Interactive graphics MiniGame: a simple Babylon.js game

Andrea Del Vecchio 1749574
delvecchio.1749574@studenti.uniroma.it

1 Introduction

The aim of this project is to implement a very simple game using Babylon.js. The game uses mainly Babylon js and the Babylon js 3D GUI. A number of interactions are available (enabling/disabling the sound, changing difficulty, opening the main menu) to make the game *interactive*. The project has been tested on mozilla firefox and microsoft edge. In order to properly run it, be sure to have a web browser that supports WebGL 2.0 and an up to date javascript interpreter since classes are widely used in the source code. To directly jump to the game description go to the section 7 The Game. For additional technical information read the following section.

2 The framework

Babylon.js is a very powerful framework which offers an high level API to create 3D graphic programs that run on web browsers. It is very easy to uses and guarantees GPU hardware acceleration freeing the user from the burden of directly programming the shaders. Babylon.js uses a powerful rendering system which automatically computes the world and model view matrices considering the cameras and the object present in the scene. It also offers a set of API that serves the purpose of defining light sources which can be point lights, directional lights, spotlights or hemispherical lights for example. Objects in the scene can be equipped with materials which will react differently to the different components of the light sources.
Babylon.js offers also a powerful system that can be used to synchronize the loading process of each element in the scene through the *assets manager*. The assets manager can be istantiated to import all the sound and the desired meshes right before the actual scene is rendered so that the player never experience a **partial rendered scene**. With the asset manager, we can stack a set of tasks which will be executed one after the other. After executing a task in the stack, the asset manager will execute a **callback function**. To fully exploit the assets manager, we associate to each loading process a separate task. To exploit this mechanism we can modify the asset manager to render the scene only when

executing the last stack element callback.
The core of a Babylon js system is undoubtedly the **engine**. An engine is responsible of loading the scene and handles the render loop, in which it renders the scene vertices and execute functions. The scene, on the other hand, wraps all the objects that are used to effectively create the level, all of which are rendered in each iteration of the render loop [1].
To handle scene object interaction Babylon.js provides an handy interface called ActionManager. The action manager contains a set of triggers which can be set to execute actions when a certain event happens. For example, when two meshes instersect and the action manager is istantiated, a special kind of event is triggered and a callback fuction is executed. This system is exstensively used, for example, to create the shooting system in the game. The trigger should be considered as a simple test which returns a boolean and thus the action will be performed if and only if the result of the test is true. There are many type of actions defined by the Babylon.js API: some might set certain values of an object property, some others may execute code that will alter the status of the scene.
Each scene is represented by a Babylon.Scene istance. To fully exploit Babylon modularity, we assign to the menu and to the game level separate scene objects. To create animations, the programmer may use the babylon Animation class which, among other things, makes possible to define key frame animations, a powerful yet easy to use paradigm to define animations [2].
Babylon.js allows also the user to include a special object in their project, called *particle system*. A particle system can be used in different scenarios to simulate water flows or blood coming out from wounds. In this case the particle system has been used to simulate fire. It is quite simple to use. You mainly need to specify the mesh source, an emission rate, the emission direction and the eventual texture or colour applied to the particles.
A particle is indeed a very small cube. So it is possible to set a colour and a texture to it. The *emit rate*, on the other hand, specifies how many particles are generated per second from a given source. The user may also specify a minimum and a maximum lifetime for the particles [3].
To render the main menu and the player status information, we use the Babylon.js 3D GUI. Each element in a Babylon GUI is associated to special kind of texture which is called **advanced texture**. An advanced texture could be concieved as a container which can store other GUI elements like buttons, checkboxes, textblocks or other GUI panels. The panels are lower level containers which can contain other elements in a hierarchical way. Defining a GUI with Babylon.js is relatively straightforward and most importantly, the advanced texture is just an element of the scene, so it does not cause any conflicts with other scene objects.
To include basic **physics interaction** between scene objects we used Cannon.js, the default physics engine of Babylon.js.

[1] `https://doc.babylonjs.com/api/classes/babylon.scene`.
[2] `https://doc.babylonjs.com/babylon101/animations`.
[3] `https://doc.babylonjs.com/babylon101/particles`.

3 The environment

3.1 Description

The scene level is very simple. It is a flat 2000x2000 map placed somewhere in the space that represents an arena. In this scenario two team of tanks fight to the death: the **red** and the **blue** team tanks.

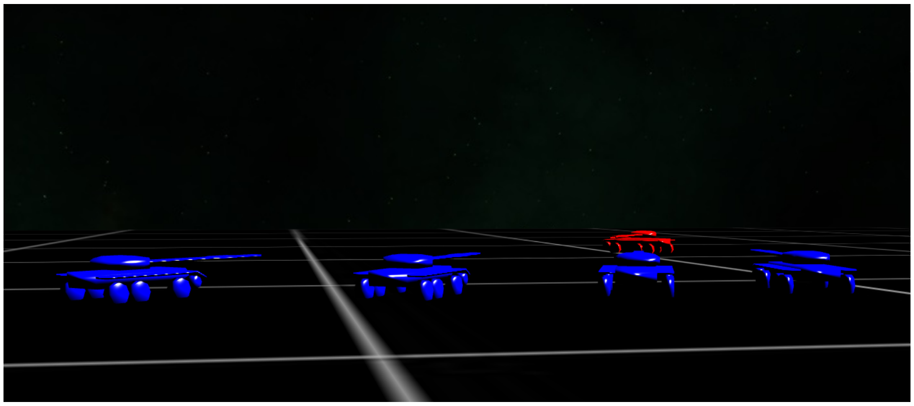
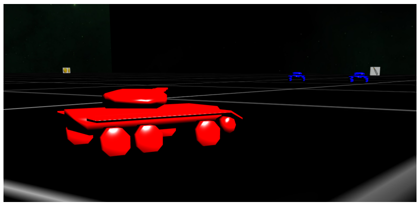
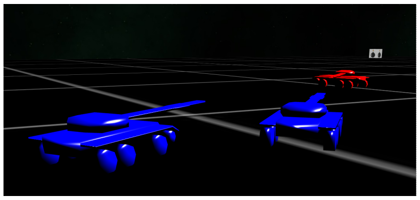


Figure 1: Overview of the fighting arena. The red tank and the blue tanks fight to the death.

The player plays as a red team tank and he is the only member of its team. He is able to move in the map with its tank collecting bonuses from the loot boxes and shooting towards the opponent tanks. The scenario is brightened by a white directional light simulating a nearby star or a big reflector. The map is completely flat, so the player tank do not need to worry about the uneven ground.





The ground is equipped with a *physic impostor*, a special Cannon.js object which allows basic physics interaction with the scene objects. It has no borders so the player needs to be careful not to move outside the map otherwise he will be teleported to its spawn position.

