Custom Tower Defence Project – Defenders!

Introduction –

This document outlines the details of the project, including how to play, special features, AI techniques that were implemented, code snippets and relevant diagrams. The purpose of this document is to justify why this project deserves a high distinction outcome for COS30002, Artificial Intelligence for Games.

Gameplay –

The game involves using towers to stop the horde of monsters from reaching the end goal. At the start of the game you are given 1300 credits which increases for every enemy you slay. Credits are used to purchase and upgrade towers, which are important as the levels become progressively difficult.

Controls:

Space – Start next level

1, 2, 3 – Put tower 'In Hand'

Right click – Place, select tower

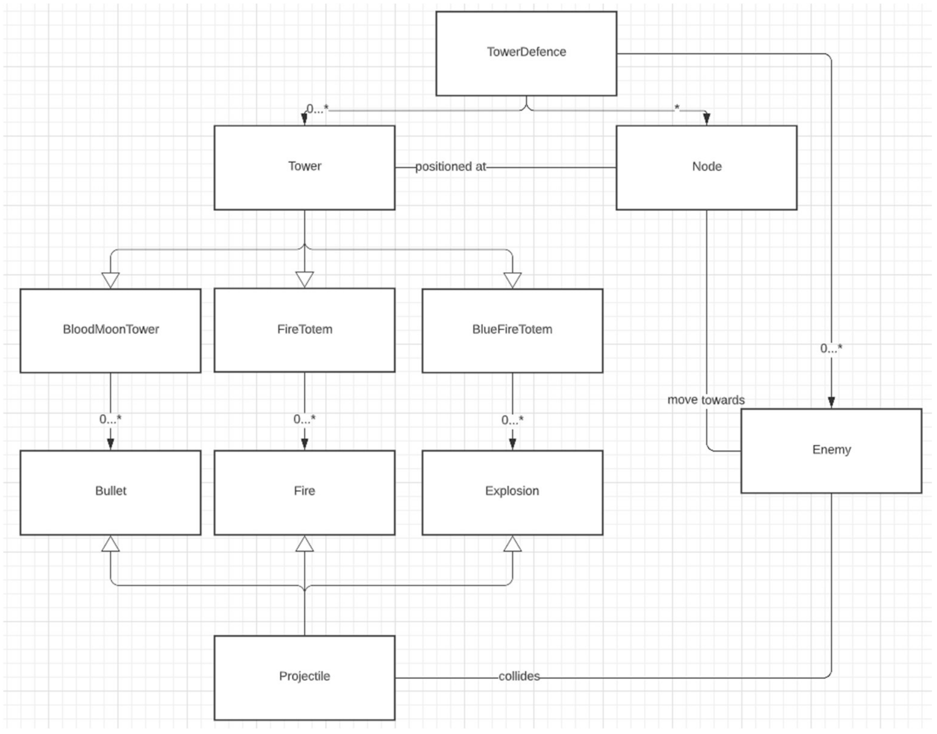
Backspace – Sell selected

u – Upgrade tower

r – Find recommended action

Left click – Clear 'In Hand'

UML Diagram –

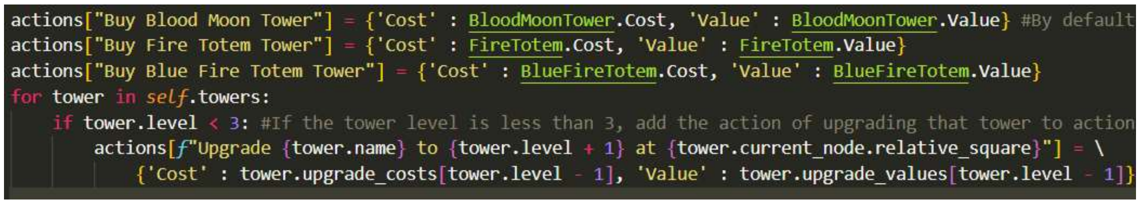


AI Techniques –

This project made use of 4 main AI techniques learnt over the semester; these include goal planning, movement/shooting prediction, path finding/searching and hierarchical state machines.

