Monument Valley

Interactive Graphics project

Fiorella Artuso 1602113
Giovanna Flore 1612975
Andrea Migliori 1607771

1 Introduction

This project is a simplified version of Monument Valley game in which the character has to find a way to reach the finish line by interacting with the environment (pushing buttons, rotate wheels etc) in order to modify the path. It consists of three levels with increasing difficulty.

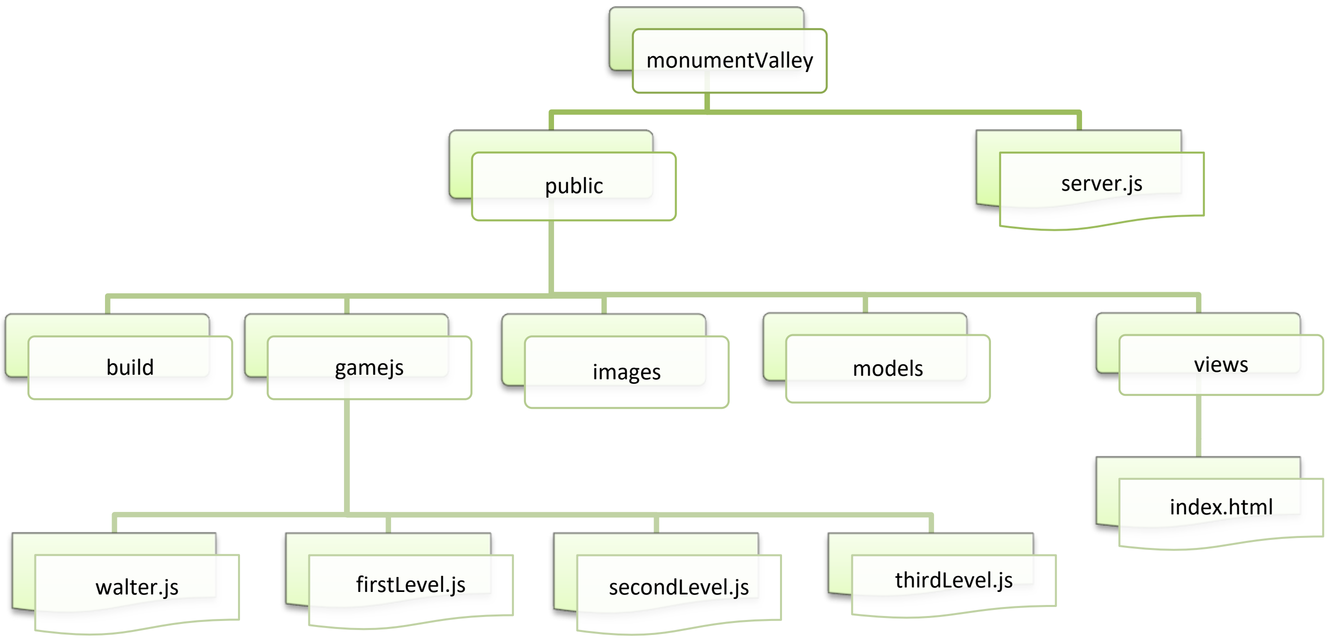
2 General aspects

2.1 Environment

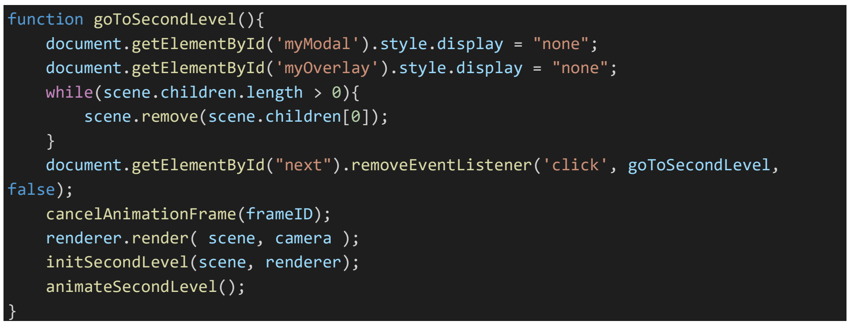
Instead of using the basic WebGL, we decided to employ Three.js which is a cross-browser JavaScript library and Application Programming Interface used to create and display animated 3D computer graphics in a web browser. It is based on WebGL, but it provides further additional facilities dealing with all implementation aspects and making easier to build a more complex application.

