Interactive Graphics

Final Project

Antonio D'Orazio

Mat. 1967788

Compatibility

Tested with Google Chrome, Microsoft Edge (chromium).

Documentation used

- Three.js Fundamentals
- Intro to JavaScript 3D Physics using Ammo.js and Three.js (all the chapters)
    - I used the code in the tutorial as a reference to setup the Physics and to detect the collisions
- Official documentation and examples for Three.js, Ammo.js

Environment used

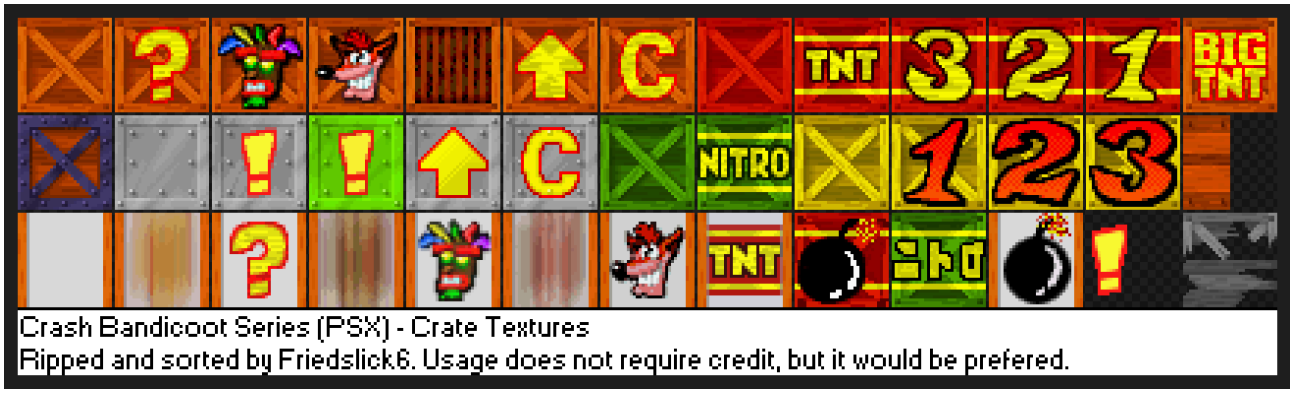
Libraries: Three.js, Tween.js, Ammo.js. Various plugins for Three.JS to achieve post processing effects, and to load GLTF/GLB Models. They are all contained in the lib folder.

Tools: I used Blender to rig the Crash model, and to prototype the animations to get the keyframes used in the Tween.js implementation of the animations. The keyframes are all stored in keyframes.js. This file is not automatically generated. Instead, it has the keyframes organized in a dictionary structure made by me by manually setting up the values obtained in the Blender animation prototype.

Models:

- Aku Aku
- Crash Bandicoot
- Wumpa Fruit
- Gem

Textures: they are all downloaded from the Web, from various sources. Some manual tweaks made with Gimp for the textures of the crates. Most of the textures come from the image below.



Crash Bandicoot Series (PSX) - Crate Textures
Ripped and sorted by Friedslick6. Usage does not require credit, but it would be prefered.

Introduction:

The project aims to recreate the gameplay and the look and feel of the Crash Bandicoot games for the Sony PlayStation from the 1990s.

The goal is to reach the warp platform at the end of the level without consuming all the extra lives. The player can spin or slide to break the different kinds of crates to get Wumpa fruits, a checkpoint, an Aku Aku mask, which protects Crash from damage.

The player must avoid touching the green Nitro crates because they are explosive.

The player can perform a slide-then-jump to take advantage of the impulse gained from the slide to jump further than a regular jump. If the player obtains 100 Wumpa fruits, he gains an extra life.

Technical Aspects:

Coding style: The main parts of the game interact through classes and modules, with some other helper function that does not belong to any class. Classes that do not use attributes provide static methods for simplicity. The classes are the following, with a brief description based on what their methods do.

