Interactive Graphics: Final Project
Dragon Playground

Master in Science of Artificial Intelligence and Robotics

Marco Vannoli Mat. 1860363 Daniele Chiaradia Mat. 1873647

11 luglio 2019

Introduction

The **Dragon Playground** offers a virtual theme park experience, where the attractions are built to work like their real versions.

The visitors will be able to walk inside the park and they can get on the attraction to see the view (*diurnal or nocturnal*) from the seats, changing the settings from the console.

Design and Analysis

To realize a theme park we need a powerful tools of 3D graphics.

For this reason, we decided to use the framework **ThreeJS**, which contains a lot of class which let us to create complex environments and set them animation and texture easily. In particular, we used Geometry Object to realize all park elements. Moreover, ThreeJS have good classes to manage the camera and light. In particular, for the realization of this project we used: **Camera**, **PointLight** and **AmbientLight**.

Our purposes consist in:

- allow visitors to walk around the park and to change orientation of their vision everywhere they want.
- realize realistic Attractions with their real movements through animations.
- allow to user to manage the typology of the vision and to change between the day and the night.
- create a realistic environment with the shadows of all elements.
- allow visitors to look the park with a panoramic view or from an attraction.

The attractions present in the park:

- Frozen Ferris Wheel,
- Volcano Eruption (Rotating Arms),
- Space Tower,
- Escape from Atlantis (inspired from *Fuga da Atlantide* of Gardaland).
- Duck Lake.

Finally, we also decided not to take online models created from others. Therefore each model and animation are made from us analyzing physic and operating features of the attractions that you can find in a real playground.

Camera

The typology of camera used is the **Perspective Camera** that allows us to obtain the vision like human eyes.

To instantiate a Perspective camera in ThreeJS we used the following code:

Source code 1: Initialization of the Perspective Camera

```
1 camera = new THREE.PerspectiveCamera(fov,aspectRatio,near,far);
```

The parameter **fov** represents the Camera frustum vertical field of the view, the **aspectRatio** stand for the Camera frustum aspect ratio, the **near** parameter describe the Camera frustum near plane and **far** is the Camera frustum far plane. These parameters were set in order to obtain a point of view closer to the vision of a visitor in a real playground.

After setting these parameters, it was necessary to set the position of the Camera on the axes x, y and z. To achieve this purpose, 3DObject of ThreeJS contains properties change the position and rotation in the space.

The 3DObjects of ThreeJS owns the position property followed by three parameters x, y and z that represent the axes. In this way we set the position of camera in the x, y and z direction.

The variables *camera_translation_x*, *camera_translation_y*, *camera_translation_z* are aimed at initializing the position of the camera. Their initial values are set in order to place the visitor in front of the playground.

To move the Camera and to change its orientation, you can change the position and rotation variables from the sliders or through some keys of the keyboard. The Javascript code to realize this movement is:

Source code 2: Move the Camera by keyboard

```
1  document.onkeydown = function(e) {
2      switch (e.keyCode) {
3          case 37: // Left arrow
4              document.getElementById("slider_camera_translaction_x").value =
5                document.getElementById("slider_camera_translaction_x").valueAsNumber - 1;
6              document.getElementById("span_camera_translaction_x").innerHTML =
7                  roundTwo(document.getElementById("slider_camera_translaction_x").
                       valueAsNumber);
8
9              camera_translaction_x = document.getElementById("slider_camera_translaction_x
                   ").value;
10             break;
11         case 38: // Up arrow
12         ...
13         // Similar code for Right arrow, Down arrow, W, A, S and D
14         ...
15         break;
16     }
17 };
```

The function modify the value of the sliders in order to set the limit beyond which the visitor can't go over. The value taken from sliders are managed in such way to obtain a movement corresponds to the scene.

Another important aspect of this project is the change of viewpoints in the scene. It was managed by radio buttons positioned on the console.

