Interactive Graphics Project

IG Sapienza

Renzo Cherubino 1764113

Matteo Fischetto 1795458

Ascanio Santomarco 1962686

27 September 2021

# Contents

1. [Introduction 2](#_TOC_250008)
2. [User Manual 2](#_TOC_250007)
   1. [Commands and User Interface elements 2](#_TOC_250006)
   2. [How to play 3](#_TOC_250005)
3. [Development environment 4](#_TOC_250004)
   1. [Developing 4](#_TOC_250003)
   2. [Library and tools 4](#_TOC_250002)
4. Technical Solutions 5
   1. Hierarchical models 5
      1. [Characters 5](#_TOC_250001)
   2. Levels 7
      1. [Trees 7](#_TOC_250000)
   3. Car 7
   4. Lights and Shadows 8
   5. Texture 10
   6. Character animations 11
   7. Vehicles animation and object detection 13
   8. Optimizations 14
   9. Music 15
5. User-game interactions 17
   1. Movement 17
   2. Camera 20
   3. difficulty 21

# Introduction

This paper is a descriptive companion to our final project for the Interactive Graphic course.

We made a clone of one of the most famous mobile game: Crossy Road.

We choose this games because it had all the prerequisites we were asked to fulfill and it looked like a fun challenge. In the following, we are going to talk about the user aspect of the game as well as all the technical aspects and the challenges we had to deal with. The end result was a good looking playable game with all the features we wanted in it.

# User Manual

## Commands and User Interface elements

In this section we will analyze all the things that the user can do in our game.

Starting from the home page the user can either:

* + - Play: which will bring to the options to pick before being able to play.
    - Commands: which will bring up the following screen showing which movement the user can do when playing.

-W: go forward

-A: go to the left

-D: go to the right

-S: go backwards

Immagine che contiene testo, clipart

Descrizione generata automaticamente

* + - Credits: brings to a page stating all the students involved in the project.

After hitting play, the user is presented with a page where he can select:

* + - Character: which character the player wants to use. All characters jump the same distance in the same amount of time, in order to not give any of them advantage over the others.
    - Day/Night: either play with the sun as the light source, or a lamp post.
    - Difficulty: pick the level of difficulty of the game, Easy/Medium/Hard.
    - Mist: if the user wants a little more fun can add a fog effect.

In game on the upper right a button will pause the game and a menu will pop up where the user can either resume the game or go back to the starting page. Clicking the upper left button will change the light from day to night and vice-versa.

## How to play

The goal of the user is to help the old lady or her pets to cross the streets and reach the safe grass field on the other side avoiding all the dangerous drivers. But if characters goes outside the viewable area, it will get lost and it’s game over!

# Development environment

## Developing

The project is made of three types of files: javascript, css and html. The environment is the basic WebGL.

## Library and tools

The tools we used were:

* + - Three.js: a library that simplifies the interaction between the programmer and WebGL, this was used to create the scene with all the objects in it, perform the movements of the animations, create the lights and shadows and add the textures.
    - OrbitControls: a sub library of three.js used to control the camera movement, which was extremely useful to check if the game was be- having correctly.
    - Browser chrome and its built-in console: to test the project during the developing using logs and alerts.
    - Visual Studio Code: it was the IDE picked by the group to code. We wanted to uniform it since some IDEs indentation is not compatible with all other IDEs.
    - Github: we’ve used Github to share the code between the group, see the commits on the repository, check the difference between versions.

# Technical Solutions

## Hierarchical models

We will now analyze all hierarchical models used in the project.

The advantage of the hierarchical model approach is the simplicity in doing movement and positioning relative to the other members of the hierarchy which made the animations quite simpler to implement.

### 4.1.1 Characters

We will only analyze one of the hierarchical animal, the dog.

This is the graph of the final model, to avoid cluttering, we have defined as Face Details all son nodes of head: both ears, both eyes and the nose, which are all brothers.