- **GameManager**: to load the heaviest objects in the game (i.e., sounds, models, textures), and to show/hide the loading/game over/game win screen.
- **StatsUI:** to instantiate and update the UI of the game, showing the current number of collectables and lives.
- **PlayerController:** to instantiate the player, and to handle: the movements, the actions, Aku Aku, death and respawn.
- **Animator:** to initialize and control the player animations.
- **CrateManager:** to instantiate the crates, and to trigger the different actions when a crate is broken, given the different types.
- **Collectable:** it provides a simple collect method used by its child classes.
- **WumpaCollectable:** to instantiate, animate, and collect the Wumpa fruits.
- **AkuAkuCollectable:** to instantiate and collect Aku Aku.
- **GemCollectable:** to instantiate, animate, collect, and let fall down the final gem of the level.
- **CollisionManager:** to set up the callback for the collision and to check if there is a contact.

Camera: The original game provides two camera modes: 3d mode and side-scrolling mode. In the 3d mode, the camera follows the player towards the level, but the height, the rotation, and the position are fixed along a curve that passes through the level.

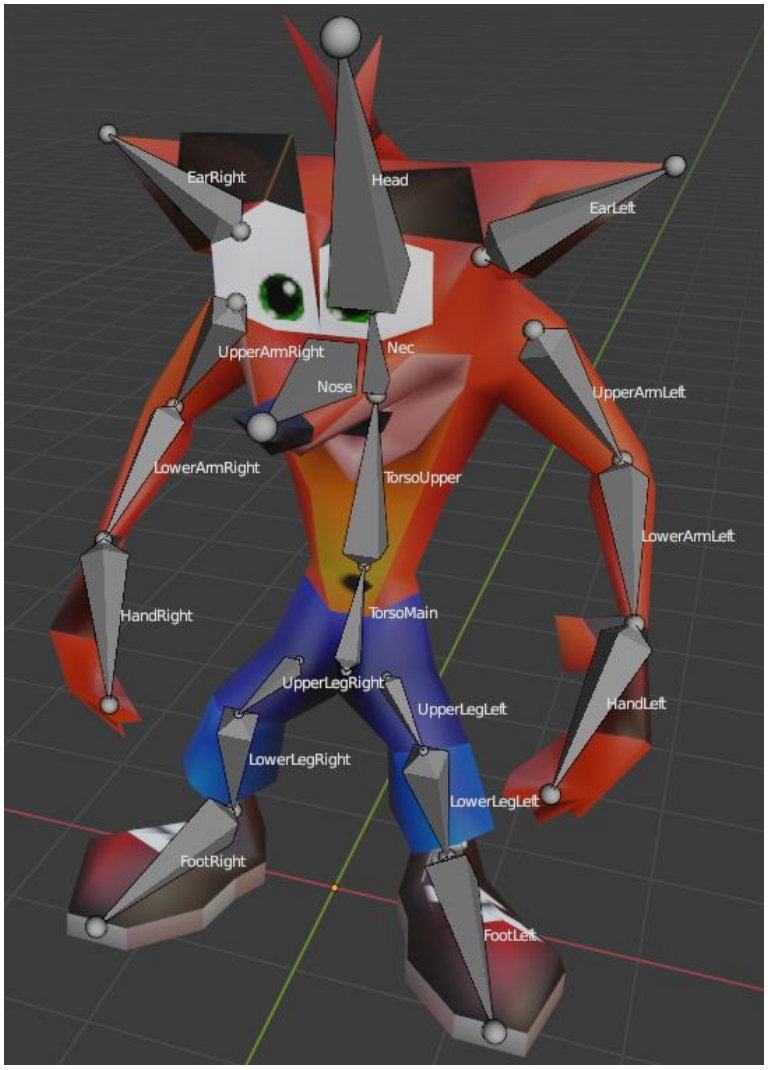
To obtain a similar effect in the project, I used the CatmullRomCurve3, with a set of key coordinates defined in level.js.

