Interactive Graphics Project

IG Sapienza

Silvio Dei Giudici 1708962 ,
Marco Morella 1693765,
Fortunato Tocci 1695457

22 September 2019

Contents

1 Introduction

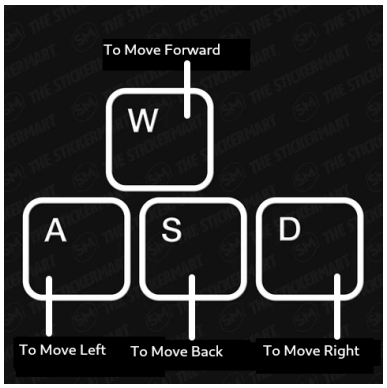
This paper is a descriptive companion to our final project for the Interactive Graphic course.
We made a clone of one of the precursor to the modern mobile games, because it had all the prerequistes we were asked to fullfil and it looked like a fun challenge. In the following, we are going to talk about the user aspect of the game as well as all the technical aspect and the challenges we had to deal with. We divided evenly the work weight between all the members while giving support to each other.
The end result was a good looking playable game with all the features we wanted during the designing period(and more).

2 User Manual

2.1 Commands and User Interface elements

In this section we are going to analyze everything 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.



- 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 of the three animals he wants to use. All animals jump the same distance in the same amount of time, in order to not give any of them advantage over the others. In the first draft, the fox jumped longer, the chicken jumped higher and descended slower.
- 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.

In game, on the upper side we will see the high score and current score reached during this run.
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.

2.2 How to play

The goal of the user is to help the animal cross the streets and the rivers to reach the safe grass field on the other side avoiding all the dangerous drivers. But if the animal goes outside the viewable area, it will get lost and it's game over!
Also, the animal has no idea how to swim across and neither has any airbags, so getting in the river or being hit by a car is game over!

3 Development environment

3.1 Developing

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

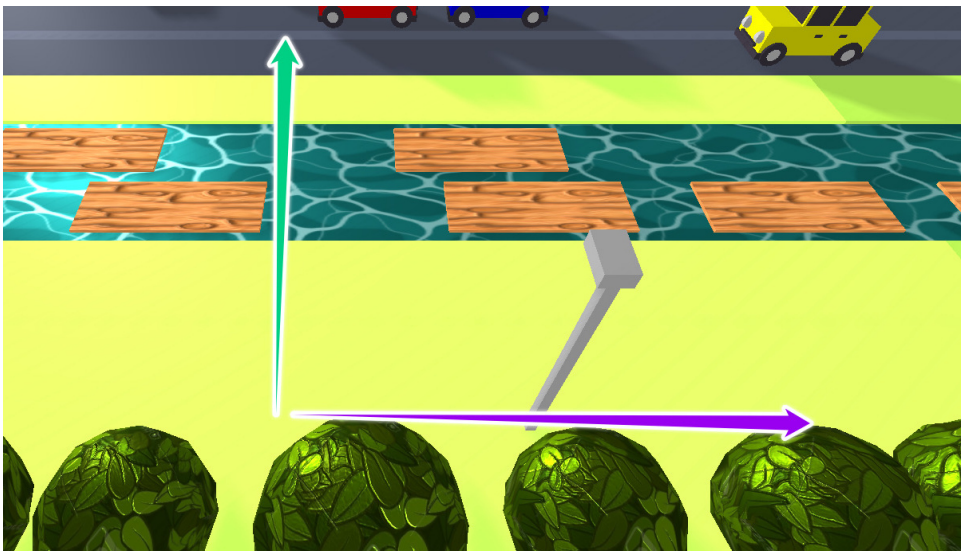
3.2 Library and tools

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 behaving correctly.
- Browser firefox and its built-in console: to test the project during the developing using logs and alerts.
- Atom: 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.

