Interactive Graphics - Final Project

Andrea Luca Antonini, 1707560

25 settembre 2020

1 Ply (a pig that fly)

This project consists in the implementation of a videogame in which a flying pig (Ply) have to eat his food and avoid incoming dangers.
The goal of the user is move Ply in order to take the food (spherical objects) and avoid the dangers (sharp objects).
Everytime Ply catches a food, it gains 5 points. Every 100 points, level is incremented, such as game speed and difficulty. Ply has 3 lives and everytime it catches a danger, lives are decremented by one. After three lost lives, game is over. Pig's movement are controlled through up and down cursor movement (with mouse or touchpad). The user can pause and resume the game in every moment pressing the space bar.
There is a music background for the game and one for the pause situation: both can be muted or regolated in the volume.

2 General background

2.1 Environment

The projected is in WebGL (Web Graphics Library). WebGL is a Javascript API for rendering interactive 2D and 3D graphics within any compatible web browser without using plugins.

2.2 Third-party libraries

- **Three.js**: a cross-browser Javascript library and API used to create and display animated 3D computer graphics in a web browser using WebGL. The inclusion in the project is done through the javascript file *three.js*, then in *index.html* this file is included as script file.
- **Google Font API**: web service that supports high-quality open source font files used in CSS file (called *ply.css*).

3 Implementation

3.1 HTML

The only HTML file present in this project is *index.html*. In this file, there are both vertex shader and fragment shader.

```
<script id="vertex-shader" type="x-shader/x-vertex">
#version 300 es

in vec4 aPosition;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

in vec2 vCoord;
out vec2 fCoord;

void main() {
  fCoord = vCoord;
  gl_Position = projectionMatrix * modelViewMatrix * vec4(aPosition, 1.0);
}
</script>
<script id="fragment-shader" type="x-shader/x-vertex">
#version 300 es

in vec2 fCoord;
out vec4 fColor;

void main() {
  fColor = vec4(vec3(fCoord,0.0),1.0);
}
```

To open the game, user has to open HTML file.
At the beginning, a div called *presentation* is displayed, in which game instructions are presented. There is only one button, i.e. *startButton*. If this button is pressed, another div (*scenario*) is showed and the game starts.
In this second div, we have two buttons, i.e. *pageReset* and *muteAudio*, that are used respectively for going back to the initial div and for mute the audio. Moreover, in this div there is a table, showing points, level and remaining lives, and a slider called *volumeSlider* used to regolate sound volume. The interaction with buttons and slider are managed in the Javascript file *ply.js*.

3.2 Javascript

3.2.1 preInit() function

preInit() function is called on the page load, i.e. is the first function to be called. In this function, we have a *startButton* listener: if the button is pressed, init() function is called.

3.2.2 init() function

init() function is called by preInit() when *startButton* is pressed.
In this function, div *scenario* is made visible and there is the initialization of all the components of the actual scenario. In fact, in the *initialize()* function are initialized the colors and the game variables used in the various functions. These game variables regard camera, pig, food and dangers parameters.
Other functions called in the *init()* function are used to create the scene (*create-*

Scene()), lights and shadows (*createLightsandShadows()*), grass (*createGrass()*), sky (*createSky()*), pig (*createPig()*), food (*createFood()*), dangers (*createDanger()*) and to set the music (*ambientMusic()*).
Finally, the *loop()* function is called.

3.2.3 loop() function

```
function loop(){
  newTime = new Date().getTime();
  gameVariables.deltaTime = newTime - oldTime;
  oldTime = newTime;
  if(gameVariables.play) {
    if (Math.floor(gameVariables.distance) % gameVariables.distanceAddFood == 0 && Math.floor(gameVariables.distance) > gameVariables.foodLastAdd){
      gameVariables.foodLastAdd = Math.floor(gameVariables.distance);
      foodOwner.addFood();
    }
    if (Math.floor(gameVariables.distance) % gameVariables.distanceAddDanger == 0 && Math.floor(gameVariables.distance) > gameVariables.dangerLastAdd){
      gameVariables.dangerLastAdd = Math.floor(gameVariables.distance);
      dangerOwner.addDanger();
    }
    if(gameVariables.life == 0){
      gameVariables.isGameOver = true;
      showGameOver();
    }
    updatePig();
    zoom();
    updateDistance();
    gameVariables.speed = gameVariables.baseSpeed * gameVariables.pigSpeed;
    foodOwner.foodAnimation();
    dangerOwner.dangerAnimation();
    grass.mesh.rotation.z += gameVariables.grassRotation;
    sky.mesh.rotation.z += gameVariables.skyRotation;
    renderer.render(scene, camera);
    requestAnimationFrame(loop);
  }
}
```

