Interactive Graphics

Final Project

Nicola Iommazzo 1693395
Alessandro Basciani 1675251
Matteo Silvestri 1774987

Indice

Capitolo 1

Introduction

This document will briefly explain the realization of the final project regarding the Interactive Graphics course. Our team decided to create a game in which the user controls a vehicle and has to totalize the maximum score by running over pedestrians in a city in the minimum amount of time.

1.1 Libraries

For this project we choose to use two different kind of libraries: the first one in order to handle the graphic part, and other one in order to handle collisions between objects. When we add an object inside our project (like a car or a palace) at first we will add it in our **Three.js Scene** in order to handle the position of the mesh, and after we will add it in our **Cannon.js World** in order to handle the collisions. Below we will generally explain how we used them, and their role in our project.

1.1.1 Three.js

The **Three.js** library [1] allows the creation of accelerated 3D animations using the JavaScript language as part a of a website, without relying on proprietary browser plugins. This is possible due to the advent of WebGL. High-level libraries as Three.js make it possible to author complex 3D computer animations that display in the browser without the effort required for a traditional standalone application or a plugin. In our project we use this library to handle all the object inside the scene, their position, the lights and the shadows of the whole game.

1.1.2 Cannon.js

The **Cannon.js** library [2] is an open source library used for handle the physics engine inside our project. For physics engine we mean for example the engine that handle collisions between two or more objects. This library provides a lot of methods and classes to handle these kind of situations. All the project parts that use the Cannon.js library will be shortly explained in the related sections.

Capitolo 2

Presentation Layer

The presentation layer of the project is composed of two *html* files. The first file that appears is **index.html**. In this file we explained how to play the game. The second file is **game.html**, where are explained the controls of the game and the game itself.

In the index file are initialized a render, a scene and a point of light, all implemented throught the **Three.js** library. The objective of this page is to let the user know the scope of the game. The texts are displayed on the canvas thanks to multiple **TextGeometry**. The style of the text is implemented with a *json* file in the directory *font*. In this directory are available a lot of text fonts: we decided to use the *serif bold*. We defined the function **addText()** for adding the text to the canvas. It requires 3 parameters: the string we want to print on the canvas, the font of the text and the vertical offset at which the text will placed. The default position is centered in the middle of the page. To do this we have initialized a 3D-vector and set its components to the maximum value of the TextGeoemtry bounding box, less the minimum one, all divided two. In this way we have placed the textGeometry in the center of the page. Then we created the material (together with the aspect and the color) for the TextGemoetry through the function **MeshPhongMaterial**, and finally we obtained the Mesh combining geometry and material.
All these features are implemented in the script tag, while in the html tag we implemented the href with the link of the game appearing with the play icon.

The second *html* file is the **game.html**, in this file we import all the library that use in the project. In the html tag we inserted the text explaining the controls of the game and there is also a small portion of *css* code, where we choose the color and the style of the page. Moreover in this file we load the external script **game.js** that will run the game itself. To start the game we click anywhere on the page and subsequently the countdown starts.

Capitolo 3

Game components

The game world is composed by many objects, of different types and different geometries. We specify that each object of the *threejs scene* is created starting from a **Geometry** with a particular shape and a **Material**: combining these two elements we can form the **Mesh** and add it to the **Scene**.
The geometries can be placed as pleasure in the euclidean space with simple transformation like traslations, rotations and scaling. The choice of the material is done between *BaseLambertMesh* and *BasePhongMaterial*: since the we want a better projection of the shadows on the other element, we adopted the *BasePhongMaterial* option for the most cases.

Moreover in order to make the elements collide each other we have also created *cannonjs bodies*. The procedure is quite similar to the first one: at first is created the **Material** and the **Shape**, then we create the object **Body** starting from the previous ones, at the end the body object is added to the **World**. The elements of the whole game can be grouped as the following:

- Vehicles
- Ground plane
- World limits
- City sidewalks
- City lamps
- City palaces

3.1 Vehicle

The vehicle is implemented both in the *threejs scene* and in the *cannonjs world*. From the point of view of Three.js it is a scene loaded through the **GLTFLoader** from an external source [3], at first scaled to adapt its dimensions to our project and then added to the scene. Much more different is the vehicle body implementation in the cannon.js world. In fact it is a **RaycastVehicle** object with a base chassis, a top chassis and four wheels. All the single bodies are kept together thanks to the constraint objects available from the cannon.js library. Since all the bodies that belongs to the vehicle has a positive mass they are subjected to the gravity force and will fall toward the ground.

