Functional Specification

Student 1: Ivan Shvydchenko, 21317061

Student 2: Jibran Jarwar, 21353811

Supervisor: Renaat Verbruggen

Project Title: HoneyMoon Engine

0. Table of contents

1. Introduction

1.1 Overview

The HoneyMoon Engine is a 2D game development tool designed to provide a more accessible entry point for aspiring game developers who find traditional engines like Unity overwhelming.

This game engine enables users to create games using Lua scripting, while the core functionality and complex processing are managed in C++. Key features include object rendering, camera control, basic physics, collision handling, and animation support, allowing users to focus on game logic without needing to dive into lower-level programming tasks.

The engine will integrate with tools like SDL for rendering, ImGUI for interface management and LuaC++ for native scripting . This approach aims to balance simplicity and functionality, making it an ideal learning platform and a gateway into games development fundamentals.

1.2 Glossary

2D Game Engine: A software framework designed to help developers create two-dimensional games. It provides essential tools and libraries to manage game elements like graphics, physics, and native scripting.

Lua: A lightweight, high-level scripting language commonly used in game development for its simplicity and flexibility. In HoneyMoon Engine, Lua is used for scripting game behaviour, allowing developers to write game logic in an accessible language.

SDL: (Simple DirectMedia Layer): A library used for handling low-level access to audio, keyboard, mouse, and graphics, making it easier to create multimedia applications, including games. HoneyMoon Engine uses SDL for rendering and managing window events.

ImGUI: (Immediate Mode Graphical User Interface): A user interface library used for creating graphical interfaces in real-time applications, often in game engines for debugging and development tools.

PNG: (Portable Network Graphic) a type of raster image file, it can handle graphics with transparent or semi-transparent backgrounds.

JPG: (Joint Photographic Expert Group) commonly used method for lossy compression for digital images. The degree of compression can be adjusted.

UI: (User Interface) anything that a user can interact with. Point of contact between humans and computers.

MP3: A means of compressing a sound sequence into a very small file. Enables digital storage and transmission

WAV: an audio format standard for storing an audio bit stream.

2. General Description

2.1 Product / System Functions

HoneyMoon is designed to provide a minimalistic yet powerful set of tools for 2D game development, focusing on essential features to introduce new users to game development and simple programming. The key functions are categorised as follows:

2.1.1 Physics Simulation

HoneyMoon includes a simple physics engine to allow for basic interactions between different Game Objects, such as:

- **Collision Detection**: Enables game objects to interact with each other by detecting when they come into contact, where the boundaries for the collision can be customised.
- **Gravity and Force**: Supports configurable gravity and forces that allow users to create object movements, such as falling, jumping, and pushing.

- **Object Properties**: Allows customisation of physical properties like mass, friction, and bounce, so users can control the effects on movement and collisions.

2.1.2 Camera Control

HoneyMoon's Camera is designed to allow the user to control what appears on the screen with options such as:

- **Camera Follow**: A simple setting to allow the camera to follow a specific object, such as a player while maintaining a centred view.
- **Zoom and Pan**: Basic controls to adjust the camera's zoom level to explore different parts of the game world.
- **Screen Boundaries**: Ensures that the camera view stays within defined game boundaries or clamps to specific areas as needed.

2.1.3 Movement Control

Movement control in HoneyMoon focuses on giving users basic methods to make objects move and respond to player input:

- **Keyboard and Mouse Input**: Provides default bindings for common movement keys (e.g., arrow keys or WASD) and allows for basic mouse interaction.
- **Acceleration**: Allows users to set an object's acceleration to achieve smooth movement and experiment with different speed settings for smooth movement control.

2.1.4 Native Scripting Support

HoneyMoon includes a built-in scripting system, which provides:

- **Scripting Language**: A straightforward scripting language or simplified interface for scripting, designed for beginners.
- **Real-Time Testing and Debugging**: Enables users to modify and test scripts, providing immediate feedback on code changes.
- **API for Game Elements**: Basic functions and commands for manipulating game objects, handling events, and controlling game flow without needing advanced coding skills.

2.1.5 Design and Layout Tools

HoneyMoon provides visual tools for users to organise and arrange elements within their game, including:

- **Scene and Object Placement**: A simple, drag-and-drop interface to move around the Game Screen, zoom in and out feature to view more objects within the game screen, and rotate objects within the game scene.
- **Layer Management**: Supports basic layering, allowing users to organise objects visually and control their render order.
- **UI Elements**: Allows users to add and position simple user interface components such as buttons, text fields.

2.1.6 Asset Import and Management

HoneyMoon will include an asset manager to organise and use game assets:

- **2D Asset Support**: Accepts standard 2D image formats (e.g. PNG, JPEG) for sprites, backgrounds, and textures.
- **Audio Management**: Basic support for importing sound files (e.g., MP3, WAV) and managing audio playback for background music and sound effects.
- **File Management:** The application will support storing files within the editor like scripts for the native scripting and any texture files etc.

2.1.7 Animation and Sprite Management

HoneyMoon includes tools for creating and controlling animations, essential for bringing characters and scenes to life:

- **Sprite Sheet Integration**: Supports loading sprite sheets, allowing users to define animation sequences.
- **Event-Based Animation Triggers**: Provides a way to trigger animations based on specific in-game events.

2.2 User Characteristics and Objectives

User Community Characteristics

The primary users of the HoneyMoon Engine are aspiring game developers, hobbyists and instructors/teachers, including individuals with a basic understanding of programming and an interest in game development. These individuals may vary in experience from beginners with limited exposure to programming concepts to more experienced developers who want an

alternative to mainstream engines. Users may have some familiarity with scripting however users can expect to become comfortable with Lua on their game development journey.

