Programming with C/C++

Academic year: 2025-2026, Fall semester

Programming assignment

The Castle Defender

Managing memory and behaviour

Student name: Djayco Coret

Student number: 2108836/u868875

1 Overview

This project had several requirements: have a functional grid; have functional towers; have functional enemies; have accurate scoring and endgame; have adaptive, in difficulty and spawning, artificial intelligence; all while writing clean, modular, OOP-based code.

The requirements all have been fullfiled, yet code quality suffers from being created by a neurodivergent person, which might make it more difficult for neurotypicals to grasp.

A simple, yet effective AI algorithm was implemented. After each death, when reaching the castle or being eliminated by a player tower, the AI will update the probability of selecting that column. This will lead to the AI selecting those columns which succesfully have attacked the castle, while avoiding those in which agents get eliminated.

Difficulties were found when an agent has to interact with another agent. Since the agent is an element of the grid, and not the other way around, there was no trivial way for agents to interact. This problem was solved by creating a simple "move" structure, which contains all information such that the game and grid can execute the move. Another problem came in the form of recursive header calls. This problem was solved by generalising classes using templates.

I worked on this project frequently, without clear schedule; just letting adhd do its thing.

2 Tasks and objectives

Inherentence was chosen because polymorphism in the templated grid object. Write your introduction here.

2.1 Battlefield

The battlefield is realised by a vector, using row-major indexing, of length $row*cols$ where the elements may point to an object in memory. Row-major indexing allows for the elements to be stored in a continuous fashion, which is more cache friendly (probably not noticable at this scale). The use of templates was necessary for modularity and recusive header calls (more on this in the challenges section). The vector stores memory adresses of the elements on the grid, where is stores a nullptr iff the position is vacant.

```
template <typename T>
class Grid {
    std::vector<T*> grid;
    int rows, cols;
    ...

    Grid(int rows, int cols) : rows(rows), cols(cols) {
        size = rows * cols;
        grid.resize(size, nullptr);
    }

    T* const operator()(int row, int col) const {
        return grid[row * cols + col];
    }

    T*& operator()(int row, int col) {
        return grid[row * cols + col];
    }

    ...
```

The game updates by iterating through all objects which belong to both the ai and the player.

2.2 Towers and attacks

Towers scan the grid using a nested for loop. Towers have access to the grid, via a pointer. Every turn all towers scan the places surrounding them (see below). If an enemy is detected, the move structure get updated. The tower only gets to shoot once, since, while the

```
void return_move(Move* move) override {
    for (int i = std::max(0, row - range); i <= std::min(
       info->rows - 1, row + range); i++) {
        for (int j = std::max(0, col - range); j <= std::
           min(info->cols - 1, col + range); j++) {
            if ( std::sqrt( pow(i - row, 2) + pow(j - col
               , 2))  <= range) {
                if (grid->operator()(i,j) != nullptr && (
                   grid->operator()(i,j)->get_id() == 'e
                   ' || grid->operator()(i,j)->get_id()
                   == 'E') ) {
                        move->row_new = i;
                        move->col_new = j;
                        move->row = row;
                        move->col = col;
                        move->shoot_bool = true;
                        }}}}}
```

2.3 Enemy movement

2.4 Scoring and endgame

Scoring and endgame gets handled by the info structure.

```cpp
struct Info {
    int rows, cols;
    int wave, health;
    int enemies_health = 3;
    int enemies_placed = 0;
    int enemies_placed_total = 0;
    int score, highscore;
    bool game_over = false;
};
```

When an enemy has health smaller than 0, it increases the score attribute by ten.

The boolean value for game over gets updates with the game. If the health attribute of the info structure is smaller or equal to zero, game over gets set to true. Additionally, in the next wave method game over gets set to true, if the current wave is bigger than the amount waves. When game over is true, a while loop breaks, ending the game.

2.5 Enemy behaviour

Enemies only have the ability to move forward, or diagonal in some instances. Enemy spawning is more complex, and involves updating values based on deaths.

Enemies can only move forward to the castle row. The enemy will only move forward if the spot is vacant, if the spot is occupied it will try to move diagonally, in either forward direction. If all legal directions are occupied, the enemy will be removed from the board. Some enemy types have a speed bigger than one, as such they will not check the spot(s) immediatly in front of them. These enemies can be considered arial units, making it a feature, not a bug.

The spawning of enemies is more complex. Firstly, the AI has a vector of size amount columns, these get initialises to one real number. The AI also has an attribute called width column adjust, more on this later. Secondly, when an agent gets removed from the grid, the owner of the agent updates it's death. For the AI class it updates the values in the vector (see equation 1).

$$x_{i,t} = x_{i,t-1} + \frac{R - D}{|i - c_d| + 1} * \alpha \tag{1}$$

Where i is the index of the column which gets updated, r_c is the boolean value which is true iff the agent has reached the castle, d if the boolean value which is true iff the agent was eliminated by a tower, c_d is the index of the column where the agent has died, and α can be seen as analogous to a learning rate. The top part of the fraction will make it such that it substract when being killed, and adding when reaching the castle. Thirdly, an element wise operation gets performed on the vector (see equation 2).

$$p_i = \frac{e^{x_{i,t}}}{\sum_{x \in \mathcal{X}}^{N} e^{x_{j},t}} \tag{2}$$

Given how the columns are somewhat similar as the output of the last hidden layer of a neural network, chose was made for a softmax. This creates a probability distribution. Finally, the AI draws from this distribution.

$$T : \mathbf{x_t} \mapsto \mathbf{p} \tag{3}$$

It should be noted that every time the AI places an enemy, it copies and transforms the vector with raw data into a probability distribution. Ideally equation 1 would update the probability distribution, but I did not know how.

3 Challenges

Multiple challanges were found and solved. Details are discussed below. 1. Recursive header calls were solved using templates; 2. Interaction problem between agents was solved by introducing a *move* structure; 3. Third item

3.1 Solved

Recursive header calls were solved by using templates. Earlier versions of the project required all headers to include all headers (i.e., the grid had to include the grid object, and vice versa). This was initially solved by having subclasses in seperate header files, but this made the program harder to comprehend. To generalise the code, instead of having to rely on the grid object header, templates were used. Which also made much of the code reusable for other objects.

3.2 Partially solved

Interaction problem between age

3.3 Not solved

Write the content for the subsection here.

4 Discussion and future work

The AI having a different learning rate for each difficult is a interesting concept in my opinion. As far as I am aware, most difficulty adjustments change the available resources of the AI. While this is an interesting concept, static tower placement limit how intersting this AI really is. To further improve the learning of the AI, one will have to make the possibility to remove or move towers. Furthermore, the AI is limited in unlearning things.

The towers use a nested loop, which keeps going after a legal move has been found, even overwriting previous moves. On one hand, this makes sure the enemy which is latest in the grid (closer to the castle) gets shot at. On the other hand, the is computationally inefficient since it does not return after a move was found. Additionally, this is not a intelligent decision maker by any means. Stronger enemies (those with a higher speed, in this case), do not get prioritised. Future work should focus on making the towers intelligent agents.