In the *loop()* function, there is the management of the creation of foods and dangers. In fact, through Food, FoodOwner, Danger and DangerOwner functions, all the creation (through Three.js), the speed and the frequence of appearance are managed.
Moreover, in *loop()* function we have the function *updatePig()*, that manages pig movements, and the function *zoom()*, used to create the zoom effect based on the cursor movement toward right (zoom out) or left (zoom in).
Finally, in *loop()* function we determine the grass and sky rotations.

```
function fly(mouse, minM, maxM, minR, maxR) {
  var border = Math.max(Math.min(mouse, maxM), minM);
  var middleM = maxM - minM;
  var delay = (border - minM) / middleM;
  var middleR = maxR - minR;
  var result = minR + (delay * middleR);
  return result;
}

function zoom() {
  camera.fov = fly(mouse.x, -1, 1, 70, 90);
  camera.updateProjectionMatrix();
}
```

3.2.4 Pig

Pig is the first hierarchical model present in this project.
Pig model is composed by body, head, legs, tail, eye, ear and wings.
I applied to all the components a PhongMaterial shader to give the wanted effect of material (color, opacity, transparency and so on). Moreover, all the components are made up with a mesh function of geometry and material.
Pig model is set to emphasize the shadow given by the directional light.
The body is an oval object built with *SphereGeometry* of *Three.js* and then scaled. It is the principal component of the hierarchical model.
The head is a spherical object (always *SphereGeometry*) that has been positioned ahead of the body. On the head we can find the eye (*SphereGeometry*), the ear (*CylinderGeometry*) and the snout (*BoxGeometry*).
Going ahead, we find the back and front legs, realized with *BoxGeometry* and

positioned in the right place of the body. Finally, we have the tail, last element of this model, positioned in the back of the figure.
The pig is created through *createPig()* function.
Function *Pig(color, transparent, setOpacity)* sets the color of the pig as pink (defined in *Colors*), transparency to false and opacity to the value of 1.

```
function createPig(){
  pig = new Pig(Colors.pink, false, 1.0);
  pig.mesh.scale.set(0.28,0.28,0.28);
  pig.mesh.position.y = gameVariables.pigInitialHeight;
  pig.mesh.position.z = 0;
  scene.add(pig.mesh);
}
```

In the function *updatePig()* there are updates of wings and tail motion and function *move()* is called. In *move()* function, we deal with the motion of the pig throughout the y-axis. In practice, a mouse listener (placed in *init()* function) gives the coordinates for dislocation on the y-axis. The pig has rotations on x and z axis too. Moreover, back legs and front legs rotate along with the body, using the difference between the mouse position and the pig position and exploiting the hierarchical model.

```
function move() {
  //Fly margins
  var pigX = fly(mouse.x, -1, 1, -100, 100);
  var pigY = fly(mouse.y, -1, 1, 100, 258);
  //Movements
  pig.mesh.position.y += (pigY - pig.mesh.position.y) * 0.1;
  pig.mesh.rotation.x = (pig.mesh.position.y - pigY) * 0.005;
  pig.mesh.rotation.z = (pigY - pig.mesh.position.y) * 0.01;
  pig.backLegDX.rotation.z = (pigY - pig.mesh.position.y) * -0.03;
  pig.backLegSX.rotation.z = (pigY - pig.mesh.position.y) * -0.03;
  pig.frontLegDX.rotation.z = (pigY - pig.mesh.position.y) * -0.03;
  pig.frontLegSX.rotation.z = (pigY - pig.mesh.position.y) * -0.03;
}
```

