Informatics 122 Final Project

Team Members: Calvin Phan, Ernest Yao, Solvin Shrestha, Robin Stoebe, Wilson Zheng

Date: March 19, 2025

Table of Contents

Table of Contents **1**
- Github Repository: Link 2
- Instructions for running each game in TGME 2
- Game Rules 2
  - Suika Game 2
  - Tetris 2
- UML Diagram 4
  - UML Descriptions 4
    - GameEngine 4
    - BaseBoard 5
    - MainMenu 5
    - GameSelectionScreen 6
    - HighScoresScreen 6
    - LoginScreen 6
    - Button 7
    - Game 7
    - Player 8
    - Board 8
    - Tiles 9
    - Physical tiles 10
- How to create a game with the TMGE 10
- Reflection 11

Github Repository: Link

Instructions for running each game in TGME

1. Run game_engine.py located in src/engines
2. After file loads, press start game
3. Then, select player amount (1 or 2 player)
4. Finally, select either Tetris or Suika Game

Game Rules

Suika Game

Objective: Create a watermelon and get a high score

Game Loop:

- Drop fruits into a container by moving the top fruit left and right using the arrow keys, and dropping with a mouse input.
- Matching two fruits will create a new, third fruit
- As fruits are merged, the score increases
- Two non-matching fruits will not combine
- A player cannot drop a fruit outside of the container
- A player cannot manually move a fruit once its been placed

Loss Condition: Fruits exceed top of container, **Game Over**

Two Player Mode:

- Both players play simultaneously
- The Player with the highest score will win once both players meet the loss condition

Tetris

Objective: Survive a long duration and get a high score

Game Loop:

- Move the current Tetrimino down over time.
- Players can input various commands
  - Move left/right, Rotate piece, Hard drop/soft drop.

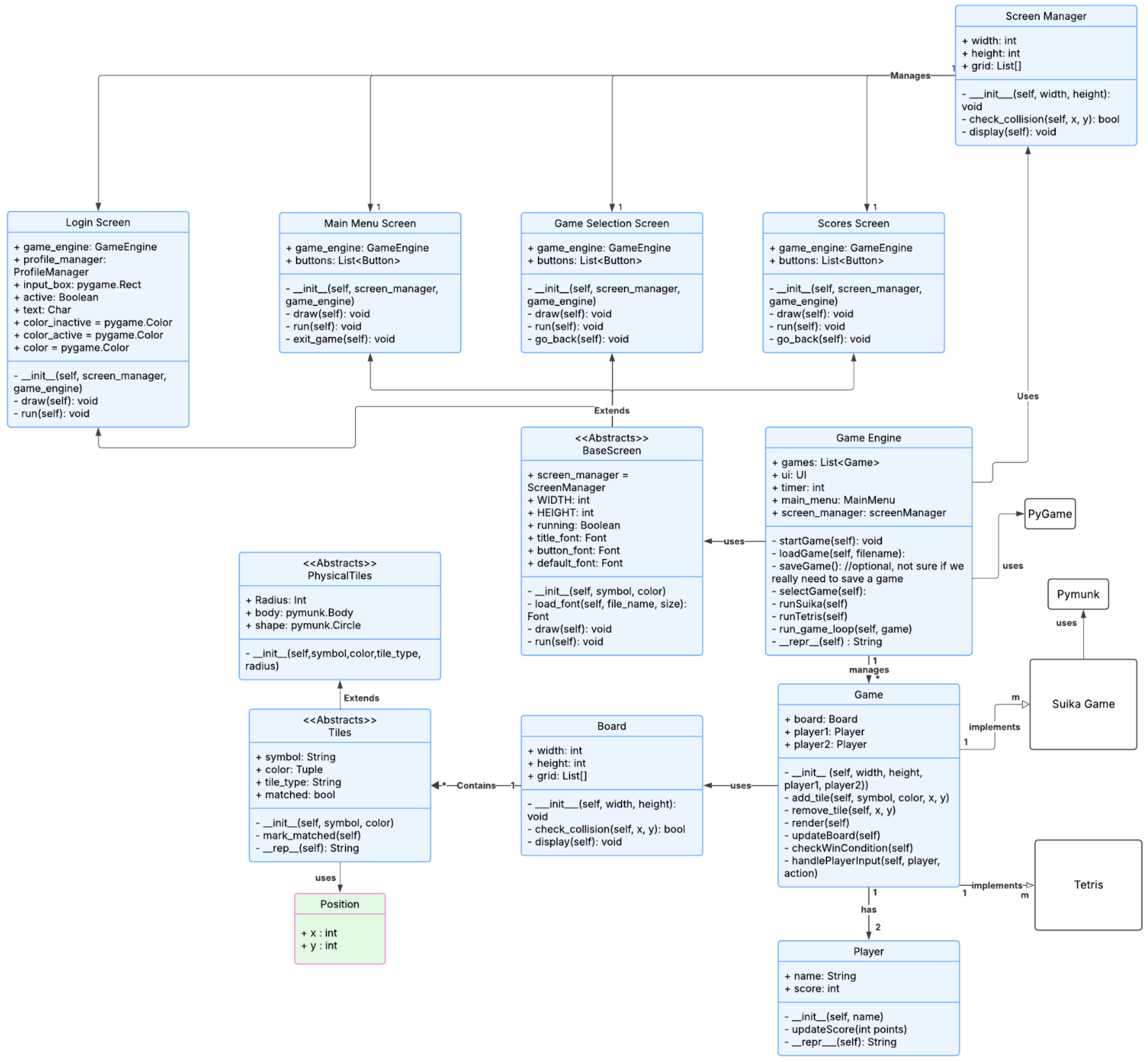
    - Player1: A: left, D: right, W: Rotate piece, S: Soft drop, Space: Hard drop
    - Player2: ←: left, →: right, ↑: Rotate piece, ↓: Soft drop, Right Ctrl : Hard drop
- After a piece lands, it locks in place and spawns a new piece
- Completing a row will remove the row, increase score, and adjust speed based on level
- Players cannot move pieces outside of the container

