Coding a 2D Game Engine for ESP32: A Showcase of Design Patterns

David Jean Raymond Gianadda

Haaga-Helia University of Applied Sciences
Bachelor of Business Information Technology
Product type thesis
2024

Abstract

Author(s)
David Jean Raymond Gianadda

Degree
Bachelor of Business Information Technology

Report/Thesis Title
Coding a 2D Game Engine for ESP32: A Showcase of Design Patterns

Number of pages and appendix pages
52 + 1

This product thesis documents the development of a 2D game engine for the ESP32 microcontroller platform. The aim of the thesis is to provide a concrete and documented example of design patterns implementation, serving as a demonstration of competency for the author and as an educational tool for the reader. The thesis is structured into an introduction, theoretical framework, practical part, and discussions chapters.

The introduction provides context for the project and outlines its scope, targeting readers interested in design patterns implementation in game development and embedded systems programming. The theoretical framework covers essential concepts such as object-oriented programming, design patterns, and clean coding principles, laying the groundwork for the practical development phase.

The practical part details the iterative process of building the game engine, focusing on key components such as input management, scene management, and core architecture. Design patterns such as Singleton, Composite, and State are integrated into the engine's architecture to enhance modularity and scalability.

In the discussions chapter, potential future features implementations are explored and the use of artificial intelligence during the thesis work is explained. Sustainability topics are discussed, and the conclusion of the work is provided.

Overall, this thesis provides a comprehensive exploration of game engine development on the ESP32 platform, with a focus on clean coding practices and the use of design patterns.

Key words
Design Pattern, Game Engine, Object-Oriented Programming (OOP), Microcontrollers, ESP32, C++

Table of contents

1 Introduction

Software development is the art of writing instructions for machines to create tools for humans. Websites, operating systems, artificial intelligences, information systems, 3D sculpting tools and video games, all software is the result of collaborative effort and creative ingenuity. Software involves continuous refinement and evolution. Over time, the need for maintainable and reusable code has driven the convergent evolution of various techniques and best practices. These advancements ensure that software can adapt to changing requirements and remain robust and effective throughout its lifecycle. This paper documents the use of those techniques and best practices in the implementation of a 2D game engine.

1.1 Scope

This thesis concludes a three-year adventure of learning software development. As a final bachelor project, its first goal is to demonstrate proficiency in software architecture by creating a well-structured and efficient 2D game engine. It also aims to showcase the practical use of design patterns in software development through their implementation in a game engine. And finally, it provides personal learning opportunities in topics such as embedded development, Internet of Things, and the C++ language.

The target reader should be familiar with software development, as the practical part of this report is focused on how to write code. The product and code examples are written in C++. Unified Modeling Language (UML) notation is also used to visualize code architecture. The concepts specifically related to game development and embedded software development are explained when relevant, so there is no need for prior knowledge in those domains to understand the thesis.

Readers who would benefit the most from this report are software development students familiar with the Object-Oriented Programming (OOP) paradigm and interested in design patterns. Hobbyist game developers interested in game engine architecture, as well as embedded software development enthusiasts looking for insights into the OOP approach in embedded systems, will also find this report valuable. In general, any developer that is not yet familiar with design patterns or who is looking for a practical example of their implementation will gain valuable information in this report.

1.2 Outcomes

The first and main outcome of the thesis is the product itself. A game engine for ESP32, in the form of a GitHub repository. This game engine will provide the boiler plate code that can be used as a base to write games for ESP32. Note that making a graphical user interface for users to interact with the game engine is out of scope. Instead, this game engine can be seen as a framework, or a base project, providing the basic architecture and tools for coding games.

The second outcome is a gaming device built from an ESP32. To develop and test the game engine, a functional gaming device is required. The components that were used and how the device was assembled are explained in the practical part of the thesis.

Finally, some basic games are written with the game engine. The games help by providing a guideline on the feature to implement in the game engine. They are also useful for visually demonstrating the progress of the project and for testing the implemented features. The demonstration games are not meant to offer groundbreaking gameplay experience, instead they can be seen as proof of concept of the game engine's capabilities.

1.3 Structure and Format

This report is split into four sections. The introduction explains the context of the thesis topic, its scope and deliverables, and the structure of the report. Then the theoretical framework summarizes the key concepts of software development that are used in the making of the product. The practical part presents the code writing process with a focus on the implementation of design patterns. It follows the structure of the product's GitHub repository branches, so that the reader can see how the code evolved during the project's iterations. Finally, the discussions chapter reflects on the results of the product and makes a comparison with the initial expectations. It is also used to discuss topics such as the use of artificial intelligence (AI) during the project and broader topics like sustainability and future iterations of the project. The report ends with the list of sources used and its appendix.

This report is formatted according to Haaga-Helia's guidelines for long reports and thesis (Thesis Coordinators 2022, 1-21). Figures and tables are used to illustrate the text. Two types of figures can be found: images and code snippets. The code snippets are formatted with the Easy Code Formatter addon for Microsoft word. Due to the frequent mention of code in the text, special formatting is used when classes, objects, methods, functions, and libraries are mentioned. This formatting is Arial 11 Bold and Italic.

2 Theoretical Framework

The theoretical framework groups all the knowledge used to produce the thesis work. It contains software development concepts, introduction to embedded software development, and research on similar products.

Chapters 2.1 to 2.3 contain all the notions about software development that will be referred to during the creation of the game engine. Each concept has been concisely summarized to serve as a refresher for the target readers. If a concept is new to the reader, or requires a deeper explanation, it is suggested to check the sources mentioned. Note that those concepts are not specific to any language, but their implementation may vary from one language to another. For that reason, some additional information about the implementation in C++ is included when relevant.

Chapter 2.4 serves as an introduction to the concept of game engines. It investigates the origins of the term and the many definitions it has. This chapter ends with the definition of game engine used during this project.

Chapter 2.5 explains what the ESP32 is. It also provides an introduction to the requirements and tools used for embedded software development.

And finally, chapter 2.6 is research on previous work done in similar topics. It includes a review of existing similar projects, libraries that may be used in the development of the game engine, and previous work done by the author that is relevant to this project.

2.1 Object Oriented Programming

Object Oriented Programming (OOP) is the foundation of this thesis. It is a software development paradigm based on the use of objects (mdn web docs 2023). The wide adoption of OOP comes from its capability to improve code modularity, reusability, and maintainability by organizing software design around objects (Herdwaria 2023).

To gauge the popularity of the OOP paradigm we can look at the study made by Kristoffer Gunnarsson and Olivia Herber, “The Most Popular Programming Languages of GitHub's Trending Repositories”. In their paper’s conclusions, the authors mention four programming languages as being represented in both their and other similar studies about popular languages. Those are Java, Python, JavaScript, and C++, all of which are OOP languages. (Gunnarsson & Herber 2020, 24)

The following sub-chapters will recap and summarize the key concepts of OOP. The main source for those sub-chapters is “The Object-Oriented Thought Process” book by Matt Weisfeld, author, college professor and software developer. Secondary sources include popular software development learning websites such as W3 schools, CodeAcademy, and GeeksForGeeks, that were used for the concise definitions they offered.

2.1.1 Objects and Classes

As explained by Weisfeld, objects are the fundamental elements of OOP. An object contains data and behaviors. The data elements are referred to as attributes and the behaviors as methods. Objects are defined by classes. A class is the blueprint of an object, it is required to have a class defined to create an instance of an object. (Weisfeld 2019, chapter 1)

2.1.2 Encapsulation and Data Hiding

The process of combining data and behaviors into objects is called encapsulation. Weisfeld explains that the benefit of encapsulation is the control of access to data and behaviors via the concept of Data Hiding. This control is used to ensure that objects only make available to other objects what is relevant for their use. (Weisfeld 2019, chapter 1)

W3 Schools explains that in C++, access control is enforced via access specifiers. The keyword `public` specifies that the attribute or method can be accessed or called from outside the class. The keyword `private` specifies that the attribute or method can only be accessed within the class. (W3 Schools s.a.)

2.1.3 Inheritance

In OOP it is possible to reuse a class and add new attributes or new methods to it, as mentioned by the Codecademy Team. This is Inheritance. Inheritance is used when multiple classes share some common parts but also need their specific attributes and methods. The class that is being inherited is the parent class or the base class. The class that inherits is the child class or subclass. (Codeacademy Team 2023)

C++ supports multiple inheritance, meaning that a class can inherit from more than a single parent class (Weisfeld 2019, chapter 1).

2.1.4 Polymorphism

Polymorphism is closely related to inheritance, as it occurs when a group of subclasses inherit from the same parent class. In their article for GeeksForGeeks, MKS075 explains that while inheritance allows subclasses to expand on the base class, polymorphism delegates the implementation of methods to the subclass, allowing objects of different classes to be treated interchangeably (MKS075 2024).

2.1.5 Composition

It was established that a class is composed of data as attributes and behavior as methods. Those attributes can themselves be objects. This process of assembling objects to create new objects is called composition. (Weisfeld 2019, chapter 1)

2.2 Design Patterns

> “By definition, Design Patterns are reusable solutions to commonly occuring problems(in the context of software design). Design patterns were started as best practices that were applied again and again to similar problems encountered in different contexts. They become popular after they were collected, in a formalized form, in the Gang Of Four book in 1994.” (OODesign s.a.)

The official title of the book mentioned in the quotation from OODesign is “Design Patterns: Elements of Reusable Object-Oriented Software”, a staple in software development literature. The authors, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, collectively known as the Gang Of Four, were awarded the prestigious AITO Dahl-Nygaard Prize in 2006 for the profound impact of their work (AITO 2022). This book is used as the main source for this chapter. The Refactoring Guru is also cited multiple times. It is a website dedicated to helping software developers write better code through the use of refactoring, design patterns and SOLID principles (Refactoring Guru s.a. c).

Design Patterns are based on the common empirical experience of all software developers. As the Gang Of Four explains it, using this knowledge allows developers to design systems faster and avoids the need to come up with new, untested, designs. Design Patterns, since they have been precisely defined, also improve the quality of communication between developers, removing the need to explain the implementation of the code as the name and the functioning of the pattern is known by every party. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 1)

Gamma, Helm, Johnson, & Vlissides say that a pattern is defined by its name, the problem it solves, the solution, and the consequences or results of the pattern’s implementation. Patterns are also classified by purpose. The three types of purpose are creational, structural, or behavioral. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 1) The following sub-chapters present all the patterns used in the project.

2.2.1 Composite Pattern

The Gang Of Four defined the Composite pattern as a structural design pattern used to group objects into tree structures made of nodes and leaves. The point of the pattern is to allow clients to interact with a node or with a single leaf in the same manner. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 4) This pattern is built on the OOP concepts of polymorphism and composition. Polymorphism is used so that clients can interact with different classes interchangeably, such as nodes and leaves. And composition is used to group objects, as is done when a node contains leaves or other nodes. What the Composite pattern does is combine those two principles by enabling the client's commands to be passed down the tree structure recursively (Gamma, Helm, Johnson, & Vlissides 1994, chapter 4).

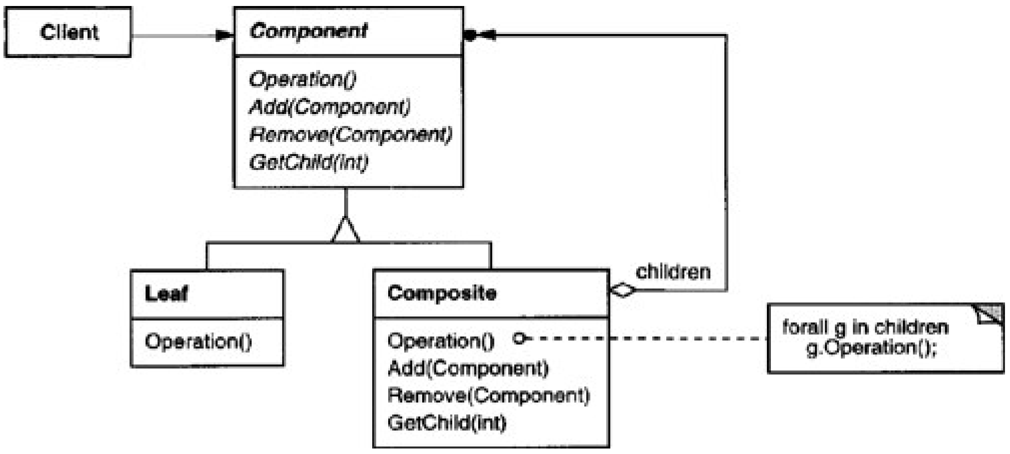


Figure 1. Composite pattern (Gamma, Helm, Johnson, & Vlissides 1994, chapter 4)

Figure 1 shows the class diagram of the Composite pattern. Both the ***Leaf*** and the ***Composite*** implement the ***Component*** interface. The ***Operation()*** method is a behavior that can be called by the client. When the ***Operation()*** is called on a composite node, the node will then call the ***Operation()*** on all of its children. Not that in figure 1, the Component is shown to specify not only the ***Operation()*** method but also all the children component management method that are used to by composites to handle the addition or the removal of other composites and leaves. The design patterns are guidelines or templates on how to structure code. Their implementations may vary according to the context. For instance, the Refactoring Guru suggests defining a component interface that contains only the methods used by both leaves and nodes (Refactoring Guru s.a. a). This can be seen in

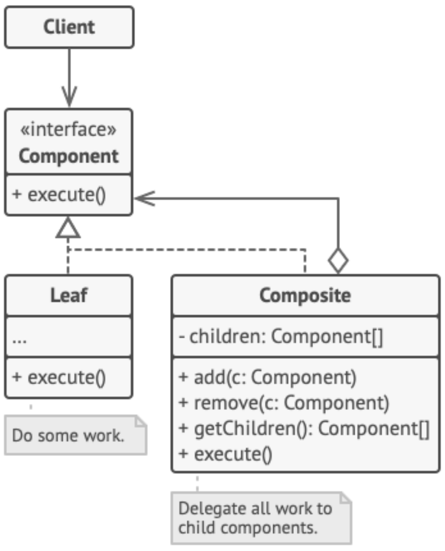


figure 2.