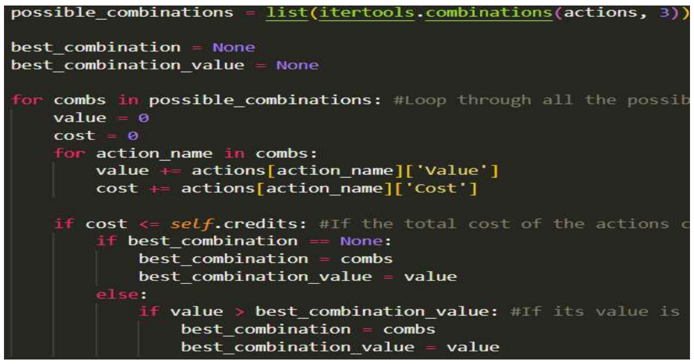
Goal Planning

I used goal planning in accordance with my recommendation system, creating a 3-action plan using all possible actions to be taken at a given time and finding the most value efficient options with the current credits available. The actions available would include the ability to purchase any tower type and upgrade any current tower that can be upgraded.



```python
actions["Buy Blood Moon Tower"] = {'Cost' : BloodMoonTower.Cost, 'Value' : BloodMoonTower.Value} #By default
actions["Buy Fire Totem Tower"] = {'Cost' : FireTotem.Cost, 'Value' : FireTotem.Value}
actions["Buy Blue Fire Totem Tower"] = {'Cost' : BlueFireTotem.Cost, 'Value' : BlueFireTotem.Value}
for tower in self.towers:
    if tower.level < 3: #If the tower level is less than 3, add the action of upgrading that tower to action
        actions[f"Upgrade {tower.name} to {tower.level + 1} at {tower.current_node.relative_square}"] = \
            {'Cost' : tower.upgrade_costs[tower.level - 1], 'Value' : tower.upgrade_values[tower.level - 1]}
```

This shows how actions available are found, the number of actions can vary but will always include at least the 3 (tower purchasing) + upgrades.

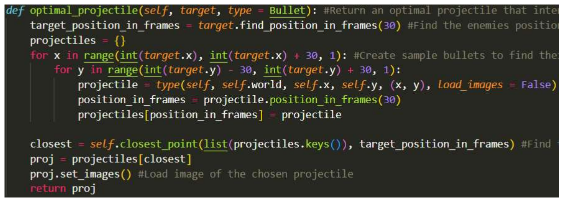


```python
possible_combinations = list(itertools.combinations(actions, 3))

best_combination = None
best_combination_value = None

for combs in possible_combinations: #Loop through all the possib
    value = 0
    cost = 0
    for action_name in combs:
        value += actions[action_name]['Value']
        cost += actions[action_name]['Cost']

    if cost <= self.credits: #If the total cost of the actions c
        if best_combination == None:
            best_combination = combs
            best_combination_value = value
        else:
            if value > best_combination_value: #If its value is
                best_combination = combs
                best_combination_value = value
```

Above is the code showing how the best value combination of actions is found using Itertools' combination function. Only if the total cost of the actions works with current amount of credits, that it can be considered to run for the position of best combination of actions. Each iteration checks if the value is currently better than the 'best' and replaces it if it is, otherwise, it is ignored.