Body

Tail

Head

Right Right Left Left

Back Leg Front Leg Back Leg Front Leg

Right Back Tail End Face details Lower Leg

Right Front

Lower Leg

Left Back Left Front

Lower Leg Lower Leg

Right Right Left Left

Back Paw Front Paw Back Paw Front Paw

 (Dog)

It is self-explanatory, every object in the second line/level is a child of Body and thus are all siblings. Each of the four upper part of the leg has a child being the lower part which has another child paw.

Let’s now see how the hierarchical model is implemented in WebGL using threejs:

    //Right back Leg

    const rightBackLegGeometry = new THREE.BoxBufferGeometry(0.42\*size, 0.6\*size, 0.4\*size);

    this.rightBackLeg = new THREE.Mesh(rightBackLegGeometry, this.skinMaterial);

    this.rightBackLeg.receiveShadow = true;

    this.rightBackLeg.castShadow = true;

    this.rightBackLeg.position.set(-0.4\*size, -0.7\*size, -0.8\*size);

    this.group.add(this.rightBackLeg);

    const rightBackDownLegGeometry = new THREE.BoxBufferGeometry(0.4\*size, 0.6\*size, 0.4\*size);

    this.rightBackDownLeg = new THREE.Mesh(rightBackDownLegGeometry, this.blackMaterial);

    this.rightBackDownLeg.receiveShadow = true;

    this.rightBackDownLeg.castShadow = true;

    this.rightBackDownLeg.position.set(0, -0.65\*size, 0);

    this.rightBackLeg.add(this.rightBackDownLeg);

    const rightBackPawGeometry = new THREE.BoxBufferGeometry(0.4\*size, 0.2\*size, 0.45\*size);

    const rightBackPaw = new THREE.Mesh(rightBackPawGeometry, this.whiteMaterial);

    rightBackPaw.receiveShadow = true;

    //rightBackPaw.castShadow = true;

    rightBackPaw.position.set(0, -0.3\*size, 0);

    this.rightBackDownLeg.add(rightBackPaw);

A leg has been deemed the most noteworthy example we could make.

Each element has to be created as a new geometry with its measures and have the material 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.

## Levels

The project is composed by a series of levels in a row. All those levels have no relation between them and are indeed separate hierarchical models.

Roads and Grass levels all have the same hierarchical model:



Base

Additional Element

In particular the pairs Base-Additional element are: Road-Car, Grass-Bush, Grass-Tree. If the base has more than one object, those objects are all brothers, children of the Base.

### 4.2.1 Trees



Trunk

Foliage

Simply enough, we made trees that have different heights, and they are made in such a way that an iteration occurs over the creation where foliage gets added veritcally until we reach the desired height, so each cube of foliage is a child of the trunk.

## Car



Body

Spotlights Front Bar

Handles

Fenders Windscreen

Tires

Glasses

and Lines

Rims

As before, spotlights(and fenders) means both of them, left and right, which are separated objects, children of body. Handles means all four of them.

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.

## Lights and Shadows

Now we are going to talk about lights and shadows organization in the game and how they impact on the performance of it. In particular we will discuss what the user can see in the space and how the objects interact with them.

First of all, in the game the user has the possibility to choose playing in night /day mode, foggy and, according to the choice made by him, we find

different sources of light:

* + - Day, when we set the day mode the unique source light is the sun. Basically we use a SpotLight with the following characteristics:

spot Light = new THREE. Spot Light ( 0 x f f f f f f , 1 ) ; spot Light . penumbra = 0 . 0 5 ;

spot Light.decay = 2 ;

spot Light.distance = 500 ;

spot Light.angle = Math.PI/4 ;

We used this type of light because it’s perfect to simulate a sunset, that is better to exploit the water’s reflection properties due to the greater angle of the light;

* + - Foggy, we add a texture and modify the illumination:

if(foggyDay){

  {

    const near = 10;

    const far = 50;

    const color = '#e6e1e2';

    scene.fog = new THREE.Fog(color, near, far);

    scene.background = new THREE.Color(color);

  }

* + - Night, in the night mode we have a street lamp in the starting point of the game. Also in this case we used a SpotLight object, but this time we decreased the angle of it (because of course a lamp can’t have the same range of the sun) and we decrease the intensity of it:

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 due to the very high number of cars in the game this would have brought to some performance slowdowns in some cases, so we discarded that idea.