2.2 Structure of the project

The following tree shows only the main directories and files.



The index.html file is the one shown once the user connects to the server and it includes all the scripts needed by the application as well as the firstLevel.js file which initializes, animates and renders the first level of the game. Once the first level is completed, the function *goToSecondLevel()*, which implements the **transition** between the two levels, is called in firstLevel.js. In fact, it deals with removing all the elements from the scene and calling the main functions to initialize, animate and render the second level.



```
function goToSecondLevel(){
    document.getElementById('myModal').style.display = "none";
    document.getElementById('myOverlay').style.display = "none";
    while(scene.children.length > 0){
        scene.remove(scene.children[0]);
    }
    document.getElementById("next").removeEventListener('click', goToSecondLevel,
false);
    cancelAnimationFrame(frameID);
    renderer.render( scene, camera );
    initSecondLevel(scene, renderer);
    animateSecondLevel();
}
```

The same holds for the two remaining Javascript files:

- in secondLevel.js, the function *goToThirdLevel()* is called
- in thirdLevel.js, the function *goToFirstLevel()* is called to restart the game

Each Javascript file implementing a level is organized according to the following structure:

- **initXX(scene, renderer)**[1] – it initializes some basic elements such as camera, lights, raycaster, keyboard controls, etc.
- **XX()**[1] – it initializes and adds to the scene the meshes constituting the level elements and it loads external models (.obj).
- **animateXX()**[1] – it creates a loop that calls rendererXX() and updateXX() functions.
- **renderXX()**[1] – it draws the scene every time the screen is refreshed
- **updateXX()**[1] – it checks some conditions to be satisfied and in that case it calls the appropriate functions.

2.3 List of all libraries used

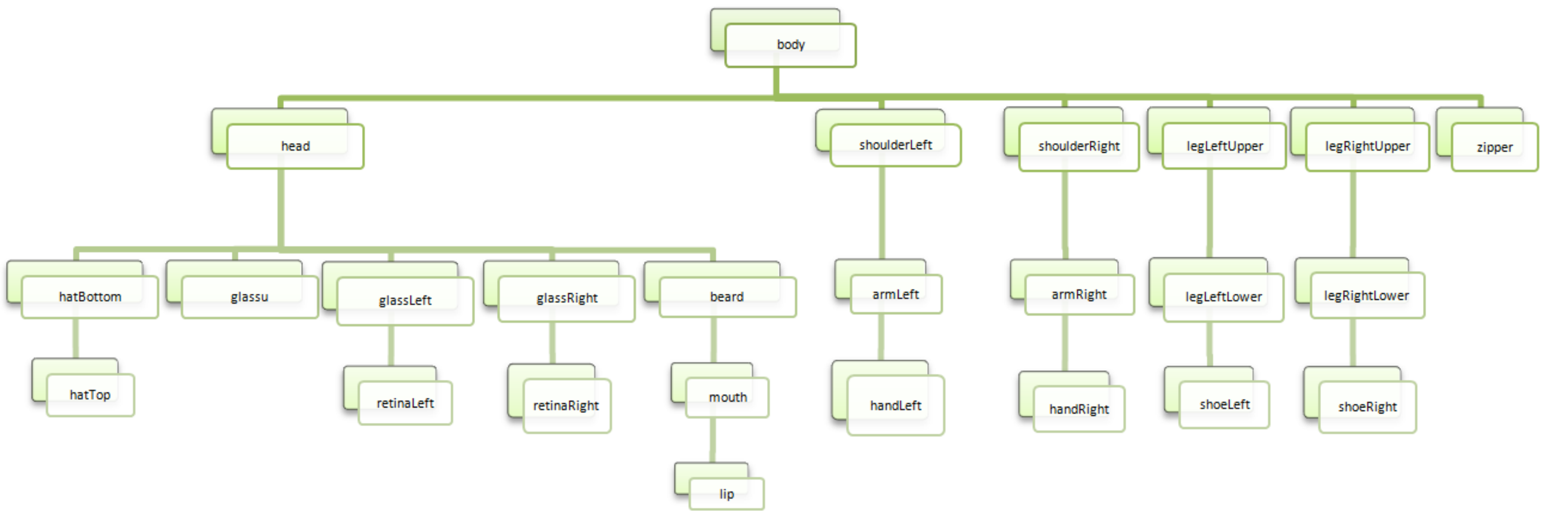
- **THREEx.FullScreen.js**
- **THREEx.WindowResize.js**
- **Detector.js**
- **WebGL.js**
- **MTLLoader.js**, **OBJLoader.js** – used to load external 3D models (.obj) together with their materials.
- **tween.min.js** – used to animate elements (translation, rotation, scale, etc) smoothly in time.
- **THREEx.KeyboardState.js** – used to control the movement of the character through the arrow keys.

