Loop3D: a lightweight 3D browser-based game engine

Diego Pérez Rueda

Final degree project
Bachelor's Degree in Video Game Design and Development
Universitat Jaume I

August 17, 2025

Project directed by: Miguel Chover Selles

To Jesús and Pilar

Acknowledgements

Firstly, I would like to thank my project supervisor, Miguel Chover Sellés, without whose expertise and guidance this project would never have come to fruition.

I would also like to thank my family and my good friends Héctor and David, because without their support I cannot be sure if I would be in this position today.

Lastly, I'd like to thank Sergio Barrachina Mir for his «LaTeX template for the creation of final degree project memoirs», which I've used as a starting point for the writing of this memoir.

Abstract

Loop3D is a browser-based 3D game engine, developed using an agent-based paradigm. This document details the technical specifications of the engine, as well as the development process behind it. In addition, a small game has been developed for the purpose of testing the engine.

This document encompasses the work done as a final degree project for the Video Game Design and Development Bachelor's Degree at Jaume I University.

The work done consists of the development of a video game engine, intended for the development of three-dimensional video games, which runs on web browsers to provide a simple and portable game development tool. As part of the project, an example game has been created using the engine. This project aims to create a tool to study the internal structure and workings of game engines, as well as serve as a baseline for the future creation of a game editor using the Loop3D engine.

Keywords:
Game Engine, 3D Graphics, Real-time Physics, Browser App

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Index

This chapter contains the initial proposal that was sent to the project supervisor, outlining the project's objectives, methodology, expected outcomes, and a preliminary timeline for development.

This project builds off of the base created by Chover et al. [1], which consists of a simplified 2D game engine that utilizes an agent-based paradigm. The engine developed as part of this project shares many similarities with Chover's engine, but has been heavily modified and has had a large amount of functionality added to work in a 3D setting.

The project uses a scripting system based on *GameScript* [2] but adapted to work in a 3D environment. Such a scripting system uses a structure of conditions, which, depending on their true or false state, trigger certain actions. Games developed using this engine would consist of a set of scenes, each containing a set of actors, which in turn would contain a set of properties and rules which define the functionality of the game. Additionally, the game itself would also contain certain properties, such as its name or camera properties.

1.1 Project motivation

Video game development has evolved significantly in the last decades, with game engines growing ever-more sophisticated, however, with this evolution come certain downsides. Particularly, 3D game engines come prepackaged with a vast array of features, many of which are not applicable to most projects. With this feature bloat also comes a rising barrier to entry to these programs.

This project aims to create a game engine that offers a streamlined and accessible alternative for developers, focusing on simplicity and transparency. By limiting the scope to essential features required for 3D game development in a browser environment, Loop3D avoids the complexity and overhead of full-fledged commercial engines. This makes it not only a suitable educational tool for understanding core engine architecture but also a practical foundation for lightweight game projects.

1.2 Related courses

- VJ1227 - Game Engines. This course provided me with the necessary knowledge of game engine architecture and functionality required to develop this tool.
- VJ1217 - Design and Development of Web Games. This course taught me how to create browser applications as well as the use of *HTML* and *JavaScript*, which is the main tool used in the making of the engine.
- VJ1215 - Algorithms and Data Structures. Data structures are intrinsically linked to game engines. In this engine in particular, a game is represented as an array of scenes, where each of them contains an array of actors.

1.3 Project objectives

The project aims to study the architecture and underlying principles behind game engine software, as well as explore a different approach that forgoes many long-time features in favor of a new paradigm, while maximizing the amount of potential games that can be made with such a tool.

To achieve those objectives, the following goals have been set:

- The creation of a functional scripting system composed of conditions and actions.
- The creation of a 3D game engine capable of running simple games.

- The creation of a functioning loading system, capable of loading games and resources.
- The creation of a game engine capable of running on most web browsers.
- The creation of a game using the engine that is based on the *Unity* tutorial *Tanks: Make a battle game for web and mobile* [7].

1.4 Time distribution

The time and task distribution of the project can be seen in table 1.1.

Task	Time	Total
Research of physics and rendering engines	10 h	10 h
Design document drafting	15 h	25 h
Deciding on game and actor properties	5 h	30 h
Creation of the scene loader	15 h	45 h
Creation of the render module	50 h	80 h
Implementation of HUD	10 h	130 h
Creation of the physics module	35 h	140 h
Creation of the input module	20 h	160 h
Creation of the logic module	35 h	195 h
Creation of the sound module	15 h	210 h
Implementation of animated meshes	25 h	235 h
Creation of a simple demo game	20 h	255 h
Writing of the thesis document	30 h	285 h
Preparation of the final presentation	15 h	300 h

Table 1.1: Time distribution

1.5 Expected results

Upon finishing the project, the expected result is the procurement of a fully functional 3D game engine that runs on most web browsers. This engine will support the creation and execution of 3D games defined through a JSON-based specification format, leveraging an agent-based paradigm and a rule-based scripting system.

The engine will feature core functionalities required for developing interactive 3D experiences, including:

- A rendering system with support for animated meshes, materials and real-time shadows.
- A physics system with support for rigid body dynamics, collision detection and filtering, and support for kinematic and static objects.
- A scripting system based on binary decision trees, enabling rule-based actor behaviors through actions and conditions.
- A sound system supporting playback, volume control, and sound looping.
- A UI system that allows for the rendering of a heads-up display.
- An input handling system capable of detecting keyboard and mouse input, including mouse position and hover events.

To demonstrate the capabilities of the engine, a complete example game will be developed using Loop3D. This game, based on the *Unity* tutorial *Tanks: Make a battle game for web and mobile* [7], will feature real-time multiplayer mechanics, input-based shooting, health tracking, and a simple user interface. It will serve as a validation case for the engine's performance, extensibility, and usability.

1.6 Tools used

Scripting of the engine:

- *JavaScript*
- *HTML*

Libraries used for the modules in the engine:

- Rendering module: *Three.js* [3]
- Physics module: *Ammo.js* [4]
- Logic module: *Math.js* [5]
- Sound module: *Howler.js* [6]

CHAPTER 2

Technical specification

Index

This chapter will explain in detail the inner workings of the Loop3D game engine, as well as specify the requirements of the example game developed using it.

The Loop3D game engine is an 3D engine capable of running on most browsers, it is able to load and run games codified in a specific JSON format which will be detailed later on. Its main distinguishing features that differentiate it from other 3D game engines are as follows:

- A scripting system composed of binary decision trees, with a reduced set of actions and conditions.
- The omission of hierarchichal structures, such as the scene hierarchy present in most game engines.
- Iteration is relegated to the main game loop, meaning that iteration statements are absent from the scripting system.

- Elimination of complex data structures for the user, such as arrays, matrices or vectors.

These changes are aimed at promoting a different game development process, based on a multi-agent paradigm.