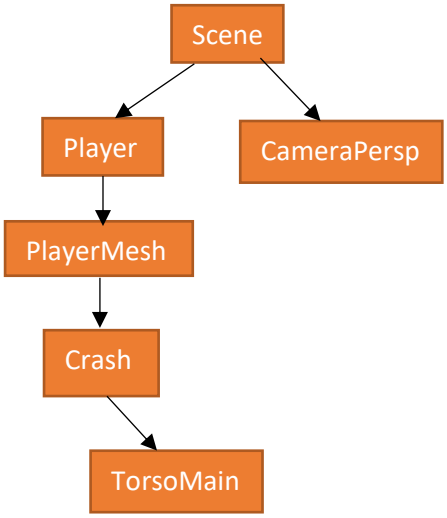
The camera position and rotation is updated at render time in the function updateCamera() in the following way:

- **3D Mode**: the y and the x of the camera follow the y and the x of the curve where the player position towards the z axis is set, as seen in the method getCurvePosAtPlayer. The rotation of the camera is a lookAt to the point of the camera, but 10 steps behind to achieve more smoothness.
- **2D Mode**: after a specific condition in the level is reached, (e.g., the player is 150 units up in the y axis), to pass in 2d mode the camera just follows the player position with some offset and without rotating.

Loading: After each model in the GameManager class is loaded, the attribute modelsLoaded is incremented by 1 and the function waitForLoading is called. Only when the counter has reached the total number of models, the function main will be called by the last call of waitForLoading. The other calls will be discarded.

Hierarchical Models: The following figures show the total set of bones for the Crash model, and the final hierarchy achieved in Three.js (other parts are not in the drawing below).





Player is the root of the character. Every force or movement is applied to it. The camera updates itself according to the Player node position. The Aku Aku mesh is attached to it.
PlayerMesh is the container of the mesh. It is used to orient the model while moving or spinning, without influencing the movements and the direction of the force applied.
Crash is the imported mesh.
TorsoMain is the first bone in the hierarchy. The others follow the structure in the image.

Physics: The physics is implemented with Ammo.js.

With Ammo, there is a physics world with its properties which is updated at each frame by performing a step. Each Three.js object is associated with an Ammo.js RigidBody. If the RigidBody is kinematic, it has a mass of zero.

Otherwise, it has a sensible mass. At each update, the computed values (transform, rotation) for each RigidBody are passed to the corresponding Three.js object's properties.

I used Ammo.js to implement movements, jump, collision detection, moving constraints.

Collision detection: The collision detection is an important part of the game because it lets the player collects stuff, break crates touch nitros and recognize the ground. The function setupContactResultCallback() setup the callback which will be executed by Ammo.js when a collision is detected.

The function CollisionManager.checkContact is called in the render loop with the player RigidBody as input.

In this function, there is a call to the function contactTest from Ammo.js, which detects collision between the player and other RigidBodies and, if there's any, it triggers the callback defined before.

Inside the callback, the applications check the tag of the object, and it triggers the consequential action.

As an example, let's see the result with a collision with a crate, the most complex interaction:

- The callback is called and the tag of the object is checked, which is "crate"
- If the crate is a Nitro, then:
    - If there is no Aku Aku to protect Crash, and if there is no invincibility the player dies
    - Otherwhise, Aku Aku dies
- If the crate is not a Nitro, and the player y is set higher than the crate (which means, he's jumping on the crate) -> Break the crate
    - If the player is not spinning nor sliding, give him an impulse towards the Y to simulate a bounce on the top of the crate.
- If the crate is not a Nitro, and the player is spinning -> Break the crate
- Otherwhise, ignore the collision.
-

Moving Constraints: I used the moving constraints on the crates, to make sure that they don't move away when pushed, but at the same time they can fall to their position while the level is instantiating.