Objectives and Requirements from the User's Perspective

From the user's perspective, HoneyMoon Engine should serve as a lightweight and user-friendly platform for building 2D games, with an emphasis on simplicity and ease of use. Key objectives include:

Ease of Scripting: Users want to focus on scripting game logic without dealing with complex backend code. By using Lua, the engine simplifies scripting tasks, allowing users to write simpler, more manageable code while leaving performance-intensive tasks to C++.

Core Game Features: The engine should support essential 2D game elements like object rendering, movement, collision detection, and physics without requiring complex setup. Users expect basic tools for these features so they can quickly prototype and develop game ideas.

Efficient User Interface: A streamlined interface for the game engine. ImGUI will allow users to access controls, debugging, and game object management in an intuitive environment.

Customization and Flexibility: Users appreciate the flexibility to add custom elements or extend the engine’s capabilities as their skills improve. The integration of Lua and C++ allows for customisation while maintaining a clear structure.

Learning Support: Since many users are novices or hobbyists, the engine will include documentation, code samples, and example projects to assist with learning game development fundamentals and Lua scripting.

Performance: While simplicity is the priority, users also expect the engine to perform well, particularly in terms of smooth rendering and quick processing of physics and collision.

2.3 Operational Scenarios

2.3.1 User Classes

2.3.1.1. Beginner Game Developer

- **Characteristics/Attributes:**
    - Educational Level: Likely some background in basic programming concepts, but limited experience in game development.
    - Age Range: 15 - 40
    - Experience with Game Development: Beginner, minimal experience. No formal training or complex technical knowledge.
    - Frequency of Use: High frequency initially, as they explore the engine and refine their first project. Expected to decrease after completion unless they begin a new project.
    - Requirements:
        - Simplified tools for creating and customising game objects, platforms, and collectibles.
        - A simple UI for placing objects, traversing the game screen and creating files or importing files.
        - Beginner-friendly physics and scripting options.
        - Easy-to-use testing and debugging features.
    - Importance: **Critical**. This user class represents a large portion of the potential audience, and the tools must allow for easy entry into game development.

2.3.1.2. Hobbyist

- **Characteristics/Attributes:**
    - Educational Level: No background or very limited in programming and game development, wants to get a feel for both to see whether they should pursue a career in IT or not.
    - Age Range: 10 - 20
    - Experience with Game Development: Limited, mostly self-directed learning, may have experience with simple scripting.
    - Frequency of Use: Regular usage as they experiment and create small projects. May shift to occasional use depending on interest level.
    - Requirements:
        - Flexible tools for scripting and experimentation with different tools like physics, cameras, UI.
        - Easy access to Lua scripting to allow coding in a beginner-friendly language.
        - Easy-to-use testing and debugging features.
    - Importance: **Moderate**. This user class benefits from hands-on learning and will likely offer feedback for improving beginner accessibility.

2.3.1.3. Instructor/Teacher

- **Characteristics/Attributes:**

- **Educational Level**: Educators or instructors with intermediate technical knowledge. Comfortable with basic scripting and game engine editors with knowledge of game development principles and techniques.
- **Age Range**: 25 - 55
- **Experience with Game Development**: Very familiar with game development concepts and programming principles.
- **Frequency of Use**: Moderate, with high-intensity usage depending on whether the module is heavy on game development or less intense for modules like computer science.
- **Requirements**:
  - Tools to create educational modules that demonstrate core game design concepts.
  - Visual aids and sample scripts for explaining code.
  - A reliable preview feature to check and demonstrate functionality to students.
  - Support for saving and exporting so students can also have their own projects to demonstrate.
- **Importance**: **Critical**. instructors shape beginner learners' experiences and expectations, where a friendly and simple processes will bring inspiration to students and bring educational success.

2.3.2 Use Cases

2.3.2.1: Creating a Simple Platformer Game

Use Case
A beginner user wants to create a basic platformer game where a character jumps between platforms and collects items.

Steps:

1. **Project Setup**:
   - The user launches HoneyMoon Engine, which opens with a clean workspace and options to create a new project. The user creates a new Project called “Platformer”
2. **Adding a Character**:
   - The user creates a GameObject which will load a default Game Object with a sprite (Texture being a PNG or JPG) attached and some default values for x cord, y cord, the height and width.
   - The user imports a custom sprite which changes the default sprite for the main character, a simple image of a character, which changes the appearance of the Game Object.

    - The user defines the character's initial position and sets the physics properties for movement and jumping.

3. **Building Platforms**:
    - The user adds More Game Objects which represent rectangular platforms to the scene, positioning them at varying heights and distances.
    - Physics properties are added for the platforms to ensure collision detection.
4. **Adding Collectible Items**:
    - The user imports and places collectible Game Object item sprites, such as coins, on various platforms.
    - Each item is assigned a script to disappear and increase the score when collected by the player character.
5. **Implementing Controls**:
    - The user creates a file for scripting in the editor.
    - The user opens the file which in turn opens a scripting editor i.e VisualStudio Code and uses HoneyMoon's scripting language Lua to define the character's movements.
    - Input keys are bound to movement (e.g., arrow keys or AWSD).
6. **Testing the Program**:
    - The user presses the button "Preview" to open another window which displays the game which was created in the editor.
    - Once the User is happy with their Testing they Press the "Stop" button in the editor to shut down the preview Window.
    - The user then makes adjustments to platform spacing, character speed, and jump height as needed.
7. **Saving and Exporting**:
    - Once satisfied, the user saves the project and exports it as a playable application.

