INTERACTIVE GRAPHICS FINAL PROJECT

2061432 Branca Max
1850518 Marcelo John Mark Alexis

Galactic Football

Project Overview

As a final project for the Interactive Graphics class we made a multiplayer game in THREE.js built around the concept of playing football matches.

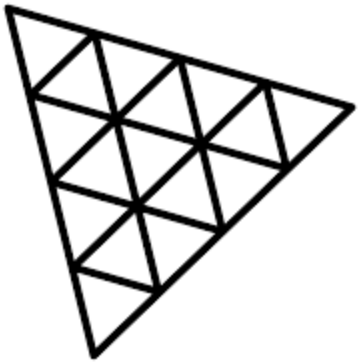
Introduction

The initial idea comes from an android game called Soccer Star, where the player can kick the ball by moving a little circle. In our project, we have implemented a similar idea, by using a human 3D model and a system that let the user kick the ball by deciding the trajectory and the velocity of the character by moving the mouse.

The game is composed of two squads and is intended as a 2-Players game. Each squad is composed of 3 characters, so there are 6 different characters in total. At every turn, the player can select which character to use, and then can kick the ball by moving the mouse in the desired direction and clicking. The player who first scores 2 goals wins. There are different animations, such as “standing”, “walking” and “exultation”. We have developed a total of 5 different animations.

The game was totally built using the Three.js framework. In the main menu, there is the possibility to access the game instructions, start a new game, or visualize all the different animations of the character.

THREE.JS



Three.js is a javascript library that allows you to simplify the creation of 3d environments compared to the use of the classic WebGL. A few elements are needed to create the environment:

- **The scene**: the container of all the objects that will be rendered;
- **The camera**: which will allow the user to view the scene in space;
- **Meshes**: physical container of the objects that will be instantiated;
- **Rendering engine**: which will allow the scene to be rendered at 60 frames per second.

Furthermore, present in the development folder of THREE.js, we used:

- The loader file for 3d objects in **GLTF** format;
- The orbit camera that allows you to better manage the focus of the camera with the mouse;
- The **FontLoader** and the TextGeometry, that allows the creation of 3D text in order to visualize the score, the goal advice and the win advice.

We also have used dat.Gui, a graphical user interface (GUI) library in Three.js that allows developers to create interactive control panels to manipulate various parameters and settings of their Three.js applications. This GUI provides a simple and intuitive way to modify and visualize properties of objects, materials, lights, and more in real-time.
Key features of dat.gui in Three.js include:

- User-Friendly Interface;
- Automatic Binding;
- Wide Range of Control Types;
- Customizability;

Lighting

The lighting setup consists of several light sources that contribute to the illumination of the scene:

• Ambient Light: an ambient light source (ambient) is added to the scene using the THREE.AmbientLight class. This light provides overall global illumination and sets the base level of brightness for the scene. It uses a white color with an intensity of 0.1.

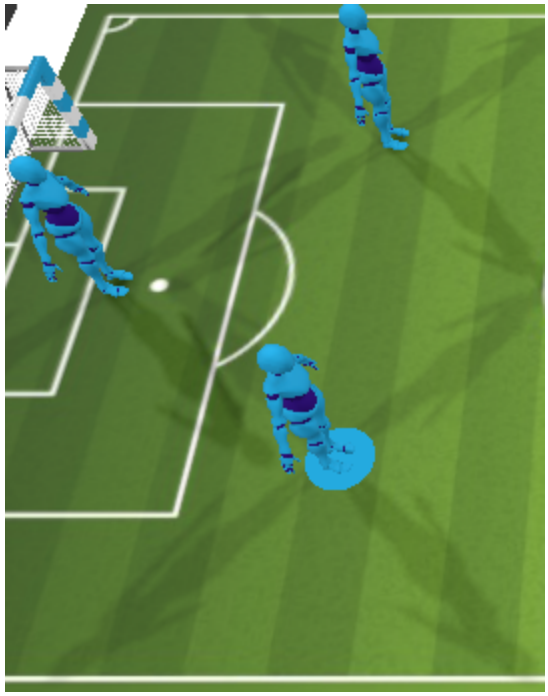
• Directional Lights: four directional lights are utilized to represent the lighting of the football pitch. Each directional light (light, light1, light2, light3) is created using the THREE.DirectionalLight class. These lights simulate the light sources positioned around the football field, emphasizing the players and their surroundings.

Shadows

Shadows are enabled in the scene to enhance the realism and depth of the game. Each directional light in the setup is configured to cast shadows by setting castShadow to true. Additionally, the shadows are defined using the following parameters:

• Shadow Camera: the shadow cameras for each directional light are set up to determine the range and perspective of the shadows.