3.2 City Buildings

The **ground mesh** is realized through a *PlaneGeometry* object oriented towards the positive y axis. There is also a ground in the *cannonjs world* so that each body having a positive mass will fall towards a solid ground. For this reason the ground body has a mass equals to 0.

As for the ground plane, the **world limits** are *PlaneGeometry* objects the but at first they are moved to the world edge and then rotated towards the center of the euclidean space. Since we want a single mesh representing the four world limits meshes, we merge them into a single geometry through the *merge()* method, and finally we create a mesh from it.
Ground limits are also implemented in the *cannonjs world* in order to block the vehicle and do not let go the vehicle beyond the city edges. All these bodies are object separated and not unified as done for the meshes.

The **palaces** and the **sidewalks** meshes are created starting from a particular *base building* structure (see *js/CityBuildings.js*). All these elements are *BoxGeometry*, but for performance reasons all these boxes have the bottom face missing (the bottom face is never shown in the game because it overlaps with the ground). Moreover the texture of the *faceVertexUvs* components relative to top face of the palaces are set to zero. This means that the palaces textures are not applied to the top face. As explained for the ground limits also these meshes are merged together, and finally created the final meshes of palaces and sidewalks.
Palaces and sidewalks have also bodies in the cannon.js world. They are *Box* bodies with a mass equals to zero so that they are not subjected to the world forces of the cannon world.

The last elements of the city are the **lamps**. In the three.js scene they are composed by the head, the pole and the base. All of them are *BoxGeometry* from which we created meshes. There are 4 lamps for each sidewalk block and the procedure is the same for all of them.

Capitolo 4

Hierachical Model

The hierachical model allows us to create, in our specific case, the **NiceDude objects**. Generally there are two methods to create a hierachical model: in the first one we use a graph model, in the second one we use a tree model. For our project we implemented the NiceDude hierarchical model through the tree representation. In this model each component of the model is expressed by a node. Each node has exactly one parent node and can have one or more children nodes. The parent of all nodes is called root node and has no parent node, indeed the node with no children is called leaf.

4.1 The NiceDude hierarchical model

The NiceDude class is implemented in the *NiceDude.js* file. Since we decided to implement our hierarchical model as a tree model we create a group object through a library function of Three.js. Subsequently we create the head, body, neck, left and right shoulders, left and right arms, left and right legs, left and right shoes. The head is the first object of the group and will be our root node. The following picture will explain the resulting structure of the hierarchical model we built. All the components have a function that build that specific

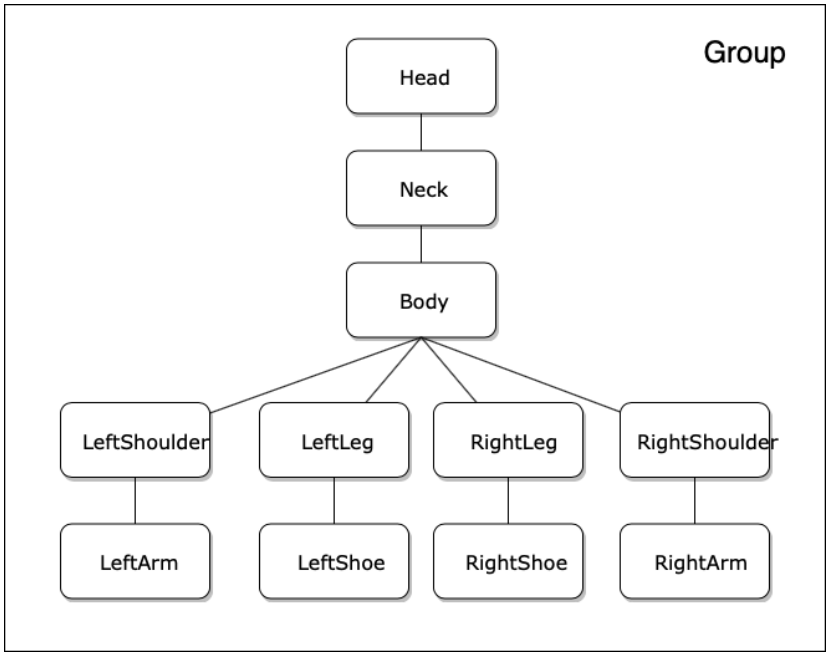

Figura 4.1: NiceDude hierarchical model implementation

part of the NiceDude by setting geometries, materials, meshes and placing the objects in a particular point of the space. All the material parts are **MeshBasicMaterial** taken from the Three.js library.

