Master of Science in Engineering in Computer Science

Interactive Graphics

a.y. 2019/2020

War of The Robots

Authors:
Andrea Franceschi. 1709888
Daniele Cantisani 1707633

September 27, 2020

Sapienza Università di Roma

Contents

1 Introduction

1.1 General Idea

The idea was to create a level shooter game where the strength of the enemies increases with each level of the game. The goal is to complete the final level.

1.2 Rules

Our character in the game aims to shoot all the enemy robots that come at him. The enemies come from four initial positions (white disks) placed in the game field.
In the main menu we have the possibility to choose between three difficulty levels (easy, medium, hard) and between two playing fields (called lab and moon). After pressing start we can start the game.
In the game there are 5 levels to pass. Depending on the level we are in and the difficulty chosen previously there are parameters that change the strength of the robots enemies such as the speed of the enemies, their life number and how much damage they take away.
The hero's normal shot completely destroys the smaller robots but it takes more shots to destroy the bigger robot. Instead, the special bullet destroys any enemy that is in its trajectory. In the first level we start with only one special bullet available and in each level we will have an extra special one that we can use at any time of the game in the most difficult moments.
If the robots reach the hero and the two characters collide then the hero loses his life. Depending on the level the robots can take 1-2 or even 3 lives for each attack.
If the player finishes all levels with at least 1 life left then he has won the game. If the player loses all the lives before that happens then he has lost the game.

1.3 Commands

The game is expected to be played on a computer. The use of the keyboard is essential to participate. The hero's movement is done with the WASD keys. W we move forward, A we move left, S we walk backwards and D we move right. The simple shot can be used by pressing the space bar while the special shot can be used by pressing V.
All these keys are designed to be used with one hand while the other hand holds the mouse. In fact, by clicking and moving the mouse we can adjust the camera and the shooting view.

2 GUI

The Babylonjs GUI library was used to create the interactive user interface in scenes.
Babylon.GUI uses a DynamicTexture to generate a fully functional user interface which is flexible and GPU accelerated.

2.1 Menu GUI

When we open the game we find ourselves in a Menu Scene. Here we can choose the Lab or Moon field (Lab is the default field) and also we can set a general difficulty of the levels of the game with three available modes that can be clicked through Radio buttons (the default difficulty is easy). When we made our choices we can start the game by pressing the **start** key.
From the menu we can also access the commands page or a project presentation page by pressing the **commands** and **about us** buttons.

2.2 Commands and AboutUs GUI

In the command scene there is an explanation with all the commands of the game. It is essential to read the commands carefully before starting the game. With a button we have the possibility to return to the main menu page.
In the About Us scene there is a brief description of who we are with our names and freshmen. Also from here we can return to the Main Menu with a button.

2.3 Game GUI

Above left there is the life of the hero. When life ends all his cells, the player loses the game.
At the top right is marked the level we are in. When the hero manages to pass a level, a sound of victory accompanied by a "Level Up" image appears on the screen for a few seconds before the next level begins.
Next to the layer icon there is a keyboard icon. By clicking the L key as suggested we can see all the commands to play the best game.
In the lower left corner are marked the remaining special bullets. In case there are none left, the icon with the number disappears.

2.4 Win and Game Over GUI

At the end of the game if we have won we have the possibility to return to the main menu through a button.
When we lose there is an additional button. In fact, we have the possibility to restart a new game directly from there.

3 Enviroments

All static scenes (Menù, About Us, Win, Game Over, Commands) are composed by a GUI with an image that cover all the screen, so the only environmental component needed is one camera. All these scenes have as a camera the UniversalCamera on which is displayed the GUI.

3.1 Game Lights

In the game scene we have defined two kind of lights: one Directional Light and one Hemispheric Light. The Directional Light emits light from everywhere in a specifyed direction, with an infinite range. This light can be seen as the sun light that hit the earth with apparently parallel rays. In our code the Directional Light is defined in the direction (0, -10, 0) so that hit all the meshes from the top giving them a shade. The light is defined to be completely white. The Hemispheric Light is a light that simulate an ambient enviroment light. Is defined by a direction (up towards the sky) and in our case is used to increased brightness of some meshes.