3.2 Implementation details

To completely render the scene, Babylon js forces the program to create a camera. So at least a camera must be specified. Since we need an external third person shooter camera or a *external* vehicle camera, we created a **follow camera** object.

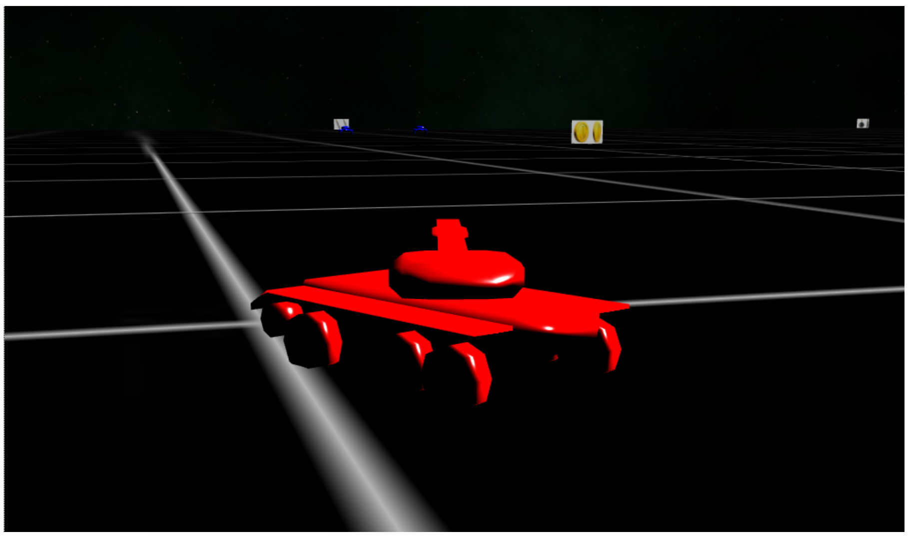


Figure 2: Overview of the third person shooter camera.

The follow camera is a powerful tool available in Babylon.js. The camera needs a target mesh to follow and can be tuned with a variety of parameters. In order to make the player able to drive the red tank, we set the tank as the target of the follow camera [4]. The API also makes possible to attach the camera to the HTML canvas so that the user can directly control with the mouse and the keyboard arrow buttons and thus we attached the follow camera to the canvas. Moreover HTML 5 provide a useful feature when it comes to program an interactive game: the **pointer lock**. With this feature, the canvas are able to *capture* the mouse pointer so that the user cannot click outside it causing undesired effects (for example closing the browser window by mistake terminating the game).

[4]Since the tank is a hierarchical model imported from the web, the camera is attached to the **root** of the tank.

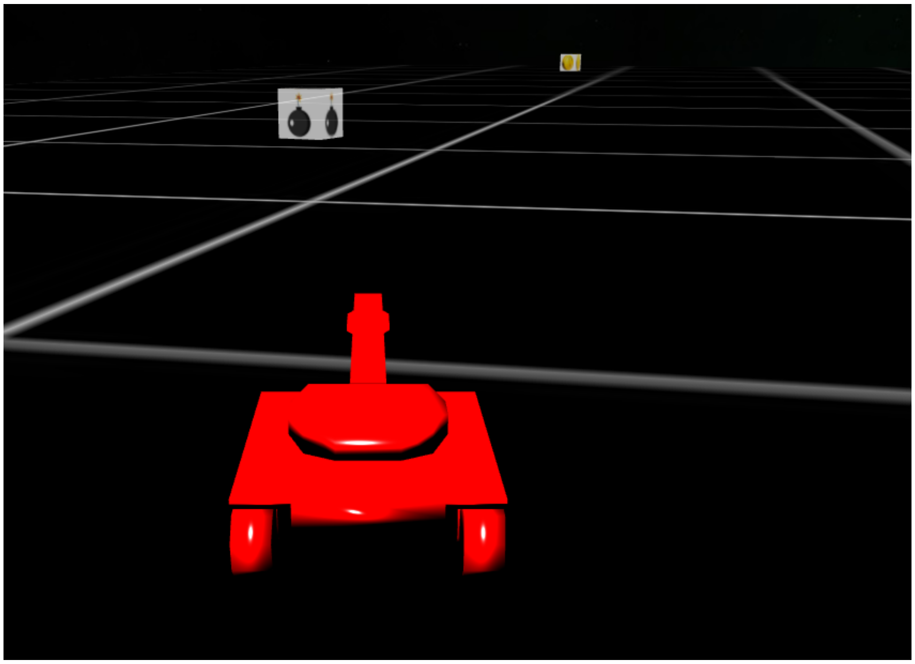


Figure 3: The scene in easy mode with two of the nine loot boxes. Near the tank, a cannon ball loot box and in the background a point loot box.

At any time the user can get full control of the mouse by simply clicking the *Esc* keyboard button.
Depending on the chosen difficulty, which can be either **easy** or **hard**, the engine will render 9 or 6 loot boxes [figure 3]. The boxes are texturized using the advanced *cube texturing* mechanism provided by Babylon.js which is also used to create skyboxes. To attach a cube texture to an object six images corresponding to the six faces of the cube are needed. The images must be placed in the same directory and needs to have a *common name* which is equal to each of the six images followed by *position tag* which must be one of (px,py,pz,nx,ny,nz) [5].
px, py, pz stands for *positive* x,y,z. nx,ny,nz stands for *negative*x,y,z. Each of the preeciding tags corresponds to a face of the box to be texturized. The position of each face follows the Babylon.js cube textures convention that is specified in the official documentation.
Each box is animated thanks to a **key frame animation**. To create a key frame animation, we have to define a list of javascript objects in the form

$$\{\text{frame: value}\}$$

where the **frame** is a frame number and the *value* field represents a possible value for an object property at that frame. In this case the boxes are equipped

[5]See `https://doc.babylonjs.com/resources/playground_textures#cubetextures`

with two key frame animations: the first one for its position in the scene and the second one for the rotation along the y axis. The animation is is composed by 100 frames and runs at 10 frames per second so that the player has an impression that the boxes **fluctuates** above the ground. This type of animation is inspired by the **Mario Kart**™ floating boxes that contain the various bonuses for the players in the race.

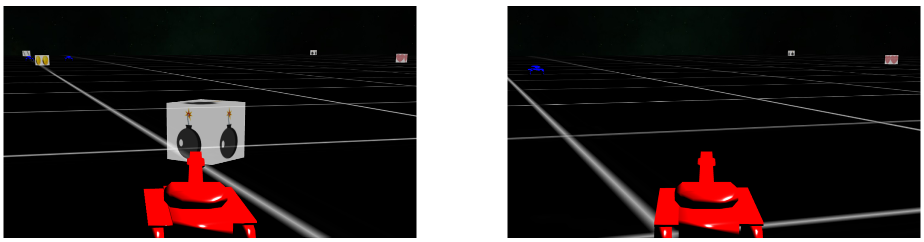


Figure 4: The player tank before and after collecting the cannon ball loot boxes.

