FARM DEFENDER

Interactive Graphics Project

Student

Graziano Specchi 1620431

Chapters

Code

The code developed is “***code2_/easy/medium/hard/impossible.js*”,** that contains the main part of the project and the *requestAnimationFrame()* call, “***FoxWalk.js*”** contains functions for animation of the fox and “***Dog.js*”** contains the code for building the dog’s structure. All the others are external libraries.

0 - Introduction

The aim of this project is to create a 3D game through the "*three.js*" library, one of the most popular Javascript libraries for displaying 3D content on the web. The player is in first person, so he can move on the field and can also jump. He is in a farm and he has a stick in his right hand. In the scene there are a windmill, a well, some trees, a square fence containing dogs, and there are foxes arriving from all the 4 sides of the fence trying to enter it and to eat the food of the dogs (a cooked chicken). The aim of the player is to scare the foxes by playing the stick in order to avoid them entering the fence. The more foxes the player scares, the more score he gets. The game finishes in one of the following four ways:

1. A fox gets too close to the fence.
2. The player goes too far from the fence.
3. The time is over.
4. The player presses ESC two consecutive times.

1 - Technologies

1.1 - Three.js

Three.js is a JavaScript library and an API (Application Programming Interface) that is used to create and display animated 3D computer graphics in a web browser without relying on proprietary browser plugin: this is possible due to the advent of *WebGL*. *Three.js* runs in all browsers supported by *WebGL 1.0*. The version that has been used is the '*r119*' released in August 2020, and some of the main features of this API are:

- Scenes: add and remove objects at run-time;
- Cameras: perspective, orthographic, cube, array and stereo;
- Geometry: planes, cubes, spheres, torus;
- Objects: meshes, particles and more;

- Materials: basic, Lambert, Phong, smooth shading, texture and more;
- Lights: ambient, directional, point, spot and shadows;
- Data loaders: binary, image, JSON and scene.

The *three.js* GitHub repository is full of ready to use useful examples like the loaders, so I used different loaders that will be cited.

1.2 - PointerLockControls.js

PointerLockControls implements the inbuilt browsers Pointer Lock API. It provides input methods based on the movement of the mouse over time (i.e., deltas), rather than the absolute position of the mouse cursor in the viewport, in order to have more sophisticated and smooth effects. It gives access to raw mouse movement, locks the target of mouse events to a single element, eliminates limits on how far mouse movement can go in a single direction, and removes the cursor from view. It is a perfect choice for first person 3D games and I used it in this project.

1.3 - Tween.js

Tween.js is a library that helps the programmer to implement animations. A tween (from *in-between*) allows to change the values of the properties of an object in a smooth way. It is sufficient to pass the properties we want to change, which final values should they have when the tween finishes running, and how long it should take, and the tweening engine will take care of finding the intermediate values from the starting to the ending point. Notice that an animation depends on the time elapsed since the call of the *start()* method on a given tween. In this way, the animation is time-consistent.

2 - The Scene

2.1 - Camera and Controls

The camera used is a perspective camera, and it is given to the PointerLockControls. There is a listener for the buttons that allow the player to interact with the scene, namely to move, jump and shake the stick. The player cannot enter the fence or any other object in the scene except the foxes that are not tall in height and then he can cross them without any problem. In order to disallow the player to enter the given places, I wrote a function that checks the camera's coordinates and if its position is not allowed, then the camera will not go ahead and its position will be set a bit backward, in order to simulate the object that the player is hurting rejected him. The velocity of the movements of the player is time based, namely there is a simple implementation of acceleration for both the movements on the plane and vertical ones (jump) that lets the player feel the experience more realistics.

2.2 - Light

There are two types of scenes: day and night.

In the day mode, there is just one HemisphereLight that simulates the daily sunny light. In the night mode, the only light that is present is emitted by four PointLight that simulates the light scattered by four fires that are close to the fence.

2.3 - Floor

The floor is a PlaneBufferGeometry. Its texture has been downloaded and it has been repeated in both S and T coordinates, so to fill it.

