Interactive Graphics Project: Escape Game

Biagio La Rosa

1 Introduction

For the project of Interactive Graphics I developed a small game called Escape Game. The goal of the user in the game is to open the door of each room and pass to the next room untill the game ends. During the developing of each room I tried to explore and show all the main features of BabylonJS library based on the features of WebGL studied during the course. BabylonJS is a rich library at hight level that provides some advanced tools as physics engine or newer filters and techniques that I avoid to use in my game because they are outside the program of classroom or includes/replaces too many basic functions. The functions used and described in this report basically allow the developer to instanciate the objects fixing its general properties without care about the coordinates of each vertex,matrices and transformations. For example thanks to the animation object you can develop each of them in easy way without rewrite code for each function and setting several limit variable and so on as in WebGL. Also the creation of interfaces with GUI is easy and more and more clean than classic HTML base.
The basic elements of each room are: the scene that is the container of all the things that the user can see, it declares the "rules" of the world and represents the rooms of the game; the objects that are meshes with various shapes and colors, some of them dynamic , some static; lights that interact with the objects in the scene and change the way in which they are seen by the user and camera that represent the eye of the user. Each of these basic elements can be enriched by several elements or tecniques that I'll explain in this report.

In order to use the library, the first step was download the library with all components available at https://babylonjs.com, import it into index.html and load the eninge into the javascript file. I organized my code into 4 javascript files: game.js contains the core functions of the game managing its flow and the gui associated; structures.js contains the class for the three room in the game and the auxiliares functions for their creation; object.js contains all animated and static objects as birds, boxes, weapons and keys; animations.js contains all the animations used in the game.

2 Classes

The classes represent objects of the game that have some properties accessible from outside and they are implemented as javascript functions. The classes developed are: Door, BasicStructure, CameraGame, Room1, Room2 and Room3 included in the file structures.js and Bird, Box, Key and Weapon inside the file objects.js.
The BasicStructure class builds an empty room with 4 walls, a door, the floor and the ceiling. You can set the dimensions of the room, the dimension of the walls and the ones of the door; you can also declare if the left and right walls have spikes as in the room2. In that case the system call a createSpikes function that try to fill each side wall with spikes made by cones (created using the MeshBuilder.CreateCylinder method). The wall that contains the door will be formed by 4 objects with a parent-child relationship: the door, the wall on the left of the door, the wall on the right and the wall on top of the door.
The Door is defined by its position and its type. Currently two types are supported: wood and magical type. The wood door is associated with a bumpTexture and a standard image texture while the magical one has a Noise Procedural Texture. The difference between the two types is visible also inside the method open of the door: they have two differents open animations and the wood door reproduces also a sound when it is opened. Both the types have an hinge but only the wood door exploit it in the animation.The room classes and the camera class are describe in other section of this report.
The box class represents the tnt boxes and wooden boxes of the game. Each box requires a shadowGenerator (described in the section Shadow) in order to add and remove shadow for the current box. The class has a method called onDestroy that performs some operations based on the type of box: for tnt box there will begin the animation of explosion and the game ends while for wooden box a particles-based animation will begin and the objects inside box will fall down to the floor. Alongside the animation also the shadow of the boxes are removed from the shadow generation when a box is destroyed. The class stores the objects inside him in a vector and presents also a method that return the position on top of the box usefull to positionate objects.
Key class and Weapon class are very similar. They have three methods: throw, pick and action. The first two call the animation for the throw action and pick action while the latter does different things: in the keys case the action method, activated when the user brings the key on hand and clicks the "e" keyboard button, checks if the pointer is on a Door object and if it is near enough; if this is the case then the action will change the room with the room linked by the door; in the weapon case the animation of weapon over the air will begin. Other elements of these classes are the mesh imported by the loader of BabylonJS, the texture, the name and some parameters that set the collision and dimensions. Note that both

use an imported mesh using the importMesh function that calls the SceneLoader of Babylon and imports the model specified in the path. Note that because this function is asyncronous all the operations, as the shadow generation, creation of class object etc, are performed inside the callback onSucces of the function. You can also specify a container of the object and the system will put the object inside that container.

