UNIVERSITY OF MANCHESTER

OBJECT-ORIENTED PROGRAMMING IN C++ PROJECT REPORT

Stratagem: A Turn-Based, Grid-Based, Strategy Game for the Windows Console

Word count: 2489

Project Code and Report Author:
Angus PLUMMER

PHYS30762 OOP in C++ Project

The University of Manchester
School of Physics and Astronomy

April 22, 2018

Abstract

A project was undertaken to create a strategy game for the Windows console in the vein of classic Game Boy Advance strategy games. This involved making a game map constructed of several types of tiles, distinct types of units that exist on and move across these tiles as well as interact with each other by attacking. The user sees and interacts with this information through a graphical user interface achieved using the Windows API. User input is received through mouse clicks in the game window. The project was further expanded to include a title screen, map selection, team size selection, and unit placement.

Contents

1 Introduction

Inspired by some of the great strategy games on the Game Boy Advance, particularly Fire Emblem and Advance Wars (see figures 1 and 2), I decided to make a turn-based grid-based strategy game with the same core components: a player has a selection of units on a game map which they can select and take actions with such as move or attack; players take it in turns to act with their units; a players turn ends when they have acted with all their units; victory is typically achieved when all enemy units have been defeated. I proposed to take this format and make a simple 2 player 1v1 battle game with a small selection of unit and tile types.

A simple path-finding algorithm was implemented to determine the tiles that units can reach from their current position and the paths they take to reach these tiles.

User interaction is achieved through a graphical user interface using the Windows API, which allowed for greater manipulation of the console output and enabled mouse input, both being essential for an intuitive user experience. Actual user input is achieved through clicking on units, tiles, or buttons. Output is all through coloured text output to the console through various levels of abstraction via rendering functions in classes. The project was expanded beyond the gameplay and associated UI to also include a pre-game menu system that allows for customisation of the game setup as well as a title and help screen.

Figure 1: Fire Emblem: The Blazing Blade. A tactical role-playing game developed by Intelligent Systems and published by Nintendo for the Game Boy Advance. Released 2003.

Figure 2: Advance Wars 2: Black Hole Rising. A turn-based tactics video game developed by Intelligent Systems and published by Nintendo for the Game Boy Advance. Released 2003

2 Code Design and Implementation

This section is by class structure, with general descriptions of the purpose of each class and how the important and least straightforward features of the class were implemented. A diagram of the class structure is shown in figure 3.

Functions are indicated by `()` after the function name for clarity but function arguments are not written.

2.1 Structs

2.1.1 Coord

The `Coord` struct is used throughout the code for map and console cell coordinates. It simplifies handling 2D coordinates by pairing together two integer values. The struct implements a multitude of operator overloads for arithmetic and comparison operations.

2.1.2 Colour Scheme

The `ColourScheme` struct contains two `ConsoleColour` enums to define a background and text colour combination. The Windows console has 16 colours defined by number from 0-15 listed and named by the `ConsoleColour` enum. The Window class enables getting and setting the console colour scheme using this struct.

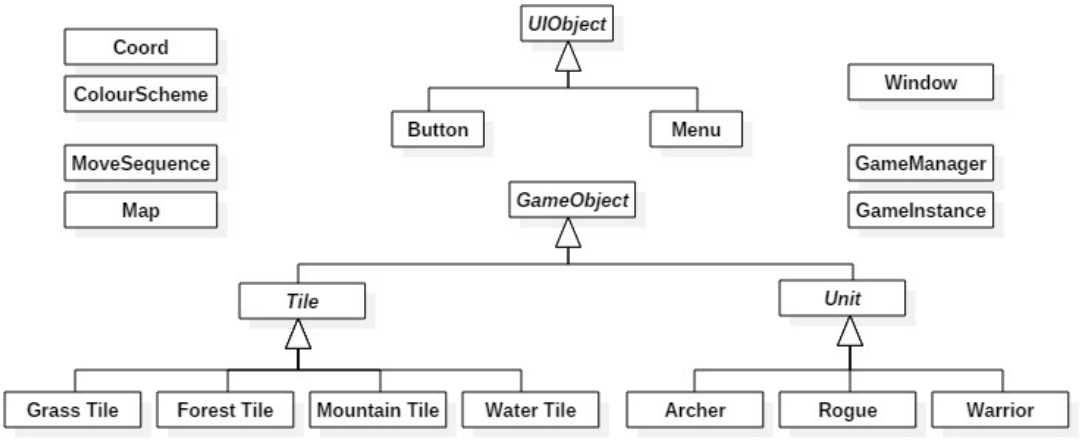

Figure 3: Class structure diagram for Stratagem. The arrows indicate the class derives from the pointed to class. Italics indicate a class is abstract.