2.1 Reach

The Loop3D game engine includes the necessary functionality to create 3D arcade-style games, below is a list of specific features included in the engine:

- A 3D rendering system based on Three.js [3], with support for real-time dynamic Shadow Mapping and skeleton-animated meshes.
- A 3D physics system using *Ammo.js* [4], including rigid body physics and collision detection and filtering.
- A 3D logic system using *Math.js* [5] composed of binary decision trees formed by a series of built-in actions and conditions, as well as support for user-defined variables and properties.
- An input system capable of detecting keyboard and mouse input, as well as mouse position and hovering checks.
- A sound system using *Howler.js* [6] that allows for independent playing, stopping and looping of sounds, as well as setting individual and global volumes.
- A loading system capable of loading games in *JSON* format and external resources, in the form of meshes in FBX format and sounds in Wave or MP3 format.

2.2 Target audience

The main userbase of the engine consists mainly of game developers whose project requirements align with Loop3D's specifications, as well as those interested in studying or analyzing the internal workings of a 3D game engine.

2.3 System requirements

2.3.1 Functional requirements

- Render: The rendering module must support the following:
  - Rendering of 3D meshes, including animated meshes.

  - Support for customizable materials.
  - Dynamic lighting and shadows.
- Physics: The physics module must support the following:
  - Rigid body physics.
  - Collision detection, including when collisions begin and end.
  - Collision filtering.
  - Distinguishing between dynamic, static and kinematic objects.
  - Non-colliding triggers.
- Logic:The logic module must support the following:
  - The function of the CRUD operations (Create, Read, Update, Delete) that is to say, it must allow for the creation of actors, the reading and editing of properties and the deletion of actors.
  - The evaluation of any logical operation, as well as events, such as input events, timers and collisions, and the subsequent execution of actions depending on the result of said evaluation.
- Loader: The loader must support the following:
  - Loading of games in JSON format.
  - Loading of meshes, including animated meshes in FBX format.
  - Loading of sounds in Wave or MP3 format.
- Sound system: The sound system must be able to reproduce or stop reproducing sounds independently of each other, setting their volumes independently or globally, as well as loop them.

2.3.2 Non-functional requirements

- Performance: The engine must run in real time, in 60 fps.
- Stability: There must be no sudden frame drops or spikes.
- Compatibility: The engine must run on *Chromium* and *Firefox* based browsers.

2.3.3 Dependencies

- Programming languages: *JavaScript* and *HTML*
- Physics engine: *Ammo.js* [4]

- Rendering engine: *Three.js* [3]
- Sound engine: *Howler.js* [6]
- Scripting engine: *Math.js* [5]

2.4 System Architecture

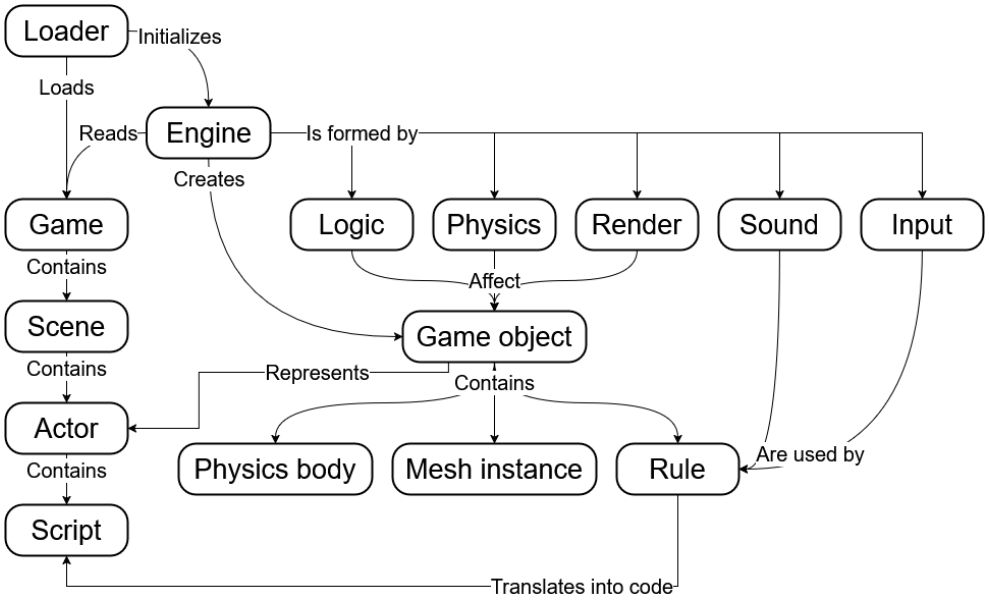

Figure 2.1: Engine Architecture

The engine consists of five distinct modules: the Input, Physics, Logic, Render and Sound modules (See Figure 2.1). These modules are coordinated to work together in generating an interactive process in the *Game Loop*, which is further broken down into two distinct loops, as can be seen in Figure 2.2. On one hand, there's the inner loop, which processes physics, input events and game logic, in that order. Then, the outer loop, which represents the game world by way of the render module. The sound module is called upon by the logic module when specified by user scripts. This loop is launched by the engine when a game is loaded by the loading system.

The engine uses a multi-agent design pattern, in which each actor keeps track of its own state, and the game is composed by putting all the actors together.

2.4.1 Game specification

A game that runs on Loop3D is an object with the following attributes:

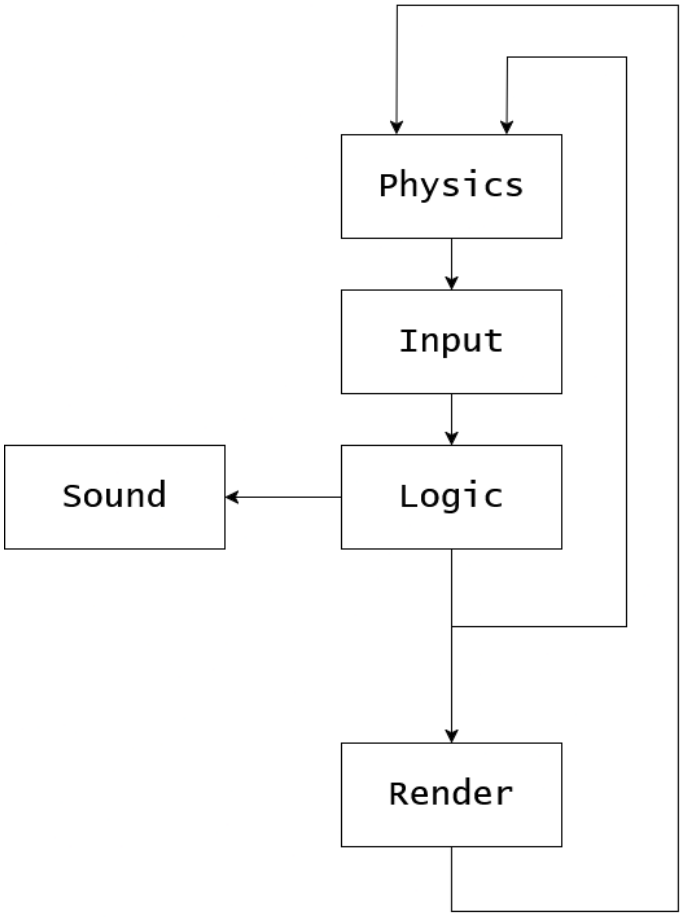

Figure 2.2: Game Loop

- "name": The name of the game.
- Camera settings:
  - "camPositionX", "camPositionY", "camPositionZ": The x, y and z components of the camera's position.
  - "camForwardX", "camForwardY", "camForwardZ": The direction the camera is facing.
  - "camTilt": The rotation of the camera along the direction it's facing.
  - "camFov": The field of view of the camera.
  - "viewPortWidth", "viewPortHeight": The width and height of the game's viewport.
  - "perspectiveType": Can be set to "orthographic" or "perspective".

