game is “level” based, each level being a different mini-game. This module is tasked with creating and switching levels. In order to add a bit of variation to the game, after loading all basic levels, this module also offers a level randomization feature (levels can be designed to use this feature), that randomizes both the order of levels and the randomizable features of each level (decided at design time).
- A **Prop manager**, to create/remove meshes and “props” (in-game objects with physical features, such as collisions, mass etc.). This module also provides functions to “highlight” a prop (using a Highlight layer from Babylon.js to render a glow around the object when it’s possible to pick it up) and to pick up the object, namely attaching it to the camera as if the player was holding it.
- Two modules dedicated to the player: the first one being the **Player** module, that provides primitives to spawn/despawn the player, trigger animations, control its movements by listening to user inputs; the second one containing the player **animations** and some functions to perform transitions between the current player geometry and the first keyframe of the next animation (to smooth out as much as possible transitions between player movement animations).
- One module with helper functions to define materials, featuring both Standard and PBR (Physics-based rendering) materials.
- A **GUI** management module, that builds on Babylon.js GUI library, to provide support for game menus.
- A **Settings** module to hold some general flags and constants, plus the user input mappings.
- A **Utils** module containing some basic utility functions.

2. User manual

2.1 Main menu

Contains the Start Game button that starts the first minigame; the Options button that opens the Options page; the Tutorial button that opens the tutorial page.

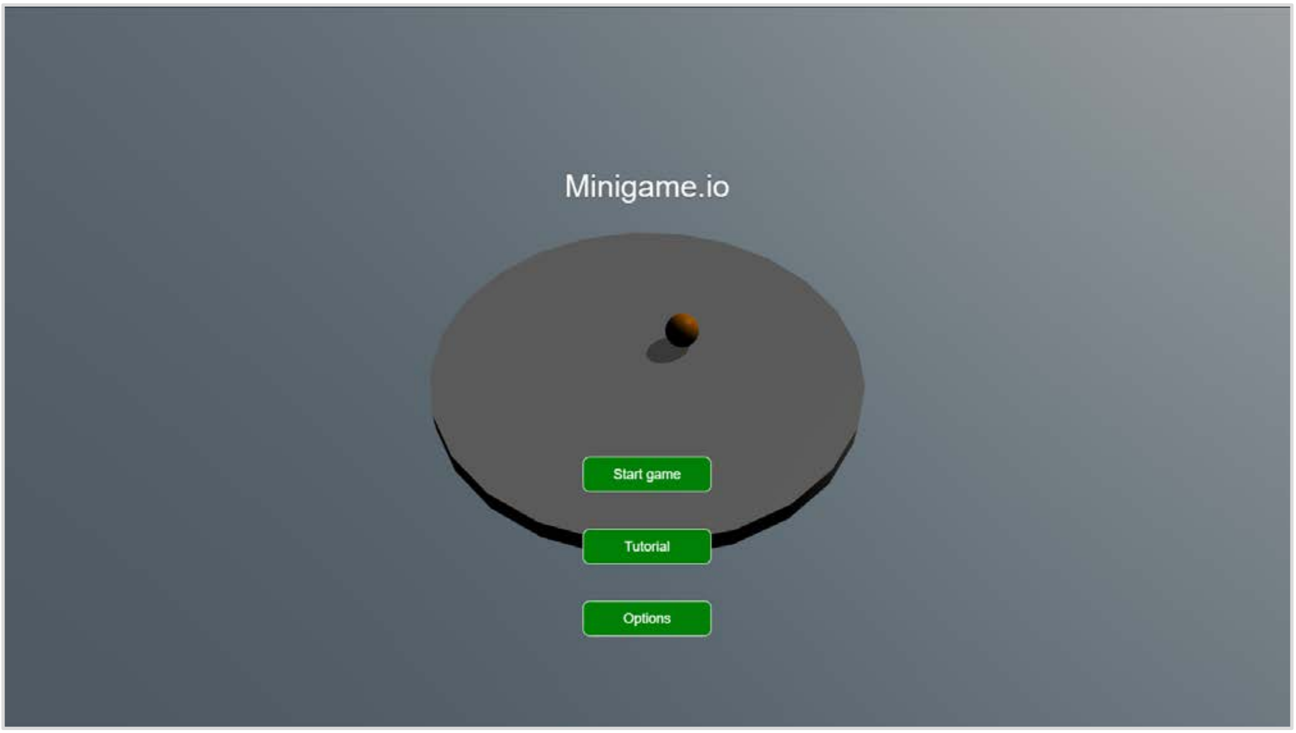
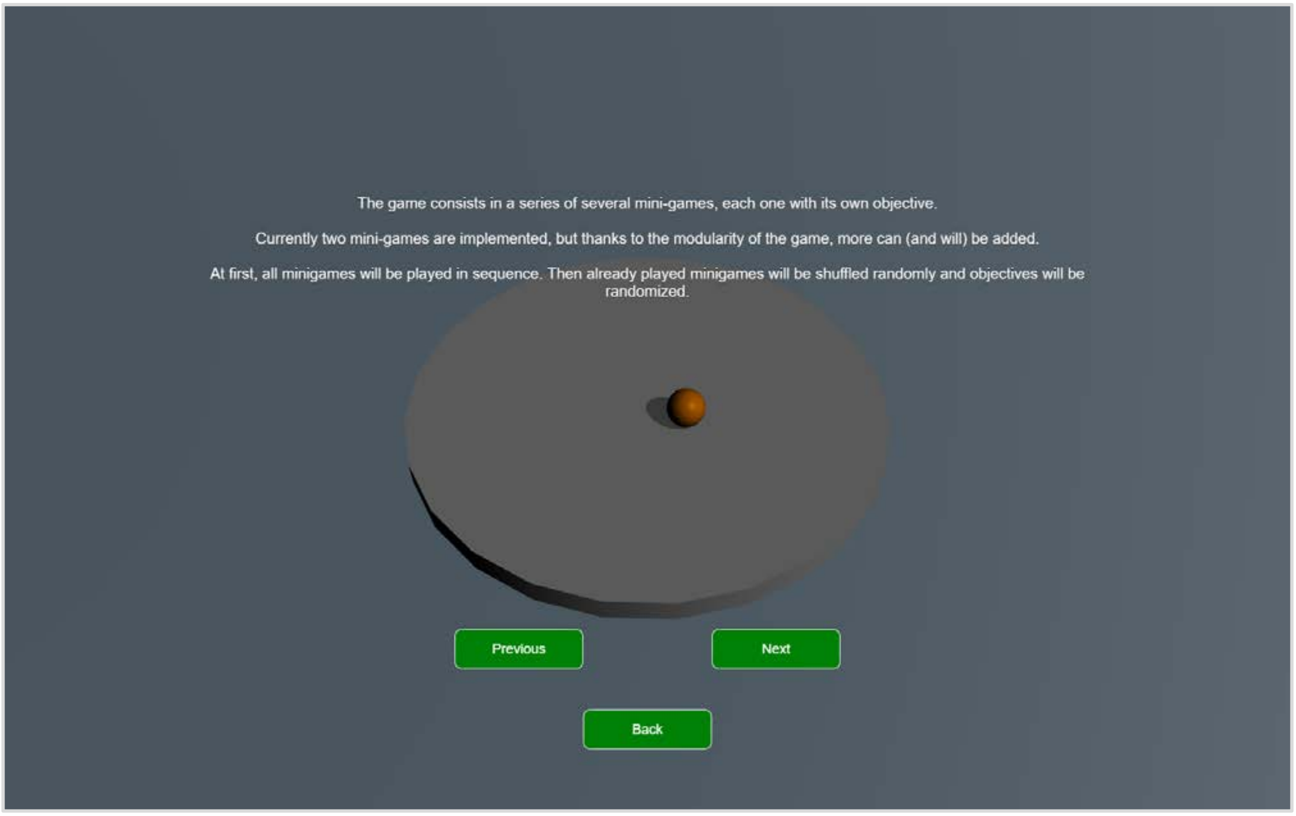