I had to setup a custom Generic6DoFConstraint, to block every degree of freedom, both translational and rotational, except for the translation in the Y axis.

Post-processing: To let the game look like it is played on old CRT televisions, I wanted to implement some post-processing, by using many plugins from the ThreeJS official repository. The modules I used are EffectComposer, RenderPass, BloomPass, FilmPass, ShaderPass, RGBShiftShader.

The EffectComposer creates the chain of post-processing effects which will be rendered in the final scene, in the same instance order. The RenderPass is the first filter in the chain, to provide the rendered scene as input for the next post-processing steps. The other Pass implements different effects. For example, the FilmPass realizes a scanline effect which was typical with a CRT television. I used the ShaderPass to execute the RGBShiftShader, to obtain a little chromatic aberration.

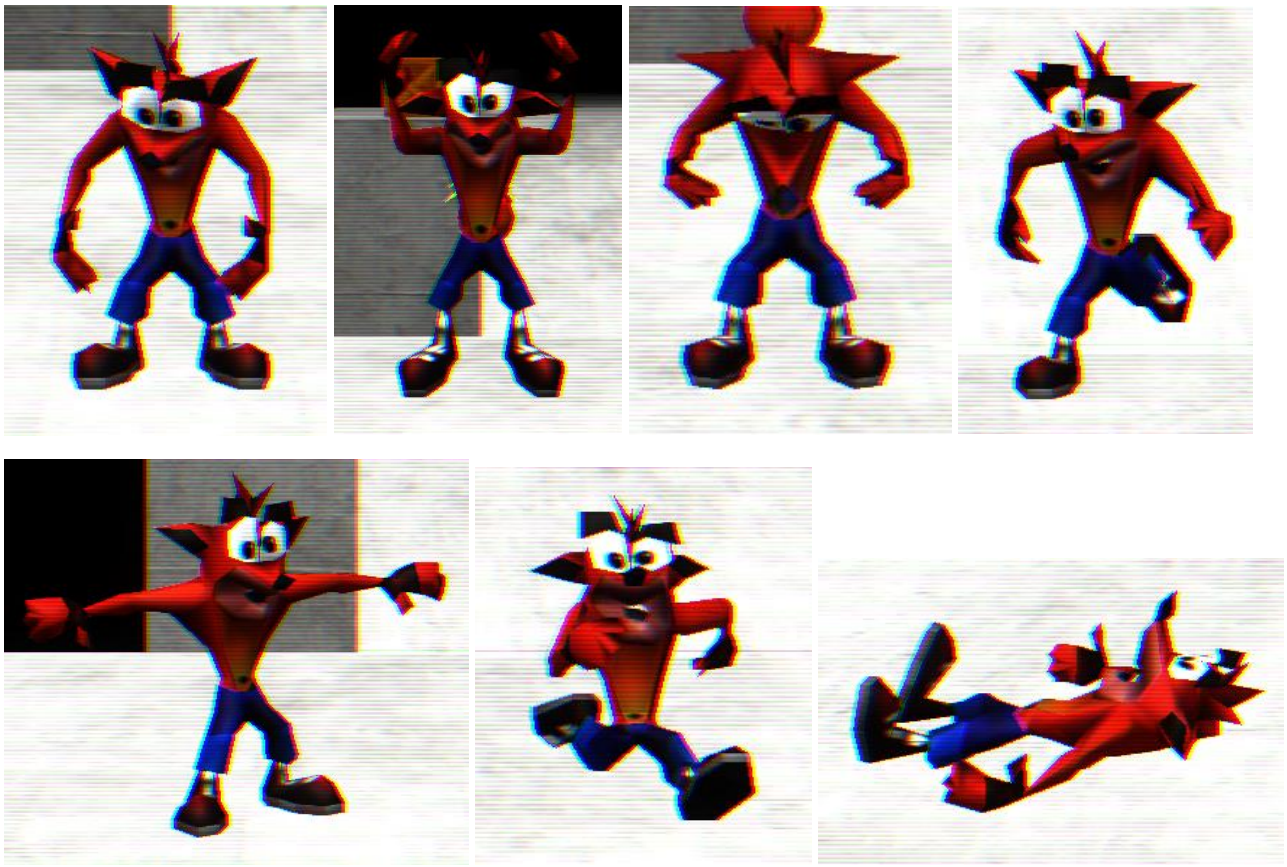
Lighting: I used a directional light and an ambient light as a global illumination for the environment.

