AOOP Report - The Snail Brawl

1st Yin-Wei Lee
Department of Photonics National Yang Ming Chiao Tung University Hsinchu, Taiwan waynelee.ee11@nycu.edu.tw

2nd Bo-Wei Wang
Department of Photonics National Yang Ming Chiao Tung University Hsinchu, Taiwan kaiwy58.ee11@nycu.edu.tw

***Abstract*—This paper introduces a tower defense game developed using Python and Pygame, named "The Snail Brawl." The game combines strategy and fun, where players deploy characters to defend against enemy attacks. This paper details the game's design philosophy, core mechanisms, and implementation process, and discusses its applications and challenges in game development.**

***Index Terms*—tower defense game, Python, Pygame, OOP, The Snail Brawl**

I. Introduction

"The Snail Brawl" is a single-player tower defense game developed based on Pygame, adapted from the original Battle Cats game but themed around snails. The motivation for this research stems from the practical application of Object-Oriented Programming (OOP) principles, aiming to explore modular design, class encapsulation, and game logic control through game development. The background issue includes the monotony of traditional tower defense games; we introduce new elements such as a reward system, level selection, and gacha mechanics to enhance the player experience. The main contributions include implementing a complete game loop, progress saving mechanisms, and applying OOP principles to improve code maintainability. GitHub link: https://github.com/unknown899/aoop_2025_group7_TBC.

II. Related Work

Related works include the original Battle Cats game by PONOS Corporation, which features cat-themed tower defense mechanics where players deploy units to resist enemy advances. Our game borrows its core gameplay, such as unit deployment, attack ranges, and tower health mechanisms, but adapts it to a snail theme and adds wallet upgrades, snail cannons, and left-right scrolling features. Additionally, we reference the level selection method from Super Mario, transforming traditional key-based selection into map hovering interactions to improve user interface friendliness. Other tower defense games like Plants vs. Zombies also provide inspiration in attack effects and unit diversity.

III. Proposed Method

The game design employs a modular structure with core classes including Cat (player units), Enemy (opponent units), Tower, Level, and YManager (coordinate manager). YManager embodies the single responsibility principle, using sqrt(x) to allocate y-coordinates to avoid unit overlaps. OOP principles applied: Encapsulation binds properties like hp and atk within classes; Inheritance and Polymorphism for unit movement and knockback (kb) methods; Abstraction integrates similar behaviors.

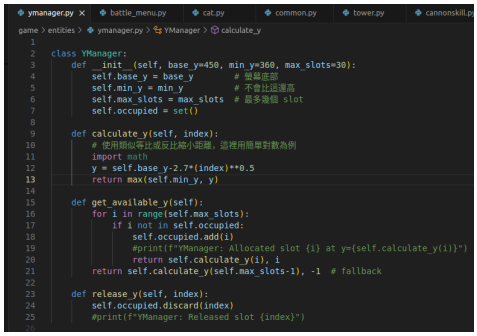

Fig. 1. Ymanager class, adjust character y-axis

The game logic is controlled by battle_logic.py, handling attack calculations and boss shock waves. config.py stores unit values such as health, attack power, and range. rewards.py manages probabilistic drops, including gold, souls, and unit unlocks. Progress saving uses JSON files like completed_levels.json and player_unlocked_cat.json.

Attack effects display based on the type, such as electric (lightning), gas (poison), and shockwave. The main loop uses asyncio for non-blocking execution, controlling FPS.

In addition, the overall game flow is implemented as a finite state machine (FSM) using an if–else control structure. The FSM consists of 10 states (excluding the quit state that terminates the program). Each state represents a distinct phase of gameplay, and state transitions are triggered by player actions or internal game events. This design simplifies logic control while maintaining clarity and extensibility.

The state identifiers and their corresponding meanings are defined as follows: A $\rightarrow$ Intro, B $\rightarrow$ Cat Selection, C $\rightarrow$ Playing, D $\rightarrow$ Main Menu, E $\rightarrow$ Paused, F $\rightarrow$ End, G $\rightarrow$ Level Map, H $\rightarrow$ Ending, I $\rightarrow$ Gacha, and J $\rightarrow$ Recharge. Transitions between these states are handled explicitly through conditional branching, ensuring deterministic behavior and predictable game flow.

Figure 4 illustrates the state transition diagram of the FSM, where nodes labeled A–J correspond to the aforementioned game states. The diagram provides a high-level overview of the

Identify applicable funding agency here. If none, delete this.

control logic and highlights how different gameplay modules are interconnected through state transitions.

