

REPORT PROJECT

Tower Defense Game

Course name: Object–Oriented Programming

Course ID: CSC10003

Instructors: Toan–Thinh Truong

Group 1:

Name	ID
Nguyen Anh Thu	23127266
Tran Hai Duc	23127173
Bui Quang Hung	23127193

Acknowledgment

We would like to thank Mr. Truong Toan Thinh for his invaluable guidance and support during our thesis project. His expertise in object-oriented programming and design patterns has been instrumental in our game development. This project has allowed us to apply theoretical knowledge to practical work and learn new skills along the way. We are deeply grateful for his mentorship.

Table of Contents

Acknowledgment	2
Table of Contents	3
Chapter 1: Introduction	5
1.1 Project Objective	5
1.2 Overview	5
1.2.1 Game Mechanics and Rules	5
1.2.2 Key Features	6
1.2.3 Guide to installation and use	6
1.3 Planning and workflow	9
1.3.1 Team Role Distribution	9
1.3.2 Collaborative Workflow	10
1.3.3 Product development plan	11
Chapter 2: Requirements Analysis	13
2.1 Game Concept	13
2.2 User Stories	13
2.3 Technical Requirements	14
2.3.1 Coding	14
2.3.2 General Rules:	15
2.3.3 Game Requirements:	15
2.3.4 Programming Language	15
Chapter 3: System Design	16
3.1 Class Diagram	16
3.1.1 UML	16
3.1.2 User flow	16
3.2 Design Patterns	17
3.3.1 Patterns Used	17
3.3.2 Purpose and Application in the System	21
3.4 Content game	22
3.5 UX/UI design	24
Chapter 4: Implementation	27
4.1 Development Tool	27
4.1.1 IDEs	27
4.1.2 Compilers	27
4.1.3 Environment and libraries setup	27
4.2 Key Features	31
4.2.1 Load and Continue Game	31
4.2.2 Leaderboard System	32
4.2.3 Audio Settings	33

4.2.4 Tower Placement Mechanics	33
4.3 Programming Techniques	34
4.3.1 Callback Mechanism	34
4.3.2 Memory Management with Pointers	35
4.3.3 Threading Techniques	35
4.3.4 Dependency Injection	37
4.4 Algorithms	38
4.4.1 Projectile Trajectory Calculation	38
4.4.2 Logic for Enemies	38
4.4.3 Bullet Logic:	39
4.4.4 Sorting Logic	39
4.5 Graphics and Rendering	39
4.5.1 Buffered Rendering	39
4.5.2 Drawing Algorithm	41
4.5.3 GDI+ Functions	42
Chapter 5: Conclusion	43
5.1 Project Outcome	43
5.1.1 Summary of Success	43
5.1.2 Features Achieved	43
5.2 Future Improvements	43
5.2.1 Suggestions for Enhancement	43
5.2.2 Additional Features for Future Development	44
REFERENCE	45

Chapter 1: Introduction

1.1 Project Objective

The objective of this project is to apply a combination of techniques, data structures, and object-oriented programming (OOP) knowledge to create a basic yet functional tower defense game. Through this project, we aim to:

- Maximize Effort and Learning
- Strive to perform to the best of our abilities, maximizing our understanding and skills to complete the project as effectively as possible.
- Develop Comprehensive Knowledge in OOP and Game Development.
- Utilizing design patterns in game development

By the end of this project, we seek to gain a well-rounded understanding of object-oriented principles and core game development techniques, providing a foundation for future projects and learning.

1.2 Overview

This tower defense game project is designed to immerse players in strategic decision-making, defending a base against waves of enemies by building and upgrading defensive towers. The project focuses on integrating foundational coding techniques, data structures, and object-oriented programming concepts. The goal is to create an engaging game experience with a balance of strategy and action, incorporating multiple gameplay features and enemy dynamics that challenge players across progressively difficult levels.

1.2.1 Game Mechanics and Rules

The core mechanics of the game revolve around players defending a designated area (the "base") by strategically placing various towers. These towers are designed to automatically detect, target, and attack incoming enemies within a specified range. The game includes the following key rules and mechanics:

- **Enemy Waves and Phases:** Each level is structured around waves of enemies, with several phases per wave. Enemies move toward the base along a defined path. The game includes different enemy types with varying characteristics, such as movement speed, health, and point values.
- **Tower Targeting and Range:** Towers are programmed to automatically detect and target the closest enemy within a specified range. The targeting logic ensures that towers prioritize threats based on proximity, optimizing defensive efficiency.
- **Bullet Tracking and Collision:** Bullets fired by towers travel in a straight line toward targeted enemies. Bullet movement is calculated based on the coordinates of both the bullet and its target. When a bullet collides with an enemy (i.e., their

coordinates match), it deals damage and is then removed from the game, freeing up memory.

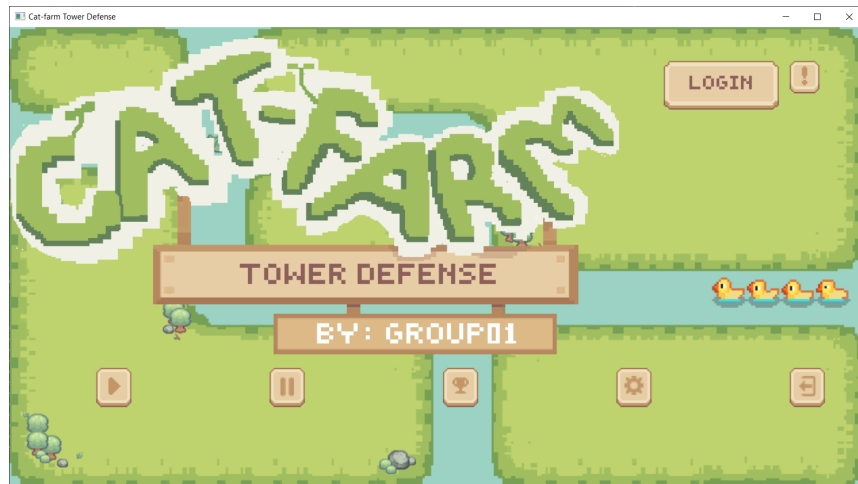
- **Enemy Movement and Pathfinding:** Enemies move along pre-defined paths created by connecting specific coordinates from the level map data. They navigate from point to point until they reach the base or are eliminated by towers.
- **Scoring System and Progression:** Players earn points by defeating enemies, with each enemy type offering a different score. When a player successfully defends the base through all waves, they advance to the next level, facing increasingly difficult challenges.

1.2.2 Key Features

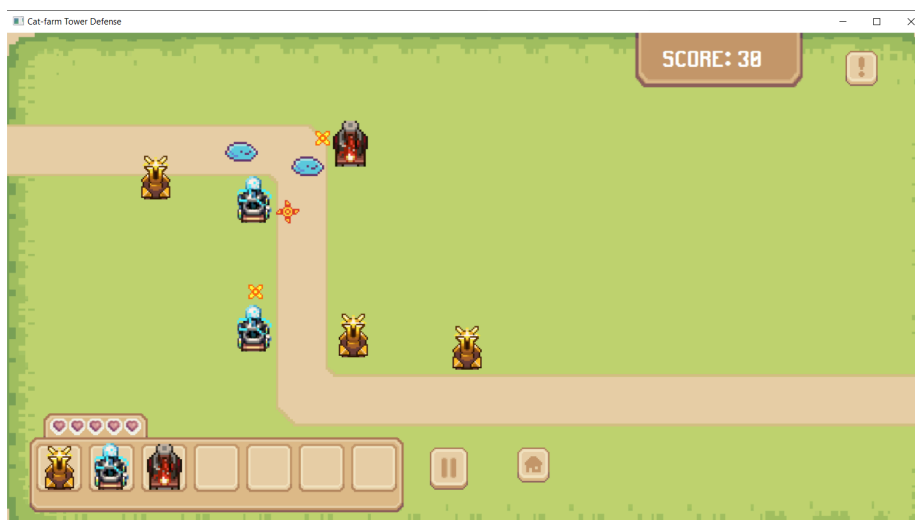
- **Varied Tower Types and Upgrades:** Different types of towers provide unique abilities, enabling players to strategize tower placements based on enemy characteristics. Certain levels may unlock special towers with enhanced attack ranges, fire rates, or damage outputs.
- **Dynamic Enemy Types and Boss Levels:** Enemy units vary in speed, health, and point value. Higher levels introduce specialized enemies, including fast-moving enemies, high-health bosses, and potentially enemies with unique abilities that require specific strategies to defeat.
- **Level-Based Structure with Progressive Difficulty:** The game includes multiple levels, each with distinct layouts, enemy patterns, and increasing difficulty. Level objectives and enemy waves vary, ensuring that players adapt their strategies to match each unique challenge.
- **User Management System:** leaderboards are included to show top scores across rounds, promoting competition and replayability.
- **Settings:** Players have control over game settings, including volume levels, background music.
- **Mouse-driven gameplay:** Offer mouse-based interaction for gameplay (click, drag-drop)
- **Double Buffering Technique:** To create a smooth visual experience, the game utilizes a double buffering technique on the console screen. This optimizes the update rate of game visuals by alternating between two screen buffers, reducing flickering and enhancing player experience.