- Lighting settings:
  - "dirLightDirectionX", "dirLightDirectionY", "dirLightDirectionZ": The direction of the light that shines down on the game world.
  - "dirLightColor": The color of said light, expressed as a hexadecimal string.
  - "dirLightIntensity": The intensity of that light.
  - "shadows": Sets whether shadows are enabled or not.
- Skybox settings:
  - "skyTopColor": The color at the top of the skybox.
  - "skyBottomColor": The color at the bottom of the skybox.
  - "skyHorizonColor": The color where the top and bottom meet.
- Physics settings:
  - "physicsOn": Sets whether to run the physics simulation or not.
  - "gravityX", "gravityY", "gravityZ": The direction of gravity.
- "sceneList": The list of scenes in the game.

Each of the scenes in the scene list contain two attributes, a name and an "actorList", which is a list containing the actors in the scene. Each actor is an object containing the following attributes:

- General settings:
  - "name": The name of the actor.
  - "positionX", "positionY", "positionZ": The position of the actor on the game world.
  - "rotationX", "rotationY", "rotationZ": The Euler angles of the actor.
  - "scaleX", "scaleY", "scaleZ": The scale of the actor.
  - "tag": Allows tagging actors for collision detection purposes.
  - "collider": Can be set to "sphere" or "box".
  - "colliderSizeX", "colliderSizeY", "colliderSizeZ": If collider is set to box, it details the size of each of the sides of the box, otherwise the largest one is used as the radius for the sphere. Can be set to -1 for the engine to calculate it automatically based off of the actor's mesh.

  - "colliderCenterX", "colliderCenterY", "colliderCenterZ": Sets the center of the collider relative to its position.
  - "screen": If set to true, marks the actor as a HUD element.
  - "sleeping": If true, the actor's rules won't be evaluated in the logic module.
  - "visible": Whether the object will be rendered in the game world or not.
  - "customProperties": An object where the user can define custom properties for the actor.
  - "spawnOnStart": Whether to spawn the actor when the scene is loaded.

- Mesh settings:
  - "mesh": The local path the to the FBX file to use as the actor's mesh.
  - "materials": A list of objects used to define the actor's material properties. Each of them contains the properties "color", "metalness", "roughness", "transparent" and "opacity". There are some premade materials as part of the engine that can be used by putting their name instead of a material object.
  - "animation": The currently playing animation.
  - "animationLoop": Whether to loop animations or not.
  - "transitionTime": The duration of interpolation between animations.

- Sound settings:
  - "sounds": A list of sounds that can be reproduced by the actor. Each of them contains the following properties:
    * "name": The name used to refer to the sound.
    * "source": The local path the to the MP3 or Wave file to use for the sound.
    * "loop": Whether to loop the sound after it finishes or not.
  - "sound": The currently playing sound.
  - "volume": The volume of sounds emitted by this actor.

- Physics settings:
  - "physicsMode": Can be set to "static", "dynamic", "kinematic" or "none" to define the object's physics behaviour.

  - "movementRestrictionX", "movementRestrictionY", "movementRestrictionZ": Defines which axes the physics simulation can move the object in.
  - "rotationRestrictionX", "rotationRestrictionY", "rotationRestrictionZ": Defines which axes the physics simulation can rotate the object around.
  - "velocityX", "velocityY", "velocityZ": The velocity of the object.
  - "angularVelocityX", "angularVelocityY", "angularVelocityZ": The rotational velocity of the object.
  - "mass": The mass of the object.
  - "friction", "rollingFriction": The friction the object exerts on objects sliding and rolling on it, respectively.
  - "bounciness": The fraction of energy kept after a collision.
  - "drag", "angularDrag": The drag the object experiences when moving and rotating.
  - "trigger": When set to true, the object doesn't affect other objects through the physics simulation, but still registers collisions.
  - "ignoreGravity": Set to true to prevent the object from falling.
  - "collisionGroup": bitmask indicating the collision group that the object belongs to, used for collision filtering.
  - "collisionMask": bitmask indicating which collision groups the object collides against.

- Light settings:
  - "lightColor": The color of the light emitted by the object.
  - "lightIntensity": The intensity of said light.
  - "lightAmplitude": The angle of the amplitude of the light.
  - "lightForward": The direction the light is shining in.
- "scripts": A list of the scripts that define the actor's behaviour.

A diagram of this game structure can be seen in Figure 2.3

2.4.2 Game objects

In Loop3D, an actor is an object that holds properties that, when put together with the rest of the actors of a game, define the functioning of said game. A game object is the representation inside the game world of said actor. It holds the same properties as the actor its based off of, adding the necessary