Movement/Shooting Prediction

In order to have towers that lock on to target, I had to ensure they were able to predict where the monsters would be positioned at a particular time and create a bullet that is around that same area at the same time. This creates an illusion that the projectiles are auto locked to a target and always hits an enemy

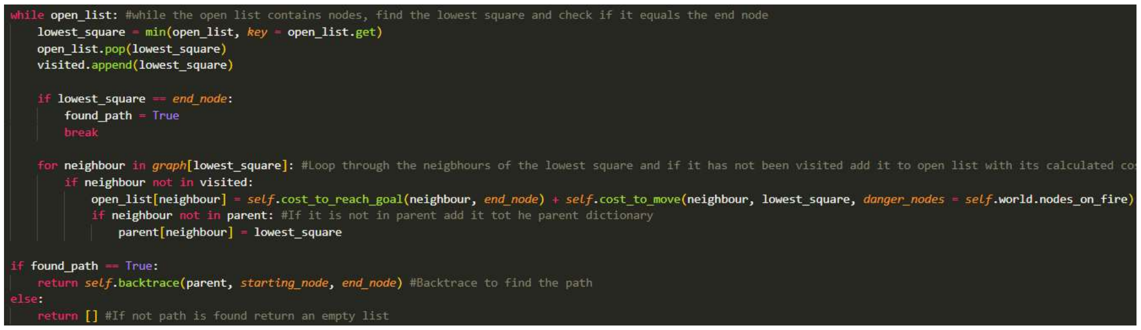


```python
def optimal_projectile(self, target, type = Bullet): #Return an optimal projectile that inte
    target_position_in_frames = target.find_position_in_frames(30) #Find the enemies positio
    projectiles = {}
    for x in range(int(target.x), int(target.x) + 30, 1): #Create sample bullets to find the
        for y in range(int(target.y) - 30, int(target.y) + 30, 1):
            projectile = type(self, self.world, self.x, self.y, (x, y), load_images = False)
            position_in_frames = projectile.position_in_frames(30)
            projectiles[position_in_frames] = projectile

    closest = self.closest_point(list(projectiles.keys()), target_position_in_frames) #Find
    proj = projectiles[closest]
    proj.set_images() #Load image of the chosen projectile
    return proj
```

The function here uses the enemies' find position in frames method to find where the target would be in 30 frames, and iterates, from that targets x position +30 and the targets y position -+ 30 and generates projectiles without drawing them onto the screen. The projectiles position in 30 frames is then found and added to a dictionary. After all the candidate projectiles have been found, the program then which projectiles estimated position is closest enemies estimated position in 30 frames, and both returns and loads the images of that variable. (The function 'closest point' was copied from stack overflow using the NumPy library)

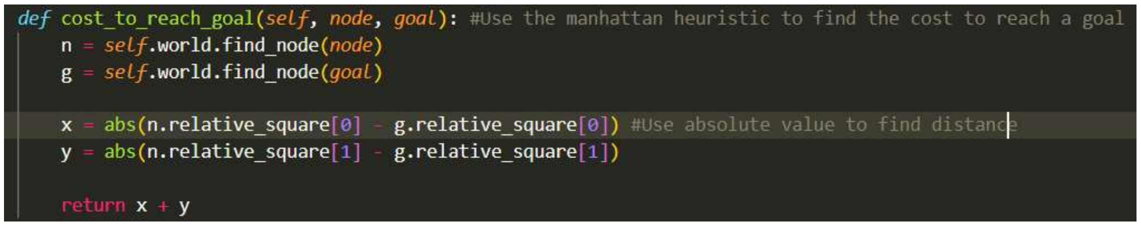
<u>Path Finding/Searching</u>

For the enemies to move from their spawn point to the goal node, they need to be able to find a path to the goal given the option to move left, right, up, and down the path nodes. To do this, I used the A* search approach, a very popular method of searching through a graph. For A* to work, it needs to use 2 different heuristics, a cost to move to a node and the nodes cost to reach goal.