1.2.3 Guide to installation and use

- Locate the project directory and open the .sln file.
- Within the IDE, initiate the build and run process.
- Upon the appearance of the game window, employ your **mouse** to engage with the game.
- Execute the game's functions by **clicking on the corresponding buttons**.



- Click on the **play button** to **start** the game (option 1)
- Choose any map to play
- Place the tower by drag-and-drop your mouse
- Click Play button to start the game



- The game will automatically save when you exit



- To resume a saved game, click on the continue button (option 2)



- To turn the sound on or off, select the setting button (option 3)



- To exit the game, select the last button, exit

1.3 Planning and workflow

1.3.1 Team Role Distribution

Role	Responsibilities	Assignment	PIC	Support
Project Manager	Allocates resources, shapes priorities, coordinates interactions with the customers and users, and generally tries to keep the project team focused on the right goal. The Project Manager also establishes a set of practices that ensure the integrity and quality of project artifacts.	Coordinating tasks, tracking progress, and ensuring that deadlines were met.	Nguyen Anh Thu	
Architect	Leads and coordinates technical activities and artifacts throughout the project. The Architect establishes the overall structure for each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between these major groupings. Thus, in contrast with the other workers, the Architect's view is one of breadth, as opposed to depth.	Architect Design. Design the core system architecture and define the relationships between classes.	Bui Quang Hung	Tran Hai Duc, Nguyen Anh Thu
Designer	Defines the responsibilities, operations, attributes, and relationships of one or several classes and determines how they should be adjusted to the implementation environment. In addition, the designer may have responsibility for one or more design packages or design	Design and User Interface. Provide feedback on user experience and aesthetics.	Tran Hai Duc	Nguyen Anh Thu

	subsystems, including any classes owned by the packages or subsystems.			
Tester	Responsible for executing testing, including test set-up and execution, evaluation of test execution and recovery from errors, and assessing the results of test and logging identified defects.	Testing and Quality Assurance. Overseeing testing protocols, with additional support in identifying and fixing bugs.	Nguyen Anh Thu	Bui Quang Hung, Tran Hai Duc

1.3.2 Collaborative Workflow

Project **Phases:**

- o **Planning:** Initial planning meetings were held to outline the project scope, set deadlines, and assign responsibilities.
- o **Design and Development:** Architecture and game design were implemented based on the outlined class structures and gameplay requirements.
- o **Testing:** Each feature was tested in multiple iterations to ensure reliability, with collaboration between the testers and developers to address any issues.

Tools and Communication: We utilized **Discord** for daily communication, **GitHub** for version control, and **Google Docs** for task management. These tools allowed for clear tracking of responsibilities, ensuring smooth transitions between phases.

Synchronization and Decision-Making: We held in-person meetings once a week, with online feedback on progress every Thursday. During these meetings, all members contributed feedback on design, implementation, and testing results, making collaborative decisions when adjustments were needed.

Milestone Tracking: Progress was tracked using Google Docs, where tasks were marked as complete only after being reviewed by the respective lead.

1.3.3 Product development plan

Week	Target
1 (30/9 - 6/10)	<ul style="list-style-type: none">- Study OOP Concepts: Deepen understanding of Object-Oriented Programming principles.- Research and Planning: Plan the design and development phases for the product.- Task Assignment: Distribute roles and responsibilities among team members.
2 (7/10 - 13/10)	<ul style="list-style-type: none">- Product Prototype Design:<ul style="list-style-type: none">· UX/UI Design· User Flow Design· Game Logic Design- Git Introduction: Familiarize the team with version control using Git.- Code Style Standards: Standardize variable naming conventions, data types, and binary file storage formats.- Set Up Development Environment: Prepare the necessary development tools and environment.
3 (14/10 - 20/10)	<ul style="list-style-type: none">● Code and develop basic classes (object classes)● Design file-saving structure● Code the interface● Research reading and displaying .bmp files on the console

4 (21/10 - 27/10)	<ul style="list-style-type: none"> ● Conducted tests and demonstrated code logic. ● Designed components for characters. ● Coded algorithms for bullets and soldier movement paths.
5 (28/10 - 3/11)	<ul style="list-style-type: none"> ● Demonstrated Map 1. ● Coded the interface, navigation menu, and main menu. ● Completed the coding for Map 1. ● Adjusted sound settings. ● Finished designing all maps and objects.
6 (4/11 - 10/11)	<ul style="list-style-type: none"> ● Coded Map 2, Map 3, and Map 4. ● Implemented win/loss logic. ● Fixed bugs and tested algorithms.
7 (11/11 - 17/11)	<ul style="list-style-type: none"> ● Coded: save game, login, and registration features. ● Implemented advanced functionalities. ● Fixed bugs and tested algorithms.
8 (18/11 - 24/11)	<ul style="list-style-type: none"> ● Fixed bugs and tested thoroughly. ● Prepared the product for submission (slides, presentation, demo video).
9 (25/11 - 1/12)	<ul style="list-style-type: none"> ● Documented the project. ● Prepared for the presentation and product demonstration.
10 (2/12 - 8/12)	<ul style="list-style-type: none"> ● Prepared for the presentation and product demonstration.

Chapter 2: Requirements Analysis

2.1 Game Concept

The core mechanic of our tower defense game centers around player strategy in selecting and placing towers. At the beginning of each round, players choose specific towers from a set of options, each with unique attributes and firing capabilities. Once towers are strategically placed, players observe as the towers automatically engage with incoming waves of enemies. This setup allows the player to witness the effectiveness of their tower placement and strategy without needing direct control over the towers during the wave. The flow continues with successive rounds, where increasing enemy difficulty challenges the player's choices, encouraging tactical adjustments.

2.2 User Stories

- **Game Script for Tower Defense:**

+ **Title:** Cat-Farm Defense

+ **Introduction:**

The peaceful Cat-Farm, home to adorable cats and flourishing crops, is under constant threat from waves of menacing creatures! These monsters have taken a liking to the farm's fresh produce and are determined to overrun it. Your mission: help Cat-Farm build powerful towers and fend off the invaders to protect the cats and their land.

+ **Objective:**

Strategically place and upgrade towers around the farm to block the creatures' path and prevent them from reaching the heart of Cat-Farm. With each wave, enemies grow stronger, so use your resources wisely to upgrade your defenses and stay ahead!

+ **Gameplay Mechanics:**

- Tower Placement: Choose the best spots for your towers along the monsters' route for maximum impact.
- Enemy Waves: Face different types of monsters in each wave, including fast creatures, tank-like beasts, and rare bosses with special abilities.
- Tower Upgrades: Collect resources by defeating enemies, then use them to upgrade towers for increased range, damage, and special abilities.
- Winning Condition: Defeat all enemies in each level to protect the farm and progress to the next, more challenging stage.

+ **Endgame:**

As you successfully defend Cat-Farm through multiple levels, the farm gradually expands, allowing more advanced defenses and powerful towers. Can you defend Cat-Farm through all levels and secure the cats' home for good? The cats are counting on you!

2.3 Technical Requirements

2.3.1 Coding

Class	Description	Requirements
cPoint	Represents a point in 2D space.	<ul style="list-style-type: none">- Use getter/setter methods- Validate input data
cTool	Provides tools and utility functions for the game.	<ul style="list-style-type: none">- Use static functions- Contain only general logic
cEnemy	Manages in-game enemies.	<ul style="list-style-type: none">- Optimize movement logic
cTower	Manages defensive towers.	<ul style="list-style-type: none">- Optimize target acquisition
cBullet	Manages bullet types fired from towers.	<ul style="list-style-type: none">- Optimize bullet movement- Free memory when bullet is destroyed- Subclass of cTower
cGame	Manages the overall game logic.	<ul style="list-style-type: none">- Optimize game loop- Coordinate game objects efficiently
MainPrg	Main program that initializes and runs the game.	<ul style="list-style-type: none">- Contain only initialization logic- Transfer complex logic to other
cPlayer	Manages player information.	<ul style="list-style-type: none">- Store player data- Implement login/logout methods

2.3.2 General Rules:

- Naming Conventions: Use camelCase for variables and functions according to coding standards.
- Comments and Documentation: Ensure each method and class has a clear description.
- Version Control (Git): Commit frequently with clear messages. Develop each feature in a separate branch (featX, where X is the feature name) and then merge into the `dev` branch. Only the team leader may merge into the `main` branch.
- Testing and Optimization: Test thoroughly before merging, ensuring optimal performance.