If the user changes the value of these radio buttons, it is called the function *manageViewCamera()* that initialize and call the right function of the selected viewpoints:

Source code 3: Change your viewpoint

```
function manageViewCamera() {
    switch ("" + viewType) {
        case "0": //first person
            if (isChangedViewType) {
                if (camera.parent != null) camera.parent.remove(camera);

                camera.rotation.z = 0;
                camera.position.y = 2;

                ...

                isChangedViewType = false;
            }
            firstPerson();
            break;
        case "1": //the panoramic view
        ...
        // Similar code for the attractions
        ...
        break;
    }
};
```

This function manages the selected viewpoint: if the Boolean variable *isChangedViewType* is set to true, the code initializes the new view removing the camera from its parent and joint it to the right object. Finally, the position and orientation are set with the new value. After the initialization the function calls the right update function of the selected view in order to obtain a realistic vision. Two examples of update functions are:

Source code 4: Some update view function

```
function firstPerson() {
    camera.rotation.x = camera_rotation_x;
    camera.rotation.y = camera_rotation_y;
    camera.position.x = camera_translaction_x;
    camera.position.y = camera_translaction_y;
    camera.position.z = camera_translaction_z;
}

function panoramicView(){
    if(camera.rotation.y == 0 && camera.position.z > 0)
        camera.position.z -= 1;
    else if(camera.rotation.y < 2 * Math.PI && camera.position.z == 0)
        camera.rotation.y += Math.PI / 250;
    else if(camera.rotation.y > 2 * Math.PI && camera.position.z < 200)
        camera.position.z += 1;
    else if(camera.rotation.y > 2 * Math.PI && camera.position.z == 200)
        camera.rotation.y = 0;
}
```

The **First Person** function update the position and orientation of the camera.

The **Panoramic** function change automatically the values of positions and rotations of the camera in order to have a global view of park.

The point of view of the attractions was created with the same way but in this cases the camera is moved by the animation of the parent.

Light

A very interesting part of this project concern the light.

Our aim was to introduce shadows and lights in order to get a realistic environments with the possibility to change between day and night.

ThreeJS contains some class to realize the illumination. In particular, we used **PointLight** and **AmbientLight**.

The **PointLight** class introduces a light source which illuminates in every directions. We decided to collocate this point behind the start position of the visitor, to have a good illumination for all attraction and to improve the vision of the park.

The class **PointLight** also allow the management of the shadows. To bring up the shadows each object of the scene must respect two conditions:

- the material of the object must be given from the class **TREE.MeshPhongMaterial** (*to show the shadow zone and the light zone on itself*)
- the parameters ***receivedShadow*** (*to show on the surface the shadows of surrounding elements with the parameter* ***castShadow*** *set to true*) and ***castShadow*** (*to show the shadow of the element on the surface of the surrounding elements with the parameter* ***receivedShadow*** *set to true*) of each elements of the scene need to be set to true.

The main light source is initialized with the following code:

Source code 5: Initialization of the day light

```
1  var light = new THREE.PointLight(0x777788, intensity_day);
2
3  light.position.set(0, 100, 400);
4
5  light.castShadow = true;
6  light.shadow.camera.near = 0.1;
7  light.shadow.camera.far = 100000;
8  light.shadow.mapSize.width = 4096;
9  light.shadow.mapSize.height = 4096;
10
11 scene.add(light);
```

At the row 1, the object is instantiated. The constructor need two parameters: the first represent the color of the light and the second intensity.

The rows 5 to 9 allow the appearance of the shadows and improve their rendering.

To change between day and night, we can obtain a good result changing the intensity of the light. If the user click on the button the web page run the following code:

Source code 6: Changing between Day and Night

```
1  if(day){
2      light.intensity = intensity_day
3      scene.background = new THREE.Color(color_day);
4      scene.fog = new THREE.Fog(0xffffff, 0, 750);
5  } else {
6      light.intensity = intensity_night;
7      scene.background = new THREE.Color(color_night);
8      scene.fog = new THREE.Fog(0x000000, 0, 750);
9  }
```

At the rows 2 and 6, we change the intensity of the light.

At the rows 3 and 7, we change the color of the sky.

At the rows 4 and 8, we change the color of the fog present in the scene.