3 Game's flow and events

In this section I am going to describe the general flow of the game, the functions that change it and the event listeners associated to the game. The game has several variables that together define the current status.
The var "running" is set to true when the game is started and false when the user return to the homepage; it is usefull for rendering process to decide what scene will be render; also the "currentScene" var is usefull for the render function because it stores the room that the engine have to display if the game is running.
The "pause" var is set to true if the user presses the "Esc" key by the method pauseGame() that stops all the current animations and freezes the status of the game blocking all the inputs for the camera; similar to the pauseGame() there are the continueGame() function that set the pause var to false and restars the animations and allows the inputs for camera and the startGame() function that resets the game status and changes the currentScene with the startingRoom (the room1).
The "hand" var will store the object on hand of the player. This var is used in several function to check what action the player can do. For example the throw action is available only if the hand contains an object.
The "action_in_progress" and "progress_animation" are two flags that avoid the use of some part of functionalities when a particular animation or action is in progress.
The "meshObject" dictionary stores all the objects of the scene and it is usefull to get an object by id or viceversa or to performs some operations. For example you can check what type of object you have in hand checking this dictionary using the id of the mesh into the hand.
Finally a series of event listeners are present in the game. In particular if the game is running you can interact with the game using the keyboard keys. For example if you click on a key or weapon the event listener associated to the click will pick the object and put it into the hand. If you press "f" when you are holding an object you can throw away the object thanks to an event listener associated to the keyboard key events. Similar event listener are available also in the menu for click on button or the key press of "Esc" key that shows up the pause menu.

4 Scene and rendering function

In order to show things in the screen of the user we have to call the function RunRenderLoop of Babylon Engine. Because it is called by the engine at each time, here we can declare all the components that must be shown at the current time. So we can put the operations that must be always performed and the rendering process of the scene.

The scene object is the major component of the system. Each scene contains all the objects that the player will see during the game. It is the effective object rendered at each time. The typical life cycle of a scene is the follwing: we start creating an empty scene (in our case a Room). then we create objects and their animations and attach them to the scene. At this point we are ready to show our room (or equivalently the scene) to the player using the method render() of class Scene. Note that you can impose to the system to do some operations each time before the rendering or forcing it to do these after the rendering process through the functions registerAfterRender() and registerBeforeRender().

In my project the scenes are the rooms, the homepage and the menu. At each time the system can display: the homepage or one room and the menu simultaneously. The passage from the home page to a room and viceversa in managed through the boolean flag "running" as explained before. The room that must be displayed is stored in a variable called "current scene". This variable assumes new value when the game starts taking the starting room as scene and when the player goes from a room to another through a door.
The architecture of the game simplify this latter case: each room has one or more doors. Each door has a property called "nextRoom" that stores the next room. When the player open the door the system automatically change the current scene with the scene stored in the property. The changing waits a fixed number of second (2) to be executed in order to allow to complete all the residual animations. This solution is needed because the rendering process is asyncronous and if you don't wait a minimum amount of time then it can happen a switch of scene in between an animation and this things can be a problem for the player of the game. The door of the last room has the "nextRoom" property set to null. When the changing function met this case then the system displays a message declaring the victory of the user. The behaviour is developed using the same tecnique used for the menu development (see section Menu below).

You can set several usefull properties of each scene object. In my case I enabled the collisions (as explained in the section "Collisions"), setup the gravity vector simulating the earth's gravity and changed the color of the background.

A last thing about the scenes is how to reset them. Each room has a function called restart that resets the scene when it is triggered. Each room implements the function based on its componets. Basically it restarts from the begin all the animations, moves all the objects to their initial position, moves the camera to its initial position and orientation, remove objects from the hand and imports or creates all the objects destroyed of the scene. This approach requires an ad-hoc behaviour for each room but I think it is the best way for reducing the performance impact and problems with asyncrony. In fact you have to store all the initial conditions usefull to restore their status for each object and manually check with if-else statements each object status (for object destroyable). An alternative approach could be dipose the scene that you want restart and all its components and then recreate a new one. But when you create a room you are implicitely creating also all the scenes reachable from it so this approach would increase the load for the engine.

4.1 Rooms development

A room is built by different functions(called from now on "type functions") based on the type of the room. The function returns the complete scene that will be rendered by the engine. The function starts declaring the scene and setting some properties, as desribed in the previous section, then it builds a basic structure. The basic structure includes a ground, a roof, a door, and four walls. Each room clearly has its specific dimensions. Given the basic structure the functions creates the textures (so each type function creates different textures for walls etc.) and assign them to the different parts as explained in the section of Texture. Inside the room the function creates lights, the shadow generator and the objects setting manually their positions and eventually their properties. Finally it is implemented the reset function described in the previous section.

