# 1 Introduction

The aim of this project is to implement a very simple game using Babylon.js. The game uses simple techniques to animate a hierarchical model representing the tank. 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. 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 (classes are widely used).

# 2 The framework

Babylon is a very powerful framework which offers high level API to create 3D graphics programs that run on web browsers. It is very easy and guarantees GPU hardware acceleration freeing the user from the burden of directly program the shaders.
Each scene is represented by a BABYLON.Scene istance. The scene is an object that contains several information that are needed to represent a level or, more generally, a scenario in a 3D graphics application. To fully exploit Babylon modularity, we assign to each level a scene object. A scene keeps track of all the meshes that must be rendered, of the textures applied to each of these meshes, of the lights and of many other important objects.
Before actually rendering the scene, we may want to load all the imported models and the audio files. Since all the models are asinchronously loaded and istantiated, avoiding to do this may cause the player to experience a partially rendered scene at the beginning of the game.
To synchronize the model and sound loading process we use an efficient mechanism provided by Babylon: the asset manager.
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. We then 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.

# 3 The environment

## 3.1 Description

The environment is very simple. It is a flat 2000x2000 placed somewhere in the space that represents a very simplified version of an arena. In this arena two team of tanks fight to the death: the **red** and the **blue** team tanks.
The player plays as a red team tank and it is the only member of its team. He is able to move in the map with its tank collecting bonuses and points from the floating loot boxes placed at certain positions in the ground map. 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 do not need to worry about terrain irregularities.
The ground is equipped with a very simple physic mechanism that allows the object to dinamically interact with it. It has no borders so the player needs to be careful not to move outside the map. If it happens, it is simply teleported into its spawn position and thus he is able to continue playing.

## 3.2 Implementation details

To correctly render the scene, Babylon js forces the program to create a camera: in order for a scene to be rendered properly, at least a camera must be specified. Specifying a camera allows Babylon to keep track of the *world* and *model view* matrices for example, to correctly render the mesh vertices. To render the scene, then, we created a **follow camera** object.
The follow camera is a powerful tool available in Babylon js framework. The camera needs a target mesh to follow and can be tuned with a variety of parameters. It is ideal to create a third person shooter camera or an *external* vehicle camera. In order to make the player able to drive the red tank, we set the tank as the target of the follow camera [1]. The API also make possible to attach to the HTML canvas a camera object so that the user can directly control it interacting with mouse and keyboard arrow buttons. In order to make the player directly able to control the tank, we attached to the canvas the tank follow camera.
Moreover HTML 5 provide a useful feature when comes to program an interactive game: the **pointer lock**. With this feature, the HTML canvas are able to *capture* the mouse pointer so that the player cannot click outside the game window (for example closing the browser window by mistake terminating the game). The user can obtain 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. The boxes are texturized using the advanced *cube texturing* mechanism provided by Babylon. To attach a cube texture to an object Babylon js needs 6 images which corresponds to the six faces of the cube. 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 a tag which must be one of (px,py,pz,nx,ny,nz).
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 convention that is specified in the official documentation.
Each box is animated thanks to a simple but very effective **key frame animation**. A key frame animation in Babylon js is represented by a special object

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

(Babylon.Animation) which is associated to a given mesh, which is the mesh to be animated. The animation is provided with a list of javascript objects in the form

{frame: value}

where the **frame** is a frame number and the *value* field represents a possible value for the given property of the object at that frame. In this case the boxes are equipped with two key frame animations: the first one for its position in the scene and the second one for the value of the rotation angle along the $y$ axis. The animation has 100 frames and runs at 10 frames per second so that the player has an impression that the boxes **floats** above the ground. This type of animation is inspired by the **mario kart** floating boxes that contain the various bonuses for the players that take part into the scene.
The boxes are also associated to a Babylon action trigger. When the red tank, which from now on we call the **hero tank** intersect one of the boxes, it will disappear and the associated bonus is collected by the player. The corresponding box mesh is destroyed to free some memory. We paid particular attention to free memory whenever was 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 must not be rendered by the scene engine is a huge waste of computation 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 and the corresponding vertices are recreated and passed to the GPU.
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. Collecting points is vital to complete the game and obtain the **win condition** as opponent tanks will keep spawning trying to hunt down the player.

- 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. Machine gun can be used to kill the opponent tanks.

- A **cannon box** represented by a big cannon ball. Collecting those boxes is again not crucial 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 much more damage than the machine gun bullets.

- A **health box** represented by a big hearth. Picking the health boxes could be a good strategy to play a safe game since the opponent AI tanks are very aggressive and constantly try to shoot cannon balls towards the player tank to hunt it down. Collecting health boxes guarantees the player

2 additional life points until a certain maximum amount of life points is reached.

