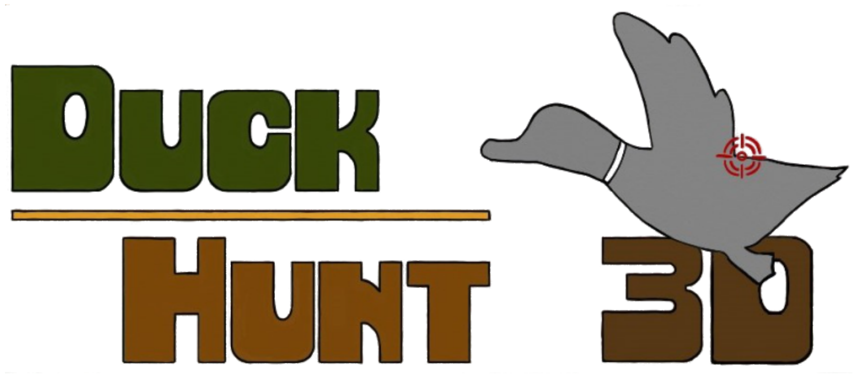
Interactive Graphics
Final Course Project

Authors:
Sveva Pepe - 1743997
Simone Tedeschi - 1762897
Claudia Medaglia - matricola
Christian Marinoni - 1745754



Professor: Marco Schaerf

June 2020

Contents

1 Introduction **2**

2 Game **2**

3 Scene **2**

3.1 Lights and Textures 3

4 3D Models **3**

4.1 Linear Models 3

4.2 Hierarchical Models 3

4.2.1 3D Duck Model 4

4.2.2 3D Dog Model 4

5 Animations **5**

5.1 Ducks animation 5

5.2 Dog animation 5

5.3 Clouds animation 6

6 User Interaction **6**

6.1 Sounds 6

7 Work organization **6**

7.1 GitKraken Boards 6

7.2 User tests 6

8 Conclusion **6**

1 Introduction

The goal of the project is to implement an interactive application that makes use of basic WebGL or advanced libraries, as in our case ThreeJS. The project cover the main aspects treated during the course, such as lights, textures, hierarchical models and animations.

2 Game

We decided to develop a 3D version of “Duck Hunt”, a famous game from the 80s, in which the objective is to hit as many ducks as possible. We designed the scene as a first-person game, where the player is located in a tall grass field and controls a rifle. Thanks to the help of his dog, that frighten the ducks hidden in the tall grass, the hunt can start. The game ends when the player miss five ducks. In addition, our game provides other functionalities like enable and disable sounds, pause the game or restart it.

The above described scene is depicted in the following figure:

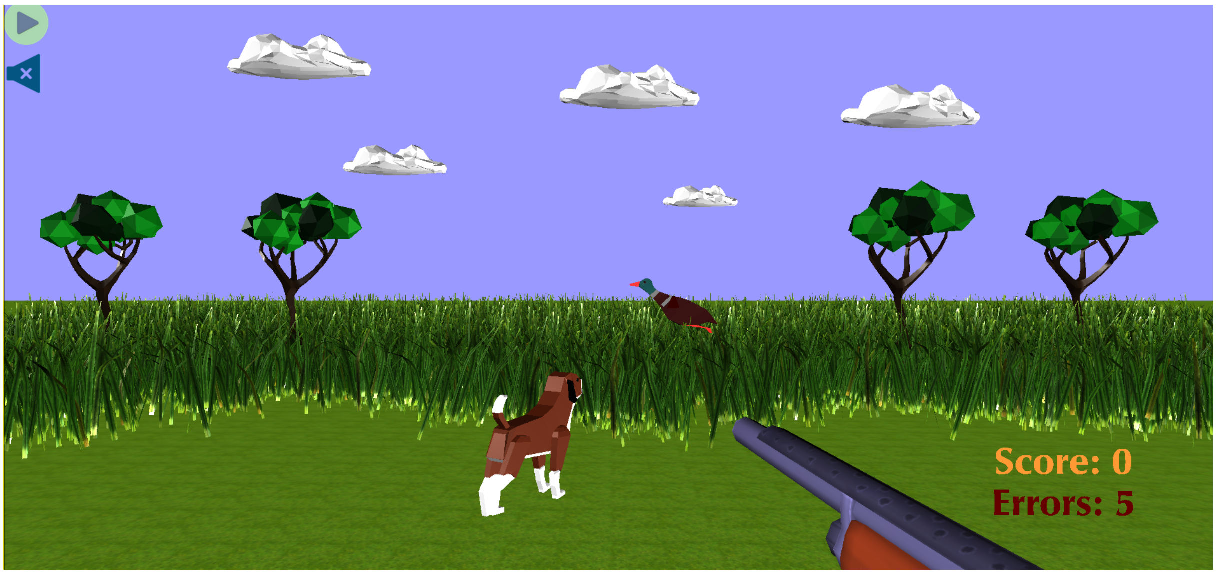


Figure 1: Starting scene of the game.