In addiction to the Spotlights we also added an ambient light in order to a better distribution of the light, specially in night mode:

ambient Light = new THREE.AmbientLight ( 0 x ffffff , 0.6) ;

For the day we have the same object but with a greater intensity equal to 1*.*1.

About that, the second aspect of the project related to lights are shadows. These are very important in order to make the scene and the animation more realistic, but at the same time they are very heavy in terms of computational power, specially without a GPU.

So it’s very important to find a compromise in terms of shadows’ definition and details and performance.

There are two options that can be added to every object and both add a functionality related to the shadow:

* + - Receive shadow, this option allows the objects to receive the black color of the shadow. This isn’t too heavy so can be added without any big trouble. Indeed we added it to every object in the game, but at the same time it’s fundamental in order to visualize the shadows;
    - Cast shadow, this other option instead enables the generation of shadow, simply denoting to Three js if the objects have to block the light or not. This operation is very heavy indeed also in the documentation of the library is recommended to use with parsimony. So we added this function only to the main parts of the objects.

In particular here we analyze what casts a shadow in the game and why:

* + - The characters cast mainly the body, head and arts’ shadow when they have them, without casting the shadow of little details like eyes, that produce an insignificant variation to it;
    - Cars, due to the high number of them, cast only the body and the windscreen that are the two main component of the object. The other are only little details;
    - Woods, they don’t cast shadow because simply there isn’t a plan when we should see their shadow, because they are floor’s element;
    - Trees, every part of them casts a shadow because they are very simple;
    - Tracks, as for the woods, they don’t cast shadow for the same reason.

## Texture

Textures are the basis of most computer graphic applications.

Before starting this section, we will see three kind of textures we have applied to the same object in our project: the trees.

Immagine che contiene edificio

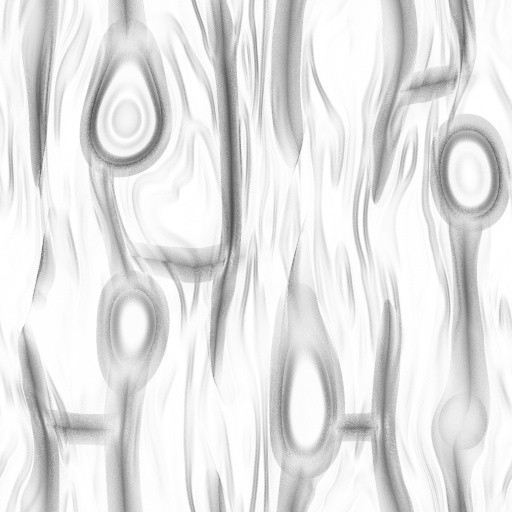
Descrizione generata automaticamente

Figure 1: Standard Figure 2: Normal Figure 3: AO

In our game, we used the textures to give detail to our models, all images used are from free-of-use libraries and sites. Briefly, when using more than one texture on WebGL, they get composed(by multiplying the ”n” textures) in WegGL’s shaders.

A normal map texture is a texture which uses rgb values to signify the orientation of the surface normal by corresponding those values to the xyz of the surface normal at any given pixel. Basically it gives a lower-detailed model(e.g. a cube) finer details with respect to the interaction with the light. In our project we’ve decided to add them for one object:

* + - Foliage and Bushes : in nature, most trees’ crown and most kinds of bushes shine under the light, we applied a normal map to capture this shininess.

And ambient occlusion(AO) texture is used to add shadow details to the object it is applied to, pronuncing its effects and giving a better sense of depth.

The decision to apply textures only to some objects was agreed by all the members because after approaching the state that can be seen in the repo, 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.

Also in the case of the water, we tried different combinations of Texture/Nor- mal Texture, but the final one without any normal mapping was way better looking than any other combination and since we already had normal textures in the project it didn’t seem a good idea to make it worse purposely.

## Character animations

Each of our animations has been hand-made by us over the hierarchical models we’ve implemented and then animated.