In addition to the finite state machine, the system architecture is further illustrated using a simplified class diagram, as shown in Figure 5. The diagram captures the main classes, their attributes, methods, and relationships, providing a high-level overview of the game structure.

main.py is the main program. It calls game_loop.py within the Game module and repeatedly executes the while loop in game_loop.py (which includes 10 state transitions) when the program starts. The Game module utilizes various functions from the UI module, primarily for rendering the interface.

Game also employs class1 (which contains level) to manage and utilize class2 (which includes cat, enemy, and tower). The data for both class1 and class2 are provided by loaders. Since the structures of cat and enemy are similar, a Python abstract class common was first implemented, and cat and enemy inherit from common to simplify programming.

class3 mainly contains various visual effects and is used by class2.

Overall, the class diagram emphasizes encapsulation, inheritance, and modularity, reflecting the design choices made for maintainable and extensible game logic.

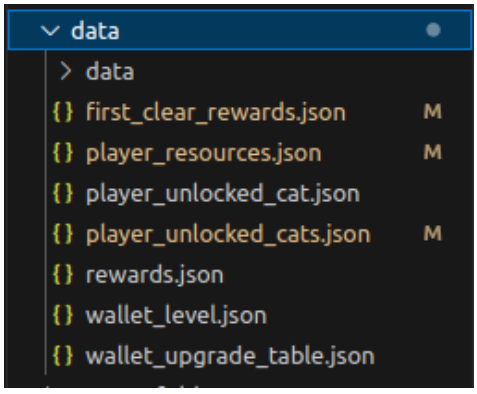

Fig. 2. data where were saved(in new version)

Fig. 3. data where were saved(in old version)

IV. Experimental Results

A. Opening Screen

After executing `main.py`, the game starts with an opening screen. The player may either click the *Skip* button or watch the full introductory animation, which presents the game background and theme.

B. Main Menu

The main menu allows players to navigate the core game systems via mouse interaction. The available options include

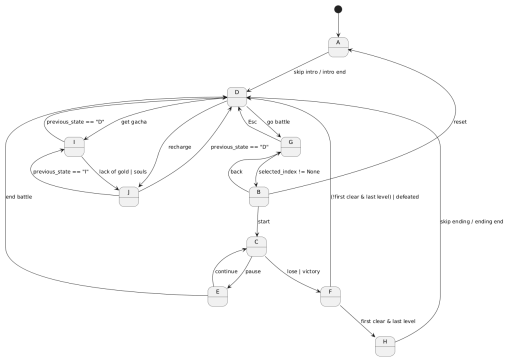

Fig. 4. Finite State Machine of the game. State A represents Intro; B represents Cat Selection; C represents Playing; D represents Main Menu; E represents Paused; F represents End; G represents Level Map; H represents Ending; I represents Gacha; and J represents Recharge.

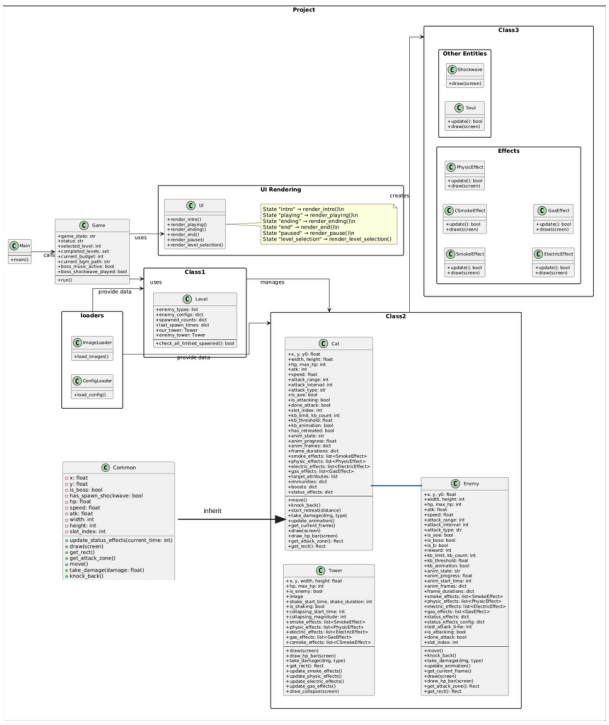

Fig. 5. Simplified class diagram of the game system. Core classes include Game, Level, Cat, Enemy, Tower, UI, loaders, effects, and other entities. The diagram highlights relationships, inheritance, and modular responsibilities within the game architecture.