Figure 1 - Main menu

2.2 Tutorial page

Contains a text box with tutorial text, two buttons to navigate the tutorial text (Next, Previous) and a button to go back to the Main menu



The game consists in a series of several mini-games, each one with its own objective.

Currently two mini-games are implemented, but thanks to the modularity of the game, more can (and will) be added.

At first, all minigames will be played in sequence. Then already played minigames will be shuffled randomly and objectives will be randomized.

Figure 2 - Tutorial menu

2.3 Options page

Contains two checkboxes: one to show a graphical render of Physics Impostors (physical colliders) and one to show a graphical render of Colliders (boxes that contain complex meshes, in order to simplify collisions).

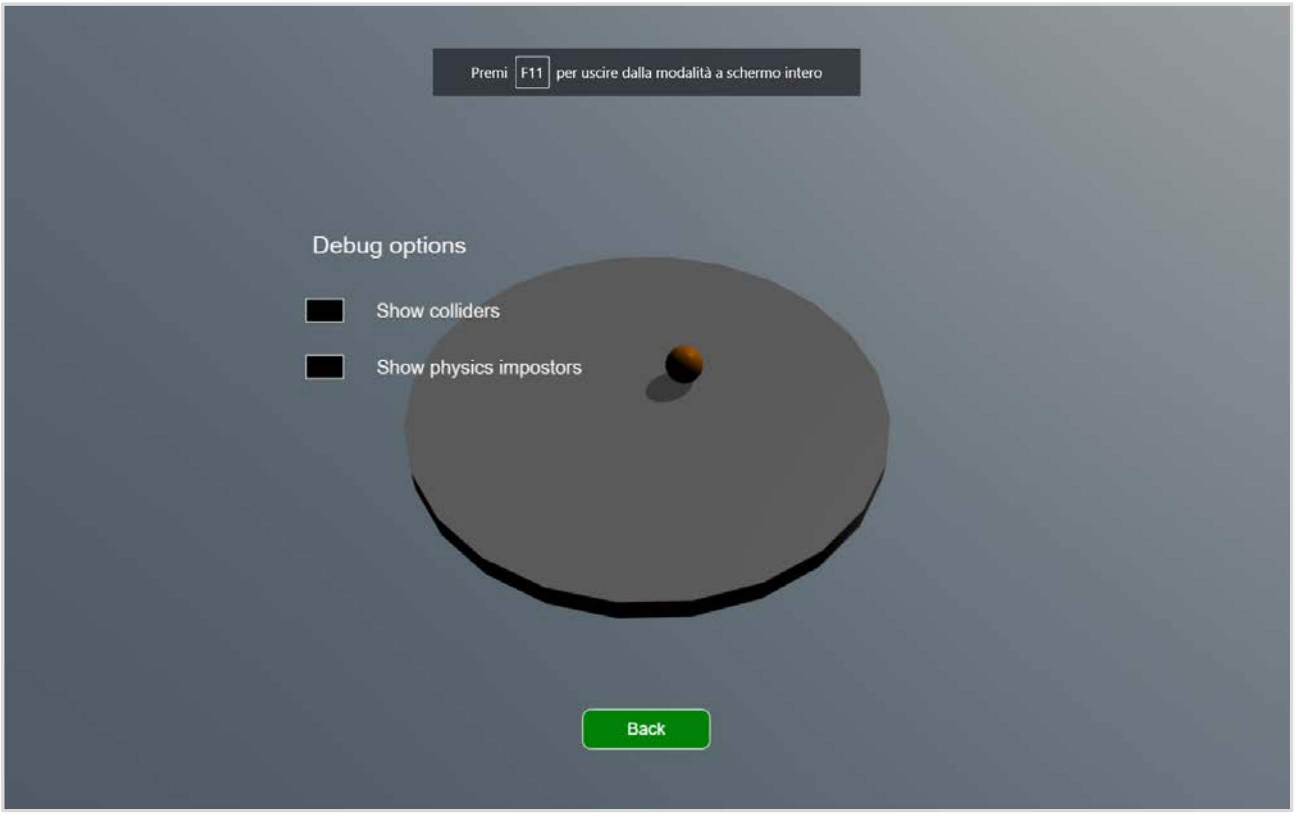


Figure 3 - Options menu

2.4 Game HUD

The game HUD is very simple and may vary between minigames. It features a score counter on the top of the screen and some tutorial text on how to perform critical actions (that disappears once an object is picked up).