• Shadow Bias: to prevent shadow artifacts, a shadow bias is set using the shadow.bias property. A small negative value of -0.0005 is applied to improve the accuracy of the shadow mapping.



Render Configuration

The renderer (renderer) is responsible for rendering the scene and displaying it on the screen. The following configurations are applied to the renderer:

• Renderer Size: the renderer is set to match the size of the window or viewport, utilizing the window.innerWidth and window.innerHeight properties.

• Shadow Mapping: shadows are enabled in the renderer using renderer.shadowMap.enabled = true. This allows the renderer to generate and display shadows in the scene.

• Shadow Map Type: the type of shadow mapping algorithm used is specified as THREE.PCFSoftShadowMap. This type provides higher-quality, softer shadows by employing Percentage Closer Filtering (PCF).

• Output Encoding: the renderer's output encoding is set to THREE.sRGBEncoding to ensure proper color representation and gamma correction in the rendered scene.

Galactic Football’s Models

• Human 3D Model: the human 3D model used in Galactic Football was obtained from the official Three.js website. This model was used for the football players and for the football fans. The hierarchical model comprises different mesh components, including the arms, legs, torso, neck, and head. The human model was imported and integrated into the game using the Three.js library. All the animations for the human 3D model were developed individually using JavaScript, by modifying the position and rotation of the arms, legs, torso, neck, and head.

• Football Pitch 3D Model: the 3D model of the football pitch was created using the Blender software. Blender offers powerful tools for modeling, texturing, and rendering 3D objects. This model serves as the backdrop for the gameplay, providing a realistic and immersive environment for the characters and the ball.

• Ball 3D Model: the 3D model of the ball used in Galactic Football was sourced from the internet.

User Interaction

Turns

The gameplay of the game is entirely turn based. During each turn, the player can decide which character to use (by pressing the Q or E keys, respectively to select clockwise or counterclockwise). Once the character is selected, the player draws the trajectory of movement. The length of the arrow, which indicates the trajectory of movement, also determines the speed of the ball once hit.

At the end of the shot, if the player scores a goal, the game displays a GOAL banner and the game resets, updating the score counter and returning all characters to their starting positions. The turn changes as soon as a player has finished moving their character. The game ends when a player reaches 2 goals.

Interactive user-robot lines

The provided code snippet is used to normalize the user's click on the canvas and obtain the precise position of the click on the pitch, in order to derive the correct end position of the arrow that describes the trajectory of the kick. Let's analyze the code step by step:

- const normalizedX = (endMouseX / window.innerWidth) * 2 - 1;
- const normalizedY = -(endMouseY / window.innerHeight) * 2 + 1;

In this part, the code takes the user's click coordinates (endMouseX and endMouseY) on the canvas and converts them to normalized device coordinates (NDC). The canvas is typically represented by the viewport, and NDC ranges from -1 to 1 in both the x and y axes.

- const cameraPosition = camera.position;
- const cameraDirection = vector.sub(cameraPosition).normalize();

The code retrieves the camera's position and direction. cameraPosition represents the position of the camera in the scene, while cameraDirection is calculated by subtracting cameraPosition from the vector and normalizing the result. The resulting cameraDirection represents a unit vector indicating the direction the camera is facing.

- const distance = -cameraPosition.y / cameraDirection.y;
- const clickPosition = cameraPosition.clone().add(cameraDirection.multiplyScalar(distance));

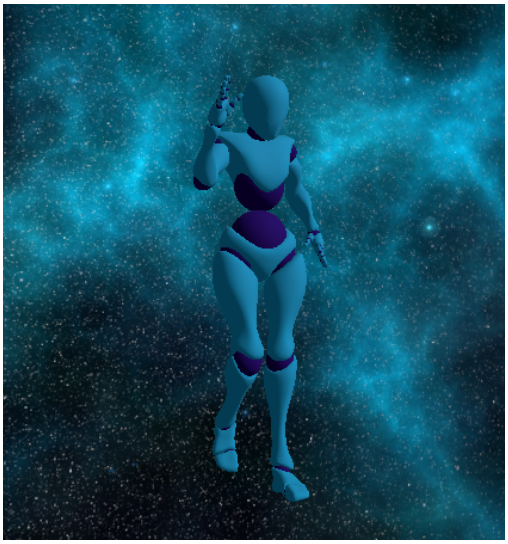
The code calculates the distance along the camera direction (cameraDirection) using a simple ratio calculation. This distance is then used to find the 3D position (clickPosition) by adding the camera position (cameraPosition) to the scaled camera direction.