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 is provided with a texturized 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

Before implementing the various tank features, we need to talk a little bit about the model. The imported mesh is a collection of **independent** boxes. It means that the tank is not a hierarchical model and so the structure must be given at runtime.
For simplicity I decided to set the mesh named **"Box"** as the root of the model. All the pieces, apart from the cannon, are set as *children* of the root whereas the cannon is set as a child of the turret. Babylon js implements a simple mechanism through the *addParent, addChild, removeParent,removeChild* interface which istantiate a hierarchical tree like model on the fly. 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.

## 4.2 Animating the turret

One of the key animations in the game scene is the animation of the turret. In this case, no key frame animation is used since we want to rotate the turret differently from time to time following the mouse movements. The turret animation is handled in a separate function. 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. Babylon js offers a very simple function to obtain the position that is right in front of the camera at a given distance: *camera.getFrontPosition(distance)*. This function returns a vector which represents the direction towards which the follow camera points at. We call $\beta$ the angle between the position of the camera and the position of the frontVector. To obtain this angle we need make few considerations. The difference between the camera front position vector and the camera position gives the vector that joins them. The angle between the camera position and the vector difference is $\beta$ the angle we are looking for. We know also that the x component of the difference vector is the sine of the angle

$\beta$ and the z component is the cosine of the angle $\beta$. To obtain beta we simply need to solve this simple non linear system:

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

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 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. Once we obtain $\beta$ we simply rotate the turret along the y axis by $\beta$ radians. Since the turret is a parent node of the cannon in the hierarchical model, rotating the turret will inevitably rotate the cannon that is the final effect we want to achieve in this animation

### 4.3 Dealing with collisions

Babylon js offers a very simple collision system that avoid two meshes to fully intersect. To enable the collision system for a given mesh we simply have to set a boolean variable for the mesh object. So not all the meshes have the collision system enabled. In order for the tanks to deal with collisions we create 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. Whenever we move the tank, we also move the bounding box accordingly [2].

### 4.4 The hit bounding box

In order to handle cannon ball hit we use another smaller bounding box which form now on, we will call the **hit bounding box**. The hit bounding box is much smaller as we do not want the player to experiment any phantom range damage 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. In the final version of the game, both th two bounding boxes will be invisible, but still be rendered and considered by the Babylon js collision engine.

## 5 The player tank

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

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

task. There are a lot of high quality meshes 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. Importing a highly detailed model of a world war 2 tank and not being able to animate its wheels is a quite disappointing. Luckily, I 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 need to define some basic tank environment interactions. The ground is not equipped by any *friction force* so we need to find a way to simulate acceleration, braking and steering.
The first thing we did is to provide the player with a more *game like* movement system: instead of using the arrow keys to move the tank or to use left mouse button, we address WASD for that purpose. Achieving this is quite straightforward through javascript event listener. First a global *WASD handler* object is istantiated to keep of which button of the WASD grid is pressed. Secondly, we enabled a javascript event listener on the WASD keys that trigger a boolean flag in the handler when the 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* Babylon js built-in functions and rotation is applied using the *rotate* function of the Babylon.Mesh object. This is a very simple working system but it is not enough. We want the tank to actually accelerate up to a predefined top speed when W is pressed and we want the tank 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 speed of the tank. On the contrary, when S is pressed the speed parameter is decrease by the same value. Once the speed parameter becomes negative, the tank starts moving backwards. Pressing A or D does not affect the tank speed parameter. If the player stops pressing one of the WASD buttons the tank immediately freezes.
Moving the tank should cause the wheels to rotate as they get in contact with 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 accomplished by applying a red colore 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 two team 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 cannon ball source the program retrieves both the position and the orentation of the tank cannon and update 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 cannon ball.
In order to simulate a fire ball effect we used a properly tuned particle system to simulate a fireball.
Babylom js allows the user to use a special object called particle system. It 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. A particle system object 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 could be considered as a very small cube. So it is possible to set a colour and a texture to it. The emit rate specifies how many particles are generated per second from a given source. Babylon allows the user also to specify a minimum and a maximum lifetime for particles.
To simulate the fire ball we used also a *sphere emitter* object which set a spherical emission region centered in the emitter mesh. The maximum lifetime of the particles is set to a very low value (few milliseconds) and a fire texture has been applied to them. The emitter source of the particle system is the sphere representing the cannon ball and we set the emission region to move togheter with the cannon ball.
The ball can react to other meshes in the scene through the physics engine. In order to throw the cannon ball toward the direction pointed by the cannon ball we simply apply an impulse. The strength of the input is a parameter defined at runtime and can be changed. The impulse is applied to the center of the sphere mesh and points towards the direction of the cannon turret. The physics engine, then, applies an earth like gravity ($g_y = -9.81\ \frac{m}{s^2}$) to the cannon ball resulting in a parabolic trajectory. The ball reacts with the environment in the following way:

- If the cannon ball intersect 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 intersect the ground it will explode and cause no damage to any of the tanks. After the explosion, the cannon ball will disappear, the particle system will be stopped and the mesh is disposed.
- If the cannon ball does not hit anything after 4 seconds it disappears and the corresponding mesh is disposed for efficiency reasons.