2.3.3 Game Requirements:

- Implement all core functions.
- Collision effects.
- Save/Load game functionality.
- Pre-game setup: display level details (characters, map, available tower positions) for player selection.
- Include at least 4 levels.
- Menu functionality.
- Sound options: background music, gameplay sounds with on/off modes.
- Additional features as required.

2.3.4 Programming Language

In developing our Tower Defense game, we chose C++ as the primary programming language due to its performance efficiency and control over system resources. C++ allows us to optimize game performance, which is crucial for real-time gameplay, especially as the complexity of the game increases with more towers and enemies.

C++ Advantages:

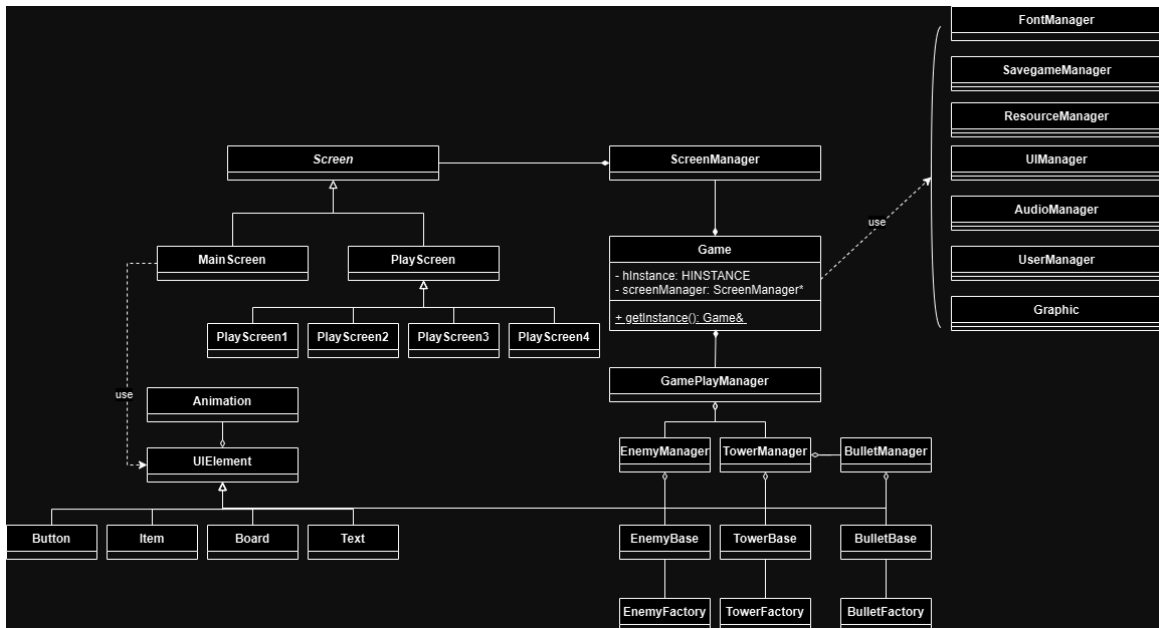
- **Performance:** C++ is a compiled language, providing faster execution speeds, which is essential for a smooth gaming experience.
- **Object-Oriented Programming:** The object-oriented nature of C++ helps us organize code into reusable classes, facilitating better management of game components such as towers, enemies, and projectiles.

Chapter 3: System Design

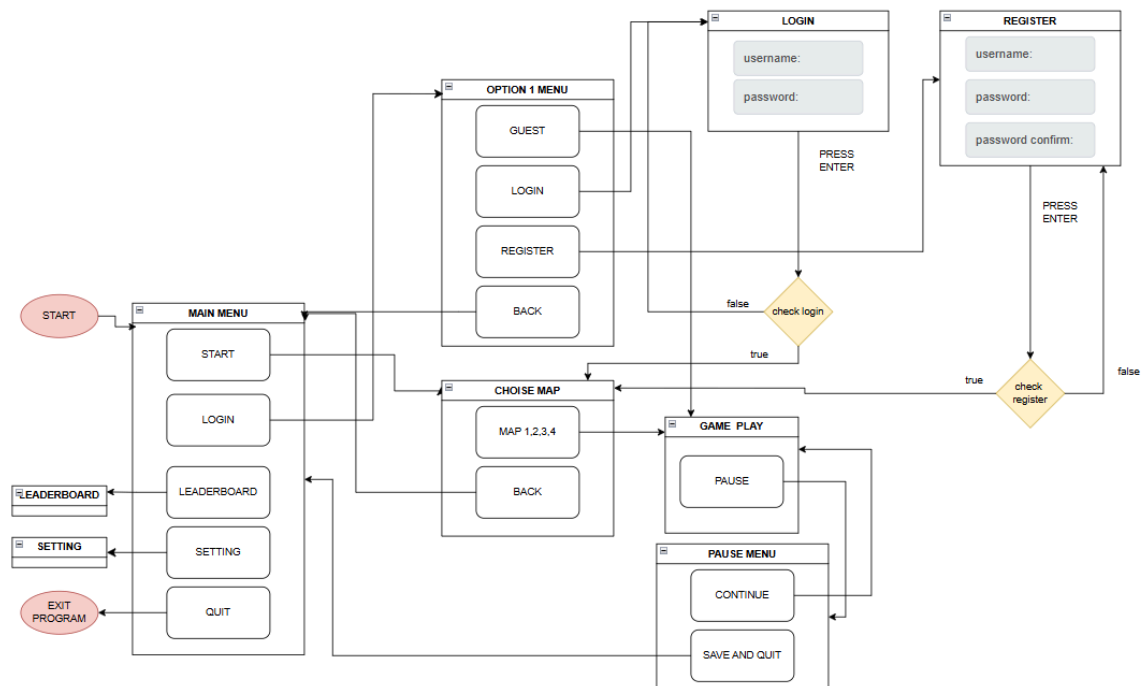
3.1 Class Diagram

3.1.1 UML

- General system diagram of the game



3.1.2 User flow



- **Mainboard Screen**
 - **Login**
 - **Start Game**
 - **Continue**
 - **Leaderboard**
 - **Setting**
 - **Exit**
- **Register Screen**
 - **Username**
 - **Password**
 - **Password Confirmation**
- **Login Screen**
 - **Username**
 - **Password**
 - **Register**
- **Leaderboard Screen**
- **Setting Screen**
 - **Volume**
 - **Background Music**
 - **Game Speed (x1, x1.5, x2)**
- **Game Setup Screen**
 - **Instructions Panel**
 - **Placement Controls**
 - **Temporary Save for Placement**
 - **Start Game (Play Button)**
- **Game Screen**
 - **Pause**
 - **Menu (Continue, Main Screen)**
 - **Auto-save (If logged in)**
- **End Game Screen**
 - **Win or Lose**
 - **Menu (Next Level, Exit, Display Score)**
- **Choose Map Screen**
 - **Level Selection**
- **Load Game Screen**
- **Select Level and Resume Progress**