- clickX = clickPosition.x;
- clickY = clickPosition.y;
- clickZ = clickPosition.z;

Finally, the code extracts the x, y, and z coordinates of the clickPosition vector, representing the precise position of the user's click on the pitch.

By performing these calculations, we can normalize the user's click on the canvas and determine the exact position on the pitch where the arrow describing the trajectory of the kick should end.

Animations



Running

The first step involves calculating the differences in the character's position coordinates (specifically, diff_x and diff_y) relative to a target position (clickX and clickZ). These differences are then used to compute the corresponding angle in radians using the Math.atan2 function. By assigning this calculated angle to the character's rotation along the y-axis (object.rotation.y), the character is oriented towards the target direction.

Next, the function proceeds to adjust the rotations of the character's arms and forearms (left_arm, right_arm, left_fore_arm, and right_fore_arm) to create a running animation. These rotations are defined using predetermined angles in radians.

The animation continues with the arms' movement: the left_arm.rotation.x and right_arm.rotation.x properties are incremented or decremented based on the value of the downarms variable. This produces an up-and-down motion of the arms.

Similarly, the animation handles the movement of the character's legs. The rotation angles of the left_up_leg, left_leg, right_up_leg, and right_leg objects are adjusted based on the value of the downlegs variable, resulting in a forward-and-backward leg motion.

Finally, the function sets specific rotation angles for the character's neck (neck.rotation.x) and head (head.rotation.x), giving them fixed positions during the running animation.

By following these steps, the running function creates a realistic animation of a running character, providing an engaging visual experience.

Exultation

The function begins by calculating the differences in coordinates between the character's current position and the target position. These differences are used to determine the character's rotation along the y-axis, causing it to face the target direction.

Next, specific rotation angles are applied to the character's right and left forearms, arms, and arms' z-axis rotation. Additionally, the forearm rotations around the y-axis are set to zero.

The animation of the arms involves an alternating movement. The downarms_exult variable determines whether the arms should move down or up.

The rotation angles of the arms are adjusted accordingly, creating the desired motion.

For the leg animation, the rotation of the spine is modified within a specific range. The variable spinelegs determine whether the spine should rotate positively or negatively along the z-axis.

Similar to the arm animation, the legs' motion involves alternating movements. The downlegs_exult variable controls whether the legs should move down or up. The rotation angles of the legs are adjusted accordingly, achieving the intended animation.

Overall, the exultation function generates a dynamic and celebratory animation by adjusting the rotation angles of various body parts based on predefined conditions.

Standing

The standing function animates a character in a standing position. Here's a concise description of its functionality:

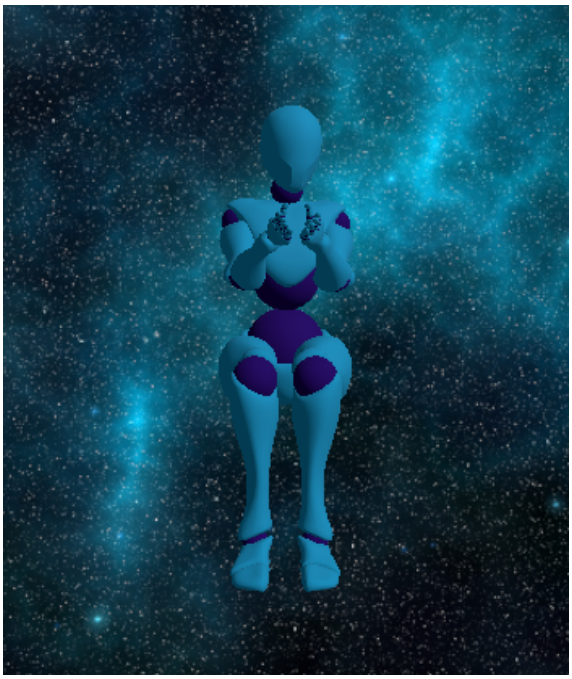
The function begins by calculating the differences in coordinates between the character's position (object.position) and the position of a ball (ball.position). These differences are used to determine the character's rotation along the y-axis (object.rotation.y) using the Math.atan2 function.

Next, the function adjusts the rotation angles of various body parts to create a standing pose. The left_arm, right_arm, left_fore_arm, and right_fore_arm objects have their rotation angles along the z-axis (rotation.z) set to predefined values (Math.PI * -0.4 and Math.PI * 0.4), while the left_fore_arm and right_fore_arm objects have their rotation angles along the y-axis (rotation.y) set to -Math.PI * 0.1 and Math.PI * 0.1 respectively.

