Final project presentation - Interactive graphics 2020/2021

Emanuele Patriarca matricola 1798796

This project is about developing a side-scrolling 3D videogame infrastructure where two players can battle against each other with the two controllable characters of the game: a dragon and a wizard. Every time a player hits the enemy's character using his character's attack he is given one point. The points are shown top right of the screen, red for the dragon and blue for the wizard.

Libraries and models

The project is mainly based on the **Three.js** library, used to create the scene with all its major components such as the camera, the lights, the meshes …
Two additional libraries have been used throughout the project:

1. **Tween.js** for the animations part, to create nice in-betweens starting from a bunch of keyframes.
2. **Cannon.js** as a physics engine, to add properties to the scene like gravity, collision detection, forces, …

All the models in the game have been realised by Quaternius, a 3D Artist whose works can be found at his web site quaternius.com.
In particular the models inside the projects come from his Ultimate Nature pack, RPG Characters and Animated Monster pack.

Technical aspects

There are two main files that compose the game:

1. An HTML file that defines a basic structure for the page that contains the game.
   The canvas where Three.js draws into and the player's points to be displayed are defined here.

2. A Javascript file that contains the script that realizes the game.

The first thing the script does (other than loading all the needed modules with the import keyword), is to set up a WebGL **renderer** using the canvas retrieved from the HTML page.
Here we set that we want shadows to be displayed and realistic lights effects.
A **camera** and a **scene** are created; the camera is a Perspective Camera, so it needs all the parameters to define a frustum to be created: near and far planes, a fov and the aspect ratio of the canvas.

Then we have the setup of the **lights**.
Two lights are added to the scene: an HemisphereLight (a light that resembles an AmbientLight but takes into account the color of the background and that of the ground when globally illuminating the scene equally) and a DirectionalLight (a light that gets emitted in a specific direction). While HemisphereLight doesn't emit any shadow, DirectionalLight does, so the script sets up a shadow camera and a shadow map for the light. In this way the renderer will render the shadows created by that light.

Once this is done, the **models** are loaded using a GLTFLoader. In fact all the models are .gltf files , a type of file used to represent 3D models in all their aspects: hierarchical properties, skeleton bones, textures, meshes, …
The loader returns a gltf object for every model loaded, from where the root of the model can be retrieved.
Once the loading process of all the models is completed, an **init function** is called.
This function serves many purposes, for every model:

1. Takes the root of the model from the gltf object returned by the loader and adds it to the scene.

2. Put the model in the right initial position inside the scene.

3. If the model has animations, it creates an instance of the GameObject class to represent the animated model. The class contains properties like the root of the model, the name, the array of animations, the initialization function and the update function. The initialization function is called after the object creation in order to initialize the model. The update function is called inside the rendering loop, to update the position of the model in case it is performing an animation or it is receiving an input to which responde.

4. Initialize the animations, if the model has some, then add them inside the animations array of the respective GameObject instance.

Every **animation** of the game is represented by a Group object of the tween.js library.
To these types of objects are added several Tween objects.
A Tween is an object that takes a starting keyframe and an ending keyframe and automatically creates the in-betweens given how much time passes between the start and the end and the easing function.
Each Tween can have a onUpdate function, that is called after each update of the tween; in the code this function is used to change the position of the model using the value of the current in-between created.
An animation is obtained by chaining several tweens together.
Calling the update function of a Group object makes the respective animation to play.
In the code, for each animation in the game, a function takes care of creating the tweens starting from the keyframes for each part of the model that moves in the animation.
The animated models in the game are the dragon and the wizard: the dragon can perform a fly animation while the wizard has a run and an idle animation.

A class InputManager has been implemented to take care of the state of the keys being pressed. A keycode can be added to the class in order to monitor the corresponding key. Every added key has two properties: down and justPressed.

Down is true when the key is pressed, false when it is released while justPressed is true when the key has just been pressed and false immediately after.
This class has an update function called inside the rendering loop that changes, if needed, the states of all the added keys using also event listeners. (The events are received by the browser after keys are pressed)

The **physics engine** initialization begins with the creation of a World object, that represents the environment where all the physics simulations happen.
Then a solver is created, which takes care of detecting the collisions between the bodies that are moving around in the scene, together with a special material added to the world that is used to simulate all the properties of a body material that affects physics such as friction and restitution coefficients.
In the world are contained :

1. Two boxes representing the bounding boxes of the dragon and the wizard, used to know when a model is hit by the other model. Whenever a box is moved around the world, the respective model will follow it so that the model is always inside the box. This is done inside the update function of each animated model copying the box position into the model position.

2. A plane representing the ground.

3. Two planes representing invisible walls, placed so that the models can only navigate along the z axis from left to right and vice versa.

Using the right keys (Shift for the dragon, and O for the wizard) players can make their characters shoot balls at each other.

This behaviour is implemented inside the update function of each character. When the key is pressed a ball is created (both the body that

needs to be added in the world and the mesh that goes into the scene) and it is given to it an initial velocity (3D vector).
A collide event listener is attached to the ball, so that when the ball collides with something, it is added to an array of bodies and meshes to be removed from respectively the world and the scene. At every iteration of the rendering loop, all the elements of this array are removed so that the balls are not rendered and simulated anymore.
If the collision happened with the body of the other character a point is added.

The players can use WAD (dragon) or IJL (wizard) to move their character left, right, or up with a jump.

This is also implemented in the update function of each character.
When moving left or right, first the character rotates if it is in the wrong direction (the direction of the character is saved in the respective GameObject instance), then the respective bounding box is translated so that the model follows it. When translating the wizard simultaneously plays the run animation , while the dragon always performs the fly animation. If the wizard isn't running it plays the idle animation.
The jump action is obtained by adding a vertical velocity to the bounding boxes just when the justPressed property of the jump key is true.

The rendering loop repeatedly :

1. Removes the balls' meshes and bodies that already made a collision.

2. Updates the simulation of the physics engine.

3. Updates the animated models calling their update functions.

4. Updates the position of the ball's meshes according to the position of the respective bodies (these are all the balls that don't have made a collision).

5. Updates the state of the inputManager.

6. Renders the scene.