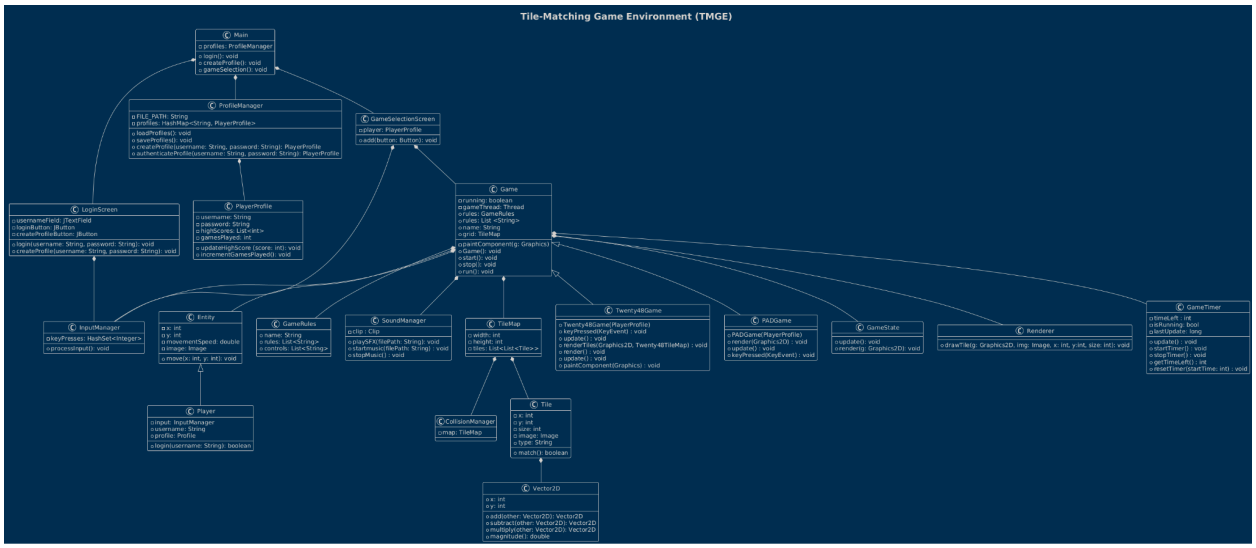
TGME Documentation

A Tile-Matching Game Engine by: Kelsey Beaulieu, Jackson Podgorski, Megan Santagata, Alice Tran, & Albert Youssef



Developer Guide

There are many parent classes already created for use for developers that are described above. We recommend making child classes if you need extra methods and attributes (which you likely will).

Running the engine:

The game engine runs from LoginScreen.java which will create a GUI and ask a player to create a profile and log in before accessing any games. A player's high score in each game will be saved, and can be viewed. Developers do not need to change LoginScreen or Player classes to make a functioning game.

Making a Tile:

We recommend starting small with the tile class first. This class already has an x and y coordinate, size, and an image. You may want to add functions in this class such as checking if a tile is empty, the value of a tile, or updating a tile value. Additionally, if tiles are moving together as a clump, a new class that houses the tiles can be made.

Making a TileMap:

A TileMap houses the Tile objects. The parent class provides a width and height attribute and a 2D matrix, tiles, in which to store Tile objects. You may want to create a child class to add methods such as initializing your tilemap with tile objects, adding tiles, removing tiles, and checking for specific patterns on your map. We also recommend any tile movement logic be implemented here such as checking if tiles can be placed in a certain spot, and moving tiles on the board.

Making a Game Class:

The Game class is abstract, so a child class must be implemented to create a game. The game class is meant to be a bridge between all elements of a game: the player, the tilemap, the UI, etc. The game class should handle taking input from a user and calling the correct functions on all of the game objects.

The game class sets up new instances of these elements upon init(). The game will require and store a player profile. There are two crucial methods to creating a game.

The update() method is where you will implement the game logic that needs to be executed on each frame. This could include updating the positions of game objects, checking for collisions, and handling user input. This should also include checking for win or lose conditions, ending the games, and replacing highscores. There are optional additional classes such as CollisionManager and GameLogic that can be used to make your game more modular which should be stored in this class as well.

The render() method is where you will implement the drawing logic for your game. This method is called to render the game objects to the screen. It is responsible for creating all of the visuals of the game on the screen dynamically with each frame update, including tiles, text, and images in the 2D space.

The developer's child Game class init() should call start to start the game after initializing all additional attributes. We recommend adding a keylistener in the init() as well.