Reproducible Bug: Difficulty rotating pieces near the edges of the container

Loss Condition: If new Tetrimino cannot spawn without collision, **Game Over**.
Two Player Mode:

- Both players play simultaneously
- The Player with the highest score will win once both players meet the loss condition

UML Diagram



UML Descriptions

GameEngine

Attributes:

- List<Games>: A list of available games
- Timer(int): Timer variable
- main_menu(MainMenu): For managing the main menu
- screen_manager(ScreenManager): For managing the screens

Methods:

- __init__(self, screenManager): Initializes the GameEngine with a ScreenManager
- startGame(Self): Stars the selected game
- loadGame(self, fileName): Loads a selected file
- saveGame(self, fileName) Saves current game to a file
- selectGame(self): Switches ot the game selection screen
- runSuika(self): Runs a suika instance
- runTetris(self): Runs a tetris instance
- run_game_loop(self,game): Runs the main game loop for the selected game
- __repr__(self): String
    - Returns a string representation of the game engine

BaseBoard

(Abstract Base Class for All Screens)

Attributes:

- screen_manager: Reference to ScreenManager
- WIDTH, HEIGHT: Screen dimensions (1200x800)
- screen: The main pygame display surface
- title_font: Font used for titles
- button_font: Font used for buttons
- running: Boolean indicating whether the screen loop is active

Methods:

- __init__(self, screen_manager): Initializes screen properties and fonts
- draw(self): Abstract method for rendering the screen elements
- handle_event(self, event): Abstract method for handling input events
- run(self): Runs the screen loop, processing events and updating the display

MainMenu

Attributes:

- buttons: List of Button instances for navigation
- game_engine: Connected to game engine

Methods:

- __init__(self, screen_manager, game_engine): Initializes the main menu with navigation buttons
- draw(self): Renders the title and menu buttons
- run(self): Runs the main menu loop

GameSelectionScreen

Attributes:

- buttons: List of Button instances for selecting Suika, Tetris, or going back

Methods:

- __init__(self, screen_manager, game_engine): Initializes game selection screen with buttons
- draw(self): Displays the screen title and selection buttons
- run(self): Runs the selection screen loop

HighScoresScreen

Attributes:

- buttons: List of Button instances for navigation

Methods:

- __init__(self, screen_manager, game_engine, player): Loads the player's high scores
- draw(self): Displays the player's high scores and name
- run(self): Runs the high scores screen loop

LoginScreen

Attributes:

- game_engine(GameEngine)
- profile_manager(ProfileManager)
- input_box(pygame.Rect: Used to collect login name
- Active (Bool)
- text (char)
- color_inactive(Pygame.Color)

- color_active(Pygame.Color)
- color(pygame.Color)

Methods:

- __init__(self, x, y, width, height, text, font, color, hover_color, action): Initializes a button with text and an action
- draw(self, surface): Renders the button on the screen
- check_click(self, event): Detects and executes the button's action on a click

Button

Attributes:

- x, y, width, height: Position and size of the button
- text: Button label
- font: Font used for text
- color: Default button color
- hover_color: Color when hovered over
- action: Function executed when clicked

Methods:

- __init__(self, x, y, width, height, text, font, color, hover_color, action): Initializes a button with text and an action
- draw(self, surface): Renders the button on the screen
- check_click(self, event): Detects and executes the button's action on a click

Game

Attributes:

- Board: An instance of the board, manages the state of the game boar
- Player1 and Player2: An instance of the Player class, tracks the state and actions of a player

Methods:

- __init__(self, width, height, player1, player2)
    - Initialized the game with a board of specified dimensions and two players
    - Sets up the game environment

- add_tile(self, symbol, color, x, y)
    - Places a tile on the board at a specified coordinate if the position is valid
- remove_tile(self, x, y)
    - Removes a tile from the board at (x,y)
- render(self)
    - Renders the current state of the board
- Intialize_board(self)
    - Abstract method to set up initial state of the board, ensures that all game implementation define how board is initialized
- update_board(self)
    - Abstract method to update the board state according to game rules

