Interactive Graphics project: KAKAMORA DANCE

Claudia Melis Tonti: 1888489

The project

This project is a try to emulate the Kakamora's dance by the Disney animation movie Moana (the clip video "Kakamora's dancing" can be easily found on Youtube).
The animation can be controlled by the user by starting and stopping it and changing the point of view.

Manual

Wait for the model to load, then to make the animation start push every button on the keyboard. To change the point of view, click and drag. To zoom in and out scroll with two fingers up and down on the trackpad.

Used Environments

The environments used are WebGL and Blender. The animation has been built using Javascript and then showed by html file. The model is done by hand with Blender 3D and then exported as .glb file. The .glb file was too big to upload on Github repository, so it has been compressed by the use of gltf-pipeline, then with the use of DRACO decoders the compressed file could be decoded for the three.js's gltf loader.

List of all the used libraries

The libraries used are MVnew.js implemented by the textbook and three.js.

The model has been done by hand on Blender and then imported with the gltf loader of three.js.

The threejs libraries have been imported not by the use of nodejs but directly by the threejs url (https://unpkg.com/three@0.120.1)

Description of all the technical aspects of the project

The initialize() function

The purpose of the initialize() function is to built everything that is necessary to load the scene.

The function builds a Three.Scene and adds to it 3 spotlights and a camera. Then, a renderer is defined such that the canvas (where the scene is displayed) will adapt to the browser's window dimensions.

Then the GLTF loader is defined. The gltf loader makes possible to WebGL to read a gltf/glb file. In our case the file is a glb file wich contains every information about the UVs maps of the objects, materials, and textures. In order to maintain these information the original file is too big to make it able to upload on a Github repository, so the file has been compressed with git-pipelines and then the compressed file is readed by a DRACOLoader of the three.js library. A problem of this model is that it has too much information because of the huge amounts of faces and details of the character. This means that the application uses a lot of GPU to make it work. We tried to solve the issue by not calling the requestAnimationFrame for each step, but this is not possible if we want an animation, so this solution has been removed.

In the loader every object's "castShadow" property is set to true. This makes the renderer detect the object shadows. Plus, the object called "Plane" has the property "receiveShadow" set true to project the main character's shadow on it. In our animation we can't see every shadow because we decided to not set every spotlight's "castShadows" true, because the animation would be very slow due to the renderer computations. These properties' setting is done by calling a traverse method on the scene.

To make the animation realistic a skeleton has been added to the model. In order to have a hierarchical structure every skeleton's bone is managed as a node. The structure of the nodes is built in the initNodes function and then a traverse function renders the position and the rotation of each node.

Because the load function of the model works in an asynchronous way it is managed by a Javascript promise in order to build the animation keyframes' array after the complete loading of the scene in the init() function. The load function can give a reject result in case of error and a resolve result in case of success.

Initially, the axesHelper and the SkeletonHelper of the model were visible to make it simple to build each rotation.

The animation

To make the animation a keyframes' array is initialized. Each element of the array contains a t_0 value, which represents the time when the movement should be done, an array of rotations, the torso_translate and the torso_pos. The torso_translate and the torso_pos are two Vector3s which indicates the position of the torso. There are two of it because one is used to manage the position of the root bone of the skeleton and one the coconut body of the character. This because in Blender 3D setting the coconut body as the parent of the root bone of the

skeleton spoiled the position of it, so the two objects are managed separately. Afterwards, this has been useful to manage the vibration of the coconut when the character hits it with hands.

Every keyframes' element is built by a function which task is to initialize a new array of Eulers (which represent the rotations) set to (0,0,0,'XYZ') (the last parameter indicates the order of the rotations on the axes), then every Euler of the previous rotations values is copied in the previously defined Eulers' array with the function copy(). The initialization of a new set of Eulers for each keyframes' element is necessary because if we reuse the same array to build new keyframes every array would overlap each other and the keyframe will be the same for the full animation. This happens because of the asynchrony of the loader. Despite this the memory should not be used completely thanks to the Javascript garbage collector.

Each keyframe is added to the array in the addAnimation function. Every rotation is updated considering the last array of rotation to simplify the update of each object's local frame.

In the function animationStep is managed the smoothness of the transition between two frames. This function takes two subsequent keyframes and computes every value between two different rotations and positions based on the current time value. So, given two angles theta0 and theta1 and two timesteps t0 and t1 and the current time t the new theta is computed in this way:

Theta = theta0 + (t-t0)/(t-t1)*(theta1 – theta0)

The same method is used to manage the position's changes of the torso.

The updateStep() function makes the animation restart from the beginning when the last timestep of the keyframes is reached if the transmission value is set to true.

User's interactions management

To make the user start the animation the event of clicking a random key is handled by document.body.onkeyup method. This event changes the transition's value.

Moreover, the point of view changing is managed by the OrbitControls by three.js, which is defined in the initialize function. Here the camera's damping factor, minimum and maximum distance and maximum polar angle's value are set.