The boxes are associated to a Babylon.js trigger [figure 4]]. When the red tank, which from now on we will call the **hero tank**, intersects one of the boxes, the box will disappear and the associated bonus is collected by the player. The loot box mesh is then destroyed to free some memory. Since the red and the blue tanks are composed by a considerable amount of vertices, we paid particular attention to free memory whenever it is possible to optimize the scene. Both the red and the blue tanks are composed by a considerable amount of vertices, so keeping in memory the position of unused objects which should not be rendered by the scene engine is a huge waste of computational resources. After a period of time of 5 seconds the boxes will respawn to their original positions and are ready to be collected again by the player. Then, the corresponding vertices are recreated and passed to the GPU for rendering.
There are mainly four type of boxes:

- A **point box** represented by a big yellow coin ™. Collecting these boxes will make the player to earn a point. The amount of points a player has collected so far is shown in the top left corner of the screen. Since opponent tanks will keep spawning, collecting points is crucial to complete the game and obtain the **win condition**.

- A **machine gun box** represented by a big machine gun bullet. Collecting those boxes is not vital but it allows the player to obtain 10 additional machine gun bullets. The machine gun can be used to kill the opponent tanks. The amount of machine gun bulltets is shown in the top left corner of the canvas.

- A **cannon box** represented by a big cannon ball. Collecting those boxes is not mandatory to win the game but it is still useful as it allows the player to collect 2 additional cannon balls. Cannon balls can be used to kill opponent tanks and deals more damage than machine gun bullets. The amount of cannon balls is shown in the top left corner of the canvas.

- A **health box** represented by a big red cross. Picking the health boxes could be a good strategy to play a safe game since the opponent tanks are very aggressive and constantly try to shoot towards the player tank to hunt it down. Collecting health boxes guarantees the player 3 additional life points until a certain maximum amount of life points is reached. The amount of life points is shown in the top left corner of the canvas.

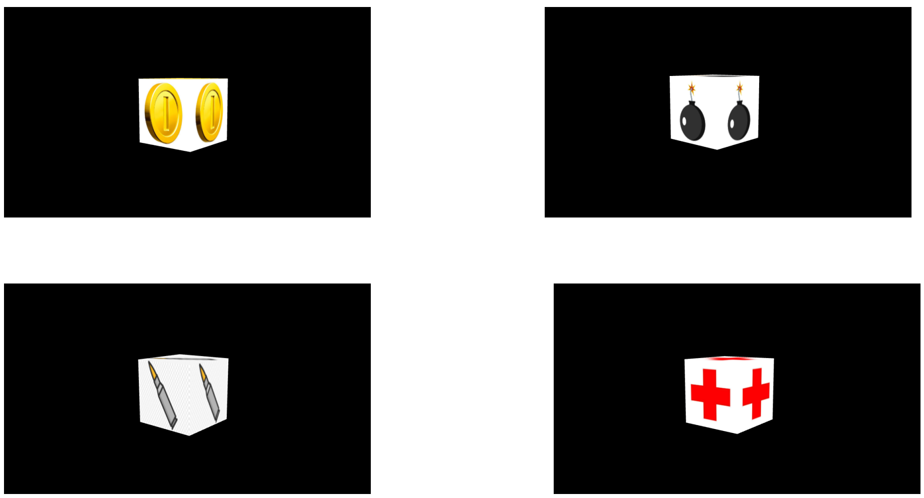


Figure 5: The four type of loot boxes that are present in the scene.

At any time during the game the player can go to the main menu by simply pressing the *m* button. This will cause the current game to be terminated and the corresponding scene to be disposed. From the main menu the player is able to change the difficulty of the game, to enable or disable the sound or to start a brand new game.
The environment has also a skybox. The texture of the skybox is taken from the official babylon js documentation and attached as a cube texture to a big box that surrounds the scene.

4 The tank model

4.1 The hierarchical model

The tank model is imported from a website which contains a series of free models that could be downloaded and used in Babylon.js projects [6].

[6] See also `https://clara.io/library?query=tank&gameCheck=true` for more information.

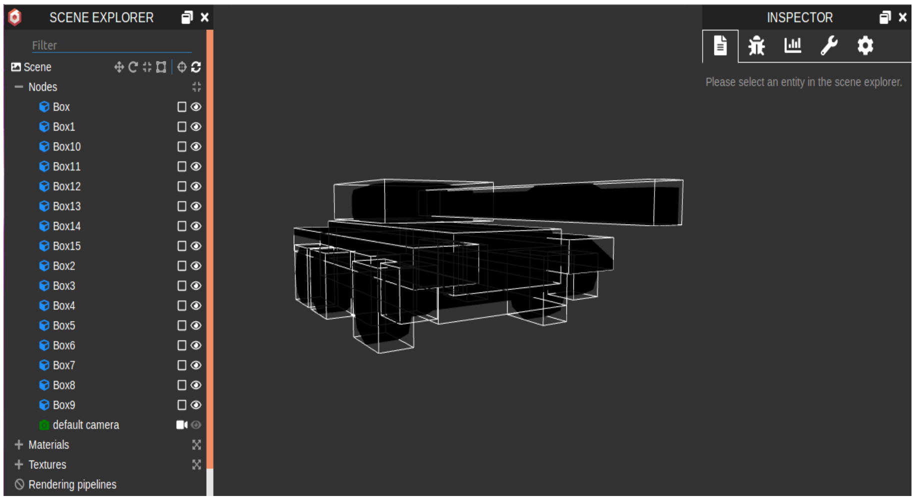


Figure 6: The imported model viewed through the scene inspector of the Babylon.js sandbox. All the component meshes are highlilghted toghether with the corresponding names.

The imported mesh is in a form of a set of unlinked boxes [figure 6]. It means that, at this stage, the tank is not a hierarchical model and managing it in the scene could be a challenging task.

