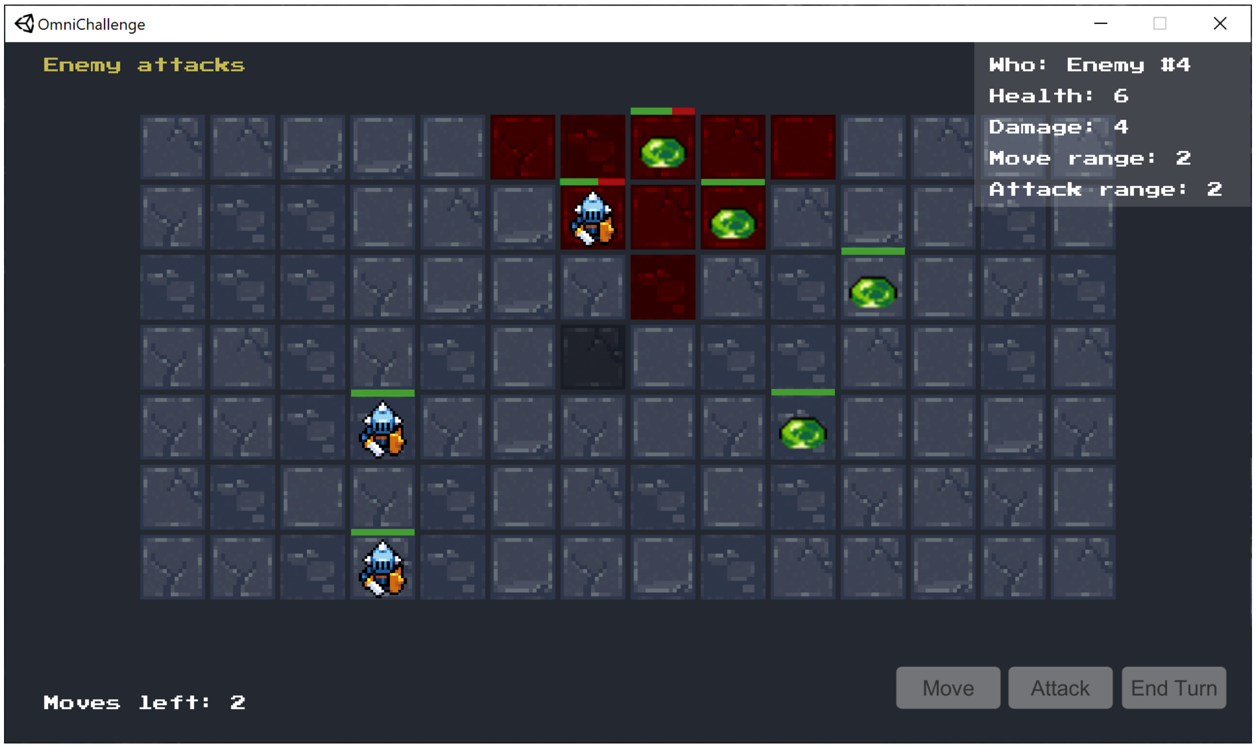
**Project name:** OmniChallenge

**Short description:** Turn-based game where player controls ally units and should defeat enemy units controlled by basic AI. The game supports both mouse and touch controls.

**Author:** Mike Smahin



## General info

OmniChallenge is an abstract name of the game. The game is turn-based where the player controls ally units. Ally units are spreaded randomly on the left side of the battlefield. Number of units from both sides could be tuned in the Unity Editor.

Every round consists of a number of moves. During every move the player could select a different ally unit and decide whether to move or attack. After clicking the corresponding button the colored range appears. Green color for moving and red color for attacking. The player could attack only if the enemy is within the attack range. Also the player could end the turn prematurely. Enemy side starts to move or attack only after clicking the “End Turn” button. The same rules are applied for the enemy side as well. The basic AI controls the actions of the enemy side.

The game ends after one side eliminates the units of the other side. The corresponding screen appears. The player could restart the game by pressing the “Restart Game” button.

