Interactive Graphics - Final Project

Joseph Aguilar - 1721755
Francesco Starna - 1613660

Abstract

Three.js is a 3D javascript library based on WebGL, that allows the computation on the browser of 3D computer graphics applications. Together with pre-built models and Tween.js, for smooth animations, we present our final project for the Interactive Graphics course of the master's degree in Artificial Intelligence and Robotics of Sapienza University of Rome. The project is a simple and intuitive game called **Yoshi Corre Forever**, which is a "non-stop running" kind and its objective is to avoid obstacles and reach the highest score possible.

Environment

Three.js is a Javascript library for 3D web computer graphics based on WebGL. The latter in turn, is a low level 3D graphics library that allows OpenGL to be executed on the browser. Three.js in fact, is a high level implementation of OpenGL, hiding the pragmatic and time consuming coding of gl programs, vertex-shader, fragment shader, buffers, projections and rendering.

Including the library to the project is just importing into the `index.html` the file

```
<script src="js/three.js"></script>
```

The four three.js fundamental objects that manage and set up the webGL environment are:

1. `THREE.Scene`
2. `THREE.Object3D`
3. `THREE.Camera`
4. `THREE.WebGLRenderer`

The scene sets up the gl programs and data for buffers (vertex, color and texture coordinates), vertex shader and fragment shader, that are individually specified every time an object3D is added to it. A more specific child of the object3D is the `THREE.Mesh` that links a geometry with the material properties, giving the final object with all the needed buffers in order to be rendered with vertices, textures and colors.

The fundamental properties of an object3D, used the most for manipulating it are:

- modelViewMatrix: it is passed to the shader for computing the position of the object
- normalMatrix: it is passed to the shader and used to calculate lighting for the object
- position: a Vector3 representing the object's local position
- rotation: object's local rotation (Euler angles), in radians.
- scale: object's local scale

The camera is responsible for handling the projection matrix, used to define the view volume of the scene. For this project the perspective projection was chosen, because it is designed to mimic the way the human eye sees, manipulating the `fovy`, `aspect`, `near` and `far` parameters. Finally, the WebGL renderer, the most important component of the Three.js library, is the one responsible for linking the shaders with the gl programs, setting up the 3D graphics in the canvas element, and it is in fact called to render with the `requestAnimationFrame` function as follows: `renderer.render(scene, camera)`

Geometry

The Three.js library allows us to build geometries easily, based on the one we need. The geometries we built in the project are the one representing the world, while the characters are pre-designed models we downloaded and adapted to our needs.

World

```
var field = new THREE.Mesh(
      new THREE.CylinderGeometry(radius, radius, length, radial),
      new THREE.MeshBasicMaterial({map: worldTexture})
)

var cloud = new THREE.Mesh(
      new THREE.BoxGeometry(width, height, depth),
      new THREE.MeshPhongMaterial({color: white})
)
```

We first created a cylinder geometry with a grass texture[1], representing the field above which the player runs. The resulting mesh is then rotated 90 degrees about the x axis, in order to make it viable for the player.
We also created a base cloud mesh, that is a cube geometry which `x`, `y`, and `z` `position`, `orientation` and `scale` properties were changed randomly. An object3D that includes a variable number of these single meshes together forms a cloud. Finally the clouds were arranged around the rotated cylinder according to simple circle math

```
x = (r + h) * cos(angle)
y = (r + h) * sin(angle)
```

where `r` corresponds to the radius of the cylinder, h is an increase to the radius (in order to show the clouds above the field) and the `angle` is a growing parameter from `0` to 2π. The final world 3D object representation obtained is shown in the image[2].

Models

The characters' geometry was not created by ourselves. We preferred to import pre-built 3D models ready to load and use in the scene. The models' extension is GLTF (GL transmission format) and were loaded with the asynchronous function

```
var loader = new GLTFLoader()
loader.load('models/yoshi/yoshi.gltf', function (gltf) {
      gltf.scene // do something with the Object3D model
})
```

that loads the model and makes it as a 3D object ready to animate and add it to the scene. In the list[3] are listed all the models used in the project.