4 Technical Solutions

In order to better understand some of what we will discuss later, we want to introduce the directions that are seen in the game:



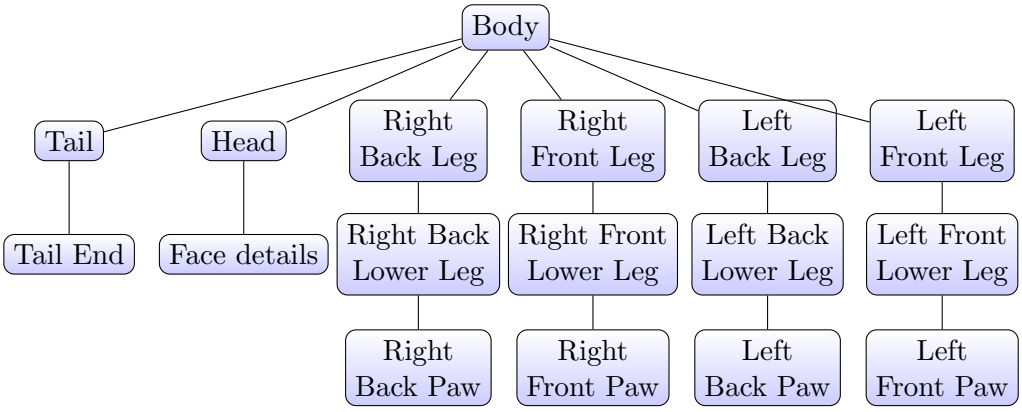
The green arrow represents the positive direction of the Z axis, while the purple one represents the positive direction of the X axis.
The Y axis is trivially the direction from the ground.

4.1 Hierarchical models

We will now analyze all hierarchical models used in the project. The vantage of the hierarchical model approach is the simplicity in doing movement and posi- tioning relative to the other members of the hierarchy which made the animations quite simpler to implement.

4.1.1 Animals

We will only analyze the most complex hierarchical animal model, the fox, since the other two are similar and easier.
This is the graph of the final model, to avoid cluttering, i've defined as Face Details all son nodes of head: both ears, both eyes and the nose, which are all brothers.



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:

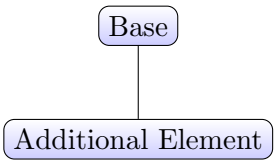
```
const rightBackLegGeometry =
    new THREE.BoxBufferGeometry(0.2, 0.4, 0.2);
this.rightBackLeg =
     new THREE.Mesh(rightBackLegGeometry, this.skinMaterial);
this.rightBackLeg.position.set(-0.25, -0.3, -0.45);
this.group.add(this.rightBackLeg);

const rightBackDownLegGeometry =
    new THREE.BoxBufferGeometry(0.2, 0.4, 0.2);
this.rightBackDownLeg =
    new THREE.Mesh(rightBackDownLegGeometry, this.blackMaterial);
this.rightBackDownLeg.position.set(0, -0.4*size, 0);
this.rightBackLeg.add(this.rightBackDownLeg);

const rightBackPawGeometry =
    new THREE.BoxBufferGeometry(0.2, 0.1, 0.2);
const rightBackPaw =
    new THREE.Mesh(rightBackPawGeometry, this.whiteMaterial);
rightBackPaw.position.set(0, -0.25, 0);
this.rightBackDownLeg.add(rightBackPaw);
```

This is the standard way in three js to create a hierarchical model. 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 represent the main part of the animal(the body) and positioned w.r.t.the body. Afterwards the lower part gets created and added to the upper leg and the same happens

to the paw in relation to the lower.

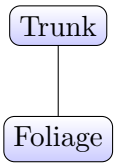
4.2 Levels

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,Rivers and Grass levels all have the same hierarchical model:



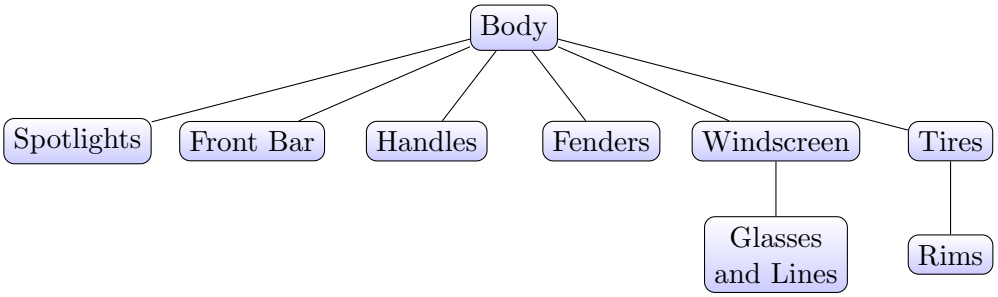
In particular the pairs Base-Additional element are:River-Log, Road-Car, Grass-Bush, Grass-Tree. If the base has more than one object, those objects are all brothers, children of the Base.