We will analyze the most complex and articulate one : the Old Lady.



The Old Lady, since is the main character, has one of the best(funniest animation).

this.vAngle += speed \* goingFastOldyLady;

    //check if i'm going up or down

    if(this.group.position.y >= 3 || descending){

      this.group.position.y-= Math.sin(speed)\*1.2 \* goingOldyLady;

      descending = true;

      legRotation = (-33 \* Math.PI / 180 /19 ) \* goingOldyLady;

      leftArmRotation = (-63 \* Math.PI / 180 /19 ) \* goingOldyLady;

      //rightArmRotation = (-33 \* Math.PI / 180 /19 ) \* goingFastOldyLady;

    }

    else{

      legRotation  = (33 \* Math.PI / 180 /19 ) \* goingFastOldyLady;

      leftArmRotation = (63 \* Math.PI / 180 /19 ) \* goingFastOldyLady;

      //rightArmRotation = (-33 \* Math.PI / 180 /19 ) \* goingFastOldyLady;

      this.group.position.y+= Math.sin(speed)\*1.2 \* goingFastOldyLady;

    }

    if(step) {

      this.rightLeg.rotation.x += legRotation;

      this.leftLeg.rotation.x -= legRotation;

      this.leftWing.rotation.x += leftArmRotation;

    }

    else {

      this.rightLeg.rotation.x -= legRotation;

      this.leftLeg.rotation.x += legRotation;

      this.leftWing.rotation.x -= leftArmRotation;

    }

    step\_iter = step\_iter + 1;

    if (step\_iter==38) {

      step = !step;

      step\_iter = 0;

    }

    //had to speed up the movement since i'm using a different incremental function

    if(asse=='z') this.group.position.z = this.group.position.z + 1.0387\*dist\*goingFastOldyLady;

    if(asse=='x') this.group.position.x = this.group.position.x + 1.0387\*dist\*OldyLady;

Apart from the jump animations, all characters have another animations:

* CrashAnimation: happens whenever a car and the animal comes in contact. We wanted to achieve a rather humorous effect, in fact the animal gets propelled towards the air way faster than it should be, while rotating along two different directions. The details of how the detection happens will be explained in the next section.

All animations were done by using the hierarchical model of the animals and then using 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).

## Vehicles animation and object detection

The other class of objects moving is the cars’s class.

Simply enough, the cars move along the x axis, either with it or in the opposite direction. Since we wanted 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.

A more interesting aspect about this class of objects is the crashes detection system. Indeed we needed a way to get cars to detect character trying to traverse them. We find this need also for trees and lamp, because we must find a way to prevent the character to traverse them without finishing the game. Now since they are handled in different ways (although they are very similar in spirit) let’s analyze all of them separately:

* + - Cars detection, here the main technique to detect a clash is using hit- boxes. Indeed, all the cars and animals are equipped with an object, called 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:

1. if(!crash){
2. if ((tot > referencePositionAnimal.z + 2.5 ) || referencePositionAnimal.x >33 || referencePositionAnimal.x <-33) {
3. crash = true;
4. pause = true;
5. outrun = true;
6. eventMsg("Outrunned!");
7. }
   * + Tree detection, the last detection system is the one used with trees. In this case we need to prevent that the animal beats the trees, so we need to add a condition in the *actionOnPressKey*, in order to block the animation. The interesting fact is that of course the tree can block only one animation, i.e. the one that go in front of the tree, so this is another element that must be managed. Every time the user asks to move the animal it verifies that no tree is hindering the animal, calling the *checkTrees* method. The if statement in this function uses the same idea used with the other detection, so we don’t report it. What is interesting is that we pass to the function the possible future position of the animal after the jump and then it checks if it’s possible to do it:
8. function checkTrees(position){
9. var lengthTrees;
10. var i, j;
11. var trees;
12. var referencePosition = new THREE.Vector3();
13. var length = actualListTracks.length;
14. for(i = 0; i < length; i++){
15. trees = actualListTracks[i].trees;
16. lengthTrees = trees.length;
17. for(j = 0; j < lengthTrees; j++){
18. trees[j].trunk.getWorldPosition(referencePosition);
19. if( (Math.abs(referencePosition.x - position.x) <= animal.sideX + trees[j].sideX) &&
20. (Math.abs(referencePosition.z - position.z) <= animal.sideZ + trees[j].sideZ) ){
21. return true;
22. }
23. }
24. }
25. return false;
26. }

So using only one function we can do a check for every direction, because when you call *checkTrees* you know where to go and the direction of the animation.