Figure 2. Composite Pattern from the Refactoring Guru (Refactoring Guru s.a. b)

2.2.2 Singleton Pattern

The Singleton pattern is explained to be a creational design pattern. It is used to enforce the existence of only one instance of a class in the code. It also makes this class accessible anywhere in the code. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 3)

As shown in figure 3, the Singleton itself is the only class found in this pattern. It is specified that the Singleton is in charge of creating itself and making itself available to the rest of the code. This can be achieved by making the constructor method not public. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 3)

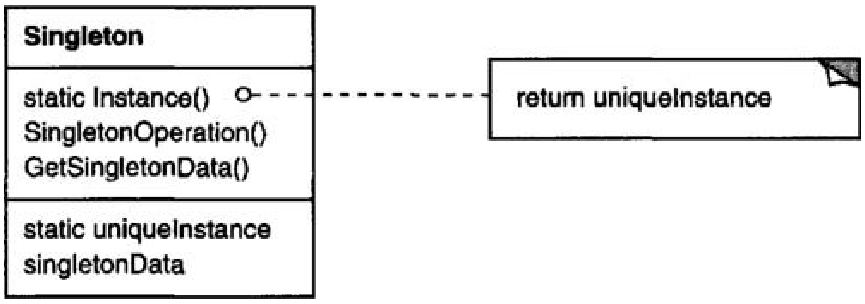


Figure 3. Singleton pattern (Gamma, Helm, Johnson, & Vlissides 1994, chapter 4)

2.2.3 State Pattern

The Gang Of Four presents the State design pattern as a behavioral design pattern used to change the state of an object. There are two main scenarios that would require a change of state in an object. The first is when the behavior of the object changes according to its state. The State pattern allows for such changes at runtime. The second is when complex conditional statements are involved. By encapsulating each state in a separate class, the State pattern treats each state as an independent object, allowing for flexible and maintainable state changes. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 5)

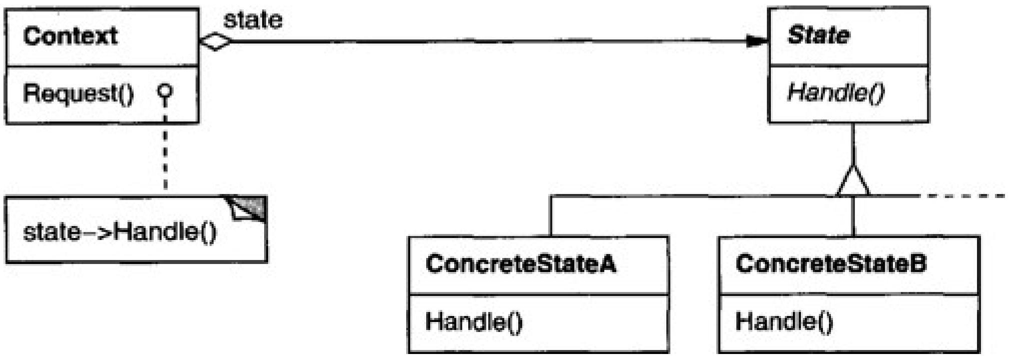


Figure 4. State pattern (Gamma, Helm, Johnson, & Vlissides 1994, chapter 5)

For example, given a ***Calculator*** class, the context, with a request method, ***calculate()***, that takes two numbers as a parameter. The ***Calculator*** could have states to represent add, subtract, multiply and divide operations. The ***Calculator*** has a single instance of a state for each of the operations and only one of this state is its current state. When the ***Calculator*** method ***calculate()*** is called, it will in turn call a handle method in its current state as can be seen in figure 4.

2.2.4 Observer Pattern

The Gang of Four defines the Observer pattern as a behavioral design pattern used for scenarios where changes in one object need to be propagated to other objects without tightly coupling them. This approach ensures flexibility and maintainability of the code. (Gamma, Helm, Johnson, & Vlissides 1994, chapter 5)

The ***Observer*** pattern consists of two main types of participants: the ***Subject*** and the ***Observer***. The ***Subject*** maintains a list of its ***Observer***, also known as dependents, and provides helper methods for handling ***Observer*** instances. The ***Observer*** defines an updating interface that is called by the ***Subject*** whenever a change in the ***Subject***'s state occurs. This pattern uses the concept of loose coupling, to allow the ***Subject*** and ***Observer*** to vary independently. (Figure 5.)

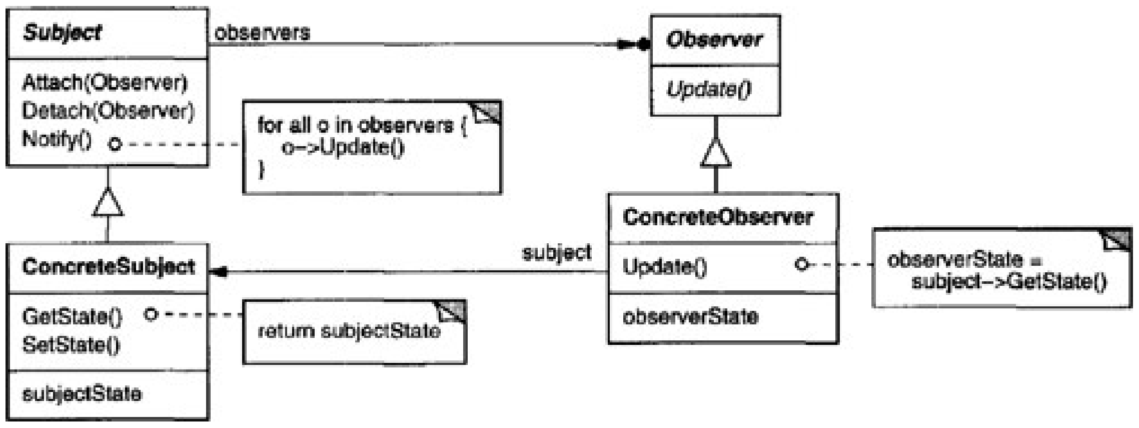


Figure 5. Observer pattern (Gamma, Helm, Johnson, & Vlissides 1994, chapter 5)

2.3 Clean Coding

> “Clean code is a term used to refer to code that is easy to read, understand, and maintain. It was made popular by Robert Cecil Martin, also known as Uncle Bob, who wrote "Clean Code: A Handbook of Agile Software Craftsmanship" in 2008. In this book, he presented a set of principles and best practices for writing clean code, such as using meaningful names, short functions, clear comments, and consistent formatting.” (Codacy 2023)

In the “Clean Code: A Handbook of Agile Software Craftsmanship" Robert Cecil Martin gather the definition of clean code from renown programmers. While they all accentuate different aspects of clean code, two main approaches emerge. The first is strictly about the code, how it should be split, and what it should accomplish. The second approach is about the developers and how the code should be written to be read by humans. (Martin 2008, chapter 1) Both aspects are expended upon in the following sub-chapters.

2.3.1 Readable Code

According to Martin, when working on a program, be it for implementing a new feature or to fix a bug, more time is spent reading than writing. And for that reason, they advise to consider the readability of the code by humans to ensure the maintainability of the code. (Martin 2008, chapter 1) This chapter summarizes some of the key concepts defined in Martin’s book.

In their book, Martin also says that writing readable code implies meaningful and descriptive names for variables, functions, and classes. Good naming conventions help convey the purpose of a code element without needing additional comments. Names should be precise, avoiding ambiguity and providing clear context. (Martin 2008, chapter 2) For instance, a variable name ***elapsedTime*** is more informative than a vague name like ***t***.

Comments are blocks of text written in the code but that are not compiled. They are used by developers to explain and document their code. In Martin’s book, comments are said to be another crucial element for readability. They should be concise, relevant, and kept up to date with any code changes. Old comments should be removed to avoid confusion, and so do code snippets that are commented out. (Martin 2008, chapter 4)

Formatting is how the code is arranged in its file. Elements of formatting include indentation, spacing, line breaks, and the organization of code into logical sections. Martin explains that adopting a consistent coding standard or style guide can help teams maintain uniformity, making the codebase easier to navigate and understand (2008, chapter 5). It is not uncommon to find automatic formatting tools in modern code editor, simplifying this process.

2.3.2 SOLID Principles

Weisfeld explains SOLID as a set of five coding principles meant to solve common issues found in non-reusable code. Those principles were defined by Robert Martin and the SOLID acronym comes from Michael Feathers. (Weisfeld 2019, chapter 12) The following definitions are from Weisfeld's book, they are presented as direct quotations because Weisfeld did an excellent job of summarizing them.

Single Responsibility Principle:

> "The Single Responsibility Principle states that a class should have only a single reason to change. Each class and module in a program should focus on a single task. Thus, don't put methods that change for different reasons in the same class. If the description of the class includes the word "and," you might be breaking the SRP. In other words, every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated in the class." (Weisfeld 2019, chapter 12)

Open/Close Principle:

> "The Open/Close Principle states that you should be able to extend a class's behavior, without modifying it." (Weisfeld 2019, chapter 12)

Liskov Substitution Principle:

> "The Liskov Substitution Principle states that the design must provide the ability to replace any instance of a parent class with an instance of one of its child classes. If a parent class can do something, a child class must also be able to do it." (Weisfeld 2019, chapter 12)

Interface Segregation Principle:

> "The Interface Segregation Principle states that it is better to have many small interfaces than a few larger ones." (Weisfeld 2019, chapter 12)

Dependency Inversion Principle:

> "The Dependency Inversion Principle states that code should depend on abstractions." (Weisfeld 2019, chapter 12)

2.4 Game Engine

The main outcome of this thesis is a game engine. Hence the importance of defining exactly what a game engine is, especially since multiples definitions exist. The following sub-chapters look at the origin of the concept of game engine. They also gather modern definitions of the term from diverse sources. Finally, the definition of game engine for this project is given.

2.4.1 History

Jason Gregory, the author of the book “Game Engine Architecture”, explains that the term "game engine" originated in the mid-1990s with first-person shooter (FPS) games like Doom by id Software. Doom had an advanced separation of concern in its code, meaning that elements such as game assets and gameplay rules were not tightly coupled. This allowed developers to license and modify the game by creating new content with minimal changes to the engine. This innovation led to the rise of the "mod community," where gamers and small studios used provided toolkits to develop new games by altering existing ones. (Gregory 2017, chapter 1)

Before that, each new game had to be developed as a completely new project. It is suggested that this was in part due to the hardware limitations of the time that forced developers to write highly specific code which prevented reusability. (Andrade 2015, 1)

2.4.2 Definitions

While the term game engine originated from developer starting to reuse code, the term kept evolving. Here are a few different definitions found in diverse sources. On their website, arm, the CPU manufacturer, defines a game engine as a software development environment that streamlines video game creation across various programming languages, and that includes components like a 2D or 3D graphics rendering engine, a physics engine, artificial intelligence (AI) for player interaction, a sound engine, an animation engine, and other features (arm s.a.). Andrade also refers to a game engine as a software used for creating video games, mentioning that the components of the game engine are often a grouping of third party-libraries brought together to offer an abstraction layer for game development (Andrade 2015, 2). Those definitions are most relevant to modern game engines such as Unity, Unreal Engine and Godot, that offer software with advanced graphical user interfaces for game development.

In their book, Gregory highlights an interesting point regarding their definition of a game engine :

> “The line between a game and its engine is often blurry. When a game contains hard-coded logic or game rules, or employs special-case code to render specific types of game objects, it becomes difficult or impossible to reuse that software to make a different game. We should probably reserve the term “game engine” for software that is extensible and can be used as the foundation for many different games without major modification.” (Gregory 2017, chapter 1)

This paragraph suggests a broader definition of the game engine where the engine is considered as the code from a game that can be reused for making other games. This goes along with the definition found in a game development YouTube channel who describes it as (Giant Sloth Game April 2023, min. 1:30-2:30) “the collection of underlying systems that support a game” adding that an editor is not mandatory and that the engine could be as specific as one wants it to be, in regards of both game types or the platform that the game or engine runs on.

This second set of definition is more relatable to the context of this thesis, as the objective is to develop all the boilerplate code used for creating games, specifically in 2D and for the ESP32 platform.

2.5 ESP32

The ESP32 is a microcontroller from the company ESPRESSIF with networking capabilities using Bluetooth and Wi-Fi, making it an IoT device (CircuitSchools Staff 2022). Figure 6 shows a picture of an ESP32 development board.

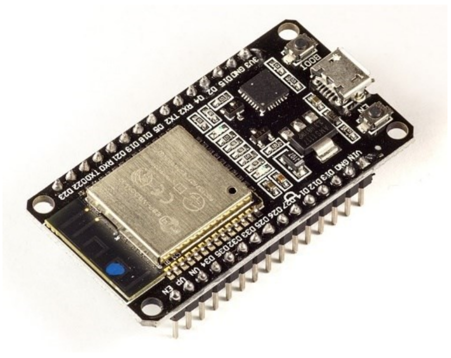


Figure 6. ESP32 Espressif ESP-WROOM-32 Dev Board, Ubahnverleih, 2018 CC0 1.0

The following chapters aim to introduce the process of developing ESP32 by defining important related concepts. They also explain what tools and steps are used to write code that runs on an ESP32.

2.5.1 Important Concepts

There are a few concepts that need to be defined to understand what the ESP32 is, such as what are embedded systems, what is a microcontroller, and what is the Internet of Things (IoT).

Embedded Systems

> “Embedded systems, also known as embedded computers, are small-form-factor computers that power specific tasks. They may function as standalone devices or as part of larger systems, hence the term "embedded," and are often used in applications with size, weight, power, and cost (SWaP-C) constraints.” (Brett 2024)

Microcontrollers

> “A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.” (Lutkevich 2019)

Internet of Things

> “The Internet of Things (IoT) describes the network of physical objects—“things”—that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the internet. These devices range from ordinary household objects to sophisticated industrial tools.” (Oracle s.a.)

2.5.2 Development Environment

An ESP32 requires software to perform tasks. This software, like the one that will be developed during this thesis, must be written on a computer. The computer then compiles the code and builds the application before uploading it onto the ESP32 (ESPRESSIF s.a.). An integrated development environment (IDE) is generally used to simplify this process by providing a code editor and a compiler. Many other tools are often found in IDEs, like the serial monitor which can be used to log messages from the ESP32 in real-time. The code itself is typically based on a framework that provides essential libraries and tools for developing applications on the ESP32. Arduino Core and ESP-IDF are the two most common frameworks used to develop code for the ESP32 (ESPBoards 2023).

As explained by ESPBoards, Arduino Core uses the Arduino programming language, which is based on C and C++. A project is referred to as a “Sketch” and the code is written in a file with the same name as the project and a “.ino” file extension. It is possible to write C++ classes and use them in Sketch. Two functions must be implemented in the Sketch file. The ***setup()*** function is called once when the board turns on, then the ***loop()*** function that is executed repeatedly while the board is powered. Arduino Core provides user-friendly abstraction when it comes to library management and code compilation. Meanwhile ESP-IDF, the official framework for ESP32 made by

Espressif, has a lower-level approach that provides more control over the code, but makes it more complex to learn. (ESPBoards 2023)

ESP-IDF can be used on Eclipse with a plugin, Arduino proposes their own IDE for Arduino Core, and both ESP-IDF and Arduino Core can be used with Visual Studio Code, the popular IDE from Microsoft, with the installation appropriate of Visual Studio Code extensions. (ESPRESSIF s.a.; Arduino 2023)

2.6 Similar Projects

This following sub-chapters will mention previous searches and similar projects. They are about existing game engines for ESP32, libraries that will be used in this thesis, some pre-built ESP32 gaming devices and similar previous projects from the author of the thesis.