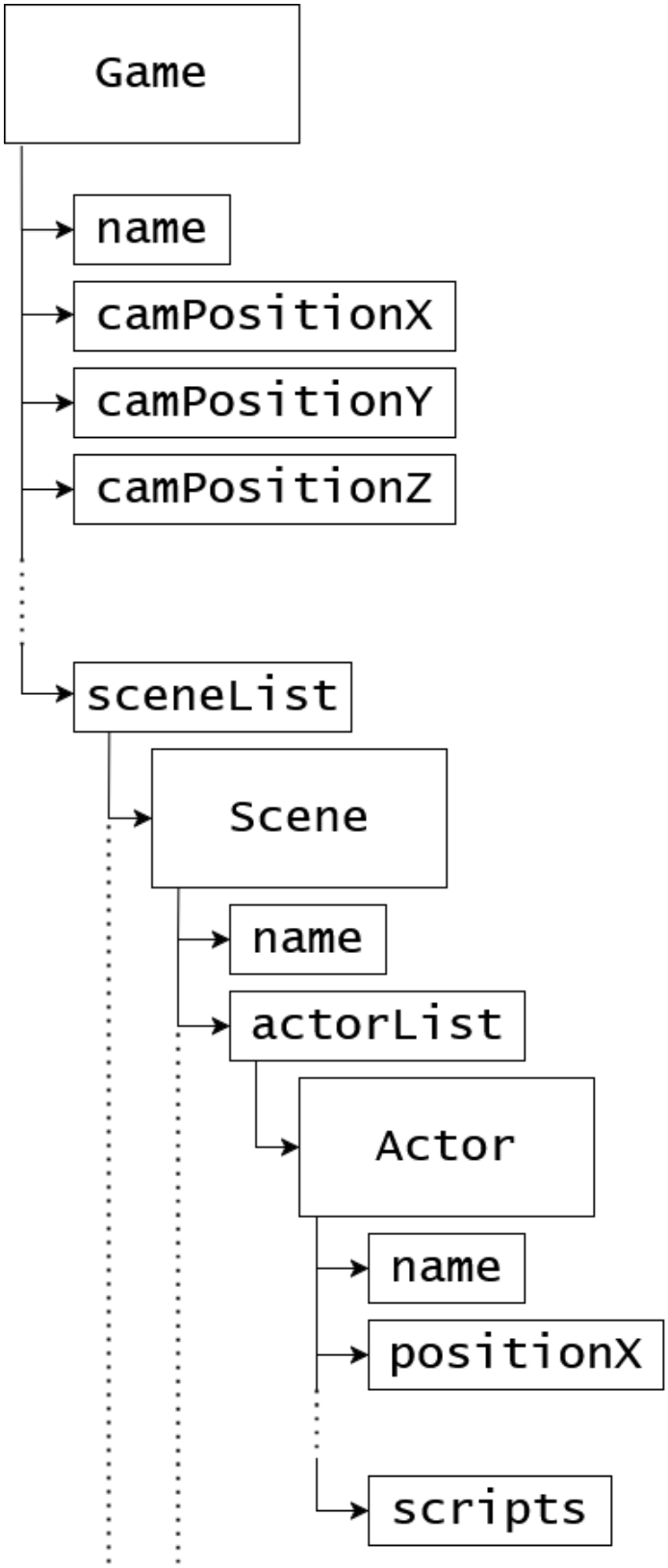

Figure 2.3: Game Structure Diagram

```
gameLoop(newTime) {     // Outer loop
    this.animationRequest = window.requestAnimationFrame(this.gameLoop.bind(this));
    this.frameTime = (newTime - this.currentTime) / 1000;
    if (this.frameTime > 0.1) this.frameTime = 0.1;
    this.accumulator += this.frameTime;
    while (this.accumulator >= this.deltaTime && this.loopRunning) {    // Inner loop
        this.physics.update(this.deltaTime);            // Update physics simulation
        this.activeGameObjects.forEach((gameObject) => {    // Evaluate game object rules
            gameObject.fixedUpdate();
        });
        Input.restartInput();               // Reset input states for the next frame
        this.time += this.deltaTime;
        this.accumulator -= this.deltaTime;
    }
    this.render.update(this.frameTime);     // Render the game onto the screen
    this.currentTime = newTime;
}
```

Figure 2.4: Code for the game loop

attributes to hook it up to all the modules of the engine. Essentially, actors are blueprints to create the game objects that collectively make up the game. When an actor is "spawned" this means that a game object has been created and put into the game world, according to the blueprint provided by the actor.

2.4.3 The game loop

As was mentioned, the game loop consists of two parallel running loops. Each iteration of the outer loop takes a variable amount of time and results in a render on the screen. The inner loop runs precisely 60 times per second, and each iteration updates the physics simulation, evaluates all game object rules and resets input events. See figure 2.4 for a code snippet corresponding to the game loop.

2.4.4 The loading system

The loading system is in charge of loading games, meshes in FBX format, and sounds in wave or MP3 format. To load sounds it uses the *Howler.js* [6] library, and for meshes it uses an implementation from the *Three.js* [3] library. For games, the *JSON* format was used.

When a game is loaded by the loading system, it is passed to the engine. The engine then initializes the physics, render, sound and input modules, then the first scene is initialized. When a scene is initialized, all the actors in the scene are spawned into the game world. Finally, the game loop begins running. To change scenes, the game loop is paused, all the game objects in the game world are deleted, the actors on the new scene are spawned and then the game loop is resumed.

2.4.5 The physics module

When it comes to the physics module, game objects hold a reference to a physics body, which can be created and interacted with through the Rigidbody class. This class holds static functions used to create or alter physics bodies. The functioning of the physics module itself is defined in the Physics class, which is mainly in charge of running the physics simulation and calculating collisions. When two physics bodies collide, this is stored in the game objects that are connected to said physics bodies. These collisions are sorted by tags and further sorted by game object names, so that they can be easily looked up. Collisions have three possible states: "enter", which is true the frame that the collision begins; "stay", which is true while the collision persists; and "exit", which is true the frame the collision ends. Each physics update the physics module advances the simulation, then calculated collisions and then updates the properties of the game objects in the scene accordingly.

2.4.6 The input module

The input module is in charge of detecting and handling input events, which mainly consists of keeping track of the mouse position, detecting when the mouse hovers a specific game object and detecting keyboard and mouse button input. This is done through the Input class, which sets up the event listeners necessary for said operations and then handles all input events. When it comes to mouse and keyboard button input, a dictionary is kept which tracks the state of all keys. Keys can have three states: pressed, which is true the frame the key is pressed; down, which is true while the key is held down; and released, which is true the frame the key stops being held.

2.4.7 The logic module

The logic module is largely composed of the Rule class. This class receives the scripts defined by the user and translates them into compiled code that executes the desired actions and conditions on each game object. This module is also in charge of running logic updates by executing said compiled code every frame. The list of actions and conditions included in the engine is as follows:

- Actions: There are three main actions which accomplish the CRUD operations in Loop3D, these are:
  - "edit": Receives a property and a value, and sets the property to said value. This fulfills the *Read* and *Update* functions.
  - "spawn": Receives an actor and an object with actor attributes, and spawns said actor with the attributes specified. This fulfills the *Create* function.

- "delete": Deletes the game object from the game world. This fulfills the *Delete* function.

Aside from these, there are more actions included in the engine to make development more straightforward, although all of these could be substituted by a sequence of "edit" actions.

- "move": Receives a direction and a speed and moves the game object that runs it in said direction.
- "move_to": Receives a point in space and a speed and moves the game object towards that point.
- "rotate": Receives a direction and an angle, and rotates the game object around the specified axis by the given angle.
- "rotate_to": Receives a direction and a speed, and rotates the game object towards that orientation at the specified speed.
- "rotate_around": Receives a point, an axis and an angle, and rotates the game object around that point and axis.
- "push": Applies a force in the given direction to the game object.
- "push_to": Applies a force that moves the game object towards a specific point.
- "impulse": Applies an instantaneous force in a given direction to the game object.
- "impulse_to": Applies an instantaneous force that moves the game object towards a specific point.
- "torque": Applies a rotational force around a given axis to the game object.
- "torque_impulse": Applies an instantaneous rotational force to the game object.
- "set_timer": Creates a timer with a name, duration, repetition flag, and auto-start option.
- "start_timer": Starts a previously created timer on the game object.
- "stop_timer": Stops a running timer on the game object.
- "reset_timer": Resets the timer to its initial state and duration.
- "delete_timer": Removes a timer from the game object.
- "play_sound": Plays a sound on the game object.
- "stop_sound": Stops a currently playing sound on the game object.
- "set_volume": Sets the volume of a specific sound on the game object.

  - "set_global_volume": Sets the global sound volume for the whole game.
  - "animate": Plays an animation on the game object, with optional loop and transition time.
  - "stop_animation": Stops the currently playing animation on the game object, with an optional transition time.