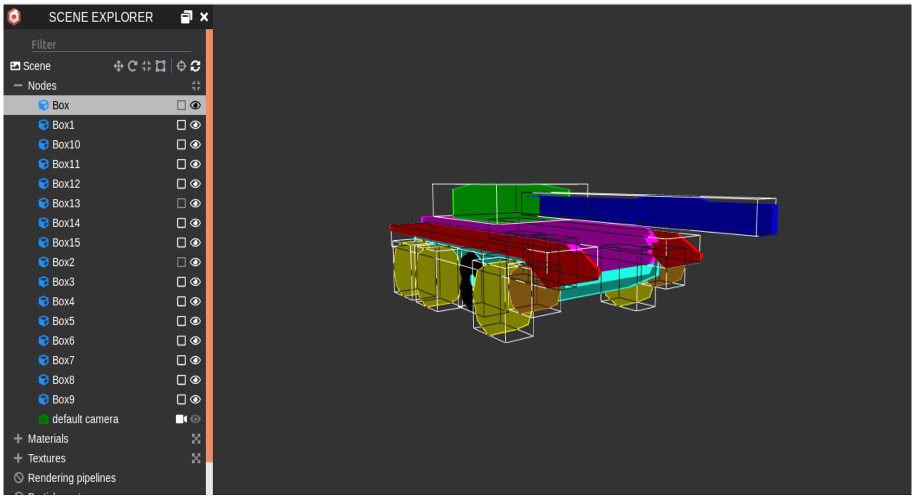


Figure 7: The imported model viewed through the scene inspector of the Babylon.js sandbox with all the model components highlighted with different colours. We have then: the root in light blue, the body in purple, the turret in green, the cannon in blue, the wheels in yellow, the front wings in orange, the wings in red. The mesh in black has been discarded from the actual game for design purposes.

Fortunately Babylon.js implements a simple mechanism through the *addParent, addChild, removeParent,removeChild* functions which is able to istantiate a hierarchical model on the fly [figure 7]. The mechanism is built so that each transformation that is applied to the root of the model is also applied to the children. Then, whenever we have to move the tank, it is sufficient that we apply a translation to the root of the model instead of applying it to each component mesh which would be dramatically difficult in some scenarios.

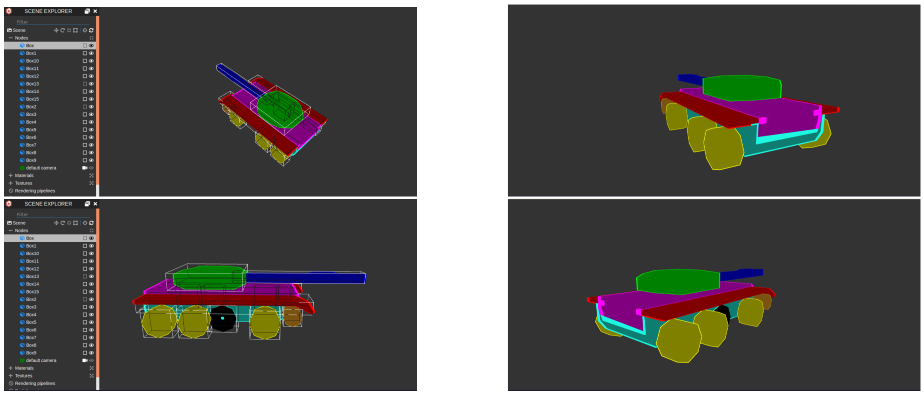


Figure 8: Two views of the different components of the tank model.

For simplicity, I decided to set the mesh named **"Box"**(the mesh in light blue) as the root of the model. All the pieces, apart from the cannon, are set as its *children* whereas the cannon is set as a child of the turret (the green mesh)[figure 8].

4.2 Animating the turret

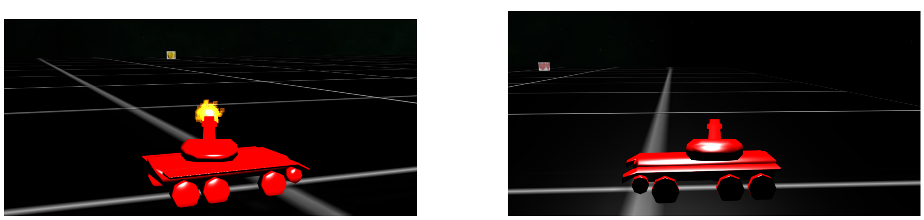


Figure 9: Two pictures depicting two different orientations of the tank turret.

One of the most difficult features in the game to implement was undoubtedly the animation of the turret. In this case, we did not use any key frame animation since we want the player to be able to rotate the turret differently from time to time following the mouse movements and doing so with a predefined set of values for each frame of the animation would have been impossible. Then, the turret animation is handled in a separate function [7].

To rotate the turret we need know exactly how much we have to rotate the tank turret to match the direction of the follow camera as in figure 9. Babylon.js comes in help with a very simple function to obtain the position that is right in front of the camera at a given distance: *camera.getFrontPosition(distance)* [8]. This function returns a vector which represents the position of the object

[7] See Tank.animateTurret() in tank.js for additional information.

[8] `https://doc.babylonjs.com/api/classes/babylon.followcamera#getfrontposition`

in front of the camera at a given distance. We call β the angle between the position of the camera and the position of this object. To obtain this angle we need to make few considerations. To perform this computation, we need two vectors mainly: the position of the object in front of the tank and the follow camera position. The difference between these two vectors gives the vector that joins them. The angle between the camera position and the vector difference is β, the angle we are looking for. We know also that the x component of the difference vector is the sine of the angle β and the z component is the cosine of the angle β. Given $d = (d_x, d_y, d_z)$ the difference vector, to obtain β we simply need to solve this simple non linear system of equations:

$$\begin{cases} \sin(\beta) = d_x \\ \cos(\beta) = d_z \end{cases} \tag{1}$$

By computing the ratio of the two equations, we know that

$$\beta = \arctan\left(\frac{d_x}{d_z}\right) \tag{2}$$

For simplicity, we want to rotate the turret just along the y axis regardless the orientation of the camera along the x and the z axes. Then, we slightly modify the difference vector replacing the y component of the difference vector to 0. Computing β though is not sufficient as this will cause the turret to be rotated also when the tank turns left or right. We want the turret to be rotated if and only if the camera moves. To do so we need to slightly adjust the angle to take into account the rotation angle of the tank. We may call this angle γ.