## 4.8 Optimizations

In this section we are going to analyze the optimization techniques adopted in the project and how much they increase the perfomance of the game.

The first optimization was added using a layered organization of the random map. Indeed, at the beginning of the development we start spawning all the tracks (Road) in the init function and then rendering all of them (consequently also their children element) for every frame. Obviously, this may bring to some slowdowns, specially in computers without GPU.

So, in order to resolve the issue, we 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. Hence, in order to render a limited number of tracks *n* in each frame, we need to assing the labels so that every track is visible for *n* layer. The following is an example of *n* = 4 that should clarify how to do it:

|  |  |  |  |
| --- | --- | --- | --- |
| *Track*3 : | 1 | 2 | 3 4 |
| *Track*2 : | 0 | 1 | 2 3 |
| *Track*1 : | 0 | 1 | 2 |
| *Track*0 : | 0 | 1 |  |

(1)

if the camera is set to layer 2 we will see tracks number 1*,* 2*,* 3. Of course in the code we find a parametric code that assigns the label correctly, using the variable called *numOf LevelVisible*. Then of course the camera needs to increase its value every time the animal comes near a new layer as we can see in the render function:

if(referencePositionAnimal.z > limitMax){

      actualTrack++;

      limitMin = limitMax;

      limitMax = mappingTracks[actualTrack];

      actualListTracks = tracks.slice(actualTrack - 1, actualTrack + numOfFrontActiveLevels + 1);

      actualLevelCamera++;

      actualLevelCamera = actualLevelCamera % 32;

      camera.layers.set(actualLevelCamera);

      }

Here the label of the camera is increased every time the animal reaches a new layer.

The same idea is used also for cars and trunks’ animations. Indeed, their movements starts only when the player is close to the track (in particular it starts only for the current, next and previous 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 *actualListTracks*. Then it will be used by the fuctions that animate them, scanning a fewer number of tracks. Also this optimization is adopted by the trees detection because they are child of the tracks, so they are affected by the activation of only a part of them. Of course if the animal trys to go back there is a piece of code that decreases the camera’s label and loads the old tracks previously crossed.

The final result is a more fluid game, in our tests we passed from 20 fps to 50 fps in similar map situations.

## 4.9 Music

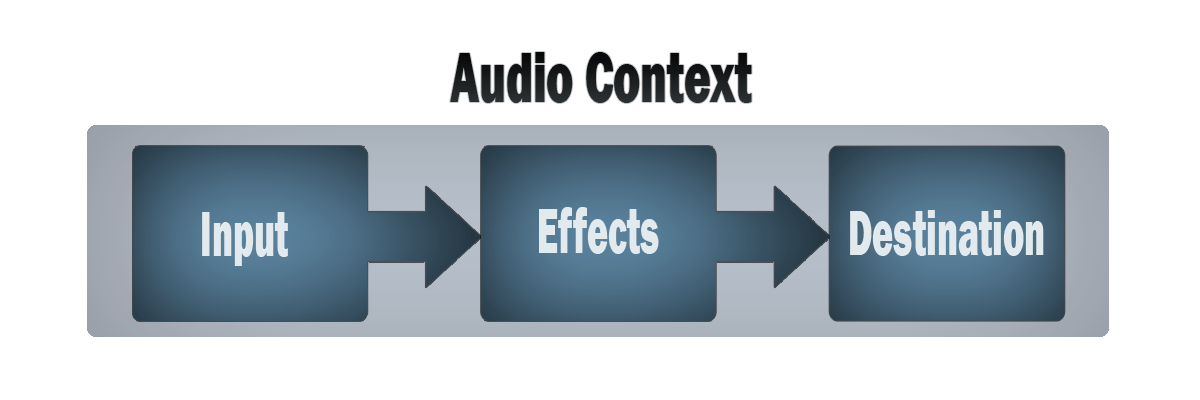
The Web Audio API involves handling audio operations inside an audio context, and has been designed to allow modular routing. Basic audio operations are performed with audio nodes, which are linked together to form an audio routing graph. Several sources — with different types of channel layout — are supported even within a single context. This modular design provides the flexibility to create complex audio functions with dynamic effects.

