SAPIENZA
UNIVERSITÀ DI ROMA

Interactive Graphics

2020 course

Christian Conti - 1316284

Luis Mautone - 1875387

Racing Time

Description

The project is a platform game in which the player commands a car in an outdoor environment to collect coins while avoiding obstacles. The idea was inspired by an Android game called *Hillside Drive* and our purpose was recreating the gameplay structure into a 3D environment.

Technologies

Libraries included are:

- Threejs, used to create and display animated 3D computer graphics in the web browser;
- Physijs, a plugin employed for physics simulations;
- Tweenjs, used to generate the inbetweening for the animations.

Development

Front-end web development was adopted in order to convert data to a graphical interface available to user interaction. The project mainly consists in:

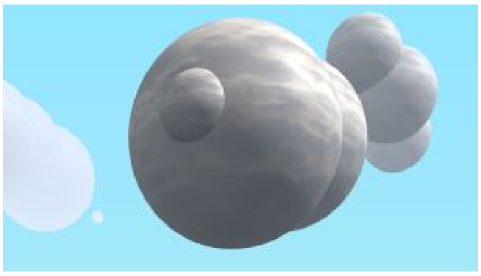
- *index.html*, an HTML file that corresponds to the main page of the application. It includes all the needed informations and files for the project and it handles their visualization and execution;
- *css* directory, it contains CSS files that establish the presentation properties for the various elements belonging to the game (background, width, height, color);
- *js* directory, it contains the JavaScript files that build the additional logic needed for the application to run (scene configuration, model classes, user controls);
- texture directory, it contains all the textures used in the project

1 Scene

The scene is a Physijs scene set up with the usual Threejs scene elements: a perspective camera; a renderer that draws its output to an HTML element (*world*) using all the window's content area. Since it is a Physijs scene, we can set up the gravity. An event listener is added to the window element to catch *resize* events and adjust the renderer output size and the camera. The background color is a linear gradient that gives the image of a sky with different shades of illumination. We added also a fog effect.

1.1 Scene models

The game environment includes a cylinder that rotates around its y-axis and will be the ground where the car will move. It is a Physijs Mesh with mass zero so that it is not affected by gravity and become a static object. It is created with a Physijs material that gives low restitution but high friction to elements getting in contact with it.



Then, sky is created. Sky mesh is only a placeholder where to put cloud meshes. Clouds are composed from three to six spheres of different dimensions aligned randomly along the x-axis, each one with a texture. Then clouds are distributed following a precise angle, given by the number of clouds created, in a circular fashion around the sky mesh using polar coordinates.

```
this.nClouds = 60;
var stepAngle = Math.PI*2 / this.nClouds;
```

Each time a new set of clouds is added, its meshes are rotated with a precise angle to follow the curvature of the cylinder:

```
for(var i=0; i<this.nClouds; i++){ (...)
var angle = stepAngle*i;
cloud.mesh.position.y = Math.sin(angle)*height;
cloud.mesh.position.x = Math.cos(angle)*height;
cloud.mesh.position.z = rangedRandom(-500, 200);
cloud.mesh.rotation.z = angle + Math.PI/2;
```

Since *clouds* rotate around *ground mesh*, *sky* radius is greater than the *ground mesh* one. Then we created the obstacles for the car:

- **trees:** a hierarchical model consisting in a cylinder mesh as trunk (root) and three cone meshes (crowns), each one child of the previous one. Trees are distributed over ground like clouds: a dummy Physijs mesh (*forest.mesh*) holds all other trees distributed around it. It must be a Physijs mesh because otherwise all its children would not have Physijs parameters. Trees and crowns have their own texture.

- **rocks:** as for trees, rocks use a dummy Physijs mesh that holds all rock meshes, distributed on ground like clouds and trees. Rock geometry is an icosahedron that shows only half of the mesh since the other half is inside *ground mesh*, to have the effect of a rock sticking out of the ground.