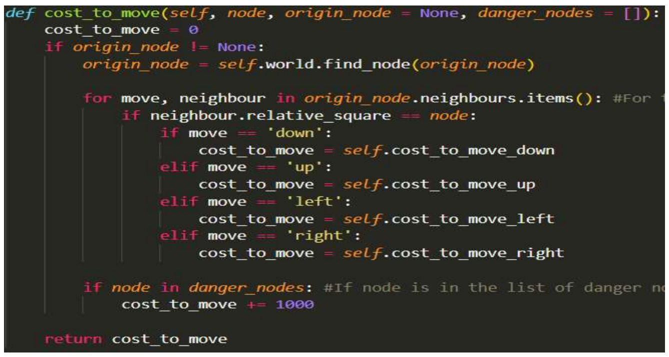
```python
while open_list: #while the open list contains nodes, find the lowest square and check if it equals the end node
    lowest_square = min(open_list, key = open_list.get)
    open_list.pop(lowest_square)
    visited.append(lowest_square)

    if lowest_square == end_node:
        found_path = True
        break

    for neighbour in graph[lowest_square]: #Loop through the neigbhours of the lowest square and if it has not been visited add it to open list with its calculated co
        if neighbour not in visited:
            open_list[neighbour] = self.cost_to_reach_goal(neighbour, end_node) + self.cost_to_move(neighbour, lowest_square, danger_nodes = self.world.nodes_on_fire)
            if neighbour not in parent: #If it is not in parent add it tot he parent dictionary
                parent[neighbour] = lowest_square

if found_path == True:
    return self.backtrace(parent, starting_node, end_node) #Backtrace to find the path
else:
    return [] #If not path is found return an empty list
```

This outlines the main functionality of the A* search function of my project, where it continues to iterate until an the 'open list' (contains unexamined nodes) is empty. It examines nodes one by one, starting with the nodes with the lower 'cost' based on the heuristics. The neighbour of the node is examined and added to the dictionary along with its heuristic cost. Once the node currently being checked matches the end node, the search stops and returns the path by back tracing from the end node to the start node using a dictionary of parents which is also updated in every iteration. In the case of all nodes being examined without any solution, an empty list is given instead, indicating a failed search.



```python
def cost_to_reach_goal(self, node, goal): #Use the manhattan heuristic to find the cost to reach a goal
    n = self.world.find_node(node)
    g = self.world.find_node(goal)

    x = abs(n.relative_square[0] - g.relative_square[0]) #Use absolute value to find distance
    y = abs(n.relative_square[1] - g.relative_square[1])

    return x + y
```

The first heuristic used is the cost to reach goal, this was simply the Manhattan distance, or simply the number of nodes between the node and the goal node (inclusive). To find this, we just used the absolute values after the difference between the x and y values was found and returned the sum of both these values