The UI of the game consists of the several parts:

- Central. The battlefield with the alive units.
- Top-Left. The current action of the selected unit. Like “Enemy thinks”.
- Bottom-Left. Number of left moves.
- Top-Right. The stats of the selected unit. The player could also select enemy units but only to review their stats.
- Bottom-Right. Three buttons to control the units and the game itself.

Also all the units have the health bar above their heads (if the slimes have the one).

## Structure of the project

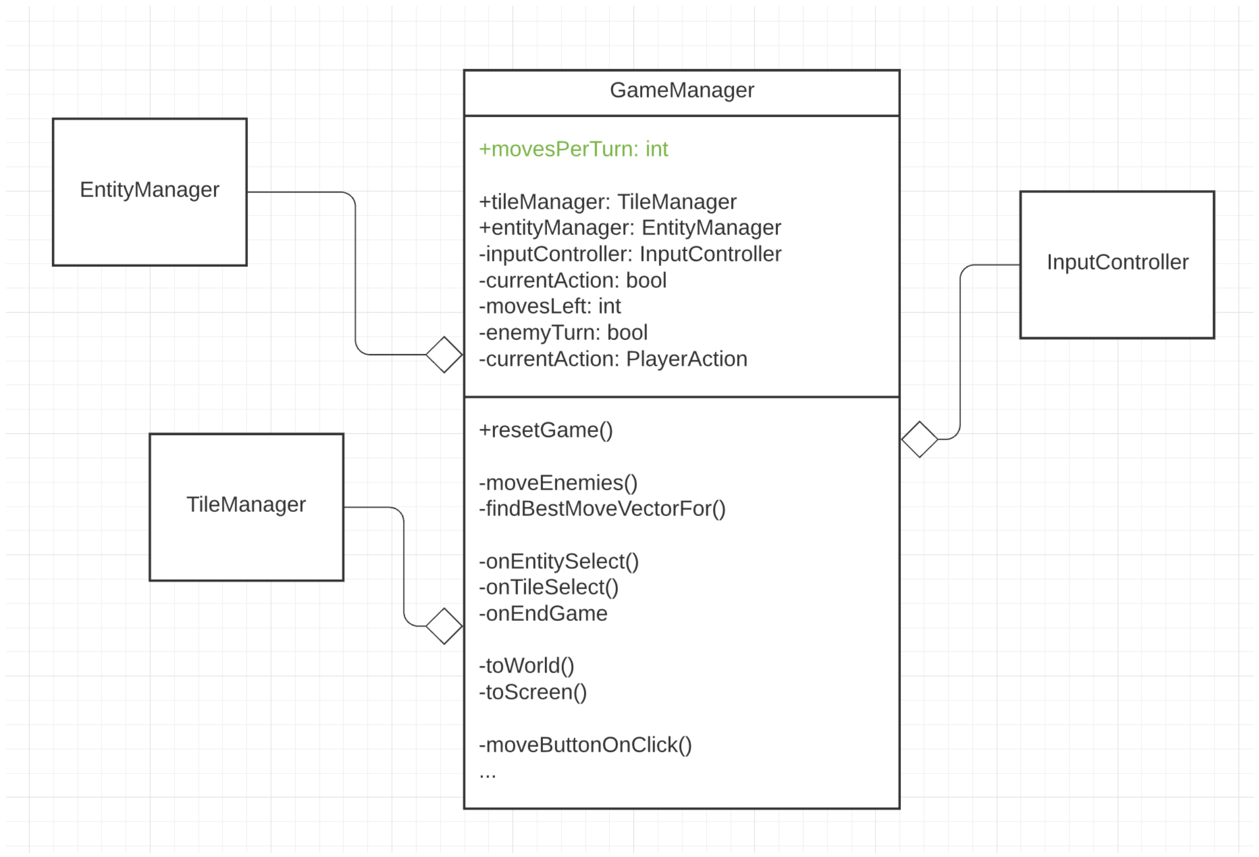
OmniChallenge

- Assets
    - Fonts
    - Prefabs
    - Scenes
    - Scripts
        - Entities
    - Sprites
- Executable
    - OmniChallenge.exe
- Packages
- ProjectSettings
- Documentation.pdf

# General architecture

There are a couple of modules in the game.

