STEFANO CASALUCI - SIMONE CAVALLARO

1608338 - 1605177

INTERACTIVE GRAPHICS

MAZE-UP

MazeUp is a ThreeJS based game where you have to find a key to exit the maze before the time runs out. The game is mostly based on a WebGL environment but other tools have been used:

- HTML + Javacript: used to create the canvas element which provides a drawing area on the web page and to manipulate models and their variables
- Three.js: a cross-browser Javascript library used to create and display animated 3D computer graphics in a web browser based on WebGL. Mouse, key and maze are all Three.js models
- jQuery: a Javascript library used for manipulation, event handling and animation, that simplifies the usage of Javascript functions. This is used, for example, to manipulate HTML elements such as *$('#element').method();*
- Keyboard.js: a Javascript library used for binding keys and combos of the keyboard. This is used to show the instructions, to go back to menu, to rotate the axis while the POV(Point of View) is in First Person and generally to move the mouse to get out of the maze
- Box2D.js: a physics engine used to give the overall environment natural dynamics such as impacts, linear velocity, friction etc.
- Dat.gui.js: a lightweight graphical user interface (GUI) used to change variables in Javascript. In our project is used to zoom in and out and to create the main menu

MOUSE

It is the main character of the game and the player must control it in order to escape the maze after the key has been found.
In the *createPhysicsWorld()* and *updatePhysicsWorld()* functions we define the physics of the mouse such as its "hitbox", its friction and its speed based on the user input.
In the *createRenderWorld()* function we begin the definition of the mouse model using ThreeJS models: a mouse is defined as a hierarchical model made up of

- A torso
- A head
- Four legs
- A tail

For each of these components we define a new *CubeGeometry(),* a new *MeshBasicMaterial()* applying a texture and we add this parts to the global *mouseMesh*.
Once this is created, the mesh is added to the scene:

scene.add(mouseMesh);

In the *updateRenderWorld()* function, we create the movement of the different parts of the mouse, increasing or decreasing the value of *mousepartX.rotation.z* in order to simulate a back and forth movement while the mouse moves through the maze.
Still in this function, we also define the rotation of the entire body of the mouse with respect to the pressed arrow key, if "third person" POV is selected: through a *switch* we select the correct key pressed in the *document.onkeydown* event and we rotate the body for a quantity of *Math.PI/32* which gives the best results, in the sense that the mouse rotation feels more natural rather than sharp. Instead, if "first person" POV is selected, we perform the switch directly in the document.onkeydown function, where the mouse is allowed to rotate only on left or right.
The position of the mouse is also used later in the code to check if the key has been collected and to check if the player has found the exit of the maze.

KEY

As said above, the key is a special model in the game used to correctly exit the maze and proceed to the next level. It is a ThreeJS model made up of two meshes:

- The core of the key
- The outline of the key

The full mesh is created, similarly to the mouse, in the *createRenderWorld()* function but as seen in the code, the material changes with the respect to the night/day mode: in the first case, we use *MeshPhongMaterial()* which is a material for shining surfaces; in the second case, we use a *MeshBasicMaterial()*, to avoid excessive brightness in the environment. It is important to notice that another option for the material was the *MeshLamberMaterial()* that uses the *Gouraud* model to calculate shading. The *Phong* shading model, used in the *MeshPhongMaterial()*, calculates a per pixel shading, which gives more accuracy at the cost of some performance.
The core of the key is defined as a *OctahedronGeometry()* while the outline of the key is a *Torus Knot:* a particular geometric shape defined by a pair of coprime numbers. Different couples lead to different geometries and, in our case, we chose the numbers 2 and 3 to create our knot.
The core of the key is drawn with a red color, the outline of the key with a yellow color.
Once the mesh is created, the key is added to the scene as well:

scene.add(keyObj);

The key, which rotates on both *x* and *y* axis, is positioned in a legit part of the maze (see below) in the *generate_maze_mesh()* function with:

keyObj.position.set(keyX, keyY, 1);

where *keyX* and *keyY* are "random" generated with respect to the maze composition.
Once the player finds the key, it is removed from the scene and the variable "key" is set to *true* to signal that it is found(it is also indicated on the screen when the key is found), waiting to be generated again for the next level:

scene.remove(keyObj);

LIGHTS

In the game, since we have two different game modes (day/night) we use two different type of lights: an ambient light in case of day mode, and a spot light in the case of night mode. This can be configured through the main menu by the player.
Lights are defined in the *createRenderWorld()* function and are created using ThreeJS: the ambient light is created as well

new THREE.AmbientLight(0xffffff)

while the spot light is created as well

new THREE.SpotLight(0xffffff)

Note that *0xffffff* means a pure white light.
For the ambient light there is nothing more to configure and therefore we can directly add it to the scene if we are in day mode. But for the spot light we also need to set up some properties: *angle, penumbra, decay, intensity* etc.
This is because this kind of light gets emitted from a single point in one direction, along a cone that increases in size the further from the light it gets, while the ambient light globally illuminates all objects in the scene equally.
The light position and direction are set and updated in the *updateRenderWorld()* function such that the light always stays on top of the camera.

