SAPIENZA
UNIVERSITÀ DI ROMA

Crossy Streets

Interactive Graphics Final Project

Submitted by:
Ascanio Santomarco 1962686
Renzo Cherubino 1764113
Matteo Fischetto 1795458

Table of Contents

1. Introduction

1.1 Idea

Crossy Streets is a game inspired on the famous game *Crossy Road* for mobile phone. In this game, the selected character has to cross the streets full of cars to avoid to get at the end of the road.

We can summarize the game through the following features:

- A city scenario that alternates streets and sidewalk.
- possibility to play the stages on two different time-days: day and night;
- possibility to play with or without fog;
- three different characters that the player can choose from;
- three levels of difficulty, from easy to hard;
- possibility move the player in every direction;
- a viewable area that moves with time, and has to be taken into account to force the player to keep moving forward.

1.2 Libraries

To realize this idea, we decided to use the following libraries:

- ThreeJS: library used to create and display 3D computer graphics on all Web Browsers supported by WebGL: it creates a scene in an HTML canvas object where different graphical objects can be added.
- OrbitControls: module used to fix the camera on a specific target.
- SoundJS: ince the game requires some sound to be played during its experience, we have used a library that abstract the HTML5 AudioImplementation.

1.3 Structure

The files of our project implementation are defined as:

- Menuindex.html, credits.html, tutorial.html, selection.html, menu.css: these are the html files that handle the menu aspect.
- Model, texture folders: here there are the models and textures of our characters and objects.
- Sound, img, icon folders: there are the menu details implementations.
- thingsRoad.js, crossyStreets.js, three.js, OrbitControls.js: are the js files used for the game realization.

2. Models

In this chapter, we will describe in detail what type of methods have been used to add different models to the scene.

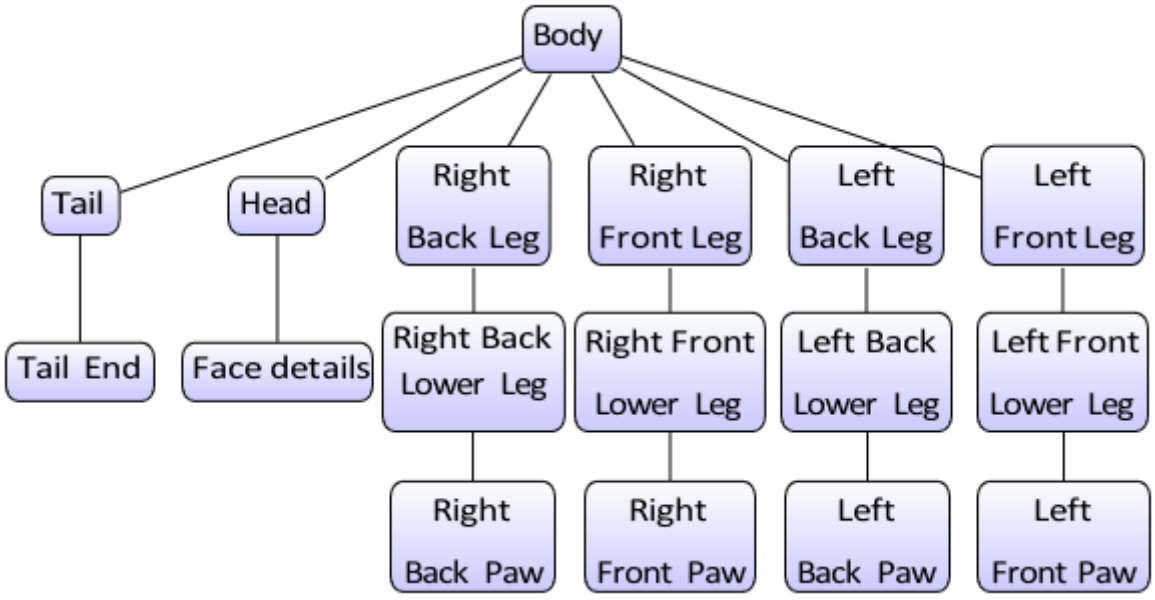
2.1 Simple Models

Among the simple models we have all the models that compose the scenario, such as the wall at the side and the bushes, which are initialized as box geometries with a texture on them as material.