We recommend storing game rules, background images, timers, sounds, and any other game specific information as an attribute in this class. A game's TileMap should be stored in this class as well.

Collision Manager:

A class that has a TileMap as an attribute and can be extended to handle collisions if you want to implement that logic outside of your TileMap.

Entity:

The abstract Entity class can house game objects other than tiles, if they are needed for your game. Entities have an x,y position, movement speed, and image attributes.

GameLogic:

An abstract class that has functions checkWinConditions(), isGameOver(), updateScore(), getScore() that can be overwritten for a more modular approach. Game logic can also be implemented in the Game class for a less modular approach.

Font Manager:

The Font Manager loads and manages fonts for each game, ensuring that the text is rendered consistently across all screens.

Adding Files:

Images, sounds, and other files that enhance the game can be added to the assets folder.

Sound Manager:

Sound Manager is responsible for handling all audio elements, including background music and sound effects. It does this by loading audio files from the assets directly and playing them in sync with game events.

Renderer:

The Renderer is responsible for handling all graphical game components on the screen using Java graphics APIs.

Vector 2D:

The Vector 2D class is used to manipulate tiles within the tilemap. It controls shifting tiles and can also be used to handle acceleration for gravity and other more complex movement, based on the game being implemented.

GamePanel:

An abstract class that is used to nest game panels inside the main game frame. This is necessary when using Java Swing to create more than one game instance in the case of multiplayer. If you wish to display rules, a timer, or score, they can be attached to the game panels.

Since local competitive multiplayer is used, the game must support two instances in one window. This can be done by creating two panels inside of the game panel class, each with their own instance of the Game class, and attaching them to the frame.

Development Cycle

What Went Well

The initial development of the game engine was straightforward and was created early on. We outlined the manager classes we would need, as well as the abstractions and interfaces that could be extended to create specific games. The initial design was well thought out and allowed us to have a solid template for working on our games. We also had plenty of communication with our entire team throughout the process, and divided the work evenly between both games, as well as the game engine.

Challenges Faced

Implementing multiplayer was trickier than we anticipated. We wanted to use Java threading to create 2 game states for each player to control independently, but we ran into a lot of issues with multiple key bindings, as well as creating the same initial game state but independent inputs. We used Java Swing and the panels were tricky to work with in terms of formatting and getting them to do exactly what we wanted them to do. In the future we would definitely use JavaFX.

Game Rules: 2048

In 2048, the goal is to combine tiles on a 4x4 grid to create a tile with the number 2048. Players move tiles causing all tiles to shift and combine if they are the same number. The game ends when the player creates a 2048 tile, or when the player has no valid moves left.

Players can move tiles in four directions by pressing arrow keys. Tiles will move in that direction as far as possible, without overlapping with other tiles. If two tiles are the same number, they will combine during a move. If a tile is a result of a combination done by the current move, it cannot be combined again. If the move is valid, a new 2 tile will spawn in a random open spot on the

board. If no tiles can move or be combined in the selected direction, then the move is invalid, and a new tile will not spawn.

The player's high score is the highest value of the tiles combined together that the player has reached. For multiplayer, players will compete to solve the puzzle with the shortest time. Whoever has the shortest time and solved the game will be the winner of the two players. If the player does not create the 2048 tile, it will be considered a lost game, and the high score will not be updated.

Game Rules: Puzzles & Dragons

In Puzzles & Dragons, the goal is to achieve the highest score possible by matching colored tiles on a grid within a 30-second time limit. Players move a selection of cursors using WASD and can lock onto a tile with the F key to swap it with adjacent tiles. When three or more contiguous tiles of the same color align horizontally or vertically, they are cleared from the board, and new random tiles fill the gaps–each cleared tile adds to the score. Matches of more than 3 tiles apply a multiplier to the points earned.

A health bar gradually decreases over time and is penalized if a move does not result in any match, while successful matches restore some health. The game ends when the timer runs out or the health bar reaches zero. Each game session logs the highest score into the player's profile resulting in a fun competitive game with a high skill ceiling.

Instructions for Running the Game

To run the game, run the main() function in LoginScreen.java. This will create the login screen where a player can log in or create a profile and then log in. Player data saves into a file within the Project folder. Once logged in, a player can select one of two games, 2048 or Puzzles & Dragons. The selected game will start. Games can have different win and lose conditions, or the player may be able to stop a game. If a player ends a game, and it is not considered a loss, and the score is better than their previous highscore, or it is their first time playing the game, their score will be stored in their player profile.