Thanks this solution, we achieved the aim to add the shadow but the areas hidden from the light was very dark.

To resolve this problem we used the class **AmbientLight** that apply a light everywhere in the scene.

Source code 7: Ambient light

```
1 var ambientLight = new THREE.AmbientLight(0xffffff, 0.6);
2 scene.add(ambientLight);
```

At the row 1, the object is instantiated. The constructor need two parameters: the first represent the color of the light and the second the intensity (*like PointLight*). Since the PointLight is already illumining the scene, we set the intensity of this light a low value to give a low light to the dark areas.

Finally, we needed to improve the light during the night.

Hence, we added some PointLight to illuminate the attractions to continue the experience even though the dark.

These new PointLight are put in an array and during the change from day to night (*and vice versa*) the web page update also their intensity.

Source code 8: Night lights

```
1 lights.forEach(function(element) {
2     element.intensity = (day? 0 : light_points_night)
3 });
```

During the day the intensity of these lights is 0 (*light off*) otherwise we put the float value of the variable **light_points_night** (*light on*).

Scene and Models

To realize the environment we used much Geometry classes of ThreeJS.

The obtained object are related to get some **hierarchical models** and **groups** (*from the class **Group** provided by ThreeJS*). In particular, we used the **groups** to aggregate all elements belonging to the same category that don't influence the motion of the other elements of the group. Whereas, all elements that influence the motion of the other elements of the group we joined them in a hierarchical models. To realize this connection we add the child element to the parent element by the method *add* (*of all 3DObject of Three*).

Foreground

The first important structure of environment is the foreground. The foreground was created to include all elements that make up the base of the park:

- the infinite ground,
- the trees,
- the base of playground,
- the fence and the decorations:
  - the fences,
  - the castle tower,
  - the dragons.

The trees are positioned with random coordinates in the scene, but we forced them to appear around the park.

Source code 9: Choice of the trees coordinates

```
1 do {
2     tree.position.x = Math.random() * 1600 - 800;
3     tree.position.z = Math.random() * 1600 - 800;
4 } while (
5         tree.position.x > -(sizeParkX / 2) - 40 &&
6         tree.position.x < (sizeParkX / 2) + 40 &&
7         tree.position.z > -(sizeParkZ / 2) - 10 &&
8         tree.position.z < 210 );
```

Until the coordinates of the axes x and z of a tree is out from the range of the park the cycle try again with new random values.

The dragons are collocated on the sides of the front fence of the park. Their structure are divided in three parts:

- the **body** consist of an object of the class **TorusGeometry** (*provided by ThreeJS*)
- the **neck** consist of an object of the class **TubeGeometry** (*provided by ThreeJS*)
- the **head** consist in a group with the muzzle, mouth (*upper and lower*), teeth and fire.

This three parts are connected in a hierarchical model.

Escape from Atlantis

The attraction Escape from Atlantis was created starting from group that joins all the elements which make up the structure:

- the **base** consists of an object of the class **BoxGeometry** (*provided by ThreeJS*). It is joined to the group of Escape from Atlantis.

- the **slider** consists of an object of the class **BoxGeometry** (*provided by ThreeJS*) rotated in order to obtain the desired inclination. It is joined to the group of Escape from Atlantis.
- the **water planes** consist in an object of the class **PlaneGeometry** (*provided by ThreeJS*) in order to follow both base and slider. They is joined to the group of Escape from Atlantis.
- the **fence plane with water** consist in two object of the class **CylinderGeometry** (*provided by ThreeJS*) and **PlaneGeometry** (*provided by ThreeJS*). They are joined to the group of Escape from Atlantis.
- the **fence which closes the bases** consist in one object of the class **TorusBufferGeometry** (*provided by ThreeJS*) but clipped in half. It is joined to the group of Escape from Atlantis.
- the **dome** consist in a group which includes the four pillars, the roof and decorative red circle. The pillars are created with an object **CylinderGeometry** (*provided by ThreeJS*), the roof with an object **ExtrudeBufferGeometry** (*provided by ThreeJS*) whereas the decorative red circle consists of two object (one **CircleGeometry** (*provided by ThreeJS*) and one **TorusBufferGeometry**). The group that joins this elements of dome pillars is joined to the base.
- the **wagon** consist in one object of the class **CylinderGeometry** (*provided by ThreeJS*) in which another object of the class **ConeBufferGeometry** (*provided by ThreeJS*) is joined. The whole wagon is joined to the group of Escape from Atlantis.