2.2 Hierarchical Models

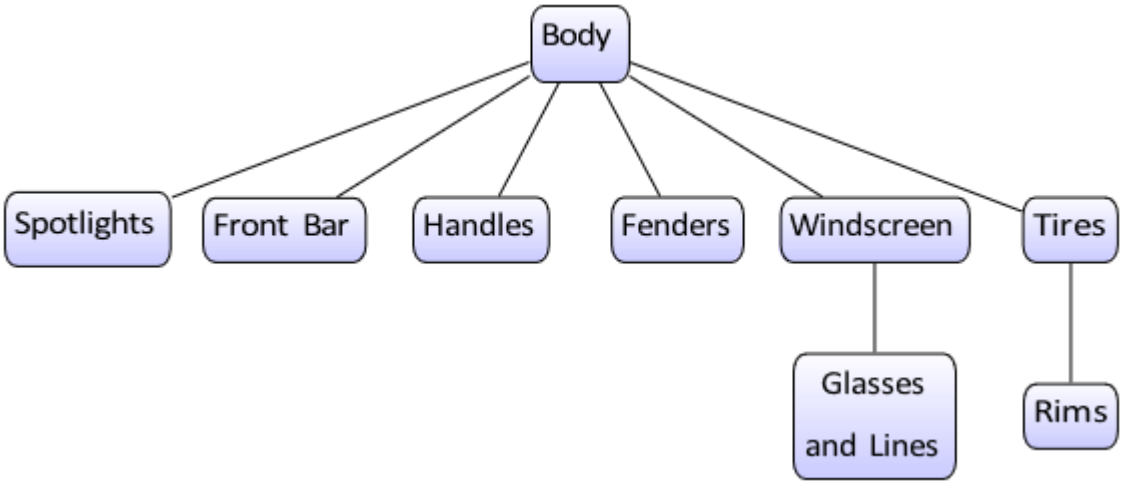
Hierarchical models give the opportunity to manipulate more objects at the time following a given set of constraints. This concept has been used in the following models within the game:

The three playable characters: the old lady, the sheep and the dog.
To analyze an example, the dog is composed like the graph in the image describes.
To avoid cluttering, we consider all face details all son nodes of head: both ears, both eyes and the nose, which are all brothers.



Each element is initialised as a new geometry with its measures, then its material is added (which can include a texture as well as many other properties). The upper part of the leg gets added to this.group, which represents the main part of the animal (the body) and positioned within the body. Afterwards the lower part gets created and added to the upper leg and the same happen to the paw in relation to the lower.

The car model: Here, for convenience, when talking plural we consider both left and right of the symmetric componenst of the car, but they are separated objects. Windscreen(which is the upper half of the car) has two types of children: glasses, one for each side, and lines, which divides the lateral glasses in order to have the front and back windows for each side. They are all brothers, children of windscreen. Each of the four tires is a child of body and has a child rim.



The levels: The project is composed by a series of levels in a row. All those levels have no relation between them, but each level is the father of some additional elements: the cars, the tunnels and the polelights are sons of the streets; the trees and the bushes are sons of the sidewalks.

2.3 Textures

Some of the models are rendered with their textures to give detail to the scene. We also used a normal component that is multiplied to his real texture for giving a better effect to the shaders. This in particular is applied to the trees and the bushes, which have this foliage that provides a better lighting.

The decision to apply textures only to some objects was made to give to the final game a minimal and simple view, since all subsequent texture implementations were making the project look worse, less sharp and more messy than it needed to be. For this reasons we didn’t put any texture on the animals or the ground.

3. Animations

3.1 Overview

All the animation have been implemented by us, using the hierarchical models of the three characters previously mentioned and with the use three js's function:

- object.rotation.axis : which sets/offsets the rotation of an object along one axis by a certain angle in radiants.

- object.rotateOnAxis(new Axis), which is used to rotate correctly along the original axis when the animal is rotated(and thus has its relative axis moved).

For the Old Lady her animations are implemented in a different way, her legs alternate on the jumping movement together with the left arm, while the stick she uses to stand keep still.

The animations of the sheep and Dog have similar details, they start doing little jumps leaving the legs going back in order to express the energy applied to the ground in the jumping motion. Starting from how many renders/jump functions were needed by the animal to conclude the jump animation, we divided the functions to do the exact opposite rotations to go forth and back in place.

Here an example of forward and backward directions movement:

```
if(this.group.position.y >= 2 || descending){
  this.group.position.y-= Math.sin(speed)*1.5*goingFastDog;
  descending = true;
}
else{
  this.group.position.y+= Math.sin(speed)*1.5*goingFastDog;
}

if(direction == "ahead"){
  if(counter == 0|| counter == 34){
    this.group.rotation.x = 0;
    this.group.rotation.z = 0;
    legRotation = 0;
  }
  else if (counter <=16){
    this.group.rotation.x-=Math.PI*(goingFastDog/8)*(1/16) ;
  }
  else if(this.group.rotation.x < 0) {
    this.group.rotation.x+=Math.PI*(goingFastDog/8) *(1/16);
  }
}
else if (direction == "behind"){
  if(counter == 0|| counter == 34){
    this.group.rotation.x = 0;
```

```
        this.group.rotation.z = 0;

        legRotation = 0;
      }
      else if (counter <=16){
        this.group.rotation.x+=Math.PI*(goingFastDog/8)*(1/16) ;
      }
      else if(this.group.rotation.x > 0) {
        this.group.rotation.x-=Math.PI*(goingFastDog/8) *(1/16);
      }
    }
```

Other animated object in the level are the cars, which emove along the x axis, either with it or in the opposite direction. We realized a minimalistic approach and we have given to all the cars black tires, we noticed that making the tires rotate had no visible effect and was only stressing the processing so we scrapped that animation

Talking about the crash animations, whenever the selected character enters in contact with a car it get launched and rotating in the air:

```
crashAnimation(){
  this.group.position.y+=3*crashSpeed;
  crashSpeed+=(1/8)*crashSpeed;
  this.group.rotation.y += rad(15 );
  this.group.rotation.z += rad(10);
  if(this.group.position.y > 20){
    eventMsg("Hit by a car!\n GAME OVER!");
  }
}
```