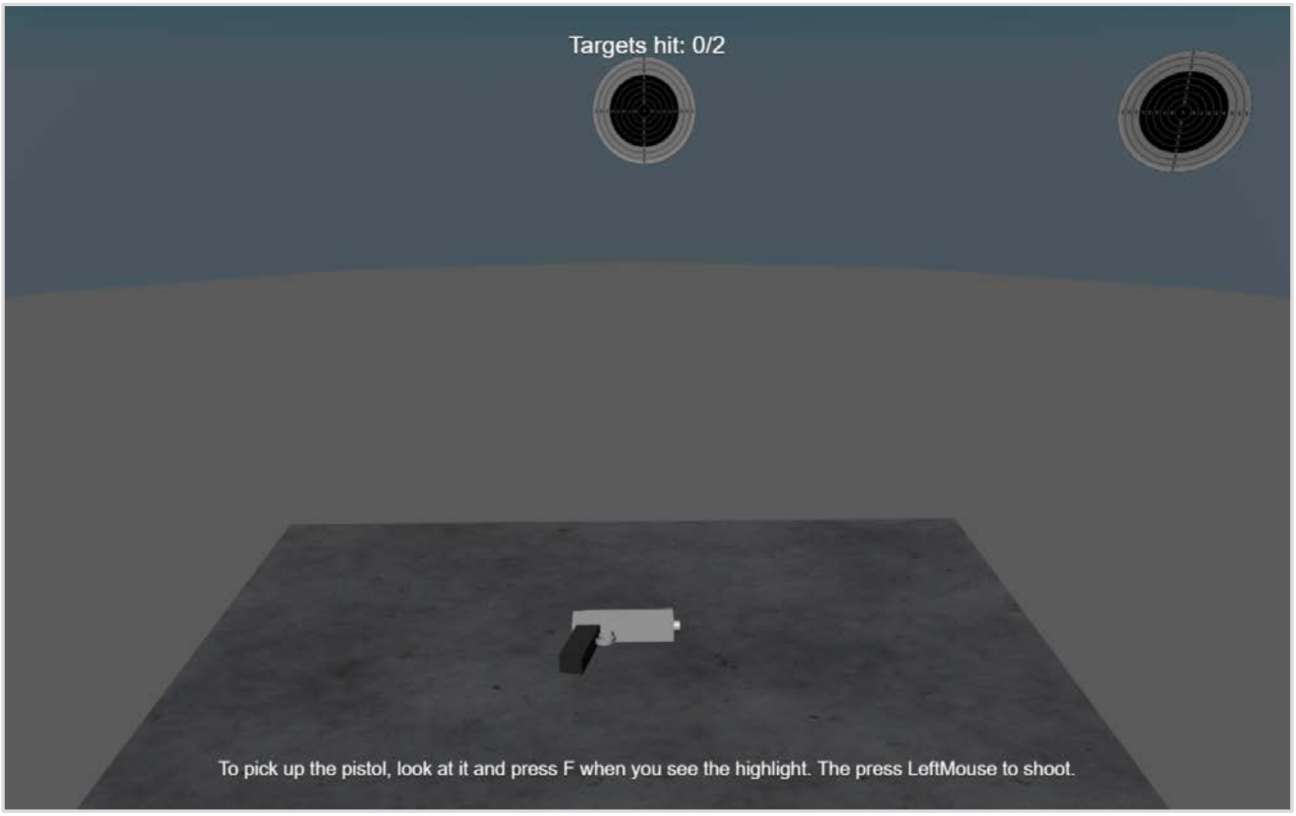

Targets hit: 0/2
To pick up the pistol, look at it and press F when you see the highlight. The press LeftMouse to shoot.

Figure 4 - Game HUD

2.5 Pause menu

By pressing M during the game, it is possible to pause it and open the pause menu (the game will keep running in the background). This menu contains a button to resume the game (this can also be done by pressing M), one to show the tutorial page and one other to skip the current level.

