Technical Documentation of MECH.AI

This is a live document. The content can change anytime.
Words with yellow highlights are needed to discuss further.
Disclaimer: we are not focusing on UI for now.

V 0.0.3 – Implement procedural environment
By Ardiawan Bagus Harisa

Changelog:
V0.0.2 –
V0.0.1 –

Contents

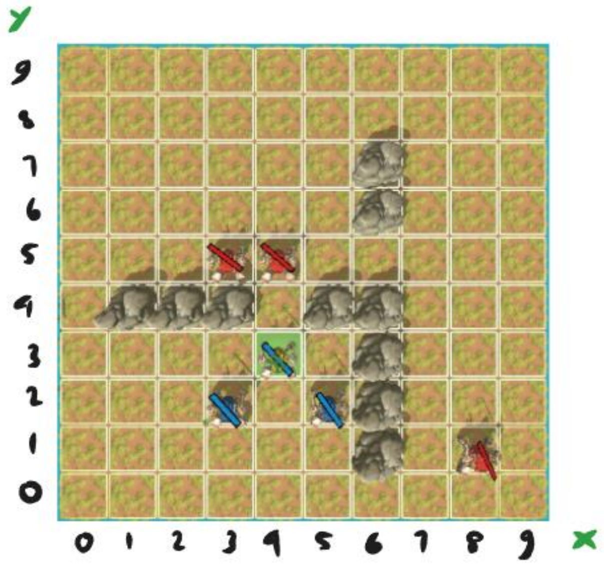
Game Design

World

The world of this game uses a board, 10x10 grids size but may varies in the future. One unit of the board is called “Grid”.

Board

The board contains grids in which each grid has an object. The size is 10x10 grid.



Grid

Grid is the smallest unit on the board. It was selected by the coordinate/index [X, Y] and ID. The grid can have an obstacle or a robot.

Default	Obstacle	Robot

Obstacle

A grid obstacle has only one type, it is a mountain rock. The grid obstacle used to prevent robot action, movement and attack. This obstacle can be destroyed by the robot attack.

Robot

Just a tile to represent any robot’s position.

Default

By default, the type of tile is default, which means and empty and passable tile where later in the gameplay player can hover after choosing an action such as attack and movement.

Robot

In this game, a unit controlled by Player is a “Robot”. The Robot has a combination of actions. The actions are different by the chosen weapon. What makes a difference is status, range, and behavior. The status is based on the used equipment.

Status

The Robot has:
- Damage (current damage, temporary damage).
- Energy (Total energy, current energy).
- Defend (Total defend, temporary defend).
- Health (Total health, current health, temporary health).

Equipments

- Armor (Defines constant value of defend and health).
  The use of this item makes the robot stronger, having an initial big health point and defend point. (only affect robot status).
- Helmet (Ratio to constraint the armor result).
  The use of this item is changing the Armor result value. The robot health point can decrease while the defense point increases, and visca verse. (affect the armor)
- Weapon (next point).

Weapon Types

The Robot must bring a weapon to have action in battle. The weapon makes a different damage result and action range. These weapons differ by its range, **Default, Short, Long,** and **Projectile**.

Default Weapon

The default weapon or initial weapon before the robot picks any weapon. This has minimal status.

Short Weapon

This type is intended to make a robot frequently chase an enemy. And results in huge damage in close combat. Therefore, this type has a close range attack and a long range move.

Long Weapon

This type is intended to make a robot keep distance from an enemy. The robot attacks from a far distance and deals more damage compared to close attack deals less damage. The movement range is similar to the attack range.

Projectile Weapon

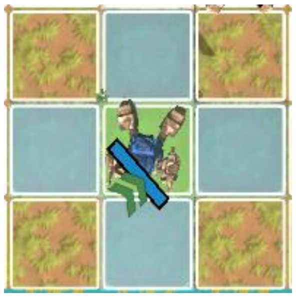
This type is intended to make a robot cover itself to not be detected by an enemy. With this weapon, a robot can attack through an object. The attack range is big to reach enemies from far. Because the attack range is big, the robot using this weapon can only move a few grids.

Robot Action

The robot has multiple actions used to destroy the enemies. Each action needs energy to be done.

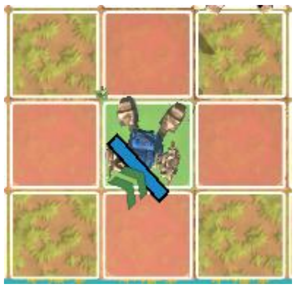
Move

The move action is used to change the robot position from one to another position. Move action can only be done one turn per robot. The robot can use this action again after the next round. This action is indicated by a blue highlight.



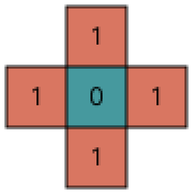
Attack

The attack action is used to damage the robot enemy. This action can decrease the number of enemy health points. Attack action can only be done one turn per robot. The robot can use this action again after the next round. This action is indicated by a red highlight. Action with a *through* keyword can ignore the obstacle or the other robot.