Audio nodes are linked into chains and simple webs by their inputs and outputs. They typically start with one or more sources. Sources provide arrays of sound intensities (samples) at very small timeslices, often tens of thousands of them per second. These could be either computed mathematically (such as [OscillatorNode](https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode)), or they can be recordings from sound/video files (like [AudioBufferSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode) and [MediaElementAudioSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/MediaElementAudioSourceNode)) and audio streams ([MediaStreamAudioSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioSourceNode)). In fact, sound files are just recordings of sound intensities themselves, which come in from microphones or electric instruments , and get mixed down into a single, complicated wave.

Outputs of these nodes could be linked to inputs of others, which mix or modify these streams of sound samples into different streams. A common modification is multiplying the samples by a value to make them louder or quieter (as is the case with [GainNode](https://developer.mozilla.org/en-US/docs/Web/API/GainNode)). Once the sound has been sufficiently processed for the intended effect, it can be linked to the input of a destination ([BaseAudioContext.destination](https://developer.mozilla.org/en-US/docs/Web/API/BaseAudioContext/destination)), which sends the sound to the speakers or headphones. This last connection is only necessary if the user is supposed to hear the audio.

A simple, typical workflow for web audio would look something like this:

1. Create audio context
2. Inside the context, create sources — such as <audio>, oscillator, stream
3. Create effects nodes, such as reverb, biquad filter, panner, compressor
4. Choose final destination of audio, for example your system speakers
5. Connect the sources up to the effects, and the effects to the destination.



So we use the threejs music library to implement the music in the application:

function loadSounds() {

  sound.addEventListener("fileload", startSoundtrack);

  sound.registerSound("audio/City\_sounds.ogg", 'ambient');

  sound.registerSound("audio/Subway\_Surfers.ogg", 'music');

  sound.registerSound("audio/Death\_sound.ogg", 'death');

}

function startSoundtrack(event) {

  if (event.id == "ambient") {

    soundtrack = sound.play('ambient');

    soundtrack.volume = 0.3;

  }

  if (event.id == "music") {

    soundtrack = sound.play('music');

    soundtrack.volume = 0.3;

  }

}

# 5 User-game interactions

We will explain how the interactions are implemented. UI elements are skipped since they are simple html/js functions setting up flags and activat- ing part of main code.

## 5.1Movement

We use a javascript function(visible in crossyRoads.js) onKeyDown to check if any between A,S,W and D has been pressed. Then each character has an actionOnPressKey which responds differently based on the key pressed. We will present the code for the press of W for the Dog, some of the more wordy parts have been replaced by pseudocode, the full js code can be seen in the repository. Start jumping:

actionOnPressKey(referencePositionAnimal) {

    if(inMotion){

      this.jump(0.06, state);  //keep it going till the jump is complete, we don't want the animation to stop mid-air, neither the user to press too many buttons together

    }

    else{

      referencePosition.copy(referencePositionAnimal);

      if (keyWDown){

        referencePosition.z += 3.75;

        if( !checkTrees(referencePosition) ){

        currentScore++;

        inMotion = true;

        this.group.rotation.y = 0;

        if(state != 'ahead' && state != 'behind'){

          var temp = this.sideX;

          this.sideX = this.sideZ;

          this.sideZ = temp;

        }

        state = "ahead";

        counter = 0;

        oldPos = this.group.position.z;

        this.jump(0.06, state);