3.2.5 Airplane

Airplane is the second hierarchical model of the project.
It consists of cockpit, engine, tail, lower and upper wings, left and right supports, propeller and blade.
As in the case of the pig, a PhongMaterial shader is applied to all the components in order to render yellow and red airplanes on the background.
All the components of an airplane are made with *BoxGeometry* and related together in a hierarchical model to give a sense of motion to the entire scene.
A single airplane is created with the execution of the function *createAirplane()*.

```
function createAirplane() {
  airplane = new Airplane();
  airplane.mesh.scale.set(0.25, 0.25, 0.25);
  airplane.mesh.position.y = 100;
  scene.add(airplane.mesh);
}
```

Airplanes' creation is defined in the *Sky* object, in which is defined also the irregular displacement of the airplanes.

```
Sky = function(){
  this.mesh = new THREE.Object3D();
  this.numElem = 12;
  this.airplanes = [];
  var stepToAdd = Math.PI*2 / this.numElem;
  for(var i=0; i<this.numElem; i++){
    var c = new Airplane();
    var s = 0.10+Math.random()*0.15;
    this.airplanes.push(c);
    var a = stepToAdd*i;
    var h = 800 + Math.random()*300;
    c.mesh.position.z = -300-Math.random()*100;
    c.mesh.rotation.z = a + Math.PI/2;
    c.mesh.position.y = Math.sin(a)*h;
    c.mesh.position.x = Math.cos(a)*h;
    c.mesh.scale.set(s,s,s);
    this.mesh.add(c.mesh);
  }
}
```

3.2.6 Food

Food creation is made through *createFood()* function, that simply call the function *FoodOwner(num)*.
FoodOwner(num) function uses two arrays (*foodInScene[]* and *foodStock[]*). First of all, the function add to *foodStock[]* the prefixed number of food. Then, it add food to the scene with *foodInScene[]* array, respecting distance and frequence parameters. These operations are made in *FoodOwner.prototype.addFood()* function.

```
FoodOwner.prototype.addFood = function(){
  var num = Math.floor(Math.random()*5)+2;
  var amp = Math.round(Math.random()*10);
  var dist = gameVariables.grassRadius + gameVariables.pigInitialHeight + (-1 + Math.random() * 2) * (gameVariables.planeHeight-20);
  for (var i=0; i<num; i++){
    var food;
    if (this.foodStock.length) {
      food = this.foodStock.pop();
    }else{
      food = new Food();
    }
    this.mesh.add(food.mesh);
    this.foodInScene.push(food);
    food.distance = dist + Math.cos(i*.5)*amp;
    food.angle = -(i*0.02);
    food.mesh.position.y = -gameVariables.grassRadius + Math.sin(food.angle) * food.distance;
    food.mesh.position.x = Math.cos(food.angle) * food.distance;
  }
}
```

The food animation is managed in the function *FoodOwner.prototype.foodAnimation()*.
In the same function, I manage the moment in which the pig catches the food, with the corresponding increment in points.
To all the food is applied a texture, configured in the specific function *configureTextureFood()*, in order to give to the spheres the aspect of real food through Three.js *TextureLoader* function.

```
function configureTextureFood() {
  texture = new THREE.TextureLoader().load("Textures/food.jpg");
  texture.wrapS = THREE.ClampToEdgeWrapping;
  texture.wrapT = THREE.ClampToEdgeWrapping;
}
```

3.2.7 Danger

Dangers are created with the same mechanism as Food, i.e. calling the function *createDanger()*, that in turn creates an instance of DangerOwner.
DangerOwner has two implemented functions, i.e. *addDanger()* and *dangerAnimation()*, that manage the creation and the animation of dangers.
In function *dangerAnimation()*, there is also the managament of the situation in which pig bumps against a danger. In this case, a life is lost and the table in the scene is updated through the function *updateLife()*.