2.2 The Game Window

The `Window` class enables manipulation of the console window and console buffer, and mouse interaction within the window through the Windows API using `Windows.h`.

The constructor sets and locks the window dimensions, enables mouse interaction, sets the console buffer size to be the same as the window size to remove the scrollbars, and initiates the input, output, and window handle member variables.

The most important functions in the class are, firstly, `GoTo()` which sets the position of the console cursor to an input `Coord`. This function enables updating console cells individually as they need updating. Output to the console limits the speed of the program so updating only the areas of the screen that need changing provides an enormous performance boost over a complete refresh.

Secondly, `MouseClickPosition()` which returns the location of a mouse click. The function reads the input record of events that have been sent to the game window using the Windows API. If a mouse-down is detected in the input buffer, its position in the window is recorded and the function repeatedly checks if the mouse is still held down. If it is, the position is updated. If it is not, the function returns the position. The function thus only returns when the mouse has both been pushed down and released and a whole click has been completed.

2.3 Game Objects

The `GameObject` class is an abstract base class for all entities that exist on the game map, namely units and tiles.

Both units and tiles have a `Clone()` function which returns a `unique_ptr` to a copy of the object. This is to simplify the copy constructors and assignment operators of `Map` and `GameInstance`. `CloneHelper()` is declared for each derived class and returns a raw pointer to a copy of the derived object. Together, these allow for iterating through a container of base class objects and producing copies of the correct derived class type using the overridden functions.

2.3.1 Tiles

The `Tile` class is an abstract base class for the different tile types. The game map is constructed from tiles and the units occupy tiles on the map. Tiles have properties which influence the units ability to attack and move. These properties are stored as integer member variables and are different for each tile type as to provide interesting gameplay.

`Render()` outputs the tile to the console by printing its tile marker char to a rectangular region defined by its location on the map and the properties of its parent map.

2.3.2 Units

The `Unit` class has many stats as integer member variables which affect the gameplay of the unit. The derived unit types have different values for these stats that provide them different strengths and weaknesses for strategic gameplay.

The `UnitState` enum lists the four independent states a unit can be in. The state of the unit

affects, for example, whether the unit can be selected, if it can act, and how it is rendered though a `switch` statement.

`AttackableUnits()` returns a `vector` of pointers to attackable units, determined by the position of the units on the map and the attack range of the attacking unit. This is used in `CanAttack()` and `HighlightAttackableUnits()`. `Attack` and `AttackedBy()` handle the damage calculations when attacking.

A unit can only move and attack once per turn. The `moved_this_turn_` and `attacked_this_turn_` Booleans keep track of this.

`Kill()` is called when `set_current_hp()` attempts to set the unit's `current_hp_` equal to or below zero. This sets the unit to `STATE_DEAD`, runs a death animation, then removes the unit from the game.

The pure virtual function `CanTraverse()` is overridden in the derived unit classes and determines whether a unit can enter a given terrain type. It takes an input terrain tile pointer and returns false if a dynamic cast to the unit's forbidden tile types is successful.

`ReachableTiles()` returns a `vector` of tiles which are reachable by the unit. `MoveTo()` moves the unit to a reachable tile and animates its movement to the tile through multiple calls to `AnimateMovement()`. Both of these functions utilise path-finding which is detailed in section 2.5.

2.4 The Game Map

The `Map` class has a 2D `vector` of `unique_ptrs` to `Tile` objects. Unique pointers are used to indicate that a `Map` has ownership of the tiles and to simplify the clean-up of the heap. `Map` also has a `vector` of raw pointers to the units on the map, which indicates access rather than ownership. The other member variables define the dimensions of the map and rendering information.

Maps are loaded using `LoadMap()` which takes a 2D `vector` of integers and creates a 2D `vector` of `unique_ptrs` to `Tiles`. The maps are stored in `raw_maps.h`.

`Map` serves primarily to give access to tiles and units at given coordinates, add and remove units from the map, and render the map entirely or at a specific coordinate.

2.5 Path-finding

The A* search algorithm was implemented for determining the tiles a unit can reach in its movement action and a legal path to reach these tiles. The `MoveSequence` class used in the path-finding functions contains a pointer to the tile on the map that it corresponds to, a pointer to its parent `MoveSequence` object in the sequence, the movement cost to reach the tile from the starting tile, and a heuristic cost to reach the target tile calculated as the Manhattan distance [1].

