GO - REPORT

[BY JAMAINE CHRISTIAN JAC105]

AUGUST 8, 2024

INTRODUCTION

This report presents the design and development of a turn-based strategy game, GO. The game is played on a 9x9 grid, where two players take turns placing stones on intersections, aiming to capture their opponent's stones by occupying all liberties while creating territory on the board. The game ends when both players pass consecutively, with the winner determined by the area controlled on the board.

The project emphasizes the application of key software engineering principles, such as modularity and encapsulation, to create a well-structured and maintainable codebase. The game's architecture was designed to ensure flexibility and robustness, with a focus on clear class responsibilities and interactions.

This report will detail the principle components of the design, interaction between classes, and key algorithms implemented in the game. The success of this project will be evaluated in a conclusion at the end.

SOFTWARE DESIGN

PRINCIPLE COMPONENTS OF THE DESIGN

The game is designed with several classes that are used for controlling different aspects of the game. Here are the classes below:

- GridManager: Creates the 9x9 grid tiles (spec 2.1).
- SceneLoader: Used for switching between scenes – home screen to the main game and vice versa.
- MainMenu: Manages the user interface controlling the home screen (spec 2.5).
- PauseMenu: Manages the user interface controlling the pause menu (spec 2.5).
- PlayerTurn: Controls player turn and updates the user interface for the current players score (spec 2.2).
- GameLogic: Controls the end game condition and the user interface display when finished (spec 2.3 & 2.4 & 2.5).
- SearchAlg: Uses different functions for searching the grid for different rules of the game (spec 2.1 & 2.4).
- StonePlacer: This is the class the handles all the stone interactions with the board such as placement, removal and highlighting (spec 2.1 & 2.2 & 2.5).

Each of these classes are crucial for ensuring that the game operates according to the rules and specification.

INTERACTION BETWEEN CLASSES

<u>Start Screen</u>

The game starts at the home screen where you can read the instructions, begin the game or quit the application, this part of the game utilises button interactions through MainMenu to start or quit the game.

During Gameplay

When play has been selected, GridManager will draw the grid tiles on start up. The 'pass turn' button uses the PlayerTurn class to switch between players and if this happens twice consecutively then GameLogic will cause the end of the game. The pause icon uses PauseMenu to control the pause panel, this will not allow any updates in StonePlacer to stop stones from being placed when the pause menu is activated. When the game is not paused, StonePlacer is updating the check where the mouse position is, if a tile is clicked it will use SearchAlg to check all the rules in the game to see if the stone can be placed. If the stone is placed, PlayerTurn will cause the turn to switch between the players which causes StonePlacer to initialise a different colour stone onto the board when it is placed and changes what conditions are being checked within SearchAlg.

Game Over

GameLogic will cause the game over panel to pop up. It uses PlayerTurn and SearchAlg to calculate the number of stones on the board, territory owned and the amount of captures the winner has gotten, to calculate the final score, which is displayed on the game over panel.

CLASS BREAKDOWN

GridManager

This class uses a serializable game object of the grid tile prefab and a serializable integer for setting the size of the grid. On the start function, a nested for loop which loops to one less than the size of the grid and creates a new game object of the grid tile, makes sure it sorts on top of the background and then positions it. The reason for looping one less than the size of the grid is because we will be placing stones on the intersections and not on the tiles themselves therefore this will create a 9x9 grid of intersections.

SceneLoader

This class has two functions which are both used on button presses, these are:

- GetActiveScene() –sets a Scene object as the current scene and returns this Scene object.
- LoadScene(int sceneNumber) -   it will use PlayerTurn.resetTurn() to reset to default score and make sure black player goes first. Then to load the intended scene SceneManager.LoadSceneAsync(sceneNumber) is used

MainMenu

This class is intended for controlling the interactions of two buttons on the home screen and has two functions to correspond to two of the buttons on the user interface, these are:

- Play() – uses SceneLoader.LoadScene() to load the game.
- Quit() – uses Application.Quit() to quit the application.