4.2.1 Trees



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.
We skip over Log and Bushes hierarchical models because they are made by one single element.

4.3 Car



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.

4.4 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 mode or day mode and, according to the choice made by the 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 with the following characteristics:

  ```
  spotLight = new THREE.SpotLight( 0xffffff, 1 );
  spotLight.penumbra = 0.05;
  spotLight.decay = 2;
  spotLight.distance = 500;
  spotLight.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;

- 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 decrease 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.spotLight = new THREE.SpotLight( 0xffffff, 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 the 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:

```
ambientLight = new THREE.AmbientLight( 0xffffff, 0.6 );
```

the latter is the ambient light during the night. For the day we have the same object but with a greater intensity equal to 1.1. Here we report an image of the street lamp in action generating also some shadow.

About that, the second aspect of the project related to lights are shadows. These are very important in order to make more realistic the animation

and the scene, 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 enable the generation of shadow, simply denoting to Three js if the object 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 animals 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 shuold see their shadow, because they are floor's element;
- Trees, every part of them cast a shadow because they are very simple;

- Tracks, as for the woods, they don't cast shadow for the same reason.

4.5 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 logs floating the river.

Figure 1: Standard

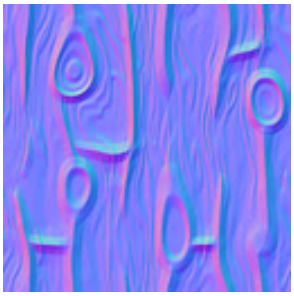


Figure 2: Normal

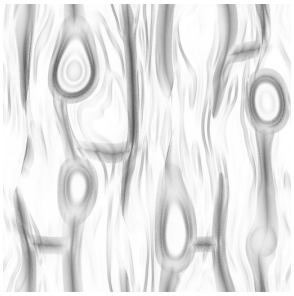


Figure 3: AO

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 only to two objects:

- Logs : in order to make the logs more realistic(and as a collateral, more "dynamic") we've added to them a normal map. But since they are made of wood, they should not reflect much light and thus we made them a MeshPhongMaterial, which is a three js material which won't shine as much as the Lambert one. While we were not using the Phong material, it was reflecting an absurd amount of light with the normal mapping active and would've been an extremely realistic effect on most other materials.

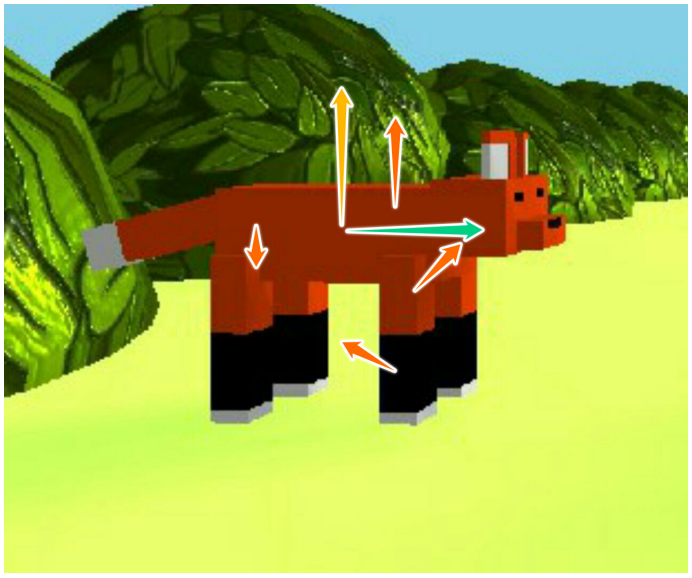
- Foliage and Bushes : in nature, most trees' crown and most king of bushes shine under the light, we applied a normal map to capture this shininess.

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 implementation 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/Normal 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.

4.6 Animals animations

Each of our animations has been hand-made by us over the hierarchical models we've implemented and then animated.
Since they are all quite similar, we will analyze the most complex and articulate one : the Fox.



In the previous figure, the orange arrows represent the rotations(with respect to the x axis going in the opposite direction from the viewer towards the fox) which will affect the animal, namely the torso rotation upward, the upper leg and lower leg rotating in opposite direction. Furthermore the animal will traslate toward the directions it is facing, Z positives(green arrow) in our image, and upward in the Y axis(yellow arrow).
When the Fox reaches the peak of its jump, it will suffer rotations opposed to the one we've described, in order to reset for the next jump, since it is simply the inverse of these, an image was deemed unnecessary. The code

behind the jumps can be seen in the User-game interactions chapter.
Before doing all of this, when recognizing an input from the user, we check what key he has pressed and based on that we rotate the animal to do the correct jump.
As we were discussing, the Fox has the most complex animations, since we wanted a feel of simpicity for the sheep, and the chicken has inherently a simple jump. Thus the latters have parametric jumps where, by the use of trigonometric functions we apply rotations back and forth.
The fox is based on a counter animation. After noticing how many renders/jump functions were needed by the animal to conclude the jump animation, we divided the functions and in the first half we do the animation described in photo, while in the second part we do the exact opposite rotations to get back in place.
To compensate this complexity difference, the Sheep has also flapping ears which move when jumping and the Chicken flaps its wings.
Apart from the jump animations, all animals have two other animations:

- SunkAnimation: which happens whenever an animal falls in to the river. This animation shows the animal sinking(dropping hastly along the y axis) and a splash effect starts, which prompts the creation of splash particles(squares of the same color of the river, for simplicity) that rise and fall from the point of sinking.

- CrashAnimation: happens whenever a car and the animal comes in contact. We wanted to achieve a rather humurous 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).

4.7 Vehicles animation and object detection

The other two classes of objects moving are the cars and the logs on the rivers.
Simply enough, both of these 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 with white rims, we noticed that making the

tires rotate had no visible effect and was only stressing the processing so we scrapped that animation.

4.8 Vehicles animation and object detection

The other two classes of objects moving are the cars and the logs on the rivers.
Simply enough, both of these 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 these classes of objects is the clashes and sinking's detection system. Indeed we need a way in order to cars can detect an animal that try to traverse them and logs can know if there is an animal on them, so that they notify the river that the animal can't sink. We find this need also for trees, because we must find a way to prevent the animal to traverse them without finishing the game. Now since they are handle 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 hitboxes. 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 hitbox to simplify the clash's check and also to make it very efficient, because we can claim that a clash happened when two hitbox are overlaped. This condition can be verified very quickly with an unique if statement, that checks if the border are too close each other with respect all axes:

