# Interactive Graphics

## Final Project



Antonio D'Orazio

## Mat. 1967788

## Compatibility

Tested with Google Chrome, Microsoft Edge (chromium), Brave Browser.

## Documentation used

* [Three.js Fundamentals](https://threejsfundamentals.org/)
* [Intro to JavaScript 3D Physics using Ammo.js and Three.js (all the chapters)](https://medium.com/@bluemagnificent/intro-to-javascript-3d-physics-using-ammo-js-and-three-js-dd48df81f591)
* 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](https://sketchfab.com/3d-models/aku-aku-61a6be99796242b6a28582657245c44a)
* [Crash Bandicoot](https://sketchfab.com/3d-models/crash-bandicoot-6a8340ca318d4b35a5c741415cae3c1e)
* [Wumpa Fruit](https://sketchfab.com/3d-models/wumpa-fruit-058451af54264f028c5c297e9d45eaa3)

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

Introduction:

The project aims to recreate the gameplay and the look and feel of the Crash Bandicoot games for the Sony PlayStation from 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 type of crates to get Wumpa fruits, a checkpoint, an Aku Aku mask, which protects Crash from a 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 which does not belong to any class. Classes which 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 theheaviestobjects 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
* **CollisionManager:** to setup 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 which passes through the level.

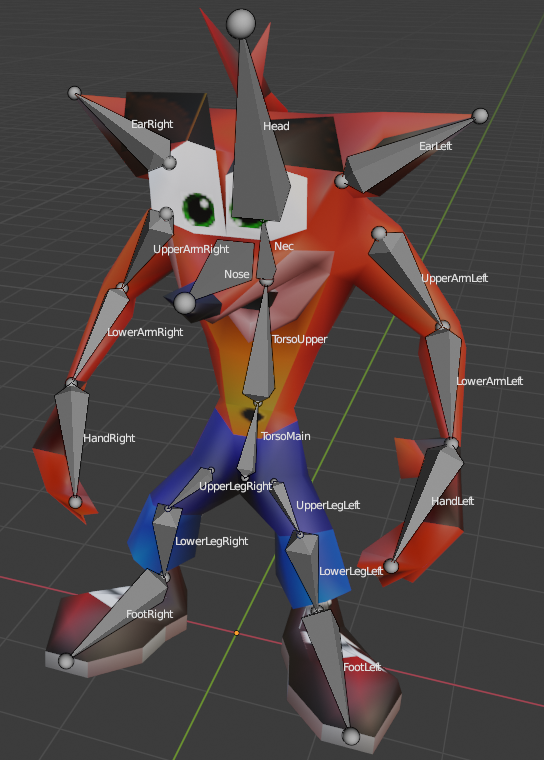
To obtain a similar effect in the project, I used the CatmullRomCurve3, with a set of key coordinates defined in level.js.

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 shows 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

Scene

CameraPersp

PlayerMesh

Crash

TorsoMain

*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.

*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 theimported 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 isupdated at each frame by performing a step. Each Three.js object is associated to 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.

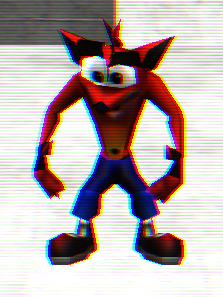
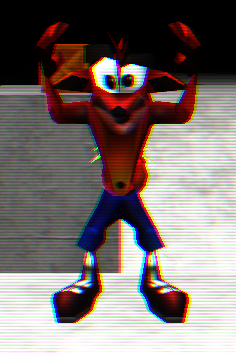
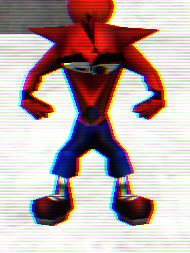
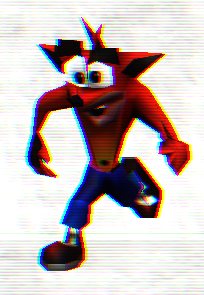
**Collision detection:**

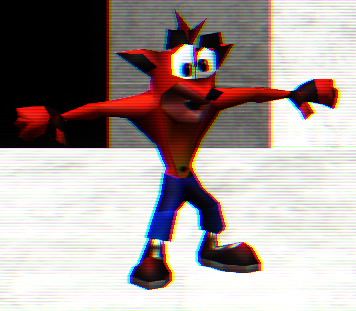
**Post processing and lights:** To let the game look like old CRTs, I wanted to implement some post processing, by using many plugins from the ThreeJS official repository. The modules I used are: EffectComposer, RenderPass, BloomPass, FilmPass, ShaderPass, RGBShiftShader.

The EffectComposer creates the chain of post-processing effects which will 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 of chromatic aberration.

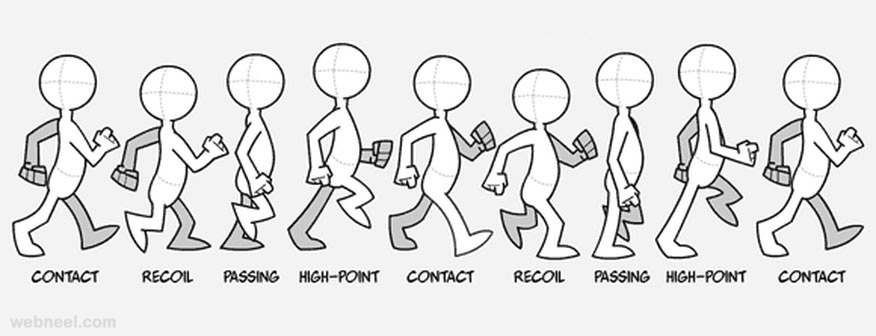
**Animations:** The animations are all implemented using Tween.js. The Animator class provides all the animation for the player, while other animations for other objects are instantiated on the fly (e.g., the moving platform) or in their respective controllers.

The file keyframes.js contains all the keyframes for the player animations. The animations in the file 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 quaternion obtain is the product between the default quaternion, which represents the starting rotation, and the current frame quaternion, which represents a rotation offset. In this way, I used a single update function for each Tween, 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, and the moving block uses Tween.js to move towards the y axis, to let the character reach the second part of the level.

Other 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.

**HUD:**

**Free mode:** The free mode disables the gravity of the application and the dynamic camera.

Theplayercan move vertically by pressing Space and Left Shift.

By pressing C, the player can place a crate of the type selected in the overlay menu. Note: due to the lighting, excessive placing of Nitro crates can slow down the application.

By pressing V, the player can place an environment part of the type selected in the overlay menu.

Implemented Interactions

**Main Menu**

**Overlay Menu //TODO Turn on-off the camera curve, turn on-off the post processing**

**Character Gameplay**