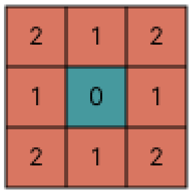
Highlight Shape

Cross



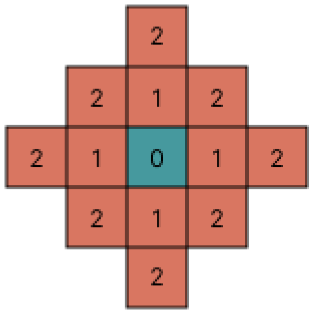
The minimal value is 1. The maximal is whatever. Example in the left image, **cross range: 1-1**

Square



The minimal value is 1. The maximal value is whatever greater than the minimal value and must be an even number. Example in the left image, **square range: 1-2**

Diamond



The minimal value is 1. The maximal value is whatever greater than the minimal value. Example in the left image, **diamond range: 1-2**

How Energy is used for Action

The move and attack action need an amount of energy to execute. The amount of energy is based on which grid is selected between (min - max), the value name is (distance). It is a default on how energy is used. Otherwise there is an Additional behavior named *Energy Reverse*, it is similar to default but with calculation (min+max) - (distance). If the energy is not enough to reach within grid distance, then the highlight can be subtracted.

Weapon Type Effect

The weapon type can affect the action behavior.

- Range (min-max).
- Energy used (how far the robot selects grid range).
- Highlight shape.
- Additional behavior.
- Damage result.

The effect of weapon type on move action:

Type	Range	Highlight	Additional
Default	1-1	Cross	-
Short	1-4	Square	-
Long	1-3	Cross	-
Projectile	1-2	Diamond	-

The effect of weapon type on attack action:

Type	Range	Highlight	Damage	Additional
Default	1-1	Cross	1	(Energy reverse)
Short	1-2	Square	7 - distance	-
Long	1-7	Cross	2 + distance	-
Projectile	3-8	Diamond	3 + (distance * 0.5)	(Through)

Gameplay

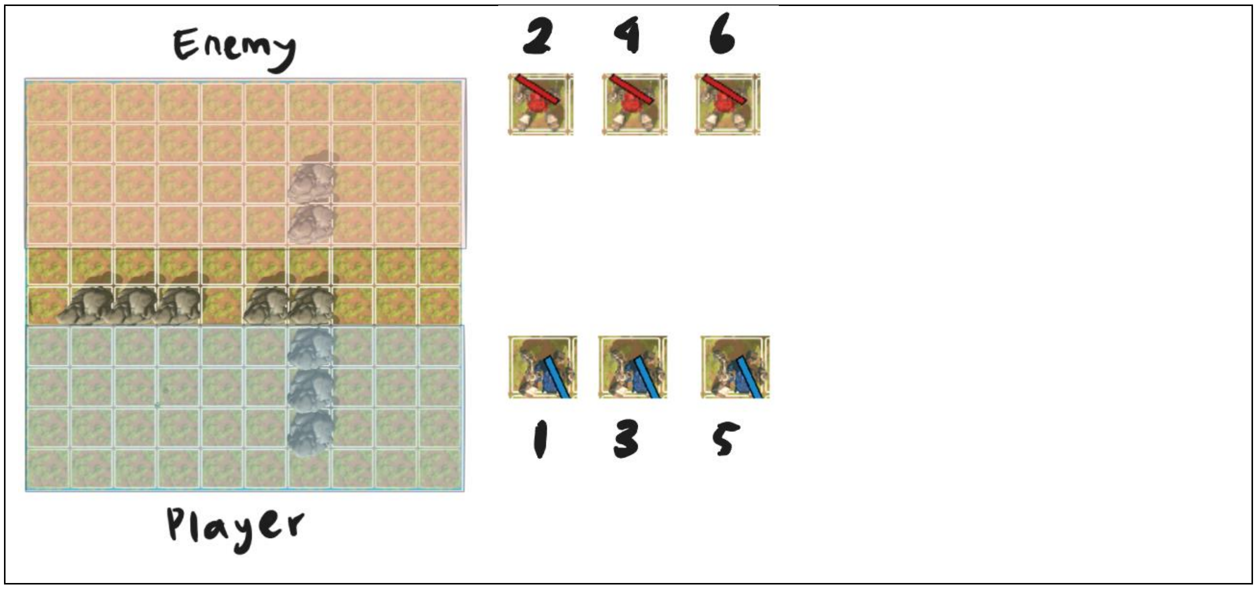
A turn-based game combines with a strategy game where two teams compete with their robots. In the board 10x10 grid, the player robots and the enemy robots select a bunch of actions within their turn.

Select Map

A map of the board can use a custom-created or generated one. The custom-created is created by the game designer by placing obstacles manually. Or uses generators that place obstacles automatically. The map generator uses the Wave Function Collapse algorithm.

Robot deployment

The game starts with the player and the enemy deploying each of their robots at the legal grid. This is intended for players starting with their strategy. Robots are deployed alternately between Player and Enemy like a chess game.



Battle

Uses available action that robot has to destroy robot enemies. The player can manage their robots actions in one each round.