Expected Outcome
The user successfully creates a playable platformer level where the character jumps between platforms, collects items, and scores points which displays using the games UI. HoneyMoon allows the user to experience the complete development process, from initial setup to final testing, in a beginner-friendly way.

2.3.2.2: Designing an Interactive Learning Module

Use Case
An instructor wants to create an interactive learning module for students, where they will learn game development concepts.

Steps:

1. **Project Setup**:
    - The user starts a new project and names it "Class102 Test"
2. **Creating GameObjects:**
    - The user creates multiple gameObjects to show different types of adjustments that can be applied and what effects they carry.
3. **Importing**:
    - The user imports some files for the different textures and scripting files that are used to explain the layout and structure of the file manager.
4. **Scripting:**
    - The user writes some simple programming script to go through and explains the layout and structure.
5. **Testing the Program**:
    - The user enters “Preview” and observes the output of what was programmed by them, checking the interaction between the Game Object and the environment.
    - Once the user is satisfied they press the “Stop” button to stop the preview and go back to the editor to make any necessary changes.
6. **Saving and Exporting**:
    - Once satisfied, the user saves the project and exports it as a playable application.

Expected Outcome
Students can engage with the module, adjusting properties and seeing how changes affect object behaviour in real-time, enhancing their understanding of basic game development concepts. HoneyMoon’s interactive design tools allow the instructor to create an educational experience with minimal setup.

2.3.2.3: Experimenting with Programming using Game Development as a way for interactive learning

Use Case
A hobbyist user wants to experiment with programming but wants to do something fun at the same time by creating a game for themselves.

Steps:

1. **Project Setup**:
    - The user starts a new project and names it "My First Project."
2. **Creating The Environment:**
    - The user creates some gameObjects.
    - The user then adds some collision to at least two gameObjects for experimentation.

    - The user also adds a camera Object to follow a gameObject.
3. **Writing the Script:**
    - The user creates a Lua file in the file manager
    - They Open the file with any preferred IDE and write some Lua code and experiment with the different functionalities e.g. Physics.
4. **Testing the Program**:
    - The user enters “Preview” and observes the output of what was programmed by them, checking the interaction between the Game Object and the environment.
    - Once the user is satisfied they press the “Stop” button in the editor to make further adjustments to improve their code.
5. **Recording the Results**:
    - The user can record or capture screenshots to analyse the results of different scripting methods.
6. **Saving and Exporting**:
    - Once satisfied, the user saves the project and exports it as a playable application.

Expected Outcome
The user gains hands-on experience with programming without having to learn the in and outs of the game Engine and can go straight into practise with some minor set up. HoneyMoon allows the user to experiment and learn programming principles without requiring advanced skills.

2.3.2.4: Building a Simple Puzzle Game

Use Case
A user wants to create a basic puzzle game where players must arrange objects to complete a shape.

Steps:

1. **Setting Up the Project**: The user creates a project named “Shape Puzzle.”
2. **Adding Puzzle Pieces**:
    - The user imports images of puzzle pieces and arranges them randomly in the scene.
    - Each piece is given properties to enable snapping into place when positioned correctly by manipulating the x and y coordinates and taking advantage of mouse input.
3. **Defining Puzzle Logic**:
    - The user writes a simple script for each piece to check its position relative to the goal and snap into place if close enough.
4. **Creating Win Conditions**:

    - The user defines a win condition that triggers once all pieces are in place, showing a “You Win!” message.
5. **Testing the Program**:
    - The user enters “Preview” to try the puzzle, ensuring that pieces snap correctly and that the win condition activates, once finished to stop the Preview the user presses the “Stop” button in the editor.
6. **Saving and Exporting**:
    - Once satisfied, the user saves the project and exports it as a playable application.

Expected Outcome
The user creates a fully functional puzzle game with clear objectives and win conditions. HoneyMoon’s design tools and scripting make it easy for users to build and test logic-based gameplay elements.

2.3.3 Basic Use Case Diagram

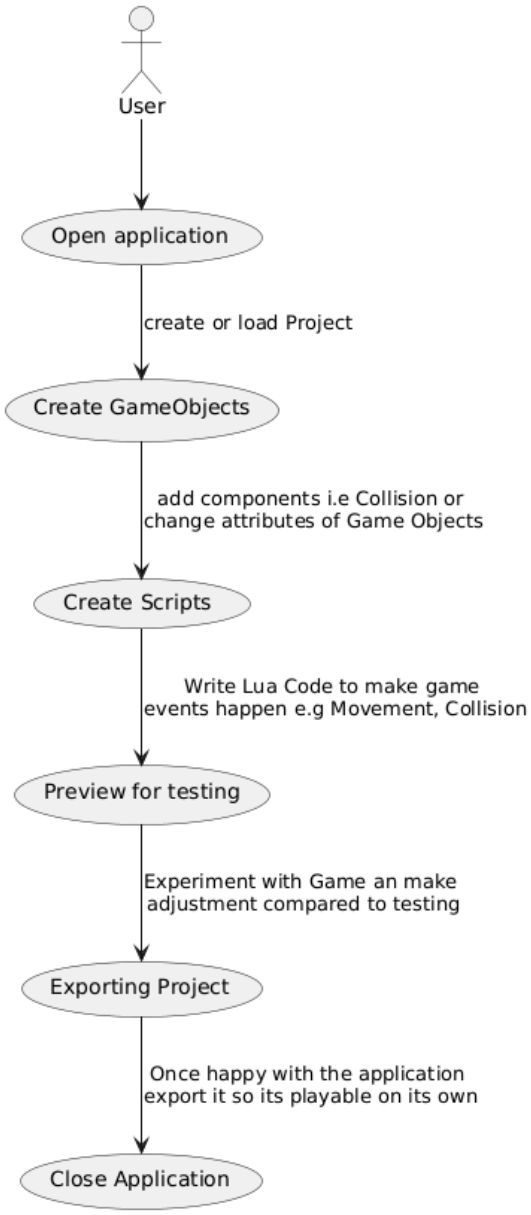


Figure 1: User interaction flowchart for the HoneyMoon Engine, depicting the steps from Starting the application to closing the application.