3.2 Game Camera

The playing camera is defined as orbital camera called ArcRotateCamera. This kind of camera points towards a given target position and can be rotated around that target with the target as the centre of rotation. In our case this camera point to the character and can be rotated with the mouse. This is possible because we have attached the camera controls to the canvas. In addition we have setted some parameters to improve playability like a zoom limit and an up ad down rotation limit.
This camera has been defined as if it was enclosed in an ellipsoid that will check the camera collisions.

3.3 Game Textures

A lot of different textures have been used to give the game a good graphics. In paricular we can see the ground of the moon scenario.
That ground is defined as a mesh with one texture loaded from an image. In particulare the texture was created from a lunar surface image provided by NASA. The texture is applied to the diffuse and specular property of the ground so that is is also visible in light reflection.
Different textures have been used for the robots. The wheels of both enemies is a single texture of a cylindrical piece of steel. When applied it gives the idea of a kind of tire.
For the smaller robot it was given a more metallic off effect with a gray texture.

For the larger robot the textures are different for each part of the body: The white head have an opaque texture, the central body have a decorated metal texture while a more shiny metal for the arms and the underside of the body. Two others important textures to take into accounts are the textures applyed to the box that simulates the sky and the horizon. This box is called SkyBox and is created by apply to it a CubeTexture. The latter is built by composing six continuos images one for each face of the cube. In our game we have two different CubeTexture one for each ambientation.

4 Imported Meshes

4.1 Hero: BrainStem

The main character is an imported model provided by Babylonjs. The name of the model was Brain Stem and it is also the name by which we called our hero. The 3D model of the hero is available on the Babylonjs website in the *Graphics Library Transmission Format* (.gltf).
In order to better import the model without predefined animations, we used a 3D graphic program (Blender) to transform the model from the starting format into the (.babylon) format. In addition we used Blender to change all the materials (and therefore the colors) of the parts that compose the robot.
The imported model is composed by a Skeleton and by a lot of meshes. The Skeleton is a common structure used to animate models composed of multiple meshes. In particular it is composed by a hierarchy of bones each defined by a name, a parent and a transformation matrix. Each mesh of the hero model is related to one bone so that by moving a bone all the meshes attached to it will move.
Immediately after importing the model its components have been positioned and scaled so that the character has an attack position and the right size.

