ROBOT ESCAPE

INTERACTIVE GRAPHICS - FINAL PROJECT REPORT

CAMILLA IORIO 1852512 - SEPTEMBER 25, 2023

SUMMARY

THE IDEA

In this report, we will delve into the video game **"ROBOT ESCAPE"**, an endless runner that places a robot at its center, whose survival depends on the ability to navigate obstacles.

"ROBOT ESCAPE" provides a dynamic gaming experience, requiring quick reflexes to guide the robot through a changing and hostile environment. The goal is clear: avoid obstacles to keep the robot moving and out of harm's way.
This report will thoroughly examine the game's mechanics, encompassing various facets of interaction such as character animations and audio experiences.

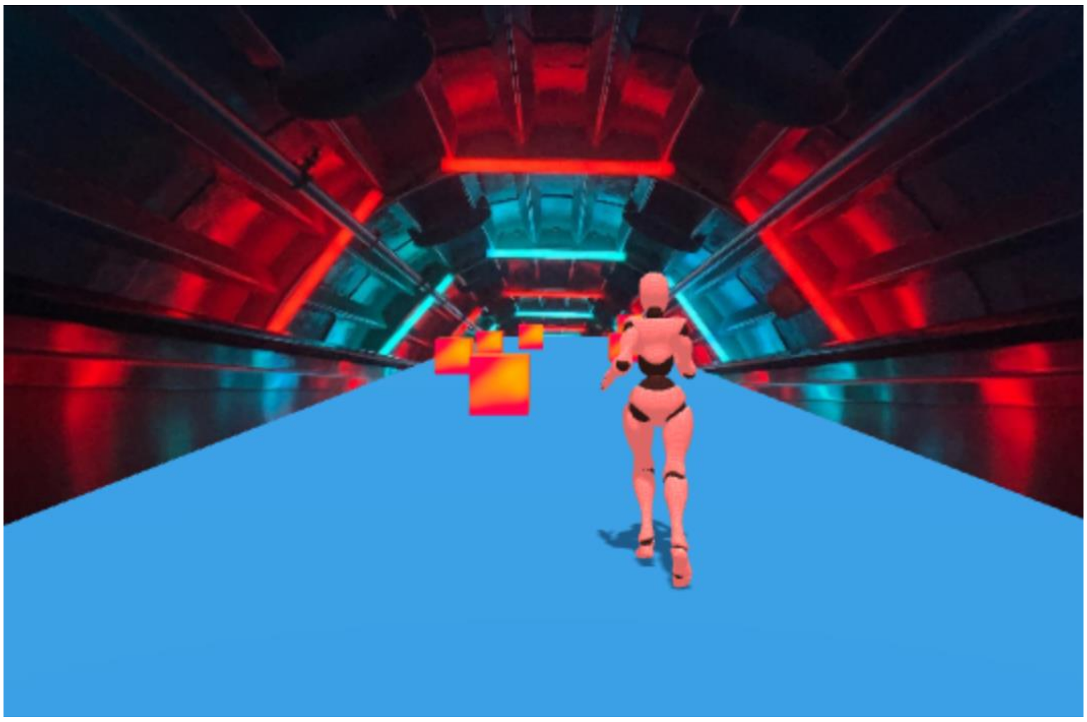
ENVIRONMENT

The codebase has been structured using **HTML, CSS**, and **JavaScript** to establish a solid foundation for the project. These programming languages collectively facilitate the creation of a dynamic and interactive gaming experience.

In order to enhance the visual aesthetics and three-dimensional rendering of the game environment, the implementation of **Three.js** has been employed. This library simplifies the integration of 3D graphics into web applications, contributing to an elevated level of immersion and realism within the game.

Additionally, for the smooth execution of animations that bring characters to life, the inclusion of **tween.js** has been essential. This library plays a crucial role in orchestrating seamless transitions and movements throughout gameplay, ensuring a captivating and fluid user experience.

In addition, the **GUI** library has been strategically employed to manage various aspects within the animation showcase screen. This encompasses the precise control of the spotlight's position and the seamless navigation through animations, enhancing the overall accessibility and user engagement within this specific interface.



IMPLEMENTATION

GAME STRUCTURE

The core structure of the game model is constructed using the Box class, which has been extended from *THREE.Mesh*. This class introduces specific functionalities for managing cubic objects, including parameters such as width, height, depth, color, position, velocity, and acceleration along the z-axis.

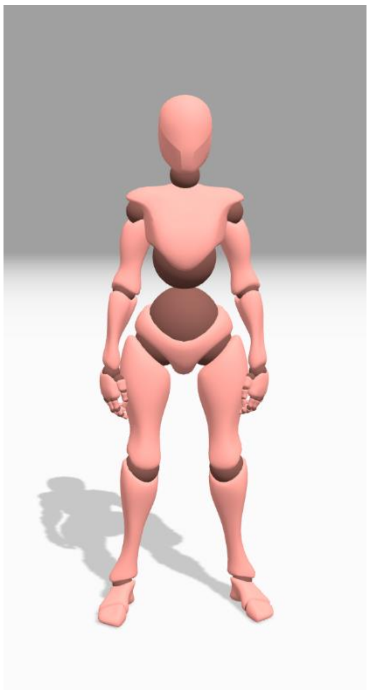
Within the class, three methods have been implemented:

1. Method **update_sides**:
   This method is employed to update the cube's sides based on its current position. It updates the values of *dx, sx, ground, top, front, and back*, representing the cube's sides.
2. Method **update**:
   This method is invoked to update the position and physics of the cube in every frame.
   It calls *update_sides()* to ensure that the cube's side values are up to date. If *zAcceleration* is true, it increases the speed along the z-axis. The cube's position along the x and z axes is updated based on its velocity. It calls the *gravity* method to handle gravity.
3. Method **gravity**:
   This method manages gravity's effect on the cube.
   It applies a gravitational force along the y-axis by decreasing the y-component of the velocity.
   It checks for collisions between the cube and an object named *base* using the ***crash()*** function. If a collision occurs, it reverses the cube's y-velocity and reduces it by 50%, simulating a bouncing effect. If no collision occurs, it updates the cube's position along the y-axis based on its y-velocity.