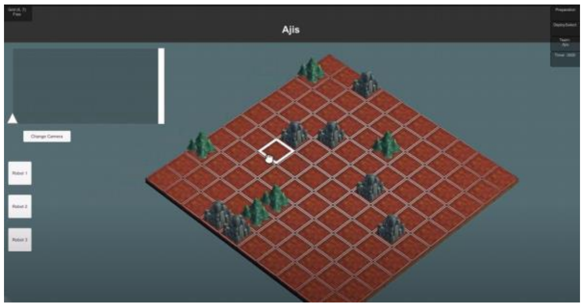
Objective

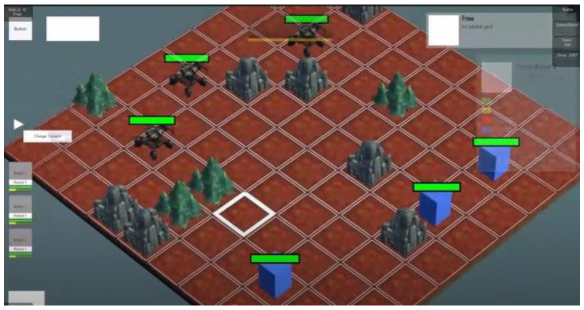
The game finishes when all the robot enemies are destroyed.

Technical Design

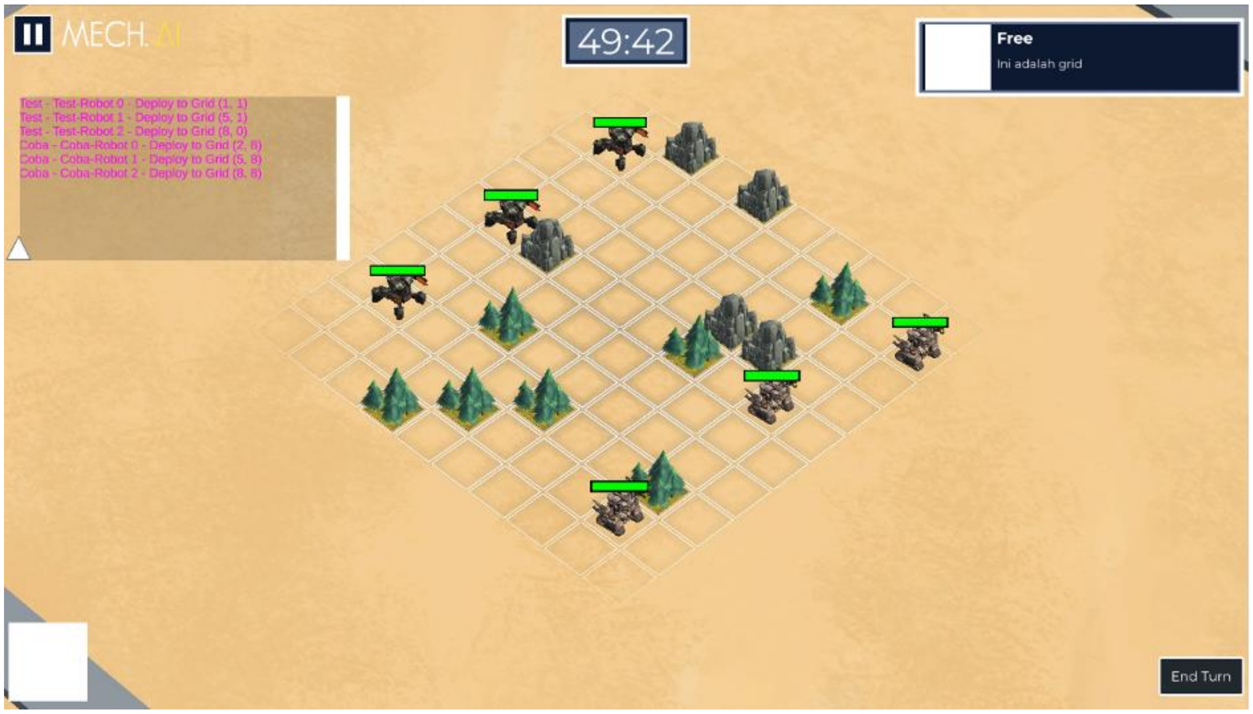
Camera

The game projected as isometric top-down view. Camera movement is controlled with the WASD and arrow keys. The camera speed is set in the parameter and player can also drag the view using left click+drag on the tiles.