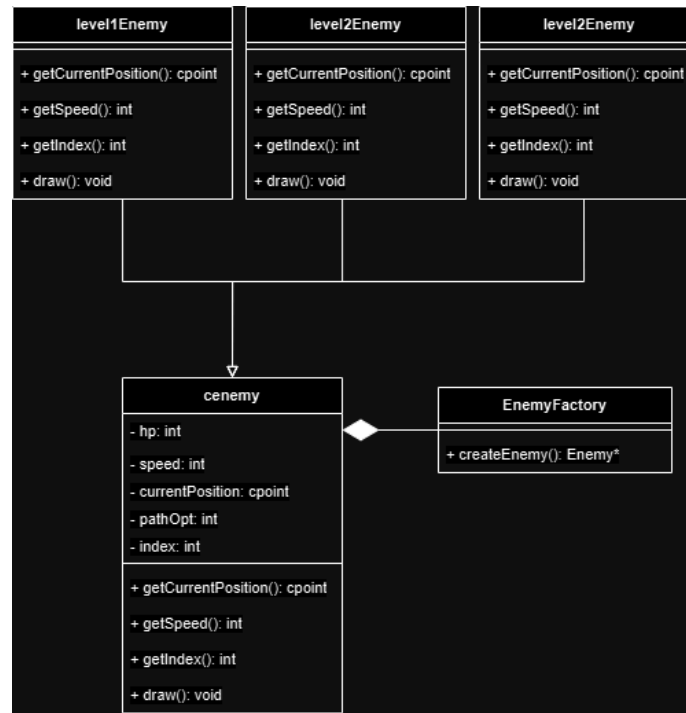
3.2 Design Patterns

3.3.1 Patterns Used

- **Creational Patterns**

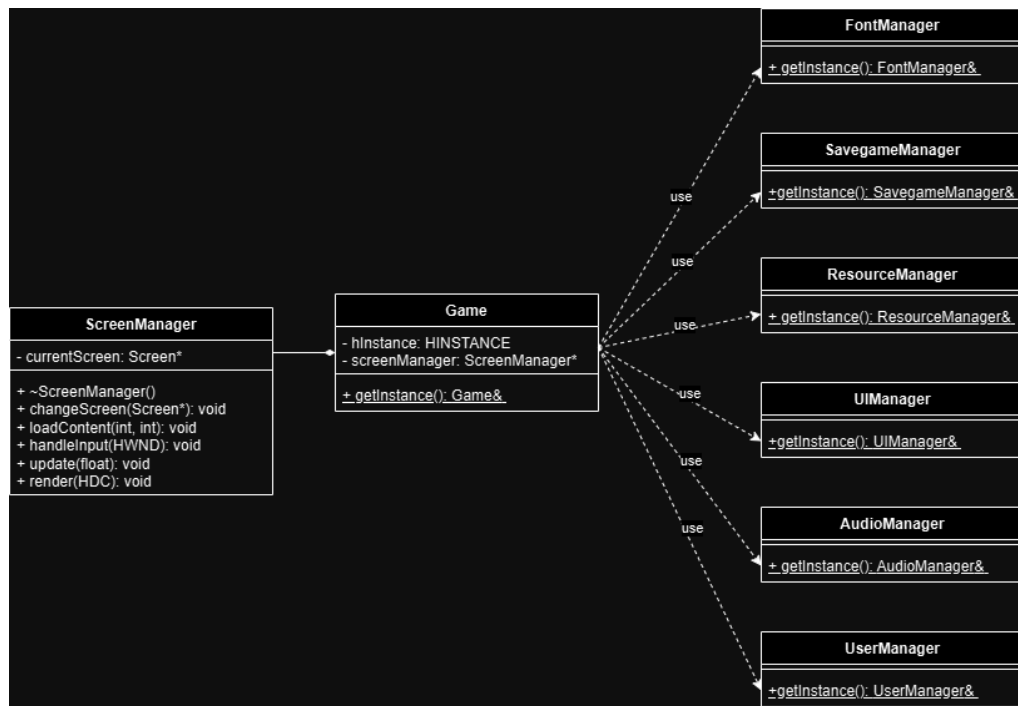
Factory Method:

- We used the Factory Method pattern to create different types of towers. Instead of instantiating tower objects directly, the game calls a factory class that determines which specific tower to create based on player selection. This approach simplifies the addition of new tower types in the future without altering existing code.



Singleton:

- To ensure that only one state is active at a time. The State Manager handles state transitions, such as moving from the Main Menu to Gameplay, and centralizes control over updates and rendering. By implementing the Singleton pattern, the State Manager provides a globally accessible instance, simplifying communication and ensuring consistency across the game. This design enhances scalability and maintains efficient game flow management.



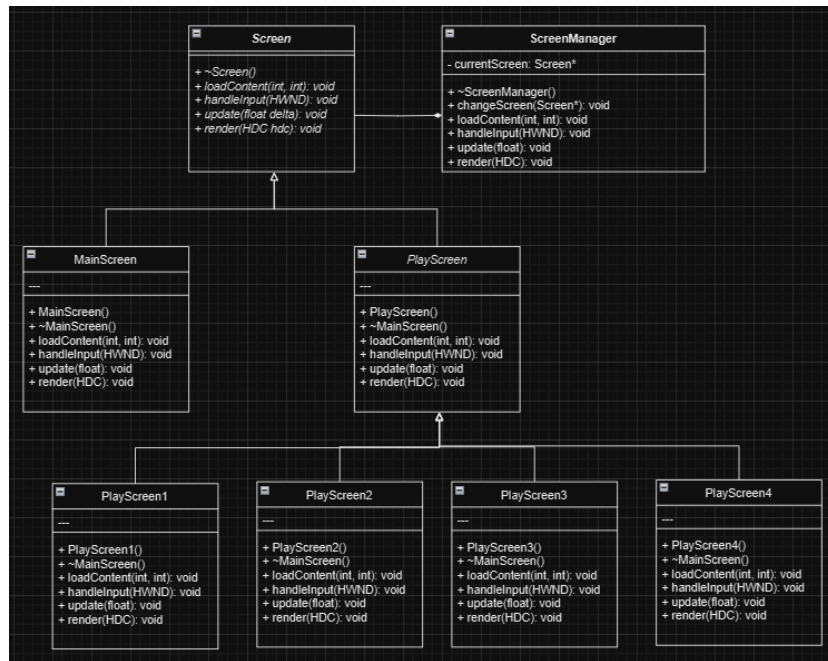
- Behavioral Patterns

+ Observer:

- The Observer pattern is used to manage the game state and notify different components when a change occurs. For instance, when a tower is placed or upgraded, the game UI updates to reflect the new status. This pattern helps maintain loose coupling between the game logic and the user interface.

+ State:

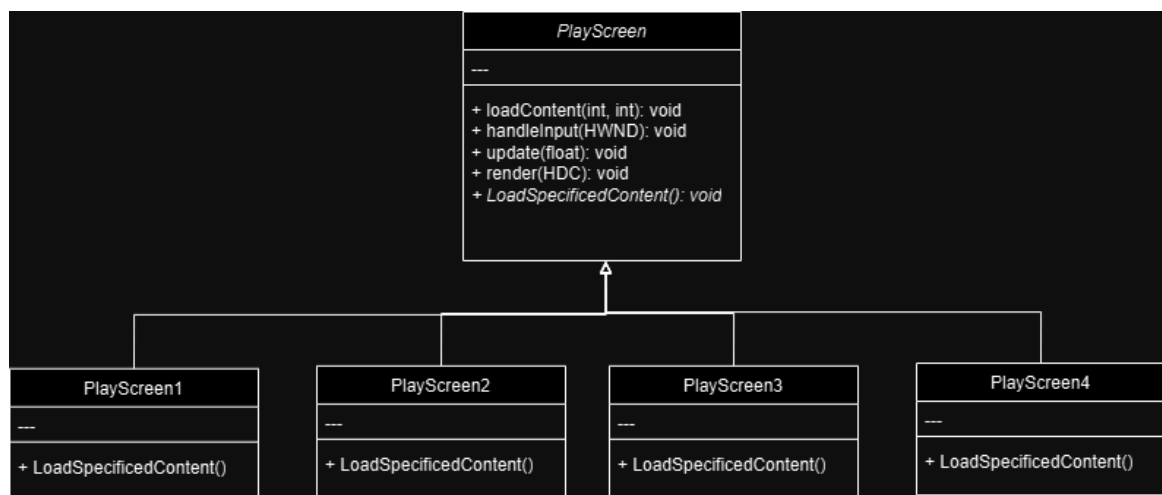
- Manage various game states, such as Main Menu, Gameplay, Pause, and Game Over. Each state is implemented as a separate class with standardized methods `loadContent()`, `handleInput()`, `update()`, and `render()` ensuring clear and modular behavior for each state.



Template Method Pattern

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

The PlayScreens exhibit a high degree of code similarity, making inheritance a suitable approach for code reuse. However, the need for map-specific customizations requires the ability to override inherited methods. This architecture leverages inheritance to promote code reuse while allowing for polymorphic behavior.