Following the same approach used for the construction of the game platform, the Box class was further extended to create the **Cube** and **Enemy classes**. The Cube class serves as the structural model to delineate the boundaries within which the robot operates within the environment. Conversely, the Enemy class is utilized to construct the elements that the robot must evade in order to ensure its survival.

ROBOT MODEL AND ANIMATIONS

The robot model was sourced from the official Three.js website. Within the game, the robot possesses the ability to move both left and right using the A and D keys, respectively. Additionally, the robot can execute a jump action by pressing the spacebar.



Two animations have been integrated into the robot's functionality, facilitated by the tween.js library within the functions ***running(model)*** and ***jump(model).*** Throughout the game, the robot is in a perpetual state of running, with the jump animation activated solely upon pressing the spacebar.

From the game's main menu, players have the option to initiate gameplay or enter a different environment, allowing for a detailed examination of the animations. Users can also choose between the two available animations for display.

TEXTURE, LIGHTS AND AUDIO

The utilization of the *THREE.TextureLoader()* function has facilitated the integration of textures into the game scene. A captivating ambiance has been established by incorporating a tunnel-like background image, imbuing the environment with a sense of depth. To further enhance the gameplay experience, textures have been applied to the obstacles, making them visually distinctive. In contrast, a uniform color scheme has been chosen for the game platform, ensuring a clear distinction from other game elements.

To illuminate the scene effectively, two types of lighting sources have been introduced. An Ambient light with a cool light serves to provide overall illumination, while a bright white Directional light has been employed, casting dynamic shadows and lending realism to the environment.

For the animation showcase screen, a Spotlight has been strategically positioned, creating a focused illumination that draws attention to the featured animations.

To immerse players in the game's atmosphere, background audio tracks have been seamlessly integrated both during gameplay and within the animation showcase screen. These audio tracks are designed to loop continuously, ensuring an uninterrupted and engaging auditory experience for players.

GAME MECHANISM

Within the ***animate()*** function, a dynamic game mechanic has been implemented to manage the appearance of enemies in the game world. This mechanic operates as follows:

1. At each frame of the game (controlled by the variable *frames*), the code evaluates whether it's time to spawn a new enemy.
2. To control the frequency of enemy spawns, a conditional statement checks if *frames* is divisible by *spawnRate* with no remainder. When this condition is met, a new enemy is spawned into the scene.
3. To add variability to the game, the *spawnRate* value is continuously adjusted. Initially, it starts at a higher value, but as the game progresses, it decreases, making enemy spawns more frequent. This introduces a challenging element as the game advances.
4. The newly created enemy is positioned randomly along the x-axis (horizontally) within a specified range and is placed at the starting y and z positions. This randomness adds unpredictability to enemy appearances, enhancing the game's difficulty and replay value.
5. The spawned enemy is then added to the game scene and included in the *enemies* array for further game logic.

In addition to managing enemy spawns, the *animate()* function also handles game-over conditions. These conditions are monitored continuously throughout gameplay, and when specific criteria are met (when the robot collides with an obstacle), the game transitions to the game-over state.

FUTURE IMPLEMENTATIONS

In consideration of future project enhancements beyond the current delivery, several compelling ideas have been envisaged to enrich the gaming experience:

1. **Score Tracking System**: The implementation of a scoring system is on the horizon, where players will earn points based on their survival duration within the game. This addition not only introduces an element of competition among players but also offers a tangible reward for skillful gameplay.
2. **Multiple Difficulty Levels**: To cater to a wider range of players, the introduction of multiple difficulty levels is in the pipeline. This will allow both newcomers and seasoned gamers to enjoy the game at their preferred level of challenge, ensuring accessibility and engagement for all.
3. **Diverse Enemy Types**: The game's antagonist lineup is set to expand. Expect the introduction of diverse enemy types, each with unique characteristics and behaviors. For example, enemies that hover at higher altitudes will demand nimble maneuvers from players, requiring the robot to dip or dive to evade them. This variety will add depth to gameplay and keep players on their toes.
4. **Varied Environments**: Offering players the freedom to select from a variety of distinct environments is in the works. This feature will not only enhance visual diversity but also introduce different challenges and obstacles, ensuring a fresh and engaging experience with each choice.
5. **Customization Options**: To empower players to personalize their gaming experience, the animation showcase screen will transform into a customization hub. Here, players can choose from an array of characters and environments, tailoring the game to their preferences. This personalization feature adds a layer of immersion and ownership to the gameplay.
6. **Jump Limitation**: A prospective feature involves implementing restrictions on the number of jumps a player can execute. This addition is intended to bring a level of control to the gameplay experience, preventing users from executing an unlimited number of jumps. By introducing this limitation, the game aims to strike a balance between player convenience and maintaining a fair level of challenge, ultimately enhancing the overall gameplay experience.

These future implementations are poised to elevate the game to new heights, offering a richer and more engaging experience for players while fostering greater versatility and customization.