If you want add some operations associated to the rendering process of the given room you can call the scene.registerBeforeRender() function or scene.registerAfterRender() function that perferorm the operations inside them immediately befor or after the rendering process. In the third room, for example, here it is inserted the function that moves pseudo-casually the birds, while in the second room it is inserted the control of the collision with the walls and the management of sound. These functions are used also in the menu scene described below in order to show the panel for dead or victory.

4.2 Menu

The menu of the game and the home page are built using the GUI functionalities of Babylon.js and they are displayed through the management of the scenes. In particular the in-game menu is put into a special scene that is rended alongside the scene of the current room. This behavior is possibile setting autoclear property of the menu's scene to false. This setting allow to show the scene among another putting one of the two in background (in my case the room) and the other in foreground. The homepage is a different scene rendered in place of the two previous ones when the game's flag running is set to false.
The construction of each screen starts with the declaration of AdvancedDynamicTexture object and the creation of the FullscreenUI. This object will be the general container of the GUI. Each element will be attacched to it through the function addControl(element) and removed from the screen with the function removeControl(element).

The in-game menu is composed by four buttons created by BABYLON. GUI. Button. CreateSimpleButton method and each of them is associated to an event triggered when the user clicks on it. The behaviour of the event clearly depends by the button. Typically after the click a new popup screen is displayed showing some information. These information are put inside a panel made by the element Rectangle of the library. The panel is a container, it is attacched to the screen and other objects can attached to it creating so a hierarchy.
For example in the keybinds menu the screen has three "children" the two columns of text created using TextBlock objects and the "resume" button. The advantage of this approach is that you can define position, dimension, etc. refering to the "parent" of the element. This allow to the screen to be more flexibile. So if you attach an element to the panel and set the position of it to the "top" it will be displayed to the top of the panel. The "popup" mechanism of a screen, previous cited, is realized simply removing objects from the hierarchy (which has as root the Advanced Dynamic Texture) and adding element to it at each time the user click on enabled button. Each button is customizable chaning the colors, the dimension, the trasparency, the corner and so on.

The buttons of the in-game menu are: "restart" and "exit" that, after to ask a confirmation in order to avoid missclick, they reset the current scene in the affermative case or change the current scene with the home page respectively. The "Keybinds" button shows the use of the keyboard's key and mouse movement associated to the respective actions inside the game. Finally the "Volume Setting" button allows the user to change the volume of game's sounds using a slider. These buttons are displayed using the menu_button function that return one of the three

type of buttons available in my game. The types are different typically for relative position respect to their container, form and use. The title, for example, is rendered using the type "menu_title" that return a button not-clickable positionated on the top border of the container and centered with a background color blue and with no trasparency enabled. The menu buttons instead has a trasparency and is positionated on the right of the screen.

The homepage is rendered when the game is in the state "not running" inside an if statement in the render loop. It is composed by a simple panel with 3 buttons: "start new game", "keybinds" and "volume". The last two buttons exibits the same behavior of the in-game menu, but this time it showing the info inside the panel of homepage. The first one, instead, set the state of game to "running" and restart the game in a such way the user wil start from the first room.

5 Lights

The lights are one on of the most important things in the scene. The rendering of a scene, the colors and illumination of the objects perceived by the user are directly influenced by their type and properties. Babylon provides the standard light types, spotlight, pointlight and directional light, and the hemishpheric type. I used all the types but directional light. Each of them has specific properties that reflects the classical equations in classical webGL. All of them are simply points in the space from where can there start rays of light. The final color rendered in the scene is a combination by the color of material and light. Note that the lights defined above influence only the diffuse and specular color of an object and I used only these two type of color. You can also define an ambient color in the scene and set the ambient color of the object to obtain another visual effect but I never used this functionality in my game.

For each light you can set up several properties. The principal used properties are: the specular and diffuse color, that set the color of ray that hit the object and the color of reflection; the range that defines how far the light impact the objects (so objects behind this limit don't receive light); intensity that basically change the brightness and direction for light that use it for their behavior.
In the first room I used a pointlight that emits light in every direction starting from the initial position. I put the initial position inside a sphere that tries to simulate a chandelier. The chandelier is a static hierachical object covered by a highlightLayer that adds a blur effect to the object and with a material with a yellow emissive color. The combination of these element return an effect similar to the normal light bulb in a room.