3.2 Collision detection

Talking about the detection system of these classes of objects we needed to get cars to detect the character trying to traverse them. We find this need also for trees, because we must find a way to prevent the animal to traverse them without finishing the game. all the cars and characters are equipped with an object, the hitbox, that is a simple parallelepiped that surrounds the objects. In general it's smaller than the original object in order to trigger the clash detection in a more realistic way. We use this type of hit_box to simplify the clash's check and also to make it very efficient, because we can claim that a clash happened when two hitboxes are overlapped. This condition can be verified very quickly with an unique if statement, that checks if the borders are too close to each other with respect to all axes. As defined:

```
if( (Math.abs(referencePosition.x - referencePositionCharacter.x) <= this.sideX + t
his.characterReference.sideX) &&
```

```
        (Math.abs(referencePosition.y - referencePositionCharacter.y) <= this.sideY
+ this.characterReference.sideY) &&
        (Math.abs(referencePosition.z - referencePositionCharacter.z) <= this.sideZ
+ this.characterReference.sideZ) ){
crash = true;

        }
```

The code compute where the clash happened with respect to the car, simply using the sin and cos values. This can be done because sin and cos are higher or lower to a certain threshold based on the center's position of the character:

```
          if( (sin >= this.sin) ){
            console.log("Sinistra");
          }
          else if(sin <= -this.sin){
            console.log("Destra");
          }
          if(cos >= this.cos){
            console.log("Avanti");
          }
          else if(cos <= -this.cos){
            console.log("Dietro");
          }
```

Sin and cos are computed keeping in mind also the direction of the car taken in consideration.

4 Lights

Now we will cover up all the different lights involved in the scene.

First of all, in the game the user has the possibility to choose playing in night/day mode, foggy or not, with different sources of light that come in play.

4.1 Ambient Light

To give a good distribution of the light on both night and day we added an ambient light with two different values.

4.2 Point Light

During the day, we used a particular light source that resembles the sun in color and type of lighting to imitate a sunset.

```
spotLight = new THREE.SpotLight( 0xFCCDA4, 1 );
  spotLight.position.set( 80, 100, 450 );
  spotLight.penumbra = 0.05;
  spotLight.decay = 2;
  spotLight.distance = 5000;
  spotLight.shadow.mapSize.width = 4096;
  spotLight.shadow.mapSize.height = 4096;
  spotLight.shadow.camera.near = 10;
  spotLight.shadow.camera.far = 2000;
  scene.add( spotLight );
```

4.3 Street lamps

In the night time, are spawning street lights at the left side of the road. These lights are initialized as SpotLight objects, the light is directioned to particular angle that resembles an actual street light.

```
this.spot Light =  new  THREE. Spot Light (  0 x f f f f f f  ,   0 . 6   ) ;
```

The initial idea was to add headlights to every car, but this would have brought to massive lag, so we only only put a street light every three levels of road.

4.4 Other

Another piece of optimization is made at the shadows. The shadow are casted only from the main parts of the objects, in particular for the cars, to avoid lag.

5 Other

Now we will cover up all the different lights involved in the scene.
First of all, in the game the user has the possibility to choose playing in night/day mode, foggy or not, with different sources of light that come in play.

5.1 Level implementation

The level is generated by randomizing the elements defined in it. We start spawning all the sidewalks and roads in the init function and then rendering all for every frame. In order to don't slow down the computers during the rendering of the level we decided to simply render only the tracks that are close to the character, in particular the tracks that are visible from the camera. This is done using Layer, an module provided by Three js, that allow us to label every element in the game with a value. The latter is used by the camera to know if it must render an object or not. Indeed, by default every object in the space (camera included) has value zero, as if it is a unique layer. The random generation of the road is defined by a number between 1 or 4.

The same idea is applied to the cars, their movements starts only when the player is close to the track. In order to do that we use the previous check to know if we need to update the data structure that contains all the active tracks in the game, called actualListT racks. Then it will be used by the fuctions that animate them, scanning a fewer number of tracks. if the character go back there is a piece of code that decreases the camera's label and loads the old tracks previously crossed.

The difficulty of the game is based on the choice done by the player, he can choose between easy, normal and difficult. Depending on the difficulty selected the number of cars, their velocity and other elements of the level increase or decrease modifying the specified variables.

Finally we decided to place the camera like the original Crossy Road game, an orthographic like view of the game, making clearly visible to the player all the elements needed.

```
camera = new THREE.PerspectiveCamera(75, width / height, 0.1, 1000);
 camera.position.set(-15, 50, -30);
```

5.2 Music

The music used in our project is taken as inspiration from another famous game called Subway Surfers. While for the menu options sounds and death we selected them from what we found on some sites. The music was imported by the use of the library Soundjs.

5.3 Menu

All the details of the menu, from the animation, font and shadows were implemented using the Cascading Style Sheets, in particular in the file named menu.css. We decided to apply a colorful background representing a city street.