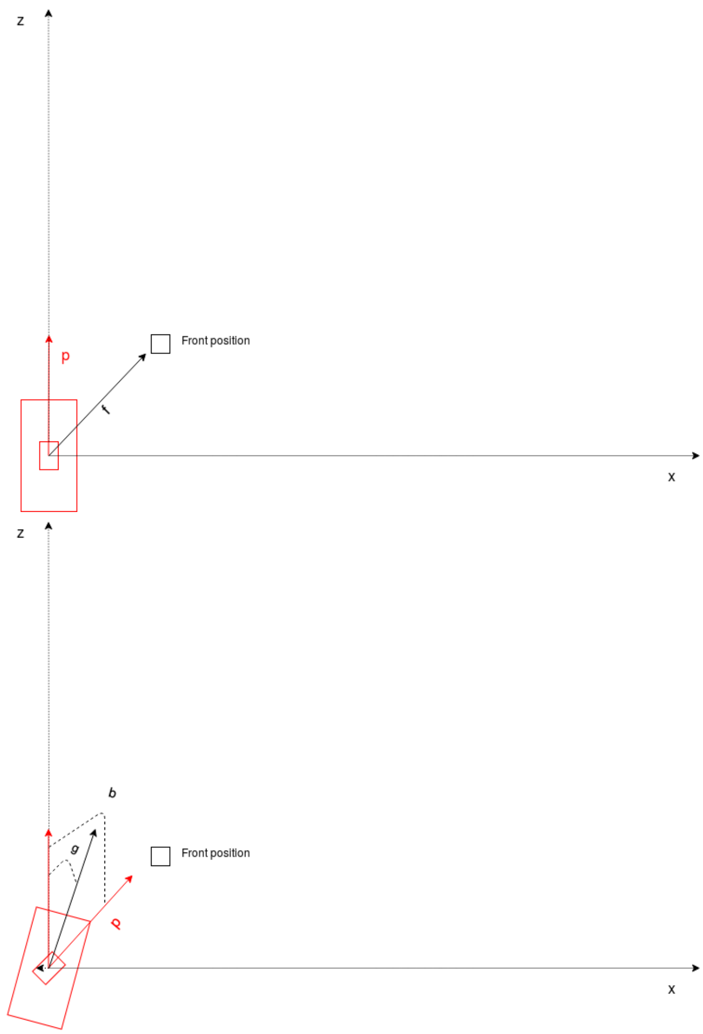


Figure 10: Simple scheme of the two phases of the tank turret roation. To accomplish the rotation we must consider also the tank rotation represented in this figure by the angle g. b is $\hat{\beta}$, the angle we want to compute to perform the rotation.

Suppose we turn right the tank by $\frac{\pi}{4}$ radians which corresponds to a rotation $-\frac{\pi}{4}$ along the y axis for the camera position vector. If I solve the system in the equation 2, we obtain an angle which is offset by γ. To get rid of this effect, we just need to subtract γ from β obtaining:

$$\hat{\beta} = \beta - \gamma$$

Once we obtain $\hat{\beta}$, we simply rotate the turret along the y axis by $\hat{\beta}$ radians and the job is done. Since the turret is a parent node of the cannon in the hierarchical model, rotating the turret will inevitably rotate the cannon which is the final effect we want to achieve in the whole animation.

4.3 Dealing with collisions

Babylon.js implements a very simple collision system that, if enabled, avoid that two meshes fully intersect. To enable the collision system we simply have to set a boolean variable in the mesh object. So not all the meshes have the collision system enabled.
We created a simple bounding box that surrounds the tank model and moves togheter with it. The bounding box dimensions are set so that each tank is completely surrounded by its bounding box and the other tanks can't intersect it [figure 11]. Whenever we move the tank, we also move the bounding box so that the tank and the box become a unique moving body [9].

4.4 The hit bounding box

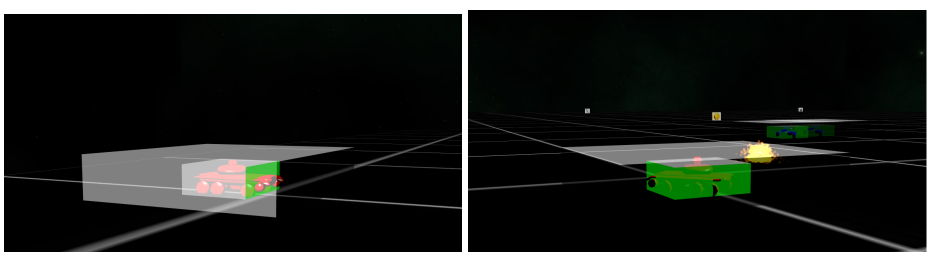


Figure 11: The hit (green) and the collision (gray) bounding boxes of the models.

In order to handle cannon ball hits, we use another smaller bounding box which form now on, we will call the **hit bounding box** [figure 11]. The hit bounding box is much smaller as we do not want the player to experiment any *phantom range damage* [10]
that will make the whole gaming experience quite frustrating. When moving the tanks, we handle the hit bounding box in the same way we handled the collision bounding boxes moving them toghether with the tank. In the final version of the game, both the two bounding boxes will be invisible, but still be rendered and considered by the Babylon.js collision engine.

[9]See the tank.move() function documentation in *tank.js*

[10]In video games, a *phantom range* is a very annoying problem that arises whenever the hit box/boxes of a given model is/are far larger than the model itself. This will cause the players to experiment damage from blows that actually hit a region in the space which is clearly far from the the player character, causing frustration and ruining the game experience. Actually many AAA games are recognized to have this problem.

5 The player tank

As we said before, the player tank is a mesh imported from a website. Choosing an appropriate mesh which allowed me to animate the various components was quite a difficult task. There are a lot of high quality tank models available online. The problem was substantially the fact that those meshes cannot be completely animated as the various part are not directly accessible by Babylon.js functions. Importing a highly detailed model of a world war 2 tank and not being able to animate the crawler is quite disappointing. Luckily, we managed to find this very basic tank model which comes as a collection of boxes of a given size arranged in such a way that they resemble a tank. Despite the minimal design the model is functional: all the boxes can be accessed and animated independently from one another.

5.1 Animating the model

5.1.1 Moving the tank

Once imported, we needed to define some basic interactions. We did not provide the ground to any form of friction force, so we had to find a way to simulate acceleration, braking and steering.
The first thing we did was to provide the player with a more *game like* movement system: instead of using the arrow keys or the left mouse button to move the tank, we addressed WASD for that purpose. Achieving this is quite straightforward through javascript event listener. First a global *WASD handler* object is istantiated to memorize which button of the WASD grid is pressed. Secondly, we enabled a javascript event listener on the WASD keys that triggers a boolean flag in the handler when each button is pressed. This results in the tank moving forward when W is pressed, backward when the S button is pressed and to rotate clockwise and counter clockwise when D and A are pressed respectively. Movement is done through the *moveWithCollision* [11] Babylon js built-in functions and rotation is applied using the *rotate* function of the Babylon.Mesh object. It is a very simple system and it works but it is not enough. We want the tank to actually accelerate up to a predefined top speed when W is pressed and to actually **break** when the S button is pressed. To accomplish this, we define a parameter *speed* for the tank object. The speed parameter regulates the movement speed of the tank. At the starting point the speed is set to 0. When the player presses W, the speed parameter is increased by 0.01 and this value is multiplied to the translation vector that defines the movement of the tank. On the contrary, when S is pressed the speed parameter is decreased by the same value. Once the speed parameter becomes negative, the tank starts moving backwards. If the player stops pressing W and S, the tank will gradually slows down until it completely freezes.
Moving the tank should cause the wheels to rotate as they get in contact with

[11] `https://doc.babylonjs.com/babylon101/cameras,_mesh_collisions_and_gravity`.

the ground. To accomplish this, we rotate along the x axis all the wheels clockwise when the speed parameter is greater than zero and counter clockwise when the tank moves backward.
The wheel rotation is performed at every render loop iteration.