2.6.1 Other ESP32 Game Engines

Many software developers show enthusiasm for game development and will develop games on any platform. The ESP32 is no exception, and this chapter lists some of the existing projects that can be used to make games on ESP32.

FabGL is a library for ESP32 that is used to control various types of displays such as VGA, Color NTSC/PAL Composite, I2C and SPI displays, and it also offers support for inputs, sound, graphical user interface and retro-game consoles emulation. (fdivitto 2023) Most relevant to this paper, the library is also a game engine as it provides a scene and sprites code base to write games as well as a collision detection class.

MicroGameConsole is a repository with a project meant to run on ESP32 with SSD1306 OLED display and push buttons (benricok 2023). The repository contains the code for pong-like and snake-like games.

The ESP Little Game Engine, from the ***esp8266_game_engine*** repository by corax89, provides a user interface to help make games. The project is composed of an online emulator of the ESP8266 based gaming device, an editor to write the code, a sprite making tool and a library for ESP development. (corax89 2021) The library is used in the online editor but can also be used as any other Arduino library. Functions are defined to be used for game making. The functions include sprite management, tiles management, strings and shapes rendering, and even particles emission.

2.6.2 Graphics Libraries

In embedded software development, it is common practice to use libraries to handle low-level code required to use sensors and components. Displays, for instance, are complex components that are most often used with libraries. This chapter groups popular graphics libraries for different embedded development displays.

The ***TFT_eSPI*** library by Bodmer is said to be usable to control TFT displays with a variety of different drivers. It is also compatible with many microcontrollers. With over three-thousand stars on the GitHub repository and over a thousand forks, this library has a strong community behind it and has been frequently used in tutorials. (Bodmer 2024)

Adafruit is an American company that makes and sell development boards and components. They also provide code to power their components like the ***Adafruit-GFX-Library***, a graphics library that can be used to power all Adafruit's displays when paired to hardware specific library that they also provide (Adafruit 2023).

Finally, ***FabGL*** by fdivitto has been mentioned as a game engine in the previous chapter. Though, it should also be cited as a graphics library since it is mainly described as a display controller on its GitHub repository. (fdivitto 2023)

2.6.3 IoT Experimental Project

The ***Haaga-Helia-IoT-Experimental-Project*** is the project that inspired this thesis topic. As mentioned on the GitHub repository of the project, it was made during the Spring 2024 IoT Experimental Project course of the Haaga-Helia University of Applied Sciences. The project is an arcade-like game made from an ESP32 and a set of sensors. In the game, a player controls a ship and tries to avoid incoming asteroids. The particularity of this game is that the controls change over time and go from a conventional joystick to touchless controls with a proximity sensor to orientation-based controls with a gyroscope sensor. (Gianou 2024)

This project was made collaboratively by Isabelle Stransky-Heilkron, Sara Erbismann and David Gianadda. David, the author of this thesis, was responsible for the game development part of the ***Haaga-Helia-IoT-Experimental-Project***. While the game is functioning, it was developed without clean coding in mind and expanding on it would require a lot of refactoring. The project made during this thesis is a new take on making a game engine for ESP32, with a focus on the reusability of the code instead.

3 Practical Part

The practical part of this thesis is organized by iterations. Each iteration starts by setting up the goals that should be reached by the end of it. Then the development work of the iteration is detailed and explains what was implemented and how it was implemented. Alternatives solutions and why they were not used are also mentioned.

The methodology used for this practical part is based on agile software development, although no specific methodology was strictly implemented. The concept of iterations, as used in this project, takes inspiration from Scrum methodology's sprint but without using a product backlog and the goals settings done at the beginning of each iteration is inspired by the Kanban approach.

3.1 Iteration 0 - Hello World

This chapter is about all the steps taken to be ready to start the development work on the game engine. Those steps include selecting the components used to build the gaming device, wiring all the components together, setting up the development environment, and testing the components.

The name of "Iteration 0 – Hello World" is based on the common software development practice of the "Sprint 0" which consists of setting up the development environment for a project up to a point where an "Hello World" message can be sent or displayed, demonstrating that the development work is ready to start. Therefore, the goal by the end of this iteration is to be ready for the development of the actual game engine.

3.1.1 Device & Components

To develop a game engine for ESP32, we need a gaming device. For this project we will build our own device by connecting inputs and outputs to a microcontroller. This chapter is about the components that were selected for the custom gaming device.

Note that an alternative to building the gaming device ourselves is to use pre-assembled ESP32 centric gaming devices that are commercially available. This option was discarded as part of the motivation behind this thesis topic was based on the opportunity to work with microcontrollers and components.

The brain of the device is the ESP32 microcontroller. All other components, like inputs and outputs, are directly connected to it. It also hosts the code used to get data from inputs and to send data to outputs. There are many models of ESP32 to choose from, varying in form factor, features, and memory capacities. The ESP32-S3-WROOM-1-N16R8 was selected for its Flash and PSRAM capacities. More Flash memory allows for more code to be uploaded to the microcontroller, and

PSRAM can be useful when displaying graphics. The ESP32-S3-WROOM-1-N16R8 has 16 MB of Flash and 8 MB of PSRAM. ESP32 boards typically have between 4 MB and 16 MB of Flash and from none to 16MB of PSRAM. (Figure 7.)

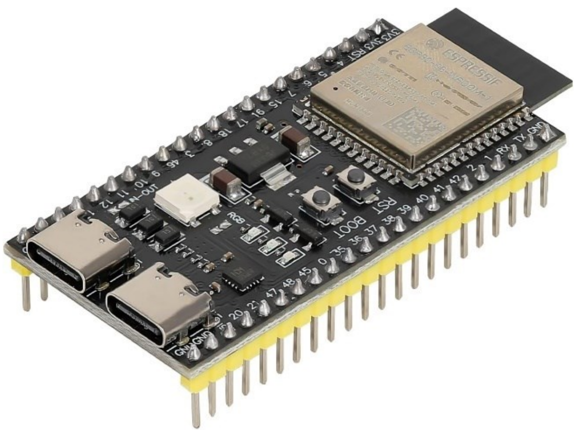


Figure 7. ESP32-S3-WROOM-1-N16R8 development board (Amazon 2023)

A display is needed to output the graphics of the game. Initially, a 3.5inch 480x320 MCU SPI Serial TFT LCD Module Display with an ILI9488 driver and an SD card slot was selected. For simplicity, this display is referred to as the TFT display in the rest of this document. (Figure 8.)

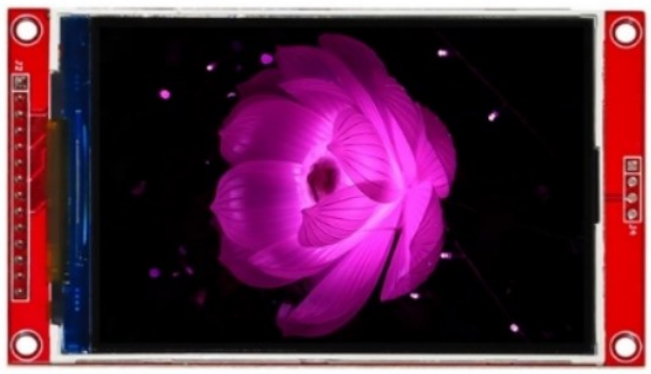


Figure 8. TFT LCD Module Display (Elecrow s.a.)

This choice was based on the screen size and resolution, its compatibility with the ***TFT_eSPI*** library, mentioned in the theoretical framework, and the integrated SD card slot. Unfortunately, this proved to be a poor choice for reasons that are discussed in chapter 3.3. This display had to be switched during the development process.

The replacement display is the Adafruit Monochrome 2.7" OLED Graphic Display. The resolution is much lower than the one from the previous display with only 128x64 pixels. The reasons behind this choice are explained in chapter 3.3. (Figure 9.)

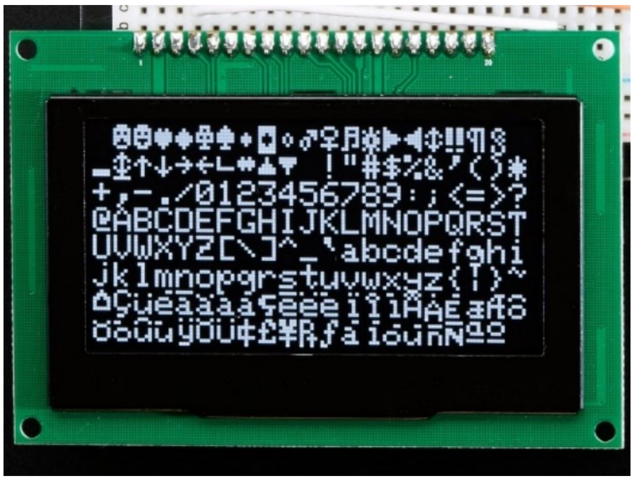


Figure 9. Adafruit Monochrome 2.7" OLED Graphic Display (Adafruit s.a.)

Finally, any gaming console needs inputs, such as buttons and joysticks. Inputs enable the player to interact with the device.

Buttons come in many different types and shapes. The ones used in this project are known as push button switches. Keyboard switches were selected for their satisfying feel, and for design preferences. Note that any push button switches can be used instead. A push button switch operates by closing a circuit when pressed, allowing current to flow between its terminals. Figure 10 shows the exact model used in the project.

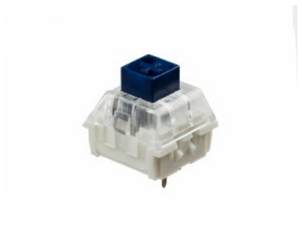


Figure 10. BOX Navy Clicky Switch from Novelkeys (MaxGaming s.a.)

The last component is a joystick. A joystick is essentially two potentiometers that read analog values. One potentiometer is dedicated to the “X” axis, and one is dedicated to the “Y” axis. Each axis can have their value accessed by reading the pin of the microcontroller they are wired to. The specific joystick for this project is a brandless component that also features an integrated push button like shown in figure 11.

Figure 11. Joystick mounted on a PCB (Sintosen palvelut s.a.)

The prototype device went through two phases during the development of the project. At first, all components were mounted on a breadboard. This version used the TFT display and some basic buttons. (Figure 12.)

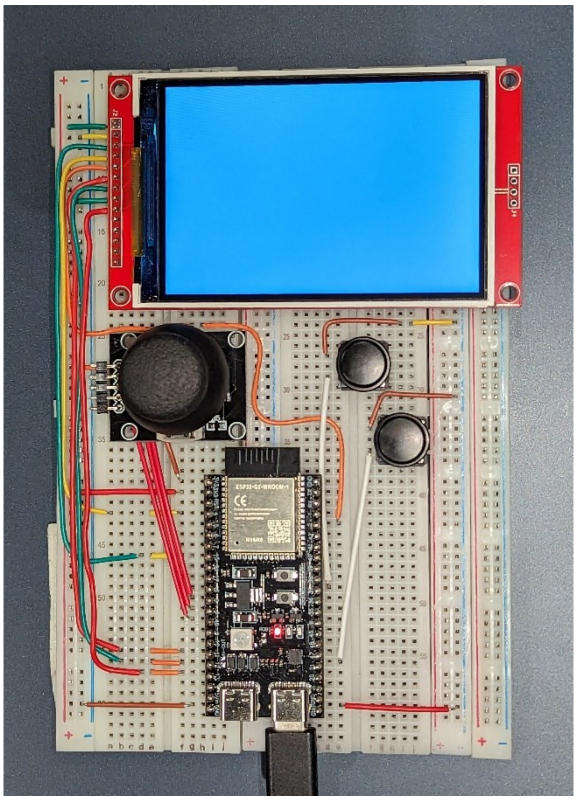


Figure 12. Phase 1 of the gaming device prototype

The second phase is an improved version of the prototype, as seen in figure 13. A mounting piece was designed in the open-source 3D modelling software Blender with the FreeCAD addon. All the components previously mentioned in this chapter were measured and modeled into Blender. They were then placed in a satisfactory manner and the mounting piece was modelled to hold them all together. The mounting piece was 3D printed on a resin 3D printer at the Haaga-Helia 3D Lab.

This version of the gaming device uses the OLED display and has a total of five keyboard switches used as buttons.

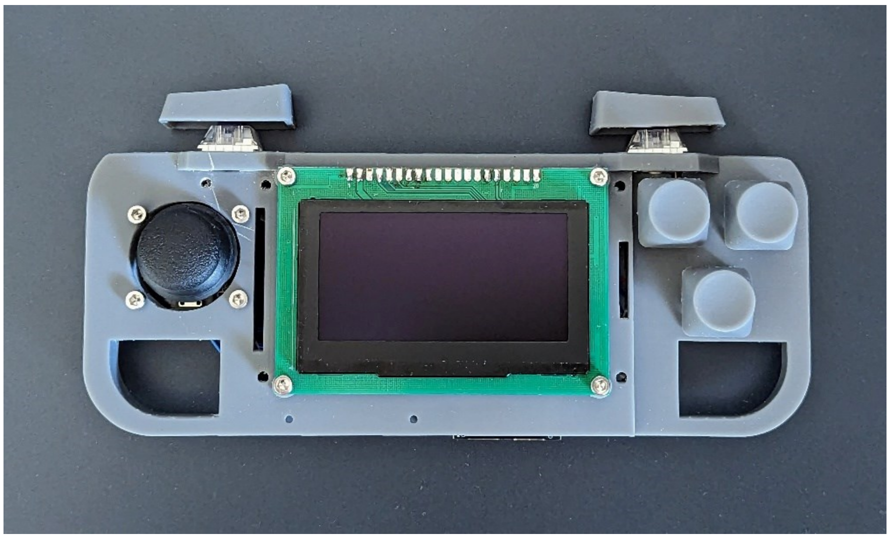


Figure 13. Phase 2 of the gaming device prototype

Note that all the components, appart from the joystick, used during this thesis were acquired and financed by the author of the thesis. The joystick was provided by the 3D Lab from Haaga-Helia.

3.1.2 Development Environment

To start developing the code of the game engine, an integrated development environment (IDE) is required. Popular IDEs for ESP32 have been mentioned in the theoretical framework. For this project, Visual Studio Code from Microsoft with the Arduino extension was chosen. This choice is based on Visual Studio Code's integrated Git tools and PowerShell terminals, as well as personal experiences with it.

The Arduino Core framework was selected based on personal experiences and for its beginner-friendliness. The ***Haaga-Helia-IoT-Experimental-Project***, overviewed in the theoretical framework, had already been made using the Arduino Core framework. Furthermore, as this thesis aims to provide some educational content to the reader, the simpleness of the Arduino Core framework was deemed more fitting than the more complex ESP-IDF framework.

Version control with Git will be used to manage and save the project to a remote GitHub repository. This repository is licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license, making it easy for anyone to access and use the project. Since one of the goals of this project is to be educational, the repository will feature a dedicated branch for each iteration found in the practical part of this thesis. This structure will allow readers to easily access the code produced in each iteration, facilitating their understanding and learning.

