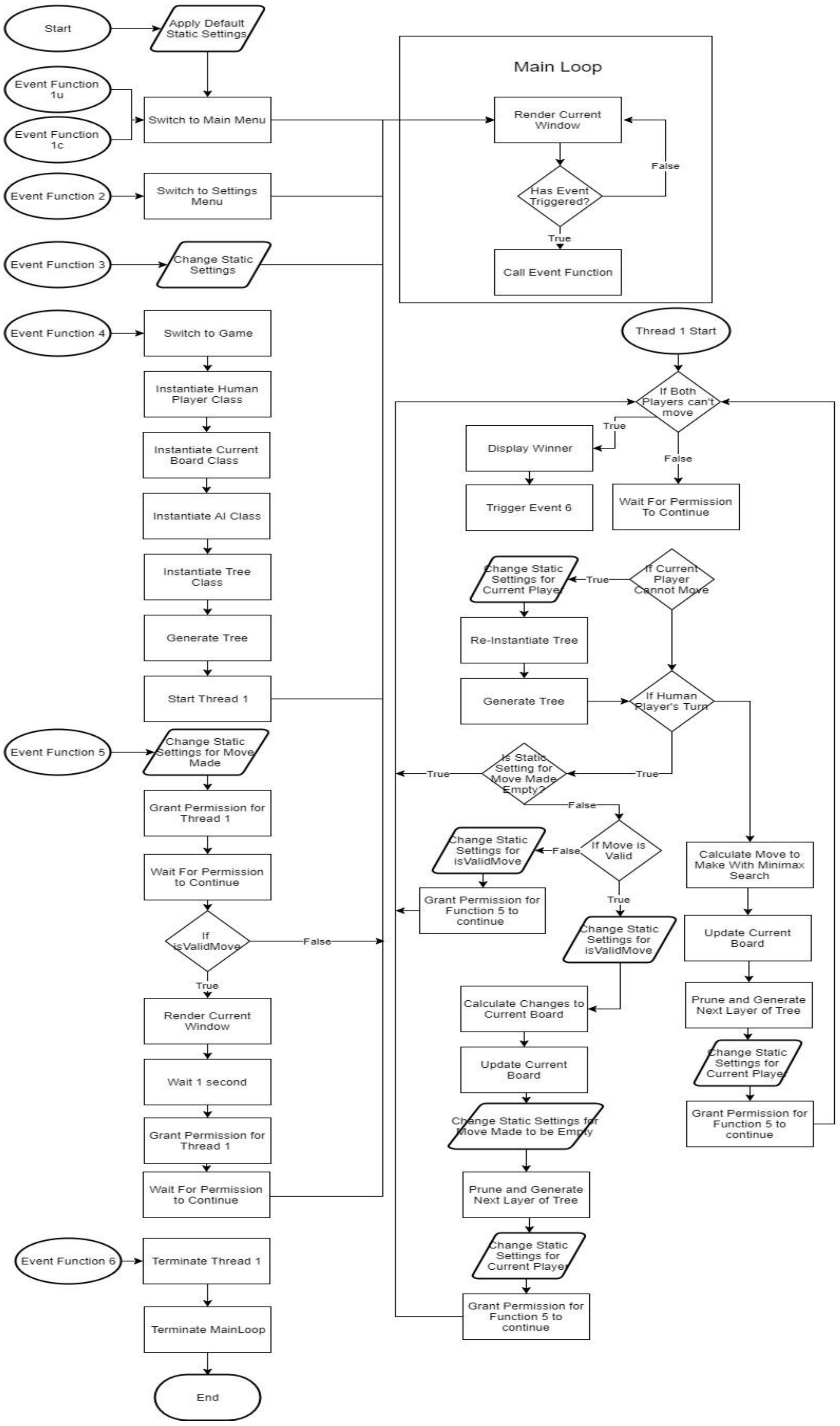
System Design

Project Technology

For this project I am going to utilise the python programing language (version 3.6.3) and will be using the PyCharm integrated development environment to help support the final implementation of the project.

High level overview


Start
Apply Default Static Settings
Event Function 1u
Event Function 1c
Switch to Main Menu
Event Function 2
Switch to Settings Menu
Event Function 3
Change Static Settings
Event Function 4
Switch to Game
Instantiate Human Player Class
Instantiate Current Board Class
Instantiate AI Class
Instantiate Tree Class
Generate Tree
Start Thread 1
Event Function 5
Change Static Settings for Move Made
Grant Permission for Thread 1
Wait For Permission to Continue
If isValidMove
False
True
Render Current Window
Wait 1 second
Grant Permission for Thread 1
Wait For Permission to Continue
Event Function 6
Terminate Thread 1
Terminate MainLoop
End
Main Loop
Render Current Window
Has Event Triggered?
False
True
Call Event Function
Thread 1 Start
If Both Players can't move
True
Display Winner
Trigger Event 6
False
Wait For Permission To Continue
If Current Player Cannot Move
True
Change Static Settings for Current Player
Re-Instantiate Tree
Generate Tree
If Human Player's Turn
True
Is Static Setting for Move Made Empty?
True
False
If Move is Valid
False
Change Static Settings for isValidMove
Grant Permission for Function 5 to continue
True
Change Static Settings for isValidMove
Calculate Changes to Current Board
Update Current Board
Change Static Settings for Move Made to be Empty
Prune and Generate Next Layer of Tree
Change Static Settings for Current Player
Grant Permission for Function 5 to continue
Calculate Move to Make With Minimax Search
Update Current Board
Prune and Generate Next Layer of Tree
Change Static Settings for Current Player
Grant Permission for Function 5 to continue

To allow for the support of a graphical interface I am designing the program to be event driven. To do this I will be using a main loop that will be responsible for refreshing the current window but to also listen for various events that trigger throughout the course of the programs execution. The following events can be triggered by the system:

Event Function 1u

This event is triggered by the system after the program has been initialised and begins, when this occurs the interface will be switched to the main menu window.