5.1.2 External color and design

The player tank belongs to the red team by default. We then need to find a way to distinguish the player tank from the opponent tank team. This can be easily obtained by applying a red colored material object to the player tank surface. In Babylon.js a standard material will have four color components that will react differently from light sources:

- A diffuse colour component that will react properly with the diffuse component of the light sources.
- A specular colour component that will react properly with the specular component of the light sources.
- An ambient colour component that will react properly with the ambient component of the light sources.
- An additional emissive component that will render a colour which is independent of the light sources that are present in the scene.

For design purposes we set the diffuse colour and the emissive colour of the player tank to red. The diffuse component is intended to make the tank material sensible to react to the light sources, the second one to render a red colour even if in the scene there is no light source with a red diffuse component or there is no light source at all.

5.2 Shooting

The main feature of this game is to simulate an arena battle between two team of tanks. So there must be a way for both the player and the opponent tanks to cause some damage. In this game the tanks will have two weapons mainly: the **cannon** and the **machine gun**.

5.2.1 The cannon

The cannon is the main weapon of the tanks. It can shoot fire cannon balls towards the enemy and cause a damage of exactly 2 life points. The cannon ball source is placed directly in front of the tank cannon. Whenever the player shoots a cannon ball the program retrieves both the position and the orentation of the tank cannon and set it as the spawn position of the cannon ball.
The cannnon ball is rendered using a simple sphere mesh and has a red colour

to simulate the heat of the thrown cannon ball.

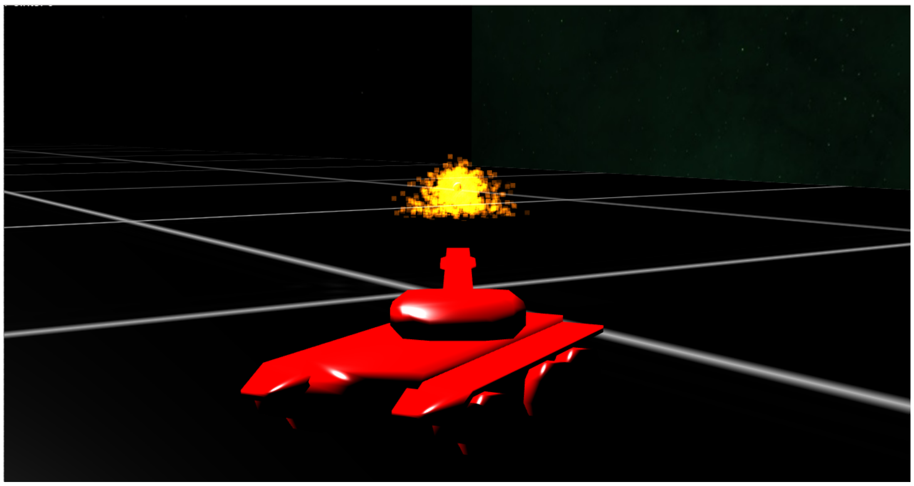


Figure 12: This image shows the cannon ball and the fire particle system.

In order to simulate a fire ball effect, we used a properly tuned particle system to simulate fire. To give the particles a spherical trajectory, we used also a *sphere emitter* object which sets a *spherical emission region* centered in the emitter mesh. The emitter source of the particle system is the sphere representing the cannon ball and we set the emission region to move toghether with the cannon ball as shown in figure 12.
The maximum lifetime of the particles is set to a few seconds and a fire texture has been applied to them.
The ball is equipped with a Cannon.js physics impostor and thus can react to other meshes in the scene thanks to the physics engine. In order to throw the cannon ball towards the direction pointed by the cannon ball we simply apply a physic impulse. The impulse is applied to the center of the sphere mesh and points towards the direction of the cannon turret. The strength of the impulse is a parameter of the Tank class and is tuned to create a balanced shooting system. The physics engine, then, applies an earth like gravity ($g_y = -9.81\ \frac{m}{s^2}$) to the cannon ball so that the latter will have a parabolic trajectory towards the shooting direction. The ball then reacts with the environment in the following way:

- If the cannon ball intersects the hit box of one of the opposite team tanks (friendly fire is not allowed), it will explode and cause damage to the target tank.
- If the cannon ball does not hit anything after 4 seconds it disappears and the corresponding mesh is disposed for efficiency reasons.

The code associated with the cannon ball action manager is executed when the cannon ball intersects the **hit bounding box** of the opponent tank. The action code stops the cannon ball particle system, disposes the cannon ball and starts another particle system which emits particles directly from the opponent tank body to simulate the explosion. If the game sounds are enabled, the function also reproduces an explosion sound once the cannon ball hits the opponent tank.

5.2.2 The machine gun

Very similar to the cannon ball, it represents the secondary weapon of the tanks. The machine gun is very easy to use but deals less damage.

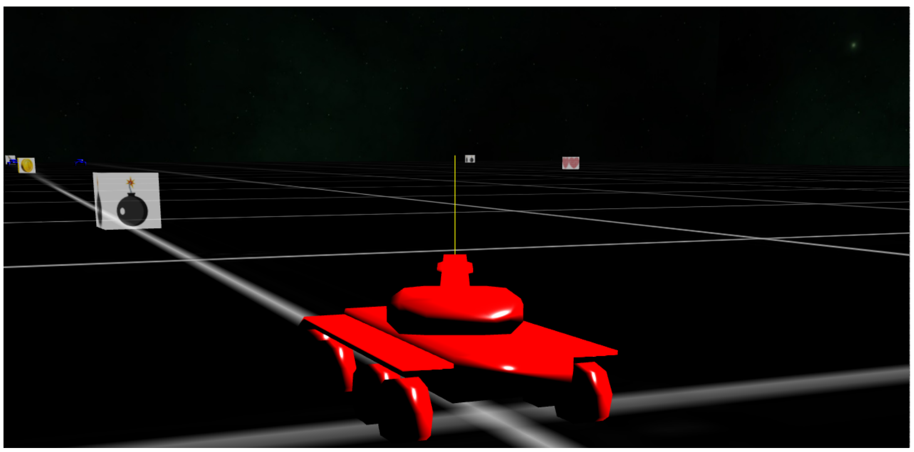


Figure 13: This image shows the tank using the machine gun.