3.1.3 Testing the Components

The last step of iteration 0 is to test each component to make sure they are properly wired, and to ensure that they are not damaged or malfunctioning. Some components require an external library to be used, this will also be covered in this chapter.

Buttons

Buttons can be tested with the Arduino sketch from figure 14. The code works as follows. First, the constant for each of the button's dedicated pin is defined. Then, the pin mode is defined using the ***pinMode()*** method from the Arduino Core framework. Finally, in the loop, the value of each pin is accessed with the ***digitalRead()*** method from the Arduino Core framework, and it is printed in the serial monitor. The value should be "1" when the button is not pressed and "0" when pressed.

```
#define BUTTON_A_PIN 37
#define BUTTON_B_PIN 21

void setup()
{
    Serial.begin(115200);
    pinMode(BUTTON_A_PIN, INPUT_PULLUP);
    pinMode(BUTTON_B_PIN, INPUT_PULLUP);
}

void loop()
{
    Serial.print("A : " + String(digitalRead(BUTTON_A_PIN)) + " | ");
    Serial.println("B : " + String(digitalRead(BUTTON_B_PIN)));
    delay(100);
}
```

Figure 14. Button testing code

Joystick

The joystick is composed of two potentiometers and one button. The "X" and "Y" are analog values that have dedicated pins. Those values can be read by calling the ***analogRead()*** method with the pin's number as parameter. The "X" and "Y" values range from 0 to 4095 and are expected to be around 1900 and 2000 when the joystick is not touched. Note that, due to their analog nature, the value of "X" and "Y" may fluctuate. Figure 15 shows the code used for testing the joystick's axes. The joystick's button is being tested in the same manner as previously done in figure 14. The axes do not require setup. In the ***loop()***, both axis and the button's value are accessed and displayed in the serial monitor.

```
#define VRX 8
#define VRY 3
#define SW 46

void setup()
{
    Serial.begin(115200);
    pinMode(SW, INPUT_PULLUP);
}

void loop()
{
    Serial.print("X : " + String(analogRead(VRX)) + " | ");
    Serial.print("Y : " + String(analogRead(VRY)) + " | ");
    Serial.println("SW : " + String(digitalRead(SW)));

    delay(100);
}
```

Figure 15. Joystick test code

Display

The display plays a crucial role in the game device. To develop games, basic functions like drawing shapes, drawing images, and displaying text are needed. Thankfully, we do not need to implement those functions as they can be found in libraries that handle all the low-level code used to display graphics on a screen.

The TFT display, presented in the theoretical framework, is the display that was first meant to be used for this project, and it was used during the development work until iteration 2. The ***TFT_eSPI*** library is used to handle the display. It can be downloaded from the libraries' GitHub repository where the set-up process is also detailed (Bodmer 2024). The library requires to be set up according to the display we are using.

The code in figure 16 was used to test the TFT display. The code starts by including the SPI library from the Arduino Core framework and the ***TFT_eSPI*** library. Then an object from the ***TFT_eSPI*** library is instantiated as ***tft***. In the ***setup()*** function, some initialization methods are called on the ***tft*** instance and the message “Hello World!” is printed on the actual display. The expected outcome is shown in figure 17.

```
#include <SPI.h>
#include <TFT_eSPI.h>

TFT_eSPI tft = TFT_eSPI();

void setup()
{
    tft.init();
    tft.setRotation(3);
    tft.fillScreen(TFT_BLACK);
    tft.setTextSize(5);
    tft.setCursor(48, 120);
    tft.print("Hello World!");
}

void loop()
{
}
```

Figure 16. TFT display test code

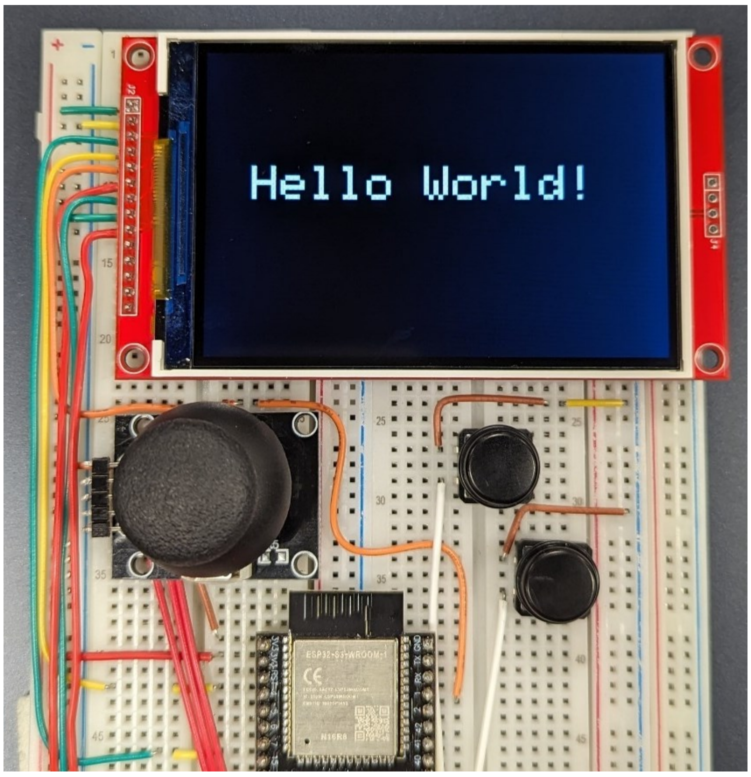


Figure 17. Hello World on TFT display

The TFT display was later replaced by an OLED display. The library used to drive the OLED display is made by Adafruit, who also manufactures the display itself. Adafruit's approach to writing graphics libraries is different to the approach used by Bodmer. Instead of using a single library and a set-up file, Adafruit uses two libraries. The first is the ***Adafruit-GFX-Library*** that contains all the functions to interact with the display (Adafruit 2023). And the second is specific to the display used and defines the display's properties. The second library used in this project is

Adafruit_SSD1325_Library (Adafruit 2024). The approach of dedicated libraries is more user-friendly whereas the set-up file approach is more open for users' tinkering.

The code used to test the OLED display comes from the ***Adafruit_SSD1325_Library*** and can be found in the examples folder under "ssd1325test" (Adafruit 2024). The expected result is a bunch of Adafruit logos being displayed on the screen and moving down in a falling motion, as seen in figure 18.

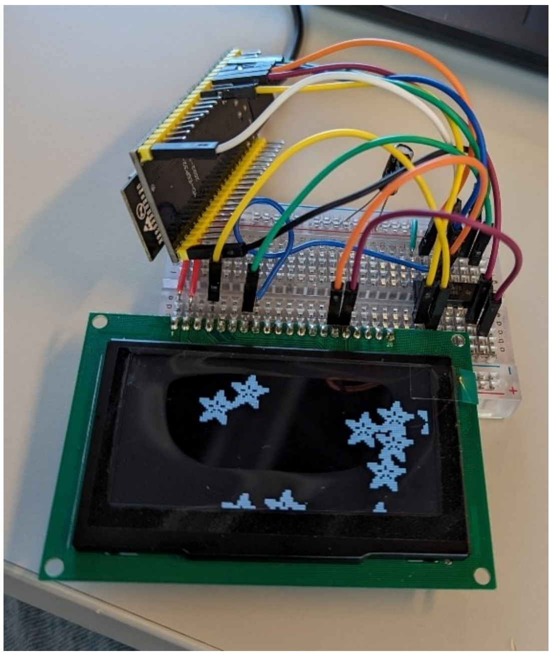


Figure 18. Adafruit logos on OLED display

3.2 Iteration 1 – Engine's Core & Inputs

Iteration 1 marks the beginning of the actual project. From now on, all the code can be found in the GitHub repository of this project (appendix 1). Each iteration found in this report has a corresponding branch in the repository. Branches are typically deleted once they are merged into the main branch. However, for this project, each branch has been preserved so that the code produced in each iteration is easily identifiable. It is advised to read each chapter of the practical part with the GitHub repository open on the corresponding branch. Branch can be selected on the top left side of the webpage as shown in figure 19.

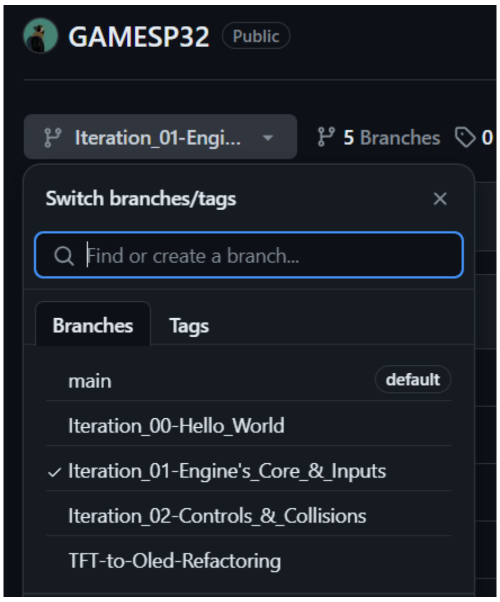


Figure 19. Branch selection on GitHub

The goals for this iteration are to :
- get user inputs and have them accessible anywhere in the code.
- implement the core loop of the game engine.
- demonstrate the core loop with an animated screen saver.

3.2.1 Inputs as Objects

In iteration 0 we already covered how to get the values of the joystick and buttons. The code used for testing the inputs was entirely written in the Arduino Sketch file. Now, we want to encapsulate the logic used for inputs into objects.

We start by defining an interface, named ***AbstractInput***. This class will define what behaviors must be implemented by the concrete inputs. By applying the Dependency Inversion Principle, we enforce the use of polymorphism. This will later be useful to handle any input, regardless of it being a joystick or a button. This is an example of the application of the Liskov Substitution Principle.

The ***AbstractInput*** class has two virtual methods, ***begin()***, and ***getValue()***. The ***begin()*** method will be used for inputs that require a setup, like the buttons do. The ***getValue()*** is simply used to return the current value of the input. This may seem like it would not work with the joystick that has three distinct values. However, the joystick does not need to be represented in the code as a joystick object. Instead, it can be broken down into two ***JoystickAxis*** and one ***Button***, all of them implementing the ***AbstractInput***. (Figure 20.)

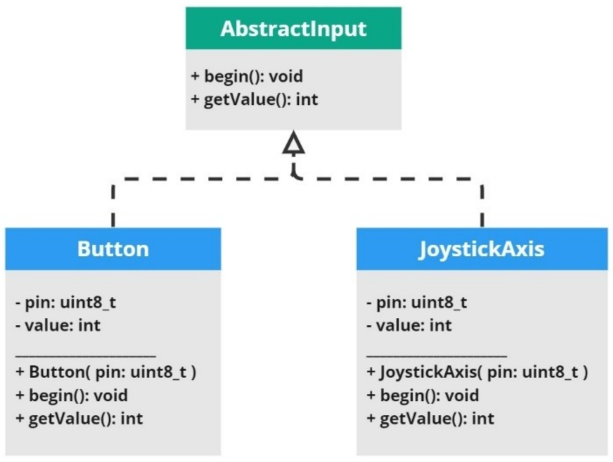


Figure 20. Abstract and concrete inputs class diagram

3.2.2 Input Manager as Singleton Pattern

Inputs are a key part of any video game. Inputs enable players to move characters, to drive vehicles, to navigate menus, or to attack enemies, and to perform many more actions, making interactivity a defining feature of video games and setting them apart from other forms of media.

Given the wide variety of scenarios and use cases for inputs, the game engine should provide a way to easily access inputs programmatically. We will implement a class, the ***InputManager***, whose responsibility is to make the current value of a given input readable from anywhere in the code. The Singleton design pattern is ideal for such a scenario.

The ***InputManager*** has a ***map*** of ***String*** and ***AbstractInput*** attribute as well as methods to add and remove ***AbstractInput*** to this ***map***. The ***getInputValue()*** method is used when the client needs to access the current value of an input, it takes a ***String*** as parameter. This ***String*** is the name of the ***AbstractInput***, that is also used to organize the ***AbstractInput*** in the ***map***. The name attribute had to be added in the ***AbstractInput*** for this implementation of the ***InputManager*** to work. (Figure 21.)

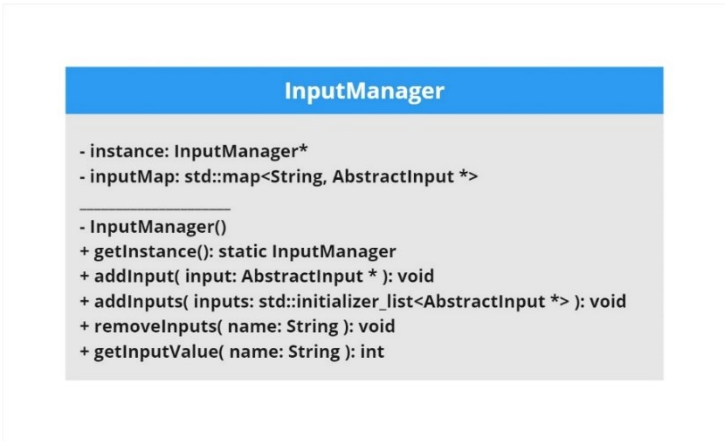

InputManager
- instance: InputManager*
- inputMap: std::map<String, AbstractInput *>
- InputManager()
+ getInstance(): static InputManager
+ addInput(input: AbstractInput *): void
+ addInputs(inputs: std::initializer_list<AbstractInput *>): void
+ removeInputs(name: String): void
+ getInputValue(name: String): int

Figure 21. InputManager class diagram

The use of composition and polymorphism with this Singleton implementation makes the code easy to expand without having to modify the code of the class. If a new input type was to be added to the game device, all the new code required is a new class that implements the ***AbstractInput***, a line to instantiate this new class and a line to add it to the ***InputManager***'s ***inputMap***. This is an example of the Open/Close principle. Figure 22 shows the UML for the complete input management system.

