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

Generative artificial intelligence technology statement

I used Anthropic's Claude to write a LaTeX template based on the project report template, but all C/C++ code logic and implementation are my own.

1 Overview

In this project a castle defender has to be developed. A castle defender game is a game where the user has to defend a castle from enemies. The user has the ability to place towers which will shoot and eliminate enemies. The game will end if the castle has ran out of health or all enemy waves have been spawned and cleared.

Most requirements that have been set out in the project description have been fullfiled. Code quality is far from optimal. Initially the code was developed as a multi agent system engine, yet the requirements mention turn based movements, instead of distributed movements.

A simple, yet effective artificial intelligence was implemented. The artificial intelligence draws from a probability distribution containing all legal moves, which is a transformed vector containing real valued numbers. After an enemy dies, the artificial intelligence updates the vector such that it will avoid the column where it was killed by a tower.

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

The grid was implemented using a template that can hold any datatype. A grid object was made, which has virtual methods, that can be used as a superclass for all object which are placed on the grid. The tower, enemy, and castle inherit from the grid object. By initialising the grid using the grid object datatype, runtime polymorphism can be used, such that all objects derived from the grid object can be placed on the grid. The grid objects will select a move or action, which the game object executes (e.g., killing an enemy or moving forward). A structure was implented for simpler interaction between parts of the program. An owner class was created as a baseclass for both the AI and player. The owner has access to all grid objects owned by that owner.

2.1 Battlefield

The battlefield is realised by a vector from the standard library, using row-major indexing, of length $row * cols$ where the elements may point to a grid 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.

The grid updates via the game header, more in this in the game section.

```
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];
    }
```

2.2 Grid object

A grid object was used as a base class for other objects. Runtime polymorphism is used such that the object's methods being held by both the grid and owner can be overriden for specific behaviour.

2.2.1 Tower and castle

Towers scan the grid using a nested for loop. Towers have access to the grid, via a pointer. If an enemy is detected, the move structure get updated. The tower only gets to shoot once, since, while the

2.2.2 Enemy

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.

2.3 Movement and other actions

2.4 Owner

The owner object is the base class for both the player and artificial intelligence. The owner has a vector containing pointers to grid objects. Additional information, such as the dimensions of the grid.

It should be noted that both the grid and an owner point to the same object. More on this in challenges.

2.4.1 Player

The player adds a upgrade tower method.

2.4.2 Artificial intelligence

Placing The artificial intelligence spawns enemies by drawing from a probability distribution. Everytime a enemy gets spawned, the artificial intelligence transforms the vector containing logits into a probability distribution (see equation 1). These logit get initialised to 1, such that one starts with a uniform probability distribution.

$$T : \mathbf{x_t} \mapsto \mathbf{p} \tag{1}$$

Every element corresponds to a unique column, where x_i refers to the i-th column. To transform the logit vector to a probability vector, an elementwise softmax is applied (see equation 2.

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

The softmax allows for the summed values to be 1, and all values to be between 0 and 1. The updating process, allows for the logits to become negative. Exponentiating each element, thus, results in positive values.

Learning The logit vector gets updated using a formula that prioritises learning close to the location of death (see equation 3. The formula set out to do the following: decrease the value when an enemy dies because of enemy fire, increase the value when an eemy dies because it has reached the castle, and update the value of neighbouring columns in a similar, yet scaled fashion.

$$x_{i,t} = x_{i,t-1} + \frac{R - D}{|i - c_d| + 1} * \alpha \tag{3}$$

Where R is the boolean value if the enemy has reached the castle, D is the boolean value if the enemy has died because of enemy fire, i is the index of the column which is being updated, c_d is the index of the column where the enemy has died, $\alpha is the learning rate, and \mathrm{x}_i$ is the logit value corrosponding to column i.

Above the division it $R - D$, returns -1 when an enemy was killed by a tower; and return 1 when it has reached the castle. Resulting in The top part of the fraction will make it such that it substract when being killed, and adding when reaching the castle.

2.5 Scoring and endgame

Scoring and endgame gets handled by the info structure.

```
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.

2.6 Enemy behaviour

3 Game

The game header functions as an executive system, looping through all grid object being owned by an owner.

4 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. Pointers and manual memory

4.1 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.

4.2 Partially solved

Interaction problem between age

5 Discussion and future work

The AI having a different learning rate for each difficult is a interesting concept in my opinion. Ideally equation 3 would update the probability distribution, but I did not know how. 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.