Skinned Mesh (Yoshi)

A specific mention goes to Yoshi's 3D model. It is in fact not only a simple mesh, with the already listed manipulation properties for position and orientation in the world, but it also has a skeleton with bones, capable of manipulating the meshes of the model by stretching the joints, to simulate real movements of the body. The Three.js object `THREE.Skeleton` is an array of `THREE.Bone()`. Each one is basically an object3D with position, orientation and transformation matrices properties. The difference is that the skinned mesh that contains the skeleton, has skin's weights and indices, used to transform the bones animating the vertices. The transformation matrix responsible for manipulating the bones transformations is the `bindMatrix` property. The Yoshi model was provided with a full skeleton with bones thanks to a 3D model application like Blender, making it possible to easily animating it without taking care of building the skeleton on our own. We first instantiated the model loading it, and then we set the body components retrieving the bones.

Light

For the project only ambient light and directional lights were used.

Directional Light

A directional light is a light that gets emitted in a specific direction. This light will behave as though it is infinitely far away and the rays produced from it are all parallel. The common use case for this is to simulate daylight and that was the use in the project presented. This light can cast shadows and the model used for the shadows was the Phong model.
A white (in RGB #FFFFFF) directional light with intensity of 1 was set with position 0, 1, 0. That means that the light will shine from top to down. The code used to set the light was the following:

```
var directionalLight = new THREE.DirectionalLight(0xffffff, 1)
directionalLight.position.set(0, 1, 0)
scene.add(directionalLight)
```

Ambient Light

This light globally illuminates all objects in the scene equally. This light cannot be used to cast shadows as it does not have a direction. This light is considerably darker since the goal was only to achieve brighter colors in the game. That's why the color #2F2F1F was used as follows:

```
var light = new THREE.AmbientLight(0x2f2f1f , 2)
scene.add(light)
```

The lights are placed in the first lines of the module script at the `index.html` file.

Animation

Part of the core work on the project concerns the animation engine we created in order to build the best playing experience possible. To do that, we decided to use a smooth animation library called Tween.js.

Tween.js

A tween is a concept that allows to change the values of the properties of an object in a smooth way. For example, suppose that we want to change the position of an object:

```
new TWEEN.Tween(yoshi.position)
      .to({x: yoshi.position.x+10, y: yoshi.position.y+10}, 1000)
      .start()
```

That means tweening the property position x and y increasing them of `10` units in `1` second. Tween.js library then takes care of the right interpolation, giving also the possibility to decide the preferred function. For a group of tweens that work and coexist in the same animation, Tween.js gives the possibility of instantiating first the tweens, collecting them in a `TWEEN.Group` and then starting the entire group. Other cool and useful methods provided are the `chain` method, used for combining different tweens and starting them one after the other, the `repeat` method, used for looping a tween, and the `onComplete` callback that we used very much for combining different interaction within the world between different objects.

World Motion

In a first approach to the game workflow, we tried to move Yoshi and the camera following him around the cylinder, but we ended in a 90 degree flip of the camera view, when the lower part of the cylinder was reached by the camera. Said so, we decided to change the approach and move only the world around the player (field, sky, characters and obstacles) and let the position and orientation of the player be the same, giving the illusion of a real running animation. In this regard, we created a global mesh, where we added all the object3Ds in the scene, and we simply accelerate the rotation about the z axis incrementally. The rotation was simply added to the game `update()` function in order to be rendered by the browser, with no need to use Tween.js.

Models Animation

All the models in the game (less than the still brick blocks) were animated using Tween.js. Each one presents different interaction in the world:

- Coin: rotating about the y axis infinitely
- Egg: rotating about the y axis and floating up and down slowly
- Mushroom: jumping smoothly using an easing quadratic function
- Goomba: Moving left and right on the z axis and oscillating about the y axis

Moreover, when the player hits one of them, a `catch()` callback is fired, performing different animations and playing different audios, depending on the character. The implementation of the models animation can be seen at the `js/models/object.js` file.