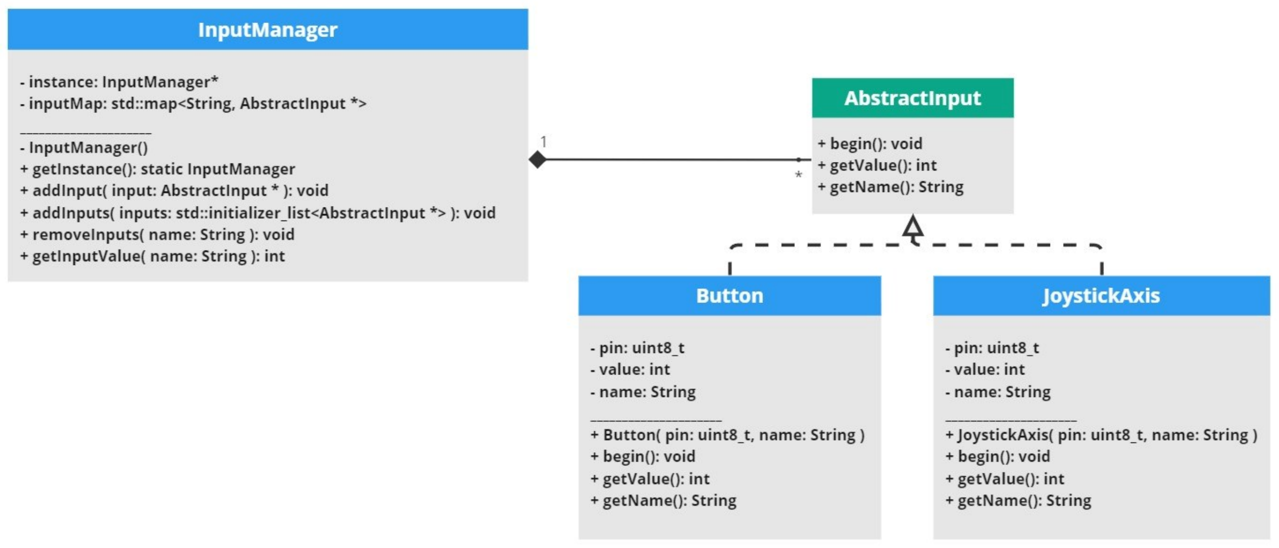

InputManager
- instance: InputManager*
- inputMap: std::map<String, AbstractInput *>
- InputManager()
+ getInstance(): static InputManager
+ addInput(input: AbstractInput *): void
+ addInputs(inputs: std::initializer_list<AbstractInput *>): void
+ removeInputs(name: String): void
+ getInputValue(name: String): int
1
*
AbstractInput
+ begin(): void
+ getValue(): int
+ getName(): String
Button
- pin: uint8_t
- value: int
- name: String
+ Button(pin: uint8_t, name: String)
+ begin(): void
+ getValue(): int
+ getName(): String
JoystickAxis
- pin: uint8_t
- value: int
- name: String
+ JoystickAxis(pin: uint8_t, name: String)
+ begin(): void
+ getValue(): int
+ getName(): String

Figure 22. Input management architecture

3.2.3 Loop

To build the base of the game engine, it is important to understand how video games work. Let us use the example of a flipbook. In the creation of a flipbook animation, every frame of the animation is drawn on the pages of the book. Once this process is completed, the pages can be flipped to play the animation. Similarly, a video game consists of a succession of frames. Although, the frames must be drawn in real-time instead of in advance. The first role of the engine is to ensure that a loop is maintained, like a continuous flipping of pages.

Every project done with the Arduino Core framework must implement two functions, ***setup()*** and ***loop()***. We can simply use this ***loop()*** function as the engine's base loop operation. Note that passages in the ***loop()*** function from the Arduino Core framework happen as fast as the chip can handle them. A common practice in Arduino and ESP32 development is to introduce a delay at the end of the ***loop()*** function. The ***delay()*** function can stop the code execution for a given amount of time and is often used to slow down the execution of the code. It could be a solution to set a frame rate in the game engine. Although, this might result in inconsistent frame rate due to the game logic that could take more or less time to be executed depending on its complexity.

Instead, we will implement a function that checks the amount of time that elapsed since the beginning of passage in the ***loop()***. If the amount of time is equal to or greater than a set frame duration the next frame can start, otherwise the function keeps evaluating the elapsed time. (Figure 23).

```
void waitUntilEndOfFrame()
{
    static unsigned long lastLoop;
    while (millis() - lastLoop < FRAME_DURATION_MS)
    {
    }
    lastLoop = millis();
}
```

Figure 23. Frame duration control function

3.2.4 Engine's Core as Composite Pattern

The second role of the engine is to manage the ongoing processes within each frame or loop iteration. We will refer to it as game logic. It includes rendering the background, handling the behavior of non-playable characters, rendering UI, detecting collision and all other events that can happen in games. It is all the code that is executed during each frame.

To tackle the complexity of the game engine, the game logic will be broken down and encapsulated into game objects. Knowing that those objects will be performing actions, and that they will be rendered on the display, polymorphism should be used to abide to the Liskov Substitution Principle. This can be done by defining an abstract class for the game objects. Figure 24 shows the new

AbstractGameObject class, used as the interface for game objects. The interface specifies two methods. The ***update()*** method will be used to perform game logic, such updating the coordinates of a moving character or updating a score. The ***render()*** method is used to render the object on the display, which is why it takes a reference to an instance of a ***TFT_eSprite***. This is the object used to draw on the display. It comes from the ***TFT_eSPI*** library. All game objects should be updated and rendered in the Sketch file ***loop()*** function.

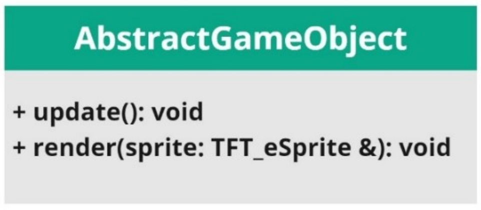


Figure 24. AbstractGameObject class diagram

At this point, it would be possible to create some concrete game objects, classes that implement the ***AbstractGameObject***, and make a game. Figure 25 shows a supposition of what this would look like. In the ***loop()***, all game objects are individually updated then are all rendered on by one.

```
void loop() {
  // Update environment
  background.update();
  tileset.update();
  foreground.update();
  // Update player
  player.update();
  // Update enemies
  enemy1.update();
  enemy2.update();
  // Update UI
  lifeUI.update();
  scoreUI.update();
  // Render environment
  background.render(sprite);
  tileset.render(sprite);
  foreground.render(sprite);
  // Render player
  player.render(sprite);
  // Render enemies
  enemy1.render(sprite);
  enemy2.render(sprite);
  // Render UI
  lifeUI.render(sprite);
  scoreUI.render(sprite);
}
```

Figure 25. Example of messy game engine code

The example from Figure 25 is here to showcase what we want to avoid. Readability is not great and there is a lot of repetition in the example code. Those issues would only get worse when the complexity of the project increases. However, this counter example does contain a clue to a solution to its problems. As suggested by the code's comments, some logical groups appear. Those groups are the environment, the player, the enemies, and the user interface (UI).

The Composite design pattern is an ideal solution to this situation. It provides a hierarchical tree structure which allows for a logical grouping of the elements. With a Composite pattern, all UI elements, enemies, and environment elements of the fictional example from figure 25 can be grouped into nodes. Figure 26 illustrates what the tree would look like for this example.

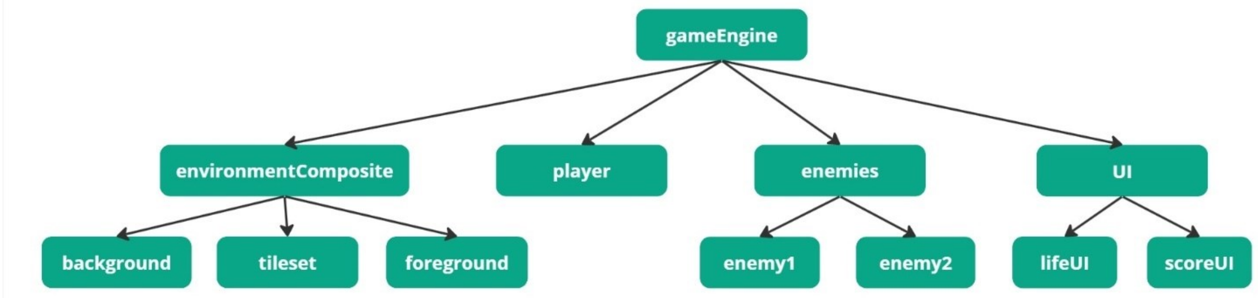


Figure 26. Example of composite tree for a well-organized game engine

GameEngine is a new class that implements ***AbstractGameObject*** along with a ***vector*** of ***AbstractGameObject*** and helper methods to add and remove elements to this ***vector***. It serves as the root node of the Composite's pattern tree. The ***GameEngine***'s purpose is to recursively call the ***update()*** and ***render()*** method down the composite tree. The new ***loop()*** function would change to be what is shown in figure 27.

```
void loop() {
  gameEngine.update();
  gameEngine.render(sprite);
}
```

Figure 27. Example of well-organized game engine code

For the implementation of the Composite pattern in the game engine, the ***AbstractGameObject*** class from figure 24 is used as the interface that must be implemented component of the tree. In our case, the operation related to adding and removing child components will not be included in the ***AbstractGameObject*** class. As mentioned in the theoretical framework, whether both leaves and nodes should implement the child component helper functions or not depends on the sources or the context of the implementation.

We also implemented a composite node, the ***GameEngine***, which is the root node of the tree. To complete the core of our game engine, a leaf node is implemented. The ***RenderEngine*** is responsible for pushing the buffered frame to the display. From the ***TFT_eSPI*** library, we are using an instance of the ***TFT_eSprite*** object, which allows us to pre-render a frame in memory before pushing it to the display. This buffering technique helps with avoiding screen flickering. Figure 28 shows the UML implementation of the game engine's core as a Composite pattern.

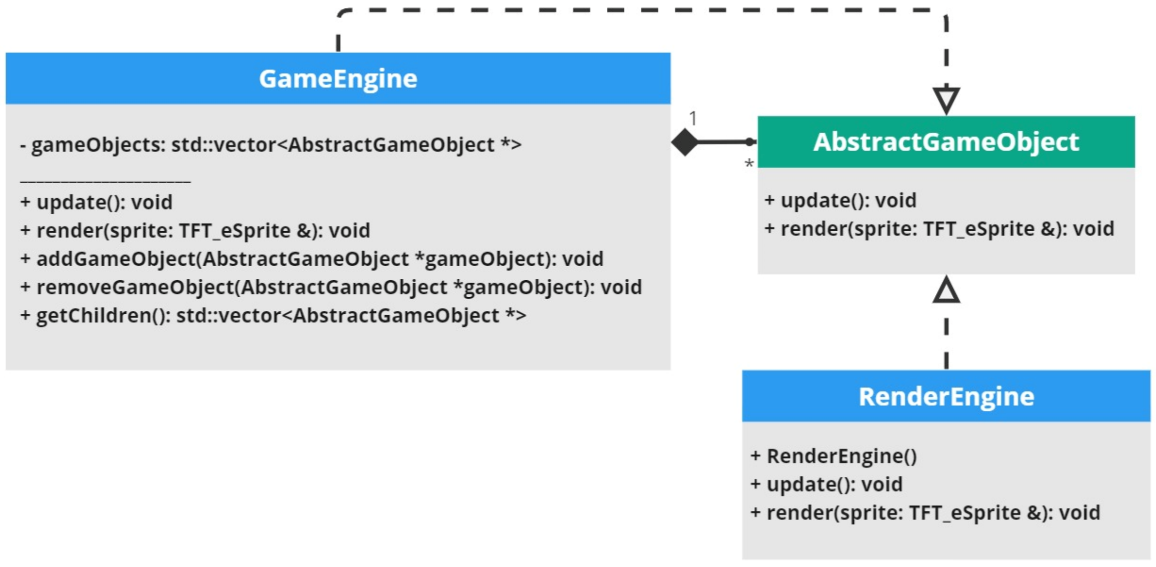

GameEngine
- gameObjects: std::vector<AbstractGameObject *>
+ update(): void
+ render(sprite: TFT_eSprite &): void
+ addGameObject(AbstractGameObject *gameObject): void
+ removeGameObject(AbstractGameObject *gameObject): void
+ getChildren(): std::vector<AbstractGameObject *>
1
*
AbstractGameObject
+ update(): void
+ render(sprite: TFT_eSprite &): void
RenderEngine
+ RenderEngine()
+ update(): void
+ render(sprite: TFT_eSprite &): void

Figure 28. Architecture of game engine core as Composite pattern

3.2.5 Screen Saver Demo

To conclude this iteration, a demonstration should be implemented. This demonstration is not part of the game engine but serves as testing of the work done in the iteration. As mentioned in the goals for this chapter, the demonstration should be a screen saver that displays an animation in a loop.

It was thought that it would be interesting to ask ChatGPT, the large language model (LLM) based chatbot from OpenAI, to generate the class that should display a bouncing ball on the device's screen. By providing information, such as the design pattern used for the game engine architecture and the code of the ***AbstractGameObject***, ***GameEngine***, and ***RenderEngine***, ChatGPT was able to return an almost ready-to-use ***DemoBounceSphere*** class. Only the path of the "includes" and a line performing an unnecessary rendering operation had to be modified from the code generated by ChatGPT. Figure 29 shows the Composite pattern tree for the screen saver demonstration.

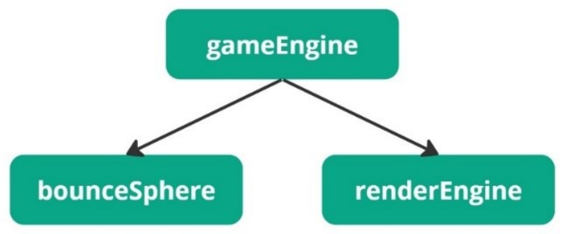


Figure 29. Composite tree of screen saver demonstration

3.3 Display Issue

This chapter is dedicated to an issue that emerged at the end of iteration 1, when running the bouncing sphere screen saver demonstration. The animation of the sphere appeared to be slow and jittery. Code to measure the duration of each passage in the ***loop()*** method was immediately implemented, as shown in figure 30, and revealed that it took 728 ms to display a single frame. This means that the game engine was running at approximately 1.37 frames per second (fps). For reference, according to a blog post from Saraswat, the slowest commonly used frame rate is 24 fps for movies, whereas video games often aim for 60 fps or more (26 November 2023).

```
void loop()
{
    unsigned long startTime = millis();

    gameEngine.update();
    gameEngine.render(display.getSprite());

    unsigned long endTime = millis();
    unsigned long duration = endTime - startTime;
    Serial.println("Loop iteration duration: " + String(duration) + "ms");
}
```

Figure 30. Loop function with loop iteration duration measurement

3.3.1 Identifying the Issue

To solve this issue, let us firstly look at the process of displaying the image on screen. We are using Bodmer's ***TFT_eSPI*** library which offers two solutions for displaying images. The first is to use an instance of the ***TFT_eSP***I object that has methods such as ***fillCircle()*** that will draw a filled circle on the display. On each call of the ***TFT_eSPI*** drawing method, the screen will be updated instantly. The newly drawn item will appear on top of what was being previously displayed. With this method, each object would have to be cleared before being drawn again, to create an animation. This can either be done by redrawing the object in its previous location but with a background color and then drawing it again in its new location, or by clearing the whole display with a background color. This method was initially used during the ***Haaga-Helia-IoT-Experimental-Project*** and the result was unsatisfactory as it produced an intense flickering effect.