2.4 - Dogs

There are three dogs in the fence, each one of a different scale. Both the structure of the dog and its animation are programmed using just *three.js* primitives without any external libraries. I defined a class for the dog entity, in which I built from scratch each component of the dog through the *three.js* mesh primitives, I assigned to it the textures and gave the right scaling, translation and rotation to each component. The texture for the fur has been downloaded from the web, while the texture for the face, eyes, mouth and teeth has been drawn manually through GIMP for this specific case, and in order to load it, I used the MeshPhongMaterial primitive so that in the IMPOSSIBLE mode in which is night it is possible to see the different light effects on the dog's surfaces. In this class, shading is calculated using a Phong shading model. This calculates shading per pixel (i.e. in the fragment shader, AKA pixel shader) which gives more accurate results than the Gouraud model used by MeshLambertMaterial, at the cost of some performance. The animation consists of moving the dog's legs, head and tail and translating it a bit forward, through the *three.js* primitives of rotation and translation. Moreover, if the player disturbs a dog by playing the stick against one of them, all the dogs will accelerate the rotation of their tails and the score will be decreased. Figure 1 shows the structure.

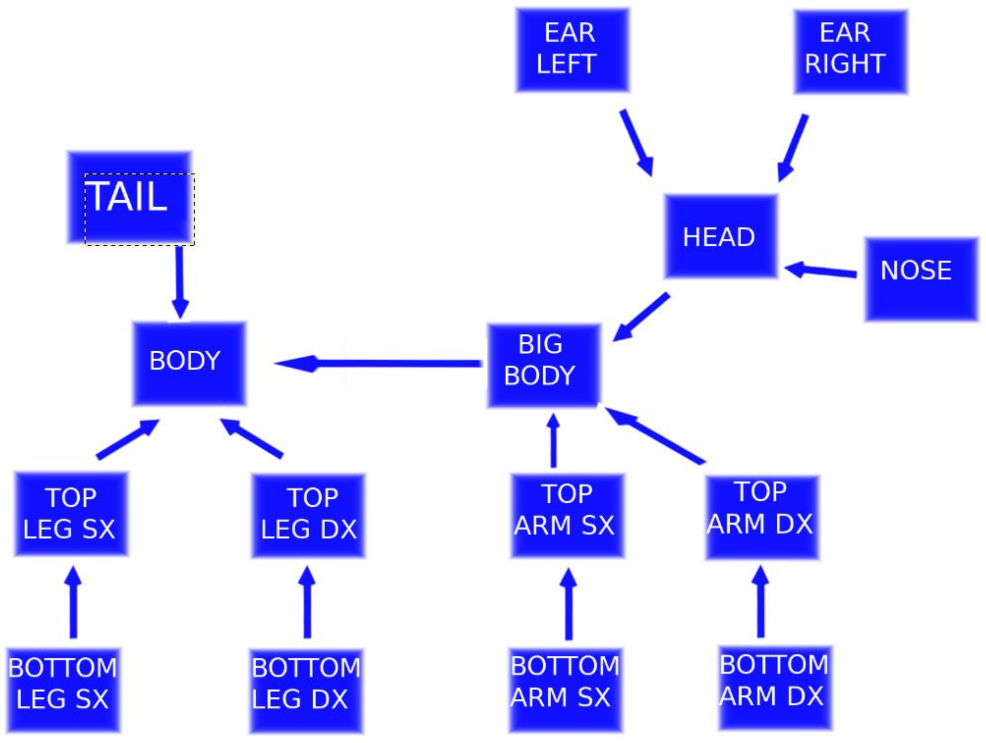


Figure 1) Hierarchical structure of the dog. Arrow starts from the child and points to its parent.