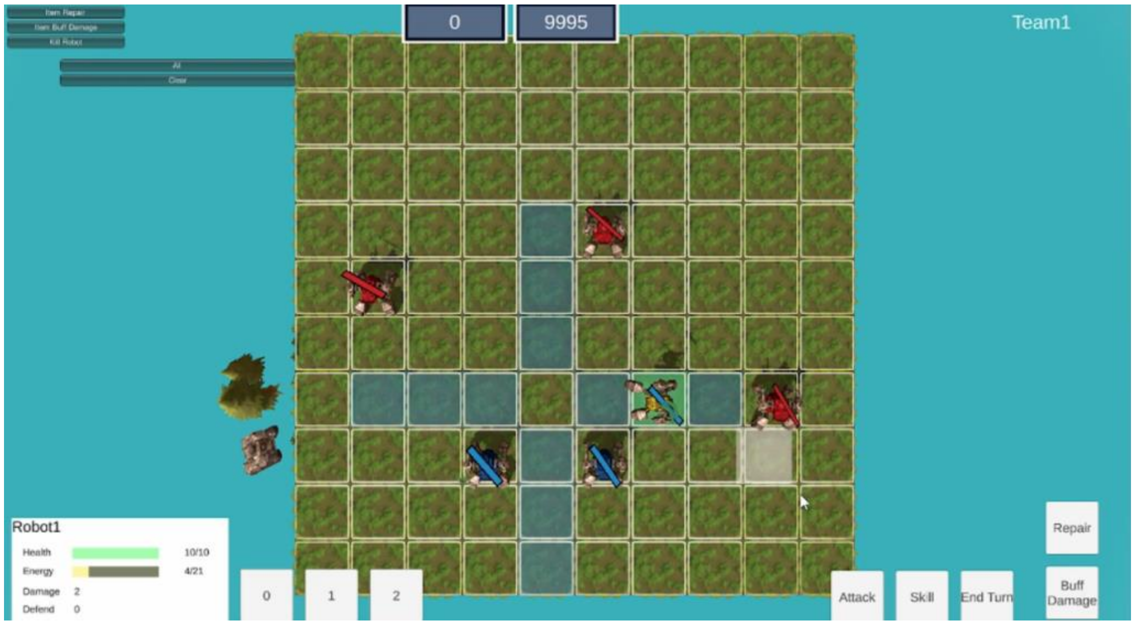




Past prototypes

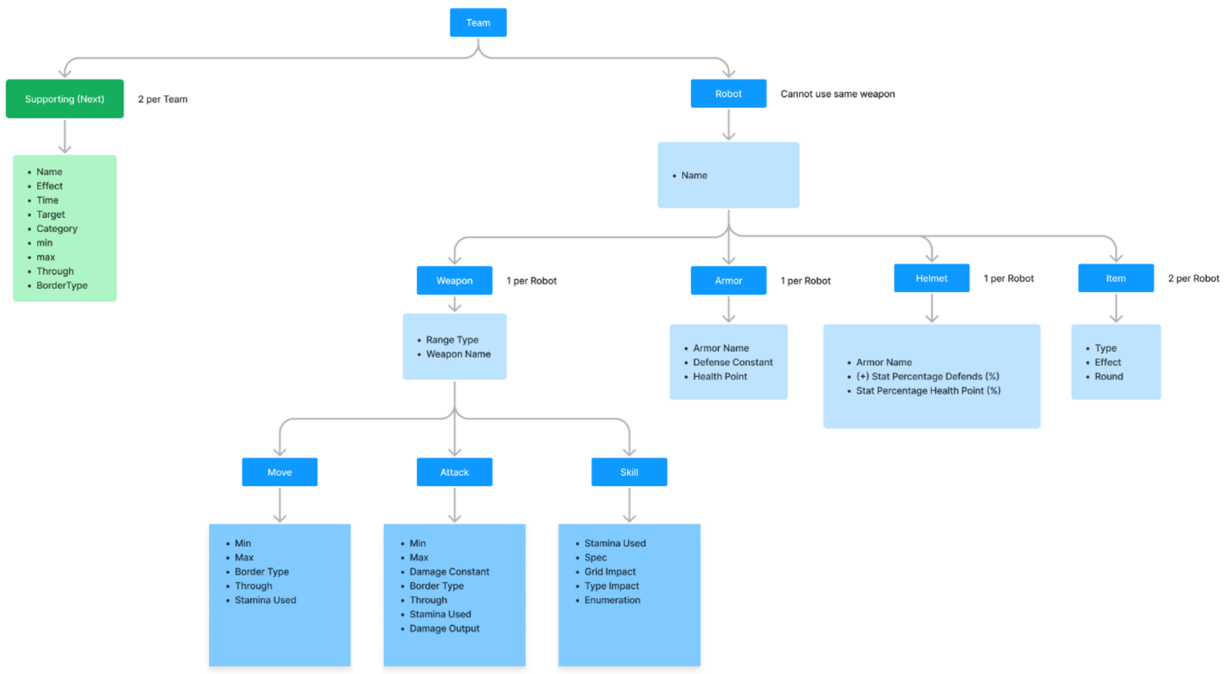


```
Test - Test-Robot 0 - Deploy to Grid (1, 1)
Test - Test-Robot 1 - Deploy to Grid (5, 1)
Test - Test-Robot 2 - Deploy to Grid (8, 0)
Coba - Coba-Robot 0 - Deploy to Grid (2, 8)
Coba - Coba-Robot 1 - Deploy to Grid (5, 8)
Coba - Coba-Robot 2 - Deploy to Grid (8, 8)
```



Robot Structure

This section explains the properties and behaviour of a robot.



Properties

HealthPoint	Robot’s health point
StaminaPoint	Robot’s stamina/ energy to perform action
StartPosition	Robot’s starting position when the game start

Team	Robot’s controlling team
RobotAttack	Robot’s attack actions
Robot	Robot
GridController	The manager script for handling the grid behaviour

Behaviour

Awake	Initiate by assigning the robot actions and positions
Update	Update robot’s health, stamina. Also check if the robot is dead → destroy.
SetTeam	Set the controlling team.
Move	Move the robot to the target position.
CalculateDistance	Calculate the distance (tiles) from the robot’s position to the target tile.
TransitionMove	Move the robot using smoothed animation.
ResetPosition	Reset robot’s position.