PauseMenu

This class is intended for controlling a Boolean value, isPaused, to control the state of the game. It has two serializable game objects, one for the pause menu panel and one for the pass button to handle what happens when the game is paused. This class four functions:

- Pause() – turns off the pass button, turns on the pause panel and sets isPaused to true.
- Resume() – closes the pause panel, turns on the pass button and sets isPaused to false.
- Restart() – uses SceneLoader to load the active scene, essentially reloading the scene you are on.
- Quit() – uses SceneLoader to load the home scene and then sets isPaused to false.

PlayerTurn

This class stores all the information about a player and controls the turns. A bool isBlackTurn controls the player turn, two arrays of two integers control the player score and their number of captures , a Vector2Int protected Location is used to ensure an old board configuration cannot occur and integer numberOfPasses to track the number of consecutive passes . This class consists of three functions:

- PassTurn() – switches the bool isBlackTurn and then updates the score using GameLogic.SetScore().
- IncreasePassCount() – increases numberOfPasses by one.
- ResetTurns() - uses a for loop up to two to reset PlayerTurn.playerScore[i] and playerTurn.playerCaptures[i] for both players.

On every update this class will update two serializable text variables turnTracker and scoreTracker to display whose turn it is and what their score is.

GameLogic

This class controls the end of the game and what is displayed. It has three TextMeshProUGUI objects: winnerTitle, finalScore and endScoreBreakdown, which are updated within the update fucntion. This class has seven functions:

- GetWinner() – creates integer and sets it to whoever has the highest score using PlayerTurn.PlayerScore[].
- GetFinalScore(int winner) – creates integer finalScore which uses PlayerTurn to get playerScore[winner]. This then returns finalScore.
- GetAmountOfCaptures(int winner) – creates integer amountOfCaptures and uses PlayerTurn.playerCaptures[winner] to set this integer. It then returns amountOfCaptures.
- CalculateTerritory(int winner) – creates an integer amountOfTerritory and sets this to zero. Then uses GridManager.size as the upper bounds of both conditions in a nested for loop starting at zero. In this loop, SearchAlg.ownedTerritory[i,j] is used to check if that tile is owned by the winner, if it is then the amountOfTerritory integer will increase by one. This function will return amountOfTerritory.
- CalculateStones(int winner) – creates an integer amountOfStones and sets this to zero. Then uses GridManager.size as the upper bounds of both conditions in a nested for loop starting at zero. In this loop, SearchAlg.placedStone[i,j] is used to check if that tile has the winners colour stone placed there, if it is then the amountOfStones integer will increase by one. This function will return amountOfStones.
- IsGameOver() – uses PlayerTurn.number of passes to check if two consecutive passes has occurred then it will return true else it will return false

- SetScore() – creates two integers, blackScore and whiteScore, these are set as the sum of CalculateStones(colour), CalculateTerritory(colour) and GetAmountOfCaptures(colour). PlayerTurn.playerScore[] is set the the corresponding score integers.

SearchAlg

This class is used for all search algorithms needed for implementing the rules of GO! It has two 2d-arrays of integers of the size of GridManager.size for both bounds to store the stones that are placed on the board, placedStone, and who the tile is intersection is owned by, ownedTerrirtory. It also has a list of Vector2Int, emptyPoints to store where all the empty intersections on the board are. This class has ten functions:

- SetListOfEmptys() – clears the empty list and the checks all placedStone values to check where the emptys are and add this vector to the list.
- GetAdjacents(int x, int y) – takes in a position, creates a new Vector2Int list and adds the vector of one in each direction, up down left and right to the list if it is within bounds and then returns it.
- GetAdjacentEmpties(int x, int y) – the same as GetAdjacents(int x , int y) but also checks diagonals
- GetConnectedStone(int x, int y) – uses breadth first search and adds this position to a queue and marks it as checked. The target colour, int colour, is set by using placedStone[x, y] . While the queue has elements in it, it will deque the first item (first in, first out) and add this item to the list of connected stones. If it is a stone it will use GetAdjacents else it will use GetAdjacentEmpties to get adjacent positions and then if it's not checked and the same colour add to queue and mark as checked. It will then return the connected stones if they are not more than the max link otherwise it will return an empty list.
- CheckSurroundingStones(List<Vector2Int> connection) – creates a HashSet<int> surroundingColours and then for every point in the connection it will get the adjacents and if the placedStone here is a stone add this colour to the HastSet. Then if surrounding colours equals one it will return the colour as an integer else it will return 0.
- SetTerritory() – runs SetListOfEmpty() then for every empty point not already checked get the connected stones, check the surrounding intersections using CheckSurroundingStones(connection) and it will set an integer variable colour to this. If the colour is not zero (i.e., there is a stone placed), for every point of the connection set ownedTerritory of that position to the colour.
- HasLiberties(int x, int y) – gets the connected stpnes and for every stone in the connection it will get the adjacents using GetAdjacent(stone.x,stone.y) and then for every adjacent get the colour of the stone, if there is no colour, it is a liberty return true, else if none of these have a liberty return false.
- GetLiberties(int x, int y) – used for highlighting tiles later, if the stone/connection HasLiberties(x, y), get connected stones and for every connected stone, check the adjacents using GetAdjacent(x, y) and if the adjacent is empty then add this to the new list, liberties. The function then returns the liberties.
- CapturablePieces(int x, int y) – checks all adjacents if the adjacent is opposite colour to stone placed check if it HasLiberties() and if it does not then get connected stones and these to the list of capturable pieces. It will then return capturablePieces.
- CanPlaceStone(int x, int y) – if stone already return false; if owned territory return false; if it id protected location return false (stop repeated board config). Temporarily place the stone, use CapturablePieces to get check for captures. If it can capture then remove the stone and

return true else check if it has liberties once placed, then remove the stone then return true if it has liberties else return false.

<u>StonePlacer</u>

This class is used for placing the stones on the board and highlighting things such as connections, liberties and territory. It uses dictionaries of Vector2Int positions as the key and then GameObjects as the value for the stonesOnBoard and then each of the highlight types so they can later be destroyed.

It has one helper function:

- GetStoneAtPosition(int x, int y) - if the stone exists at that position return the stone game object

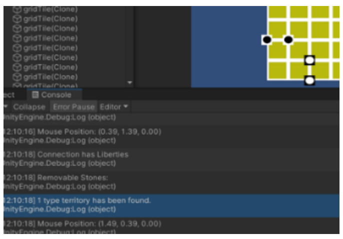
The majority of the class occurs in the update function and pulls all the other classes together to make the game playable.

In the start function, the whole grid is set to no territory and nothing being placed.

In the update function, it wont update if PauseMenue.isPause or GameLogic.IsGameOver equal true. It will destroy all previous stone/territory/liberty highlights and clear the dictionaries. It then gets the coordinates of the mouse position, it will figure out what intersection is the closest the the mouse, if the mouse is within index of the grid when the mouse is pressed it will create a stone of the correct colour and place it making sure it goes to the intersection and not the the middle of the gird tile. The stone is then added to the dictionary to be able to be destroyed on capturing and ending the game. It will then use SearchAlg to find connected stones, liberties and territory and highlight then when the mouse is hovering over a stone.

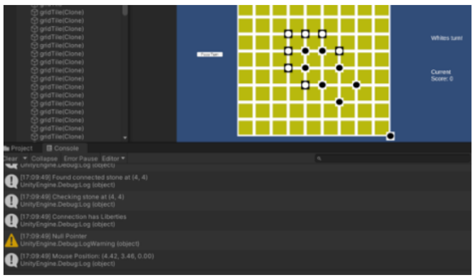
TESTING

To test the game, I used debug statements to log what was going on and also visual identification of errors by playing the game. Here are some screenshots of errors I encountered along the way, most were due to logical errors that were identified through the debugs:



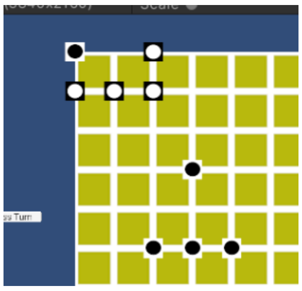
```
nityEngine.Debug:Log (object)
2:10:16] Mouse Position: (0.39, 1.39, 0.00)
nityEngine.Debug:Log (object)
2:10:18] Connection has Liberties
nityEngine.Debug:Log (object)
2:10:18] Removable Stones:
nityEngine.Debug:Log (object)
2:10:18] 1 type territory has been found.
nityEngine.Debug:Log (object)
2:10:18] Mouse Position: (1.49, 0.39, 0.00)
```