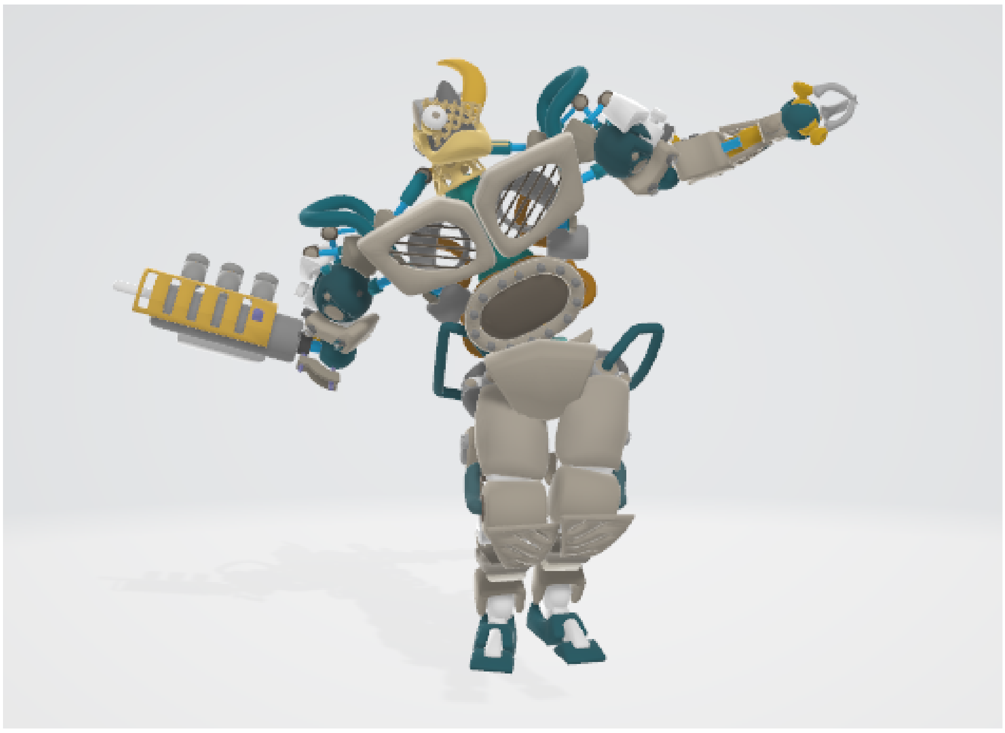


Figure 1: Mesh before importation and modification

4.2 Ambientation: Laboratory

Another imported model can be found in the ambientation called Laboratory. The whole scenario has been realized thanks to a model found online called Space Station.
Initially the model was in the *Object File Wavefront 3D* (.obj) format with the associated *OBJ Material Template Library* (.mtl) file for the color and various materials of the meshes.
The file with the materials cannot be imported so we have converted the two files into a single model with the (.babylon) format using Blender.
The whole scene is placed inside the imported model, so many meshes that made up the external setting have been disabled because they are not visible during the game. Also some internal meshes are disabled to build an open space. Otherwise some meshes placed at the center are animated to compose a futuristic hologram.

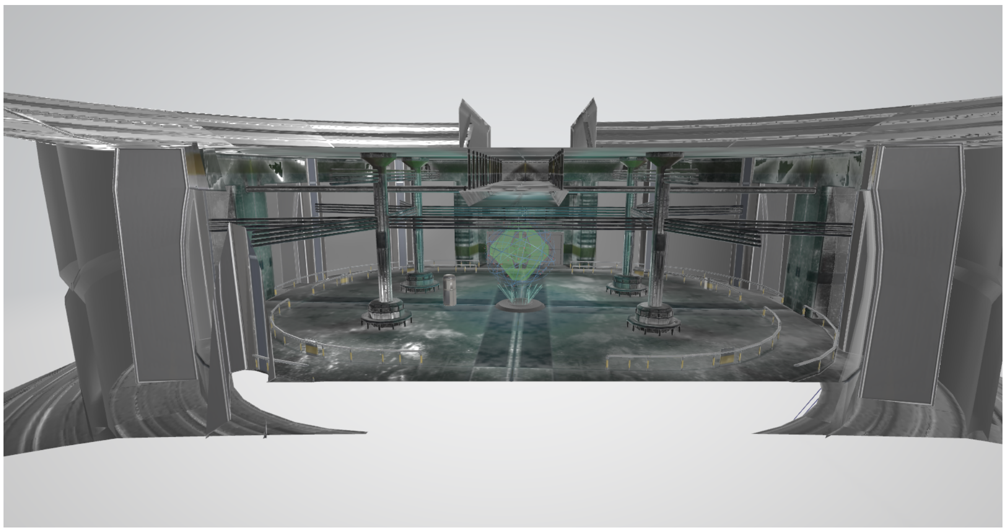


Figure 2: Mesh before importation and modification

4.3 Ambientation: Lunar Buildings

The moon ambientation has been built by importing a single model composed of several structures called ScifiFloatingCity. Also this model has been downloaded in (.obj) format with the associated (.mtl), so we had to convert it in the format (.babylon).
Initially the model was a city with very scattered buildings and a round structure connected by a street. To generate the scenario of the game, some changes were made:

- The street that surrounded the buildings and led to the ”square” has been eliminated
- The round structure in the center of the square has been moved to locate it in the center of the buildings
- Both the buildings and the round structure have been resized to compose an open space
- Some buildings have been cloned in order to reach a denser setting of buildings