## GameManager



The fields marked green could be tuned in Unity Editor.

**GameManager** is the main class in the game. Is responsible for:

- Coordinates the work of **TileManager** and **EntityManager**.
- Controls behaviour of the enemies through the basic AI. See region “Enemy logic” in the code.
- Handles button clicks. See the region “Button handlers” in the code.
- Manages **PlayerAction** states. See below.
- Decides when the game is over and shows the appropriate screen.
- Handles different events like onEntitySelect(). See “Game events” region in the code.
- Implements methods for translation coordinates from World to Screen and vice versa. See “Camera” region in the code.
- Updates screen UI. Like showing statistics of the units.

## In general

From the start **GameManager** calls resetGame(). This method populates tiles and units, resets different fields.

Then **GameManager** sets the currentAction field to Idle. This means that the game waits for the actions from the player. From this point the player could choose different units. onSelectEntity() or/and onSelectTile() methods are called. These methods among others could change the field currentAction. For example, from Idle to SelectedEntity.

The field currentAction of type **PlayerAction** has the following states:

- Idle
- SelectedEntity
- PressedMoveButton
- PressedAttackButton

Due to the field currentAction the game always knows what's going on. Also it prevents wrong actions. For example, the player cannot observe the move range of the unit but decide to attack.

So we selected the ally unity. From this point the player could hit different buttons. The corresponding handlers are called. Like moveButton_OnClick().

After hitting the button "End Turn" the game starts the enemy turn cycle (see below the description).

If the player kills all the units (or vice versa), then the onEndGame() event is called. The specific UI is shown. The player could hit the button "Restart Game" to invoke resetGame() method. And the game restarts. Also this method destroys properly old tiles and units.

## How basic AI works?

When the player hits the "End Turn" button **GameManager** starts the enemy turn sequence. It calls the coroutine moveEnemies(). There are the following steps:

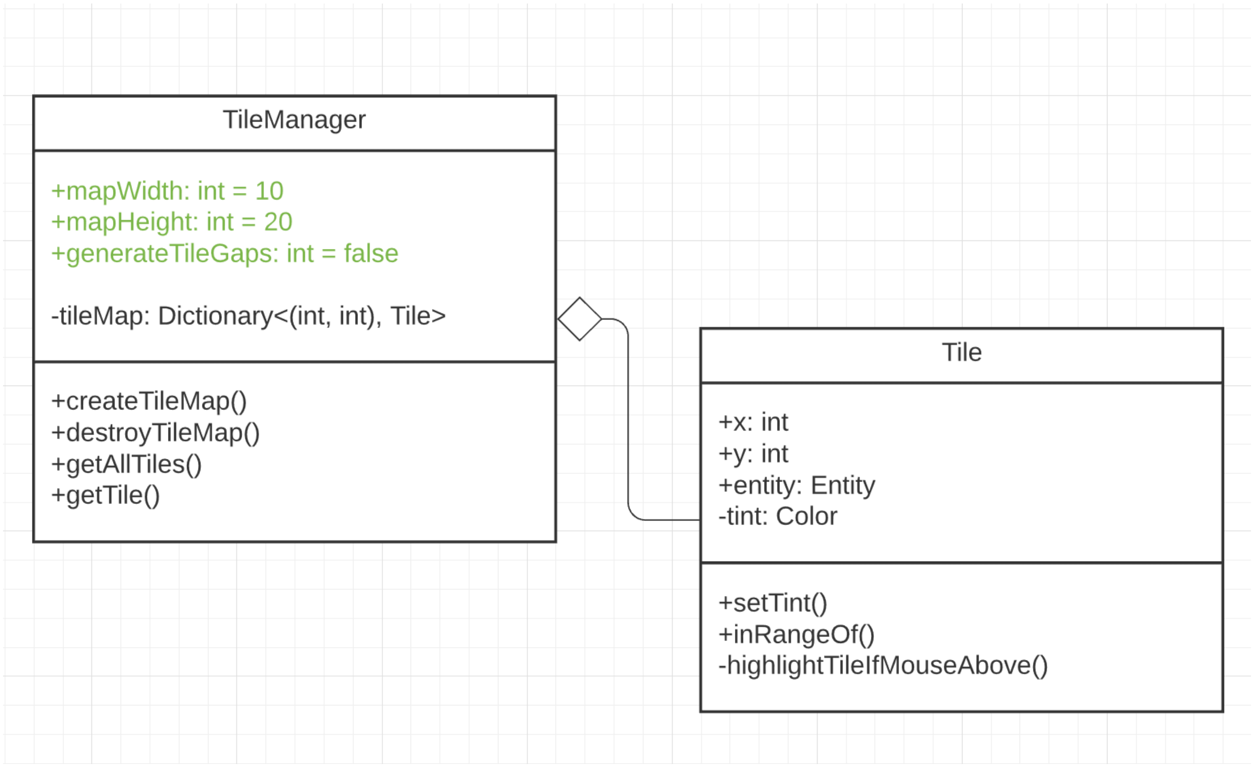
1. Select the random enemy unit.
2. Find the nearest ally unit.
3. If the ally unit is in the attack range, then attack.
4. Otherwise call the method findBestMoveVectorFor() to find the optimal direction to the nearest ally unit. And move in this direction if it's possible.
5. Repeat the sequence while at least one ally unit is alive and there are moves left.

