SAPIENZA
UNIVERSITÀ DI ROMA

Soldier animation

Facoltà di Ingegneria
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Corso di laurea in Artificial Intelligence and Robotics

Emanuele Cutrona
Matricola 1755968

A.A. 2020-2021

ABSTRACT

This project is an Animation application: a scene with a hierarchically-built soldier. The soldier can be controlled by user-interface commands. By pressing one key button the

Three.js lib has been used in HTML and Javascript languages using Firefox as browser. Animation was performed through tween.js interpolating spline and Blender keyframes. Two lights and two textures were added to the scene.

Index

1 INTRODUCTION 4

2 PROBLEM 5

3 SOLUTION 6

4 IMPLEMENTS BEING USED 10

5 REFERENCES 11

1 INTRODUCTION

Soldier Model

The model is a standard humanoid and is taken from the Three.js library. It is a fairly complex hierarchical model with 112 nodes and 101 bones.
The hierarchical structure of the soldier is traversed for Shadowing (every body part has got its own castShadow and receiveShadow properties).

The soldier model uses many sub-components (such as leftLeg and leftUpLeg). Previewed animations are different blows of hits, depending on which button is pressed. Two differerent ways of performing animations were tried. One uses Blender keyframes, the other one uses Tween.js.

2 PROBLEM

The aim of this project was an animation of a soldier for a cartoon. Basic performed animations for the soldier were side kick, left push, right push, and right knee. Translations is imagined all over the platform.
Three.js was used to generate rotations for the single joints of the soldier. Tween.js was used to create the interpolating curves (Linear spline was used).
OrbitControls.js is useful to let the camera go around with the movement of the mouse while FBXLoader to import the model and keyframes.

3 SOLUTION

Scenario:

A class called game was created which contains all the basic functions for interaction and for traversing the hyerachical structure of the tree, together with the model loader (FBXLoader).

The combat arena was made by a mesh structure which has got geometry and material.
Then two different lights were added, Directional and Hemisphere.
Background light is grey (equal quantities of red, green and blue).
Grey fog is applied for objects between 1000 and 2000 from the active camera.
Hemisphere light with sky color equals black and ground colour equals dark grey (0x444444).
Directional light is pure white, casting shadows from -100 to 180 (bottom to top) and -120 to 120 (left to right).

A texture was added to the soldier and to the combat arena.

As it will be described in the next paragraph, the frames were chosen according to which were the key-frames for each of the blows; then I simulated the animation using an appropriate delay between two renderings, with the setTimeout function:

"**setTimeout (function (){...}, delay)**",

The interpolation engine of JS allows us to automatically construct the "missing" frames between two key-frames, simplifying the project.

On the other hand, the choice of distancing key-frames with a rendering delay cannot give the feeling of continuity except by using very small delay.

I decided to perform animation in two different ways: one exploits key-framing exporting one Blender keyframe per-action at a time. This is the way elbow animation is carried out. Whereas the lethal combination and side kick animations are performed exploiting Tween.js splines. This part is explained in the charter "Animation using Tween.js".

The Three.js class called Animation Mixer lends itself to managing the whole key-framing process. The animation mixer object is attached to the model at the moment of creation. It is then updated every time in the render function. The clipAction method prepares the action and the action.play plays it.

Whilst, Tween.js is perfectly suited to move the soldier provided the armature. Then given the performing time, the easing curve (Linear.None), the final position as rotation angles and translations coordinates, the selected bone moves together with every child in its hierarchical structure.

Technical solution

A pre-existing model was used in order to exploit the boned structure from three.js. In the first technique, I made joint rotations through Blender by putting the soldier in Pose Mode and moving the bones one at a time. In the second one, every joint rotation had to be

recorded. In order to do this, I exploited the Traverse function to save the bones that would be used for animation. Only twenty-one bones have been "manually" moved ('mixamorigNeck', 'mixamorigHead', 'mixamorigHead_end', exc).
Not only the name of the bones were saved but also the bones initial positions and rotations about x, y, and z.

Interaction

Five buttons were added:

- Back To Initial Position:
  Pushing this button brings the soldier back to stand-by position, from wherever it is.
- Elbow animation:
  It throws a right elbow on the cheek of its opponent.
- Sidekick:
  In this way, it throws a powerful right kick on the side of his opponent.
- Lethal combination:
  The soldier gives two pushes (left and right hand) and hits twice with its right knee.
- Move Model:
  This enables the model translation. It translates in depth and height.

Animation

Elbow animation is completed using an appropriate delay between two renderings, with the setTimeout function:

"setTimeout (function (){requestAnimationFrame(render);}, delay)",

Back To initial Position animation uses the traverse function to set every bone to the starting rotation angles. It makes use of the auxiliary function rotateBone which, given the name of the bone, time, angle, number of repetitions and chainTo, sets the bone to the passed rotation angle.

Sidekick is an auxiliary function which is called when the relative button is pressed in the HTML page. It uses seven tween curves and five bones directly moved.