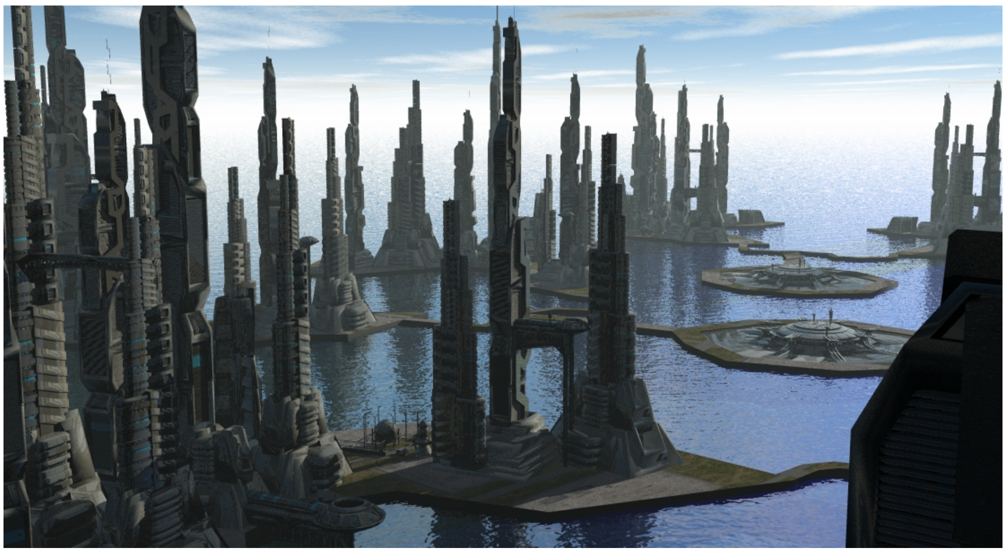


Figure 3: Mesh before importation and modification

5 Hierarchical Models

To meet the requirement of having a complex hierarchical structure we decided to create the enemy robots all through hierarchical structures.
The robots have been formed by creating numerous meshes and connecting them together thanks to the parent attribute.

5.1 Little Robot

As for the smallest robots, the father of everything is a cylinder (called **original**) to which is connected the head (a sphere), the eye (also a sphere), the body below (a cylinder) and the feet, that is the wheel with which it moves (a sphere). The arms were the most complex part. The arm consists of a first part, a second part and a conjunction (elbow) between them. A small sphere is connected to the second part of the arm from which four cylinders branch out. Other four cylinders branch out from the first four. The idea was to create a sort of dangerous double clamp for a possible opponent.
All the arm has been created exploiting the property of father and son and hierarchical exploitation. For optimization reasons after creating it, we have combined all these Mesh of the arm into a single one in order to improve the FPS and spawning. The single merge mesh is the child of the initial mesh (original). The left arm has been formed in the same way but with mirrored variables. The arms are also rotated to assume an initial position. Then with the animation they will move from the assigned position (see Animation Chapter).

5.2 Larger Robot

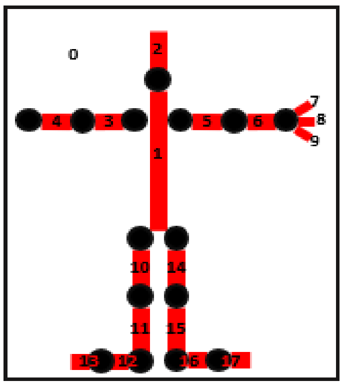
The larger robot has been developed following the same reasoning as the smaller one, improving some aspects regarding optimization. In fact the body is a set of cylinders superimposed one on top of the other (one child of the other) but at the end of their production they have been joined in two bigger meshes in order to improve their performance.
Under the body a final cylinder anticipates the sphere in which the robot moves. These two pieces are external from the joined mesh because they were important in the following animation (see Animation Chapter).
Also the arms have been completed following the lessons learned from the previous robot. The single pieces have a local hierarchical structure, and then they were joined together in a single mesh and connected as children of the main mesh. The end of the arm ends with a pointed hammer.
Like those of the smaller robots, the arms have been rotated to assume an initial position that will be modified later in the animation. As before, the left arm is equal to the right one with some mirrored variables.

6 Animations

Almost all animations have been made by using movements and rotations performed in the render loop cycle. Others animations such as the movement of the bullets or the hologram of the Laboratory have always been created by moving / rotating meshes but inserted in a function that is able to execute the commands at each frame using the ActionManager provided by Babylonjs.

6.1 BrainStem Animation

As previously said the BrainStem character is an imported model in the format (.gltf). The 3D models of this format can include predefined animations that we have eliminated after the transformation in the format (.babylon).
The imported model has a Skeleton structure shown in the following image.