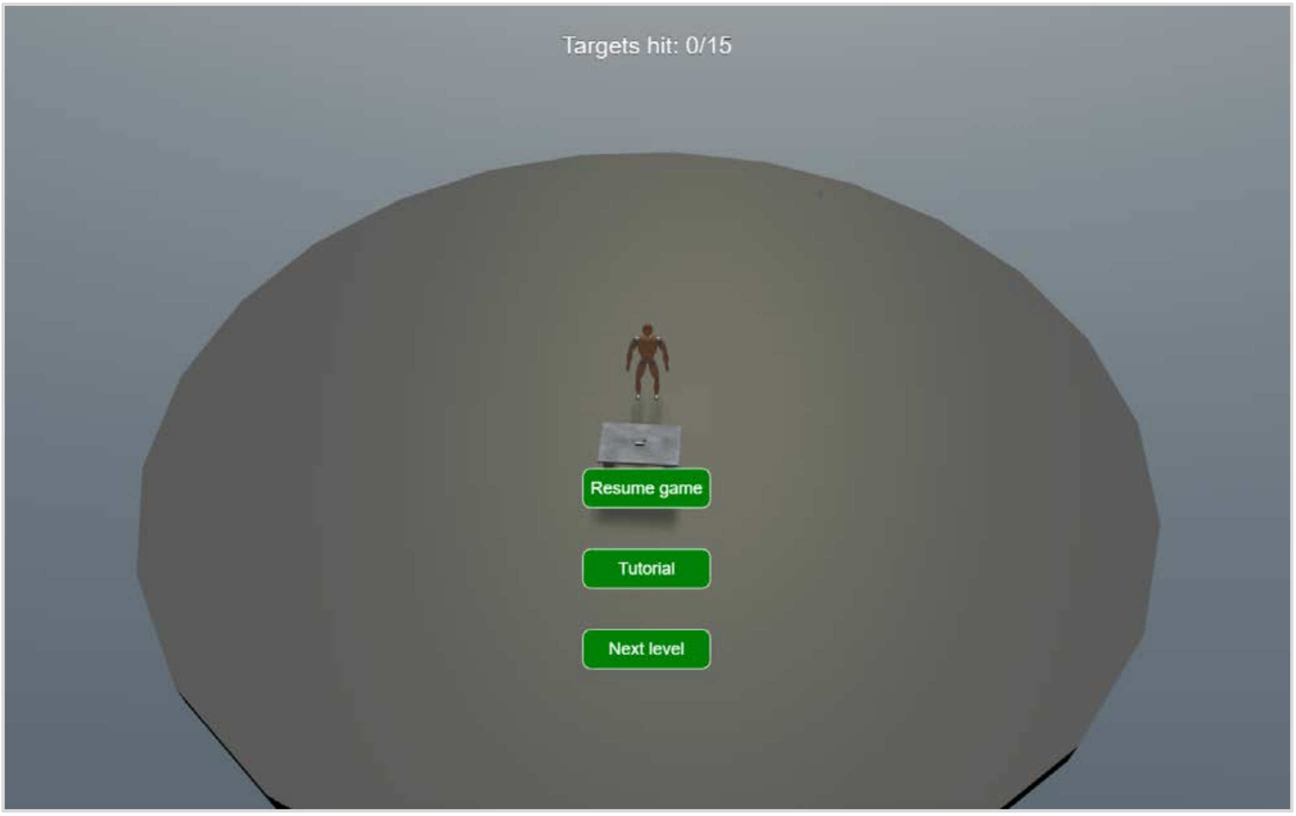


Figure 5 - Pause menu

2.6 Level change screen

When a level ends, a countdown to the following level is shown and then the next level is loaded.

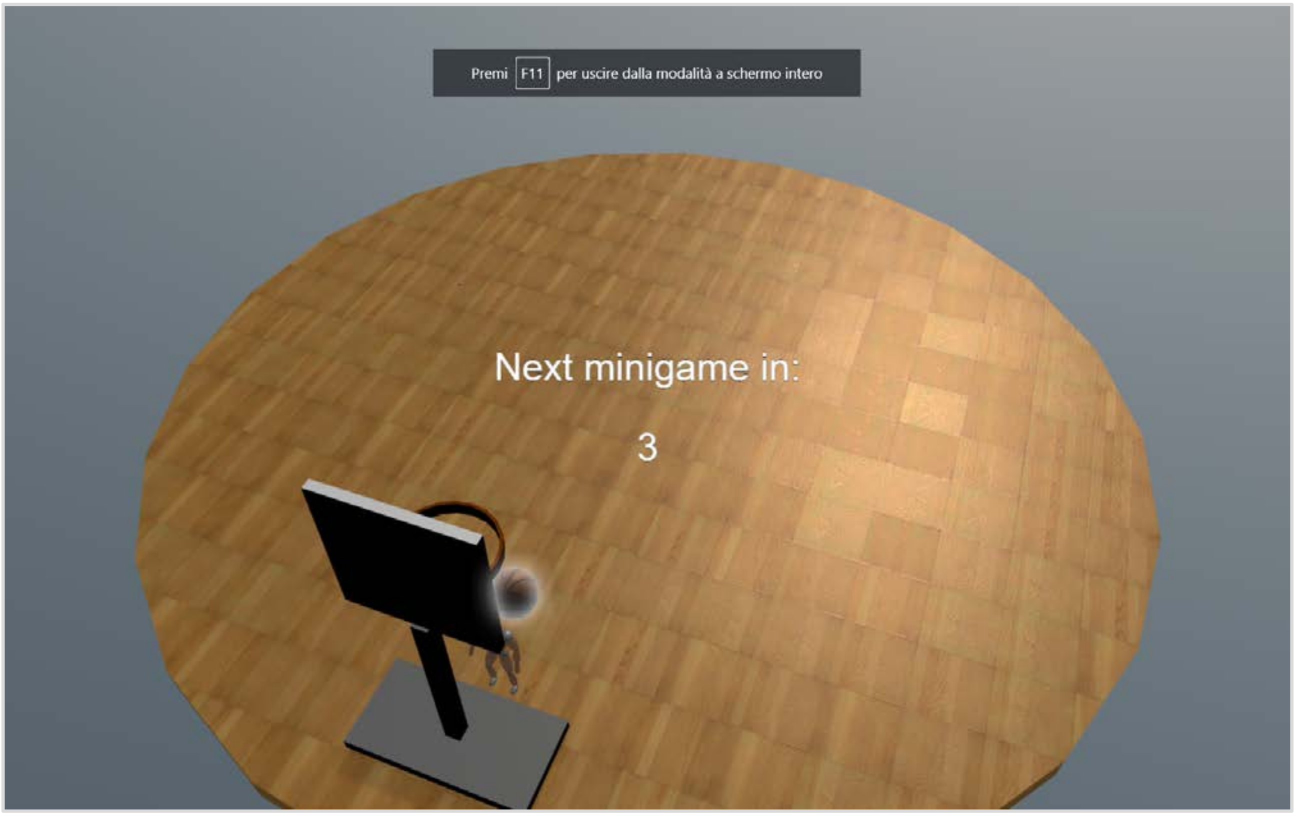


Figure 6 - Level change screen

2.7 Game commands

The player can be moved by pressing the usual keys:

- **W** (forward), **A** (left), **S** (backward), **D** (right)
- **SPACEBAR** (jump)
- **R** to switch between **First/Third person**
- **T** to view the player from the side while in third person (used to view the animations from a different angle)

The currently implemented minigames use an object pick up mechanism:

- Press **F** to pick an object if it gets highlighted when looked at
- When an object is picked up **F** can be pressed again to drop it. If it is a “shooter” object (like the pistol), projectiles (that depend on the object) can be shot using the Left Mouse Button (**LMB**). If it is not a shooter, it can be thrown by keeping LMB pressed in order to charge the launch (a progress bar will appear when doing so, showing the currently charged launch force) and then releasing it once the **LMB** is released. The launch force will be proportional to the charge time.