The second technique that can be used to display graphics is the use of ***TFT_eSprite***. As explained by Bodmer, ***TFT_eSprite*** are graphics drawn in memory instead of directly on the display. They are then pushed on to the display once ready. Display sized ***TFT_eSprite*** can be used for frame buffering. (Bodmer 2024) To draw a filled circle we would first instantiate a ***TFT_eSprite*** and then use it the ***fillCircle()*** method. This technique was implemented in the ***Haaga-Helia-IoT-Experimental-Project*** to fix the flickering issue. It is also the technique being used in this project.

Note that in the ***Haaga-Helia-IoT-Experimental-Project***, the screen resolution was 128x160 pixels, and the frame rate was never noticed to be problematically slow.

Other information we get from the ***TFT_eSPI*** library is about color depth. The color depth is the number of bits dedicated to storing data about color (Brian 2023). It is possible to reduce the color depth used to reduce the size of graphics. Bodmer mentions using this technique with ***TFT_eSprite*** if the microcontroller's memory is not sufficient for frame buffering (Bodmer 2024).

Considering all of the above, testing was done to see how different factors like rendering techniques, color depth and render size would affect the frame rate. The results of those tests are shown in table 1.

Table 1. Results of rendering speed testings

Width (px)	**Height (px)**	**Total Resolution (px)**	**Color Depth (bit)**	**Technique**	**Frame Duration (ms)**	**Frame Per Second**	**Comment**
480	320	153600	16	TFT_eSprite	728	1.37	
480	320	153600	1	TFT_eSprite	744	1.34	
240	160	38400	16	TFT_eSprite	182	5.49	
240	160	38400	1	TFT_eSprite	186	5.38	
480	320	153600	16	TFT_eSprite	728	1.37	Nothing drawn on TFT_eSprite
480	320	153600	16	TFT_eSprite	0	X	TFT_eSprite not pushed to screen
480	320	153600	16	TFT_eSPI	151	6.62	Frame is reset by drawing a display sized black rectangle. Result is flickery
480	320	153600	16	TFT_eSPI	4	250.00	Only the past location of the sphere is reset by drawing a black circle.
480	320	153600	16	TFT_eSPI	25	40.00	Draw 5 sphere, only the past location of the sphere is reset by drawing a black circle.
480	320	153600	16	TFT_eSprite	688	1.45	SPI_FREQUENCY 40000000 in user_setup.h
480	320	153600	16	TFT_eSPI	105	9.52	SPI_FREQUENCY 40000000 in user_setup.h

The testing revealed important information. Firstly, the use of lower color depths never improved the frame duration. The initial hope was that reducing the size of the ***TFT_eSprite*** would make it faster to render. Secondly, we can notice that the total resolution of the ***TFT_eSprite*** and the time it takes to display it are linked. We also learned that a ***TFT_eSprite*** of a given size always takes the same amount of time to be rendered regardless of whether we draw a circle on it or not.

On the other hand, ***TFT_eSPI*** render time is linked to the size of what is being drawn. For that reason, the best results were obtained when redrawing only the sphere instead of clearing the complete screen by drawing a black rectangle. Despite the speed, the uses of ***TFT_eSPI*** without clearing the screen has many issues and limitations:

- Its results are often flickery.
- It makes the code side of the rendering process more complex.
- It limits the number of pixels that can be changed in each frame.
- It makes it complex to use background other than solid colors.

Finally, the test done in the last two lines of Table X shows an interesting result. For both ***TFT_eSPI*** and ***TFT_eSprite***, changing a setting related to SPI frequency in the setup file of Bodmer's library showed slightly better performance. The default value for this setting, used in all the other tests, is 27000000. In the comments of the setup file of their library, Bodmer mentions that the ***SPI_FREQUENCY*** constant is the SPI clock frequency and that it directly affects the rendering speed. They also mention that a speed of 80000000 may either work or cause corruption. (Bodmer 2024) The results of rendering a circle with an SPI clock frequency of 80000000 resulted in strange graphics as shown in figure 31.

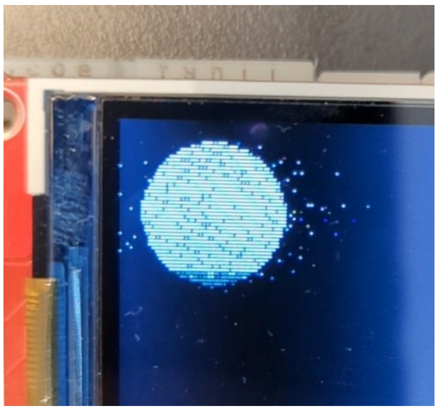


Figure 31. Corrupted display of a circle

There are two main options to solve the slow frame rate issue. The first is to only use part of the screen, with the ***TFT_eSprite*** frame buffering technique. The inconvenience being that using only a fraction of the screen is aesthetically displeasing. The second option is to use ***TFT_eSPI*** drawing with a limited number of pixels being updated each frame. This option brings a lot of complexity to the code and the results look flickery. Other options require to change the display for either another display that uses either a faster transfer protocol, or a smaller resolution, or another display technology or all of those.

It was decided that the display will be replaced by a new one. The Adafruit Monochrome 2.7" OLED Graphic Display was chosen for its unique size and resolution. The resolution is 128x64 pixels, yet the display's diagonal size is 2.7" or 6.858 cm. This is explained by the exceptionally large pixels being used. Those pixels are also limited to a color depth of one, as the display is monochrome. The hope behind this choice is that by greatly reducing the frame size, rendering time will not be an issue anymore. Note that this new display also uses SPI as communication protocol. Aesthetically, the monochrome and big pixel factors provide an arcade-like style that is coherent with the rest of the game device.

3.3.2 Refactoring

The new display requires a new library. This means that the code needs refactoring to adapt. Fortunately, by following good coding practices, the amount of code to change was kept low. At first, the ***Display*** class, used to represent the physical display in the code, was modified to create a ***DisplayOLED*** class, functioning in the same manner but with instance of the ***Adafruit_SSD1325*** class instead of the ***TFT_eSPI***.

This was then changed again when it was realized that there was no need for a ***Display*** or ***DisplayOLED*** class at all. The initial idea was to have a class to represent each type of physical component wired to the ESP32. For instance, the Button class handles all the logic related to physical buttons. This is essentially what the ***Adafruit_SSD1325*** class, from the external library, does. For that reason, and for simplicity, the ***DisplayOLED*** class was removed in a second refactor and instead, an instance of ***Adafruit_SSD1325*** was given as an attribute to the ***RenderEngine*** class.

In the GitHub repository of this project, the branches Iteration_0 and Iteration_1 are written to work with the TFT display. The branch Iteration_2 was started with the TFT display and a new branch named TFT-to-Oled-Refactoring was used to implement the refactoring explained in this chapter. During this process of debugging and refactoring and debugging, the second prototype of the gaming device was made. It uses the new OLED display.

3.4 Iteration 2 – Controls & Collisions

In iteration 1, a system to access input data via the InputManager Singleton was implemented. During this iteration, we will test this system by creating interactions between what is being displayed on screen and the user inputs. For that, a simple pong-like game will be developed. In this game, two paddles are placed on the left and right side of the screen. A ball bounces from side to side of the screen and the paddles must be used by the players to prevent the ball from exiting the screen on their side, similar to a game of table tennis.

The second feature that will be implemented in this iteration is a collision detection system. In the example of the pong-like game, this system will be used to detect collisions between the ball and the paddles. The system itself will be part of the game engine and should be reusable.

3.4.1 User-Controlled Paddle

We will start by implementing a user-controlled paddle. The paddle is an object and will have its dedicated class ***DemoPaddle***. Just like the ***DemoBounceSphere***, it is part of the ***GameEngine*** Composite pattern tree. Hence, it must implement the ***AbstractGameObject*** class and must be added to the ***GameEngine*** instance.

The ***update()*** method of ***DemoPaddle*** must implement the movement logic. It needs to get the value of inputs and move the paddle accordingly. Figure 32 shows the most basic implementation of this logic. We can also see that the ***InputManager*** is accessible from the ***DemoPaddle*** class without having to be passed down the composite tree, thanks to the Singleton property of being accessible anywhere in the project.

```cpp
void DemoPaddle::update()
{
    InputManager *inputManager = InputManager::getInstance();
    if (inputManager->getInputValue("Y axis") > 2000)
    {
        y += speed;
    }
    else if (inputManager->getInputValue("Y axis") < 1900)
    {
        y -= speed;
    }
}
```

Figure 32. Update method of DemoPaddle class

In the ***update()*** method of figure 32, we first get the reference to the ***InputManager*** Singleton. Then we change the value of ***y*** based on the “Y” axis input of the ***InputManager***. There is an issue with this code, it allows for the paddle to move out of the limit of the screen. In the ***DemoB-ounceSphere*** from the previous iteration, we were passing the screen’s height and width as parameters. This allowed us to check if the ball’s ***x*** and ***y*** values were within the limits of the screen. Though, passing the screen’s width and height to every object that may need it is redundant and cumbersome.

In this iteration we will keep using the screen’s dimensions as borders to limit the movements of the paddles. However, access to those dimensions will be implemented in a better way. There are a few options to improve this case. The first would be to use a Singleton that contains those informations. The Singleton could be the ***GameEngine*** itself, the ***RenderEngine*** or some new class made for that purpose only. Although, since the value we want to make easily accessible are constants, we will instead create a simple header file and store those constants there. We will also take this refactoring opportunity to move all constants from the Sketch file to this new ***Constants.h*** file. This makes the Sketch file more readable. And we will also replace the use of the ***#define*** keyword to declare typed constants instead. Using the ***const*** keyword allows for type safety, whereas ***#define*** cannot be typed checked (GeeksforGeeks 2022).

With the screen’s dimensions now easily accessible, the logic to prevent the paddle from going out of screen can be implemented. This is done by checking if the paddle is out of screen after moving and resetting it to the exact border of the screen if it is the case, as shown in figure 33.

```
void PaddleLeft::update()
{
    InputManager *inputManager = InputManager::getInstance();
    if (inputManager->getInputValue("Y axis") > 2000)
    {
        y += speed;
    }
    else if (inputManager->getInputValue("Y axis") < 1900)
    {
        y -= speed;
    }
    // Check for screen boundaries
    if (y < 0)
    {
        y = 0;
    }
    if (y + height > SCREEN_HEIGHT)
    {
        y = SCREEN_HEIGHT - height;
    }
}
```

Figure 33. Update method of DemoPaddle class with screen border checking

Note that for the demonstration pong-like game, we will define two paddle classes, one for the left side of the screen and one for the right side of the screen. Inheritance could have been used to reduce the redundancy of the code, however for this basic example it was not deemed useful.

3.4.2 Collision Detection

The paddle can now be moved to attempt to intercept the bouncing ball that we are reusing from the screensaver demo of iteration 1. However, the ball will pass through the paddle and does not bounce off it. It is now time to implement collision detection.

One basic way to implement the collision between the ball and the paddle would be to pass the references of the paddle to the ball. The ball would then be able to implement a bouncing logic based on overlapping coordinates, similar to how the ball checks for the display's border. Though, this is not a good approach, as it would tightly couple the ***DemoBounceSphere*** class to the ***DemoPaddle*** class. If we wanted to add another paddle, or some sort of obstacle, in the scene, we would have to refactor ***DemoBounceSphere*** and manually add the references to all the new objects. This option would also not be reusable in the game engine.

Instead, we can take advantage of the Composite pattern. There is already a composite node that has references to all the ***AbstractGameObject*** of the scene. We can create a ***CollisionDetector*** class that will encapsulate the collision detection logic and give an instance of this class to the composite node. The ***AbstractGameObject*** that are part of the composite node and that need to check for collisions will then be able to access their parent's ***CollisionDetector*** and use it.

This implementation requires some refactoring. The ***AbstractGameObject*** currently cannot access their parents, the ***CollisionDetector*** class does not exist yet, not all composite nodes should have a ***CollisionDetector***, and ***AbstractGameObject*** require a physical area that can be used for checking for collisions. At the moment, all the ***AbstractGameObject*** of the pong-like game are placed directly under the ***GameEngine*** root node of the composite tree. (Figure 34.)

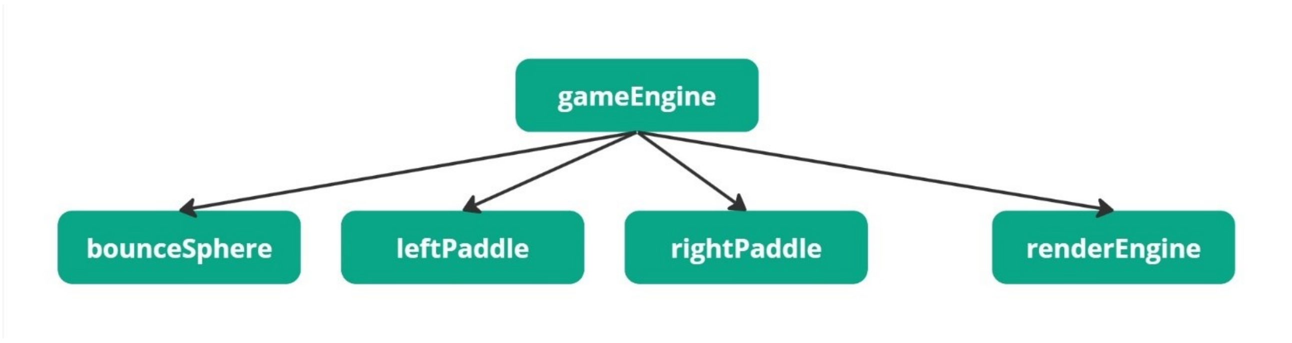
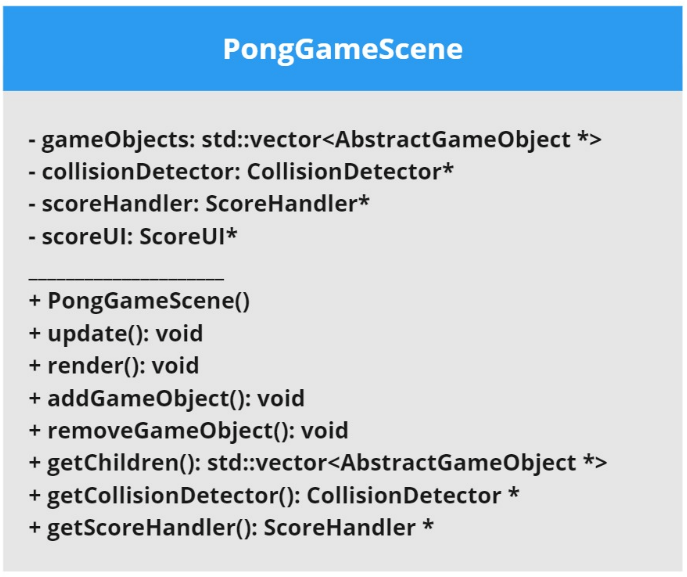


Figure 34. Game engine composite tree without sub-somposite nodes

To improve the separation of concerns and hierarchical architecture of the composite tree, we will implement a new composite node. ***PongGameScene*** will contain all the elements of the pong-like game in its vector of ***AbstractGameObject***, and it also has a ***CollisionDetector*** and a ***Score-Handler*** class to keep track of the score and a ***ScoreUI*** class, an ***AbstractGameObject*** used for displaying the score of the game. (Figure 35.) In the future we will use multiple scenes to display different state of the game, like a start scene, a game scene, or a game over scene. For this reason, we will also implement a ***GameScene*** abstract class.



```
- gameObjects: std::vector<AbstractGameObject *>
- collisionDetector: CollisionDetector*
- scoreHandler: ScoreHandler*
- scoreUI: ScoreUI*
_________________
+ PongGameScene()
+ update(): void
+ render(): void
+ addGameObject(): void
+ removeGameObject(): void
+ getChildren(): std::vector<AbstractGameObject *>
+ getCollisionDetector(): CollisionDetector *
+ getScoreHandler(): ScoreHandler *
```

Figure 35. PongGameScene class diagram

The ***CollisionDetector*** class is also implemented. All it does is define a function that expects two ***AbstractGameObject*** as parameters and then returns whether those ***AbstractGameObject*** are overlapping using the Axis-Aligned Bounding Box (AABB) collision detection method (Gregory 2017, chapter 12). Since the AABB algorithm relies on boxes to check for collision, a new class, the ***HitBox***, was introduced (figure 36). All ***AbstractGameObject*** have a default ***getHitBox()*** method that returns a ***nullptr*** to cover the case where they do not have a ***HitBox***. Developers have the responsibility of instantiating a ***HitBox*** for the ***AbstractGameObject*** that need one, like the paddles and ball of the pong-like game.

