De Filippis Stefano 1707247

Final Project

1. Concept

The project consist in the development of game of the endless run type. The main character of the game is a dragon, which can be chosen at the start of the game, and the player can control it with the directional arrows in order to survive along a straight path where there are obstacle in the form of towers and arrows. Moreover, the dragon can throw fire, whose power increases during time, and based on how much strong it is the player can destroy some obstacles and gain additional points.

2. Libraries

The libraries used for the development of the project were:

- Three.js for the graphic modeling and rendering.
- Tween.js for the animation of the models.
- Howler.js for the handling of sound in the application.
- Bootstrap for the htlm files.
- JQueary, needed in order to make Bootstrap works.
- Orbit.js for easier control of the camera in three.js

The dragon model and the fire particles, instead, where originally inspired and then quite modified starting from the example of Karim Maaloul present at https://codepen.io/Yakudoo/pen/yNjRRL

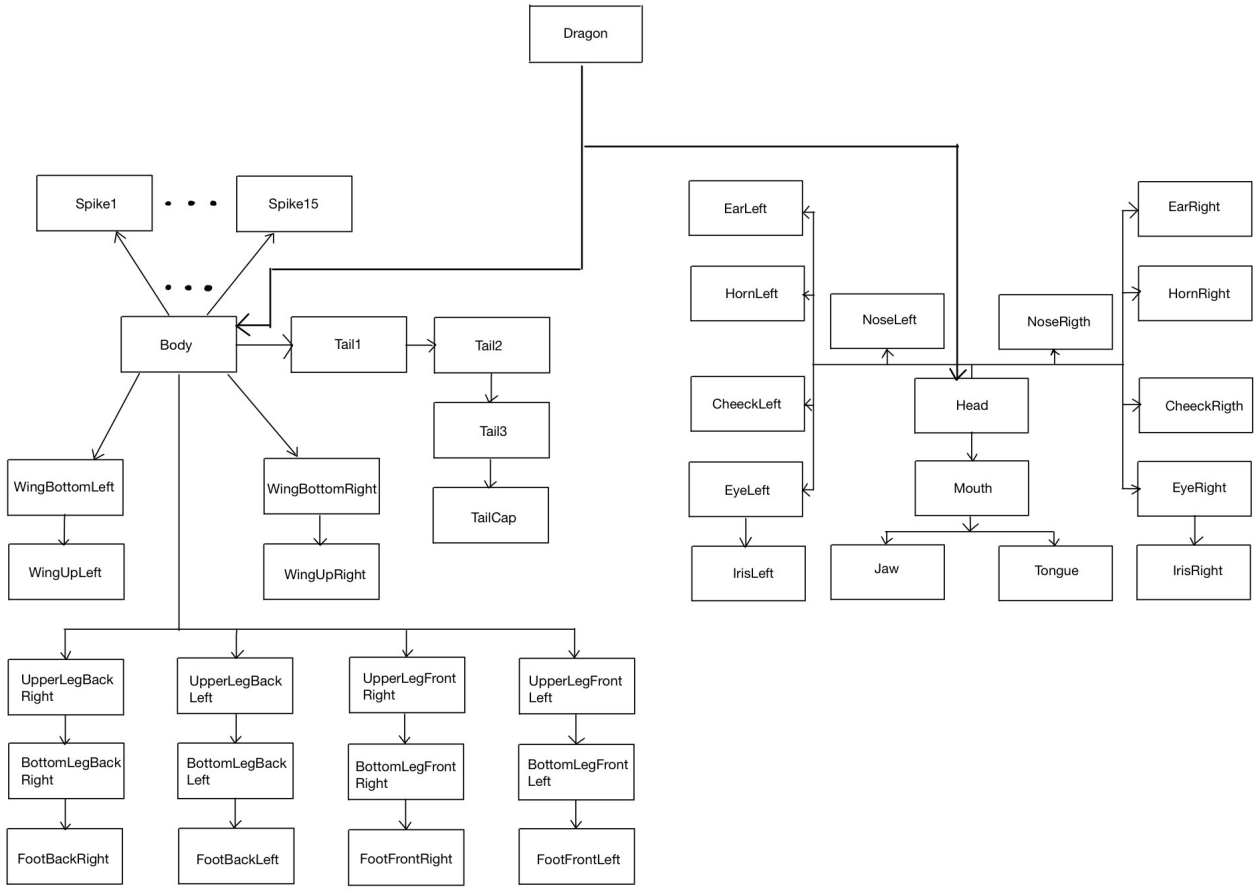
3. Hierarchical Models

In the game there are essentially three main hierarchical model: the dragon, the towers and the arrows. All of them use as mesh material a MeshLambertMaterial, in order to reflect the lights and their structures are the following.