- Conditions:
  - "compare": Receives two values and a logical operation and checks if the operation is true.
  - "check": Receives a property and checks if it evaluates to true.
  - "collision": Checks if the game object is colliding with any object whose tag matches one of the given tags, while also checking for the status of the collision.
  - "check_timer": Checks if a timer with a given name has run out on the game object.
  - "input": Checks if a specific button input, including its status.
  - "hover": Checks if the mouse is currently hovering over the game object.

2.4.8 The sound module

The sound module is handled by the game objects, each of which holds a list of sounds which can be reproduced at any moment.

2.4.9 The render module

The render module is in charge of rendering meshes and materials for the game objects, which are bundled together as an object called a mesh instance, which is held by each game object. These mesh instances are created and interacted with through the MeshRenderer class, which holds static functions to this end. Most of the work of this module, including running render updates, handling the HUD and camera and setting the skybox, are managed by the Renderer class. This class holds two separate render scenes, one for the world objects and another one for HUD objects and overlays them.

2.5 Testing and validation

A complete testing plan is detailed in Table 2.1

Test element	**Testing procedure**	**Acceptance criteria**
Scene Loader	A test scene will be created in a JSON file and attempted to be loaded; then the contents of the loaded scene will be shown in the console.	The scene loads correctly with all its elements.
Render Module	The previously mentioned test scene will be rendered to a Canvas.	The scene is displayed correctly, and all its actors are identifiable.
Actor Properties	All actor properties will be changed during runtime.	The changes in actor properties are reflected in the rendered scene.
Game Properties	All game properties will be changed during runtime.	The changes in game properties are reflected in the rendered scene.
Physics Module	Several actors will be placed in the scene so that the physics simulation makes them fall and collide with each other.	Actors behave as expected, interacting with each other correctly.
Shadow Mapping	A scene with shadow-casting objects and multiple light sources will be created.	Shadows are rendered correctly and the visual result is as expected.
HUD	A scene with a HUD will be created and the camera will be moved.	The HUD behaves correctly, staying in a fixed position on the screen.
Input Module	Each key will be pressed in succession. The detected inputs will be logged onto the console.	Input events are correctly detected, including when they begin and end.
Logic Module	A scene will be created with an actor, and a script containing a condition and an action will be added to the actor. After the test, the condition-action combination will be changed and tested again. This will be repeated until all actions and conditions have been tested.	Scripts execute correctly when the appropriate condition is met, performing the expected action.
Sound Module	A sound will be loaded and attempted to be played from an actor.	The sound plays correctly.
Animated Meshes	An animated mesh will be loaded and various animations will be played.	Animations play correctly.

Table 2.1: Testing and Validation Table

2.6 The test game

This section will detail the mechanics and features included in the test game developed for the engine, as well as explain the general concept of said game.

2.6.1 Game summary

The game, based on the *Tanks!* tutorial developed by *Unity* [7] features two tanks, one red and one blue, which can be independently controlled to move and fire explosive shells. The game is intended to be played by two players, each controlling one tank, with the goal of the game being to eliminate the other tank by shooting it down.

2.6.2 Game mechanics

- Player controls (See Table 2.2)
  - Each tank can move forward or backward, as well as rotate in either direction.
  - Tanks can fire explosive shells, which travel in an arc and explode upon contact with the floor, the edges of the map or a tank.
  - Shots can be charged by holding down the shoot button for up to 1.4 seconds, and are fired upon releasing said button. Charging increases travel distance and speed.
- Rules
  - Each tank has 100 health points at the beginning of the game.
  - Explosions remove 35 points of health from any tank they come into contact with.
  - Once a tank's health reaches 0, the tank is defeated and the opposing player wins. The players can then return to the menu screen.
  - The level is a 50 x 50 meter square surrounded with invisible walls.
  - The tanks spawn 20 meters from the center on opposite sides of the square.
- Physics
  - Players move at 5 meters per second.
  - Players rotate at 100^{o} per second.
  - Shot charge goes from 0 to 14 linearly depending on charge time.

– Shot speed is calculated from shot charge according to the following equation where $\vec{f}$ is the forward vector of the firing tank and c is the shot charge:

$$\vec{v} = (\vec{f}_x(c+3), \frac{c}{2}, \vec{f}_z(c+3))$$

	Player 1	Player 2
Move forward	W key	Up arrow
Move backward	S key	Down arrow
Rotate right	D key	Right arrow
Rotate left	A key	Left arrow
Shoot	Space bar	Left click

Table 2.2: Game controls

2.6.3 The main menu

The game features a menu, which is the first scene loaded. The menu has a button that allows the players to start the game, and a slider to control the volume of the game.

The menu can be navigated using the mouse, by clicking on the slider knob and the start button. It can also be navigated using the keyboard, by using the "w" key to select the start button and the "s" key to select the slider. With the button selected, the enter key can be used to activate it. With the slider selected, the "a" and "d" keys can be used to move it left and right, respectively.

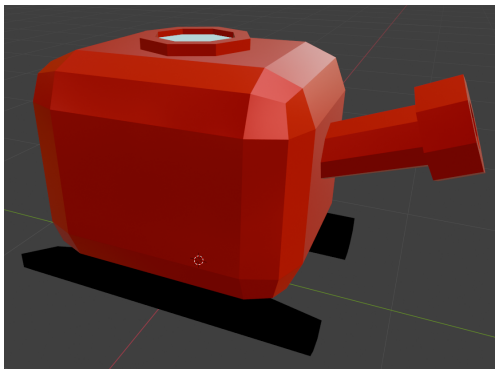

Figure 2.5: The tank model

2.6.4 Art

The art of the game was kept simple, as its main purpose is to serve as a showcase of the engine and its scripting capabilities. For this, a low-poly

artstyle was used, see Figure 2.5 for the model of the tanks. To differentiate the players, a blue and red color scheme was used. Player 1's tanks and shells are red, while Player 2's are blue. Tanks also feature a recoil animation when they shoot.

2.6.5 Sound

The game features looping background music, composed of a track by Fesliyan Studios [8], in accordance to its license, which allows its free use for commercial and non-commercial projects so long as credit is given.

There are also several sound effects in the game, specifically there is one for each of the following events:

- Menu navigation
- Tank shooting
- Shell exploding
- Tank getting eliminated

All sound effects in the game were generated using the web tool *jsfxr* [9].

2.6.6 HUD

The game features a HUD, composed of the health bars of each player, following the previously defined color scheme. Player 1's bar is placed at the top left corner of the game window, and Player 2's at the top right. They decrease proportionally to each player's health points.

While charging a shot, each player also has a dynamic indicator, showing how far their shot is going to travel, and where it's going to land.

CHAPTER 3

Development and Implementation

Index

This chapter will explain the process of developing Loop3D, detailing the work done to complete each of the objectives and meet the requirements of the project. The chapter will be organized by the tasks done, which will be explained in order. A repository containing the source code of the project can be found at the following address: `https://github.com/kiskidiego/Loop3D`.