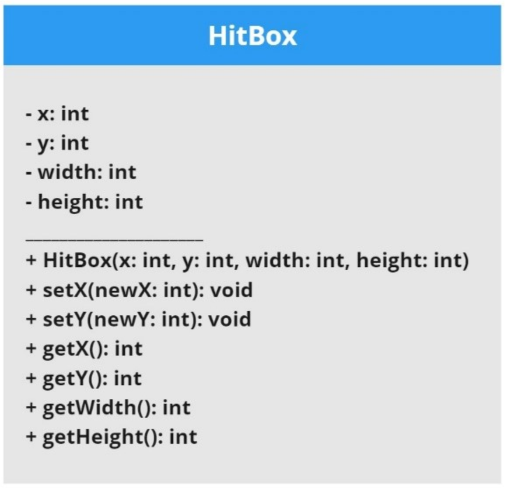


Figure 36. Class diagram of HitBox

Finally, the ***AbstractGameObject*** that need to access the ***CollisionDetector*** are given a reference to their parent. At the moment the reference to the parent is passed in the Sketch. It is not an optimal implementation but sufficient for a proof of concept. In the pong-like game demo, the ***BounceSphere*** uses the ***CollisionDetector*** of the ***PongGameScene*** to check for collisions with the other objects of the scene, in this case, the paddles.

This system works and can be used efficiently by only performing a collision check when needed, although it contains weaknesses. For example, the ***AbstractGameObject*** that checks for collisions only receives a ***Boolean*** as an answer and does not know what it is colliding with. The ***AbstractGameObject*** performing the check is also responsible for defining the logic of iterating through all the objects of the scene, which is not ideal according to Single Responsibility Principle.

3.4.3 Pong Game Demo

The demonstration game is a two-player pong-like game. The left paddle is controlled by the joystick and the right paddle is controlled by two buttons. When a player scores, their score is updated and displayed on their side of the screen. The ball automatically respawns in the center of the screen and starts moving opposite to the direction it was going when the score occurred. The game is endless and there is no other screen than the game screen. A still of the game is shown in figure 37.

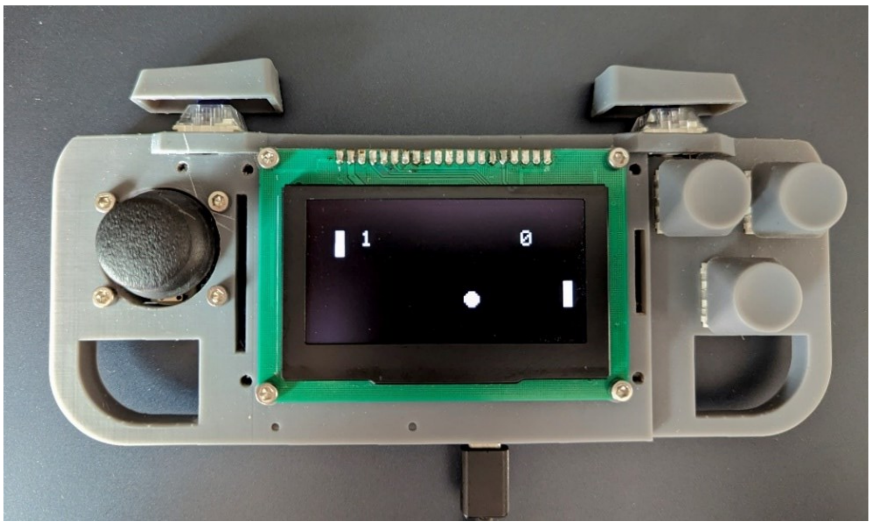


Figure 37. Pong-Like game on the custom esp32 gaming device

3.5 Iteration 3 – Scene Management & Better Collisions

The pong-like game demo works but there can never be a winner when playing the game. No winner detection game logic has been implemented at this point. We can also note that the game starts as soon as the gaming device is powered. This does not constitute the best gaming experience for players. In this iteration, a scene management system will be implemented so that games can have multiple scenes allowing them to launch on a start menu, that can then go into the game itself, and finally to a game-over screen.

3.5.1 Scene Management as State Pattern

The State pattern perfectly fits the requirements we have for scene management. As mentioned by the Gamma, Helm, Johnson, and Vlissides, it is best used when the behavior of an object, called the context, changes according to its state and when the states need to change at runtime (Gamma, Helm, Johnson, & Vlissides 1994, chapter 5).

Let us consider a new class, the ***SceneManager***, that would be the context of the State pattern. The context is composed of all its possible states. The states themselves are also classes, so the context has an attribute for each of its states. In our case, the states are what we have been calling scenes. The ***SceneManager*** has a ***currentScene*** attribute on which it will call the request method. The ***currentScene*** is the state that handles the execution of the method. In our game engine, the methods we want to request are the ***update()*** and ***render()*** method. This means the ***SceneManager*** will actually be a node of the composite tree, so it must implement the ***AbstractGameObject*** interface.

There is however an issue when using one attribute per state. Not all games will have the same number of states or scenes in our case. To solve this issue, we will modify the State pattern implementation with a technique that was already used for our ***InputManager***. Instead of an attribute for each state, we use a ***map*** of ***String*** and ***GameScene***. Our ***SceneManager*** now needs some helper functions to add and remove states, similar to what is implemented in a composite node of the Composite pattern. This solution provides a developer friendly approach. The ***SceneManager*** is part of the game engine and does not need to be changed or implemented by the developers.

The context is only one part of the State pattern implementation. The concrete states need to implement an abstract state interface. We have already made a prototype of the abstract state interface for the scenes during iteration 2. The ***GameScene*** class will be refactored into the base state of the State pattern. Though, instead of making it an interface, it will be an abstract class, meaning that it already defines some functions like the ***update()*** and ***render()*** methods, but also includes some virtual functions that developers can implement to fit the specific needs of their game's scenes. Two new methods, ***onEnterScene()*** and ***onExitScene()***, are added as virtual functions. Those methods will be systematically called when a new scene is set in the ***SceneManager***. Example use case of those methods includes:
- loading assets when entering a scene
- resetting player coordinates and score when exiting a scene
- saving a score when exiting a scene

Finally, the scene management system should be easily accessible in the code. Going from the start menu to the game will probably be handled by a controller class that checks for a specific input, whereas going to the game over screen may result from a collision during the game, be it the ball entering the goal in the pong-like game or the player dying in some platforming game. For that, the Singleton pattern will once again be used. Figure 38 shows the result of the State pattern implementation of the ***SceneManager*** that is also part of the composite tree and that is also a Singleton.

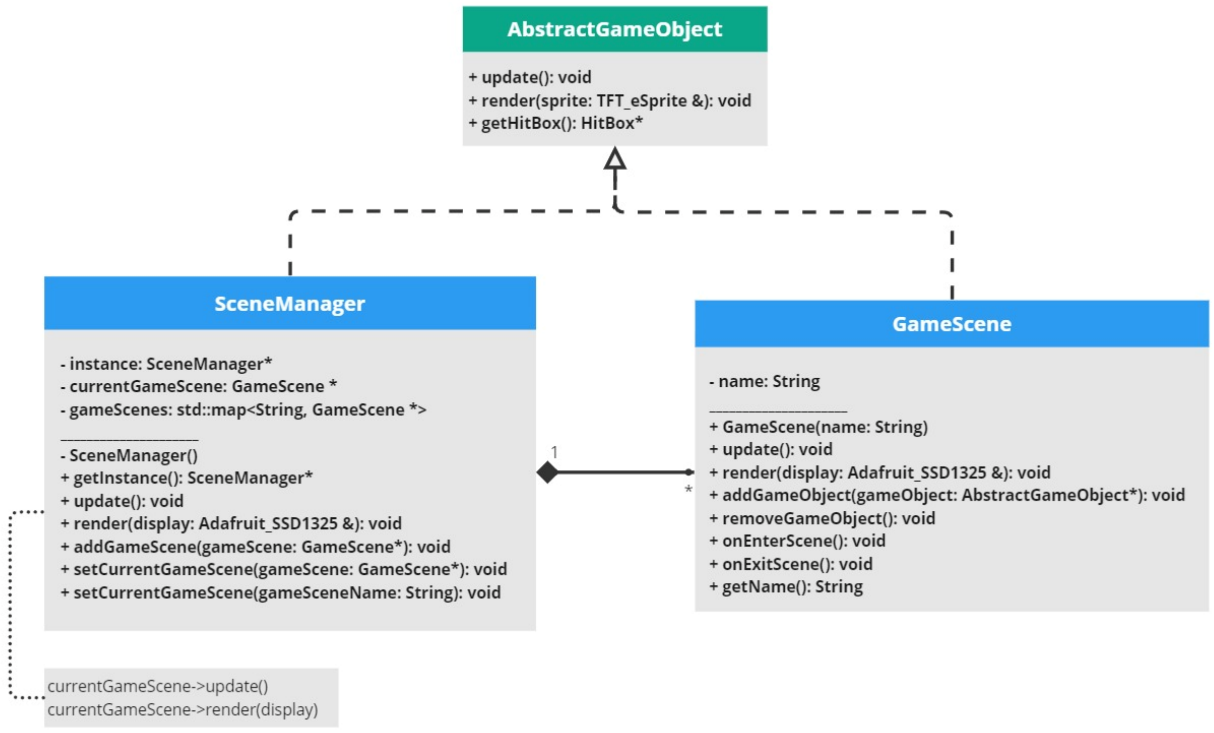

AbstractGameObject
+ update(): void
+ render(sprite: TFT_eSprite &): void
+ getHitBox(): HitBox*
SceneManager
- instance: SceneManager*
- currentGameScene: GameScene *
- gameScenes: std::map<String, GameScene *>
- SceneManager()
+ getInstance(): SceneManager*
+ update(): void
+ render(display: Adafruit_SSD1325 &): void
+ addGameScene(gameScene: GameScene*): void
+ setCurrentGameScene(gameScene: GameScene*): void
+ setCurrentGameScene(gameSceneName: String): void
1
*
GameScene
- name: String
+ GameScene(name: String)
+ update(): void
+ render(display: Adafruit_SSD1325 &): void
+ addGameObject(gameObject: AbstractGameObject*): void
+ removeGameObject(): void
+ onEnterScene(): void
+ onExitScene(): void
+ getName(): String
currentGameScene->update()
currentGameScene->render(display)

Figure 38. UML of scene management State pattern

An example of this implementation can be found in iteration 3 of this project's repository. The example is built upon the pong-like game from previous iterations. In this example there are three different ***GameScene***. Figure 39 shows a snapshot of each scene as well as the finite state machine to illustrate navigation between the scenes.

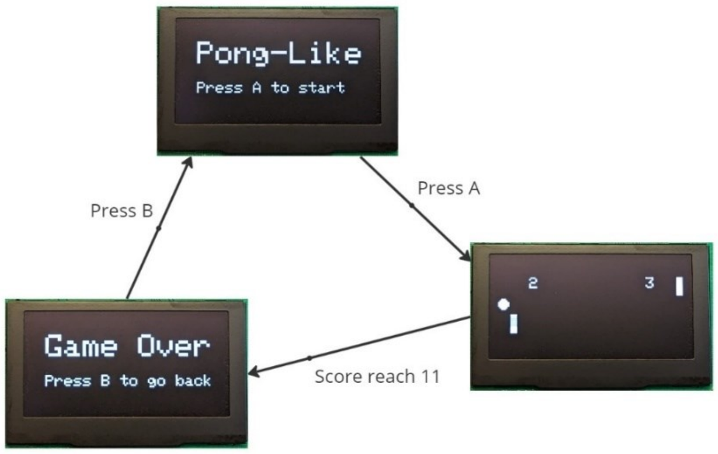


Figure 39. Finite state machine of scenes in Pong-Like demonstration game

Figure 40 shows the composite tree of the pong-like game example. Notice that the ***SceneManager*** has a special role in the tree. It is not a leaf because it holds children in the form of the ***GameScene***, nor is it a composite node because it does not iterate through each of its children. Instead, it manages what branch of the three should be active, and it ensures that only one of its possible paths is active. In figure 40, this is illustrated with grey arrows pointing to the non-active paths.

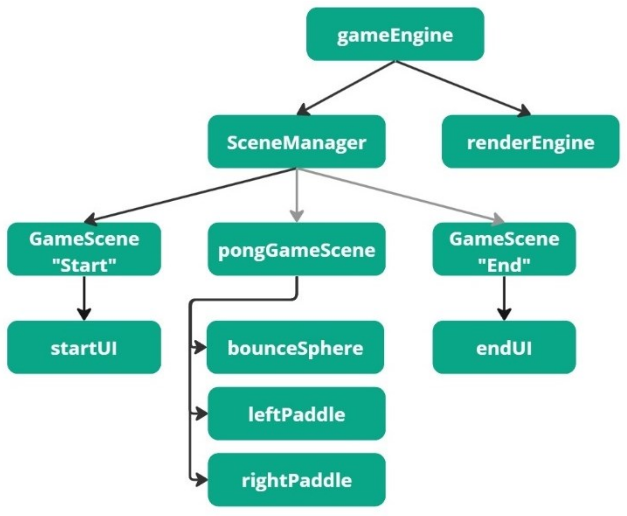


Figure 40. Composite tree of Pong-Like demonstration game

One feature that was not implemented is the ability to share data between states. This common use-case is generally handled by passing the context itself as a parameter to the states. In our case, the context is a Singleton, making it easily accessible. The issue with the game engine approach is that we have already written the ***SceneManager*** context class, and that it should be modified by game developers to include the data they want to share between states. One workaround would be to give the ***SceneManager*** a ***map*** of ***String*** and ***String*** to be used as key-value data storage, shared between the states. However, this does not allow for sharing objects references.