Stats Tables

Types

Range types

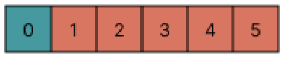
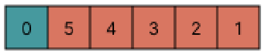
1. Short: Intended for close distance.
2. Long: Intended for long-range distance.
3. Projectile: Intended for long-range attacks & can penetrate obstacles.

Pattern/ Border types

1. Square: distance (1 to 2; allow diagonal)
2. Cross: distance (1 to 1)
3. Diamond: distance (1 to 2; allow diagonal)

Stamina types

1. Normal: the energy used is equal to the distance. Ranged min to max: 1 to 5.
2. Reverse: the energy used is reverse from max to min.

 Normal	 Reverse

Move types

Range Type	Weapon Name	Min	Max	Border Type	Through	Stamina Used
Short	Shotgun	1	4	Square	No	Normal
Projectile	Rocket Launcher	1	2	Diamond	No	Normal
Long	Sniper	1	3	Cross	No	Normal

Attack types

Range Type	Weapon Name	Min	Max	Damage Constant	Border Type	Through	Stamina Used	Animation	Damage Output
Short	Shotgun	1	2	7	Square	No	Reverse	Gunshot	Damage - Distance
Projectile	Rocket Launcher	3	8	3	Diamond	Yes	Normal	Bomb	Damage + (Distance : 2)
Long	Sniper	1	7	2	Cross	No	Normal	Arrow Shot	Damage + Distance

Skill types

Range Type	Weapon Name	Stamina Used	Spec	Grid Impact	Type Impact	Enumaration	Animation
Short	Shotgun	Constant (10)	Movement Max x 2	0	Nothing	Movement	Dash
Projectile	Rocket Launcher	Constant (10)	Impact = Damage	3 x 3	Square	Attack	Omni Fire Blue
Long	Sniper	Constant (10)	Through Obstacle & (Max + 3)	0	Nothing	Attack	Fire Ball

Armor types

Type	Defense Constant	Health Point
Aggresive	2	30
Balance	1	40
Defensive	0.7	50

Helmet types

Type	(+) Stat Percentage Defends (%)	Stat Percentage Health Point (%)
Aggresive	2	-0.4
Balance	1	0
Defensive	0	0.3

Item types

Type	Effect	Round
Buff Attack	+2 Damage Constant	1
Buff Armor	+1 Defense Constant	1
Repair	+10 Health Point	1

Supporting item types

Name	Effect	Time	Target	Category	min	max	Through	BorderType
StunGranate	Stun	Player 1 (current	Grid	Attack	1	8	Yes	Diamond

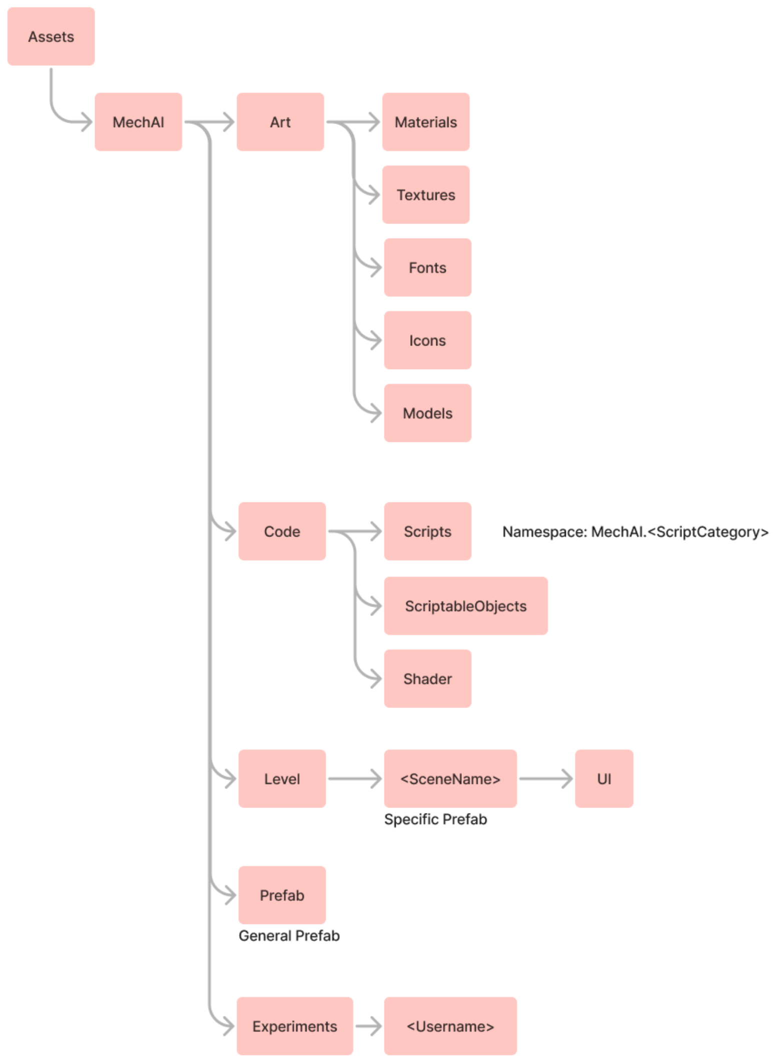
		round), Player 2 (next round)						
Smoke	Invincible	Player 1 (current round), Player 2 (next round)	Grid	Defend	1	8	Yes	Diamond
Shield	Spawning Obstacle	Player 1 (current round), Player 2 (next round)	Grid	Defend	1	8	Yes	Diamond

Game Inputs

All of the players' actions are done using UI button as shown in the previous figures to make it system independent.