To simulate the machine gun bullets we used Babylon.js *rays*. In Babylon.js a ray is, as the name suggets, a line in the context of a scene. To throw rays, we need to specify an origin, a direction and a length. By default, rays are invisible, but thanks to the *ray helper* we managed to make them visible as shown in figure 15. A Babylon.js ray intersects objects and the user can define callbacks that will be executed whenever an object is intersected. Obviously, we may not want this callback to be executed when the intersected mesh is, for example, the skybox. Babylon.js ray helpers allows the user to define precisely which meshes should be taken into account by setting appropriate boolean expressions. In this case, only the meshes which belongs to the opponent tanks should trigger this callback. Fortunately, they can be immediately spotted by inspecting their names. All the tank model meshes are given a name that contains the string ”Box” (”clone_1.Box14.Box15”,”Box7” etc ...), thus a simple solution to this problem is to use javascript *regular expressions*. If the target mesh name is the format described above, then it matches some regular expressions and when the ray intersects it, the appropriate callback will be triggered.

The callback structure is very simple. It decreases the target tank health and performs a simple test: if the health of the target tank is less or equal than 0, the target tank dies, otherwise nothing happens.

5.2.3 The tanks death

If the tank health reaches a value of 0, the tank must disappear from the scene. If it is the case, then a special death function is performed.
The tank death should be handled with care. When it comes the time of disposing a tank we must be sure that, not only the model will disappear, but also that all the objects that are associated to it are destroyed to **free memory space**. Then, before disposing the tank meshes we ensure that:

- All the tank particle systems are shut down.
- All the tank particle systems are disposed.
- The hit and collision bounding boxes are disposed.
- The tank meshes are disposed.

After destroying all the tank components, the function has a different behaviour depending on the fact that the target tank is the player tank. If it is the case, then another routine is called and a *game over* menu is displayed allowing the player to do nothing but to go to the main menu to start another game.
If the eliminated tank is an opponent tank, then a timeout of 5 seconds is set, after which the opponent tank is respawned. To handle the respawn of opponent tanks, we first retrieve its spawn point, which has been previously stored inside the obejct representing the tank. Then we create another clone of the tank model, and move it to its spawn position so that it can rejoin the fight.

6 The opponent tanks

The opponent tanks are handled inside a loop that is executed at every rendering iteration. In this loop each tank executes its policy.

6.1 The opponent tank policy

Currently the opponent tanks adopt a very simple random aggressive policy: at every rendering iteration a random number is generated. If this number is greater than a certain parameter which is defined by the tank policy, then the tank will shoot towards the player otherwise, it won't shoot. To decide whether the blue tank will use the cannon or the machine gun another random number is generated. If the value is greater than a certain other value (actually the policy shares the same two values in the two phases for simplicity), then the tank will switch weapon, otherwise, it keeps shooting with the same weapon.
The opponent tanks are designed to do nothing but to kill the player. They

will constantly follow the player tank trying to destroy it. For simplicity, the opponent tanks will never run out of bullets so they can potentially shoot for an infinite time, forcing the player to be dynamic trying to avoid the opponent bullets.

6.2 Movement

We handled the blue tanks movement in a separate function. We set the tank to always point towards the direction of the player tank and to follow it with a fixed speed. To accomplish this, we compute a difference between two vectors: the tank *front vector* and the player tank position.
The front vector is a vector that points towards the *front direction* of the tank. The tank front vector is always updated whenever the tank is translated or rotated. The difference between these two vectors returns the vector that joins them. The goal of this operation is to obtain the angle α between the opponent front vector and the difference vector. To do that we need to make few considerations: we know that the projection along the x axis of the difference vector represents the sine of the angle, whereas the projection along the z axis represents the cosine of the angle. To obtain the alpha angle we should solve the following non linear system:

$$\begin{cases} \sin(\alpha) = d_x \\ \cos(\alpha) = d_z \end{cases} \tag{3}$$

By computing the ratio of the two equations, we obtain:

$$\alpha = \arctan\left(\frac{d_x}{d_z}\right) \tag{4}$$

It is not sufficient to compute α. Since we used the negative front vector of the blue tank to compute the angle, if we perform a rotation of α radians along the y axis, the blue tank cannon will point exactly in the opposite direction of the player tank as shown in figure 14. To obtain the angle we are looking for, then, we need to sum π to the angle α. Then

$$\hat{\alpha} = \alpha + \pi$$

Once we compute $\hat{\alpha}$, we rotate the blue tank turret along the y axis by $\hat{\alpha}$ radians, replace its front vector with the difference vector and move the tank in the direction defined by the new front vector.