3.1 Dragon

The dragon is composed by several parts, but the main parts are the body and the heads which are then united in a single group. All the dragon parts are made by simple geometries primitives defined in three.js and in particular they belong to either the BoxGeometry or CylinderGeometry class. To the body are attached 15 spikes, the tails (which is composed by 4 parts), the wings (each composed by 2 parts) and the legs (each composed by 3 parts plus the 3 spikes on the foot). On the other hand, to the head are attached the eyes (made by

2 parts), the ears, the cheeks, the nose nostrils and the mouth (made by the jaw and the tongue).

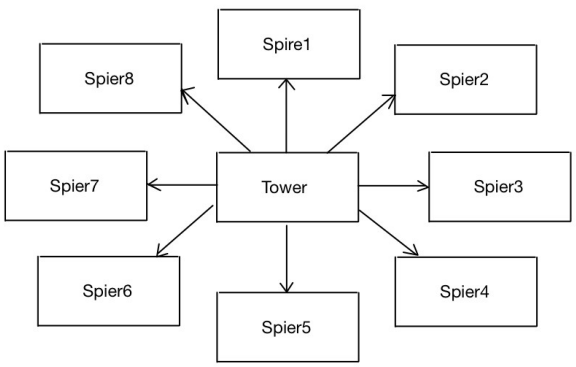


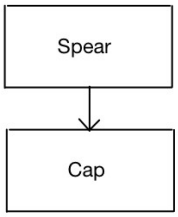
3.2 Tower

The tower model is quite simple and it is composed by a cylinder which is the base and attached to it there are 8 spiers. In this case I applied a simple texture that I needed to resize in order to have sizes as in power of 2 in order to be more computational efficient. Moreover , I needed to wrap the texture along the object in order to render it correctly and the best solution was applying the mirrored repeat wrapping along both s and t directions, 5 times each.

3.3 Spear

The spear model is simply composed by a rod which is the body and on top of it there is a small cylinder cap.





3.4 Other models

In addition to the three hierarchical models just illustrated, in the application there are also other objects.

- The mountains: simply modeled as cones with squared bases.
- The pavement: is made by 2 consecutive parallelepiped that translate on the z-axis and create the illusion of the moving pavement. Also in this case a texture is applied, using the same criteria of the tower texture but repeating 4 and 60 times respectively in the s and t directions.
- The fire of the dragon: it is modeled as cubes that changes orientation and position when the fires is thrown. In this case the object are not treated as a single system.
- The explosion particles: very similar in concept to the fire of the dragon, but now they are treated as a particle system.

4. Animations

All the object described in the previous section are animated in some way. The towers, spears and mountains simply translate along the positive direction of the z-axis and more will be said in the following logic section. The dragon, the fire and the explosion, on the other hand, display some more articulated animations.

4.1 Dragon Normal Animation

When the dragon does not throw fire or is moved by the player, the main parts that are animated are the body , the wings and the tail.

- The body is translated along the y-axis following a periodic sine function that is slightly shifted in phase with respect to the wings so that the propulsion effect of the wings on the body is rendered in some way.
- The bottom part of the wing, instead, rotates around the z-axis following the function (-/+) π / 3 (+/-) cos(wingAngle) * wingAmplitude where the signs are related to the update of the left or right bottom wing respectively. Moreover, the top wing parts follow the function (-/+) sin(this.wingAngle) * wingAmplitude. Important is the 90 degree shift in the phase between the bottom and upper part of the wing which render a more natural animation of the wing.
- All the three parts of the tails rotates around the x and z axis following periodic functions shifted of 90 degrees between them so that their trajectory in the space is

similar to an ellipse. The main difference between the 3 parts is the amplitude of the rotations that decrease following the power of 2 from the first part to the last one.

4.2 Dragon Fire Animation

When the dragon needs to throw fire, its animation is subdivided in 3 main parts: prepare to fire, fire and back to normal. All the animations are handled through interpolation using the tween.js library and the amplitude of rotations and translation of all the parts are dependent on the power of the fire accumulated up to that point during the game. Therefore more power accumulated implies a more evident animation of the dragon when throwing fire.

4.3 Dragon Player Control

This is the animation that the dragon is subject to when the player controls it through the directional arrows. When the left or right keys are pressed, the dragon needs to translate along the negative or positive x axis direction and, in addition, it also needs to rotate around the z axis of a maximum of – 45 degrees and then return to its initial orientation. When the up and down keys are pressed, instead, the dragon body simply translates along the positive/ negative y axis direction. All the translations are handled by simple interpolation, while the rotation is handled through a Bezier curve that allowed to specify the four control points needed to start at 0 rotate to -45 and go back to 0 degrees.

