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, D.A.S.R. Coret, 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 game had to be developed. In such a game the user has to defend a castle from enemies approaching said castle. The user has the ability to place towers which will shoot and eliminate enemies. The artificial intelligence has the ability to place enemies in the top rows, yet will have to decide in which column. 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 fulfilled. 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. This is a result of me beginning to code before fully reading the project assignment; which did teach me a valuable lesson.

A simple, yet effective artificial intelligence was implemented. The artificial intelligence learns by updating a probability distribution containing all legal moves, based on the deletion of enemies; this takes advantage of the constraint that both enemies that have reached the castle and those that got eliminated by towers will have to be deleted from the grid.

Difficulties were found when a grid object has to interact with another grid object (e.g., when shooting and updating health). Since the grid object is an element of the grid, and not the other way around, there was no trivial way for grid objects 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. Additionally, the use of templates helped solve the interaction problem.

I worked on this project frequently, without clear schedule.

2 Tasks and objectives

The grid was implemented using a template, such that it can hold any object. Pointers to the template object were stored, since the movement of pointers is computationally more efficient than whole objects, and more pointers to the same object were needed. A grid object with virtual methods was made, which was used as a superclass for all objects 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. Virtual methods were overriden to give grid objects unique behaviour, which also uses runtime polymorphism. 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 implemented for simpler interaction between parts of the program (see 2.3). 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, resulting in multiple pointers directed to the same grid objects, in the grid and in the owner's grid objects. Manual memory is allocated during the placement of objects, while it is deallocated either by the removal of an object or by the grid deconstructor.

2.1 Battlefield

The battlefield was 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 noticeable at this scale). The use of templates was necessary for modularity and fixing recursive header calls error (more on this in 4). The vector stores pointers of the elements on the grid, where is stores a nullptr iff the position is vacant. Each position in the vector corresponds with a unique patch.

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

```
1  template <typename T>
2  class Grid {
3      std::vector<T*> grid;
4      int rows, cols;
5      ...
6
7      Grid(int rows, int cols) : rows(rows), cols(cols) {
8          size = rows * cols;
9          grid.resize(size, nullptr);
10     }
11
12     T* const operator()(int row, int col) const {
13         return grid[row * cols + col];
14     }
```

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. The grid object receives a pointer to the grid, such that it can make observations on the grid (e.g., if object is on specific patch).

2.3 Movement and other actions

Movement is delegated by the move structure, this serves as a middle layer between the grid objects and the game/grid. The grid object has a 'return_move()' method, that changes a move structure to the preferred move by the grid object. It changes, instead of returning, since it returns a boolean value if it can make a legal move.

See below for an example, see full code for full implementation.

```
struct Move {
    int row;
    int col;
    int row_new;
    int col_new;

    bool remove_obj = false;
    bool movement_bool = false;
    bool reached_castle = false;
};
```

2.3.1 Tower and castle

Towers detect and attack enemies by iterating through all patches of the grid that might be in range: Suppose a tower with range r. This tower will start at (min(0, x-r), min(0, y-r) and end at (max(cols-1, x+r), max(cols-1, y+r)). The euclidean distance for all patches get calculated to verify if the patch is truly in range.

The move structure gets updated if the tower spots an enemy, the new row is considered the patch to target.

2.3.2 Enemy

Enemies only have the ability to move forward, or diagonal when forward path is obstructed. Enemy spawning is more complex and is discussed in 2.4.2.

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 [1]. 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) immediately in front of them. These enemies should be considered arial units, making it a feature, not a bug.

[1] In theory the code could be adapted such that the enemy could draw for the probability distribution, but in reality these enemies rarely survive a collision with a tower, since they will be close and have less health than the tower has range

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 (player or ai) point to the same object.

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. Every time an enemy gets spawned, the artificial intelligence transforms the vector containing logits into a probability distribution (see equation 1). These logits get initialised to 1, such that one starts with a uniform probability distribution.

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

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 element-wise softmax is applied (see equation 2.

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

The softmax allows for the summed values to be 1, and all values to be between 0 and 1. The to be described 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 enemy 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, $\alpha is the learning rate, and x_i$ is the logit value corresponding to column i.

Above the division, $R - D$, returns -1 when an enemy is killed by a tower; and return 1 when it reaches the castle. Resulting in the behaviour earlier described.

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.

3 Game

The game header functions as an executive system, looping through all grid object being owned by an owner. The game object has a tick method, that iterates through all object owned by an owner. The tick method initialises a move structure, passes this to the grid_object.return_move(), and executes this adjusted move. The execute move method is given below.

```
void execute_move(Move& move, Owner<GO>* owner) {
    if (move.remove_obj) {
        GO* ptr = grid(move.row, move.col);
        grid.delete_object(move.row, move.col);
        owner->update_death(move);
        owner->operator()(move.index_owner) = nullptr;
        delete ptr;
    } else if (move.movement_bool) {
        move.moved_bool = grid.move_object(move);
    } else if (move.shoot_bool) {
        grid(move.row_new, move.col_new)->take_dmg(grid(
            move.row, move.col)->get_power());
```

```
12      }
13  }
```

4 Challenges

Multiple challenges were found and solved. Details are discussed in full below. 1. Recursive header calls were solved using templates; 2. Interaction problem between grid objects was solved by introducing a *move* structure;

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 separate 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 grid objects and their objects (enemy, tower, etc) was solved by implementing a *move* structure. In earlier versions of the program, the grid objects had no way to access the grid, to access other grid objects, to shoot or check if spot is empty. By implementing the *move* structure as a middle layer, this interaction problem was solved.

5 Discussion and future work

The AI having a different learning rate for each difficulty 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 interesting 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 grid objects.