Finally, to open/close the pause menu while the game is running, press **M**.

NOTICE: Sounds might not be enabled by default. To enable them, click on the speaker icon in the upper left corner of the canvas.

2.8 Minigames

Currently two minigames are implemented:

- **Shooting range**: a pistol can be picked up in order to shoot targets in the level. In its **randomized** version, targets vary in texture, size, distance, position and number.
- **Basketball**: a basketball can be picked up and thrown at a basket in order to score free-throws. In its **randomized** version, baskets will be floating in the air and their position and number will vary.

3. Implementation

Follows an overview of each main component of this project, with an in-depth explanation of the most critical parts. Most of the code is organized in classes with static fields and methods: this is supposed to easily enforce a *singleton* pattern on most of the resources.

3.1 Main (*Main.js)*

The only purpose of this file is to call the appropriate functions from the other modules in order to start the game according to the flags set in the *Settings* (that control whether the game should show debug info of various kinds, among other things). All these calls are performed in the *createScene* function. The render loop is very simple and is called by the *executeWhenReady* callback provided by Babylon.js.

3.2 Physics (*GameEngine/Physics.js)*

Physics in this game mostly use the Cannon.js engine, with Babylon.js acting as an interface. Collisions and gravity are simulated in this way and they’re critical to the game as it is mostly centered around manipulating object (by picking up and launching or dropping them) or shooting projectiles.

The *Physics*.js file acts as an interface to the Physics engine. Babylon.js offers a very tight integration with several Physics engines among which Cannon.js was chosen. The Physics class initializes the engine in the *init* function (called in the Main.js *createScene* function). It offers the possibility of using the *PhysicsViewer*, a graphical debug tool in Babylon for in-game physics (it is used to show the bounding boxes of physical bodies).

The main functionality of this class is to provide methods that add colliders and impostors to in-game objects.

Colliders are bounding box meshes, used to contain more complex meshes. They can be viewed by checking the “*Show colliders*” checkbox in the Options menu, before starting the game. As an example, a collider is used to provide a bounding box to the complex mesh of the pistol.

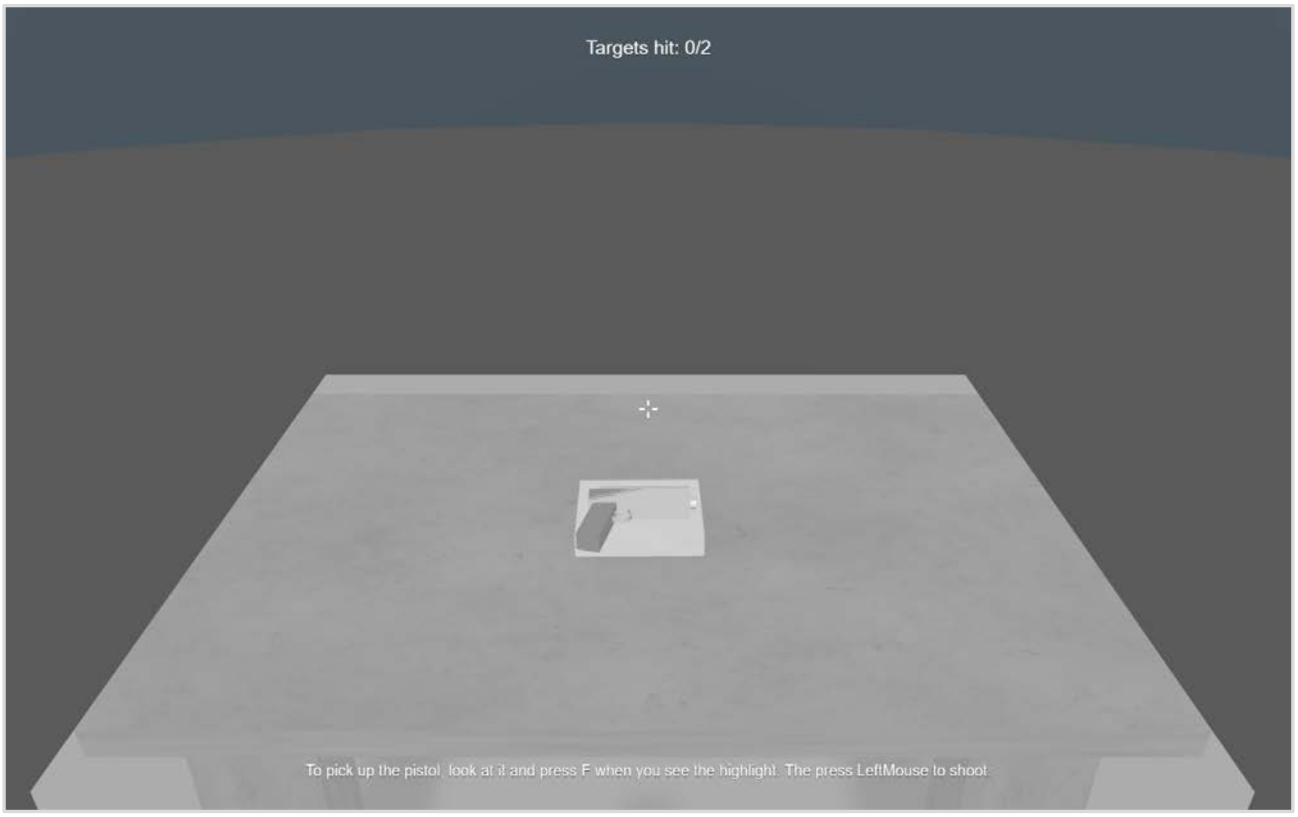

Targets hit: 0/2
To pick up the pistol, look at it and press F when you see the highlight. The press LeftMouse to shoot.

Figure 7 - Pistol with collider visible (also the collider of the table is visible)

Physics impostors are physical bounding boxes used by the Physics engine to provide collision capabilities and physical properties to the objects in game.

For most complex meshes in this game (such as the player and the pistol) a *collider* is added first and then a *physics impostor* is attached to it.