2.4 Constraints

Performance Requirements:

- The HoneyMoon Engine should provide a responsive user experience, ensuring smooth object rendering, movement, and basic physics handling. Lua scripts, which enable users to control game logic, must execute quickly and efficiently through the C++ backend to avoid any noticeable lag. The engine's C++ foundation should handle intensive processing tasks to keep user-facing interactions smooth and seamless.

Platform Compatibility:

- The engine must be compatible with a range of modern computers and laptops, ensuring that it can run as an executable on a user's local machine without requiring advanced or specialised hardware. Development choices should support scalability, so the engine remains functional across various system configurations with the existence of a graphics card within these systems.

Resource Management:

- The engine must carefully manage memory and processing resources to prevent excessive resource consumption, especially during rendering and scripting tasks. Resource management practices should include efficient handling of object creation and deletion, minimal memory footprint for the game engine UI, and optimised handling of object hierarchies to ensure the tree structure for game objects performs reliably.

User Interface and Experience:

- The UI, built with ImGUI, should be intuitive and responsive, allowing users to navigate, script, and debug without unnecessary complexity. The design must encompass users new to game engines, offering tooltips, explanations, and guidance while maintaining a professional look and feel.

3. Functional Requirements

3.1.1 Physics Simulation

- **Description:** Enable collision detection, gravity, and customisation of Game Object physical properties. Users can define how objects respond to different forces.
- **Criticality: High**. Core to gameplay mechanics and interactions especially the collision between Game Objects.
- **Technical Issues**: Accurate collision handling without significant lag.
- **Dependencies:** Requires correct rendering with events.

3.1.2 Camera Control

- **Description:** Users can configure camera size and have the option to follow a specific Game Objects and the ability to zoom in and out.
- **Criticality: Medium**. Supports gameplay navigation and cuts rendering cost to boost performance of the application.
- **Technical Issues:** Ensuring smooth transitions and compatibility with various screen sizes.
- **Dependencies:** Relies on object rendering.

3.1.3 Movement Control

- **Description:** Provides default keyboard and mouse controls for moving objects.
- **Criticality: High**. Essential for character and object movement to be able to perform simple events in the game as walking, jumping.
- **Technical Issues:** Input lag minimization, compatibility with native scripting and translating complex C++ functionality to simpler Lua functions.
- **Dependencies:** Requires physics and scripting systems.

3.1.4 Native Scripting Support

- **Description:** Allows users to script game logic in Lua, offering real-time testing, debugging, and allows for simpler programming than more complex languages like Java or C++.
- **Criticality: High**. Fundamental to allow for user creativity for scripting specific events.
- **Technical Issues:** Learning to use the LuaCpp API and embedding it into the application.
- **Dependencies:** Interfaces with all tools in the game engine from game Objects, movement, physics, and UI components.

3.1.5 Design and Layout Tools

- **Description:** Visual tools for arranging objects, layering, and adding UI elements.
- **Criticality: Medium**. Enhances user experience in organising game scenes and allows for simple use.
- **Technical Issues:** Efficient resource handling for UI rendering.
- **Dependencies:** Interfaces with everything in the editor from asset management, file management, camera, and object rendering.

3.1.6 Asset Import and Management

- **Description:** Users can import images, audio files and scripts. Able to manage assets, and organise them in the editor.
- **Criticality: High**. Enables customisation and resource management and provides simple traversal of files and editing for scripts.
- **Technical Issues:** File format compatibility, asset loading without significant memory impact and loading different files and folders in a Tree Structure.
- **Dependencies:** Supports animation, design tools, and scripting.

3.1.7 Sprite Management

- **Description:** ProvidesThe user with managing and having different textures for Game Objects.
- **Criticality: High**. Core to character customisation and having unique game Objects.
- **Technical Issues:** Loading and rendering the textures without significant memory impact.
- **Dependencies:** Relies on object rendering.

3.1.8 Animation Support

- **Description:** Provides sprite sheet support and event-based animation triggers, allowing animation control through Lua scripting.
- **Criticality: Low**. allows for more specific and engaging experience for already in place sprites that need some life.
- **Technical Issues:** Optimising sprite sheet loading and frame transitions.
- **Dependencies:** Relies on sprite management, scripting, and object rendering.

4. System Architecture

The HoneyMoon Engine is structured around modular components that interact to deliver a functional and flexible 2D game engine. This architecture distributes core functions across specific system modules, each handling distinct aspects of game development, from rendering to scripting. Below is an overview of these modules and their roles, including notable third-party components used within each.

4.1 Core Engine Module (C++ Backend)

- **Function**: This is the backbone of HoneyMoon Engine, written in C++ to manage the core game functions, including object creation, scene management, and physics. It is responsible for handling processing tasks and coordinating other modules.
- **Components**:
    - **Scene Management**: Organises game objects within a tree structure, allowing hierarchical relationships (e.g., child objects under a parent object).
    - **Physics Engine**: Manages basic physics calculations like movement and collision detection..