3.3.2 Purpose and Application in the System

Screen & Feature	Describe
Mainboard	<ul style="list-style-type: none">- Button:<ul style="list-style-type: none">+ Login+ Start game+ Continue+ Leaderboard+ Setting+ Exit
Register Screen	<p>Register the new player</p> <p>-> Username</p> <p>-> Password</p> <p>-> Password again</p>
Login Screen	<p>Login to the game</p> <p>-> Username</p> <p>-> Password</p> <p>-> Register</p>
Setting screen	<ul style="list-style-type: none">- Configure settings for volume, background music, and game speed (x1, x1.5, x2)
Leaderboard screen	<p>Showing the rank of all rounds based on player scores</p>
Game Setup Screen	<p>The player prepares the game by placing available towers in designated positions.</p> <p>Instructions Panel: (2 buttons to change placement)</p> <p>Placement Controls: (not on-screen buttons, use the 4 arrow keys for movement, space to confirm, and an Backspace button)</p>

	<p>to delete the tower)</p> <p>-> If placement is incomplete: save temporarily under the name 'guess.'</p> <p>START GAME: (ENTER)</p>
Game Screen	<ul style="list-style-type: none"> - Pause: Temporarily pause the game - Menu: Options to either continue playing or exit to the main screen. (Two buttons: Continue, Main Screen) - If logged in, the game auto-saves under the user's login name. (Always saves)
End Game Screen	<p>Win or lose</p> <p>Menu: Options to proceed to the next level, exit, or display the score</p>
Choose map screen	The player selects one of the four available levels using the arrow keys
Load game	The player selects one of the levels, saving the progress of the last turn played

3.4 Content game


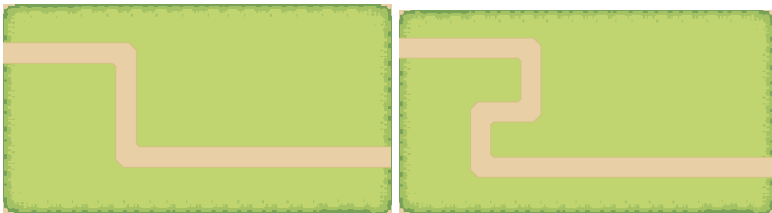
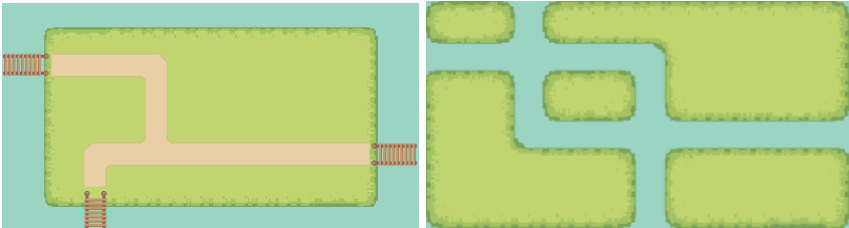



Level	Path	Towers	Enemies	Phases
1	Short, simple path with 2 turns, 6 tower positions	3 standard towers	<p>Total: 10 enemies</p> <ul style="list-style-type: none"> - 9 standard enemies (speed: 1, score: 10 points) - 1 boss (high HP, speed: 1.5, score: 20 points) 	<p>Phase 1: 9 standard enemies enter sequentially.</p> <p>Phase 2: (begins after Phase 1 enemies are defeated) Boss enters.</p>

2	Long path with 4-8 turns, 10 tower positions	4 standard towers	<p>Total: 15 enemies</p> <ul style="list-style-type: none"> - 14 standard enemies - 1 boss 	<p>Phase 1: 7 standard enemies enter.</p> <p>Phase 2: (begins when 3 enemies remain from Phase 1) 7 more standard enemies enter.</p> <p>Phase 3: (begins when all Phase 2 enemies are defeated) Boss enters.</p>
3	2 entry points, 1 exit, 3-7 turns, 1 intersection	4 standard towers, 1 special tower (if available)	<p>Total: 30 enemies</p> <ul style="list-style-type: none"> - 21-29 standard enemies (score: 10 points) - 0-8 special enemies (if available, score: 15 points) - 1 boss (score: 20 points) 	<p>Phase 1:</p> <ul style="list-style-type: none"> - 6 standard enemies enter from path 1. - 6 standard enemies enter from path 2. <p>Phase 2: (begins when 50% of enemies remain)</p> <ul style="list-style-type: none"> - 6-12 standard enemies enter from path 1. - 0-4 enemies enter from random paths. <p>Phase 3: (begins when all Phase 2 enemies are defeated)</p> <ul style="list-style-type: none"> - Boss enters from path 1. - Remaining enemies enter from path 2.
4	1 entry, 2 exits, 3-7 turns, 1 intersection	4 standard towers, 1 special tower (if available)	<p>Total: 30 enemies</p> <ul style="list-style-type: none"> - 21-29 standard enemies - 0-8 special enemies (if available) - 1 boss 	<p>Phase 1:</p> <ul style="list-style-type: none"> - 6 standard enemies enter and exit from path 1. - 6 standard enemies enter and exit from path 2. <p>Phase 2: (begins when 50% of enemies remain)</p> <ul style="list-style-type: none"> - 6-12 standard enemies enter and

				<p>exit from path 1.</p> <p>- 0-4 special enemies enter and exit randomly from 2 paths.</p> <p>Phase 3: (begins when all Phase 2 enemies are defeated)</p> <p>- Boss enters and exits from path 1.</p> <p>- Remaining enemies enter and exit from path 2.</p>
5	2 entries, 2 exits, 4 or more intersections, multiple turns (loop if possible)	5 towers	<p>Total: 30 enemies</p> <p>- 21-29 standard enemies</p> <p>- 0-8 special enemies (if available)</p> <p>- 2 bosses</p>	<p>Phase 1:</p> <p>- 12 standard enemies enter:</p> <p>- 3 from path 1 to exit 1</p> <p>- 3 from path 1 to exit 2</p> <p>- 3 from path 2 to exit 1</p> <p>- 3 from path 2 to exit 2</p> <p>Phase 2: (begins when 50% of enemies remain)</p> <p>- 6-12 standard enemies take path 2.</p> <p>- 0-4 special enemies take path 3.</p> <p>Phase 3: (begins when all Phase 2 enemies are defeated)</p> <p>- 1 boss takes path 2.</p> <p>- 2 bosses take path 3.</p> <p>- Remaining enemies take path 4.</p>

3.5 UX/UI design

Screen	Image
--------	-------

Main screen	
Map 1,2	
Map 3,4	
enemy	
Turret	
Bullet	

Menu	 <p>The image displays four distinct game menu screens arranged in a 2x2 grid. Each screen is framed by a brown, textured border with visible rivets at the corners, giving it a wooden chest appearance. The background of each screen is a light green field with a subtle pattern of small white flowers.</p> <ul style="list-style-type: none">Setting: The title "SETTING" is at the top. Below it is a speaker icon for sound settings, a green toggle switch, and a row of ten green rectangular bars of varying heights. To the right of the bars are two small triangular navigation buttons.PAUSE GAME: The title "PAUSE GAME" is at the top. Below it is a small character icon, the text "continue?", and two buttons: a checkmark and an "X".YOU WIN!: The title "YOU WIN!" is at the top. Below it is a small character icon and two buttons: a checkmark and an "X".YOU LOSE!: The title "YOU LOSE!" is at the top. Below it is a small character icon and two buttons: a checkmark and an "X".
------	---

Chapter 4: Implementation

4.1 Development Tool

4.1.1 IDEs

In the development of our Tower Defense game, we utilized **Visual Studio 2022** as our integrated development environment (IDE). Visual Studio is a powerful IDE that provides a comprehensive set of tools for C++ development, making it an ideal choice for our project.

Visual Studio 2022

- **User-Friendly Interface:** Visual Studio 2022 offers an intuitive and user-friendly interface that simplifies navigation and enhances productivity. Features such as IntelliSense provide code suggestions and automatic completion, helping developers write code more efficiently.
- **Built-in Debugging Tools:** The IDE comes equipped with advanced debugging tools that allow us to identify and fix issues quickly. Breakpoints, watch windows, and call stacks make it easy to track down bugs and optimize performance.
- **Project Management:** Visual Studio's project management capabilities enable us to organize our game files, resources, and libraries effectively. The solution explorer allows easy access to all project components, facilitating better collaboration among team members.
- **Integration with Version Control:** The IDE supports integration with version control systems like Git, which is crucial for our team to manage changes, track progress, and collaborate effectively.

4.1.2 Compilers

Visual Studio 2022 includes the **MSVC (Microsoft Visual C++)** compiler, which is known for its high performance and compliance with the C++ standard. Key benefits of using the MSVC compiler include:

- **Optimization:** The compiler provides various optimization options that help improve the performance of our game. We can fine-tune the compilation settings to achieve the best performance for our game's specific needs.
- **Compatibility:** The MSVC compiler ensures compatibility with a wide range of libraries and frameworks, allowing us to integrate third-party components seamlessly.