A more complex and intensive use of animations was made in order to build the ones for the player Yoshi. Specifically, we implemented the following functions:

- `run()`: it uses the highest number of tweens, involving different bones, like both the arms and the legs, the head, the spine, the knees, and the tail.
- `jump()`: it uses both the arms and legs, the mouth, and again uses a quadratic easing function to simulate the jump on the `y` position. Jumping is totally independent from the `run` function, and in fact can be used also while running.
- `pose()`: it brings Yoshi to its original position and orientation smoothly.
- `keyboard()`: it actually does not use Tween.js library, but it exploits the `onKeyDown` and `onKeyUp` web javascript listeners in order to start and stop moving Yoshi left and right.
- `big()`: it tweens the scale property increasing and decreasing incrementally the `x`, `y` and `z` values when Yoshi hits a mushroom, waits for a few seconds and then it brings the scale back to normal.
- `play()`: it basically lets Yoshi start running, moving the mouth and playing a sound.
- `cry()`: the animation starts when the game is over, stopping all the other tweens and involving the movements of arms, head, and mouth, also playing a sound.

The model building with all the just stated animation functions can be seen at the `js/models/yoshi.js` file. At the end, the rendering function `update(step)`, is the one responsible for updating the tweens and the `step` parameter determines the duration of the running animation, increased as the game goes on.

Game

The core work on the project involved also the game design. Before implementing the current game features we did some research on non-stop running game kind to find out what are the main characteristics and how they involve players. The `js/game.js` file implements all the game logic, the management of score values, rewards, enemies, and the rendering update function.

Logic

The first thing we have taken care of, thinking about the player experience, was to make the game less static and more dynamic and difficult the deeper the game goes on. We built a game setting class, that is basically a container of static variables with all the parameters that tune the objects dimensions, ratio and velocity, whose most important are: `gameSpeed, targetSpeed, rotationSpeed, yoshiSpeed`. The first, is the one controlling the increment of the yoshi speed and rotation speed of the global mesh, containing the world and the enemies. The target speed is the step by step increment every time a certain distance is runned, and every time the game level goes up. In this way it is easy to update the rotation speed while increasing the velocity of the running animation of Yoshi by simulating a correct run movement.

Spawn

Many different arrays as many types of rewards (coins, eggs, mushrooms) and enemies (goombas and blocks) were instantiated in order to keep track of the current state of array of enemies/rewards while playing. The main utility of these arrays is to spawn while playing, the correct number of characters, depending on how many are there in a certain moment, step by step. In fact, the functions `spawn<Reward>` and `spawn<Enemy>`, where reward and enemy represent the character name, take the number of characters to add, update the arrays and add them to the scene. When a minimum number of a certain characters is reached, the spawn function is fired, populating the world of new characters, and maintaining the playing experience steady.

Update

When talking about step by step, what it is meant is every time the `game.update()` function is called in the rendering process by the `requestAnimationFrame` web browser function. The following things are updated in that function:

- WebGL renderer
- Yoshi speed and tweens
- Energy, speed, level, distance, global rotation updates
- Spawn, collision and game status (init, playing, gameover) detection checks

Collision Detection

In order to check if the player Yoshi hits something in the world we developed an efficient collision detection system based on `THREE.Raycaster`. Raycasting is the use of ray-surface intersection in order to know if an object detects a collision. The Three.js object helps to identify the meshes that another mesh collides, indicating an origin point and a direction vector as input, and giving back the collided mesh if the collision was detected or null if not. The rays are spread starting from each vertex of the geometry, but since the models are object3D complex structures of articulated meshes, we decided to wrap each model inside a simple transparent `THREE.BoxGeometry`. Doing that, it was possible to reduce the computation of the raycaster on the vertices, because the number has decreased dramatically. The boxes can be seen in the reference image[4]. We keep and update an array of collidable meshes, in order to remove the mesh from the scene if a collision occurs. Depending on the model collided, we remove the mesh, we increase/decrease the energy, and we play a different sound. The mushroom is the only model that does not change the energy when hitted, and instead a callback is fired calling the `yoshi.big()` function.