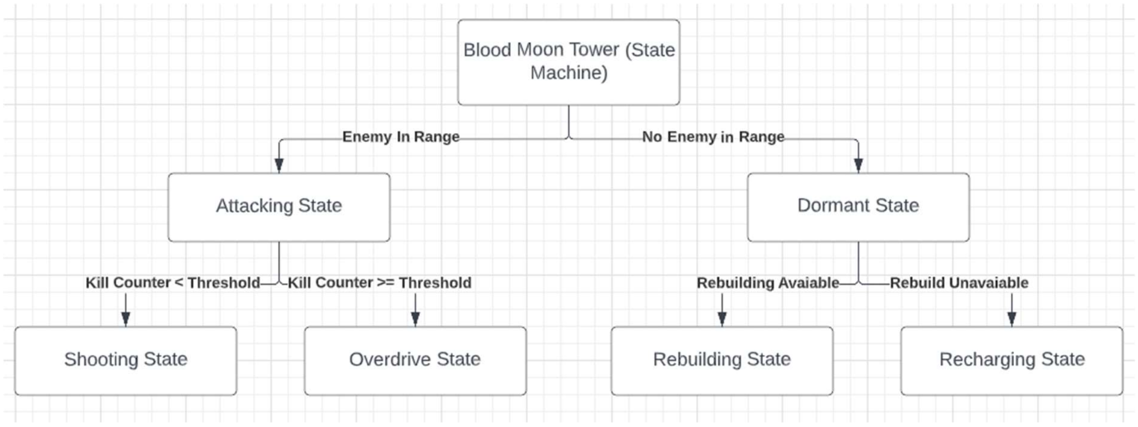


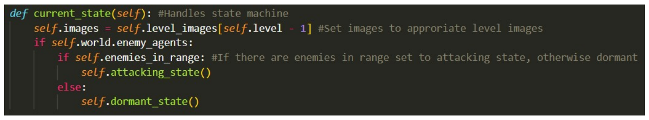
```python
def cost_to_move(self, node, origin_node = None, danger_nodes = []):
    cost_to_move = 0
    if origin_node != None:
        origin_node = self.world.find_node(origin_node)

        for move, neighbour in origin_node.neighbours.items(): #For t
            if neighbour.relative_square == node:
                if move == 'down':
                    cost_to_move = self.cost_to_move_down
                elif move == 'up':
                    cost_to_move = self.cost_to_move_up
                elif move == 'left':
                    cost_to_move = self.cost_to_move_left
                elif move == 'right':
                    cost_to_move = self.cost_to_move_right

        if node in danger_nodes: #If node is in the list of danger no
            cost_to_move += 1000

    return cost_to_move
```

The second, mandatory heuristic, for the A* search I used was the ‘cost to move’, which is just the value to move from a specific node to another specific node. In this case I separated neighbour nodes into categories based on their position on from the origin (left, right, up down) and assigned the cost to move depending on the values of the cost to move up, down etc. (these values were set to a random number from 1 – 20 when the enemy is initialized so that all monsters do not just follow the same path)

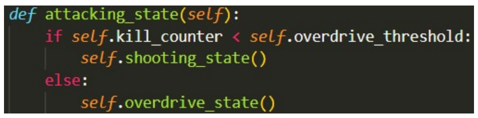
<u>Hierarchical State Machines:</u>

To add a little bit of dynamic elements to the towers, I created a state machine with different states, some of which contain even more specific sub states. All these states work together, but not at the same time, to create a tower that has more complex AI behaviours than a regular static tower with the same attack.

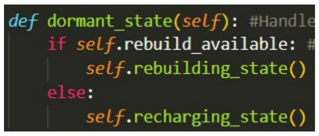




```python
def current_state(self): #Handles state machine
    self.images = self.level_images[self.level - 1] #Set images to approriate level images
    if self.world.enemy_agents:
        if self.enemies_in_range: #If there are enemies in range set to attacking state, otherwise dormant
            self.attacking_state()
        else:
            self.dormant_state()
```