0. Full Body	9. Claw 3
1. Head	10. Right Thigh
2. Torso	11. Right Leg
3. Right Forearm	12. Right Heel
4. Right Arm	13. Right Toe
5. Left Forearm	14. Left Thigh
6. Left Arm	15. Left Leg
7. Claw 1	16. Left Heel
8. Claw 2	17. Left Toe

The upper part of the skeleton was moved immediately after importing the model to bring it into the attack position (1-9 Bones).
The other bones (10-17) are moved in order to simulate the walk of the character. All the animation of the walk is done in fifty-seven steps that starting from 0 are increased at each cycle of the main render loop. At each step every bone of the legs is rotated using the *rotate()* method taking as parameters the local axes of the whole skeleton so that if the skeleton is rotating also the bones rotate with it.

6.2 Robots

The robot animations were developed using the functions rotate() and translate() in single meshes several times to form the movement of particular parts of the body.
Both robots have been applied a continuous translate in the time and as argument there were assigned a direction. This direction is exactly the position of the BrainStem (Hero) at the live time. In this way the robots always move in

the new direction in which the hero is.
To give the appearance of movement, a continuous rotation has been applied to the robot's movement spheres (the feet). The robots give the idea that they are rolling towards their target. The rotation takes place around the local z-axis of the robot. Although the robot turns in the direction of the Brainstem, the rotation axis being local does not change. In this way the correct rotation is maintained.
Robots have two different types of animations on their arms or body.
As for the small robot, a rotation is applied on the small sphere that is in the middle between the second part of the arm and the beginning of the grippers.Since the sphere is the parent, the whole structure of the grippers rotates giving the idea of an attack.
As for the larger robot, the animation is quite more complicated and consists of four parts: at the beginning the robot lowers the left arm (90 degrees downward rotation), according to the body it rotates 360 degrees, then the arm gets up and now the right arm is lowered. Finally, the robot rotates again 360 degrees in the opposite direction of the previous one.
Since in rotations the whole body rotates (since the rotate() command was given to the father of the hierarchical structure) it was necessary to give a rotation of equal and opposite force in the underlying parts of the body to ensure that the wheel does not rotate along with the rest.

6.3 Bullets and Hologram

There are two kind of bullets, and both move in the same way. Like said before we have used one function that execute one command at each frame. In this case the command ridefine the bullet meshe position by following a specified direction vector that is eaqual to the character direction.
Is important to specify that the movement of the projectiles takes place until the bullet reaches its maximum range and then it is destroyed. For the hologram the same principle of movement has been used but in this case instead of a change of position at the meshes that make up the hologram sequential rotations have been made.

7 Interactions

There are a lot of interactions in the game like bullets-enemies, player-enemy, player-environmental objects, player-ground etc. These interactions take place in two different ways:

- The use of a plugin system for physics engines (Cannon.js)
- A between meshes collision system provided by Babylonjs

7.1 Physics Interactions

As said before we have used a physic engine library so that we give at all the entities of the game some physical parameters. There exists many plugin for physic but we have chosen Cannon.js and to enable the physic on an object we have used a physics impostor provided by Babylonjs. In particular the function used is:

BABYLON.PhysicsImpostor(object: *IPhysicsEnabledObject*, ***type:*** *number*, ***options:*** *PhysicsImpostorParameters*, ***scene:*** *BABYLON.Scene* ***);***

Where the important parameters are:

1. An interface that can be whatever Babylon object that has at list the parameters of position and rotationQuaternion
2. A type that refers to the shape to which the physics must be set (Sphere, Box, Plane, etc.)
3. All the physics option that a body with mass must have (mass, friction, restitution, etc.)
4. The babylon scene to which the impostor should refers to

In our case all the impostor gives at the meshes a mass that is very big for meshes not movable and similar to reality for movable bodies. Instead all the restitution parameter is set to 0 so that when bodies collide there is no bounce. This method is mainly used to prevent a body from entering into another body. In example it is used for the clash between ”enemy” robots and BrainStem or between any movable entity of the game and the various environments.