Coins are created to give the player a purpose. Coins are rotated (x-axis) cylinder Threejs meshes that rotate on themselves around their z-axis. They are not Physijs meshes because otherwise they would be static objects with no mass hence fixed to the ground like an obstacle. On the contrary, the car should go through them, detect the collision and add a point for each coin traversed. Collisions with coins are implemented in the *loop()* function.

Raycasting is exploited to detect collisions. For each vertex of the car body we generate a ray starting from the center of the mesh (car.body.position) with direction the vector obtained subtracting the center of the mesh from such vertex. Then we determine if any of these rays intersects any face of a coin mesh.

```
var originPoint = car.body.position.clone();
(...)
var localVertex = car.body.geometry.vertices[vertexIndex].clone();
var globalVertex = localVertex.applyMatrix4( car.body.matrix );
var directionVector = globalVertex.sub( car.body.position );
var ray = new THREE.Raycaster( originPoint,
directionVector.clone().normalize() );
(...)
var collisionResults = ray.intersectObject( coin.mesh.children[i] );
```

Coins are distributed randomly on the ground like trees etc.
Finally we added ramps. They are meant to have fun with the car and the physics. They are cubes (Physijs box meshes) with four vertices equals two by two:

```
function createRamp(x, y, z, posX, posY, posZ){
var geometry = new THREE.BoxGeometry( x, y, z, 2);
geometry.vertices[4] = geometry.vertices[6];
geometry.vertices[5] = geometry.vertices[7];
(...)
```

Also ramps have a texture and are placed randomly as all other environments meshes. When choosing randomly a position for these meshes we check if that position is already taken by other meshes by measuring the distance between the object position and another object already in a list of taken positions (at first there is only the car model).

```
var objPos = new THREE.Vector3(rock.position.x,   rock.position.y,
rock.position.z );
for(var pos of takenPosArray){
    while(objPos.distanceTo(pos) < 70 ||
        objPos.distanceTo(CarStartPos) < 70 ) {
        objPos.x = rock.position.x = Math.cos(angle+newAngle)*height;
        objPos.y = rock.position.y = Math.sin(angle+newAngle)*height;
        objPos.z = rock.position.z =  radnomRoadSideZ(-340, 340, 70);
        rock.rotation.z = angle+newAngle - Math.PI/2;
```

If the position is not available, then it is changed until a free one is found.
If the position is free then we add the object position to the array of occupied positions.

```
takenPosArray.push(objPos);
```

2 Car

We created an off-road hierarchical model car exploiting Threejs and Physijs libraries.
First we defined the necessary functions with which to build car parts.

```
function createCylinderPhysTex(radiusTop, radiusBottom , height,
radialSegments, posX, posY, posZ, texture, mass) {
      var geometry = new THREE.CylinderGeometry(radiusTop, radiusBottom
, height, radialSegments);
      var material = Physijs.createMaterial(
            new THREE.MeshPhongMaterial({ map: loader.load(
'textures/'+texture ) }),
            .8, // high friction
            .2 // low restitution
      );
(...)
```

2.1 Hierarchical model

The model consists of Physijs meshes and Threejs meshes. The former ones are made to detect collisions with a physics engine while the latter ones are made for decoration. Car parts are structured in a hierarchical way as follows:

- car body (root node, Physijs mesh)
    - roof (the cabin, Physijs mesh)
        - windshield
        - left and right windows
        - rear window
    - wheels (Physijs meshes)
        - each wheel has a wheel rim
    - spare wheel (Physijs mesh)
        - spare wheel rim
    - wheel axles
    - front headlights
        - each light has a spotlight
            - each spotlight point to a target Object3d in front of the car

    - radiator grille;
    - car doors
        - each car door has a handle

Then, we named them to better manage the model. We could not use Threejs Group() feature to build the hierarchical model because of Physijs which use the compound shape construct instead.

```
var body = createBoxPhys( BodyGeom.x, BodyGeom.y, BodyGeom.z,
            StartPos.x,StartPos.y,StartPos.z,CarMass,Colors.armyGreen);
this.body = body;
this.body.add(roof);
this.body.add(fl); (...)
```

This hierarchical structure allowed us to control the car by modifying only the parameters of the car body which is the root node. Each mesh of the car has a color or a texture assigned to it . For example the roof has a military texture, while the car body has a plain green color. Physijs meshes are the *wheels* (to make contact with the ground mesh) the *body* (to detect collisions with obstacles) and the roof (to avoid the situation in which the car overturns and the roof makes no contact with the ground that incorporates it). To handle collisions we add an event listener to the car body when adding the car to the scene.