2.4 External models

The main character Walter is created in the walter.js file and its raw structure has been taken from the web (click here). However, its parts were not hierarchical so we restructured the character as a tree model.

[1] XX represents the name of the Javascript file it is contained into.

The tree model is as follows:



Walter.js file also contains the code for the character animation written by our own.

2.4.1 Other external models

First level:

- island
- wheel
- clouds

Second level:

- candy stick
- present
- cottage
- cloud
- Christmas three

Third level:

- arches
- planets (Earth, Saturn, Moon)

2.5 Technical aspects

2.5.1 Camera

We decided to use the **perspective camera** instead of the orthographic one, because the former is designed to mimic the way the human eye sees, whereas the latter makes an object's size in the rendered image constant regardless of its distance from the camera.

2.5.2 Light

We decided to use a **point light** which is a light that gets emitted from a single point in all directions so as to cast shadows.

2.5.3 Sky
The sky is realized by using a sphere geometry and a shader material which takes as input a vertex shader and a fragment shader to create a color gradient.

```
var skyGeo = new THREE.SphereBufferGeometry( 250, 20, 40);
    var skyMat = new THREE.ShaderMaterial( { vertexShader: vertexShader, fragmentShader: fragmentShader, uniforms: 
uniforms, side: THREE.BackSide } );
    var sky = new THREE.Mesh( skyGeo, skyMat );
    scene.add( sky );
```

2.5.4 Character walk animation through keyboard control
In walter.js file, the functions that are used to animate the character are present and they are the following:

- walkForward()
- walkStop()
- rotateLeft()
- rotateRight()

2.5.4.1 walkForward( )
In order to simulate the walk, this function translates the body along the z axis by using the *translateZ()* function. Since the body is the root of the tree representing the hierarchical model, in order to translate all the components, it is sufficient to simply translate it.

The movement of legs and shoulders is defined as follows: while the legRightUpper and the shoulderLeft move forward, the legLeftUpper and the shoulderRight move backward. This movement is controlled by increasing (forward) or decreasing (backward) the angles of these components by using rotation functions. When the angle becomes too big or too small the direction is changed to the opposite by using the variable *ahead* whose sign is inverted: since this variable is multiplied with the angle taken as input by the rotation functions, the legs/shoulders moving forward start moving backward and vice versa.