3.2.8 Lights

In this project, I implemented three lights: an ambient light, an emisphere light and a directional light.
The ambient light globally illuminates all objects in the scene equally.
The hemisphere light is used to shade from the sky color to the ground color, creating an effect of gradient color, typical of the skyline.
Finally, the directional light, that is implemented as a white light, is used to give shadows to the objects.
All the lights are set in the function *createLightsandShadows*, in which shadows'

parameters are set too.

```
function createLightsAndShadows() {
  ambientLight = new THREE.AmbientLight(0xdc8874, 0.5);
  hemisphereLight = new THREE.HemisphereLight(0xaaaaaa,0x000000, 1.0);
  directionalLight = new THREE.DirectionalLight(0xffffff, 0.9);
  directionalLight.position.set(100,200,200);
  directionalLight.castShadow = true;
  directionalLight.shadow.camera.left = -300;
  directionalLight.shadow.camera.right = 300;
  directionalLight.shadow.camera.top = 300;
  directionalLight.shadow.camera.bottom = -300;
  directionalLight.shadow.camera.near = gameVariables.near;
  directionalLight.shadow.camera.far = gameVariables.farLight;
  directionalLight.shadow.mapSize.width = 2048;
  directionalLight.shadow.mapSize.height = 2048;
  scene.add(hemisphereLight);
  scene.add(directionalLight);
}
```

3.2.9 Sounds

Sounds initialization is done in the function *ambientMusic* called within function *init()*. In *ambientMusic* function, the various settings for the music (such as tracks, loops and volume) are initialized.

```
// Sound variables
var sound, audioListener, audioLoader;
var volume = 0.3;
const tracks = ["Audio/threeLittlePigsRemix.mp3", "Audio/springSounds.mp3"];
var sounds = [];
var mute = false;

function ambientMusic() {
  audioLoader = new THREE.AudioLoader();
  for (let i = 0; i < tracks.length; i++) {
    sounds[i] = new THREE.Audio(audioListener);
    audioLoader.load(tracks[i], function(buffer) {
      sounds[i].setBuffer(buffer);
      sounds[i].setLoop(true);
      sounds[i].setVolume(volume);
      if (i == 0) {
        sound = sounds[0];
        sound.play();
      }
    });
  }
}
```

There are two tracks, one for the game and the other for the pause situation.
The user can interact with the sounds through a button, used to activate or deactivate *muteAudio()* function, and a slider, used to adjust music volume.
Mute button is effective only on the game music, because the "pause" music is really soft.

3.3 CSS

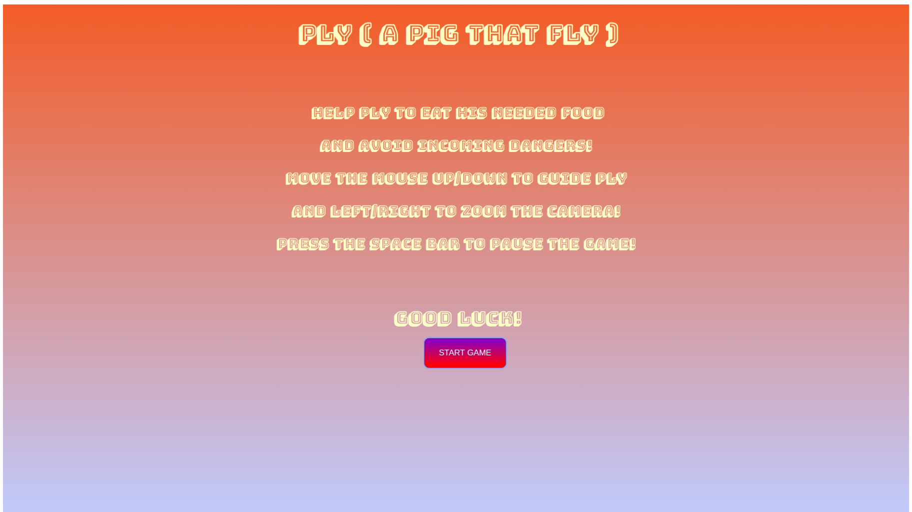
CSS file is included in the project in *index.html* file.
In CSS file, I tried to give a touch of beauty to the entire project, giving particular shades or fonts to quite everything in the scene.
In particular, I applied a characteristic font to all the text in the project. This font (*Bungee Shade*) is developed by Google Fonts and, in order to use that, I imported the above font in the CSS file in this way:

```
@import url('https://fonts.googleapis.com/css?family=Bungee+Shade');
```

Moreover, in this file there is the style creation of all the buttons, using *linear gradient* to give them a shade effect. Finally, table style is modified to show points, level and lives in a proper way.

4 Final result

4.1 Presentation



4.2 Scenario