2.2 Physics simulation

We chose to add physics simulations within the game in order to recreate the pace of the car on the ground, the bumps with the objects and the jumps performed from the ramps.
This has been achieved with the Threejs physics plugin, Physijs: the scene and the meshes have become Physijs elements; the gravity mechanism has been set for the y axis; in the loop function the physics simulation in the scene is enabled.

For the physics settings on the car mesh, the example contained in the Physijs documentation, "Costraint Car" **[1]**, was taken as an initial reference. The body of the car is a *Physijs.BoxMesh* type of mesh and has a mass of 5000; the wheels are *Physijs.CylinderMesh* mesh and have a mass of 1000. The ground is a *Physijs.ConvexMesh*, the most suitable since it corresponds to any set convex geometry and has a 0 mass, since it is an object that will always be static. In order for the ground rotation to change, it was necessary to set the object's __*dirtyRotation* flag to be true.
The trunks of the trees are *Physijs.CylinderMesh*, the rocks and the ramps are *Physijs.ConvexMesh*: since they will be static, in the same way they all have a 0 mass.

2.3 Controls and Animations

We implemented the interaction for the car driving by making use of the directional arrows:

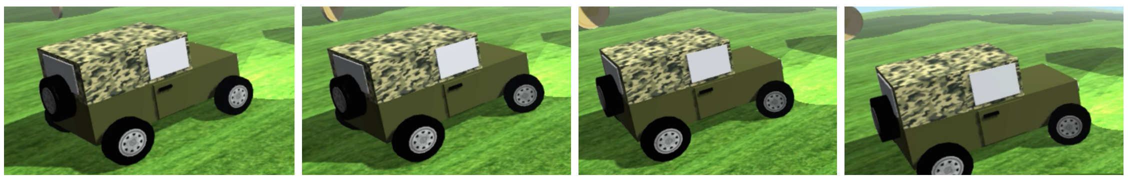
- Up arrow, once it's pressed the *rotationSpeed* increases up to a maximum value of 1.5; when the button is released the speed decreases until a constant value of 0.5.
- Right arrow, once it's pressed the car steers to the right; once released it returns to a forward direction
- Left arrow, similarly a steering to the left is performed
- Down arrow, the car stops and the rotationSpeed has a value of 0.

The possible actions mentioned above involve the execution of animations on the car model. When the car gains in speed, *rotationSpeed* increases. Its value is used to establish the rotation of the ground and the rotation of the car wheels inside the loop function.

```
ground.mesh.rotation.z += .001*rotationSpeed;
car.body.children[1].rotation.y -= .1*rotationSpeed;
```

Steering involves rotation of the wheels, for a range of -0.5/+0.5 around the y axis, and rotation of the body, for a range of -0.2/+0.2 around the same axis. Finally, a translation with values of -40/+40 is carried out on the z axis.

In order for the animations to be as fluid as possible, it was necessary to add the so-called "inbetweeners": the intermediate values for the speed, rotation and position properties that create the illusion of movement. This was possible thanks to the Tweenjs plugin which allows to manage the interpolation from a starting value to a target value in a given time interval.
The *rotationSpeed* is increased until the limit value of 1.5 is reached in 500ms, with a linear easing function. It's instead decreased until it reaches 0.5 in 100ms with an easing quadratic out function.
The steering is the result of the concatenation of three tweens: rotation of the wheels, rotation of the car body and translation of the car body.


a. forward direction *b. wheels to the right* *c. body to the right* *d. translation to the right*

Animation keyframes of a right steering

3 Lighting and Textures

3.1 Lighting

The two main lights used are *HemisphereLight* and *DirectionalLight*. The first one is a light source positioned above the scene, with color fading from the sky color to the ground color but it cannot cast shadows. For this purpose we use the directional light and its *directionalLightShadow*. This one allows to calculate shadows using an orthographic camera that is the light's view of the world and that is used to generate a depth map of the scene: objects behind other objects from the light's perspective will be in shadow. It uses an orthographic camera and not a perspective one because directional light rays are parallel.