    - **Camera Control**: Provides functions to move and control in-game cameras.
- **Key Reused/3rd Party Components**: None directly in this core module.

4.2 Rendering Module (SDL)

- **Function**: Handles all graphics and rendering tasks, enabling display of game objects and environments. The module uses SDL (Simple DirectMedia Layer) for rendering 2D graphics and handling window events.
- **Components**:
    - **Sprite Renderer**: Displays 2D sprites and textures within the game environment, allowing objects to be visually represented on-screen.
    - **Animation Renderer**: Supports basic sprite animations, where frames of an animated object are rendered in sequence.
    - **Window and Input Manager**: Manages the game window, and processes user input (keyboard and mouse) to interact with game objects.
- **Key Reused/3rd Party Components**: SDL library for cross-platform graphics rendering and window handling.

4.3 Scripting Module (Lua Integration)

- **Function**: Enables native scripting within HoneyMoon Engine by integrating Lua as the primary scripting language for game logic. This module allows users to write custom behaviours and interactions without directly modifying the core C++ code.
- **Components**:
    - **Lua Interpreter**: Executes Lua scripts written by users, enabling custom game logic for object interactions, animations, and events.
    - **Lua-C++ Interface**: Uses Lua C++, a library that facilitates communication between Lua and C++. This interface allows Lua scripts to interact with the C++ core.
- **Key Reused/3rd Party Components**: Lua scripting language and Lua C++ API for integration of Lua with C++.

4.4 User Interface (UI) Module (ImGUI)

- **Function**: Provides a graphical interface for developers to interact with the game engine, manage game objects, and access development tools. This module is built using ImGUI.
- **Components**:
    - **Object Inspector**: Allows users to view and edit properties of game objects, such as position, scale, and behaviour scripts.

    - **Hierarchy and Scene Viewer**: Displays a tree structure of game objects, showing parent-child relationships to help organise the game's scene structure.
    - **Debugging Console**: Shows real-time logs, errors, and feedback, aiding users in debugging.
- **Key Reused/3rd Party Components**: ImGUI

4.5 Resource Management Module

- **Function**: Handles loading, storing, and managing game assets such as textures, scripts, sounds, and object data.
- **Components**:
    - **Asset Loader**: Loads assets like sprites, textures, and sounds, manages them within the engine.
    - **Resource Pool**: Maintains a repository of loaded assets to improve memory efficiency.
- **Key Reused/3rd Party Components**: Open-source libraries for JSON/XML handling as necessary for data serialisation and asset management.

4.6 Version Control and Collaboration Module

- **Function**: While not part of the game engine itself, version control is crucial for development and collaboration. We will use Git for version control to manage codebase changes and Jira for task tracking.
- **Components**:
    - **Git**: Manages code versions, allowing developers to contribute without conflicts.
    - **Jira Integration**: Tracks project tasks and bugs, ensuring agile practices and iterative improvements.