The interaction between the cannon ball and the scene objects is regulated by Babylon js triggers. After the cannon ball is shot an action manager is istantiated through the Babylon.ActionManager class.
In Babylon js the action managers allows the programmer to associate to the scene meshes different types of actions triggered by asinchronous events. There are many types of triggers and thus it is very unfrequent the case in which the programmer needs to define custom triggers. The trigger should be considered as a simple test which returns a boolean value. If the value is true, the associated action is performed. There are many type of actions defined by the Babylon js API: some might set certain values to an object property, some others may execute code that will alter the status of the whole scene.
In order to properly trigger the explosion of the cannon ball once it hits an opponent tank, the action is an ExecuteCodeAction. This code associated with

this action 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 particle directly from the opponent tank body to simulate the explosion. Moreover, it will reproduce a sound of an explosion which has been loaded by the assets manager before the scene is rendered.

### 5.2.2 The machine gun

Very similar to the cannon ball, it represents the secondary weapon of the tanks in this game. The machine gun weapon is very easy to use but deals less damage than the cannon balls.
To simulate the machine gun bullets we use Babylon js *rays*. In Babylon js a ray is, as the name suggets, a line in the context of a scene. To throw rays in your scene, you need to specify an origin, a direction and a length. By default, rays are invisible, but you can make them visible defining a *ray helper*. A Babylon js ray intersects objects and you can define actions that will be executed whenever an object is intersected by a ray using the *scene.pickWithRay* function. This function will set a callback that will be executed whenever a mesh intersects the ray. Obviously, we may not want this callback to be executed when the intersected mesh is, for example, the skybox. Fortunately, Babylon js pickWithRay function allows us to define precisely which meshes should be taken into account by setting a boolean expression. In this case, only meshes which belongs to opponent tanks should trigger this callback. Fortunately, those meshes are immediately spotted by inspecting their names. All the tank model meshes have a name that contains the string "Box" ("clone_1.Box14.Box15","Box7" etc ...), thus a simple method to pick only those meshes is to use javascript *regular expressions*. So a regular expression is defined. If the target mesh name is the format described above, then it matches the regular expression and when the ray intersects it, the ray callback will be triggered.
The callback is very simple. It decreases the target tank health and then performs a simple test: if the resulting health of the target tank reaches a value of 0, the target tank dies, otherwise nothing is done.

### 5.2.3 The tanks death

If the tank health reaches a value of 0, the tank must disappear from the scene. To do so, for each scene tank, we perform a simple test whenever a cannon ball or a machine gun ray hits it. The test is actually very simple and controls if the tank health is greater than zero. If it is not the case then a special death function is performed.
The tank death should be handled with care. You must be sure that, not only the model will disappear, but also that all the objects that are associated to it are released to free memory space. Then, before disposing the tank meshes we ensure that:

- All the tank particle systems are shut down.

- All the particle systems are disposed.
- Each of the two bounding boxes are disposed.
- The tank meshes are disposed.

Then the function has a different behaviour whether or not the target tank is the player tank. If the tank is the player tank, 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.

# 6 The opponent tanks

The opponent tanks are handled inside a loop that is executed at every rendering iteration. In this loop each tank execute 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 value, then the tank will switch weapon. Otherwise, it keeps shooting with the same weapon.
The opponent tanks are designed to kill the player. They will constantly follow the tank shooting with their equipped weapon. 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 handle the blue tank movement independently of the tank movement. We set the tank to always point towars the direction of the player tank and to move towards it with a fixed speed. To accomplish this, we compute a difference between two vectors: the tank *front vector* and the opponent tank *front vector*. The front vectors are two vectors that points towards the front direction of the tank. The tank front vector is always updated whenever the tank changes its orientation with respect to any given axis (rotates). The difference between these two vectors returns the vector that joins the two front vectors. The goal

of this operation is to obtain the angle $\alpha$ between the opponent front vector and the difference vector. To accomplish 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 know that

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

Once we compute $\alpha$ we rotate the blue tank along the y axis by $\alpha$ radians, replace its front vector with the difference vector and move the tank in the direction of the new front vector.
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 simple user manual of the game. It is a simple third person arena game which involves only a single minigame.

## 7.1 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. 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. In order to make sure that you control the tank turret press the left mouse button on the canvas to have the mouse pointer locked inside the game. Move the mouse to rotate the turret. Make sure that the turret points towards the enemy tanks, as otherwise you will waste ammo.
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". To switch to the machine gun, press "2". You can always terminate the game by pressing "m".

## 7.2 Player guide

The game itself 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.
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 you. Remeber: your tank starts with 10 life points, so 5 cannon balls are theoretically sufficient 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.
The cannon is certainly the most effective weapon, but you have less ammo compared to the machine gun bullets. Moreover, you have to consider that the cannon balls are heavy and are affected by gravity. 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.