In the second room I used two lights, an Hemispheric light and a spotlight. The first one has an intensity low in order to produce a dark room, while the spotlight has a direction from roof to the ground and highlights a small part of the ground in front of the camera. Boh the lights has a red-like color that want to evoke the blood color.
In the third room finally there is only one spotlight, but this time its direction is set towards the camera along the z axis. In both the case (for the current and previous room) when you create your spotlight you must also provide an angle that define the cone of light and a number (the exponent) that define how fast the light decades inside the cone from the emission spot. Higher is its value more thight is the cone field of view at greater distance. I chose a low value in order to render enough part of the room at each time.

5.1 Shadow

When there is a light with a direction inside a room, then the objects should have a shadow associated with them. The Babylon Library creates the shadow setting some properties and instantiating a shadow generator object. This generator has the aim to produce a shadow of each object in its list. The list can be filled adding the object to its rendering list(using getShadowMap().renderList.push() or some helper function as "addShadowCaster"). The light is obtained simply blocking the light that hits the given object. Without the shadow, in fact, the light in BabylonJS passes through the object and hits also the object behind it: the shadows blocks this mechanism rendering the object more real.
Now that we have produced a shadow we have to decide where it can be shown. For each mesh that can show the shadow in its face we have to set the property "receiveShadows" to true. Normally you shoud setup to true all objects of your scene, but because each shadow implies a calculation you can decide to display the shadow only on the walls, ground and roof of the room mantaining a good visual effect and saving some power calculation for other things in your scene.
You can set several properties of shadow: for example you can decide if they would appear more solid or more soft, you can decide the dimension or the algorithm used to produce it. Clearly each decision has some impact in the system performance: softer shadows are slower than the hard ones. For my project I use the setting that allow more compatibility (so I avoided to use the newer filters) but at the same time the better effect.

An example of application is the third room. Here I associated a shadow generator to my spotlight and add the birds and the key of the room to its renderlist. Note that I add only the bodies of the bird to the list because I used the addShadowCaster method of shadow generator. This method adds to the list the mesh passed

to it and all its descendants so it allows to reduce the code length using only a line for each hierarchical object that has only one root element. For the filter I used blurExponentialShadowMap that returns blurred and soft shadow. Instead for the room one I used a more solid shadow that, in my opionion, better fits the pointlight used inside the room.

6 Textures and materials

The materials are another key component of the scene, in particular in rendering process of objects. Thanks to the material you can change the color or the texture of an object and set up how the object interleaves with the light. So the color of a mesh isn't a property of it but it is one of the material associated to the mesh. You can change the material of a mesh assigning your material to the material property of the mesh. The creation process of a material starts with the declaration of a StandardMaterial object with a name.

In order to change the color you can change ambientColor, diffuseColor, specularColor and emissiveColor properties of the material. As said before, diffuseColor and specular color contribute with the light components to render the object with lights; the ambient color reacts with the ambient color of the scene while emissiveColor is the color emitted by the mesh. The latter property is used in order to realize the lamp in the first room. I created a material with a bright emissive color and associate it to the lamp sphere. The combination of emissive bright color and the HighlightLayer object added to the scene simulates the seen of a real lamp. In the HighlightLayer you can set the blur color and if it will be emitted toward the inner mesh or outside it. You can change also the trasparency of a material setting the alpha value and this is what I've done with the lamp object.

Instead for changing the texture you have to change the diffuseTexture, specularTexture, emissiveTexture,bumpTexture and ambientTexture properties. Each of them is rendered and combined with the same type components of lights and colors. There are two main types of texture: procedural texture and image texture. The first type is based on code and its shape depends by some parameters. You can create your own procedural texture or use any of the out-of-the-box textures provided by Babylon. I used this latter type that is optimized and calculated inside the fragment shader. In particular I used the Wood Procedural Texture for the particles of the destroyed boxes and a Noise texture for the door of the second room. In the wood texture you can set the dimension, the amplitude of the waves and the color while the Noise Procedural textures has several parameters that modify in some way the noise flow. I changed the persistance to 2 , the color

and the animationSpeedFactor in order to obtain an animation faster and with noise bigger.
The image texture are Texture objects associated to a path of a texture image (typically .jpg or .png). I used this type for several object as boxes, axe, wall, key and spikes of the wall of second room.

Note that if you set the same texture/color for diffuse, ambient and specular components then the object covered by this texture/color will be not reflective. Viceversa if you set only the diffuse componet then the specular component will be white and the resulting object will be highly reflectivity. An example of such mechanism is the axe that reflect the light that hit it or the pavement of the first room, while good example of the first behaviour are the walls of the rooms that typically are poor reflective or the skin of birds.