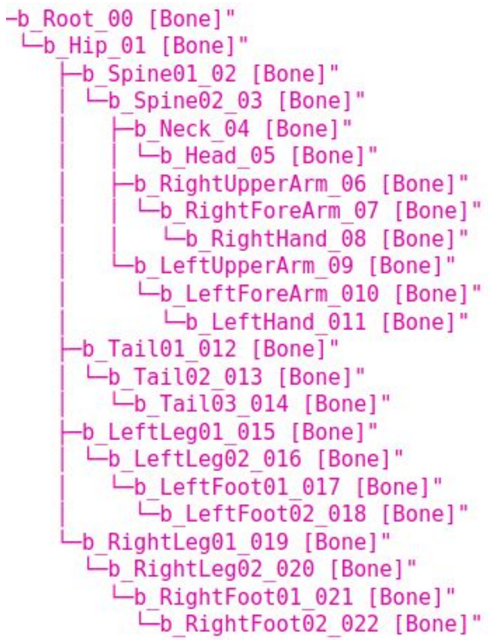
2.5 - Foxes

The foxes are spawned with a frequency that depends on the level the player chose in the main index. There are four square parts of the field in which a fox can be spawned, and these squares are in the four possible sides of the fence: north, south, east and west. A fox cannot be spawned neither too far from the fence nor too close. The foxes are always directed against the fence. For the random position I used the Math.random() function. The model of the fox (geometry, material and textures) has been downloaded from CGTRADER in GLTF format and although it included animations, only the hierarchical structure has been used, while the animations have been programmed by using the Tween.js library. The fox can perform one of two types of animation: walking or running. When a fox is spawned, it plays the walking animation toward the fence. If the player scare it by playing the stick against it within the minimum distance that is required:

1. the fox becomes red,
2. the previous tween animations are stopped,
3. a new set of tween animations starts and it consists of a chain of two animations in which in the first sequence it starts to move faster its legs and rotate, and then when it has rotated by 90 degrees it starts the second animation in which it runs away.

Note that each animation consists of many different tweens, each of them acts on a different component of the hierarchical structure of the fox.

In figure 2) there is the hierarchy of the fox's structure, taken from a snapshot of a print in the browser's console.



```
-b_Root_00 [Bone]"
 └b_Hip_01 [Bone]"
   ├b_Spine01_02 [Bone]"
   │ └b_Spine02_03 [Bone]"
   │   ├b_Neck_04 [Bone]"
   │   │ └b_Head_05 [Bone]"
   │   ├b_RightUpperArm_06 [Bone]"
   │   │ └b_RightForeArm_07 [Bone]"
   │   │   └b_RightHand_08 [Bone]"
   │   └b_LeftUpperArm_09 [Bone]"
   │     └b_LeftForeArm_010 [Bone]"
   │       └b_LeftHand_011 [Bone]"
   ├b_Tail01_012 [Bone]"
   │ └b_Tail02_013 [Bone]"
   │   └b_Tail03_014 [Bone]"
   ├b_LeftLeg01_015 [Bone]"
   │ └b_LeftLeg02_016 [Bone]"
   │   └b_LeftFoot01_017 [Bone]"
   │     └b_LeftFoot02_018 [Bone]"
   └b_RightLeg01_019 [Bone]"
     └b_RightLeg02_020 [Bone]"
       └b_RightFoot01_021 [Bone]"
         └b_RightFoot02_022 [Bone]"
```

Figure 2) Hierarchical structure of the fox taken by a snapshot of a print of it in the console.

2.6 - Stick

The stick has been created from scratch and exported from Blender in GLB format. It is attached to the camera, so that when the user moves, then the stick moves as well appropriately. When the user clicks with the left mouse button:

1. a tween animation starts, in which the stick rotates so to simulate the action of a person to shake it;
2. a Raycaster is created and activated. It is a class mainly used for mouse picking (working out what objects in the 3D space the mouse is over) amongst other things. The raycaster simply provide a ray casted from an origin towards a direction and check if the ray hits something: if so, an array of sorted objects (distance metric) is returned and actions can be taken.
   In our case, it starts from the center of the camera and it goes forward perpendicularly to the camera's surface plane, like a simulation of the player's gaze. If it intersects a fox, then the fox's escape animation takes place, if it hits a dog, the dog's anxious animation takes place, otherwise nothing happens;

3. a new listener for the click event is added only when the tween animation of the stick is over, so that the user cannot activate the stick again if its animation is not finished yet.

2.7 - Fire

In night mode, there are four fires close to the fence. Each fire is composed of two different objects: the wood and the flame. They are placed together to simulate a real fire. The wood is not animated, namely no one of its properties changes during the game. The flame is animated through tween.js and it consists of rotating it around the vertical axis and scaling it over the same vertical axis.