Capitolo 5

Textures

The textures in this project are used to customize the sidewalks, the palaces, the background, the ground, the lamps lights and the city limits. All the texture are taken from sites [4] that offers texture images for free. To load the texture image in the game we have used the loader **THREE.TextureLoader()** available in the *Three.js* library. With the *load()* method of the loader, we selected the precise texture we would import into the project. To finally add the texture to a material we specify the texture object as the *map* component of the material. All the textures have been implement following the just described process. The texture related to the city building can be found in the *js/CityBuildings.js* file, instead the background texture is in the *main.js* file.

5.1 City buildings textures

The city is composed by lots of component, but the elements customized with a texture are: palaces, ground, sidewalks, lamps lights and the city limits. We loaded a texture image for all components. Each texture is load in the same way, but what changes are the setting of the textures. Here we briefly analyze all of them.

- The **palaces** are represented by a plot that depicts windows. For these building is not necessary any customization of the texture, it is simply loaded and applied to the mesh.
- The **ground** is represented by a plot. The texture is *repeated* to infinity on the vertical and horizontal axis. This has been possible thanks to the two variables for texture objects (*wrapS* and *wrapT*) in the **Three.js** library. We can exploit them together with the *repeat* variable to say that we want the texture will repeat itself until covering the whole ground mesh.
- The **sidewalks** texture is simply loaded and applied to the mesh with no customization needed.
- The **lamps lights** are represented by a texture too. The material is build through to the *PointsMaterial* function. For this texture we specify that the size variable should be 8 and the variable transparent should be true. This will let the lamps lights texture to be transparent so that we can see what there is behind it and will not cover other objects.
- The **city limits** are composed by four plane geometry. So we load the same texture to each of that planes. For these textures we want that they should not reflect the lights of the scene so we set the *shininess* variable to zero. Morover, as for the ground texture,

we want that the texture will repeat itself for all the plane dimension, so we set properly the *wrapS, wrapT* and *repeat* variables.

5.2 Background scene

The **background** is customized by applying a texture to the *background* variable of the three.js scene. Differently from the 2D textures, now we have to apply a texture to the whole 3D scene. For this reason we use the *CubeTextureLoader()* loader obtaining a cube texture. In this case we do not need anymore just one image, but we need three or six images (depending on the implementation) that combined together will compose the 3D background.
To select the source images we used the *.setPath()* and *load()* methods of the loader object. For this project we chose a background representing the Mars [**?**] ground.

Capitolo 6

Lights And Shadows

In this part of the documentation we talk about the handling of the **lights and shadows** in our project, below we will explain our general approach for all the objects inside the scene.

6.1 General Approach

First of all we choose which kind of light to use: *DirectionaLight* or *Spotlight*. The first one is a light that gets emitted in a specific direction. This light will behave as it is infinitely far away and the rays produced from it are all parallel each others. The common use case for this is to simulate daylight. The other one light gets emitted from a single point in one direction, along a cone that increases in size the further from the light it gets.

For our project we chose the **Spotlight**, because with this we can create in the simpliest way the objects shadows in our Three.js scene. After the initialization of the Spotlight we set two different parameter. One to set the position of the cone light and another one to set the cast shadows flag. This flag is use to cast or not the shadows.

The next step is to set the shadows inside our world, for do this we needed less time than expected. We set to true the **castShadow** and the **receiveShadow** flags of the objects of the project depending on their role in the game. By default both of the flags are false. We had to set to true the *castShadow* flag for all those objects that are hit by the light, in this way we let those objects to create the shadows. The *receiveShadow* flag is responsible of the rendering of the shadows on its mesh. For this reason it is enabled only for those objects that receive shadows, like sidewalks and buildings.

Capitolo 7

User Interaction

The project is capable to interact with the user in multiple ways.
Starting from the *index.html* the user has to press the play button in order to move to the game page. In the first page is described the objective of the game thanks to a *TextGeometry* painted on the canvas (see the chapter on the Presentation Layer). Here is implemented an handler for the event *document.mousemove* [6] so that when the user moves the mouse cursor on the screen, the values of the *camera.position* related to the *threejs scene* change. We focus on the fact that this movement cannot be defined as a real animation of the text object since the mesh is not really moving in the space, but it is only the point of view of the camera that changes its position.
By clicking on the play button the user moves to the game page where are described the commands to play the game. When the user clicks on the screen the instructions disappear, the game starts and the *render()* function starts to update all the elements of the scene on the canvas.

7.1 Vehicle Controller

All the commands that refers to the vehicle movement are implemented in the file *js/Controls.js*. In this script there is a function handler for the events *document.onkeydown* and *document.onkeyup* [6]. So when the user presses or releases a button on the keyboard it is created an event that is intercepted by the function handler. Each button key is associated to a different keycode, in this way we can determine which button is pressed and differentiate the behaviour of the game. The handler modify the *keyDown* array of 320 boolean values, each one referring to a different key on the keyboard, setting to *true* the *i-th* cell if the button with keycode i is pressed, otherwise it is set to *false*.