The animation involves alternating movements of the arms and legs. The rotation angles of the arms (left_arm.rotation.x and right_arm.rotation.x) are incremented or decremented based on the downarms_stand variable. Similarly, the rotation angles of the legs (left_up_leg.rotation.x, left_leg.rotation.x, right_up_leg.rotation.x, and right_leg.rotation.x) are adjusted based on the downlegs_stand variable.

Overall, the standing function generates a static animation where the character stands with alternating movements of the arms and legs, creating a sense of liveliness.

Public Animation



The public animation is using quaternions, specifically THREE.QuaternionKeyframeTrack. Quaternions are mathematical entities that represent rotations in three-dimensional space, offering a more stable and efficient way to interpolate and animate rotations compared to other methods.

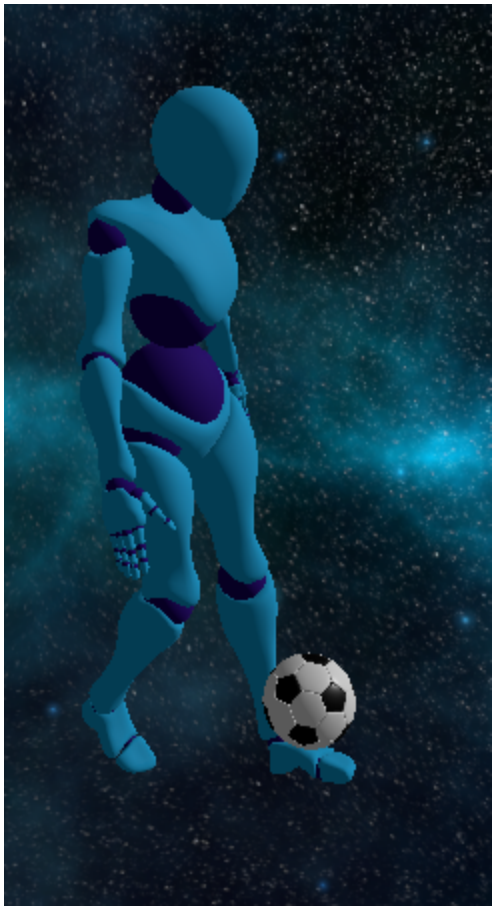
In the fanAnimation clip, each keyframe represents a specific point in time, and the associated quaternion values define the rotation of the corresponding bone at that time.

By smoothly interpolating between these keyframes, the animation creates fluid and realistic movements.

Quaternions ensure that rotations are performed correctly, avoiding common issues such as gimbal lock and providing smoother transitions between poses.

This allows for more precise control over the character's movements and helps achieve a high-quality animation of the seated audience applauding.

Juggling animation



The "palleggio" function implements an animation of a character performing juggling movements. This animation is achieved by manipulating various bones in the character's rig using quaternions.

In summary, the function performs the following steps:

Calculation of the rotation angle based on the position of an object relative to the character, followed by setting the rotation of the character's main object accordingly. Rotation of the neck and head bones to create a natural posture.

Slight rotation of the left and right forearm bones to simulate arm movements during juggling.

Control of the ball's motion, including rotation along the x-axis and a bouncing effect achieved by adjusting its vertical position.

Handling of leg movements during juggling, with rotation of the leg bones to simulate a walking motion while juggling.

By combining these bone rotations and object movements, the "palleggio" function animates the character, creating a convincing juggling motion. The use of quaternions enables smooth and realistic rotations of the character's bones, resulting in a dynamic and engaging animation.

Ball Movement

We have implemented the movement of the ball as following:

Initialization and calculation of normalized ball velocities (normball and normball2) based on the ball's current velocities (ballvx and ballvy).

Iterative update of the ball's position using the calculated normalized velocities, multiplied by a factor of 1.5. This causes the ball to move smoothly in the x and z directions.

Incrementing the "times" variable to control the number of iterations.

Handling special cases: checking for NaN values and setting them to 0 if necessary.

Updating the ball's rotation based on the normalized velocities.

Updating the bounding box (box_ball) that surrounds the ball's position and orientation.

Overall, we have used a series of calculations and iterations to update the ball's position and rotation until a certain condition (times < 8) is met, at which point the movement stops. The code provides flexibility for customization and fine-tuning of the ball's behavior based on the specific application requirements.

Conclusions

"Galactic Football" is a captivating game that utilizes Three.js animations to bring its virtual world to life. With dynamic character animations such as running, exultation, standing, and ball juggling, players can enjoy a visually stunning and immersive gaming experience. These animations, implemented using keyframe tracks and quaternion rotations, ensure smooth transitions and natural movements. By combining the excitement of football with futuristic elements, "Galactic Football" offers an engaging gameplay environment that showcases the power of Three.js animations.