A special type of texture is the bump texture. This type allow to the engine to simulate the bumps and wrinkles on the surface of a given object. This is achieved calculating the normal of the input image and passing the map obtained to the bumpTexture property of Texture object. I used this type of texture for floor of the first room and the doors. The normal maps are obtained using the site http://cpetry.github.io/NormalMap-Online/ .

It can be useful to know how the colors and textures are displayed by Babylon. Their rendering is done inside the fragment shader directly by the library, mapping the texture image into the vertices of object. This can be accompished creating a "fractional map" of the image and associating each region of the map to a region of the object. Babylon use facet of triangles to split and rendering textures and objects. A simple case can be a square texture applied to a square object. The object is composed by 8 vertices so it will be splitted in 12 triangles each of which colored indipendetly. At this point the engine creates an anologous map into the textures images (splitting it into 12 triangles) and creates a map between the two map and linked them to the vertex of object.

The last piece is the trick that I used in order to apply different textures to the sides of the boxes. This mechanism expolits the procedure described above and it is applicable to all base objects created with the MeshBuilder object of Babylon. When this object has distinct and recognizable faces than you can set for each of them a individual texture. They can be set using two parameters in the constructor of such object: faceUV(for texture) and faceColors (for colors) . The parameters are an array with length equal to the face where each element is a color or the texture. In particular for the texture it is very often used a trick that help to boost

performance and reducing the loading time. It uses an atlas that is a collection of texture store all inside a single file image. Given this image you can pass as element of FaceUV vector the coordinates of the given texture in the context of atlas. You have to specify the coordinates of two points (the bottom left and the top right point of the texture) using relative coordinates: so if your atlas is divided into 4 images disposed as a square you can specify the element between the points (0,0) and (1/2,1/2) in order to pick the element in the bottom left quadrant. I used this mechanism for TNT boxes and Wood boxes. Note that inverting some indices I have succefully rotated the texture and applied in several orientations to the TNT box.

7 Cameras

The camera represents the "eye" of the user. A scene needs a camera in order to be viewed by the user and tipically moving the camera you can explore your scene. What you see in the current time is what it is in the field of view of the camera. Babylon provides several types of camera, but for my game the best one that I used is Universal Camera. It has a control map that allow the user to use keyboard, mouse, touch screen or gamepad to move the camera. Universal Camera has several default inputs: for example the mouse movements rotate the camera and arrows buttons of the keyboard move smooothly the camera along the local Z axis. I added to these inputs the classical gaming commands W A S D to the default configuration in order to move the camera with buttons near the other buttons usefull in the game. The properties of the camera are set inside the custom class CameraGame. In particular here there are added the custom buttons, it is set the speed of the camera and there are set the parameters for collisions. In fact here we can declare to apply the gravity of the scene to our camera and set the flag checkCollision to true. Setting this flag to true will avoid to pass through objects as explained in the section "Collisions". Finally in order to be used during the game the scene must be attacched to the camera and this can be done attacching the canvas element of HTML page to the camera controls.

8 Animations

The animations allow to your scene to be more dynamic and through them you can move your objects inside the room during the game. You can produce animations in several ways: the more classic way is put movements or rotations or events into the rendering process and change their value gradually using some limits defined outside. An example of such mechanism is the fly movements of birds. They are moved inside a registerBeforeRender function of the third room, so updating their positions and orientation at each time. The function moves the birds in a pseudo-random manner, changing their rotation if they are too near to one wall, ground or roof and moving them along their Z axis if the movement improves their situation. A movement improves the situation if the sum of all the distance between the bird and all the walls is less than before.

Another way to do animations is using the Animation object of BabylonJS. An Animation object picks a property of a mesh and trasforms its value over the time. You can choose the entire sequence of values taken during the animation setting at each frame the given value. So if you want carry an object from A(position at frame 0) to B (position at frame 60) you can impose that the object will be at position C at frame 20 and in position D at frame 40 , so producing a path A-C-D-B. All these value are stored in a vector of keys. You can setup the type of animation (cyclic or non cyclic), what to do when the final destination is reached, what to do when the animation end and other usefull properties. After the creation of the Animation you have to attach it to the mesh that should be animated and starting the animation. These operations can be done in two ways: using a method of scene object that directly starts an animation passing as arguments of function beginDirectanimation() the mesh and the animation or adding the animation to the property "animations" of the mesh and in a second moment start all the animation associated to the mesh with the function beginAninmation of scene object. In my project I used both the approach for several animation (for example: picking objects, open door, bird's parts movements etc.).
An important thing about animations is how to manage them during the execution of the scene. When an animation begin I put it into a vector called onGoingAnimation in order to know at each moment what are the current animations in progress. When the user pauses the game or the character is dead then should be stopped the current animations. What I've implemented it is a function that take all the animations in the onGoingAnimation vector and pause/restart/stop them. Clearly this mechanism is applicable only to loop animations. There are many examples in the code of the application of these principles, here I'll show one example of cyclic animation and one for one-time animation.
The picking animation is a one-time animation that starts when the user click on