In order to make the walk more realistic, we decided to add a little rotation of the head and a little translation of the body up and down.

```
this.walkForward = function(){
        this.body.translateZ(  20*scale );
        if (this.legLeftUpper.rotation.x < -0.7)
            ahead = 1;
        else if (this.legLeftUpper.rotation.x > 0.7)
            ahead = -1;

        this.legLeftUpper.rotateX(0.05 * ahead);
        this.legRightUpper.rotateX(- 0.05 * ahead);
        this.shoulderLeft.rotateZ(- 0.05 * ahead);
        this.shoulderRight.rotateZ(0.05 * ahead);

        this.retinaLeft.rotateZ(0.1);
        this.retinaRight.rotateZ(-0.1);
        this.handLeft.rotateY(0.1);
        this.handRight.rotateY(-0.1);
        this.head.rotateX(0.004 * ahead);
        this.body.translateY(1 * scale * ahead);
    }
```

2.5.4.2 walkStop( )

This function sets to zero all the angles of the components involved in the animation.

2.5.4.3 rotateLeft( ) and rotateRight( )

These functions rotate the body left/right (and so all the other components) to allow the character to move in all directions.

2.5.4.4 Keyboard control

In each Javascript file the character movement is controlled through the keyboard arrow keys by using the THREEx.KeyboardState.js library. In order to do this, we first need to instantiate the keyboard:

```
keyboard = new THREEx.KeyboardState();
```

Then in the *updateXX()* function the most appropriate function, among the ones presented above, is called according to which arrow the user presses.

```
// move walter forwards
if ( keyboard.pressed("up") && canMoveWalter)
    walter.walkForward();
else
    walter.walkStop();

// rotate walter left/right
if ( keyboard.pressed("left") && canMoveWalter )
    walter.rotateLeft();

if ( keyboard.pressed("right") && canMoveWalter)
    walter.rotateRight();
```

2.5.5 Raycaster

Raycasting is the use of ray from an origin to a destination to detect intersections with objects. We used raycasting for two different purposes:

- for mouse picking
- for collision detection

2.5.5.1 Raycaster for mouse picking

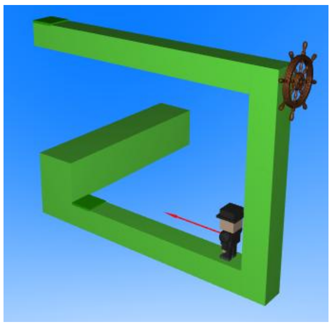
```
    mouse.x = ( event.clientX / renderer.domElement.clientWidth ) * 2 - 1;
    mouse.y = - ( event.clientY / renderer.domElement.clientHeight ) * 2 + 1;

    raycaster.setFromCamera( mouse, camera );
    var intersects = raycaster.intersectObjects( scene.children, true );
```

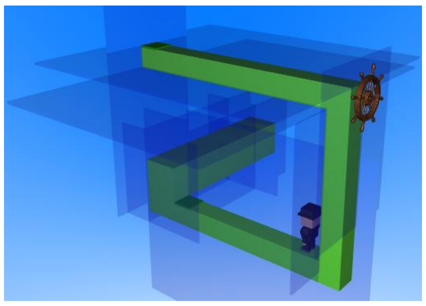
First we set the raycaster so that the ray goes from the camera to the coordinates representing the point where the user clicked with the mouse. Then we calculate objects intersecting the picking ray and we distinguish them according their ID in order to perform different additional actions.

2.5.5.2 Raycaster for collision detection

In order to handle character's collisions against scene objects, we used a ray having the character position as origin and the direction the character looks at as its direction.

We defined a list of meshes with which the character must collide: if such ray intersects one of them, the character is translated back to simulate the impact with the object.