1. False territory.

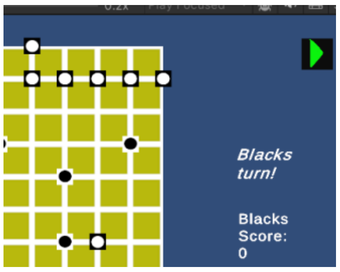


```
[17:09:49] Found connected stone at (4, 4)
UnityEngine.Debug:Log (object)
[17:09:49] Checking stone at (4, 4)
UnityEngine.Debug:Log (object)
[17:09:49] Connection has Liberties
UnityEngine.Debug:Log (object)
[17:09:49] Null Pointer
UnityEngine.Debug:LogWarning (object)
[17:09:49] Mouse Position: (4.42, 3.46, 0.00)
UnityEngine.Debug:Log (object)
```

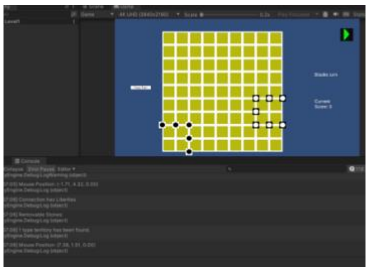
2. Not letting me capture case with no liberties



3. Territory not marked and illegal play

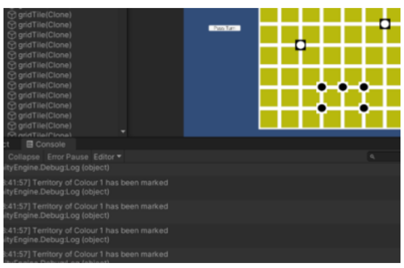


4. Score not working.



```
yEngine.Debug:LogWarning (object)
7:03] Mouse Position: (-1.71, 4.32, 0.00)
yEngine.Debug:Log (object)
7:04] Connection has Liberties
yEngine.Debug:Log (object)
7:04] Removable Stones
yEngine.Debug:Log (object)
7:04] 1 type territory has been found.
yEngine.Debug:Log (object)
7:04] Mouse Position: (7.38, 1.51, 0.00)
yEngine.Debug:Log (object)
```

5. Only one territory found.



```
ityEngine.Debug:Log (object)
:41:57] Territory of Colour 1 has been marked
ityEngine.Debug:Log (object)
:41:57] Territory of Colour 1 has been marked
ityEngine.Debug:Log (object)
:41:57] Territory of Colour 1 has been marked
ityEngine.Debug:Log (object)
:41:57] Territory of Colour 1 has been marked
```

6. False territory

CONCLUSION

This project is good as it accomplished all the requirements, as well as separating the different tasks and implementing encapsulation. This approach allows for easier identification of problems as each responsibility is handled by its own class. This project has also been made to be adaptable to boards of greater size. The use of static methods allowed for this project to have faster run times by not needing to waste time creating new objects of the different classes to be able to access them.

This project has its weakness in its difficulty to adapt this project due to the use of static methods and variables which heavily limit their uses. I also struggled updating a public TextMeshProUGUI that was not static. However, my resolve was to use static methods to return a normal value that could be used to update these texts. I also have a bug on the WebGL version but not the unity build and I'm not sure if this has anything to do with my use of static methods and why it is causing issues when it is built for the web.

In conclusion, this project was a good attempt at making a GO, using static methods to keep code readable with faster run times with a bit of adaptability for the board sizes, however, if I were to make this again, `I would not use static methods and variables and instead stick to the standard OOP principle of Inheritance.

STATEMENT OF TOOLS

Unity Editor – 2022.3.13f1
Visual Studio Code:
with Unity v1.02 and c# v2.39.29 and c# Dev Kit v1.9.55