File Structure

Follow the files structure as the image below. The Experiment only for the new feature. After your finished the feature, then move all file in your experiment to exactly the existing directory. If you want to add a directory, please discussit with project manager first so that it wont bloats the project.



Grid

As mentioned earlier, the space of the game is a 10x10 grids by default where we can use the index of x and y as identifier.

Calculating the Distance

After considerations, I suggest that we can use breadth frist search algorithm to calculate the distance of the source to the target tile. Refer to the figure.

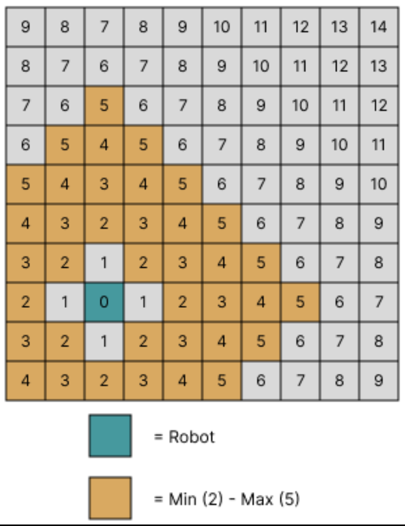
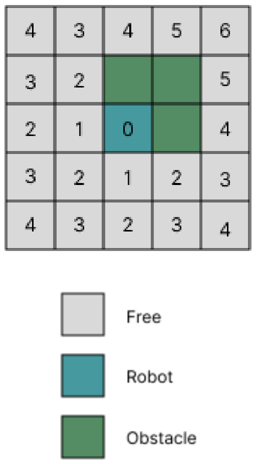
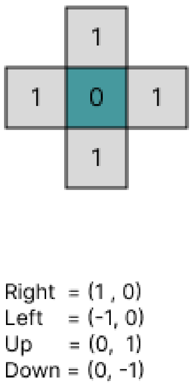
Grid States

There are three states of a grid: free, robot, and obstacle. I think it is really self explanatory.

Direction

Note that any action such as movement and attack will require direction to perform the action. There are four directions and we do not allow diagonal direction. The direction can be described by a Vector2(x,y).

1. Right (1,0)
2. Left (-1,0)
3. Up (0,1)
4. Down (0,-1)

Distance	States	Direction
		

Highlight System

Consider to split the highlight system from the grid to be a different class because this could get messy in the future, as we will need different variations of highlight such as for attack actions and for movement actions. Generally, we will spawn a quad as a highlight of a tile.

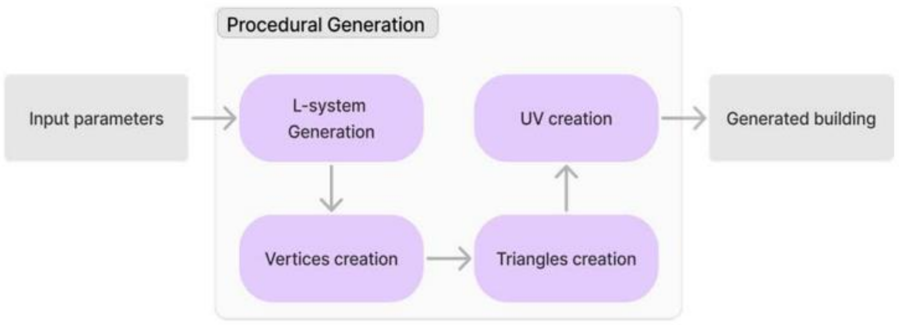
Parameters

1. Move: bool → a variable that shows what is the highlight is used for. Represented with blue tiles.
2. Attack: bool → a variable that shows what is the highlight is used for. Represented with red tiles.
3. Pattern: list → list of tiles which construct the patterns. Here, you assume that we will have structure of a pattern as list of tiles. As for now, there are four type of highlight patterns:
    a. Surround: create a 3x3 rectangle pattern.
    b. Tank row: create a triangle-type pattern.
    c. Diamond surround: create a diamond pattern
    d. Three ront: create a straight 3-grids perpendicular pattern.

Procedural Environment

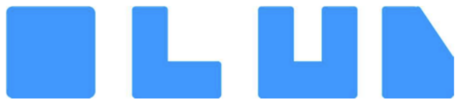
Procedural Building

This sections describe the technique to generate the building as procedural environment using L-system method.