4.4 Fire Animation

The fire animation is also divided in 2 main phases: in a first phase the actual fire particles are modeled in order to render the fire and afterwards they are modeled in order to render the effect of smoke billowing from the dragon’s nostrils. During the fire phase, the particles are created with the initial position related to the mouth of the dragon and through interpolation they change:

- Position along the tree axis.
- The rotation along the y and x axis.
- The scale along the three axis, determined through a Bezier curve and four control points, which are dependent on the fire power accumulated along the game and a random value.
- The color of the mesh that depends on the dragon chosen and handled through a Bezier curve that changes the intensity of the color until it is completely black.

The animation in the smoke phase is in principle very similar to the fire phase, but the proportions of the scaling change, the initial position of the particles is at either one of the

dragon’s nostrils and the color of the particles only changes the opacity, transforming the black cube into a transparent one.

On every cycle of the fire animation, an array of fire particles are created and then they first fly and then they are rendered as smoke particles.

4.5 Explosion Animation

The explosion effect is rendered through a particle system. Therefore, when an explosion animation needs to be created a certain number of vertices are created, along with a vector of deltas along the three axis directions, and then they are gathered in a particle system defined by three.js where we need to specify the color and the size of the particle. At the start the system is positioned at the same place of the reference frame of the object that is being destroyed and in every frame the position of the particles along the three directions are updated by using the deltas defined at the creation of the system. After a fixed number of frames, the animation is not rendered anymore, otherwise it would be infinite.

5. Camera And Projection

The projection used in the application is the prospective projection, while the camera is always positioned at some distance behind the dragon and at a height slightly higher that the dragon’s. Moreover, The lookAt vector is positioned at some point in the air and way in front of the dragon position, so that the lane in front can be seen and the dragon does not occlude completely the camera view. Finally, the camera pose can not be modified directly by the user, but the camera can change its position along the y axis when the dragon is moved along the same direction with the up/down arrow keys.

6. Lights

In the application there are 3 different sources of lights. The first one is an hemisphere light, so the light is global and in every direction. Then there are two directional lights: one is positioned behind the dragon and it is used in order to see the contrast and shadows on the body of the dragon (otherwise the body would have been to dark to distinguish the different parts). The last directional light is positioned in front of the dragon and it is used in order to render the shadows effect of the towers and arrows.

6.1 Fog And Shadow

In order to mask the insertion of the object in the distance during the game, a fog effect is used together with the camera. Therefore, the objects beyond a certain distance are seen as if gray and as they get closer their real mesh is displayed.

As far as the shadows are concerned, instead, only the tower and arrows shadows are rendered, while dragon's is not. This is due to a more computational efficient approach since the light for the shadow is in front and the dragon shadow would not have been shown anyway so I decided to not render it from the start.

It is important to notice that in three.js, the shadow of the directional light is computed using a orthographic projection seen from a camera associated to the directional light. Therefore, it has been quite challenging finding the right position for the light and the frustum proportion of this kind of projection in order to render correctly the shadow and also render the shadow of the object in the distance while keeping the computation feasible in the GPU. In the end there is still an effect of shadow being created on the floor when the object are far along the path, and this is due to the fact that the object in that moment are entering the orthographic projection camera view volume, but the problem is reduced by the fog previously described. Actually a better solution would have been enlarging the view volume of the projection, but this would have also meant increasing the quality of the shadow map and this lead to freezing problem between frames and the overall quality of the application was severely compromised.

7. Game Logic

The application itself is made up by 2 main file, the index.html and the game.html.

7.1 Index

index.html is the starting page of the application. Here the player can select the dragon that he prefers out of 3 choices. The difference between the dragons is mainly the color of the mesh and respectively the color of the fire. From the index to the game file the parameters needed to start the game are the color of the dragon chosen and the level of difficulty and both are passed to the game file by adding them as parameter of the GET request.

7.2 Game

In this other file, instead, there is the logic and the rendering loop of the application.

The main skeleton of the game consist in a lane where towers and arrows are placed and the dragon needs to survive as long as possible.

7.2.1 Lane

The lane is formed by 2 parallelepiped, to which an image texture is applied, placed one after the other and they are translated along the z axis (in the positive direction) during each frame. As soon as one of the planes is in completely in the plane defined by $z > 0$ then it is

translated back at the end of the other planes in order to have an effect of continuity. It is important to notice that since I applied an image texture on the pavement I also need to render the effect of a continuous sliding floor and this could not have been possible by using a single plane. Finally, the lane horizontally is divided in 3 parts and the dragon initially starts in the middle, but can move in each one of three sub lanes.