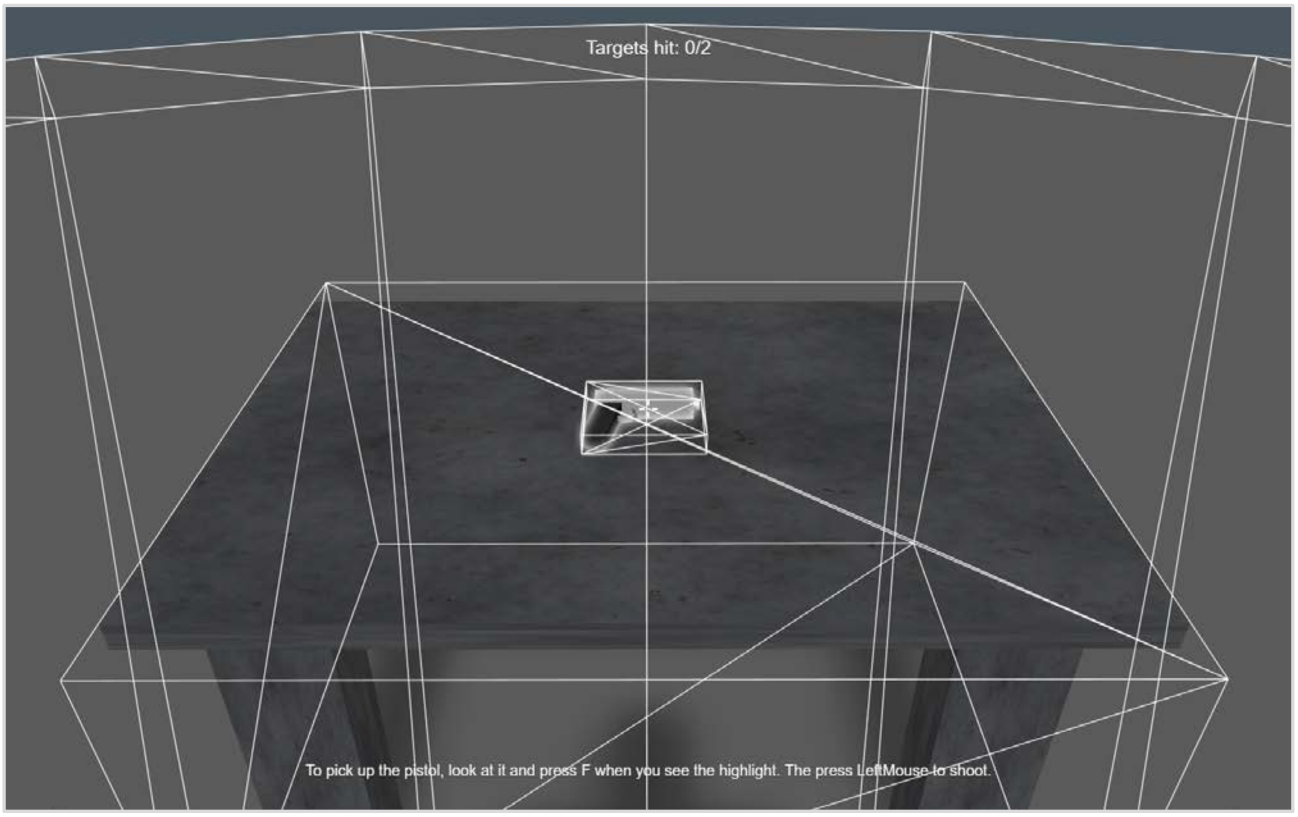

Targets hit: 0/2
To pick up the pistol, look at it and press F when you see the highlight. The press LeftMouse to shoot.

Figure 8 - Pistol with physics impostor visible (also the impostors of the table and the level platform are visible)

3.3 Game Engine (*GameEngine/GameEngine.js)*

The *GameEngine* class contains all the game control logic and some additional "global" features of the game.

The most notable feature of this module is that it controls the main "flow" of the game: loading the main menu, starting/pausing/resuming the game, changing levels. To do so, it performs the series of operations necessary to properly control the *LevelManager* and the GUI, request/release the browser pointer lock, activate/deactivate user input listeners, manage the camera.

This class also provides the basic logic for user input detection: two maps from character key to boolean are used to hold the state of each pressed key: if a *keydown* event (the key is pressed) is detected, its value is set to *true* for that key; on a *keyup* event (the key is released) instead, its value is set to *false*. To do this, using the Babylon.js *ActionManager* system, two keyboard keys listeners are registered:

- Listening to the *BABYLON.ActionManager.OnKeyDownTrigger* event, the first listener populates the *currentInputsMap* setting to *true* the value for keys currently being pressed. To detect continued press (mainly to avoid repeating multiple times an action triggered by a keypress) a "delayed" map *repeatedInputs* is populated as well: if a key has a value of *true* in *currentInputsMap* and is still held pressed (which is most likely), its value is set to *true* also in the *repeatedInputs* map (with an iteration of delay).
- Listening to the *BABYLON.ActionManager.OnKeyUpTrigger* event, the second one actually does the opposite of the first one (turning to *false* values in both maps when keys are released).

Then, using Babylon's *onBeforeRenderObservable*, code to open the menu is registered (to be triggered when the corresponding key, M, is pressed). Eventually, through the *onPointerDown* listener, pointer lock request/release code is registered.

Another key functionality of this class is creating/destroying/managing the camera. Four camera modes are available: *FREE* (not used); *FIRST/THIRD_PERSON*, that are used when a player is instanced; *MENU_CAMERA*, rotating around the center of the level platform, providing a background to the main menu.

Finally, helper methods to add/remove shadows are provided (they where added to this class to make them globally available).

3.4 Level manager (*GameEngine/LevelManager.js)*

This class is tasked with managing the creation and loading/unloading of levels, as well as the randomization features. With the *initializeLevels* method, a list containing all available levels is initialized (each element of the list is the function that initializes a certain level), excluding the menu background level that is initialized separately. Common objects (the *SkyBox*, a box with the sky texture, and the sun light) are spawned once for all levels).