The method findBestMoveVectorFor() consists of the following steps:

1. Find the nearest ally unit.
2. Define the unit direction vector.
3. Select randomly the axis to move along.

4. Try to take the biggest step within the moving range. If the tile is taken, then decrease the step length. For example, if the moveRange == 2, then the unit could consistently take the next steps: (2, 0) -> (1, 0) -> (0, 0) -> return.

## TileManager

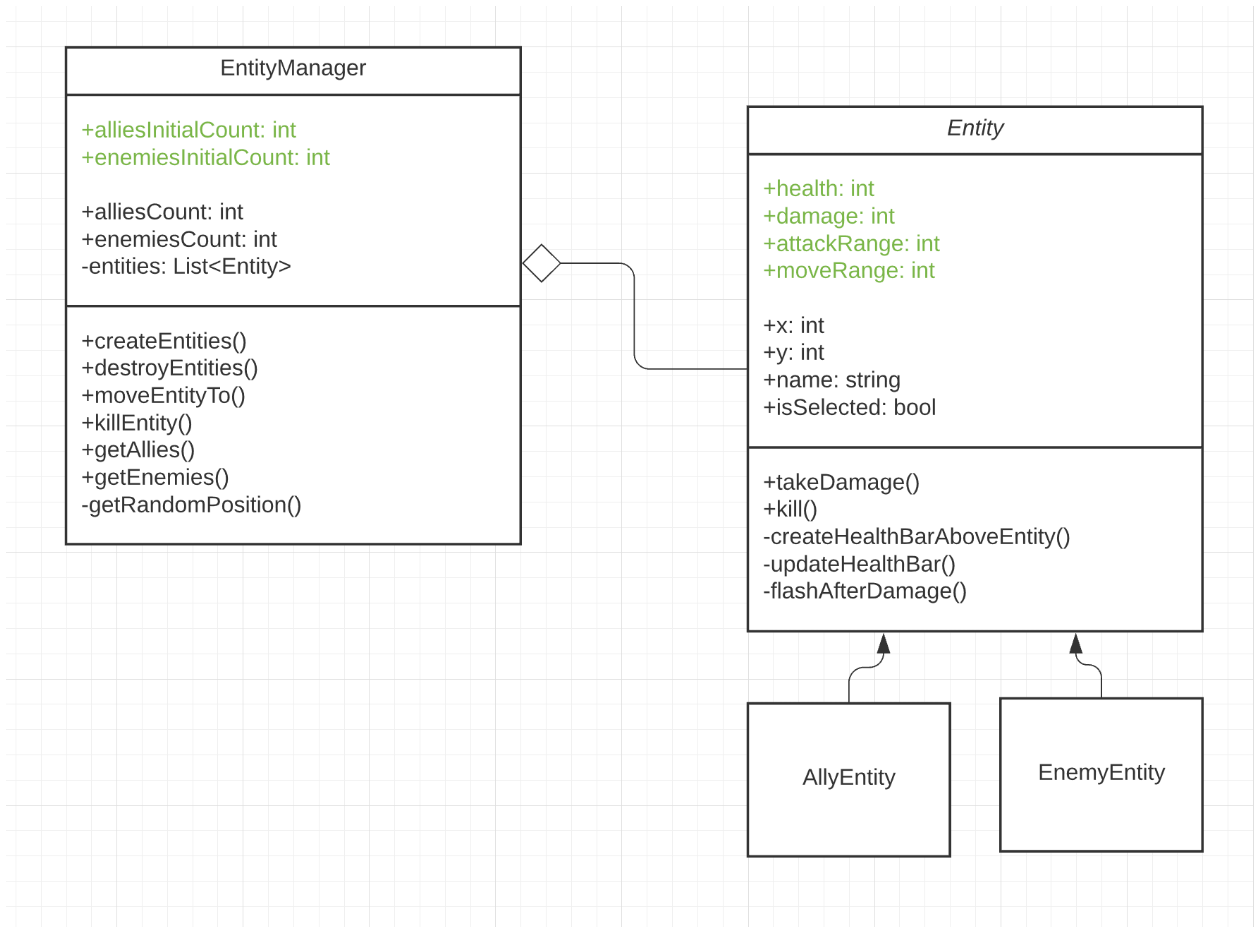


The fields marked green could be tuned in Unity Editor.

The class **TileManager** is responsible for creating, storing and providing access to the different tiles. The game battlefield consists of a grid of tiles that are stored in tileMap. The specific tile could be obtained via method getTile().

The field generateTileGaps is optional but it could be used to add random gaps between tiles on the battlefield. Of course units could not stand on the gaps.