7.2.2 Towers

The towers are of three different heights and they specify the division of the vertical “virtual” lanes that the dragon can occupy. The highest vertical lane does not allow the character to avoid the highest tower by flying over it, otherwise it would have been too simple.

The percentage of towers along the path is fixed across all different levels of difficulty, while the position in the lane and the height is chosen through a pseudo-random number generator. In every frame the towers are translated along the positive z direction and if a tower is positioned in the plane defined by $z > 0$ then it is translated at the end of the lane. For simplicity of computation all the tower instances are gathered in an array.

7.2.3 Spears

Every spear is associated to a tower, but not every tower needs to have a spear and the percentage of spears is a fixed number across the various levels of difficulty. Every spear when it is created is not visible. Only when the corresponding tower is at some distance from the character, the spear is added to the scene and it is translated along the positive z direction until is completely in the plane defined by $z > 0$, then it is removed from the scene and the whole process happens cyclically.

7.2.4 Mountains

The mountains are simply cones with a square base placed alongside the pavement. Their height is chosen randomly from a fixed set of values and every frame they are translated along the z positive direction until they are not visible anymore and therefore they are placed back at the end of the lane.

7.2.5 Dragon

The dragon effectively is still along the lane, since it is all the surrounding world that translate along the z axis. Its only movements are the animation of the body previously described, the change of lanes both vertically and horizontally based on the player input and the throwing of the fire.

7.2.6 Points

The points in the game are simply the distance covered by the dragon from the start of the game and which corresponds to the continuous translation of the world of a fixed quantity every frame. Moreover, the player can gain additional points by destroying the towers which are proportional to the height of the tower demolished.

7.2.7 Difficulty

In the game there are three level of difficulties to choose from and the difference between them is simply the speed at which the world translates.

7.2.8 Collision Detection

In order to be efficient computationally the quickest algorithm to determine whether two game entities are overlapping or not is the use of the axis-aligned bounding boxes (AABB). The overlapping area between two non-rotated boxes can be checked with logical comparisons alone. Checking if a point is inside an AABB is pretty simple — we just need to check whether the point's coordinates fall inside the AABB; considering each axis separately. If we assume that P_x, P_y and P_z are the point's coordinates, and B_{minX}–B_{maxX}, B_{minY}–B_{maxY}, and B_{minZ}–B_{maxZ} are the ranges of each axis of the AABB, we can calculate whether a collision has occurred between the two using the following formula:

$$(P_x >= B_{minX} \wedge P_x <= B_{maxX}) \wedge (P_y >= B_{minY} \wedge P_y <= B_{maxY}) \wedge (P_z >= B_{minZ} \wedge P_z <= B_{maxXZ})$$

When a collision between a fire particle and an object present on the lane is detected, if the fire power is enough to destroy the object, a new explosion animation is created and the animation is updated in every frame until it is allowed. On the other hand if a collision between an object and the dragon is detected, then the game is finished.

7.2.9 Fire Power

The fire power is a value between 0 and 10 and increases linearly with the passing time. When the fire power is greater than 2.5 than the smaller towers can be destroyed. When the fire power is greater than 5 also the medium towers can be destroyed. Finally when the power is greater than 7.5 all kind of towers can be destroyed. On the other hand, the spears can be always destroyed. Once the dragon throws the fire the the power is again set to 0.

7.2.10 Main Loop

In the following section I will show in pseudo code the main rendering loop of the application:

```
loop () {
    if (!paused and !game) {
        dragon.update()
        fireCollisionLogic()
        FireLogic()
        towersLogic()
        spearsLogic()
        explosionLogic()
        pavementLogic()
        mountainsLogic()
        updatePoint()
    }
    render(scene)
    loop()
}
```

So, if the game is still going on, the first thing I do is update the dragon normal animation (**4.1.1**), then I check if the dragon is throwing fire and if there is a collision between some object and any fire particle. Afterwards I try to render the animation of the throwing fire if necessary and update the towers and spears position and game status. Finally I check if I need to render some explosion animation, update the position of the pavement's planes and the mountains and update the score of the player.

8. Commands

In this final section I will report the commands needed to the player to play the game. The commands are really simple since the player only needs to use the 4 directional arrow keys on the keyboard to change the position of the dragon along the vertical and horizontal lanes, and the space bar to throw the fire. Finally, by clicking the P on the keyboard the game is paused and some option and possible commands are displayed.