4.1.3 Environment and libraries setup

For our Tower Defense game, we utilized **Visual Studio 2022** as our IDE, along with essential C++ standard libraries. The **iostream** library facilitated input and output

operations, while **mutex** and **thread** enabled safe multi-threading for efficient game logic and rendering. We also employed **windows.h** to manage the game window and user input effectively.

a) Win32 API

Win32 API: a core set of functions provided by Microsoft Windows for interacting with the operating system at a low level. In game development, it offers a lightweight and flexible framework for managing windows, handling user input, and rendering graphics. Unlike modern game engines, the Win32 API requires developers to build most of the functionality from scratch, providing complete control over the game loop and rendering process. It is used to handle critical game functionalities, including:

- **Window Management:** Creating and managing the game window, responding to resize events, and defining the rendering area.
- **Event Handling:** Capturing user input through keyboard and mouse events, essential for placing towers, starting the game, and navigating menus.
- **Graphics Rendering:** Using device contexts (DCs) and GDI+ (Graphics Device Interface Plus) functions for rendering game elements directly to the screen. Buffered rendering techniques were employed to eliminate flickering and ensure smooth visuals.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

// Entry point
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // Register the window class
    // Create the window
    // Run the message loop
    MSG msg = {};
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

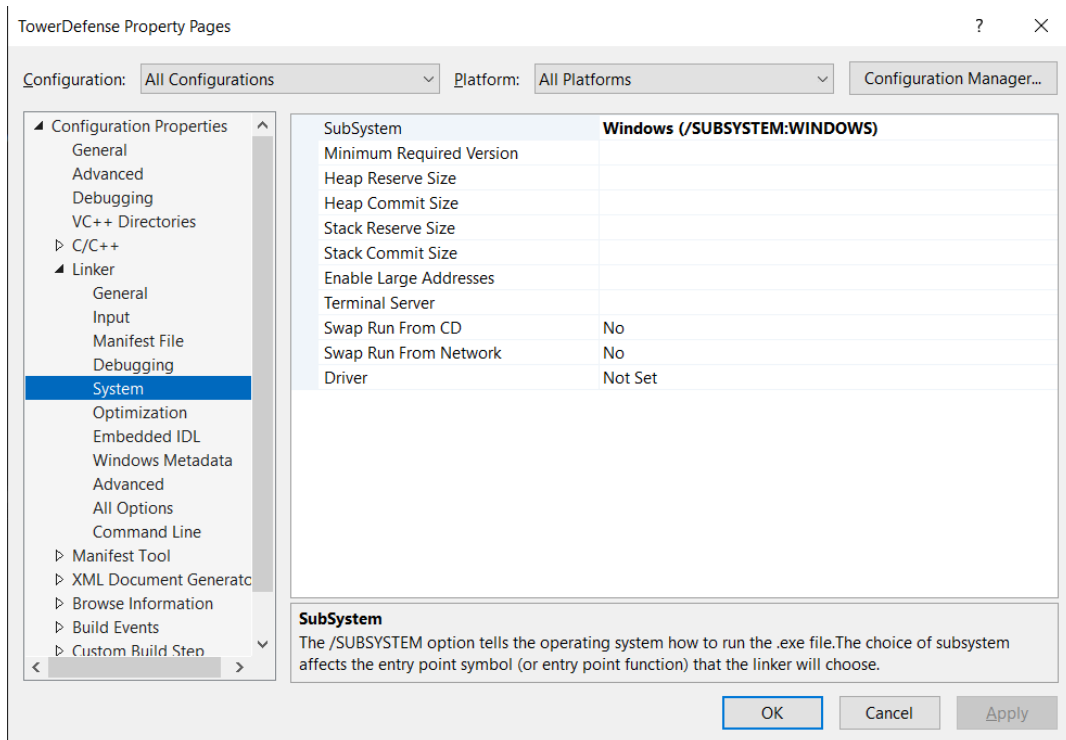
    return (int)msg.wParam;
}
```

- **Threading and Timing:**

While the Win32 API requires a deep understanding of low-level programming. By leveraging the API's capabilities, we built a custom game loop, implemented efficient rendering logic, and gained insight into how modern game engines operate under the hood.

b) Configuration window setup

- **Linker window**



To create a Windows console application, *right-click on the project file, select Configuration > Linker > System, and in the Subsystem field, change the selection to Windows*

- **Folder structure:**
 - Navigate to **Project > Properties > Configuration Properties > C/C++ > General**.
 - In the **Additional Include Directories** field, add **\$(ProjectDir)**.
This allows Visual Studio to locate all **.h** files from any subdirectory within the project.

For example, you can include header files using relative paths to their directories:

```
#include "Utils/Utils.h"
#include "Graphics/Renderer.h"
```

Alternatively, if you do not use this setting, you can specify a specific directory such as **\$(ProjectDir)Utils**. In this case, you can include files by their names directly as usual:

```
#include "utils.h"
```

Note: The order of the commands in the include paths affects the program's behavior. If you place **\$(ProjectDir)** at the beginning, the program will prioritize searching and resolving headers in this directory first. For consistency, files should be included with clear, relative paths.

- **Window Size**

Set the console window to match the standard screen size of 1280x720, an ideal dimension for game display.

- **Font Size**

We are developing a custom font system where each letter is represented by a separate .bmp image stored in the 'Assets/text' directory. The program monitors keyboard input for characters A-Z and displays the corresponding image on the screen.

c) **GDI+**

Using GDI+ in Game Development: an enhanced graphics library in the Windows operating system, to handle rendering and graphical operations. GDI+ provides more advanced features than the basic GDI, including support for anti-aliased 2D graphics, transparency, and higher-level graphics primitives, making it suitable for creating visually appealing games.

Key advantages of using GDI+ in our game development include:

- **Enhanced Graphics Quality:** GDI+ allows for smooth rendering of shapes, text, and images with anti-aliasing, giving our game a polished appearance.
- **Support for Transparency:** With GDI+, we incorporated semi-transparent effects for game elements like bullets and explosions, adding visual depth to the gameplay.
- **Simplified Drawing Functions:** Compared to GDI, GDI+ provides higher-level APIs for drawing shapes and managing colors, making it easier to implement features like health bars, UI elements, and backgrounds.
- **Custom Fonts and Text:** GDI+ enabled us to use a variety of fonts and styles for menus, scoreboards, and in-game text, improving user experience and readability.
- **Image Handling:** GDI+ supports a wide range of image formats and allows scaling, rotation, and blending, which we used for dynamic game elements such as moving enemies and animated towers.

```

void Graphic::DrawBitmap(HBITMAP hBitmap, POINT start, HDC hdc) {
    // check
    // draw
    BITMAP bitmap;
    GetObject(hBitmap, sizeof(bitmap), &bitmap);

    // create paint
    BLENDFUNCTION blendFunc

    BOOL success = AlphaBlend(
        hdc, start.x, start.y, bitmap.bmWidth, bitmap.bmHeight,
        hdcMem, 0, 0, bitmap.bmWidth, bitmap.bmHeight, blendFunc
    );

    //check
    //delete
    SelectObject(hdcMem, hbmOld);
    DeleteDC(hdcMem);
}

```

4.2 Key Features

4.2.1 Load and Continue Game

The "Load and Continue Game" feature allows players to resume their progress from a previously saved game state. The system checks for the existence of a save file, reads the data, and initializes the game accordingly. If no save file is found, an error message is displayed, and the user is redirected to the main menu.

Key functionalities:

- Check if the save file exists.
- Parse and load game data into the current state.
- Start the game or return to the menu if the load fails.

Pseudocode:

function loadGame(saveFilePath):

 if fileExists(saveFilePath):

 gameData = readFile(saveFilePath)

 initializeGameState(gameData)

 return true

else:

 displayError("Save file not found.")

```
return false
```

```
function continueGame():
```

```
    if loadGame("savefile.sav"):
```

```
        startGame()
```

```
    else:
```

```
        showMainMenu()
```

```
std::string filename;
if (mapCode == 1) {
    filename = "SaveMap1.catfam";
}
else if (mapCode == 2) {
    filename = "SaveMap2.catfam";
}
else if (mapCode == 3) {
    filename = "SaveMap3.catfam";
}
else if (mapCode == 4) {
    filename = "SaveMap4.catfam";
}
filename = "Storage/" + filename;

supportWriteFile(filename); // for loadgame
supportWriteFile("Storage/AllSaveGame.catfam"); // for leaderboard
```

4.2.2 Leaderboard System

The Leaderboard System maintains a record of the top 5 player scores. When a new score is added, the system sorts all entries in descending order and ensures that only the top 5 scores are retained. The leaderboard can be displayed in a ranked list.

Key functionalities:

- Add new scores dynamically.
- Sort leaderboard entries in descending order of scores.

- Limit the leaderboard to the top 5 players.