The Nitro crates are all instantiated with a green point light inside of them, with the same being for the Gem with a red point light. The point lights cast the shadows, to give more atmosphere to the level, especially with the dark theme.

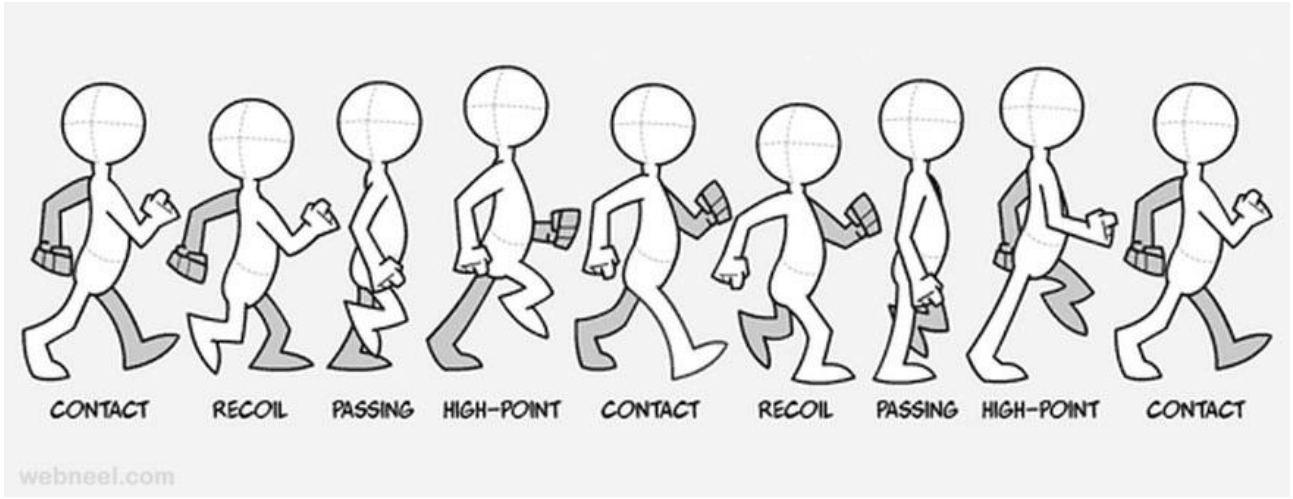
When a Nitro explodes, the light is animated with Tween.js to simulate a flash, by increasing its intensity.

Animations: The animations are all implemented using Tween.js. The Animator class provides all the animation for the player, while other animations for other objects are instantiated on the fly (e.g., the moving platform) or in their respective controllers.

The file keyframes.js contains all the keyframes for the player animations. The animations are *walk*, *jump*, *ground*, *spin*, *slide*, *death*, *idle*.



The most complex animation is *walk*. It follows the standard cycle of walking keyframes:



The keyframes are interpolated with Tween.js. Multiple keyframes are chained together, to compose the final animation, with the chain function. In case of a loop, the latest keyframe is chained with the first one.

The keyframes contain the position of the main torso, and each quaternion rotation for each bone. I used the quaternions to avoid the gimbal lock problem.