3.1 Deciding the bases of the engine

The first task that needed to be completed was deciding which libraries were going to be used for the project. First, the render engine was decided. After analyzing a few options, the chosen engine was *Three.js* [3], because it is one of the most widely used graphics engines for the *JavaScript* language, while also providing a layer of abstraction that makes it reasonably easy to work with. This decision influenced the choice of physics engine, which ultimately

was *Ammo.js* [4], since there are a variety of official examples integrating both engines on the *Three.js* website. *Howler.js* [6] and *Math.js* [5] were chosen to stay as close to Chover et al. [1]'s work as possible.

After this decision, the next task was defining the game and actor properties for this engine. Most of them are carry-overs from Chover et al. [1]'s work, however, a few had to be added or adapted to better suit a 3D environment. Finally, the list was refined to the aforementioned properties (See chapter 2.3.1). Once the properties were created, the Actor, Scene and Game classes were created, along with the Engine class.

3.2 Creation of the scene loader

The next order of business was creating the mechanism which would read a JSON file containing a game, and loading said game. An example of such a file can be found at the following address: `https://github.com/kiskidiego/Loop3D/blob/main/TestGame/game.json`. For this purpose, a FileLoader class was created. This class implements a static method to load said game files, which are then passed to the Engine class. The Engine class would then parse the game, extracting from it the scenes and actors and displaying them in the console for testing. No obstacles were found in this stage of development.

3.3 The render module

The implementation of the render module was the first challenging part of the development process. In order to properly explain how it works, we have to differentiate between the game scene and the render scene. The game scene is populated by game objects, which have physics, logic, render and sound properties among others. The render scene is what is actually visible to the player, and it's populated by render objects, such as lights or meshes. The game scene is a data structure that holds information relevant to the game engine, while the render scene is used by *Three.js* [3] to render those objects to the screen.

To implement the render module, a Renderer class was created, which manages the render scene. When a game is loaded, the Engine object creates an instance of the Renderer class, which sets up the camera, directional light and skybox, and adds them to the render scene. Once this worked, the next step of the implementation was loading meshes from an FBX file and adding those to the render scene. For this, a function was added to the FileLoader class, which takes the path of an FBX file and loads it as a render object. Then, the MeshRenderer class was created to allow game objects to interact with and alter their render objects. Materials were added using *Three.js*' [3] MeshStandardMaterial class.

A problem was encountered at this stage, which is that the loading of new render objects would take too much time and cause a lag spike. Two solutions were implemented to combat this, the first being asynchronous loading of FBX files, and the second being caching of render objects. This way, once an FBX file is loaded once, it doesn't have to get loaded again to spawn another instance of the same render object.

Put together, the flowchart for the process to create the render scene can be seen in Figure 3.1.

To implement shadow mapping, a special camera was attached to the directional light of the scene. This camera, called a shadow camera in *Three.js*' [3] API, renders the scene onto a frame buffer, and then uses said frame buffer to calculate the position and shape of the shadows in the scene. Said functionality was added to the Renderer class. This presented some problems, however, as the directional light affects the entire scene, which is of infinite size, but the texture outputted by the shadow camera cannot be of infinite size, the resolution of said texture had to be carefully picked to optimize the area of the scene affected by shadows. This could've been fixed by implementing cascading shadow maps, but this fell outside the project's scope.

3.3.1 About game object lights

Unfortunately, the functionality to allow game objects to emit light has been removed from the engine. This is due to the cost of instantiating said lights, which would cause a severe lag spike whenever generated. Multiple solutions have been proposed but none have been implemented. The functionality remains in the engine, but has been deactivated.

3.4 The physics module

The first task at hand when implementing the physics module was creating the physics world. Similarly to how the render module has a render scene, the physics module has a physics world. In this case, it's populated by physics objects, which are then manipulated by the physics simulation. To do this, a Physics class was created, which was instantiated by the Engine class when loading a game. However, some problems arose, relating to the fact that *Ammo.js* [4] has to be initialized asynchronously before any physics operations can start. To solve this, the game loading function was set to be executed as a callback to the *Ammo.js* [4] initializing function.

After setting up the physics world, the next step was creating a physics object for each actor in the scene and attaching it to the corresponding game object. For this, the Rigidbody class was created as a way to create and interact with physics objects. Then, the physics update loop was created in the

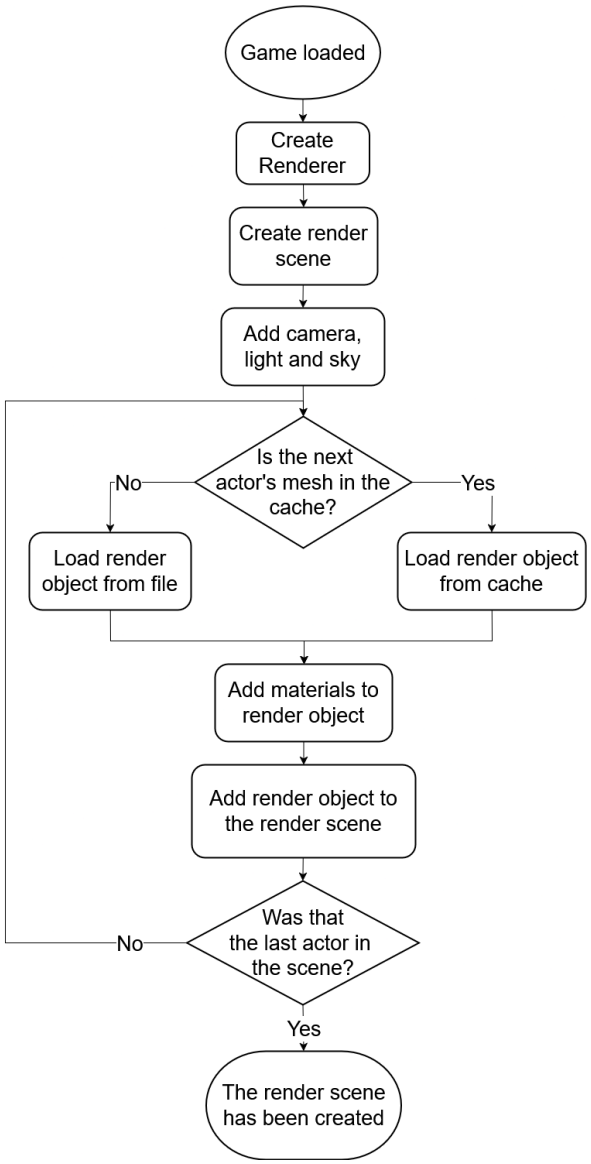

Figure 3.1: Render scene creation

Physics class. This function advances one step of the physics simulation, then detects the subsequent collisions, and finally updates the position, rotation and velocity attributes of all physics-enabled game objects.

To detect collisions, a loop runs through every manifold in the physics scene, verifying if a collision has occurred, in which case it will store a collision object on the involved game objects. collision objects are stored according to the tag and name of the colliding object, which allows for later selection of collisions. A collision object has three properties, all of which are boolean values:

- "enter" is true if this is the first physics update when the collision is occurring.
- "stay" is true every physics update while the collision takes place.
- "exit" is true in the physics update when the collision ends.