Fig. 6. Opening screen of the game

Go to Battle, *Get Gacha* (character summoning system), and *Recharge* (in-game payment system).

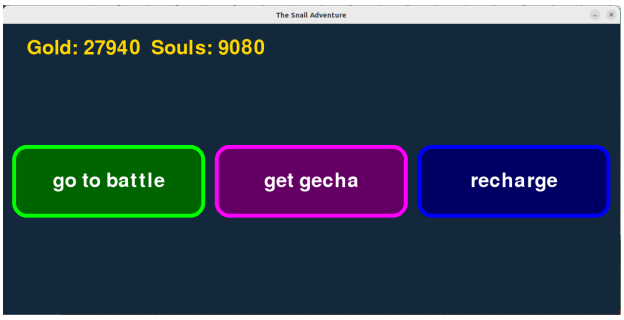

Fig. 7. Main menu interface

C. Level Map

After selecting *Go to Battle*, the player is redirected to the level map interface. Each node on the map represents a playable level. Gray nodes indicate locked levels, light green nodes indicate completed levels, and dark green nodes indicate unlocked but incomplete levels.

To ensure proper difficulty progression, levels beyond Level 1 are initially locked. Players must complete earlier levels to unlock subsequent ones. By hovering the cursor or the green translucent indicator over a node, the corresponding reward information is displayed on the right panel. The player may use the `WASD` keys or arrow keys to navigate between levels, and press `Enter` to proceed to the character selection screen.

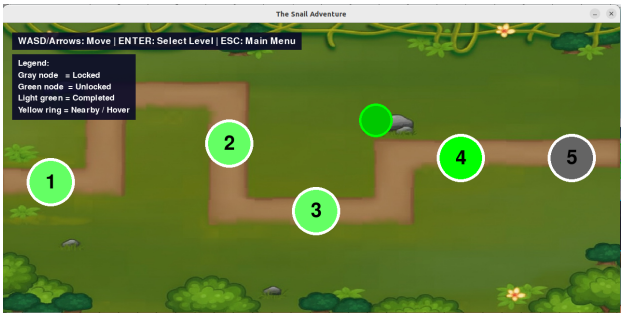

Fig. 8. Level map interface

D. Character Selection

After selecting a level, the player enters the character (cat) selection screen. The left panel displays the level status, while the upper-right grid allows players to preview character images by hovering the mouse. Clicking a character deploys it into the formation, which is shown in the bottom-right deployment area.

E. Battle System

Upon pressing *Start Battle*, the player is taken to the battle interface. The battlefield can be scrolled horizontally using the left and right arrow keys. The lower panel displays available characters, which can be deployed either by pressing the corresponding numeric keys or by clicking the icons.

Additional features include a wallet upgrade system (bottom-left), which increases income capacity at the cost of in-game currency, and a cat cannon (bottom-right), which becomes usable when the status displays “Ready!”. The game can be paused at any time using the pause key.

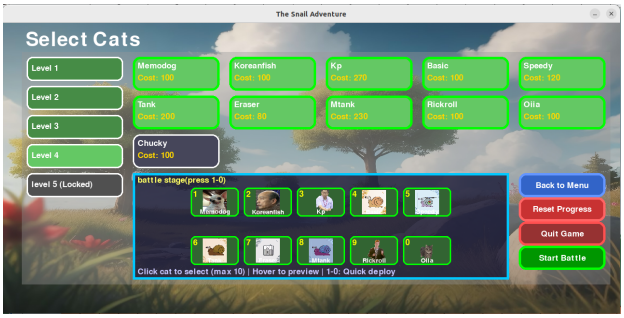

Fig. 9. Character selection interface

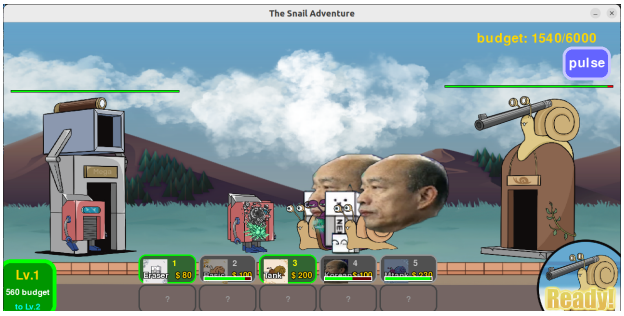

Fig. 10. Battle interface

F. Win and Loss Conditions

If the enemy base’s health reaches zero, the player wins the battle. A reward summary is displayed at the bottom of the screen, and the player may press `Enter` to confirm. Conversely, if the player’s base is destroyed, the battle is lost, and the game proceeds similarly after confirmation.