When the Animator class is initialized, the method *setBones* saves every default quaternion for each bone. In this way, during the interpolation the resulting quaternion is the product between the default quaternion, which represents the starting rotation, and the current frame quaternion, which represents a rotational offset. In this way, I used a single update function for each Tween for the player, which updates all the quaternions and the torso position, to ensure that after each animation the attributes of the bones will not collide.

Other animations are part of the gameplay:

- the falling cylinder uses Tween.js to fall slightly after when the player grounds on it
- the moving block uses Tween.js to move towards the y axis, to let the character reach the second part of the level.

Other simpler but interesting animations are:

- Aku Aku rotating and oscillating around Crash to protect him,
- The crate rotating in the HUD, the Wumpa fruit spinning and oscillating vertically, the transitions between elements in the HTML UI
- the Gem falling and bouncing to its position when all crates in the level are broken
- The Nitro crate emitting a huge intensity of light when touched

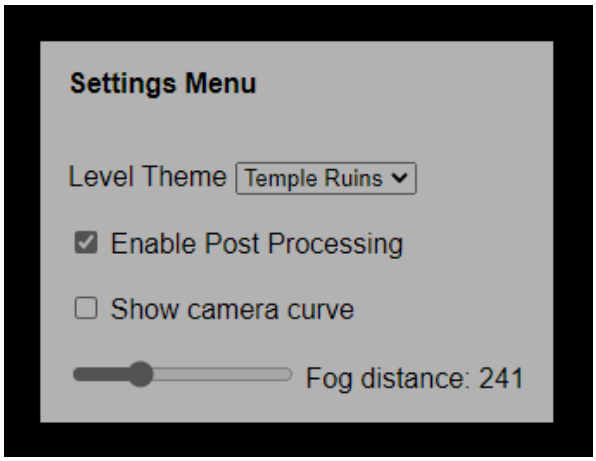
HUD: The HUD is always displayed on top. It shows the status in the game: the amount of Wumpa collected, the number of crates broken out of the total, and the remaining lives.

The HUD is implemented by an orthographic camera which has three objects attached to the spinning Wumpa model, the spinning crate, and the Crash sprite.

The counters are HTML DIV elements. Each time one of them needs to increment/decrement, the DIV innerHTML is updated.

The positioning of the div is done in JS by making a conversion between the Camera coordinates of the model/sprites and pixel coordinates plus an offset, to ensure the UI is responsive at multiple resolutions.

Implemented Interactions



Overlay Menu: There is the possibility to tweak some settings at runtime, thanks to the overlay menu. The menu is a fixed div with some controls, with their listeners assigned in app.js.

The settings that can be tweaked are:

- **Level Theme:** there are two themes inspired to the original levels: Temple Ruins and Snow Go. Changing the theme will result in a different song, and a different colour for the background and the fog. It also changes the background of the overlay menu.
- **Enable post processing:** to toggle the activation of post-processing effects. If disabled, the RenderPass is set as render to screen, while other Passes get disabled.
- **Show camera curve:** to show the curve that the camera follows during the 3D section of the level
- **Fog distance:** to set the distance of the fog.

Character Gameplay: The aim of the project was to recreate the original game mechanics of Crash Bandicoot's 1990s videogames. The actions are triggered by the JavaScript listeners, which wait for the keys to be pressed. The actions that the player can perform are:

- **WASD Keys: Move.** The character can move of a certain linear velocity, with the animation Walk.
- **Spacebar: Jump**. The character receives an impulse in direction of the Y axis. The animation Jump is triggered at the beginning, while the animation Ground is triggered when he falls to the ground
- **Key K: Spin.** The character performs a spin animation. While spinning, he can destroy the crates to collect the treasures.
- **WASD Keys + L: Slide.** The character receives an impulse towards its forward direction, and he performs the slide animation. While sliding, he can destroy the crates.
- **WASD Keys (keeping pressed) + L then Spacebar: Further jump.** The character exploits the acceleration obtained while sliding to jump further, to reach difficult sections of the level.