        }

      }

      else if (keyDDown){

        referencePosition.x -= 3.75;

        if( !checkTrees(referencePosition) ){

          if(state != 'left' && state != 'right'){

            var temp = this.sideX;

            this.sideX = this.sideZ;

            this.sideZ = temp;

          }

          state = "right";

          inMotion = true;

          this.group.rotation.y = Math.PI\*3/2;

          counter = 0;

          oldPos = this.group.position.x;

          this.jump(0.06, state);

        }

      }

      else if (keyADown){

        referencePosition.x += 3.75;

        if( !checkTrees(referencePosition) ){

          inMotion = true;

          if(state != 'left' && state != 'right'){

            var temp = this.sideX;

            this.sideX = this.sideZ;

            this.sideZ = temp;

          }

          state = "left";

          this.group.rotation.y = Math.PI/2;

          counter = 0;

          oldPos = this.group.position.x;

          this.jump(0.06, state);

        }

      }

    else if (keySDown){

      referencePosition.z -= 3.75;

      if( !checkTrees(referencePosition) ){

        currentScore--;

        document.getElementById("cScore").innerHTML = currentScore;

        inMotion = true;

        this.group.rotation.y = Math.PI;

        if(state != 'ahead' && state != 'behind'){

          var temp = this.sideX;

          this.sideX = this.sideZ;

          this.sideZ = temp;

        }

        state = "behind";

        counter = 0;

        oldPos = this.group.position.z;

        this.jump(0.06, state);

      }

    }

  }

}

Jump function:

jump(speed, direction) {

    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);

      }

    }

    else if (direction == "right" || direction == "left"){

      if(counter == 0|| counter == 34){

        this.group.rotation.z = 0;

        this.group.rotation.x = 0;

        legRotation = 0;

      }

      else if (counter <=16){

        this.group.rotateOnAxis(new THREE.Vector3(1,0,0) , -Math.PI\*(goingFastDog/8)\*(1/16));

      }

      else if(this.group.rotation.x < 0) {

        this.group.rotateOnAxis(new THREE.Vector3(1,0,0) , +Math.PI\*(goingFastDog/8)\*(1/16))

      }

    }

## 5.2 Camera

As it can be seen in game, the camera starts going forward as soon as the user makes a jump forward. And it keeps on going forward until either a game over or a win occurs also due to the user being able to outrun a lot the camera in the easier difficulties, the movement is parametric to the distance between the camera and the animal positions.

  if(highestScore == numberOfJumps){

    eventMsg("You Win!");

    crash = true;

    pause = true;

  }

  if(!crash){

    if ((tot > referencePositionAnimal.z + 2.5 ) || referencePositionAnimal.x >33 || referencePositionAnimal.x <-33) {

      crash = true;

      pause = true;

      outrun = true;

      eventMsg("Outrunned!");

    }

    else if(highestScore != 0){

      if((referencePositionAnimal.z - tot >= 0) && (highestScore < numberOfJumps)){

        tot+=diffModifier\*(1+ (referencePositionAnimal.z - tot)/4);

      }

      else if (highestScore < numberOfJumps){

        tot+=diffModifier;

      }

    }

    camera.position.set(-10, 20, tot); // posizione camera

This is done in the render position, meaning that at each frame rendering, the camera gets pushed forward and the conditions regarding win or lose status gets updated and checked.

## 5.3 Difficulty

The following table describes all values that get modified when changing difficulty.

The relative code can be seen in crossyRoad.js in the setDifficulty(difficulty) function.

Number of levels changes the number of terrain alternation that gets generated

between start and goal.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Easy** | **Normal** | **Hard** |
| **Number Of Levels** | 15 | 20 | 25 |
| **Number of Cars** | [1,2,3] | [2,3] | [2,3,4] |
| **Speeds of Cars** | [0.04,0.06,0.12] | [0.07,0.1,0.15] | [0.15,0,2,0.25] |
| **Camera’s speed modifier** | 0.035 | 0.04 | 0.05 |

A random function is applied and each line of vehicles has then a different speed.

Camera’s speed modifier can be seen in the last part of the previous code defining the base speed of the camera moving forward.