## EntityManager

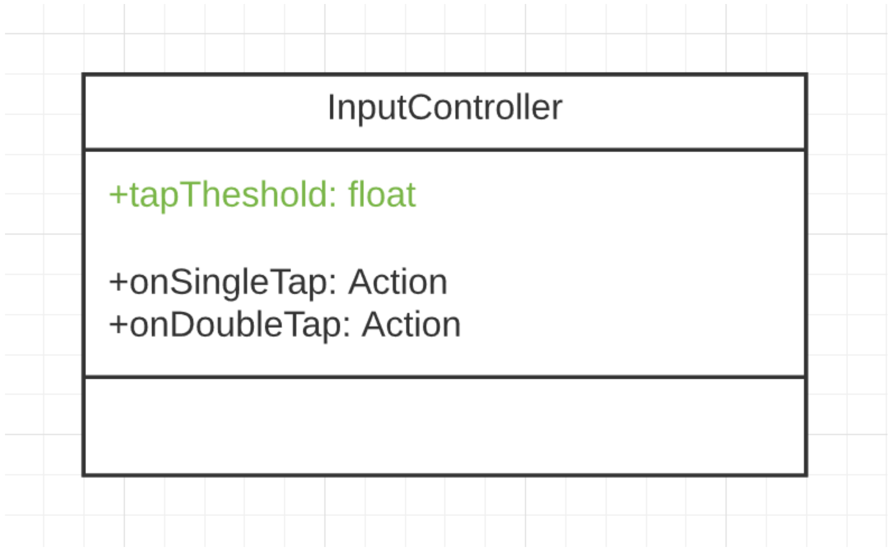


The fields marked green could be tuned in Unity Editor.

The class **EntityManager** is responsible for creating, storing and providing access to the different entities. It holds the instances of inheritors of **Entity** class.

The **Entity** class is the abstract one, but it provides all the logic. The inheritors **AllyEntity** and **EnemyEntity** are minimal at the moment but could be modified to provide unique behaviour or outlook in the future. For example, add the trail after the enemy slime moves.

## InputController



Detects the mouse clicks or the touches. **GameManager** is subscribed and listens to the event onSingleTap().

## HealthBar

The script controls the corresponding prefab and lets you update the health bar.