an object pickable and it includes two sub-animations: the rotation movement and the position movement. The first set the final rotation in a such way the object will be parallel to the world Y axis, while the second moves the object from the current position to the position in front of the camera using the method getFrontPosition of camera. Both the animations begin at frame 0 and end at frame 60 so they will be simultaneous: if you want that one animation starts before the other you can manage the frames number or use the "onAnimationEnd" property of the animation object that allow you to perform operation (or animation) after the end of the current animation executing a callback. As said before in order to be a one-time animation you have to set to false the parameter loop of method beginAnimation of the scene and if you want keep the last value of animated property you have to declare the type of change in the constructor of the animation as BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT.

Instead the movements of birds's wings are a perfect example of loop animation. The animation that rotates the wings uses the same structure of the previous one but changing 3 things: first of all the type of animations is set to BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE (instad of constant), the parameter "loop" of the beginAnimation is set to true and the keys associated to the property are speculars. They start from initial rotation, get some intermediate values and at the and of animation return to their initial values. In the case of wings they reach a rotation on X local axis of plus 60 degrees at frame 15, the initial rotation at frame 30, a rotation of minus 60 degrees at frame 45 and the initial rotation at frame 60. This sequence produces a smooth effect of progressive rotation more similar to the real one. Note that if you don't set the final rotation as the initial one then when the animation restarts they restart from the animation at frame 0 (that is the initial rotation) producing so a jerky indesidered effect.

In my project I created a function for each different animation and two auxiliares functions for rotation and positioning. Each of them creates an Animation object, fixes the keys and returns the animation to the caller. The caller (often the constructor of object or event) has the duty to call the scene method to start the animation and to manage them and this can happen in several ways. The pause menu for example has the duty to pause all the current animations and restart them only when the user click on resume; The restart menu to restart the animations; the throw animation and picking animation are called by the class of the object (key or weapon) inside the method action invoked when the user interacts with them; the wall animation is started alongside the room that contains them when this is rendered, so the controller is the room itself, and finally the birds animation are started when the birds are created and the controller is the bird.

9 Collisions

An event that you don't want to happen in your scene is that one solid object passes through another solid object or that your character can pass through walls and obstacles. In order to avoid such mechanisms you have to check when there are "collisions" between your meshes or you camera and meshes and develop a solution to the problem.
First of all I'm going to explain how to deal with Camera - Object collisions. For detection of collisions you have to set to true the checkCollision property of both camera and meshes and you have to define an ellipsoid around the camera. The ellipsoid is customizable for dimensions in any directions and acts as a barrier so you can't move in a desired direction if doing such movement will cause an intersection between the ellipsoid and a mesh with checkCollision set to true. The detection of the interesection and its avoiding (for camera movement) is done inside the engine of BabylonJS. The ellipsoid can be applied also to the meshes and move them with a method called moveWithCollision that works in the same way of the movement of the camera. This mechanism is used in my game in order to move the camera and move the object holded in hand (through moveWithCollision method).

In the second room the collisions are exploited in order to end the game. In fact if one of the two walls hits the camera then the character is declared dead and the game ends. The behavior is implemented using an invisible mesh "controller" associated to the camera. At each time, using the scene.afterRender method, the controller checks if it collides with one of the walls through the method mesh.intersectMesh: if the collision happens then the system pauses the game and shows a popup declaring the loose of the user as explained in the menu section. The check of this method works in similar way to the ellipsoid one using a bounding box around the mesh. You can set the precision of such box, but if you select the more accurate box the performance of the system could slow down and for this reason I selected the less accurate method. The intersectMesh method inside a scene.registerBeforeRender function is used also in the case of box and weapons. In particular in the first room at each time if the player has an axe on hand and perform the action of the axe then it checks if during the animation the axe interects a box: if this happen then the onDestroy method described in the section Classes is called and the box is destroyed.