Other light sources are two spotlights, placed on the car headlights, each one pointing to a 3D object that is its target.

```
var leftHeadLightLIGHT = new THREE.SpotLight(Colors.white, 2, 200,
Math.PI/4);
leftHeadLightLIGHT.target.position.set(-30,-BodyGeom.x,0);
```

3.2 Textures

Almost all game mesh materials comes with a texture.

```
var material = new THREE.MeshPhongMaterial({ map: loader.load(
'textures/'+texture ) });
material.map.wrapS = material.map.wrapT = THREE.RepeatWrapping;
material.map.repeat.set( 2, 1);
```

The texture on the car are the ones for the wheel rims, the roof and the radiator grill. Since texture are loaded as they enter the canvas, too many texture cause a framerate drop. Hence, for the sake of playability we removed (commented) bump map textures which gave the scene much more realism. All textures are resized to be square and to have dimensions of power of two to let the loading phase to be quicker without any image processing.

```
bumpMap: loader.load( 'textures/trunkBump.png')
(...)
material.bumpMap.wrapS = material.bumpMap.wrapT = THREE.RepeatWrapping;
material.bumpMap.repeat.set(4, 3);
```

4 Game

The goal of the game is to collect all the coins scattered on the path of the ground, before being hit by obstacles such as trees and rocks and completely lose the available health. The total coins are 50 and the user has 3 health starting points.

The car is controlled using the directional arrows: generally the user should accelerate to go faster and use the side arrows to decide the correct position within the driving to collect the coins, to use the ramps and to avoid obstacles. He can also stop the car by pressing the down arrow. For each collected coin a point is added to the total count; if the car climbs a ramp, the physics simulation allows an acrobatics to be performed in the air; if the car hits an obstacle, a crash simulation will be recreated and health will be decreased by one point. Moreover after a time interval of 3 seconds the car will be repositioned in a stable manner.

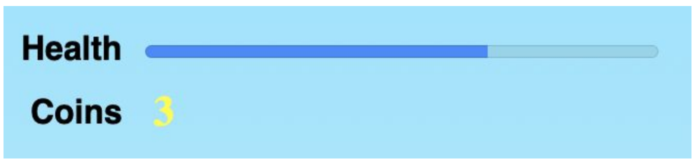
We decided to implement a repositioning procedure to simplify the behavior of the physics simulation, which would be too uncontrollable in certain cases and not have allowed to play in an enjoyable manner. In particular, the car model is removed and created again in the starting position (-100, 45, 0) if the car overturns, goes the wrong way or falls off the “ground rotating cylinder”.

Once health points end, the game is over and it will restart with the reset lives and number of coins.

5 User Interface

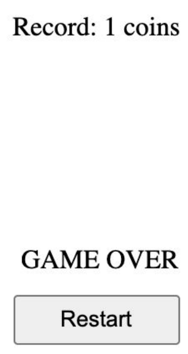
The user interface includes several elements such as buttons, labels and counters.

In the game screen, in the top right corner, there are two indicators: a progress bar showing the health percentage and a counter keeping up to date the number of coins collected until that moment in the game session.



In the bottom right corner, there is a reset button which allows the user, if he prefers, to reposition the car mesh in a stable manner in case of car rollovers or unwanted rotations.

In the game over screen, placed at the center, there is: the record of coins collected by the user, a game over label and a restart button to make start again the game.


Record: 1 coins

GAME OVER

If the user collects all 50 coins, a winning screen with a label and the restart button appears.

Before the scene is completely loaded, a screen with a loading animation is showed. The loading screen was taken from an online resource **[2]**.

Resources

[1] Physijs example reference,
http://chandlerprall.github.io/Physijs/examples/constraints_car.html

[2] Loading screen, https://jsfiddle.net/vfug1adn/19/