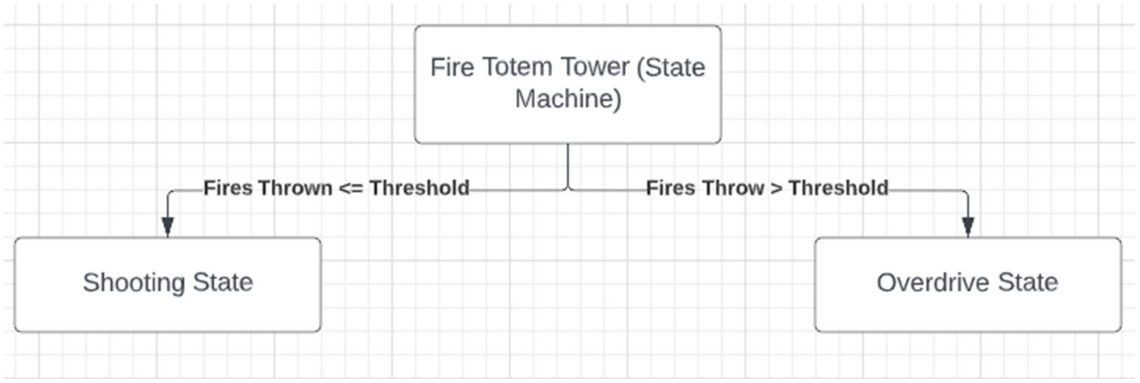


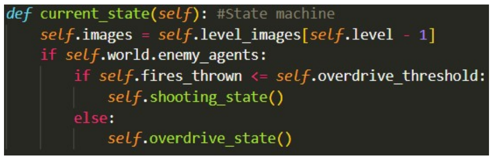
```python
def attacking_state(self):
    if self.kill_counter < self.overdrive_threshold:
        self.shooting_state()
    else:
        self.overdrive_state()
```



```python
def dormant_state(self): #Handle
    if self.rebuild_available: #
        self.rebuilding_state()
    else:
        self.recharging_state()
```

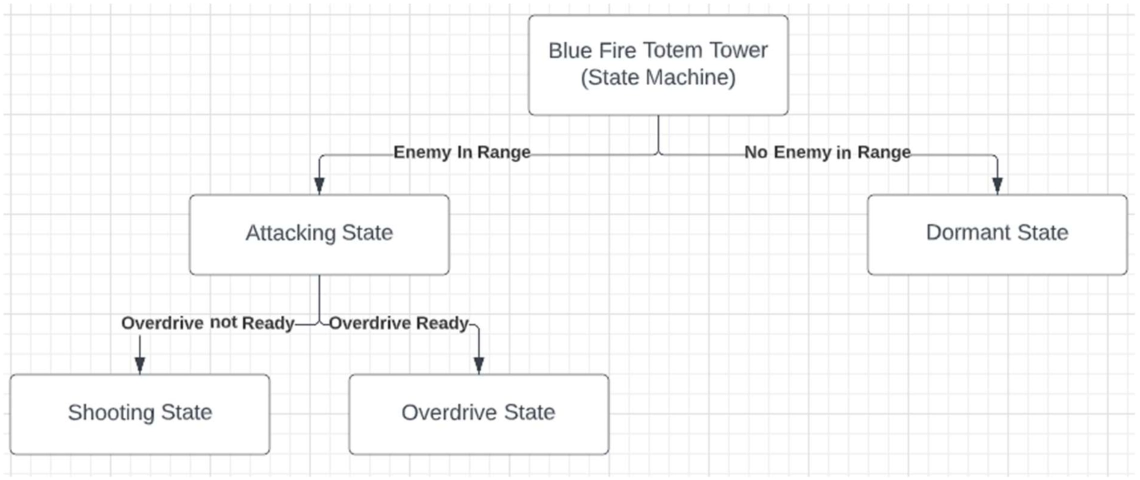
The image above is the diagram for the hierarchical state machine for the first tower, and its representation in code following it. As the diagram suggests, there are two states, each with their own two sub states which is called only if certain conditions are met. In this case the Blood Moon Tower attacks when there are enemies in range, and shoots normally until the kill counter reaches a certain point; that it when it goes overdrive. The opposite state, 'dormant' is activated when there are no enemies in range and calls a rebuilding state or a recharging state depending on if the 'rebuild available' variable is true.

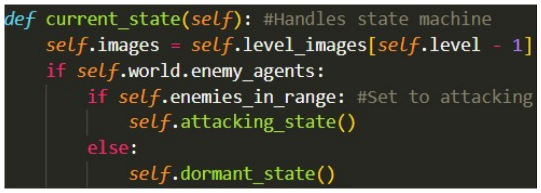


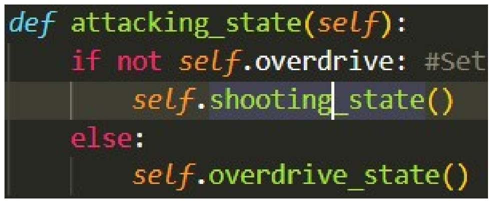


```
def current_state(self): #State machine
    self.images = self.level_images[self.level - 1]
    if self.world.enemy_agents:
        if self.fires_thrown <= self.overdrive_threshold:
            self.shooting_state()
        else:
            self.overdrive_state()
```