Base Shapes

1. C: a cube-shaped building
2. L: an L-shaped building
3. U: a U-shaped building
4. R: a cube-shaped building with a slanted side at one corner



Roof Shapes

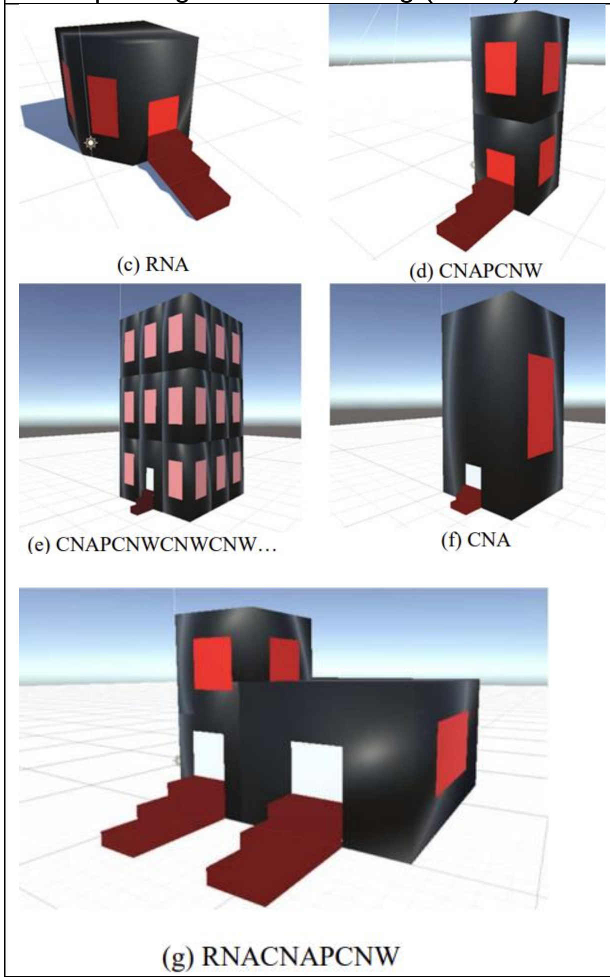
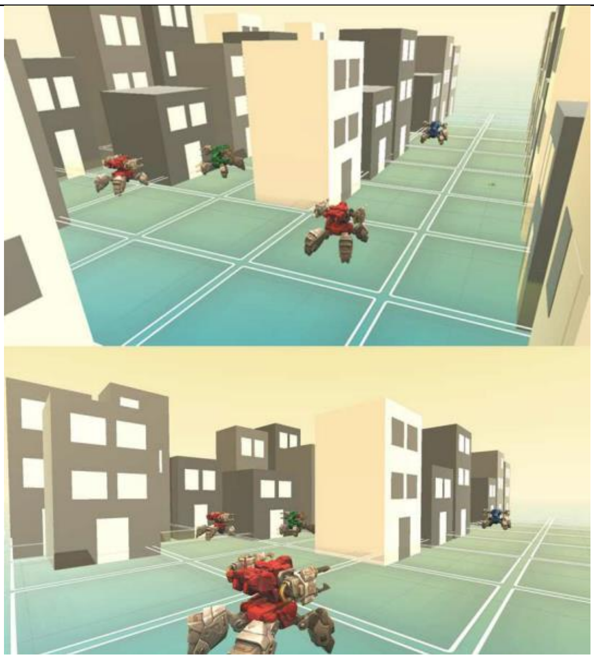
1. X: roof with a single conical point on top.
2. Y: roof with two conical points on top.
3. N: flat roof.

Accessories Shapes

1. N: add a door.
2. W: add a window.
3. S: add a door and stairs.
4. B: add a door and window.
5. A: add a door, window, and stairs.

Rotation and Pin point

1. - : rotate the building to the left.
2. + : rotate the building to the right.
3. P : define the next building position using x, y, and z.

Example of generated building (tested) and their seeds.	
 (c) RNA (d) CNAPCNW (e) CNAPCNWCNWCNW… (f) CNA (g) RNACNAPCNW	

Map Generation

Overview

The creation of battle maps is essential to maintain diversity and excitement in the game, preventing it from becoming monotonous. Producing high-quality content in a short amount of time is challenging, and procedural content generation emerges as an effective solution to this problem. In this study, we aim to apply a system that generates game maps with pacing controllable by the designer using the Wave Function Collapse algorithm. We will examine a pacing aspect that influences player experience, specifically tempo. This system enables us to generate tactical game maps and manage game pacing.

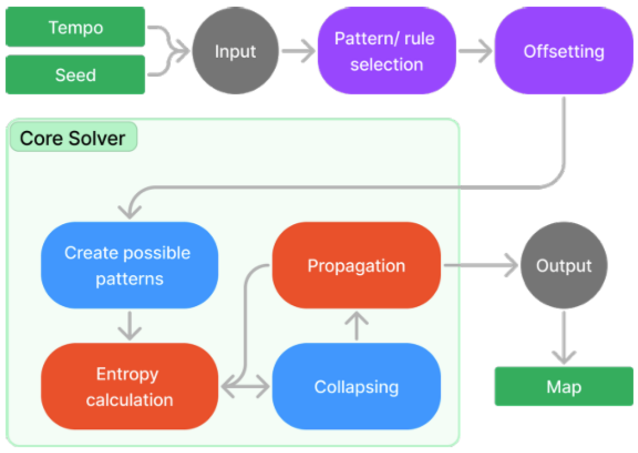
Method

The methodology section details the implementation of an efficient, high-quality solution for automatically generating maps for the turn-based tactics game MECH.AI, specifically by considering the pacing aspect of "tempo" in game design.