Each part described is joined to the group which collects the entire structure in order to compact it. Finally the whole structure of attraction is added to the scene.

The animation of the wagon is obtained with a function that evaluates the depth of the object and manages accordingly the correct position and rotation in order to give a realistic movement.

Space Tower

The Towers is the highest attraction of the park.

It is composed of:

- **the base of the tower** consists in an object of the class **BoxGeometry** placed on the ground.
- **two towers** consist in objects of the class **CylinderGeometry**. They are joined to the base.
- **the orthogonal tower** consists in an object of the class **BoxGeometry** that connect the two tower on the top. It is joined to one of the tower
- **two seats** consist in objects of the class **TorusGeometry** collocated around one of the two towers. They are joined to the the own tower.

To realize the animation of this attraction we change the altitude of the seats with different speed in ascent and descent phase.

This attraction presents spatial textures. For this reason we decided to call it **Space Tower**.

Frozen Ferris Wheel

One of the first attractions that you can see when you start the visit is the Ferris Wheel collocated at the center of the park.

This attraction is composed of tw hierarchical models:

- the **base of the Ferris Wheel**: which consist of the structure that support the wheel, which start from the ground to the center of the attraction. This model don't needs animations during the visit. The element that compose the base of the Ferris Wheel are:
  - **base** consists in an object of the class **BoxGeometry** placed on the ground,

  - **four oblique supports** consist in an object of the class **BoxGeometry** to keep up the Wheel. They are joined to the base.

- the **Ferris Wheel**: which rotate with the carriages. This part is composed of:

  - **the cylinder** consists in an object of the class **CylinderGeometry** placed at the centre of the Wheel on the oblique supports.
  - **two circles** consist in objects of the class **TorusGeometry** that delimit the Wheel. They are joined to the cylinder.
  - **the spokes** consist in objects of the class **BoxGeometry** that connect the cylinder to the circles. They represent the radius of the circumference. They are joined to the cylinder.
  - **the orthogonal spokes** consist in objects of the class **BoxGeometry** that connect the spokes (radius) to improve the structure. They are joined to the own spokes (radius).
  - **the roofs of carriage** consist in objects of the class **CylinderGeometry**. They are joined to the last orthogonal spokes (that connect the two circles).
  - **the carriages** consist in objects of the class **ExtrudeGeometry**. They are joined to the own roof.
  - **four windows** consist in objects of the class **BoxGeometry**. They are joined to the own carriage.

  To realize this part of the Ferris Wheel we used a for cycle in order to create the same structure (*spokes, orthogonal spokes, roof of carriage, carriage and windows*) for each carriage and rotate it in order to obtain the same distance between the carriages.

Source code 10: Cycle to add carriages on the Ferris Wheel

```
1  for(var rad = 0; rad < Math.PI - 0.001 ; rad += Math.PI / 6) {...}
```

  The variable rad contains the value of rotation of the structures.

To realize the movements of this attraction we rotated the cylinder in order to move the carriages. In this way the carriages are also rotated. Hence, we applied an inverse and equal rotation to each roofs in order to maintain only the translations.

Finally we decided to apply some ice Textures. This is the reason because we called this attraction **Frozen Ferris Wheel**.

Volcano Eruption and Duck Lake

The two other attractions of the park present a similar structure of the Ferris Wheel: there is an object at the center, we joined it some supports which have at the extreme the seats (*in the case of the **Volcano Eruption***) or the Duck (*in the case of the **Duck Lake***).

The animation of the **Duck Lake** consist in the rotation of the invisible object at the center of lake.