The last type of collision is that between a point and a mesh. This type of collisions can be obtained using the method pick(x,y) of the scene object; this method takes as argument the coordinates of a point and returns an object containing several information as for example if the point hits the mesh, the mesh "collided", the distance from the mesh and so on. In my game I used as point the cursor position

in order to realize some behaviours as picking objects, interact with doors and so on. For example in the case of picking an object you have to put your cursor into the mesh that you want to pick and the click on it. The interaction starts when the user click is captured by the event listener and it checks if the current position of cursor hits a mesh, if it is pickable and if the mesh is near enough: if all these conditions are satisfied than it can start the picking animation that bring the object from the room to the hand of player.

10 Hierarchy

Building a hierachy is the main way to make relationship between the elements of your scene. You can create a simple relationship parent-child declaring an element the parent of the other (so setting the "parent" property of the element). The hierarchy is very usefull to create complex objects and complex animations. In fact when you have a hierarchy you can still animate each child indipently but if you animate a parent then the change will be reflected also on the children. This is a very powerfull property of hierarchy in order to create complex animation without specifing the movement for each part. The parent-child relationship can be used in a several ways and it is used in a lot of my object and animations. The most simple case is the relationship between the spotlights and the camera in the second and third room. Declaring the light as child of camera means that the light will follow the camera position and rotation and so it light up the things in front of the camera. The same mechanism is used for the camera's controller of the second room described in the section "Collision".

The hierarchy can be used for animations when you want rotate a mesh not around the center of the mesh but in another point. This can be obtained also rotating the object using a pivot or using the world axis but the more "natural" way is to use a child-parent relationship. For example in the real door when you open it the thing that rotate isn't the door itself but the hinges that attach the door to the wall. I adopted the same mechanism in order to reproduce the door open animations: I created a hinge invisible to the user, positionated it on the right of the door and set it as its parent. When the door must be open then the hinge will be animated rotating is Y axis and so rotating also the door. Babylon.js offers a node called "TransformNode" that acts as the hinge: so it is an invisible thing that can pick elements as child or parent and can rotate or change position as the normal mesh, but that avoid the payload for the engine to load the classical mesh. I used this node for all the junctions of the birds.

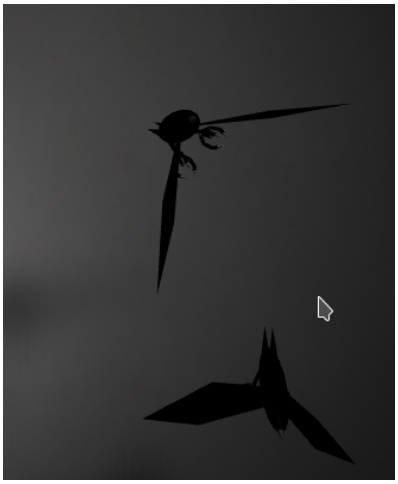

Figure 1: Birds flying

The most complexes hierarchical objects in my game are the birds visible in the figure above. Each bird is composed by a body, a tail, a beak, two wings, two legs, two palm and eight fingers plus all the junctions between phalanges and between body and legs. The root of the hierarchy is the body. Only the body is moved/rotated by the fly action while all its descendent automatically will gain the same amount of change managing so the whole object as unit. The indipendence of the children can be seen noting that the animation of the wings, or to better say of the junction of the wings, has no effect on the rest of the body. So if you stop or change that animation the rest of the bird remains the same. A more complex interactions through the hierarchy can be seen in the legs and paws. Here the top node is the leg that has as child the palm; then each palm has 4 fingers as children where each finger is a complex object composed by 2 junction, 3 phalanges and a nail. When it starts the animation of legs than also fingers are moved following thei movement. The power of hierarchy is visible also in the implementation of such object. In fact you can define all the phalanges with relative position and orientation respect to their parent so ignoring their absolute position in the scene. This allow you to define a unique parametric function that build the fingers automatically and then attach them to the palm and it is very usefull because you can change general things about a part of the object without affecting the others. The same reasoning can be applied for the rotation: when the fingers are closing each phalange is rotated of the same amount but because each rotation is relative to its parent the global animation seem to be a closing animation.

11 Sounds

In order to apply some effect sounds to our game BabylonJS provides a usefull sound engine that use web audio specification in order to reproduce the sounds into our browser. In order to mantain the full compatibility with all the types of browser, I have used only .mp3 and .wav audio. Each audio is an instance of the class Sound. This class has several useful methods and property that allow us to similate ambient, event and object sounds.