2.8 - Aim, Fence, Trees, Well and Windmill

The Aim is a 3D object created from scratch through Blender and exported in GLB format. Its material has been changed in the application in a MeshBasicMaterial so to avoid that its color changes based on the light, and its properties of *opacity* has been set to *0.2* and *transparent* to *true*.

The fence has been downloaded in obj format. In this format the files that describe the visual aspects of the polygons are stored in external .mtl files. More than one external MTL material file may be referenced from within the OBJ file. The .mtl file may contain one or more named material definitions. However in this case we have just one .mtl file.

The tree has been downloaded in obj format as well.

The windmill has been downloaded in GLTF format so that it has a hierarchical structure and in the application it is possible to rotate its pales.

The well is a 3D object created from scratch through Blender and exported in GLB format.

Notice that since the obj format doesn't offer a hierarchical structure to the programmer, i.e. the object is a unique mesh, it is highly difficult to perform animations with it. Then, only the objects that didn't require complex animations are in obj format. While the GLTF and GLB formats preserve the hierarchical structure of the objects so that a programmer can easily act on each sub-component of them independently of the others.

3 - Global Aspects

3.1 - Technical details

The scene requires some seconds to be ready, then there is an initial overlay that is semi transparent that lets the user feel that he is not in the game yet. I also used a loadingManager that helps to keep track of the loading phase. It offers some callbacks and in this case I used the "*onLoad*" callback in order to notify the user that the scene is ready. I do this by changing the content of the HTML file through the *document.getElementById(ID).innerHTML* property that allows the javascript application to fill the element of the HTML page with some text at runtime.

Moreover, the several listeners that are used in this application are set up in a consistent way, so if the user tries to move the player or to click the left mouse button during the loading phase, nothing happens.

When the scene is ready, the screen shows "CLICK TO PLAY!" so the user just clicks, the game starts and:

- The routine that periodically spawns the foxes is setted
- The dogs start to play their animations
- All the listeners for the user's interaction are listening
- The timer starts

The interaction of the player with the different objects has been already explained in the previous sections. Just notice that in order to stop the animations of a given fox when it is hitted, I added to the root node ”*_rootJoint*” of each fox the references to the tweens of the fox itself, so that I can have their references immediately when the raycaster returns the hitted object.

3.2 - Pause

The user can exit the game by pressing ESC two consecutive times if he wants. The first time he presses ESC it is shown an alert with written “CLICK TO PLAY! (or press ESC again to exit)”. This layout is thought just as a secure double check to avoid that accidentally the user exits the game by pressing ESC just one time, then it is not a pause of the game, in fact the foxes continue their animation toward the fence.

3.3 - Memory Leaks

During the deployment, I noticed that the resources of a tab of the browser were not freed completely when the page of this project was refreshed, i.e. the RAM usage increased just by refreshing the same project’s page multiple times. Then I used the “*dispose*” method on the resources (material, geometry, texture, mesh) that frees the memory and this problem was solved in this way.

4 - Levels

The game is presented in four levels. The table 1) shows the features of each level.

	EASY	**MEDIUM**	**HARD**	**IMPOSSIBLE**
Fear Distance	30 units	20 units	15 units	10 units
Fog	300 units	150 units	100 units	75 units
Frequency of foxes spawning	10000 ms	5000 ms	3000 ms	2500 ms
Stick animation time	600 ms	600 ms	400 ms	200 ms
Scenario	Day	Day	Day	Night
Aim	Yes	Yes	Yes	No

Table 1): Features for each level.

5 - Conclusions

The aim of the project was to make a browser game that meets some requirements. The project contains two hierarchical models representing foxes and dogs, each one with its peculiarities and its specific animations that included body motion and fluid arms/legs movements. Moreover, multiple instances of the same model are instantiated in the scene. Also the stick, the fire and the windmill have been animated. Of course, since they are so many, there are aspects that are not captured in this presentation, but that can be noticed by playing and looking at the code. I tried to make a general and complete picture of the work done on this project. It can of course be augmented with other types of animals, other types of interactions and effects, but still it reached the initial idea of creating, meanwhile satisfying all the requirements, a farm defender game.

References:

https://threejs.org/

https://threejsfundamentals.org/

https://github.com/tweenjs/tween.js/blob/master/docs/user_guide.md

https://get.webgl.org/