Whereas, the animation of the **Volcano Eruption** is more interesting. The situation is similar to the **Frozen Ferris Wheel** but in this case the seats have two different movements:

- the first is given by the rotation of the object at the center that translate the seats on a circumference.
- the second is given by the rotation of the arms that change the seat altitude and the orientation. Hence, to resolve this problem, we applied a inverse and equal rotation to each seats in order to maintain only the translations.

The name **Volcano Eruption** is given by the applied texture on the attraction.

Texture

The Textures of project are managed through base64 images. All image texture are converted into the format base64 and elaborated with the following code:

Source code 11: Initialize of a Texture

```
1  var image_64 = new Image();
2  image_64.src = "..."
3
4  var Texture = new THREE.Texture();
5  Texture.image = image_base64;
6  image_base64.onload = function() {
7    Texture.needsUpdate = true;
8  };
9
10
11 Texture.repeat.set(1,1);
12 Texture.wrapS = Texture.wrapT = THREE.MirroredRepeatWrapping;
```

The Texture object is instantiate and then it is set its property *image* with the base64 image. Afterwards the flag variable *needsUpdate* of Texture is set to true to trigger an update next time the texture is used; After this is set in Texture the property *repeat* that represents how many times the texture is repeated across the surface, in each direction U (horizontal mapping) and V (vertical mapping). The value of this property is set in order to obtain the best drawing up of the texture on an object. For example in a complex object like a dragon it was used a different drawing up of the texture with the same image, one repetition of the surface texture of the neck and one for the body of dragon. Finally, the texture properties wrapS and wrapT that correspond how the texture is wrapped horizontally/vertically and corresponds to U/V in UV mapping are set. Moreover, a Wrapping Modes of type MirroredRepeatWrapping is set because allow texture to repeat to infinity. The texture was created and we can put it on the material of object with the parameter *map*:

Source code 12: Apply a Texture on an object

```
1  var material = new THREE.MeshPhongMaterial( { map: Texture } );
```

User Manual

The **Dragon Playground** is totally visitable.

The user can select the typology of view from:

- **First Person**: this typology of view allow the visitor to move around the attractions and change the orientation of the view. The user can change this value from the **sliders of Position and Orientation of the Camera** (*present in the console on the right*) otherwise use some **keyboard keys**:
  - **up arrow**: to go forward
  - **down arrow**: to go back
  - **left arrow**: to go left
  - **right arrow**: to go right
  - **W**: to change the orientation of camera upwards
  - **S**: to change the orientation of camera downwards
  - **A**: to change the orientation of camera to the left
  - **D**: to change the orientation of camera to the right

- **Panoramic**: this typology of view allow the user to see the park from the air. *In this case the user can't interact with the view changing position and orientation of the camera.*
- **Frozen Ferris Wheel**: this typology of view allow the user to see the park from a carriage of the attraction. *In this case the user can't interact with the view changing position and orientation of the camera.*
- **Volcano Eruption**: this typology of view allow the user to see the park from a seat of the attraction. *In this case the user can't interact with the view changing position and orientation of the camera.*
- **Space Tower**: this typology of view allow the user to see the park from a seat of the attraction. *In this case the user can't interact with the view changing position and orientation of the camera.*
- **Escape from Atlantis**: this typology of view allow the user to see the park from a wagon of the attraction. *In this case the user can't interact with the view changing position and orientation of the camera.*
- **Duck Lake**: this typology of view allow the user to see the park from the eyes of a duck. *In this case the user can't interact with the view changing position and orientation of the camera.*

In every moment the user can change the part of the day by the button *change to night* or *change to day.*

Conclusion

The Dragon Park respected our porpoises about the creation of a simulation of a theme park experience.

The structures of environments and attractions are similar to as realistic as possible.

The whole project is focused to graphic appearance with the possibility of interaction with the attractions through the console.

With some arrangements we achieved graphic effects with a good frame rate.

References

- Documentation of ThreeJS:

  `https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene`

- examples of ThreeJS:

  `https://threejs.org/examples/#webgl_animation_cloth`

- Code for triangular prism:

  `https://stackoverflow.com/questions/27193732/three-js-creating-a-right-triangular-prism`