The A* algorithm is implemented fully in `Unit::MoveTo()` and a variation on the algorithm is used in `Unit::ReachableTiles()` (as there is no target tile). Details of the algorithm itself can be found readily online [2]. For this implementation, `lists` of `MoveSequence` objects were used as the open and closed sets because the `MoveSequence` objects contain a pointer to their parent in the sequence and items in `lists` maintain their memory address as the container changes in size, ensuring these pointers stay valid.

2.6 UI Objects

`UIObject` is an abstract base class for interactable objects that exist off the game map. The member variables determine where and how the object is displayed in the game window. `Contains()` checks whether a console cell coordinate is within the region occupied by the `UIObject`.

2.6.1 Buttons

`Button` is a class which contains a function object member that is defined when the object is initiated. The constructor can take a lambda function as an argument and `Trigger()` will run it if the button is enabled.

2.6.2 Menus

The `Menu` class is an optional container for `Button` objects which maintains a collection of buttons as a vertical list. `HandleMouseClick()` triggers any button in the menu that contains the location of the click. There are also functions for rendering, adding buttons, and clearing all buttons.

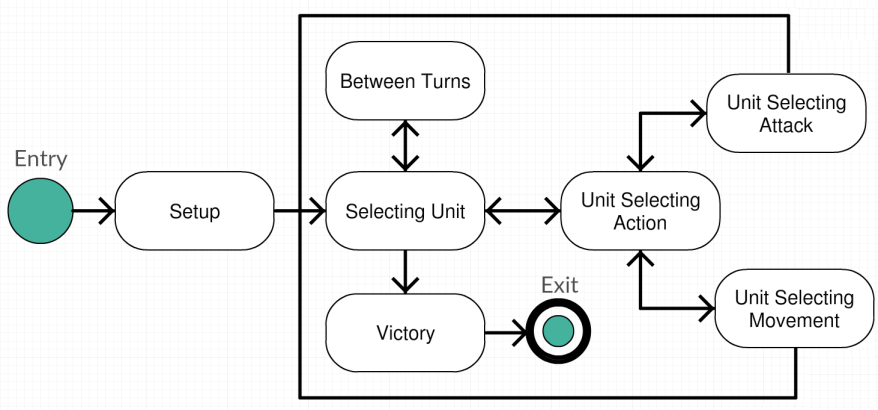

Figure 4: Flow chart of the independent finite states of the `GameInstance` class.

2.7 Game Instance

The `GameInstance` class handles the actual gameplay. It is designed as a finite state machine with states labelled by the `GameState` enum. The state determines what is presented to the user and what they can do. A flow chart for the states is shown in figure 4.

`GameInstance` contains `unique_ptrs` to the `Map` and all the `Units` of the game to indicate ownership of these objects. Units can be added and removed using `AddUnit()` and `RemoveUnit()` and the map is loaded and rendered as a whole by `LoadMap()` and `RenderMap()`.

`Run()` runs the main gameplay loop. This consists of obtaining the location of any mouse click and running `HandleMouseClick()` for that location and then automatically ending the turn if all units on the current team have finished acting. `HandleMouseClick()` uses a `switch` statement to change its behaviour based on the `GameState` of the `GameInstance`. For instance, when in `STATE_SELECTING_UNIT` the user can only select a unit or trigger the surrender or end turn buttons to the right of the map. Whereas when in `STATE_UNIT_SELECTING_MOVEMENT`, the user can only select a tile for the `selected_unit_` to move to. If the player does not select a valid tile then `UnChooseMovement()` will run, bringing the `GameInstance` back to `STATE_UNIT_SELECTING_ACTION`.

The class has functions for entering and leaving the different states, displaying menus and buttons, configuring units at the start and end of turns, handling victory, and selecting and deselecting the `selected_unit_`.

The context menu is a `Menu` that is displayed when in `STATE_UNIT_SELECTING_ACTION`. A list of the `selected_unit_`'s possible actions is presented to the user and when clicked on will run the relevant state entry function for that action.

`Victory()` is triggered when `RemoveUnit()` removes the last unit from one of the teams. The game then waits for the user to click anywhere at which point the user will then be taken back the title screen.