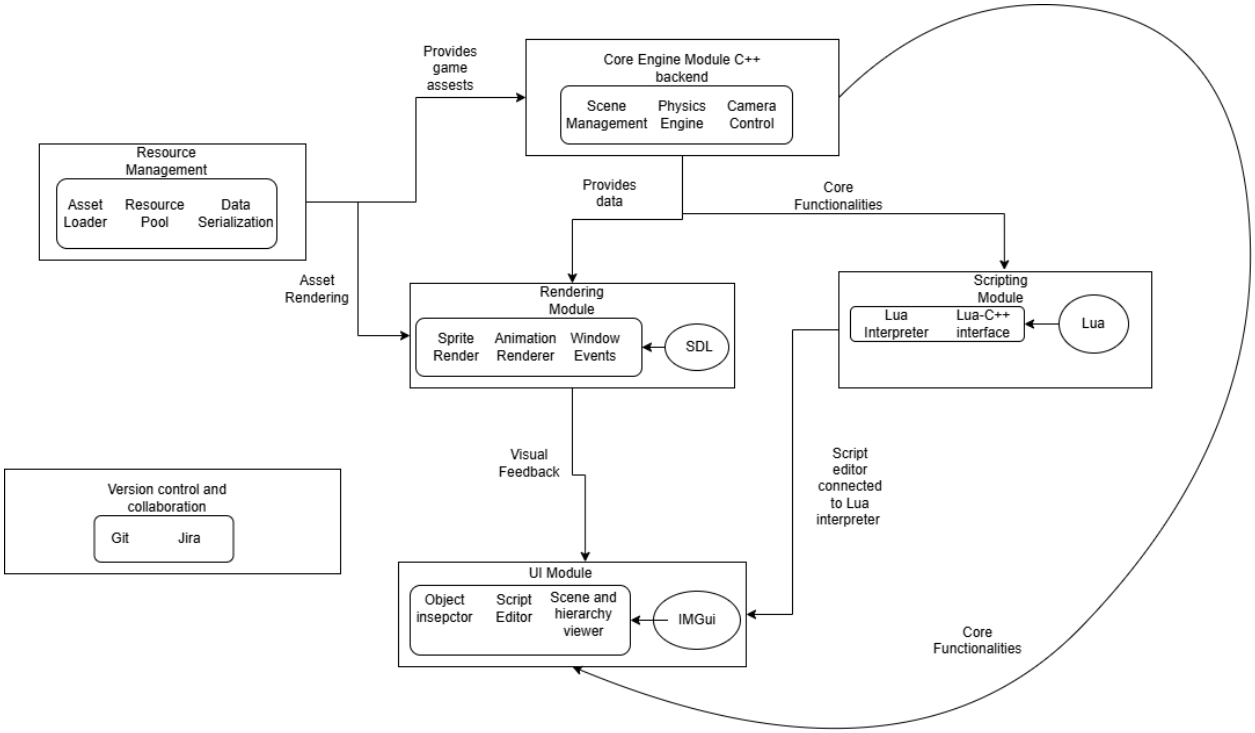
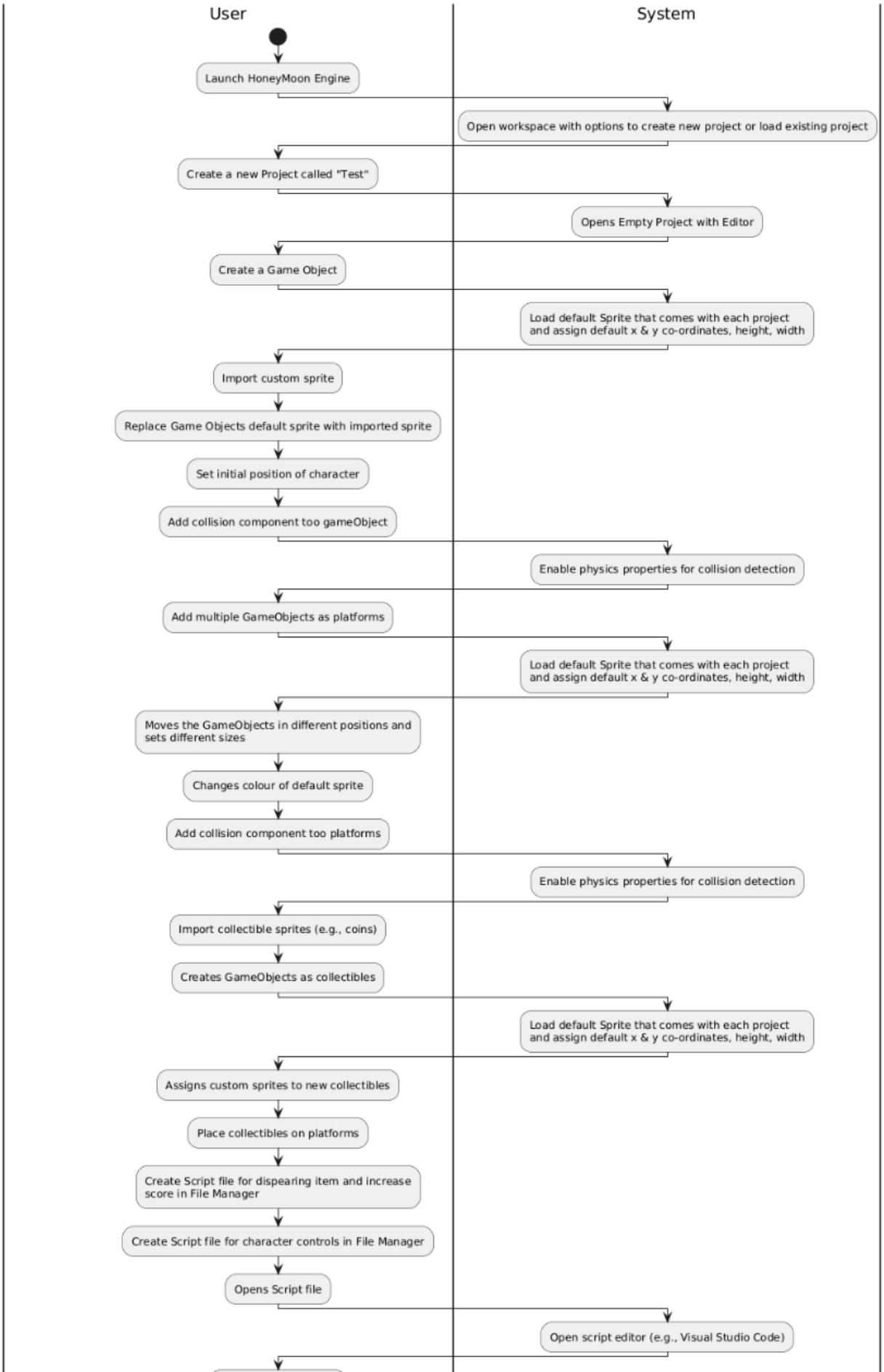


Figure 2: High level System Architecture Diagram for HoneyMoon Engine

5. High Level Design


User
System
Launch HoneyMoon Engine
Open workspace with options to create new project or load existing project
Create a new Project called "Test"
Opens Empty Project with Editor
Create a Game Object
Load default Sprite that comes with each project and assign default x & y co-ordinates, height, width
Import custom sprite
Replace Game Objects default sprite with imported sprite
Set initial position of character
Add collision component too gameObject
Enable physics properties for collision detection
Add multiple GameObjects as platforms
Load default Sprite that comes with each project and assign default x & y co-ordinates, height, width
Moves the GameObjects in different positions and sets different sizes
Changes colour of default sprite
Add collision component too platforms
Enable physics properties for collision detection
Import collectible sprites (e.g., coins)
Creates GameObjects as collectibles
Load default Sprite that comes with each project and assign default x & y co-ordinates, height, width
Assigns custom sprites to new collectibles
Place collectibles on platforms
Create Script file for dispearing item and increase score in File Manager
Create Script file for character controls in File Manager
Opens Script file
Open script editor (e.g., Visual Studio Code)