After finishing the physics module, the game loop was implemented in order to properly test it, as well as to test the synchronization between the physics and render modules. This loop is in the Engine class, and, in its first version, it called the physics update loop at a regular interval, while rendering the game onto the screen every screen update, using the *JavaScript* requestAnimationFrame function.

3.5 Implementing the HUD system

The HUD system in Loop3D is implemented by way of creating a second render scene, this time with a fixed orthographic camera, and overlaying the output of said camera onto the output of the regular render scene. To implement such a system, when spawning an actor, its "screen" property is checked. If said variable is true, then the physics object associated to the actor's game object is removed from the physics world, and its render object is removed from the render scene and placed into the HUD scene.

3.6 The input module

To implement the input module, an Input class was created. This class is instantiated by the Engine class on loading a game, and upon instantiation creates *JavaScript* event listeners for the press and release of any button, as well as for mouse movement.

The Input class keeps a dictionary whose keys are the names of the physical keys on a keyboard and mouse. The dictionary's values are input objects. An input object holds three values:

- "pressed" is true during the logic update when the button is first pressed.
- "down" is true while the button is held.
- "released" is true during the logic update when the button is released.

Two more functions were implemented, one to keep track of mouse position, and one that takes a game object and returns true if the mouse is hovering it. It calculates the result using a *Three.js* [3] ray cast.

3.7 The logic module

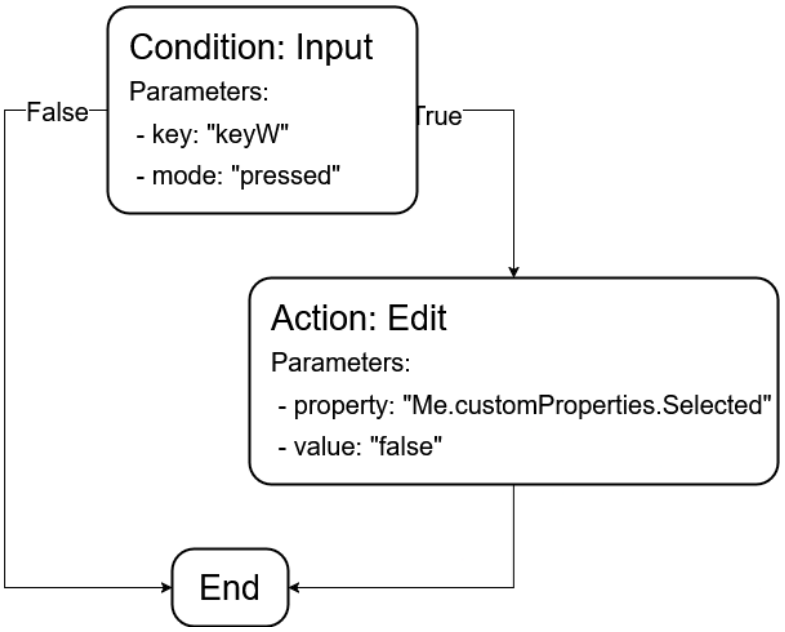

Figure 3.2: A user-defined rule

The logic module is governed by rules. These rules are user-defined scripts that come attached to actors as *JavaScript* objects, as seen in Figure 3.2. To translate these objects into compiled, executable code the Rule class was implemented. This class has a function for each action and condition included in the engine; these functions parse rules into strings of code, which is then compiled using the *Math.js* [5] library (Figure 3.3). When a game object is created, its rules are looped through and their parsing functions are called. An update function was implemented into the GameObject class, which evaluates these compiled expressions. Finally, to include the logic module into the game loop, every game object's update function is called after the physics update.

```
edit(params) {
    return (params.property + " = " + params.value);
}
delete() {
    return ("Engine.delete(" + this.gameObject.name + ")");
}
```

Figure 3.3: Parsing functions for the edit and delete actions

3.8 The sound module

The sound module is implemented through the *Howler.js* [6] library. To implement it, the "sounds" property was added to the Actor class. This property is an array of objects with a source, name, volume and loop values. When an actor is spawned onto the game scene, its sounds are parsed into Howl objects. These objects have a Play function, which is called to reproduce the sound. A wrapper was created as part of the logic module, to call this function through user-defined scripts.

```
"sounds": [
  {
    "name": "background_music",
    "source": "./TestGame/Resources/Sounds/background_music.mp3",
    "volume": 1,
    "loop": true
  }
],
```

Figure 3.4: A "sounds" object

A problem was encountered during testing of the sound module. Due to security restrictions, browsers don't allow the creation of sound contexts before the user has interacted with the page. To solve this, before loading the game a prompt was added to the page, asking the player to click it to start the game. Upon clicking on the page the sound context is initialized and the game is loaded.

3.9 Developing the test game

3.9.1 The menu screen

The first part of the game that was developed was the menu screen. This is a scene with the following actors:

- The title of the game
- The play button
- The volume slider
- The volume knob

Of these, the volume slider is purely cosmetic, and the other ones have scripts. The title has a script that plays its animation on scene start up. This was implemented by adding a custom property to it, to keep track of whether the animation had started playing. The play button uses a similar method to play the background music. The play button also includes a script for navigation and another one for starting the game, which is done by changing the active scene from the menu scene to the game scene. The volume knob also has a script for navigation, using similar methods to the play button. It also has a script for moving or dragging the knob, which also changes the volume of the game to match.

3.9.2 Tank movement

The next part of the development process was setting up the game scene, this meant adding the floor and the invisible walls. The floor also includes code to play the background music.

Once that was done, the movement of the tanks was implemented. They both use similar scripts to detect input and apply the appropriate velocity and rotation. During this part of development, a problem was encountered with the way rotations were implemented in the engine, which lead to gimbal lock. The implementation was changed to use quaternions more prominently than it already did to fix this issue.

3.9.3 Shooting

To implement the shooting of shells, it was important that their velocity was set by the tank at the moment of their creation. For this purpose, the "spawn" action was modified to accept a list of parameters to override the actor's default attributes. Using this, the velocity of the shells was set. Then the "Explosion" actor was created, which is used to reduce the health of the tanks when a collision occurs.

3.9.4 HUD and indicators

The healthbar actors were implemented to help players keep track of their health. The shot distance indicators were also added, by using an approximation to set their length so that the shell launched would land as close as

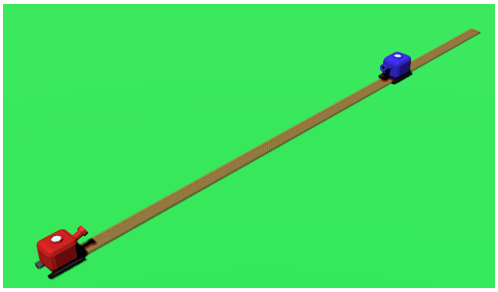

Figure 3.5: Player 1's range indicator

possible to the tip of the indicator. The approximation used is according to the following equation, where c is the charge amount and l is the indicator length:

$$l = \frac{c^2}{8} + 3$$

Finally, dynamic camera movement was added, which moves the camera so that the tanks are always in the frame no matter where they are.

