Interactive Graphics

Final Project

Nicola Iommazzo 1693395
Alessandro Basciani 1675251
Matteo Silvestri 1774987

Indice

1 Introduction **2**

1.1 Libraries 2

1.1.1 Three.js 2

1.1.2 Cannon.js 2

2 Presentation Layer **3**

3 Game components **4**

3.1 Vehicle 4

3.2 City Buildings 5

4 Hierachical Model **6**

4.1 The NiceDude hierarchical model 6

5 Textures **7**

5.1 City building textures 7

5.2 Background scene 8

6 Lights And Shadows **9**

6.1 General Approach 9

7 User Interaction **10**

7.1 Vehicle Controller 10

8 Animations **11**

8.1 NiceDude movements 11

9 Conclusion **12**

9.1 How to test the project 12

9.2 Bugs 12

Riferimenti bibliografici **12**

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 collect 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 building) we will add that at first in our Three.js Scene in order to handle the position of the meshe and after 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 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 such 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 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 collision between two or more objects. This library provides a lot of methods and classes to handle of 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 file html. The first file that appears is**index.html**, in this file we explained how to play at the game. The second file is **game.html**, in this file there is a countdown and are explained the controls of the game.

In the game 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 scope of the game. The instructions are displayed on the canvas thanks to multiple **TextGeometry**. The style of the text is implemented with a file **json** in the directory font. In this directory are available lots of text fonts: we decided to use the **serif bold**. The function for adding the text to the canvas is *addText()* passing as parameters the text you want to write, the font of the text and the position of the text. The position is centered in the middle of the page. To do this we 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 set 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 processes are written in a javascript section, while in the html section we implemented the href with the link of the game appearing with the play icon. The second file in html is the **game.html**, in this file we import all the library that use in the project, in the part of html we explain the control for the game. To start the game we click anywhere on the page and subsequently the countdown starts. In this file there is a part of css where we choose the color and the style of texts.

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 modified as pleasure in the euclidean space with simpe transformation like traslations, rotations and scaling. The choice of the material is done between *BaseLambertMesh* and *BasePhongMesh*: since the we want a better projection of the shadows on the other element, we adopted for the most cases the *BasePhongMaterial* option.

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 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 whole elements of the game can be grouped as the following:

- Vehicle
- Ground plane
- World limits
- City sidewalk
- 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, 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 constraints 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 towards the ground.

3.2 City Buildings

The ground mesh is realized through a *PlaneGeometry* 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* 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 three.js 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 method 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 and can have one or more children node. 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 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 shoulder, left and right arm, left and right leg, left and right shoes. The head is the first object of the group and will be our root node. All the components have a function that build that specific part of

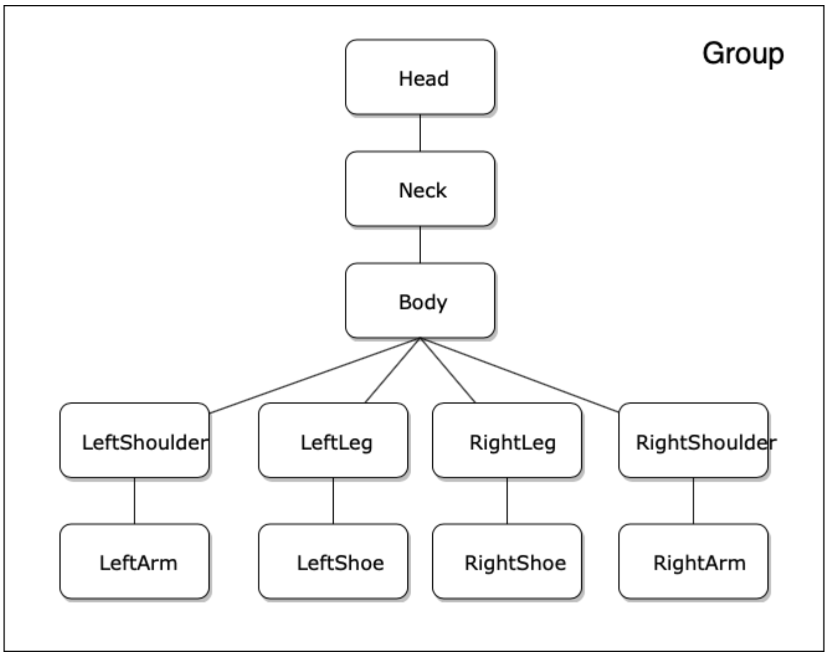

Figura 4.1: NiceDude hierarchical model implementation

the NiceDude by setting geometry, material, mesh and placing the model in a specific point of the scene. All the parts have the material **MeshBasicMaterial** taken from the Three.js library.