Player

Attributes:

- Name (String): The name of a player
- Score (int): The player's current score

Methods:

- __init__(self, name)
    - Initializes a player with a given name and sets their score to 0
- updateScore(self, points)
    - Updates the player's score by adding a specified amount
    - Allows the games to increment the players score
- ___rep___(self): String
    - Returns a string representation of the player
    - Used for debugging

Board

Attributes:

- Width (int): The number of columns on the board
- Height (int): The number of rows in the board

- Grid (List[]): A 2D list representing the board's tiles, each cell can contain a tile object or None (an empty cell). Stores the current state of the board

Methods:

- __init__ (self, width, height)
    - Initializes the board with specified dimensions and creates an empty grid where all cells rae set to none
    - Sets up the board for use in the game
- check_collisoin(self, x, y): bool
    - Checks if a collision occurs at a specified point (x,y).
    - Returns true if a collision occurs
    - Used to determine if a tile can be placed at a specific location without overlapping or going out of bounds
- display(self)
    - Prints a text-based representation of the board in the console
    - Provides a simple way to visualize the board during development

Tiles

Attributes:

- Symbol (String): A string representing a tile (e.g., "O" for Tetris", "A" for suika). Defines a visual representation for the tile
- Color (Tuple): An RGB tuple representing the tile's coloring
- Tile_Type(String): The category of tile: (e.g.,"fruit" for Suika, "block" for tetris).
- matched(bool): A flag indicating whether the tile was matched. Mainly for suika, but allows for future tile-matching games to be implemented

Methods:

- __init__(self, str, tuple)
    - Initializes a tile
    - Sets matched to false by default
- mark_matched(self)
    - Marks a tile as matched by setting matched to True
- __repr__(self): String

    - Returns a string representation of a tile

Physical tiles

Attributes:

- Radius(Int): Radius of a tile, used for circular shapes in suika-like games
- body(pymunk.body): Enables physics-based movement and interactions
- shape(pymunk.Circle): Represents the tile as a circle with a specified radius

How to create a game with the TMGE

1. Overview of the TMGE Framework
   The TMGE framework consists of the following core classes:
    - Tile: Represents a single tile in the game. Each tile has a symbol, color, and type.
    - Board: Manages the game grid and provides methods for adding/removing tiles and checking for collisions.
    - Player: Represents a player in the game, including their name and score.
    - Game (Abstract Base Class): The base class for all games. You must extend this class and implement its abstract methods.
    - GameEngine: Manages the game loop, handles player input, and runs the selected game.
2. Technical Prerequisites
    - Have Python (3.7 or later) installed
    - Install pygame dependencies for rendering
3. Creating the Game
    a. Create a new python file and extend the Game class
    b. Implement the required abstract methods:
        i. initalize_board(): Setup the initial state of the board
        ii. update_board(): Update the board state based on game rules
        iii. check_win_condition(): Check if a player won
        iv. handle_player_input(): Handle player input for movement/actions

    c. Once your game class is ready, add it to the GameEngine so it can be selected and played
        i. Add it to the games list in game_engine.py
4. Run the GameEngine to start your game
5. Implement Game-Specific Logic
    a. Since each game will have unique rules and mechanics, they will require game-specific logic. Some examples include: Gravity/Physics, different player inputs, tile matching, different scoring, different win/loss conditions

Reflection

This project has definitely had some ups and downs over the course of the quarter. Initially, our group was excited to choose two completely different tile-matching games of suika and tetris to implement - since it was going to be a challenge and suika was newer to this type of game genre.

Our group faced two major challenges - the implementation and understanding of the TMGE and overall communication. There was a definite difference of scope in how the final project was described in lecture and in the canvas assignment, leading to confusion amongst our team. In the lecture, there was more importance placed on the various games that students were allowed to create whereas in the canvas assignment, there is more of an emphasis on the game environment itself. As a result, our team initially placed more importance on the individual games rather than creating a TMGE that would support the creation of future games. We ended up having to reevaluate our project and rewrite a bunch of code so that it would meet project requirements.

The other major challenge we had as a group was team communication. Internally, we didn't meet to discuss the project until about a week into the team's formation, and communication throughout the project didn't really pick up until the final week of the course. Another pain point was that we didn't set any clear roles or tasks for each team member, rather we picked up tasks as they needed to be implemented. Not meeting with the professor as a group also contributed to the confusion with the TMGE requirements.

Some high points of our project include the full implementation of Suika and Tetris, as well as the final submission of the project. Once both games were completed, there was a real shift in team morale as we were nearing the deadline. While there were tweaks we needed to make to the TMGE, as well as the games themselves, we felt much closer to the finish line and it really helped the team make strides in the final week.

Overall, this project has been a valuable learning experience. We should've asked clarifying questions closer to the beginning of the project to avoid confusion, and communicated more often to make sure we were all on the same page.