CAMERA

The camera is basically a *perspective camera,* created with *three.js* library, which is the most common projection mode used to mimic the way human eye sees. Based on the point of view chosen in the main menu(explained later), the camera is set in two different mode:

- Third Person: the camera is added directly to the overall scene, and set on top of the maze, in such a way it can always see a portion of the maze, following mouse movements.

```
var aspect = window.innerWidth/window.innerHeight;
camera = new THREE.PerspectiveCamera(120, aspect, 1, 1000);
camera.position.set(1, 1, 5);
```

  In *updateRenderWorld()* function its position is updated based on mouse position. It is done in a way that the camera continue to follow the mouse when it moves, as it remains at the center of the scene.

```
camera.position.x += (mouseMesh.position.x - camera.position.x) * 0.1;
camera.position.y += (mouseMesh.position.y - camera.position.y) * 0.1;
```

- First Person: here things are little more complicated. The camera is added on top of the mouse tail, positioned and rotated in such a way it seems that we are looking through the eyes of the mouse. In particular, in the *gameLoop()* function, the camera is rotated of 90° backwards on both y and z axis in such a way it looks what the mouse has in front of.

```
var aspect = window.innerWidth/window.innerHeight;
camera = new THREE.PerspectiveCamera(30, aspect, 1, 1000);
```

```
camera.position.set(-0.4, 0, -0.6);
```

In this game mode are also used two important variables, called “camX” and “camY”. They are necessary to help to update in a correct way the light position. Since the light is always positioned on top of the maze, in ‘third person’ we have not problem because the light follow camera movements continuously. But in ‘first person’ the camera is positioned on top of the tail when the game start and its position never change, it moves simply because the mouse moves. This generate problems from the fact that if we don’t update the camera position, not even light position is updated. So the two variables come to help us : they are initialized to the camera position coordinates and then updated in *updateRenderWorld()* function based on the mouse position. In this way the light always light up the path in front of the mouse.

```
camX += (mouseMesh.position.x - camX) * 0.5;
camY += (mouseMesh.position.y - camY) * 0.5;
spotL.target.position.x = camX;
spotL.target.position.y = camY;
spotL.target.position.z = camera.position.z;
spotL.position.x = camX;
spotL.position.y = camY;
spotL.position.z = camera.position.z + 5;
```

In both cases, for each state of *gameLoop()* function the camera perform its functionality through ‘renderer.render(scene, camera) instruction.

USER INTERACTION

The user can interact with the game in several ways:

- Main menu: shown when the game starts, it is a menu built using *dat.gui.js* and enables the user to tune the game with respect to POV(Point of View, Third Person or First Person), difficulty (the duration of the timer), maze size (changes the *mazeDimension* variable), kind of texture (selects a different texture for the walls) and day/night mode (applies a different type of light to the environment). The menu function (*setup()*) is called in the *$(document).ready()* function and saves the parameters chosen by the player in different variables, used throughout the entire code to create/render the environment

- Instructions: available only when the game is started, the user can hold down the spacebar to show a pop-up explaining how to play. This is done through *keyboard.js* binding to the specific spacebar key: *KeyboardJS.bind.key(‘spacebar’, ...);*

- Back to main menu: once the game has started, the user can go back to the main menu by pressing the escape key. This works similarly to the instructions: *KeyboardJS.bind.key(‘esc’, ...);* Is important to notice that we’re not simply going back to the main menu, we’re just reloading the page to force the html page to show us the menu: this is done with *location.reload();*

- Zooming: once the game has started, the user can relocate the camera, bring it closer or further with respect to the mouse. This is possible only when *Third Person* is chosen as POV. This particular interaction is done via *dat.gui.js:* dragging the bar around will increase or decrease the value of *camera.position.z* and, in other words, the distance between the mouse and the camera