The proposed approach involves:

- Implementing the Wave Function Collapse (WFC) algorithm.
- Using map generation data in the pre-battle phase for experiments.
- Conducting tests to measure the model's ability to create maps that align with the tempo set by the game designer.

The expected outcome is a procedural model capable of producing efficient and high-quality maps in MECH.AI.

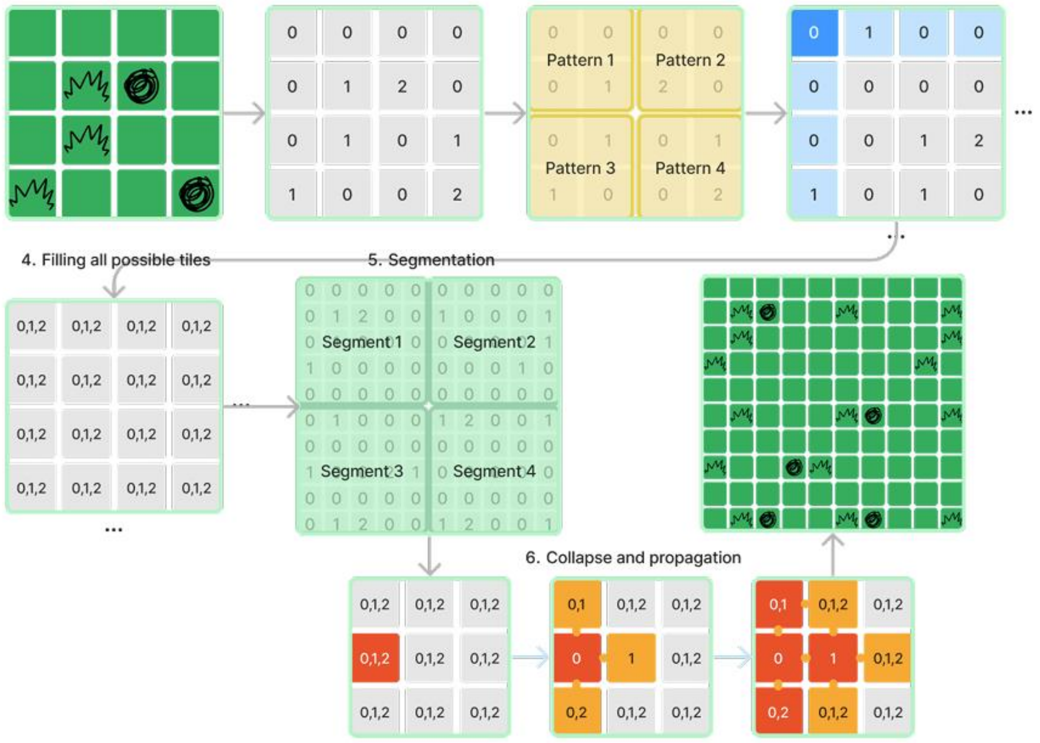


Detail on WFC

Map Generation (Wave Function Collapse - WFC): The game maps are represented by a combination of passable and obstacle tiles in a 10x10 grid and are generated using the WFC algorithm. The WFC system generates maps in a pre-battle state, involving several processes:

- **Seed Patterns & Adjacency Rules**: Patterns are created from combinations of tiles and input seeds. Seed patterns, which are smaller than the output map, define adjacency rules—conditions governing relationships between neighboring elements.
- **Entropy, Weight, Propagation**: Entropy indicates uncertainty in the system and is used to assign weights to tiles, influencing pattern propagation to ensure consistent constraint enforcement across the map.
- **Core Solver**: This is where the main WFC process occurs. It begins by constructing a seed pattern, converting input into a 2D matrix of tile indexes, and identifying patterns using an n x n tile filter. An offset is added to the edges of the seed to ensure possible neighbors. The 10x10 output map grid is prepared, with each tile initially containing all possible types. A starting tile is randomly chosen, and cells undergo a collapsing process to converge to a final tile type. The solver calculates entropy, applies constraint distribution, and collapses tiles, generally prioritizing those with the lowest entropy. This iterative process continues until all cells collapse.
- **Segmentation**: The output map is segmented into four 5x5 tile segments to achieve balanced obstacle distribution while maintaining seamless connections through offsetting.
- **Pacing Implementation**: The final step evaluates whether the generated map conforms to the designer's target tempo. While this study focuses on tempo, other pacing aspects like movement impetus and threat can also be applied. Two game design patterns are

used as tempo factors: deployed obstacles and maze patterns. The deployed obstacles factor analyzes the number of obstacles in deployment areas, assuming more obstacles lead to higher tempo due to increased player actions. Conversely, fewer obstacles result in lower tempo. The maze pattern factor focuses on optimal obstacle placement to increase player movement, thereby raising tempo. Tile weights are identified using a pathfinding algorithm (A∗) to determine the likelihood of tiles being passed through, with heavier weights typically converging in the middle of the map.



Results (tested)