2.8 Game Manager

The `GameManager` handles the menu system for the game setup and contains a `GameInstance` object which is run from within `PlayGame()`. The class functions similarly to the `GameInstance` class with a finite set of states determining what is displayed to the user and what they can do. A flow chart of the states is shown in figure 5.

Navigation through the states is achieved through buttons which trigger the entry functions for a state. The entry functions display the relevant information to the user and set the state so they can interact correctly with the mouse.

The class has a static `GameManager` member which is accessible through the static function `Game()`. This provides global access to a single shared instance of `GameManager` through which the `GameInstance` and `Window` are also accessible. This simplifies the code by providing access to these objects without having to pass around references to them in many functions.

3 Results

The game functions as intended. A typical scenario of the game is shown in figure 6. The game has been play-tested with about a dozen people and the

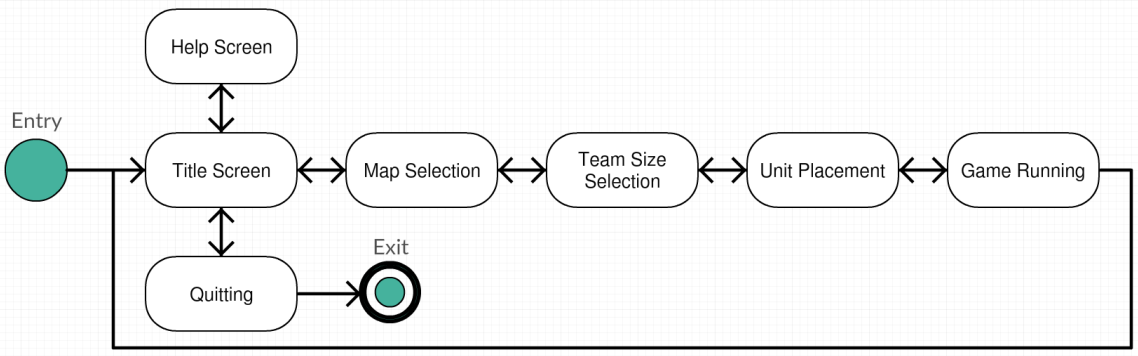

Figure 5: Flow chart of the independent finite states of the `GameManager` class.

feedback was positive: the UI is intuitive and the gameplay is generally enjoyable although, without quick access the unit and tile stats, strategic play was difficult. Many of them tried to break the game and find bugs but were unsuccessful.

As this is a game, the results are much better experienced than explained so I recommend playing a game or two to see for yourself!

4 Discussion and Conclusions

While the project was overall a success there are many avenues for further improvement. In terms of the code itself, a base class `Object` from which `GameObjects` and `UIObjects` derive could reduce repetition and simplify rendering of `GameObjects` by storing the console coordinate of the objects as is done by `UIObjects`.

The state system could be refined to have actual state classes with their own entry, exit and mouse click handler functions instead of using enums, asserts, and switch statements.

The raw maps, tile types, unit types, ascii art, and help text could be made readable from file. They could then be changed without the need for recompiling which would become useful if the code increased in size.

Currently the code is highly coupled but this is difficult to change without significant restructuring.

Figure 6: Screen-shot of a typical scene during a game of Stratagem.

Nonetheless it could be simplified by using the Component pattern [3]. There are also functions in the `Unit` and `Tile` classes that could be moved to the `Map` class, such as the path-finding.

External graphics libraries could be used which would alleviate the limitations of using the console.

In terms of game features, play-testing showed that users wanted helpful information to be readily accessible rather than just on the help screen. Allowing a user to right click on a `GameObject` or `UIObject` and be given useful information about it would help new players. Something like this would undoubtedly be necessary if a wider range of unit and tile types were added.

To add more complexity and strategy to combat, a status system could be added so that units could be, for example, poisoned or blinded and this would apply a sustained effect to the unit. Units could also be given abilities to use alongside their normal attacks. For example, a whirlwind attack for the warrior which damages all adjacent enemies.

Realistically if this project were taken further it would be sensible to recreate it in a game engine or a graphics library where there are the tools and resources to simplify much of the code.

5 References

[1] Black, P.E., "Manhattan Distance", `https://xlinux.nist.gov/dads/HTML/manhattanDistance.html`, 2006. Accessed on 22/04/2018.

[2] "A* Search Algorithm", `https://en.wikipedia.org/wiki/A*_search_algorithm`, 2018. Accessed on 22/04/2018.

[3] Nystrom, R., "Component", `http://gameprogrammingpatterns.com/component.html`. Accessed on 22/04/2018.