- Mouse movements: the most important interaction in the game, allows the player to move the mouse around the maze using arrow keys. There is a difference between the key combos that are used in *Third Person* and the key combos used in *First Person* (see below), but, in general, pressing the up arrow key will make the mouse go up in the maze, pressing the left key will make the mouse go to the left and so on so forth. The position of the mouse is updated in the *updateRenderWorld()* function:

```
var stepX = mouse.GetPosition().x – mouseMesh.position.x;
var stepY = mouse.GetPosition().y – mouseMesh.position.y;
mouseMesh.position.x += stepX;
mouseMesh.position.y += stepY;
```

  Where *mouse.GetPosition()* returns the position of the mouse in the "physics world": it's position is updated in the *updatePhysicsWorld()* function.
  As said above, there is a difference between the key combos used in *Third Person* mode and *First Person* mode. In the first case, the commands are the most intuitive: pressing the up arrow key will make the mouse move up in the maze, pressing the left arrow key will make the mouse move to the left etc. In *First Person*, things change a bit: the user can now move the mouse forwards or backwards only (since with the camera placed on the tail of the mouse the user can only see in front of the mouse, as if he was the mouse eyes), using the up arrow key or the down arrow key. Left and right arrows will only cause the player to move the camera to the left or right respectively (i.e. to perform a body rotation of a 90 degree). It is possible to always use the up and down arrow to move the mouse up and down even though the body has rotated with *keyboard.js* by changing the active axis after a rotation to the left or to the right:

```
KeyboardJS.bind.axis();
KeyboardJS.unbind.axis();
```

MAIN LOOP

The *index.html* file execution starts at *$(document).ready()* function that is a special jQuery function invoked when the document is fully loaded. Here we hide *help, levels, fails, timer, instructions,* and *back* elements from the canvas while the *menu* element is shown. We also prepare the *instructions* element, by setting it at the center of the screen (but it remains hidden). Then we bind the spacebar key (using *keyboard.js*) to show the instructions once pressed and to hide the instructions once released. A new *renderer* element is created (*new THREE.WebGLRenderer()*) and finally the *setup()* function is invoked. The setup is the part of the code where the menu is created (using *dat.gui.js*) and it is made up of:

- 4 dropdown menus (POV, Difficulty, Maze, Texture)
- 1 checkbox (for day or night mode)
- 1 button (the play button)

As said above, this menu helps the user to configure its game experience by choosing a particular aspect for the maze or a particular difficulty, making the game very easy or very hard. The parameters for all these elements are then saved in different variables. Once the user has completed the setup, he can press the play button which changes the *gameState* variable (a special variable used to distinguish between the different game phases, see below) to the value *'initialize'*.

Finally, the *requestAnimationFrame()* function is invoked, which requests the browser to perform a specific animation calling a specific function (*gameLoop()* in this case). The *gameLoop()* function is a big *switch* where we perform specific action with respect to the value of *gameState*:

- '*initialize'* state is the first phase, where we first show the *help, levels, fails, timer* and *back* elements and we hide the *menu* element from the canvas. We also initialize the timer, generate the maze, create the physics and the render worlds, create the zooming menu in the top right corner and set the camera with respect to the chosen POV. Finally, we change the value of *gameState* to *'fade in'*.

- In the '*fade in'* case, as the name suggests, we perform a small transition to the game by the increment of the light intensity (only if the spot light is used). In the end, we change the *gameState* value to *'play'*.

- In the '*play'* case, we update both physics and render worlds, we set the axis correctly with respect to the rotation of the mouse body, we bind the escape key to allow the player to go back to the main menu, we control if the timer has expired (see below) and then we check if we have found the key (in this case we update the *key* element to reflect this, and we remove the *keyObj* from the level) and if we can proceed to the next level (by checking the mouse position and if the key has been collected). In the positive case, we update the *level* element and the *mazeDimension*, while switching to the *'fade out'* case; in the negative case, we update the *fails* element and we switch to the *'fade out'* case.

- In the '*fade out'* case, we perform a small transition by decreasing the light intensity (only if spot light is used) and then we set the value of *gameState* to *'initialize'* once again to recreate the world.

- The last possible case of the *gameState* variable is *'timeout',* set when the timer expires. Here we perform a transition by decreasing the light intensity (if spot light is used) and we update both *timer* and *fails* elements to the correct values. In the end, the *gameState* is set to *'initialize'* once again. The final part of the *gameLoop()* function is

requestAnimationFrame(gameLoop);

meaning that we call the same function over and over again until the player decides to exit the game.