The next tower, the fire totem, is much simpler having only two states to change to, changing states depending on whether the current number of fires already thrown exceeds or equals a threshold. The fire totem starts off in its shooting state, but after the condition has been met, goes into overdrive, where the condition eventually resets and starts back at its original state.




```python
def current_state(self): #Handles state machine
    self.images = self.level_images[self.level - 1]
    if self.world.enemy_agents:
        if self.enemies_in_range: #Set to attacking
            self.attacking_state()
        else:
            self.dormant_state()
```


```python
def attacking_state(self):
    if not self.overdrive: #Set
        self.shooting_state()
    else:
        self.overdrive_state()
```

The third and final tower, the Blue Fire Totem, has 2 top level states and two low level states of attacking. Like the first tower, this tower goes to attacking state when an enemy is in range and dormant when there is none. When in the attacking state, it can go from a regular shooting state to an overdrive state which is decided on whether the Boolean variable, 'overdrive' is true or false.

Tower Types + Enemy Types –

Blood Moon Tower: Shoots predictive bullets towards enemy units when in shooting state, in overdrive it shoots bullets in towards all directions. In its rebuilding state, it teleports to a new location where there are the most enemies. In its recharging state, it simply builds up teleport charges. Upgrades increase its range, damage, and rebuild cooldown

Fire Totem: At its shooting state, it throws a fire projectile at a random node within its range, which cause a change in environment when it reached is destination. In its overdrive, it amplifies this by throwing 5 fire projectiles simultaneously. Upgrades increase its duration and lowers its threshold to reach overdrive

Blue Fire Totem: Shoots predictive explosive projectiles towards enemy units when in its shooting state, if the tower is in overdrive, it shoots 10 consecutive, fast fire rate bullets towards the target that are also predictive. When dormant, the tower builds up its overdrive charges till it hits the cap. Upgrades increase the damage, AOE damage, and the explosion radius.

Enemies: All enemy types fall under the same class implementation as they all follow the same rules, with slightly different starting values in health, speed, credits worth and lives worth (image shows weakest to strongest monsters starting from Bug). All enemies follow a path of nodes which is initialized in creation, but have the ability to change their generate path if the dynamic environment undergoes changes (in particular, nodes that are on 'fire'). They all use the same search algorithm in A* but can vary in heuristic costs.

Notable In Game Features:

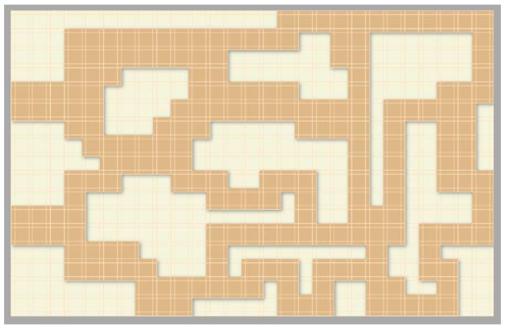
AI Related Features

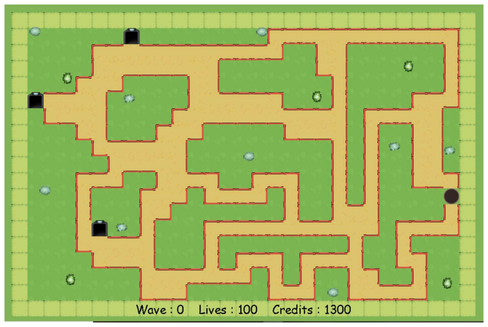
- Recommendation system
- Action finding
- Goal planning
- Enemy path finding
- Dynamic environment
- Enemy path changing
- Predictive projectiles
- Predictive movement
- Single layered state machine
- Double layered state machine