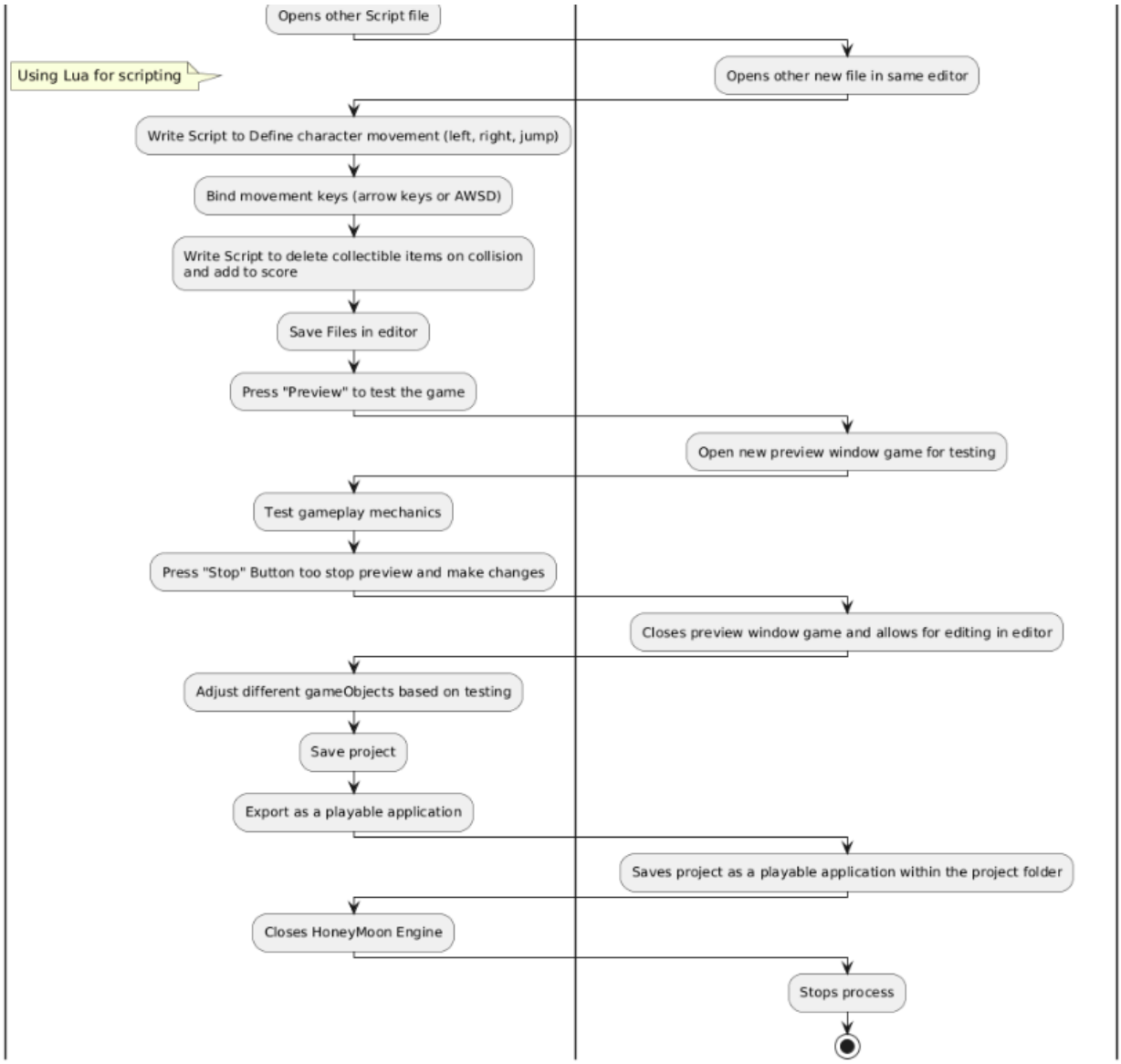


Figure 3: DFD model showing the process of the User and System

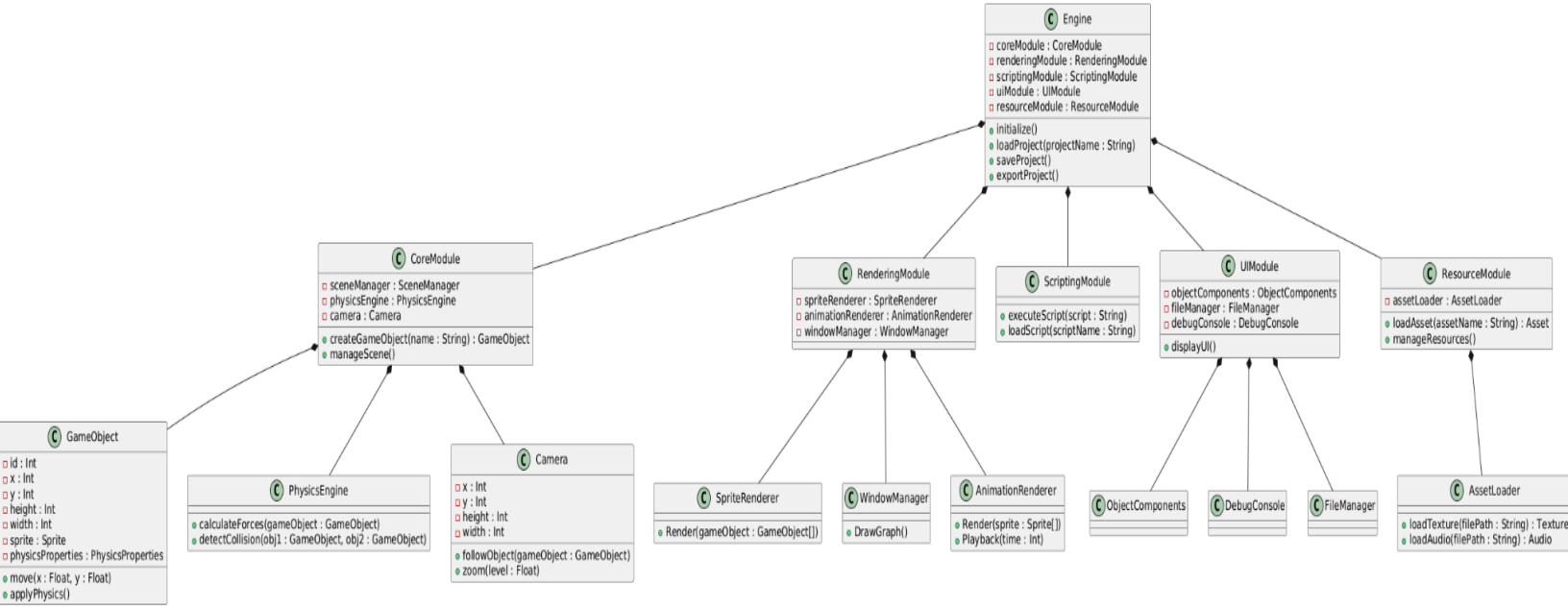


Figure 4: Object Model Showing the different Components within the System

6. Preliminary Schedule

Our preliminary schedule for the HoneyMoon Engine will outline major tasks, dependencies, and resource requirements. Our timeline runs from September 2024 to March 2025.

September: We will focus on establishing project fundamentals by identifying a supervisor and drafting a project proposal. During this we will initialise all required libraries and begin creation of “Gamescreen” and “GameObject” classes which will bring us into late September.

October: With the approval of our project we will shift into creating our foundational components, being a second application screen, game object class and a basic UI. We will also begin creation of our functional specification to be submitted in November.

November: Finishing the functional spec in early November and diverting most of our focus into creating a filesystem as well as finishing our organising of game objects into a hierarchical relationship.

December: Focus will be on core functionalities, particularly implementing collision detection and camera controls. This period will be adjusted based on our SEM1 examinations as well as Christmas/New Year break.