An example of ambient sound is the timer sound of the second room. It starts when the player enter into the room loading it alongside the objects of the room and playing it in registerAfterRender function of the scene. The choice of the registerAfterRender is motivated by the fact that if I put the play method in registerBeforeRenderor in the code of the room creation then the sound will be played when the room is instanciated. This happen before it is rended because the room is created when the door of the previous room is created. In order to play continuously the sound I set a flag (loop) into the constructor and one of the room object. The first allow to replay the sound when this finish, while the value of the second is changed only the first time the sound is played and allow to not run multiple instance of the object each time the registerAfterRender function is called.

The event sounds are associated to events in the game. They typically are played only one time during the event and then silenced. The loading code is put in the room creation in order to reduce the delay from the start of the event while the play action is put inside the method of the event before or after the animation. Examples of such behavior are the door open event,the action event associated to the axe and the sounds associated to the destruction of the boxes. In all these cases when the user interact with the object also a sound is played.

The last type of sound is the object sound. This type is associated with a mesh (through a method of the Sound class) and tyical has a "attenuation" factor that increase or decrease the sound on the basis of the camera position. This mechanism could implemented in the Room 2 where a sound can associated to both the moving walls and the sound become more and more stronger during the animation when the camera become near. Unfortunately I don't found a good sound file for the event described and I avoided to implement it.

Regarding the volume audio this can be set both globally or for each sound. Each sound object has a method called setVolume where you can set the volume of the sound from 0 to 1. In my project I allow to change the global volume using the volume setting menu. This can be done passing to the method setGlobalVolume

of the audio engine the desidered volume. In our case the desidered volume will be the current value of the slider in the menu. The starting value is 1, the highest.

12 User Manual

The Escape Game is a game where you are a man that has to escape from a building. It is inspidered by the popular attractions Escape Rooms where you are closed inside a room and you have to solve several riddles in order to exit from the room. In this adaptation you have to pass through the door of each room until up you will win the game and obtain your freedom.

The game starts with the homepage displayed in Figure 2. As you can see you have 3 options:

Figure 2: The HomePage

The first option is "Start new game" and, as the name suggest, clicking on it will be starts a new game starting from the first Room. The second option allow you to see what are the commands of the game and they are described also in this section later. Tha last option allow you to set the volume of the game. There are not lot of sound in game but in some rooms there is a background sound. You can change the volume simply moving the slider in the volume settings menu.

After clicking the "start new game" button you will be inside a room.Each room has a single door that can be opened or closed. If the door is closed you have to

find a key in order to open it otherwise you can simple open it positionating in front of it and pressing the "E" key. In the Figure 3 you can see the first room and the door in the wall in front of us.

Figure 3: An example of room

The basic movements commands that you can do in the rooms are the following:

- Rotate the view: using the mouse movements;
- Go Forward: You can use "Arrow Up" key or "Q" key in order to move forward the player;
- Go Left: You can use "Arrow Left" key or "A" key in order to move left the player;
- Go Right: You can use "Arrow Right" key or "D" key in order to move left the player;
- Go Backward: You can use "Arrow Down" key or "S" key in order to move left the player;
- Run: Hold down the "SHIFT" key.

Inside the rooms you can interact with several objects. Some of them can be carry on hand. In order to bring the object you have to be near to it and click on it. If the object is carryable then there will starts an animation that bring the object from its current position to the position in front of you. If the object is not

carryable then none animation will start and your hand will reamin empty. Note that if you decide to carry an object while you are holding another then the game automatically throw away the object in your hands.

Figure 4: An axe on hand

When you have an item in hand, as shown in Figure 4, you can do two things:

- Throw away the object pressing the "F" key. This action will move the object in your hand to the roof of the current room. Clearly you can bring it later simply clicking on it.
- Use the item pressing the "E" key. This action depends by the item's type. If the item is a weapon then the action is an attacking action and can be used to destroy some objects of the rooms (wooden boxes and tnt boxes). If the item is a key the action will open the door in front of you if there is one otherwise has no effect.

Finally at each time you can interact with the menu on the right shown in Figure 2 and Figure 3. It has 4 options: the first two are analogous to the option in the homepage while the option "restart" allow you to restart the game from the current room re-starting from the initial position and situation; and the option "exit" that allow you to return to the homepage. Remember that you can pause the game at each time simply pressing the "Esc" key: this option will stop the time of each room so you can take your time and resume when you are ready.