Since we also needed the character not to fall out the path, we created invisible walls around the various meshes of the path against which the character collides every time he tries to overcome them.

```
var ray = new THREE.Vector3(0, 0, 1);
var collidableMeshList = […];
var origin = new THREE.Vector3(walter.body.position.x, walter.body.position.y, walter.body.position.z);
var dir = new THREE.Vector3().copy(ray);
dir = dir.applyMatrix4( matrix );
var rayc = new THREE.Raycaster(origin, dir, 0.6, dist);
var intersections = rayc.intersectObjects(collidableMeshList);
if (intersections.length > 0 && translateCollision)
    walter.body.translateZ(-1);
```

2.5.6 OBJ and MTL loaders

All external 3D models are loaded by using OBJ loaders and their materials are loaded with MTL loaders. An example of such loader is the following:

```
function createIsland(){
    var onProgress = function ( xhr ) {
        if ( xhr.lengthComputable ) {
            var percentComplete = xhr.loaded / xhr.total * 100;
            console.log( Math.round( percentComplete, 2 ) + '% downloaded' );
        }
    };

    var onError = function ( xhr ) { };

    THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );

    new THREE.MTLLoader()
        .setPath( 'models/island/' )
        .load( 'low-poly-mill.mtl', function ( materials ) {

            materials.preload();

            new THREE.OBJLoader()
```

```
                .setMaterials( materials )
                .setPath( 'models/island/' )
                .load( 'low-poly-mill.obj', function ( object ) {
                            …
                    scene.add( object );

                }, onProgress, onError );
        } );
}
```

3 First level

3.1 Technical aspects

Technical aspects of the first level are described below.

3.1.1 Path creation

It's important to notice that the whole path as been created by our own as a tree model by using the Group class.

3.1.2 Perspective illusion

The perspective illusion consists in the fact that at the beginning mesh6 is laid on mesh5 as it actually appears to be (Figure 1), whereas, once the user clicks on the wheel, the position of mesh6 is changed so that it is attached to mesh3 (Figure 2), even though it still appears to lay on mesh5 (Figure 3).

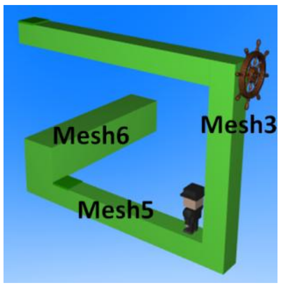


Figure 1

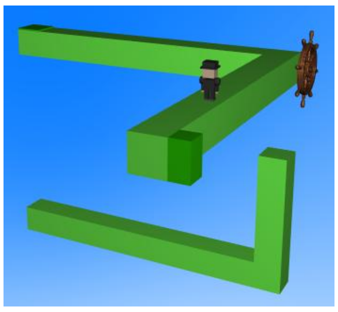


Figure 2 *(camera rotated)*

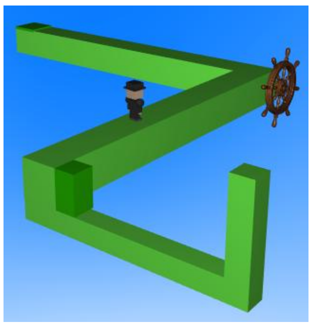


Figure 3

This effect is achieved by scaling down mesh6 once its position is changed, because otherwise, being closer to the camera, it would be bigger and the user would be aware of these changes, losing the illusion.

The main issue was related to the fact that the character is on top of mesh6 while the above mentioned transformations happen, so this brought to the character not being correctly positioned and scaled. In order to solve this problem we first scale down appropriately the character and then we compute its new position in proportion to the new mesh6 dimension and also taking into account the new mesh6 position, so that it appears to be coherent to its previous position from the user's point of view.

Below the code used to achieve such result is presented.
First we calculate the coordinates of extreme points of mesh6 and the character position on the XZ plane before all the transformations.

```
var x1 = firstLevel.mesh6.matrixWorld.getPosition().x - firstLevel.mesh6.geometry.parameters.width/2;
var x2 = firstLevel.mesh6.matrixWorld.getPosition().x + firstLevel.mesh6.geometry.parameters.width/2;

var z1 = firstLevel.mesh6.matrixWorld.getPosition().z + firstLevel.mesh6.geometry.parameters.depth/2;
var z2 = firstLevel.mesh6.matrixWorld.getPosition().z - firstLevel.mesh6.geometry.parameters.depth/2;

var walterX = walter.body.position.x;
var walterZ = walter.body.position.z;
```