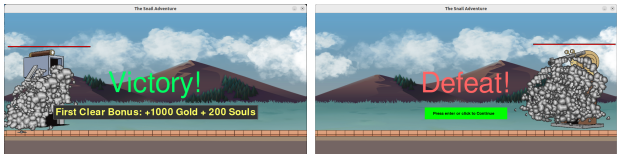

Fig. 11. Victory (left) and defeat (right) screens

G. Gacha System

The gacha system allows players to obtain characters through random draws. By clicking the *Roll Single* button, one draw is performed at the cost of a specified amount of Gold and Souls. If the player lacks sufficient resources, they are automatically redirected to the recharge interface.

Possible outcomes include duplicate rewards, newly obtained characters, or no reward.

Fig. 12. Gacha interface

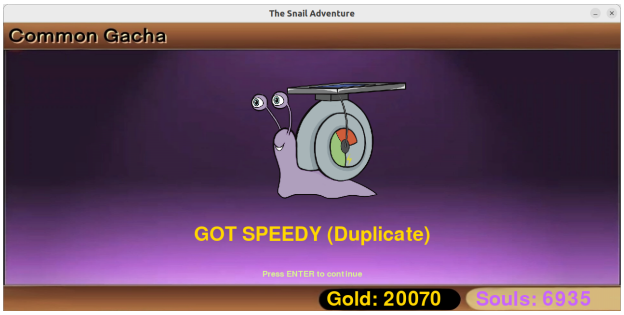

Fig. 13. Gacha result: duplicate reward

Fig. 14. Gacha result: new character obtained

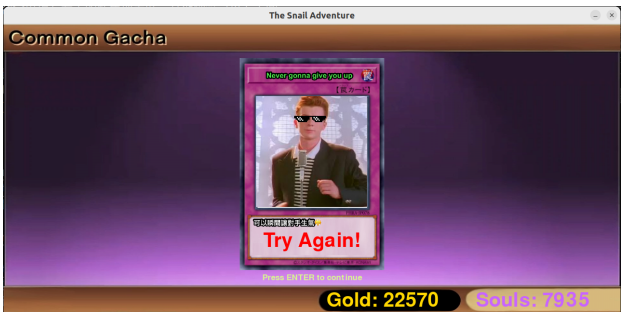

Fig. 15. Gacha result: no reward

H. Recharge System

The recharge system enables players to purchase in-game currency. Players select a payment plan from the left panel and enter a credit card number to complete the transaction.

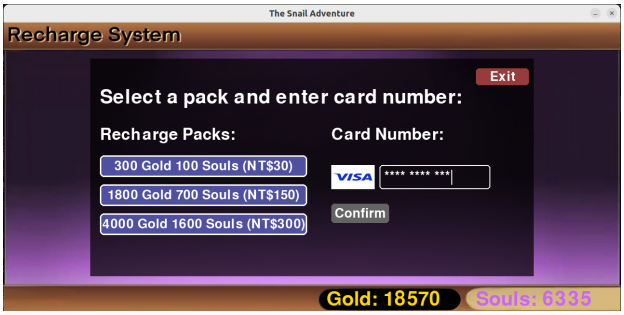

Fig. 16. Recharge interface

TABLE I
EXAMPLE OF PLAYER UNIT PROPERTIES

Unit	Health	Attack	Range
Basic Snail	Medium	Medium	Medium
Speedy Snail	Low	Medium	Small
Tank Snail	High	High	Large

V. CONCLUSION

This game successfully implements tower defense core mechanisms and applies OOP to enhance code quality. Future prospects include perfecting unit effects (e.g., attack reduction, slowing), adding fluctuation towers and other snail cannons, item systems, and infinite stage extension. Through this project, we demonstrate that Python and Pygame are suitable for small-scale game development, with potential future expansions to multiplayer modes or AI enemies.

ACKNOWLEDGMENT

Thanks to the Department of Electronic at National Yang Ming Chiao Tung University for providing resource support, and to team members for their collaboration.

REFERENCES

[1] PONOS Corporation, The Battle Cats," Mobile Game, 2008.
[2] Pygame Community, Pygame Documentation," https://www.pygame.org/docs/, 2023.
[3] IEEE, "How to Create Your Conference Paper," IEEE Conference Templates, 2023.