3.5.2 Event Driven Collision Detection with Observer Pattern

The last feature planned for the development process of the game engine was a revision of the collision detection system. It could not be implemented due to time constraints; however, this chapter explains why and how the use of the Observer pattern would improve the collision detection system.

The current collision detection system has several limitations that affect code quality and maintainability. The primary issue is the tight coupling between game objects and the collision detection logic. Each ***AbstractGameObject*** is responsible for checking collisions with other objects in the scene, which violates the Single Responsibility Principle by combining collision detection logic with the object's primary functionality. This approach also leads to duplicated code and a lack of flexibility, as each game object must implement its own collision checking mechanism.

Implementing an event-driven collision detection system using the Observer pattern addresses these issues by decoupling the collision detection logic from individual game objects. In this system, the ***CollisionDetector*** class is responsible for detecting collisions and emitting events when collisions occur. This is in line with the Single Responsibility Principle, as the ***CollisionDetector*** focuses only on collision detection. ***AbstractGameObject*** subscribe to collision events through an ***EventManager***, which handles the registration of listeners and broadcasting of events. This decoupling means that game objects do not need direct references to each other or the collision detector, enhancing modularity and maintainability.

4 Discussions

The discussions chapter reviews the work done during the thesis and discusses topics such as potential future features for the game engine and more design patterns that could be used. It also reflects on the sustainability aspects of the project and ends with the conclusion of the thesis.

4.1 Future Features Development and Patterns

During this thesis, the foundation of the game engine was designed, implemented, and demonstrated. Key features like input management, scene management, core architecture and basic collision detection are functioning. However, many more features could be implemented. This chapter gives some examples of features that could be developed in future work on the project started in this thesis.

Most games and gaming devices have more than one type of output. While the display is used for visuals, adding an amplifier and speaker to the game device would enable sounds to be played. A sound management system would programmatically handle playing sound files. This would imply storing sound files in the project, accessing the sound files, and accessing the sound management system to play, pause and stop sound.

Another interesting feature would be animation management. In our engine, the 2D rendering is mostly handled by the ***Adafruit-GFX-Library***. With animation management, developers could give a set of sprites to a game object and those sprites would then be changed every given amount of time to create an animation. This feature would also most likely require storing image files.

Physics simulation is a common feature of game engines, it simplifies the processus of applying physics to objects to make them move. Simulated physics makes it easier to handle movements like jumps and falls. Physics simulation engines sometimes incorporate particles systems for environmental elements like rain, dust, or splatters. Although, particles would be basic with the low resolution of the OLED display used in our game device. Collision detection is a key element of this feature, although it is only a small part of it.

Networking is another potential feature that could enhance the game engine, especially with the ESP32, capable of networking via Wi-Fi and Bluetooth. Future work could include developing an interface to simplify the connection to Wi-Fi. This UI could be integrated into the composite tree, as one of the scenes of the ***SceneManager***, and be used as a home screen. Networking capabilities could also enable interactions with the internet, such as saving scores online and fetching data into games. These are common use cases for ESP32 embedded systems, and many libraries are available for this purpose. Finally, networking could allow developers to create online multiplayer games. However, this feature would require some sort of online game synchronization system like the development of specific networked game objects that can share their attributes between client devices to ensure synchronization. Unity uses a similar system called Netcode for GameObjects, which provides a framework for building multiplayer games by synchronizing game objects across networked clients (Reeve 2024).

To support the development of these features, various design patterns could be employed. The Observer pattern was already mentioned for an improved version of the collision detection. It could also be used for implementing networked game objects. The State pattern could be used for managing animations. With states corresponding to different animations of a character such as idling, running, attacking, and jumping. For particle emission, that we could implement in the physics simulation engine, the Flyweight pattern would optimize memory usage and performance by sharing common data among particles, reducing redundancy, and improving efficiency.

Finally, additional design patterns could be used to enhance the existing code of the game engine. The Factory pattern could be employed for improving the creation of objects like inputs or game entities, ensuring a clean and modular architecture. The Adapter pattern could be used to make the engine compatible with multiple graphic libraries, such as Bodmer’s and Adafruit’s libraries, by adapting their interfaces to the engine's requirements.

4.2 Use of AI

In the development of this thesis, ChatGPT by OpenAI was used for both the development process of the project and for the redaction of the report. During coding, ChatGPT was mostly used for debugging and refactoring purposes. This allowed for a faster development process. Secondary uses of ChatGPT includes helping with learning C++ specifics and discussing design patterns. It provided valuable insights and suggestions. As mentioned in iteration 1, the code used for the bouncing ball screensaver demonstration was generated by ChatGPT.

In the redaction phase, ChatGPT was used to improve readability by refactoring complex sentences, and by suggesting synonyms and alternate formulations. While AI was used to improve the form of the content, it was not involved in content generation.

4.3 Sustainability

This chapter addresses the environmental sustainability aspects of software development in general. It is also used to discuss the educational sustainability of the game engine developed in this thesis.

The ESP32 is by design a low-power device. This contrasts with the trend of personal computers becoming always more powerful and energy consuming. Optimization has always been a key aspect of software development. It used to be important because hardware had strong limitations and software developers had to be careful about the efficiency of their software otherwise it would not fit on the machine. Nowadays, hardware is so efficient that in many cases, inefficient code might go unnoticed, and yet it is the responsibility of developers to care about the efficiency of their software for energy consumption reasons. Design patterns, like the Flyweight pattern, can help optimize resource usage and minimize environmental impact.

Education plays a crucial role in sustainability by cultivating knowledge and skills development. To support this, the game engine developed in this thesis is openly available on GitHub under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

4.4 Conclusions

The main objectives of this thesis were to develop a 2D game engine for ESP32 and to demonstrate how design patterns can be used to create a clean and maintainable codebase. To realize those main objectives, a secondary one had to be completed. Building a gaming device around an ESP32 was the first task undertaken during this thesis. It was mostly successful as the device was prototyped and improved over two iterations. A major issue with the hardware arose during the development work. The slow rendering of the initial display gave the opportunity to investigate the working of lower-level protocols used in embedded software development by means of testing. This valuable lesson will be a reminder to not overlook low-level protocols.

Regarding the game engine, the implemented features are the core loop, input management, 2D rendering, collisions detection and scene management. One of each type of design pattern was successfully implemented into the game engine. The input management system and the scene management system both use a creational pattern, the Singelton. The core of the engine is built upon the Composite pattern, a structural one. And a behavioral pattern is also used in the scene management system with a State pattern. The Observer pattern could not be implemented, although the benefits of its usage have been described. Other patterns were mentioned for future features implementation.

This thesis concludes a three-year adventure of learning software development. It has been the opportunity to review and present all the concepts learned through this journey, from the fundamental class of OOP to the intricate combinations of design patterns used for architecting the game engine. It emphasizes all that is dear to the author in the world of software development and serves as the ideal conclusion to their bachelor formation.

Sources

Adafruit s.a. Adafruit Monochrome 2.7" OLED Graphic Display. URL: https://www.adafruit.com/product/2674 . Accessed: 11 May 2024.

Adafruit 2023. Adafruit-GFX-Library. URL: https://github.com/adafruit/Adafruit-GFX-Library . Accessed: 22 May 2024.

Adafruit 2024. Adafruit_SSD1325_Library. URL: https://github.com/adafruit/Adafruit_SSD1325_Library . Accessed: 24 May 2024.

AITO. 2022. The AITO Dahl-Nygaard Prize Winners For 2006. URL: https://sites.google.com/aito.org/home/aito-dahl-nygaard/2006-winners . Accessed: 28 May 2024.

Amazon 2023. ESP32-S3-WROOM-1-N16R8 development board. URL: https://www.amazon.com/YEJMKJ-ESP32-S3-DevKitC-1-N16R8-Development-ESP32-S3-WROOM-1-Microcontroller/dp/B0CDRM6BGQ . Accessed: 11 May 2024.

Andrade, A. 2015. Game engines: a survey. EAI endorsed transactions on serious games, 2, 6, pp. 1-6.

Arduino 2023. Visual Studio Code extension for Arduino. URL: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.vscode-arduino . Accessed: 13 May 2024.

arm s.a. What is a Gaming or Game Engine?. URL : https://www.arm.com/glossary/gaming-engines . Accessed: 26 March 2024.

benricok 2023. MicroGameConsole. URL: https://github.com/benricok/MicroGameConsole . Accessed: 24 May 2024.

Bodmer 2024. TFT_eSPI. URL: https://github.com/Bodmer/TFT_eSPI . Accessed: 22 May 2024.

Brett, D. 2024. What Are Embedded Systems?. URL: https://www.trentonsystems.com/en-gb/blog/what-are-embedded-systems . Accessed: 4 May 2024.

Brian, P. 2023. Color Depth. URL: https://techterms.com/definition/color_depth . Accessed: 22 May 2024.

CircuitSchools Staff. 2022. What is ESP32, how it works and what you can do with ESP32? URL: https://www.circuitschools.com/what-is-esp32-how-it-works-and-what-you-can-do-with-esp32/ . Accessed: 29 May 2024.

Codacy 2023. What Is Clean Code? A Guide to Principles and Best Practices. URL: https://blog.codacy.com/what-is-clean-code . Accessed: 6 April 2024.

Codecademy Team 2023. What is Inheritance in Object-Oriented Programming? URL: https://www.codecademy.com/resources/blog/what-is-inheritance/ . Accessed: 30 March 2024.

corax89 2021. esp8266_game_engine. URL : https://github.com/corax89/esp8266_game_engine . Accessed: 24 May 2024.

Elecrow s.a. TFT LCD Module Display. URL: https://www.elecrow.com/3-5inch-480x320-mcu-spi-serial-tft-lcd-module-display.html . Accessed: 11 May 2024.

ESPBoards 2023. ESP-IDF vs Arduino Core: Which Framework to Choose in 2023. URL: https://www.espboards.dev/blog/esp-idf-vs-arduino-core/ . Accessed: 13 May 2024.

ESPRESSIF s.a. Get Started. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html . Accessed 13 May 2024.

fdivitto 2023. FabGL. URL: https://github.com/fdivitto/FabGL . Accessed: 24 May 2024.

Gamma,E., Helm,R., Johnson,R., & Vlissides,J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. Place of Publication unknown. E-book. Accessed: 30 March 2024.

GeeksforGeeks 2022. Difference between #define and const in C? URL: https://www.geeksforgeeks.org/diffference-define-const-c/ . Accessed: 14 May 2024.

Gianou 2024. Haaga-Helia-IoT-Experimental-Project. URL : https://github.com/Gianou/Haaga-Helia-IoT-Experimental-Project . Accessed: 24 May 2024.

Giant Sloth Games. April 2023. So you want to make a Game Engine!? (WATCH THIS before you start). Online video. URL: https://www.youtube.com/watch?v=3rcka6P2cVI . Accessed: 25 May 2024.

Gregory,J. 2017. Game Engine Architecture. 2nd Edition. A K Peters/CRC Press. Place of publication unknown. E-book. Accessed: 26 March 2024.

Gunnarsson, K. & Herber, O. 2020. The Most Popular Programming Languages of GitHub's Trending Repositories. Bachelor's thesis. School of Electrical Engineering and Computer Science (EECS). URL: https://www.diva-portal.org/smash/get/diva2:1463849/FULLTEXT01.pdf . Accessed: 5 May 2024.

Herdwaria, S. 2023. Exploring the Need for Object-Oriented Programming. URL: https://dzone.com/articles/exploring-the-need-of-object-oriented-programming . Accessed: 28 May 2024.

Lutkevich, B. 2019. microcontroller (MCU). URL: https://www.techtarget.com/iotagenda/definition/microcontroller . Accessed: 4 May 2024.

Martin, R. C. 2008. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. Place of Publication unknown. E-book. Accessed: 6 April 2024.

MaxGaming s.a. BOX Navy Clicky Switch from Novelkeys. URL: https://www.maxgaming.fi/fi/switches/box-navy-clicky-switch . Accessed: 11 May 2024.

mdn web docs 2023. OOP. URL: https://developer.mozilla.org/en-US/docs/Glossary/OOP . Accessed: 13 April 2024.

MKS075 2024. Difference between Inheritance and Polymorphism. URL: https://www.geeksforgeeks.org/difference-between-inheritance-and-polymorphism/ . Accessed: 13 April 2024.

OODesign s.a. Design Patterns. URL: https://www.oodesign.com/ . Accessed: 30 March 2024.

Oracle s.a. What is IoT?. URL: https://www.oracle.com/internet-of-things/what-is-iot/ . Accessed: 4 May 2024.

Reeve, A. 2024. About Netcode for GameObjects. URL: https://docs-multiplayer.unity3d.com/netcode/current/about/ . Accessed: 31 May 2024.

Refactoring Guru s.a. a. Composite. URL: https://refactoring.guru/design-patterns/composite . Accessed: 20 May 2024.

Refactoring Guru s.a. b. Composite Pattern from the Refactoring Guru. URL: https://refactoring.guru/design-patterns/composite . Accessed: 20 May 2024.

Refactoring Guru s.a. c. Hello, world!. URL: https://refactoring.guru . Accessed: 28 May 2024.

Saraswat, M. M. 26 November 2023. Frame Rate - Everything You Need To Know. 100ms blog. URL: https://www.100ms.live/blog/frame-rate#commonly-used-frame-rates . Accessed: 20 May 2024.

Sintosen palvelut s.a. Joystick mounted on a PCB. URL: https://kauppa.sintosen.com/product/1926/ . Accessed: 11 May 2024.

Thesis Coordinators 2022. Guidelines for Long Reports and Theses. Haaga-Helia. Helsinki. URL: https://www.haaga-helia.fi/sites/default/files/file/2022-02/Guidelines_for_long_reports_and_theses_2022_0.pdf . Accessed: 7 May 2024.

Ubahnverleih. 2018. ESP32 Espressif ESP-WROOM-32 Dev Board. URL: https://commons.wikimedia.org/wiki/File:ESP32_Espressif_ESP-WROOM-32_Dev_Board_%282%29.jpg . Accessed: 29 May 2024.

Weisfeld, M. 2019. The Object-Oriented Thought Process, 5th Edition. Addison-Wesley Professional. Place of Publication unknown. E-book. Accessed: 28 March 2024.

W3 Schools s.a. C++ Access Specifiers. URL : https://www.w3schools.com/cpp/cpp_access_specifiers.asp . Accessed: 30 March 2024.

Appendices

Appendix 1. Link to GAMESP32 GitHub Repository

URL: https://github.com/Gianou/GAMESP32