General Features

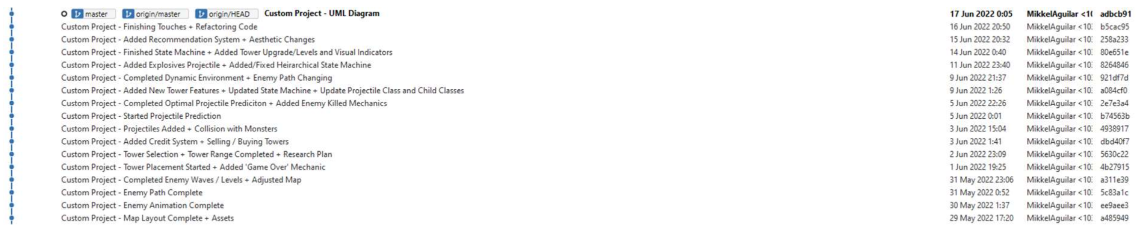
- Purchase/Buying towers
- Selling towers
- Upgrading towers
- Recommendation system
- Tower range indicators
- Credit system
- Lives / Game over indicators
- Enemy spawning queues
- Tower 'In Hand' mechanics
- Enemy and tower sprite animation
- Changing state indicator

Changes in map design





Bitbucket Commit History:



master origin/master origin/HEAD **Custom Project - UML Diagram**	**17 Jun 2022 0:05**	**MikkelAguilar <1(**	**adbcb91**
Custom Project - Finishing Touches + Refactoring Code	16 Jun 2022 20:50	MikkelAguilar <10:	b5cac95
Custom Project - Added Recommendation System + Aesthetic Changes	15 Jun 2022 20:32	MikkelAguilar <10:	258a233
Custom Project - Finished State Machine + Added Tower Upgrade/Levels and Visual Indicators	14 Jun 2022 0:40	MikkelAguilar <10:	80e651e
Custom Project - Added Explosives Projectile + Added/Fixed Heirarchical State Machine	11 Jun 2022 23:40	MikkelAguilar <10:	8264846
Custom Project - Completed Dynamic Environment + Enemy Path Changing	9 Jun 2022 21:37	MikkelAguilar <10:	921df7d
Custom Project - Added New Tower Features + Updated State Machine + Update Projectile Class and Child Classes	9 Jun 2022 1:26	MikkelAguilar <10:	a084cf0
Custom Project - Completed Optimal Projectile Prediciton + Added Enemy Killed Mechanics	5 Jun 2022 22:26	MikkelAguilar <10:	2e7e3a4
Custom Project - Started Projectile Prediction	5 Jun 2022 0:01	MikkelAguilar <10:	b74563b
Custom Project - Projectiles Added + Collision with Monsters	3 Jun 2022 15:04	MikkelAguilar <10:	4938917
Custom Project - Added Credit System + Selling / Buying Towers	3 Jun 2022 1:41	MikkelAguilar <10:	dbd40f7
Custom Project - Tower Selection + Tower Range Completed + Research Plan	2 Jun 2022 23:09	MikkelAguilar <10:	5630c22
Custom Project - Tower Placement Started + Added 'Game Over' Mechanic	1 Jun 2022 19:25	MikkelAguilar <10:	4b27915
Custom Project - Completed Enemy Waves / Levels + Adjusted Map	31 May 2022 23:06	MikkelAguilar <10:	a311e39
Custom Project - Enemy Path Complete	31 May 2022 0:52	MikkelAguilar <10:	5c83a1c
Custom Project - Enemy Animation Complete	30 May 2022 1:37	MikkelAguilar <10:	ee9aee3
Custom Project - Map Layout Complete + Assets	29 May 2022 17:20	MikkelAguilar <10:	a485949

This project took approximately 3 weeks to complete, although there were many breaks in between. Many or all of the simple, easy features to implemented were completed within the first week of working. The second week saw a slowdown in work rate (due to other workloads catching up). The third week consisted of the completion of the harder AI features such like the shooting and the enemy path changing.