Then we compute along each axis the ratio representing the character position with respect to the original mesh6 dimensions.

```
var fx = (walterX - x1) / (x2-x1);
var fz = (walterZ - z1) / (z2-z1);
```

After translating mesh6 to the new position, we update the matrix representing its coordinates.

```
firstLevel.mesh6.updateMatrixWorld();
```

At the end, the new character position is computed by taking into account the new mesh6 position and the above ratio.

```
walter.body.position.x = firstLevel.mesh6.matrixWorld.getPosition().x - firstLevel.mesh6.geometry.parameters.width/2 +
                         fx * firstLevel.mesh6.geometry.parameters.width;
walter.body.position.y = 25;
walter.body.position.z = firstLevel.mesh6.matrixWorld.getPosition().z + firstLevel.mesh6.geometry.parameters.depth/2 -
                         fz * firstLevel.mesh6.geometry.parameters.depth;
```

3.1.3 Camera zoom animation

At the end of the first level, once the character is on the island, camera zooms in smoothly towards the character and a modal allowing to pass to the second level is shown.
This is achieved by our custom function *setupCameraPositionTween()* which takes as input the initial camera position, the final camera position and the duration of the animation.

```
var cameraXYZ = new THREE.Vector3(walter.body.position.x + 30, walter.body.position.y +10, walter.body.position.z +30 );
setupCameraPositionTween(camera.position, cameraXYZ, 5000);
```

Such function uses the tween.js library to perform the animation.

```
function setupCameraPositionTween( source, target, duration, delay, easing ){
    var l_delay = ( delay !== undefined ) ? delay : 0;
    var l_easing = ( easing !== undefined ) ? easing : TWEEN.Easing.Linear.None;

    new TWEEN.Tween( source )
        .to( target, duration )
        .delay( l_delay )
        .easing( l_easing )
        .onComplete(function() {
            clearModal();
            showModal("first level completed", "next");
        })
        .start();
}
```

3.2 Implemented interactions
In the first level we implemented three main interactions.

3.2.1 Clicking the wheel
When the user clicks the wheel, the already mentioned transformations happen (see paragraph 3.1.1) so that the mesh6 attaches to mesh3. The rotation of mesh3 and the wheel is achieved by using a tween animation.

3.2.2 Character is on the first elevator
Through the *updateFirstLevel()* function we periodically check whether the character is on the first elevator or not. If so, we disable the character movement and we scale and translate the elevator with a tween animation to achieve the effect of a sort of tower rising from the floor and to allow the character to reach the top of mesh6.

3.2.3 Character is on the second elevator
Through the *updateFirstLevel()* function we periodically check whether the character is on the second elevator or not. If so, we disable the character movement and we translate the elevator with a tween animation to achieve the effect of a small platform rising from the floor and reaching the island which represents the finish line.

4 Second level
4.1 Technical aspects
Technical aspects of the first level are described below.

4.1.1 Path creation
The path has been created as a tree model by using Group class.
The environment is divided into three distinct parts and to pass from one part to the other the character has to push some buttons placed on platforms that are slightly displaced from the main path.
On each edge of every block that is not connected to other blocks there are invisible walls that prevent the character from walking outside the path and that are made visible once meshes are joined.

4.1.2 Gravity Illusion
The peculiarity of this level is that the character walks on different planes as if the gravity changes while he changes area of the path.

4.2 Implemented interactions

In order to complete the level, the character has to reach buttons that transform the environment and change the path rotating the blocks so that the character can go from one area of the path to another.

4.2.1 First and second button

First and second button have a similar behaviour: function *updateSecondLevel()* checks whether the character is on one of the two buttons and in that case a tween animation translates the button in order to simulate the act of being pushed by the character. This action causes the rotation of the corresponding mesh: if button1 is pushed then mesh2 is rotated, if button2 is pushed then mesh5 is rotated.
Once the character leaves the button's coordinates, the button is translated back to its original position with a tween animation that takes 5 seconds.