At each cycle of the *render()* function it is invoked the *CarController()* function that checks the keyDown array and performs control actions on the vehicle.
When the *ESC* button is pressed the game is paused and all the timers, together with the *render()* function, are stopped and can be resumed by pressing the left click button of the mouse. With the *W, A, S, D* key buttons the user controls the direction and the speed of the vehicle. Moreover has been implemented the control to brake the vehicle (*SPACE* button) and to reverse the vehicle if it become impossible to move the car (*Q, E* buttons).

Capitolo 8

Animations

This chapter describes the animations realized in the project.

As explained in the previous chapter the text movement in the index page is not a real animation, so we skip directly to the game page. The animated objects in the game are the pedestrians (NiceDude meshes) and the vehicle. Regarding to the vehicle animations they are handled by the *cannon.js* library. By calling methods on the *cannon.js vehicle* we it moves in the *cannon.js world*, so the only thing we we did is to do overlap the *three.js vehicle model* with the moving *cannon.js vehicle*. With this process the car appears to move but we precise that the vehicle animations are not directly handled by us.

What is fully created by our team are the NiceDude movements implemented in the *animate()* method of the NiceDude object (see *js/NiceDude.js*).

8.1 NiceDude movements

The movement of a single NiceDude body can be distinguished in:

- the movement it does around the building
- the movements of its parts (arms, legs and feet)

Starting from the movement around the building it is a quite simple translation along the axis of the NiceDude direction. The things get more complicated when we have to avoid the lamps and in this way rotate around the building. So what we did is to check at each call of the *animate()* method if it is near to one of the four lamps of its square: if it is near the lamp we rotate the NiceDude object of 45°. By continuing to translate the object it will reach a position where is not anymore near to the lamp: when this happens we rotate again the object of 45°. In this way we have totally rotated the NiceDude object of 90 °and moreover we avoided the collision with the lamp.

Dealing with the body parties animations they are realized by rotating the parties of a little constant so that by calling the *animate()* method at each cycle of *render()* they appear to move in a smooth way. When the specific part reaches the maximum angle value we invert the constant rotation value so that the parts rotate in the opposite direction. We keep to notice that the constant values to rotate the body parts are defined as instance variables of the NiceDude object: in this way each NiceDude object has its own variables and by changing them we do not modify the other values. In few words we solved a concurrency problem by spatially separating the values on which the critical section works.

Capitolo 9

Conclusion

In conclusion we can say that with this project, our group, has managed to achieve the goals previously set, we have managed to create a city with buildings, sidewalks and lamps. Our goal from the beginning has always been to create an interactive game in order to respect the required features, and most important to entertain the user. In this game the primary objective is pass through the largest number of pedestrians on the sidewalk before the time limit expire by driving a car. The user is responsible of driving the car with the classic commands of the most famous videogames. The presentation layer is simply and intuitive: when our car strikes one of the pedestrians it disappears, and at the same time our total score increses.

Obviously like all the kind of projects, we found a lot of troubles with the handling two different kind of library(Three.js and Cannon.js) together, however we have fixed the most of the problems.

9.1 How to test the project

To test the project is very easy. It can be done through the github page or by downloading the entire folder of the project and run it offline. By following the second choice you should simulate a server on your machine, otherwise the most of the resources the game uses will not be found. We suggest to download the *live-server* [7] application and run it by command line from the downloaded project folder.
(on command line.) #projectfolder$ live-server

9.2 Bugs

In this section we discuss about the bugs of the project that are not still resolved. The most problematic bug is that of the collisions: sometimes the collisions between vehicle and city objects are not logged in the game. Since we implemented the collisions through the Cannon.js library, we rely on the *collision* event that the physics engine automatically sends when two body collides. The bug consists of the missing creation of the collision event, and conseguently we can not receive the event collision. We specify that we do not have to send the event message, but it is the cannon.js library that has to do it. The only thing we can do is to receive the event message and implement an handler to do all the necessary operations for our project (remove the niceDude hitted and increment the total score).

Bibliografia

[1] THREE.JS Docs, https://threejs.org/docs

[2] CANNON.JS Docs, https://schteppe.github.io/cannon.js/docs

[3] Sketchfab.com, https://sketchfab.com,

[4] TextureLibs.com, http://texturelib.com,

[5] Emil Persson, aka Humus, http://www.humus.name

[6] Javascript events, https://www.w3schools.com/js/js_events.asp

[7] Application live-server, https://www.npmjs.com/package/live-server