```
if((Math.abs(referencePosition.x - referencePositionAnimal.x)
    <= this.sideX + this.animalReference.sideX) &&
  (Math.abs(referencePosition.y - referencePositionAnimal.y)
    <= this.sideY + this.animalReference.sideY) &&
  (Math.abs(referencePosition.z - referencePositionAnimal.z)
    <= this.sideZ + this.animalReference.sideZ) ){
    crash = true;
  }
```

  It's also possible to 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 animal:

```
if(sin >= this.sin)
  //left of the car
if(sin <= -this.sin)
  //right of the car
if(cos >= this.cos)
  //front of the car
if(cos <= -this.cos)
  //back of the car
```

Of course sin and cos are computed keeping in mind also the direction of the car taken in consideration;

- Logs and sink detection, this type of check is conceptually very similar to the privious one, but in this case we mainly need only the x and z axes. Indeed, in order to know if an animal is on a river or not we simply check its position in the space with respect to the river. If it is within the border of the river it means that the animal may be in a sink situation. After this check in the *doCheck* function of the river, it also verify if there is a trunk or not under the animal before triggering the sink animation, blocking the game:

```
if((Math.abs(referencePosition.x - referencePositionAnimal.x)
    <= this.sideX) &&
  (Math.abs(referencePosition.z - referencePositionAnimal.z)
    <= this.sideZ) &&
  referencePositionAnimal.y <= animal.restHeight && !checkIsWood){
      splash = true;
  }
```

Not mentioned before, the river also check the height of the animal. This is done in a different way with respect x and z, because we only need to know if the animal is at the level of the floor. In a very similar way, all the woods verify the same condition monitoring only x and z, so that the river know if the animal is on the trunk:

```
if( (Math.abs(referencePosition.z - referencePositionAnimal.z)
    <= this.sideZ) &&
  (Math.abs(referencePosition.x - referencePositionAnimal.x)
    <= this.sideX) )
      this.isWood = true;
else
      this.isWood = false;
```

setting the *isWood* variable that will be used by the river;

- 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 ask to move the animal it verifies that no tree is hindering the animal, calling the *checkTrees* method. The if statement in this function use 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 check if it's possible to do it:

```
referencePosition.z += 3.75; //when you move on
                        //(in front of the animal)
if( !checkTrees(referencePosition) )
  //do the jump animation
```

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.

4.9 Optimizations

In these 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 (River or 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 animal, 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) have 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:

$$\begin{array}{lllll} Track3: & 1 & 2 & 3 & 4 \\ Track2: & 0 & 1 & 2 & 3 \\ Track1: & 0 & 1 & 2 & \\ Track0: & 0 & 1 & & \end{array} \tag{1}$$

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 assign the label correctly, using the variable called $numOfLevelVisible$. Then of course the camera need to increase its value every time the animal come near a new layer as we can see in the render function:

```
if(referencePositionAnimal.z > limitMax){
        actualLevelCamera++;
camera.layers.set(actualLevelCamera);
}
```

here the label of the camera is increased every time the animal reach 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 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 try to go back there is a piece of code that decrease the camera's label and load the old tracks previously cross.

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

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 activating part of main code.

5.1 Movement

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 animal has an actionOnPressKey which responds differently based on the key pressed. We will present the code for the press of W for the Sheep, 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){
```

```
    //continue the jump until its end
    this.jump(<params previously saved>);
  }
  else{
    if (keyWDown){
      checkTrees();
      //check if any tree is obstructing the jump, if true, don't move
       currentScore++;
      //Resetting parameters and preparing for inMotion
      inMotion = true
      this.jump(speedSheepUp, dist, 0, 'z');
      }
    }
   //other keys
```

Jump function:

```
jump(speed, dist, gradi, ax) {
  this.group.rotation.y = rad(gradi);
  this.vAngle += speed*goingFastSheep;
  //check if jumping up or down, keep moving or switch it if
  //jump peak reached
  checkJump();

  //rotations were done with trigonometry and arbitrary parameters.
  const legRotation = Math.sin(this.vAngle) * Math.PI / 6 + 0.4;
  applyRotation(legs);
  //check the direction and slightly move the animal accordingly
  moveForward(ax);
  const earRotation = Math.sin(this.vAngle) * Math.PI / 3 + 1.5;
  applyRotation(Ears);
  //check if the jump has to end
  if(this.group.position.y <= 0.4){
        resetFlagsAndHeight();//inMotion = false is the most important
        //update the score which is seen in the higher part of
        //the screen only if we are moving toward the goal
    if(ax=='z') {
      if(sign == 1){
        document.getElementById("cScore").innerHTML = currentScore;
        if(currentScore > highestScore){
          highestScore = currentScore;
          document.getElementById("hScore").innerHTML = highestScore;
        }
      }
}
```

5.2 Camera

As it can be seen in game, the camera starts going forward as soon as the user make 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 == numberOfJumpsToDo){
  //reached the goal, stop the game and output victory screen
  flag = false
  eventMsg("You Win!");}
if(!flag){
  if ((tot > referencePositionAnimal.z + 1.5 ) ||
    (referencePositionAnimal.x.isOutOfBounds){
    //Animal got out of the viewable area, game over
    stopGame();
    eventMsg("Outrunned!");
  }
  //move only if user jumped at least once forward
  else if(highestScore != 0){
    if(referencePositionAnimal.z - tot >= 0){
      //setUp the camera based on the distance with the animal
      tot+=diffModifier*(1+ (referencePositionAnimal.z - tot)/4);}
  }
  camera.position.set(0, 15, tot);
```

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

	Easy	**Medium**	**Hard**
Number Of Levels	6	10	16
Number of Cars	[0,2,3]	[1,2,3]	[2,3,4]
Speeds of Cars	[0.04,0.05,0.05,0.12]	[0.06,0.08,0.15]	[0.15,0.18,0.25]
Speeds of Logs	[0.01,0.02,0.05]	[0.03,0.04,0.1]	[0.05,0.06,0.13]
Camera's speed modifier	0.035	0.04	0.05

alternation that gets generated between start and goal.
Speed of cars and logs are two lists used when creating such vehicles. 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.