7.2 Mesh Collision Interaction

The second method used to make interaction between entities is the Mesh Collision. The collision between meshes is monitored through the function *intersectsMesh()* that takes as input two parameters: the first is the mesh to

be checked and the second is the precision of intersection.
Of course, this function checks only once, so to check if a mesh meets another one during the game you have to call it either in the loop render, or with a SetInterval with a very low repetition time.
This is the method used to manage the life of the various characters (including bullets). In fact, when a bullet hits an enemy or part of the environment, the function is activated. In the case where the bullet hit an enemy, the bullet is destroyed and the enemy lose life. Instead if a bullet hit one of the wall it is just destroyed.
This function is also activated when one ”enemy” robot hit the character. In this case the intersectMesh() is inserted in one setInterval that gives the time between the enemy hit (enemy hit ratio). During the game is possible to see that the little robots hit faster but make minus damages, instead the big robots hit slower but make big damages.

8 Optimization

One of the fundamental part of our project is the continuous research during the development of a maximum optimization of the scenes so that the game could play even on low performance devices. In order to reach our goal of a frame rate around 60 fps we have done a lot of manual optimization plus the inclusion of the SceneOptimizer tool provided by BABYLONJS.

8.1 Manual Optimization

The following is a list of most important optimization that we have done:

- Both the IntersectMeshes method and the Physic used for the interactions are much more performing on simple meshes like boxes. To increase the interactions performances we basically have included every complex mesh (like BrainStem, Ambientation, Robits, etc.) in other boxes meshes. This has greatly increased the performances of the interactions but decreased the general performance of rendering because in the scene there were many more meshes (especially those used to define the limits of the ambientation).
- In order to reduce the impact on the performances of the meshes in the setting (perimeter walls) we have "freezed" the static meshes World Matrix that specify the position, rotation and scaling of one mesh. This because the World Matrix is evaluated on every frame and this is computationally heavy.
- Another optimization used on the walls that delimit the environments was the unfold of the vertices and the stop of using the indices to reuse the vertices by faces (thing done automatically by Babylonjs).
- We also had to do a good optimization on "enemy" robots because they are composed of many meshes and therefore with a high render weight. In order to do that mush of the robots meshes are merged in one so that the final robot result as composed by a few meshes.
- The computation of the light on the meshes is very heavy. Babylonjs allow to arbitrarily exclude some meshes from the light impact. To increase the performances we have excluded some meshes from the directional light where it is not mush visible, like on the bullets and on the skybox.
- One small but overall important optimization is the creation of many equal meshes (like walls) by using the *clone()* function and so reducing the draw mesh calls.

8.2 Scene Optimizer

In case all the optimizations we thought were not enough to reach the target fps rate, we have also inserted in the game scene the SceneOptimizer.
This is the last choice because it increases fps by degrading the rendering quality at run time without any possible control by the user.
We have created the SceneOptimizer in such a way that it activates only when the device cannot reach 60 fps. The optimizer check the fps status every 500 ms and applies different levels of optimization according to need. The levels defined by us are:

Level 0: -Disabling shadows -Disabling lens flare

Level 1: -Disable post processing -Disable particles

Level 2: -Tries to reduce the size of render textures

Level 3: -Disable render target

9 Sounds and Music

To make the experience more realistic we decided to add music both in the background or in specific moments of the game. We used copyright free Ashamaluev-Music music for the background sounds in the main menu and in the game.
Other sounds are the sounds of the gunshot (both the small one and the special bullet), the sounds of victory or defeat in the game and the level change.
In addition, a metallic noise has been inserted to simulate the main hero's walk.
The sound is made as you move with the WASD keys.
Music:
Victory - Epic Motivational Background Music / Action Cinematic Music
Virtual World - Sci-fi Background Music / Documentary Music Instrumental

10 Libraries

We have imported many libraries of babylonjs to perform many basic functions such as using physics for characters, importing external objects and models and much more.
Here are all the imported libraries:

- babylon.max.js
- babylon.inspector.bundle.js
- babylonjs.loaders.min.js
- babylonjs.materials.min.js
- babylonjs.postProcess.min.js
- babylon.gui.min.js
- babylonjs.proceduralTextures.min.js
- babylonjs.serializers.min.js”
- babylonjs.serializers.min.js
- cannon.js