Lethal combination uses thirteen bones and seventeen tweens.js. Yo-yo was used to make pushes and the right knee come back and to repeat the hit with the knee.

In the imported model, every bone was present twice, in one case in a single structure with no children, in the other one with the full family information. The bone itself was also present between its children field and as its own parent. To guarantee that moving a bone would also move its children, tweens were always called on the bone parent (which is, itself, but with the full family structure).

Every bone needed for the animation has been assigned to a module variable, so that it could be easily called from each part of the module. A bone can often be moved several times during an animation. These steps are different tweens, as intermediate positions are important for the realism of the animation.

Animation using TweenJs

TweenJs (https://github.com/tweenjs/tween.js/) is a Javascript library used to create smooth animations. It achieves this result by interpolating numerical values over time. For ThreeJs applications, this mainly means updating an object's position and rotations over time.

We will call any single object's animation created in this way a tween. All that a tween needs is the initial and final position of a variable, and the time in which the change must happen. At every iteration of the animation loop, Tween calculates the current state of the object for the given time-step, allowing the renderer to show it on screen. This will make the transition from initial to final state appear on screen as a smooth movement.

Additional parameters can be included for easier and smoother animation. In this project, several of these optional parameters were used:

- Easing: changes the easing function used for interpolation
- Delay: delays the starting moment of a tween. Useful to create longer animations where different parts move starting at different times
- Chain: chains a tween to another one. This means that the chained tween will start only when the first one ends
- Repeat: makes an animation repeat any amount of time
- Yoyo: it only has effect if used along with repeat. The tween will act in a yoyo-like manner, bouncing from start to end position, and back to start

This is a more demanding approach with respect to the first one, because it requires every single bone to be moved independently from each other. On the other hand, this technique guarantees more freedom, as every movement can be executed as desired.

Even though Blender was not used to export frames in this case, it was still used as an assistance tool to understand the required rotations that should be applied to the bones to reach the desired position. This allowed to use a differential approach when defining the tweens. The initial position was always defined as the current body position/rotation, and the final position was defined as the current position plus or minus a rotation angle.

During the animation process, two helper objects from Three.Js were used to make rotations easier to understand: classes SkeletonHelper and AxesHelper. The names are quite self-explanatory. The first one draws the skeleton of the model, which is all the bones as thin segments. The second one draws the global reference system. Figure 1 shows these helpers on screen (x-axis in red, y-axis in green, z-axis in blue).

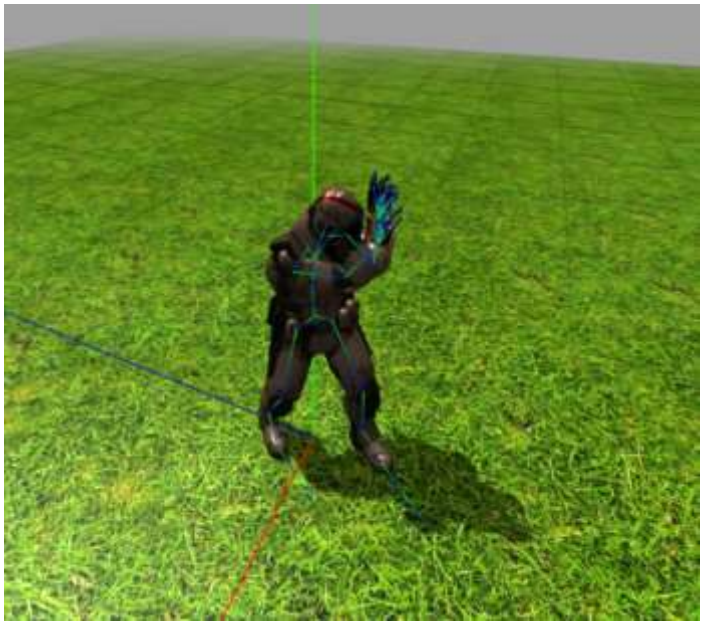

Figure 1: AxesHelper and SkeletonHelper

Texturing

The fighter model is one of the example models present in Three.Js. It was imported in Blender and exported back in the FBX format. During this process, the texture was lost from the model itself. To add back the texture, the traverse method was used to loop through every component of the model and to find the mesh components. The material of these parts was set as ThreeJs's MeshStandardMaterial, colour 0xCBB44D with skinning enabled. This material was mapped with the original texture image, extracted from the model.

Figure 2 shows the model without any material applied to the mesh, with the standard material and with the final texture.

Figure 3 is the images used for texture. On the left, the model texture and on the right the floor texture. The floor texture has been repeated 4 times in both directions to cover the whole area visible within the fog.

Figure 2: Model with no mateiral, with material, with texture

Figure 3: Model and Floor Texture

4 IMPLEMENTS BEING USED

Tool:

Blender

Libraries:

Three.js,

Tween.js

OrbitControls.js

Three.min.js

FBXLoader.js

Models:

webgl_animation_skinning_blending