User Manual

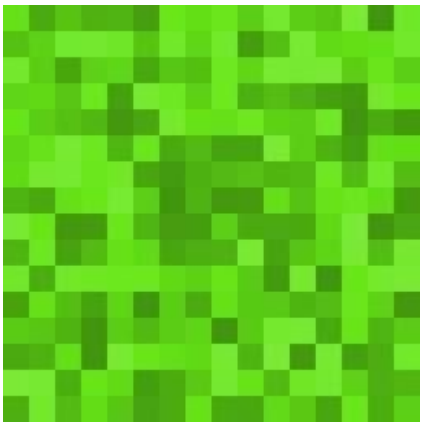
- ➔ When the web page is launched on the browser a loading screen indicates that the models are being loaded. When Yoshi appears on a gaming background, the application is ready to start the game.
- ➔ In that ready state, before starting the game, it is possible to interact with yoshi, tapping the **R** key for starting/stopping running and **Spacebar** for jumping. At any time the **Enter** starts the game.
- ➔ When the game starts, Yoshi is posing at the center of the world and the camera view starts to move towards it, at the end of which the Yoshi starts running and the game scores start counting. The distance score increases as Yoshi runs and every time it reaches 1000 meters, the level score goes up. The energy score decreases progressively as distance and levels increase.
- ➔ It is possible to move Yoshi horizontally by the use of the arrows keyboard **Left** and **Right**, and also to jump with the **Spacebar** key in order to avoid obstacles vertically.
- ➔ The objective of the game is to run the highest distance possible, avoiding obstacles, Goombas and brick blocks, and increasing the energy score by eating Eggs, and collecting Coins. The Mushrooms are special objects, because they let Yoshi grow up and become invincible for a few seconds.
- ➔ Have a good run!

Conclusion

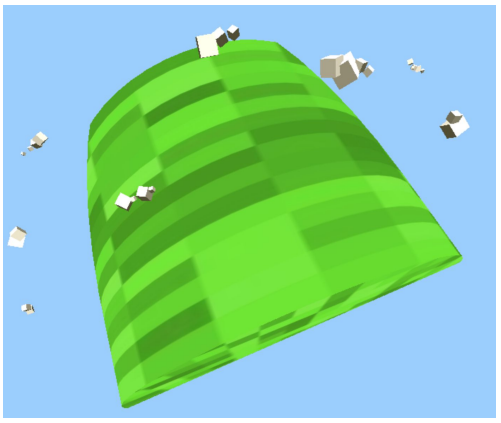
Three.js results to be a great starting point for 3D animation. It is not intended as a game engine, but with some effort, it was possible to build an entire game. The animations were surprisingly good looking even though they were entirely programmed from scratch only assisted with Tween.js to smooth them. Three.js was easy to learn, plenty of examples, with a good documentation and a large community with a lot of 3rd party plugins. The overall performance is really good, even though a loading screen was added to make the program wait until all the 3d models are loaded. We are finally satisfied with the results obtained and we are thankful for having had the chance to build a project together using the knowledge we have learned during the Interactive Graphics course, thanks to the professor Marco Schaerf, and to apply them with a modern 3D library as Three.js is.

References

Grass texture for the field



World 3D object



List of 3D models imported

- Yoshi (the player character)
- Goomba (the brown moving mushroom enemies)
- Block (the still brick blocks obstacles)
- Coin (small value rewards)
- Egg (high value reward)
- Mushroom (let yoshi grow and become invincible for a few seconds)

BoxGeometry of the models