3.9.5 The ending message

The ending message was implemented as two actors, each of them containing a message to congratulate one of the players. Once they are visible, the next time a click is detected or the enter key is pressed, the menu scene will be loaded.

3.9.6 Game scene overview

Put together, the following is a list of all actors in the game scene, as well as their behaviours:

- Floor
  - Play music
  - Set skybox colors
  - Change back to the menu scene
- North wall
- South wall
- West wall
- East wall

- Tank 1
  - Movement and rotation
  - Shooting and health
- Tank 2
  - Movement and rotation
  - Shooting and health
- Bullet 1
  - Explode
- Bullet 2
  - Explode
- Explosion
  - Damage tanks
- Health bar 1
  - Change size according to the health of Tank 1
- Health bar 2
  - Change size according to the health of Tank 2
- Health bar frame 1
- Health bar frame 2
- Camera controller
  - Dynamic camera movement
- Indicator 1
  - Show the travel distance and landing spot of shots fired by Tank 1
- Indicator 2
  - Show the travel distance and landing spot of shots fired by Tank 2
- Ending message 1
  - Congratulate Tank 1's victory
  - Return to menu scene

- Ending message 2
  - Congratulate Tank 2's victory
  - Return to menu scene

CHAPTER 4

Results

Index

The results of the project have been positive, with Loop3D achieving its primary objectives and demonstrating robust functionality across all implemented modules. The testing phase included both unit and integration tests, and concluded with the development of a fully functional game to validate the engine in a realistic use case.

4.1 Test coverage and validation

A series of comprehensive tests were run on the engine (see Table 2.1), checking all modules and functionalities in the engine. Each test focused on specific criteria relating to the project requirements and objectives.

- Scene Loader: The loader correctly parsed JSON game files and instantiated scenes and actors as intended. Console logs verified that every property was read and applied, confirming reliable deserialization and initialization.
- Rendering System: Render scenes displayed all actors, meshes, and animations correctly. Shadow mapping functioned within expected con-

straints, and mesh caching effectively reduced load times and prevented frame drops.

- Actor and Game Properties: Runtime property changes to both actors and global game settings (e.g., lighting, skybox colors, camera parameters) took immediate visual and behavioral effect in the running scene.

- Physics System: Rigid body physics, collisions, and collision filtering worked consistently. Collision states (enter, stay, exit) were correctly triggered and could be used reliably in scripting. Stability was maintained across stress tests with multiple interacting actors.

- HUD: The UI elements remained fixed to the viewport, as expected from screen-space objects. Their separation from the world render scene allowed for consistent rendering regardless of camera movement.

- Input System: All keyboard and mouse inputs were correctly detected. Button states (pressed, down, released) updated as expected, enabling reliable scripting of game logic based on player input.

- Logic System: Every available action and condition in the scripting system was tested. Binary decision trees successfully controlled actor behavior and responded dynamically to input, physics events, and timers.

- Sound System: Sounds triggered correctly and could be played, stopped, looped, and volume-controlled both locally and globally. Browser limitations on audio initialization were effectively bypassed through a startup interaction prompt.

- Animated Meshes: Animation playback (including transitions, loops, and manual stopping) was visually smooth and performed with no issues.

With the exception of game object lights, which were deactivated due to performance concerns during instantiation, all tested features met their criteria. The engine also maintained a stable frame rate (60 FPS) in Chromium and Firefox-based browsers under typical loads, so long as a dedicated graphics card was being used to run the browser.

4.2 Test game validation

To validate the engine's practical usability, a complete game was developed based on a *Unity* tutorial [7]. This game served as an end-to-end test of the engine's rendering, physics, scripting, audio, input, and UI capabilities.

Key outcomes include:

- Gameplay Functionality: The game successfully implemented two-player controls, projectile physics with charging mechanics, health tracking, and win conditions using Loop3D's scripting system alone.
- Performance: Real-time performance was maintained throughout gameplay, without noticeable drops in frame rate.
- Script Reliability: All game logic was implemented using the engine's rule system without direct access to traditional control structures like loops, highlighting the power of the binary decision tree paradigm.

4.3 Summary

The Loop3D engine successfully fulfills the goals set at the start of the project. It demonstrates that a lightweight, browser-compatible 3D game engine is feasible and effective, and it highlights core engine infrastructure and architecture common to most game engines, while disposing of unnecessary functionality.

The following outcomes summarize the results:

- All engine modules operate as intended under test conditions.
- A feature-complete game was successfully built using Loop3D, demonstrating real-time performance and usability.
- All critical acceptance criteria from the testing plan were met.

CHAPTER 5

Conclusions

Index

This project was quite an enriching experience. during its development, many of the skills learned during years of study in the Video game design and development bachelor's degree were put to the test. It's also allowed the acquisition of new knowledge on some subjects that aren't thoroughly touched upon in the degree, such as the internal workings of game engines, or more advanced *JavaScript* programming.

This project has long been a subject of personal interest, due to a long-standing ambition to build a custom game engine. In retrospect, there are some aspects that could have been approached differently, as well as features that would be valuable to add or improve upon in the future. Nevertheless, Loop3D stands as a positive development experience.

5.1 Future work

As mentioned before, there are some absent features which would be very beneficial for the engine, not the least of which is object pooling, as this would likely allow for the addition of game object lights, as the problem with instantiating light sources would hopefully be avoided. Other potential features include spatial sound, cascading shadows and precalculated shadow maps.

There are also three main lines of future research that can be potentially pursued using this engine.

- Firstly, there's the issue of creating a dedicated visual editor for the engine to allow the creation and debugging of games without the need of directly writing its *JSON* representation.
- Secondly, taking advantage of the simplicity of the scripting system of the engine, it could be plausible to create an LLM capable of creating games autonomously, directed by instructions in natural language.
- Lastly, due to the lightweight nature of the logic module, there could be a way to run it in the GPU using compute shaders, instead of the CPU. This opens up the idea of an engine that runs entirely, or almost entirely, in the GPU, allowing for massive speed improvements.

Bibliography

[1] Chover, M., Marín, C., Rebollo, C. et al, *A game engine designed to simplify 2D video game development.*, Multimed Tools Appl 79, 12307–12328 (2020).
https://doi.org/10.1007/s11042-019-08433-z

[2] Marín-Lora, C., Chover, M., *GameScript: a simplified scripting language for video game development.*, Multimedia Systems 31, 70 (2025).
https://doi.org/10.1007/s00530-024-01574-8

[3] Three.js https://threejs.org/

[4] Ammo.js https://kripken.github.io/ammo.js/

[5] Math.js https://mathjs.org/

[6] Howler.js https://howlerjs.com/

[7] Unity Technologies, *Tanks: Make a battle game for web and mobile*, Unity Learn (2025).
https://learn.unity.com/course/tanks-make-a-battle-game-for-web-and-mobile

[8] Fesliyan Studios https://www.fesliyanstudios.com/

[9] Eric Fredricksen, *jsfxr - 8 bit sound maker and online sfx generator*, (2011).
https://sfxr.me/