By calling *nextLevel*, the next level is loaded. There are two possible situations:

- Not all levels in the list have been loaded already, therefore the next level is loaded.
- All levels have been loaded already, therefore the next level is chosen randomly among the levels in the list and is itself randomized. The randomization logic is defined in the level creation function and is activated through a boolean parameter of the function.

The available levels are:

- **Shooting range**: a pistol is spawned on a table (both are collision-enabled but only the pistol is pickable). In the base version two targets spawn in random positions in front of the player. In the randomized version, targets vary in number, size, position and texture.

- **Basketball**: a basketball is spawned on the ground as well as a basket. In the base version, the basket is mounted on a stand. In the randomized version, a single basket may appear or there might be more than one floating in the air, each one at a different height and position.

Notice that randomization also makes sure that randomizable objects are spawned sufficiently distant from each other.

3.4 Props and Materials (*Meshes/PropManager.js)*

“Props” (abbreviation of *Properties*) are objects that the player can interact with (which therefore usually also have physical properties). The *Props* class contains functions to spawn said objects, a function to “despawn” (correctly remove from the game) a prop and a function to “respawn” one (remove and spawn again).

3.4.1 *metaData* field

Most props hold in their *metaData* field (provided by Babylon) a dictionary with important data to be attached to the object itself for further use: initial position/rotation; boolean flags that determine whether the object is *pickable* (it’s possible to pick it up), it is *picked* (it’s being held by the player), it is *highlighted*, it is a *shooter* (it can shoot projectiles) and in this case, in the *projectile* field, the function that spawns said projectiles.

3.4.2 Custom condition checking

Babylon offers a very useful listener mechanism (*Observers*) for a limited set of events (*Observables*), to execute random actions when one of these events is triggered.
The function ***attachObserver*** implemented in the *Props* module expands this mechanism further, by storing for in-game objects a *condition list/function* pair, such that when the condition is met, the function is executed.
This is achieved by using a map from object *uniqueId* (a Babylon native field containing a unique id for each scene node) to said *condition list*/*function* pair.
The conditions are checked before each render loop using the *onBeforeRenderObservable* observer: for each object registered in the map, for each condition list/function pair registered for that object, if the conditions in the list are met, the function is executed.
For example, this mechanism is used to respawn objects (or the player itself) when they fall off the level platform. The *detachObservers* function instead unregisters all observers for a certain object.

3.4.3 Highlight and Pick-up

This class also provides functions to **enable/disable** the **highlight** around objects: in order to do so, the *HighlightLayer* from the Babylon engine is used and it is limited only to objects that are *pickable*. Whenever an object is being looked at, these functions add or remove the object from the highlight layer (if the object is hierarchical, they do so recursively on all nodes of the hierarchy) and set the *highlighted* field in the object *metaData* accordingly.

Another key function is used to “pick up” or drop objects. The *togglePickObject* toggles the picked-up state of an object in the following way:

- If the *picked* field of the object *metadata* is *false*, the object has to be picked up, therefore this field is set to *true*, the *pickedObject* field of the *Player* controller class is set to this object reference and using a *onBeforeRenderObservable* observer the object position is kept aligned with the camera (that is supposed to be in first person all the time, otherwise the object is dropped), by also setting its angular velocity to be null (so the object always faces the camera in the same way). A reference to this observer is saved in the object *metadata* (in case it has to be unregistered later).

- If the *picked* field of the object *metadata* is *true*, the object has to be dropped or launched (this is determined by the *launchForce* parameter). The observer is removed and all fields previously mentioned are reset. If the *launchForce* parameter is *undefined*, the object linear velocity is set to zero so that the object will fall vertically to the ground, otherwise the object will be launched with a velocity proportional to its value, in the direction the camera is facing.

3.4.4 Highlight and Pick-up

The following objects are noteworthy:

- The *pistol* is a pretty complex hierarchical model with a fall-off condition checker attached to it (it gets respawned when it falls off the platform).
- The *basketball* prop is pickable and it has a fall-off condition checker, that respawns the ball when it falls under a certain altitude. Also, using the *OnIntersectionEnterTrigger* observer, it plays a sound when bouncing off surfaces. The sound is spatial (meaning that its volume depends on the distance of the object from the camera).
- The *basket* is a hierarchical model. The backboard has a texture on only one of its faces: to achieve this result the UV textures of its faces had to be manually altered so that the applied texture showed up only on this face. It has colliders for any one of its parts. In addition, inside its hoop, it has an invisible cylindric collider, dedicated to detecting when the ball passes through the hoop, therefore scoring a point. To avoid detecting false positives; the ball center has to come very near its center; its vertical velocity has to be negative; the ball has to be not picked up by the player.
- The *projectile* is the object being shot by *shooter* objects. It is composed by a spherical “head” and a cylindric body (to resemble pistol bullets). It has a despawn condition when it falls under a certain height and a condition that detects when a shooting range target is hit. This last condition checker despawns the target, scores a point and spawns a new target if this is required by the current level. It also emits a gunshot sound when spawned.

3.4.5 Materials (*Materials/materials.js)*

Materials are loaded at the game initialization, given their relatively low resource consumption. This design choice allowed to avoid loading screens in between levels. Most materials are PBR, obtained using Babylon.js custom features: in most cases the layered textures comprise an albedo layer, a bump map, a metalness map, a displacement map and a roughness map.

3.5 Player

Given its importance and complexity the code regarding the player is split in three different modules.

3.5.1 Player control and movement (*“/PlayerController.js, “/PlayerAnimations.js, “/PlayerModel.js)*

The class *Player* contained in the file *Player.js* contains all the main logic regarding the spawn/despawn of the player, its movement according to user input and the movement of the camera when attached to the player. It also starts/stops animations based on the current movements and state of the player.

Instead of using a state-machine, the player state is modeled using boolean flags to represent state information (whether the player is falling, jumping, on the ground etc.).