```
// doc file
if (!readFile())
{
    userElements.push_back(make_shared<TextElement>(L"There are no user", customFont, RGB(255, 255, 255), startPos));
    return;
}

sortUserByScore(); // sort
int numberUser = min(users.size() - 1, 4);
//
for (size_t i = 0; i <= numberUser; ++i)
{
    string str = to_string(i + 1) + ". " + users[i].getName() + " " + to_string(users[i].getScore());

    wstring userInfo = Utils::stringToWstring(str);

    POINT pos = { startPos.x, startPos.y + static_cast<LONG>(i * yOffset) };
    userElements.push_back(make_shared<TextElement>(userInfo, customFont, RGB(255, 255, 255), pos));
}
```

4.2.3 Audio Settings

The Audio Settings feature enables players to adjust the game's audio preferences, such as volume and mute options. It provides a simple interface where users can set the volume level or toggle mute.

Key functionalities:

- Toggle mute on/off.
- Display an audio settings menu for user interaction.

```
void AudioManager::playBackgroundMusic()
{
    if (!backgroundMusic.empty())
    {
        PlaySound(backgroundMusic.c_str(), NULL, SND_FILENAME | SND_ASYNC | SND_LOOP);
    }
}
```

- Adjust master volume with a valid range (0–100).

```
DWORD volume = static_cast<DWORD>(musicVolume * 0xFFFF);
waveOutSetVolume(0, MAKELONG(volume, volume));
```

4.2.4 Tower Placement Mechanics

The Tower Placement Mechanics feature allows players to strategically place towers on the game map. The system validates the placement area and ensures sufficient resources (gold) are available before allowing the placement. Invalid actions, such as placing a tower outside the designated area or insufficient gold, trigger error messages.

Key functionalities:

- Validate tower placement areas.
- Deduct the correct cost from the player's resources.
- Create a tower object at the specified position if conditions are met.

Pseudocode:

```
function placeTower(towerType, position):  
    if isPositionValid(position) and isEnoughGold(towerType.cost):  
        createTower(towerType, position)  
        reduceGold(towerType.cost)  
        return true  
    else:  
        displayError("Invalid position or insufficient gold.")  
        return false
```

```
function handleClick(x, y):  
    if isWithinPlacementArea(x, y):  
        selectedTowerType = getSelectedTowerType()  
        placeTower(selectedTowerType, {x, y})  
    else:  
        displayError("Cannot place tower here.")
```

4.3 Programming Techniques

4.3.1 Callback Mechanism

In the Win32 API, the callback mechanism is an essential design pattern that enables applications to process user interactions such as keyboard and mouse inputs. The callback function, often referred to as the window procedure (**WndProc**), is automatically invoked by the system whenever an event related to the application window occurs.

Key Roles of the Callback Mechanism:

- **Handling Window Events:** The **WndProc** function processes events like window creation, resizing, minimizing, or closing. Each event corresponds to a specific message (e.g., **WM_CREATE**, **WM_SIZE**, **WM_CLOSE**), which is sent to the callback function.
- **Keyboard Input:** The system sends keyboard events to the **WndProc** function via messages like **WM_KEYDOWN**, **WM_KEYUP**, or **WM_CHAR**. This allows the

application to detect key presses and releases. For example, in our game, these events are used to move the cursor, place towers, or trigger actions like starting or pausing the game.

- **Mouse Input:** Mouse interactions, such as clicks and movements, are captured using messages like `WM_LBUTTONDOWN`, `WM_RBUTTONDOWN`, or `WM_MOUSEMOVE`. These events enable the game to detect tower placements, interact with menus, or manage drag-and-drop operations.

4.3.2 Memory Management with Pointers

In this section, we explain the use of pointers and memory management techniques in the game development process. Since the game involves dynamic resource allocation (such as creating towers, enemies, bullets, etc.), proper memory management is essential to avoid memory leaks and ensure the program runs efficiently. We use C++ smart pointers provided by the standard library (`<memory>`) to manage memory effectively.

We use **smart pointers** (`std::unique_ptr` and `std::shared_ptr`) in the game to handle dynamic objects. These pointers are designed to automatically manage memory, ensuring that resources are released when no longer in use.

- **`std::unique_ptr`:**
This is used when a single ownership of an object is required. When the object goes out of scope, the memory is automatically deallocated. This is ideal for objects that are owned by a single entity, such as a tower or an enemy, which doesn't need to be shared between multiple components.
- **`std::shared_ptr`:**
- This pointer is used when an object needs to be shared among multiple components. It ensures that the object remains alive as long as there is at least one `std::shared_ptr` pointing to it. Once all the references to the object are destroyed, the memory is automatically freed.

Declare: `std::shared_ptr<Screen> newscreen;`

Use: `newscreen = std::make_shared<MainScreen>();`

4.3.3 Threading Techniques

- **Main Game Loop**

The main game loop operates as follows:

- Process Windows messages.
- Update game state and animations.
- Render the current frame to the screen.

Pseudo-code:

```

while (gameRunning) do
    if (messageAvailable) then
        handleWindowsMessages()
    end if

    deltaTime ← calculateDeltaTime()

    screenManager.update(deltaTime)

    screenManager.render()
end while

```

- Input Handling on a Separate Thread

Input handling is moved to a dedicated thread to ensure responsiveness, even during heavy rendering operations. This thread continuously polls input states and forwards them to the **ScreenManager**.

Pseudo-code:

```

startThread(
    while (gameRunning) do
        screenManager.handleInput()

        sleep(16ms) // Polling interval
    end while
)

```

- Audio Management with Independent Threads

Audio playback for background music and sound effects operates on separate threads. This prevents audio operations from interfering with game updates or rendering.

- **Background Music Thread:** Loops the background track independently of other game processes.

Pseudo-code:

```

startThread(
    while (musicRunning) do

```

```

        playBackgroundMusic()
    end while
)

```

- **Sound Effects Thread:** Handles sound effect playback on demand without disrupting the main game loop or background music.

Pseudo-code:

```

startThread(
    if (soundEffectTriggered) then
        playSoundEffect()
    end if
)

```

- **Thread Synchronization**

To avoid race conditions, synchronization mechanisms like mutexes are employed when accessing shared resources such as the input state or audio buffers. For example, when updating shared input data:

Pseudo-code:

```

lock(inputMutex)
updateInputState()
unlock(inputMutex)

```

4.3.4 Dependency Injection

By exclusively using *ResourceManager* as a dependency:

- The *Game* class is relieved of the responsibility of directly managing resources, allowing it to focus solely on core game logic.
- You can effortlessly modify or extend the resource management approach (e.g., instead of *ResourceManager*, employ a server-based resource loading system).
- Testing and maintenance become more straightforward as *ResourceManager* can be mocked in unit tests.

4.4 Algorithms

4.4.1 Projectile Trajectory Calculation

Algorithm for calculating bullet trajectories based on tower and enemy positions.

```
float cpoint::distance(const cpoint& a) {  
    return sqrt((x - a.x) * (x - a.x) + (y - a.y) * (y - a.y));  
}
```

Update bullet trajectories: It first calculates the direction vector from the projectile's current position to the target's position, then computes the length of that vector. The direction is normalized to ensure the projectile moves at a constant speed based on its speed attribute. Finally, the current position of the projectile is updated by adding the normalized direction vector, causing it to move closer to the target. This approach ensures smooth, consistent movement of the projectile toward its target.