4.2.2 Mesh2 and mesh5 rotation

When button1 is pushed the function *setUpMesh2RotatateAnimation(*) is called:

1. mesh2 rotates around the z axis by 90° degrees so that it is aligned to mesh1
2. in the callback *onComplete()* the visibility of walls placed between mesh1 and mesh2 is set to false so as not to constituting an obstacle anymore and allowing the character to move from one block to another

When the character leaves the button1 the function *setUpMesh2RotatateBack()* is called:

1. the function starts the animation with a delay of 5 seconds; in this way the character has enough time to reach mesh2
2. with the callback *onStart()* the visibility of walls is set to true so as to stop the character if it didn't reach mesh2 before the animation begins
3. the mesh rotates back
4. if the character has reached mesh2 on time its movement is blocked and mesh2 is rotated by using the function *setupMesh2RotateBackWithWalter()* which takes into account the rotation of mesh2 together with the character's one
5. when the block goes back to its original position the player can control the character again

Now the character is on the second part of the path and it walks on a different plane.
The behaviour of mesh5 is identical except that it moves when button2 is pushed and rotates around the y axis allowing the character to pass from the second to the third part of the path.

4.2.3 Third button

When the character reaches the third button its movement is blocked and the button is translated through the function *setupButton3Animation()*. In its onComplete() callback the function *setupMesh7RotateAnimation()* is called and mesh7 is rotated. The difference with the other buttons is that this one is not translated back to its original position because the mesh rotates only once.

4.2.4 Mesh7 rotation

When *setupMesh7RotateAnimation()* is called*:*

1. the callback *onStart()* calls the function to rotate the model of the cottage
2. the callback *onUpdate()* rotates the character so that it rotates together with mesh7
3. the callback *onComplete()* gives back to the player the ability of moving the character

Now the character is upside down and when he gets to the cottage the level is completed.

5 Third level

5.1 Technical aspects

It's important to notice that the castle as been created by our own as a tree model by using the Group class.

5.2 Implemented interactions

In the third level we implemented five main interactions.

5.2.1 Character is on the first and second elevator

They have been implemented like the first elevator of the first level.

5.2.2 Clicking the movable elevator

There are three main positions that the movable elevator can reach in a circular fashion every time the user clicks it in order to allow the character to move to different levels of the castle. This animation is achieved by translating this component along the y axis with the tween.js library.

5.2.3 Clicking the carriages

Every time the user clicks the carriages, they move from side to side of their rails so as to allow the character to pass to the other flank of the castle. This animation is achieved by translating this component along the x axis with the tween.js library.

5.2.4 Character is close to the first door

Through the *updateThirdLevel()* function we periodically check whether the character is close to the first door or not. If so we:

1. disable the keyboard control
2. make the character look towards the first door through the *lookAt()* function
3. call the *translateWalterInDoor()* that uses the *walkForward()* function to make the character walk inside the door. Once the end of the animation is detected through the *onComplete()* callback, it translates the character on the y axis so as to reach the second door, it makes the character look outside the second door and it calls the *translateWalterOutDoor()* function

4. the *translateWalterOutDoor()* function calls the *walkForward()* function to make the character walk outside the second door. Once the end of the animation is detected through the *onComplete()* callback, it returns the keyboard control back to the user.

Such mechanism, obtained through the tween.js library, has been thought to give the illusion of the character entering the first door, going upstairs and exiting the second door.

5.2.5 Character is on the button

Through the *updateThirdLevel()* function we periodically check whether the character is on the button or not. If so, we translate the button down on the y axis so as to give the idea of the button being pressed. In order to create a road towards the finish line, such button triggers the following animations:

1. the upper part of the towers rotates along the y axis
2. the two components constituting the final road are translated outside the upper part of the towers and they are joined
3. the final door comes out of the right tower by using a translation

Each of these animations is realized as a tween animation.