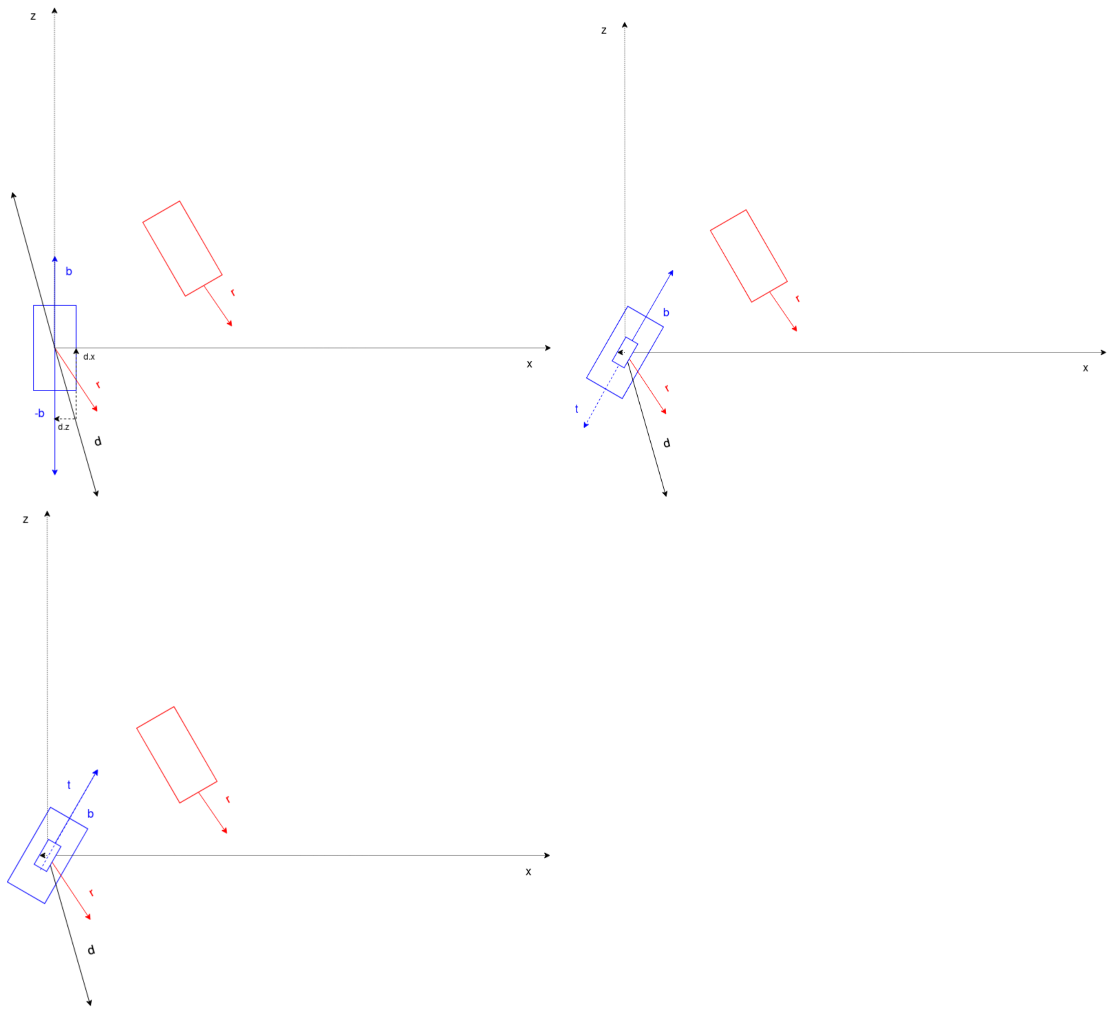


Figure 14: Scheme of the three rotations. In the first image, the difference vector d si computed. In the second one the blue tank is rotated clockwise by α radians. t is the vector specifying the direction of the turret, b is the front vector of the blue tank. In the third image $\hat{\alpha}$ is computed and the final rotation of the turret is performed.

By continuously doing this in the render loop, the blue tanks will never stop following the hero tank.

7 The game

This section aims at illustrating a user manual of the game.

7.1 Description

The game is a simple third person arena game in which the player tank and the blue tank team fight to the death. The arena is a simple flat map in which the tanks are able to move. Both the player tank and the opponent team can shoot with two weapons: the cannon and the machine gun. The player tank is also able to restore life points, to collect cannon and machine gun bullets from the loot boxes that float in certain positions of the map. The goal of the player tank is to collect a sufficient number of points to win the stage. The amount of points needed to win changes with the game difficulty which can be either **easy** or **hard**. In easy mode the player needs to collect 5 points whereas in hard mode the points needed are 10. The player can collect points by picking the *point loot boxes* which are easily spotted by a big gold coin texture. In easy mode, there are two point loot boxes that respawn after 5 seconds whereas in hard mode there is just one loot box making more difficult for the player to win the stage. The respawn time of the point loot box remains the same.
The player will face an opponent team of blue tanks that have the only purpose of killing him. Also the number of opponent tanks varies toghether with the game difficulty. In easy mode, there are just two tanks, in hard mode there are four of them. In each of the two game difficulties, the blue tanks will have infinite ammo for their weapons whereas the player tank has just few limited amount.

7.2 Commands

At the beginning of the game the player will be placed in front of the main menu. In this menu, he can choose the difficulty of the game which can be either easy or difficult, enable or disable the sound and start a new game.
When the match begins, the player starts with the red tank. The tank can be moved pressing WASD and the turret can be rotated by simply moving the mouse. Press the left mouse button on the canvas to have the mouse pointer locked inside the game and fully control the tank turret. Move the mouse to rotate the turret. Make sure that the turret points towards the enemy tanks, as otherwise you will waste bullets.
Press the left mouse button to shoot. You can shoot with two different weapons: the cannon and the machine gun. To use the cannon, press "1" whereas to switch to the machine gun, press "2". You can always terminate the game by pressing "m". To sum up:

- W : move forward/accelerate.
- S : move backward/break.
- A : turn left.
- D : turn right.
- Left mouse : shoot.

- M : main menu.
- L : free camera.
- 1 : switch to cannon.
- 2 : switch to the machine gun.
- T : Use tank follow camera.

7.3 Player guide

The game is very simple, but the very aggressive blue tank policy may cause the player to rapidly lose life points without even noticing it. A simple strategy to win a match could be the following.

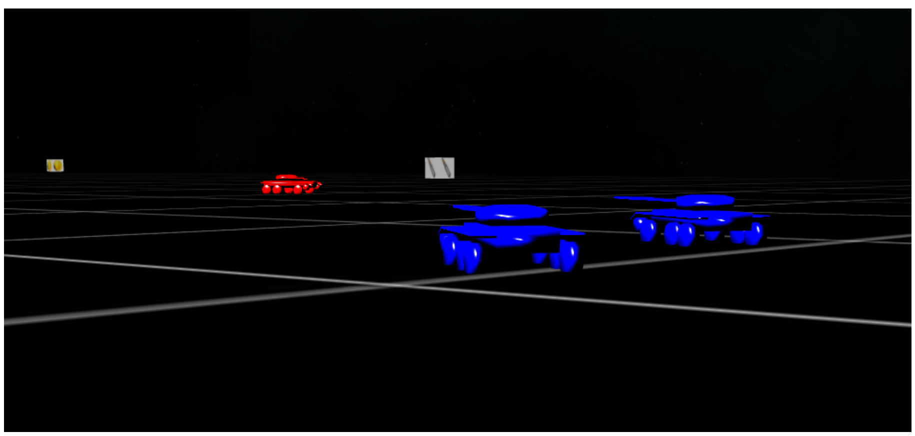


Figure 15: Screenshot of the game scene.

Move continously around the map. Keep turning left and right to avoid opponent cannon balls and never stop moving for any reason. Avoid being exactly in front of the opponent tanks as this will probably cause their cannon balls to hit the player tank. Remeber: your tank starts with 10 life points, so 5 cannon balls are more than enough to hunt you down. Always focus of picking the loot boxes. Aim at the point loot boxes first, but if you see your life to go below 4 points, change your objective and try to pick some health loot boxes.
Hunt down few opponent tanks if you can,exspecially on hard difficulty, but never stop hunting the point loot boxes. Keep in mind that, differently from you, they have infinite cannon balls and machine gun bullets, so, sooner or later, they will manage to destroy you expecially if you play careless .
The cannon is certainly the most effective weapon, but you starts with just 10 cannon balls. Moreover, you have to consider that the cannon balls are heavy and are affected by gravity. Then shooting with the cannon will make you cause

more damage, but you will be less precise. On the contrary, the machine gun is a weaker but indeed a more precise weapon. Mix them to achieve letalithy.