Capitolo 5

Textures

The texture in this project are used on the side walk,palaces,background,street,street light and the limit of city. All texture are an upload a photo, for upload the photo it was used the method of **Three.js**. The method is **THREE.TextureLoader().load("path photo")**, with this we can load for every component a photo. Moreover we can use other method for setting the photo. All the component of the city have been implement in the file CityBuildings.js, instead the background is implemented in the file main.js.

5.1 City building textures

The city is composed than more component, but the elements that use a texture are palace, street, side walk,street light and the limit of the city, we load a photo for all component. Every texture are load in the same method for the all element of the city and every components are represent than geometry cube.

- The palaces are represented by a plot that depicts the windows. In this case we use a method for upload a photo and do set the material variable. The material variable initialize the **MeshLambertMaterial**, is a method of Three.js, with it we can set the map of palaces with the texture.
- The street light are represented by a texture. The appearance of material is set than the method **PointsMaterial** with the map variable set at texture and the size set at 8 and the variable transparent set a true. Finally we merge the material with the cube represented the light.
- The street are represented by a plot. The texture is repeat to infinity on the vertical and horizontal axis. This is possible using two method of the **Three.js** library. We can use **wrapS** and **wrapT**. This two method are set to "repeat wrapping". These methods have been used because the photo was depicted too close.
- The side walk are represented by a photo. The texture is upload and in this case we use the method **MeshPhongMaterial** and set the variable map with the texture.
- The limit of the city is composed that four plane geometry,for create this plane we use a method of library **Three.js**. Every plane has a loaded texture in the same method as the others. In this case for the aspect that material we use the method **MeshPhong-Material**, with this method we set the texture, shininess at zero. In this case as the case of the street the texture is repeat to infinity on the vertical and horizontal axis with the same method.

5.2 Background scene

The texture it represent the background is upload through a method of the library **Three.js.** This method is different from cases of the city. In this case we use **CubeTextureLoader()**, with this method we create a cube texture. The different from upload a normal texture and a upload a cube texture is that in the cube texture the images are an array of 6 images as opposed to a single image. For select the foto use the method **.setPath("path photo")**, with this we set the path of the photo, and finally use the method **.load(urls)** for insert the type of the photo. The photo of background represent Mars.

Capitolo 6

Lights And Shadows

In this part of the paper we talk about the handling of the lights and shadows inside 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 use Directionalight or Spotlight, the first 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, 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. In our project we choose the second type of light, because with this we can create in a simplest way the object's shadows inside 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 need less time than expected, we must change the **castShadow** and the **receiveShadow** flag inside all the created objects(the flag is always initialized to false), we must change the castShadow flag of all the objects hit by the light, because in this way we permit to the objects to create the shadows in order to respect the position of the Spotlight, another important change is that of the receiveShadow, but only for 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* 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*. 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 entertain the users. In this game the primary objective is pass through the largest number of pedestrians on the sidewalk before the time limit expire with an Object 3D car, the user can move the car with the classic commands of the most famous video games(explained in the previous chapter), and when the car strike one of the pedestrians they disappear, and in the same time the final score increase. Obviously like all kind of project we found a lot of trouble with the handling two different kind of library(Three.js and Cannon.js) together, however we have fix the most of the problems, but like every project there are the bugs, in the next section at first we describe how to test the project and after we will talk about the bugs found.

9.1 How to test the project

Test the project is very easy, the most recommended way to start the project is to simulate a server(you can use live-server for simulate a local server) and open trough that the "index.html" file and after you can follow the instructions for start to play.

9.2 Bugs

In this section we discuss about the bugs of the project, the most problematic bug is the Cannon.js's bug, we can see this problem when we hit a pedestrian with the car and one of them doesn't disappear but the car goes through them, this problem come from the merge of Cannon.js and Three.js, and unfortunately this problem does not depend on us, but rather on the libraries used, which together have compatibility problems.

Bibliografia