3 Scene

The scene shown in Figure 1, which is a ThreeJS object, has been obtained adding textures, lights, and different 3D models. The dog and the ducks are hand-made models while the other are taken from SketchFab. We used perspective camera to make the scene as realistic as possible to let the center of projection coincide with the user’s eyes. The prospective uses as parameters: *fovy, aspect, near and far*. *Fovy*, which stands for “field of view y-axis”, identifies how wide the eyes open along the y direction. *Aspect* represent the ratio between the width and height of the canvas. *Near* and *far* are any positive numbers representing the minimum and maximum distances of the object, with the restriction that near is always less than far. For the camera is defined also the *lookAt(x, y, z)* method, where the *x, y* and *z* are the coordinates of the scene. The texts on the bottom right corner instead, have beed modeled using TTFLoader of ThreeJS, where through a Mesh the relative colors have been applied. Moreover, in order to make our application responsive we add a dedicated Listener to adapt the window size based on the device resolution. Finally, to improve the user experience antialiasing has been used.

3.1 Lights and Textures

The ground texture was created by repeatedly applying a texture on a plane, using a TextureLoader. Then, the texture is added to the scene through the use of meshes that map texture coordinates into world coordinates.
A normal map has also been added to the *MeshStandardMaterial* associated with the floor, useful for creating more realistic lighting effects, as well as a displacement map, which modifies - albeit slightly - the position of the vertices of the mesh.
For the various models of the scene, the textures were imported with the model itself. For the sky, represented by a plane, a material was used which is associated with the desired blue colour.

Three directional lights have also been added to the scene. Two of them were placed on the left upper and bottom right corner respectively of the scene to reproduce a sunny day, otherwise using only one of them we obtain either dark clouds or dark objects. The third one has been introduced to illuminate texts because they are ahead of other elements and so the previous lights were not able to light up also them.

4 3D Models

In this section we explain the models we have included in our project. They are splitted into the following two categories:

- linear models: objects that are treated as an atomic entity;
- hierarchical models: objects composed by various sub-objects.

4.1 Linear Models

The linear models are models that are treated as singles entities. In our game are present the following four groups of linear models: Rifle, Trees, Clouds and Bushes. They contain 1, 4, 5 and 11 instances respectively that can be observed in Figure 1. We did this categorization to handle different kinds of objects in different ways, because objects belonging to different groups are indipendent to each other. Trees/bushes positions and orientations have been preset and remain static along the entire gameplay. Clouds positions instead, vary over time and rifle orientation can be controlled by the user. These last two aspects will be further explained in Sections 5 and 6 where we will provide technical details.

4.2 Hierarchical Models

Hierarchical models are models composed by different sub-objects that allow to represent relationships between such objects. The major benefit of hierarchical structures is the possibility to handle animations in a simple and efficient way, because, for instance, if we want to apply a rotation to the whole object we need only to perform it to the root of the object itself, instead of applying n rotations to each individual component.

4.2.1 3D Duck Model

The first hierarchical model that we introduced in our project is an hand-made 3D model of a duck, created with Blender. Its structure is divided into four components: left and right wings, torso (including also the head) and legs. Such structure allowed us to reproduce the desired behavior, which consists in a simultaneous diagonal translation and a synchronous upward rotation of the wings. The above described hiearchical structure is shown in the following figure:

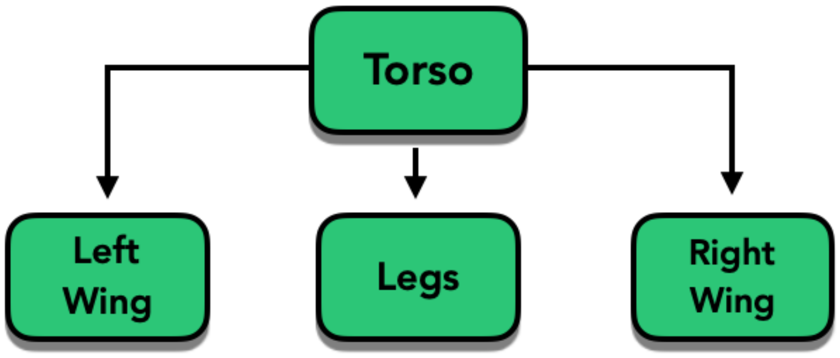


Figure 2: Hierarchical model of the duck.

4.2.2 3D Dog Model

The second hierarchical model that we used in our project is again an hand-made 3D model created with Blender, representing a dog. Its structure is divided into ten components: left and right upper front legs, left and right upper back legs, left and right lower front legs, left and right lower back legs, torso (including also the head) and tail.