```
cpoint e = target->getCurrentPosition();  
  
cpoint direction = e - currentPosition;  
  
float length = std::sqrt(static_cast<float>(direction.getX() * direction.getX() +  
direction.getY() * direction.getY()));  
  
cpoint normalizedDirection = direction.normalized(length, model->getSpeed());  
  
this->currentPosition = this->currentPosition + normalizedDirection;
```

4.4.2 Logic for Enemies

a) Movement logic

Enemy movement follows a path based on map data, which contains a list of coordinates representing key points (such as turns). The game reads two points at a time, and the enemy moves from the first point to the next along this path until it reaches the end. This system allows enemies to follow complex routes with turns, simulating a natural movement pattern toward the objective.

b) Selection Logic (for Tower Targeting):

Each tower has a defined attack radius. The tower scans for enemies within this radius and compiles a list of all those detected. Among these, the tower identifies the nearest enemy by calculating the distance from each enemy's position to the tower's position. The enemy with the shortest distance is selected as the target.

4.4.3 Bullet Logic:

a) Movement logic

When a bullet is fired, it moves toward the coordinates of the selected enemy target. This movement is done step-by-step by adjusting the bullet's X and Y coordinates (X_b , Y_b) to match the enemy's coordinates (X_e , Y_e):

- If $X_b < X_e$, increment X_b by 1.
- If $X_b > X_e$, decrement X_b by 1.
- If $Y_b < Y_e$, increment Y_b by 1.
- If $Y_b > Y_e$, decrement Y_b by 1.

b) Collision

The process in (a) continues until the bullet's coordinates reach those of the enemy target ($X_b, Y_b == X_e, Y_e$), at which point a collision is detected, and the game registers the hit.

```
checkCollision() {  
    if (currentPosition.distance(target->getCurrentPosition()) < model->getSpeed()) {  
        return target;  
    }  
}
```

4.4.4 Sorting Logic

- Sorting algorithms used for leaderboard rankings or other in-game functionalities.

4.5 Graphics and Rendering

4.5.1 Buffered Rendering

In the Tower Defense game, we implemented a buffered rendering technique to optimize graphics and ensure smooth visuals. The main idea of this approach is to use an off-screen buffer for drawing all game elements before copying the final output to the screen. This prevents flickering and provides a seamless visual experience.

- **Setup the Buffer:** A device context (DC) compatible with the screen's DC is created using `CreateCompatibleDC`. Additionally, a bitmap is created as a drawing surface for the buffer, matching the size of the game window retrieved via `GetClientRect`. This bitmap is then selected into the buffer DC, making it the target for all subsequent drawing operations.

- **Clear the Buffer:** Before rendering the game elements, the buffer is cleared using a solid brush to fill it with a background color, ensuring no residual graphics from previous frames. In this case, a black background (`RGB(0, 0, 0)`) is applied.

- **Render to the Buffer:** Game elements are rendered to the buffer by calling `render(bufferDC)` from the `screenManager` object. This method handles all drawing operations, such as towers, enemies, bullets, and user interface elements, onto the buffer DC.

- **Copy to Screen:** Once rendering is complete, the contents of the buffer are copied to the screen using `BitBlt`. This ensures that only fully drawn frames are displayed, preventing visible partial updates.

- **Resource Management:** After rendering, all resources such as the bitmap, brush, and buffer DC are properly released to avoid memory leaks. This includes restoring the original bitmap into the buffer DC and deleting any temporary objects.

By employing buffered rendering, the game maintains consistent performance and visual quality, even during complex scenes with multiple moving elements. This approach is crucial for creating a polished gaming experience.

Code example:

```
// Tạo bộ đệm
GetClientRect(windowHandle, &clientRect);

HBITMAP bufferBitmap = CreateCompatibleBitmap(hdc, width, height);
HBITMAP oldBitmap = (HBITMAP)SelectObject(bufferDC, bufferBitmap);

// Xóa bộ đệm trước khi vẽ
HBRUSH brush = CreateSolidBrush(RGB(0, 0, 0)); // Màu nền (đen)
FillRect(bufferDC, &clientRect, brush);
DeleteObject(brush);

// Vẽ vào bộ đệm
screenManager->render(bufferDC); // Vẽ màn hình vào DC bộ đệm

// Copy nội dung từ bộ đệm ra màn hình
```



```
BitBlt(hdc, 0, 0, width, height, bufferDC, 0, 0, SRCCOPY);  
  
// Giải phóng tài nguyên
```

4.5.2 Drawing Algorithm

a) Logic:

Single Object Example: Here's a basic drawing algorithm example for an object displaying the letter "A":

- Clear "A" at its previous position.
- Update "A" to a new position based on predefined logic.
- Draw "A" at the new position.
- Allow a short delay for display.
- Repeat the cycle.

This drawing process consists of four key steps:

- Clear the previous position
- Update to the new position
- Draw at the new position
- Delay to display the image
- Check the repeat condition

b) Animation:

The animation system is designed to handle multiple frames of an image sequence that are displayed consecutively to simulate motion. The core techniques include:

- Frame-based Rendering: Each animation consists of a series of images (frames) that are switched based on a timer.
- Timer-based Control: A timer mechanism ensures smooth frame transitions over a fixed interval.
- Dynamic Loading: Frames are dynamically loaded using `Graphic::LoadBitmapImage` to optimize memory usage.
- State Management: The animation can be started, paused, or stopped to fit gameplay needs.

This is an example of sequence frames:



Code example:

```
animationTimer += deltaTime;

    if (animationTimer >= animationSpeed)
    {
        animationTimer = 0.0f;
        currentFrame = (currentFrame + 1) % frames.size();
    }
```

4.5.3 GDI+ Functions

- Details on the specific GDI+ functions used for drawing shapes, text, and images.

Chapter 5: Conclusion

5.1 Project Outcome

5.1.1 Summary of Success

The Tower Defense game meets the expectations outlined at the project's inception, delivering a functional and interactive gaming experience. The game can be controlled using both keyboard and mouse, ensuring accessibility for a variety of users. All core features were successfully implemented, including level progression, collision effects, sound controls, and save/load functionality. The game also features at least four distinct levels, each with increasing difficulty, providing players with a challenging yet enjoyable experience.

5.1.2 Features Achieved

- **Control Options:** Fully functional keyboard and mouse controls for seamless interaction.
- **Basic Functionalities:** Complete gameplay loop, including tower placement, enemy attacks, and level transitions.
- **Collision Effects:** Visual and sound effects when towers attack enemies, enhancing immersion.
- **Save/Load System:** Players can save their progress and resume from the saved state at any time.
- **Level Diversity:** A minimum of four levels with distinct layouts and challenges.
- **Sound Controls:** On/off toggle for background music, adjustable volume, and dynamic sound effects during gameplay.
- **Smooth user interaction** through keyboard-controlled tower placement.
- **Real-time rendering** with buffered graphics using GDI+.
- **Functional game mechanics:** tower attacks, enemy waves, and level progression.
- **A polished user interface** with menus for level selection, game settings, and a pause menu.
- **High score tracking** and save functionality to enhance replayability.
- **Login System:** Players can log in to personalize their experience and track their progress, ensuring saved data is linked to their account.

5.2 Future Improvements

5.2.1 Suggestions for Enhancement

- **Advanced AI for Enemies:** Introduce more intelligent enemy behaviors to increase difficulty and unpredictability.

- **Visual Upgrades:** Enhance visual effects and animations, especially for collisions and explosions, to improve the game's appeal.
- **Improved Level Design:** Add more diverse obstacles and strategic elements to levels for deeper gameplay.
- **Custom Settings:** Allow players to customize control mappings and interface preferences for a more personalized experience.

5.2.2 Additional Features for Future Development

- **Dynamic Music System:** Integrate a dynamic music system where the soundtrack changes based on gameplay intensity.
- **Tower Upgrade System:** Add an in-game shop or upgrade system to customize and enhance towers during play.
- **Achievements:** Implement an achievement system to reward players for completing specific challenges.
- **Online Leaderboards:** Introduce global leaderboards to compare scores and progress with other players.
- **Multilingual Support:** Provide support for multiple languages to make the game accessible to a wider audience.

Working against a tight deadline, our team focused on implementing the core features of the product. Through this process, we acquired a solid understanding of the Windows.h API. While there are areas for improvement, the project has equipped us with the foundational skills necessary to expand and enhance our game, serving as a springboard for our careers in game and application development.

REFERENCE

1. Microsoft. Your first Windows program. *Microsoft Learn*. Retrieved November 15, 2024, from <https://learn.microsoft.com/vi-vn/windows/win32/learnwin32/your-first-windows-program>
2. Game Programming Patterns. *Game Programming Patterns*. Retrieved November 15, 2024.
3. Petzold, C. *Programming Windows* (5th ed.). Microsoft Press, 2012.
4. Oualline, S. *Practical C++ Programming*. O'Reilly Media, 2003.
5. Blagy, D. (n.d.). *dewcin_yt* [GitHub repository]. GitHub. Retrieved November 15, 2024, from https://github.com/danielblagy/dewcin_yt
6. Refactoring Guru. (n.d.). *Refactoring and Design Patterns*. Retrieved November 15, 2024, from <https://refactoring.guru/>