There can only be one instance of player at a time, therefore a singleton pattern is enforced in the *spawn* function. To spawn the player:

- if an instance of the player already exists, it is despawned.
- then the player model is created, a collider and a physics impostor are positioned at its feet, with a height being approximately a third of the height of the player model (in order to avoid the player “toppling over” the collider is “wide and flat”)
- animations are initialized from the *PlayerAnimations* class
- all listeners and observers that allow its movements need to be activated/deactivated and the camera has to be correctly set up in order to follow the player movements.

There are three different functions to activate these listeners (with their counterparts to deactivate them):

- *attachPlayerMovementControls*: all the logic that controls player movements and animations is contained in an *onBeforeRenderObservable* observer (therefore executed at every rendering loop). Roughly, the following operations are performed at every iteration:
    - The player impostor rotation and angular velocities are reset to zero (excluding the vertical velocity), to prevent the player from “toppling over”
    - A *feeler ray* from the bottom face of the player collider box is used to detect whether the player is on the ground and the player state is updated accordingly.
    - If the player is falling and the falling animation has not yet started, all animations are stopped and this animation is started.

    - Handling of the various possible user inputs is performed (if the corresponding keys are pressed). It generally features an update of the player or game state followed by the start/stop of an animation.
    - If the player velocity is non null, velocity is damped (cutoff if too little). This method was preferred to using attrition from the physics engine as it proved to be prone to inconsistent behaviors.
- *attachCameraRotationControls*: this function registers two observers:
    - the first *onPointerObservable* observer sets the camera rotation vector by tracking mouse movements across the screen.
    - the second *onBeforeRenderObservable* observer updates the actual rotation and position of the camera in the scene. This is done asynchronously because the *onPointerObservable* event is fired only if the mouse pointer is moved, therefore the need for the second observer arises from the fact that the camera position has to be updated even if the mouse pointer is not moved (as the camera has to move with the player)
- *attachPickingTargetCheck*: this function registers two observers as well:
    - the first *onAfterRenderObservable* observer checks which object is being looked at using the *pickCurrentAimedTarget* from the *Utils.js* file and highlights this object if it's *pickable* depending on its distance.
    - the second *onBeforeRenderObservable* observer checks if the key for picking up objects (the F key) is being pressed and in that case picks up/drops the object being currently looked at. If the Left Mouse Button is being pressed instead, the object launch is charged (while doing so, a GUI progress bar is shown and updated with the current launch value).

To detect the press of the Left Mouse Button (LMB) two handlers are used, bound using normal browser event listeners:

- the LMB down event handler stores this input in the *GameEngine* and, if a shooter is being held, shoots a projectile in the direction of the camera.
- the LMB up event instead launches the object if it's not a shooter, with the currently accumulated launch force

Finally, the movement functions use some basic geometry to compute the player movement direction, setting the player linear velocity according to user input. The jump is obtained by applying a vertical impulse to the player and letting the gravity do the rest.

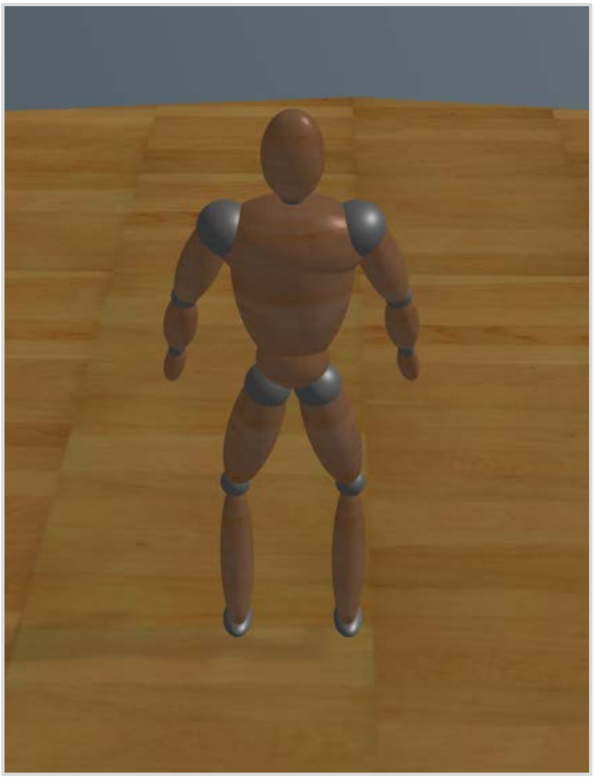


Figure 9 - The player model

3.5.2 Player model

The model is hierarchical and is obtained by composing ellipsoids. The design is supposed to remind of a drawing mannequin. The *metaData* of each part is used to store initial rotation/position for use in the animation.

3.5.3 Animations

The animations use the native system offered by Babylon.js. The *PlayerAnimations* class contains function that create and return animations. In order to obtain the synchronous motion of all interested parts of the player model, animations are grouped using Babylon's *AnimationGroup* objects. All animations are "parametric" in the sense that all their values (angles and positions) are computed parametrically with respect to some initial constants (this helped in the animation "tuning" phase) and then are added to the initial rotation/position store in the player model *metaData*.

To have a better view on player animations, please press **T** while in third person, (this moves the camera at a 90 degrees angle to the player).

This class also contains functions that automatically build an animation group to perform a smooth transition between the current state of all parts of the player model and the starting state of the first keyframe from a certain animation and then start that animation group. The only one function of this kind being used is *transitionToAnimationGroup*.

3.6 GUI (*GameEngine/GUI.js*)

The GUI is built using the Babylon.js GUI library. It consists of several menu pages each one containing text and/or buttons, a page changing system for these menus and some “overlays”, text and graphical elements that can be shown or hidden when necessary during the game. Noteworthy examples of overlays are the in-game crosshair, rendered as a GUI element, and the launch charging progress bar. Also sounds are played according to user inputs and GUI events.

4. Conclusions

The presented project is intended as a “proof-of-concept” of a game (in fact its level design is very basic and only intended to show the features of the game). It is built with a strong focus on modularity and extendibility, in order to be easily expandable. Babylon.js proved to be a very powerful tool, with a very active community and even though the documentation is somewhat poor in some cases, the contribution of the community easily provided fast answers to most of the issues encountered during development.