The hierarchical structure of the dog is depicted below:

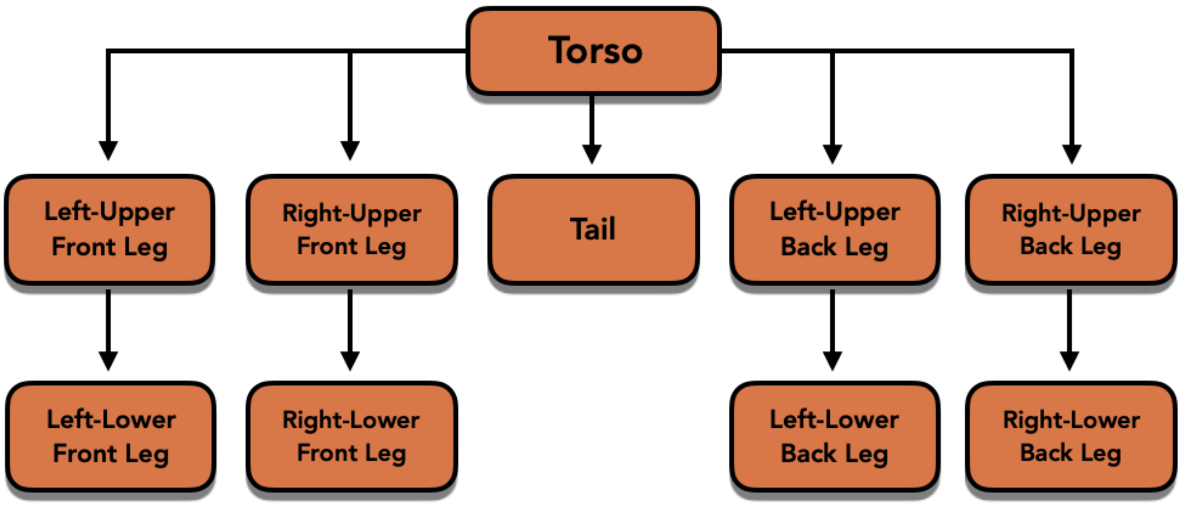


Figure 3: Hierarchical model of the dog.

We exploit such structure to let the dog move towards the bushes with few basic movements. To achieve this, we simply translate the "Torso" node, which is the root of the hierarchy tree, and automatically all other components, which are attached to it, will follow its movement. In the meanwhile, we alternate the legs movements back and forth to reproduce a walk. Additional details will be provided in the Animations section (5).

5 Animations

Following the project requirements, we have integrated four different animations into the project:

- one for the ducks (both for the movement in the sky and for the wings)
- one for the dog
- one for the clouds in the sky

For each animation, we have developed different solutions to ensure an effective, scalable (and not impossible to obtain) result.

5.1 Ducks animation

The translatory movement of the ducks in the sky was obtained by interpolating a keyframe sequence. To ensure low predictability of the starting point and flight path, we made the whole process random. The starting position is a point having fixed y and z components (semantically it is positioned on the straight line given by the intersection of the planes $y = -0.2$ and $z = 0$) and x component randomly extracted in the [-1.1) range. A second point is then selected, which differs by the y component and which is used to generate the trajectory. Finally, a keyframe sequence is automatically generated and saved in the i-th element of the *x_keyFramesDucks* and *y_keyFramesDucks* arrays.

These operations are carried out by calling, respectively, the *chooseStartingPoint*, *chooseDirection*, *generateKeyFrames* functions within the *animationBirds* function, which is responsible for updating the position of the ducks (and the rotation of the wings). The path of each duck is interrupted as soon as its position is outside the camera's field of view; to verify this, a frustum and the containsPoint method were used.

While translation is applied to the entire duck group, wing rotation is applied to each wing individually. At first, we had implemented a solution based on keyframes, however, we saw that the final result was not good at all. We preferred to use a gradual increase (decrease) within two defined values. Each wing is managed individually and its rotation values are reset as soon as the duck is "removed" from the game.
We also made a distinction between left and right wing, and between duck flying to the right and duck going to the left.

5.2 Dog animation

Also for the dog, the animation consists of three types of movement:

- the translation of the entire dog group into space

- the rotation of the upper part and lower part of the legs
- the tail wagging

Moreover, the animation consists of three phases:

- at the beginning of the first level, the dog enters the scene walking. Its presence is also functional since, as already mentioned, its barking causes the ducks to start flying
- the dog maintains his position throughout the game, wagging his tail in the meantime
- after the game over, the dog moves and leaves the scene

All these animations were achieved by setting keyframes manually.

5.3 Clouds animation

The clouds are divided into two groups according to their direction and their animations are differentiated based on the group they belong to. The clouds that have to go to the left will see the x component of their position decrease over time, the clouds that have to go to the right will see the x component increase.

This x component is increased (decreased) until it is lower than (greater than) a certain threshold, after which the position of the cloud is restored to a predefined value.
The threshold changes according to the position of the cloud: those with a lower z (i.e. placed further away from the camera) need to travel a greater distance before "exiting" the screen.

6 User Interaction

6.1 Sounds

7 Workflow

7.1 GitKraken Boards

7.2 User tests

8 Conclusion