Immagine che contiene testo, Carattere, logo, Elementi grafici

Descrizione generata automaticamente

Crash Bandicoot Three JS

Final Project

Alessandro Nicolella-Michelangelo Tronti

Interactive Graphics

June 2023

Introduction

The game wants to give to the old-style gamers a throwback into the 90s, when the Crash Bandicoot era began. It is all WebGL based, for which we’ve used:

* **JavaScript:** to build the logic of the game, with the various object spawns and all the mechanics of the game.
* **HTML+CSS:** for the front-end of the application and menu management.
* **ThreeJS library:** it simplified us the writing process of WebGL applications.
* **TweenJS:** to implement clearer animations for the run of the 3D model.

# Game Concept

# The main source of inspiration for this project is the 1996 game **Crash Bandicoot** by Naughty Dog for the PlayStation console, one of the most iconic and memorable platform games of the early 3d era. Our project’s name is a word pun of the second sequel of the original 90s trilogy, Crash Bandicoot 3 Warped, with our most used library in the project, ThreeJS. Our level starts in a forest near to a pirate bay, which is the level end, where a ship got destroyed and all the stuff on in got tossed all around. The goal of the game is to arrive at the end of the level without lose all the lives, avoiding obstacles, collecting fruits and break special or normal boxes.



Figure 1: Gameplay

**Game Mechanics**

The user controls *Crash Bandicoot* with the keyboard. He can move the character forward, backward, to the left and to the right with the w, s, a and d key respectively and can jump by pressing space. The goal of the demo is to reach the end of the level, avoiding the spikes and the rolling rock that can kill *Crash* and collecting all the *Wumpa* fruits that spawn on the road or inside the boxes. When the player collects more than 15 fruits, he wins a life. Crash can only move on the course, which can be viewed as a big corridor. The player can also destroy the boxes that lay on the map by jumping on the top of them. Inside these boxes, the player can find some more *Wumpa* fruits, or a *Aku Aku* mask. This mask can let the player take an other hit without losing a life. If Crash touches the spikes or the rocks he loses a life. When the player life count reaches zero, the game is over and he his presented with a screen that shows him his score. If the player reaches the ending level platform, he has beaten the game and can play it again by re-loading the page.

# Development

The development of the project is divided into different parts, so that in each one we can focus on different aspects of the game.

## Playable character: Crash Bandicoot

Crash Bandicoot is the main character that the player can interact with. The 3D model is imported from SketchFab and is made of many parts and bones, that can be manipulated through JavaScript code and ThreeJS, loaded in an array called playerBones. There are almost 150 bones, including fingers and hair of the model, but we used only the parts that moves the model to simulate a run animation effect.

We implemented a simple running cycle using the Tween.js library to interpolate the rotation angles for the legs and the arms. In order to achieve a sense of realism, the animation is repeated endlessly while the user continues to press the buttons to run, but, thanks to the function *yoyo*  function *,* the animations of the various joints are mirrored while they return to the initial state.

The user can also jump, in order to break the boxes that contain the bonuses, or to avoid some incoming obstacles. When Crash jumps, the character extends his arms above his head and closes its legs.

The physics and the animation for the jump are tied to the framerate.

The character model is loaded into the scene in the *init* function and it is placed at the beginning of the course. Then, in the *animate*  function loop, the game logic updates the player position by checking the user input and then updates the camera location, which is always behind Crash and follows him.

The game logic then checks if the player is colliding with some other model. The function that detects the collisions, *checkCollision*, given two models, checks if their bounding boxes overlap. A bounding box is built as an instance of a *Box3* Three.js model, and its dimensions are the maximum volume occupied by a 3d model. In order to make the game more enjoyable, the size of the collision boxes have been tweaked. In particular, the boxes of the obstacles are slightly smaller, and the ones for the crates are much larger.

**Bonuses**

In this playable demo there is a small set of bonuses that the player can collect, which are the main ones that are featured in the original PlayStation games. The player, in particular, can collect *Wumpa* fruits, which can be located inside some boxes or can lay on the course, and the *Aku Aku* mask. The crates containing the fruits are simple looking boxes and store five fruits. The crates with the question mark, however, contains the mask bonus. When the player collects a bonus, a corresponding sound effect is played. There is an effect for the fruit, one for the mask, one for the winning of a new life and one for the breaking of the boxes. As stated before, if the player collects more then 15 fruits he gets one extra life, and, if he has collected the mask, he can take one extra hit without losing a life. The player can only break the boxes by jumping on them from the top. If the player walks towards a box without jumping on it, he will collide and it will obstruct his passage.

Fruits that lay on the course rotate around themselves with a simple animation.

All the models for the fruits, the mask and the two types of crates are freely accessible on Sketchfab.

During the *init* function all the bonuses are spawned on the map in random locations, and in the animation loop, after the collision checks, game logic is updated. When the player destroys a box, its model is removed from the scene.

**Obstacles**

Two types of obstacles are featured in this demo, spike traps and rolling rocks. Spike traps are static obstacles, while rolling rocks are dynamic and move on the x axis. Crash can avoid these obstacles by jumping or by passing near without touching them. The obstacle’s colliding boxes have been tweaked a little in order to make the game more forgiving and enjoyable. When the player touches an obstacle, the game logic puts him in jump mode and teleports him forward. During this process, Crash’s characteristic “woah” sound effect is played. While spike traps are fixed in place, rocks have a simple animation tied to the framerate that rotates them on the z axis and translates on the x axis back and forth.

The obstacles are spawned on the map in the *init* function, and in the animation loop collision are checked and the state of the rolling rocks is updated. By touching an obstacle Crash loses a life. When the player runs out of life, the game over screen it’s loaded. Both the obstacle models are freely available on sketckfab.

**Map**

In order to render the terrain and the map a few ways were explored. In an earlier revision of the project, we tried to implement the terrain as a simple plane geometry whose surface had been perturbated with a displacement map. This approach, although very simple to implement, lacked in terms of visual fidelity and of simplicity in interaction with the other parts of the code. In particular, since in displacement mapping all geometry data is computed during the rendering phase and is not stored in the application, implementing the collisions with the terrain was a very hard task. So, we opted to implement all the terrain as a single mesh and started searching for a fitting setting online. In the end, we selected this pirate-themed level with a a clear and delimited course that passes through it. Moreover, since all the course lies flat on a plane, we just let all the objects stand on that single plane. The terrain mesh was put inside of a Sky Box to give the impression to see an open landscape. The face of the box that sits in front of the player represents a view of the sea, to give the illusion that our player is located on an island in the ocean. To strengthen this illusion, a semi-transparent texture of the water has been added at the end of the level and there is also a simple sea sound effect that becomes more noticeable when the player is near the end of the level (since its volume linearly depends on the player position).

The lighting in the scene follows the standard Phong model implemented in Three.js. In particular, the scene is lit by a single directional light source located in (0,500,125) that points downwards.

All the logic for rendering the terrain and for the lighting model is implemented in the *init* function.

**Sounds**

The majority of the sound effects featured in the project are taken from the original Crah Bandicoot game. In particular the background music is the original’s intro theme, while the fruit, the mask, the box, the damage and the new life effects are exactly the same. The sea sound effect, instead, was sourced from a free sample library accessible on Youtube. In order to play all the necessary sounds, we implemented four different Three.js *audio players :*

* One for the music.
* One for the sea sound effect.
* Two for the gameplay elements.

In particular we used two different players for the gameplay elements because the player can perform more than one action that requires a specific sound at a time ( e.g. breaking a box and collecting a fruit or a mask).

**User Interface:**

**Conclusions:**