January: Continuing development on physics functions, refining camera capabilities, and updating the UI. By this stage, core elements of the engine should be in place.

February: Bug fixes, performance enhancements, and overall stability improvements will be prioritised. Allows time for user and unit testing. Depending on how functional the system is based on our goals we may add new functionality or simply focus on refining what has been done.

March: Extension of February with a focus on ensuring the engine is running to our desired state.

Resource requirements include a modern computer or laptop capable of running development tools and handling rendering tasks.

The software stack will feature Visual Studio Code as the IDE, SDL for rendering, ImGUI for UI, and Lua C++ for Lua integration.

Git will be used for version control, and Jira will facilitate task tracking as well as agile practices.

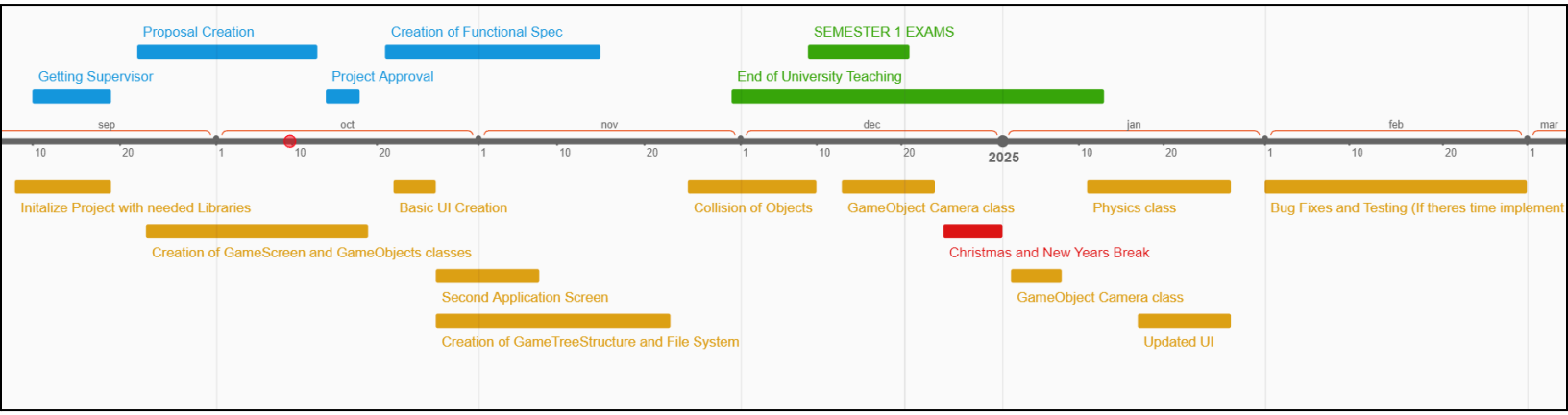


Figure 5: Preliminary Schedule for the Project

7. Appendices

Websites used for creating Diagrams:

"Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams.," *PlantUML.com*. https://plantuml.com/

draw.io, “Flowchart Maker & Online Diagram Software,” *app.diagrams.net*, 2024. https://app.diagrams.net/