Event Function 6

This event is triggered by the system when both players cannot move on any given turn thus fulfilling the win condition for Othello. When triggered thread 1 will be terminated and then the main loop will be ended and the program will exit completely.

The following events can be triggered by the user:

Event Function 1c

This event can be triggered by the user when they interact with the interface when it is set to the settings menu window. This allows them to transition back to the main menu.

Event Function 2

This event triggers when the user interacts with the main menu in order to access the settings menu. When activated, the current window is terminated and the current window is set to the settings menu.

Event Function 3

This event triggers when the user interacts with the settings menu in order to alter the basic settings of the game, these include: the dimensions of the game board, the colour of the player and the overall game difficulty. When activated the static class that holds the settings in memory will be updated to reflect the user's choice.

Event Function 4

This event triggers when the user interacts with the main menu so that the game may begin. When this event is triggered, the main menu window will be terminated and replaced with the game window. After this, all of the central classes that govern the game itself are instantiated, these include: the human player class, the AI class, the current board class and the tree class (Each of these will be further explained when I cover class descriptions). Finally, thread 1 will be initialised and started.

Event Function 5

This event triggers when the user interacts with the game menu in order to make a move. When this triggered, thread 1 will be given permission to continue and will evaluate the current move of the user, if it is invalid, the thread will alter a static setting to indicate to the main program the move was invalid and the thread will return to waiting. When this occurs the program will return to the main loop and wait for another input by the user. However, if the move was valid, thread 1 will coordinates on the board where changes were made and will update the current board accordingly. Then, thread 1 will alter the static setting for the user's current move to be empty so that it can

Event Function 6

This event can be triggered by the user after interacting with the interface when it is set to the main menu window or the game window. This allows them to exit the program.

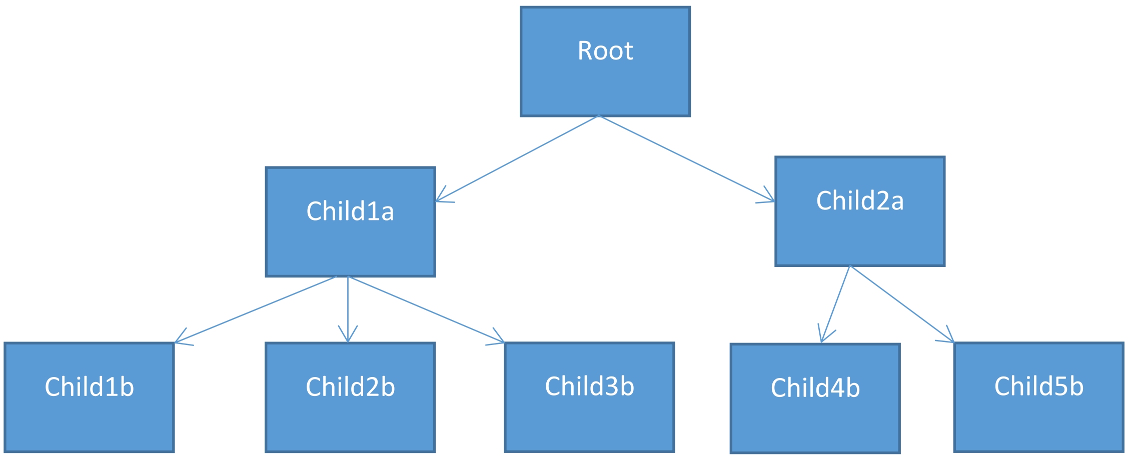
Key Data structures

Game Tree

In order for the AI to make a decision it needs to be able to search through the possible boards that can be made by the both players and be able to calculate the path that will give the AI the greatest score over its opponent. A game tree data structure is an excellent to represent this as it shows a clear hierarchal progression of levels (that can be used to represent the number of turns as the AI searches downwards) and also functions as a directed graph (allowing nodes to point to other nodes along the tree). These properties make the game tree ideal for my requirements.

Given that python does not have an inherent tree datatype available for my use, I will need to use other simpler data types in order to mimic the functionality of one. A possible solution would be to use a series of key value pairs where the key represents the parent board and values represent the children of those parent boards. Furthermore, in order to confine this into a single data structure I will be storing these key value pairs in a list that can be altered if certain nodes are needed to be removed or added.

A visual representation of this data structure can be seen here:



And the proposed data structure that will be used to mimic this:

[{Root: [child1a, child2a]}, {child1a: [child1b, child2b, child3b]}, {child2a: [child4b, child5b,]} …]

In this representation each node is a reference to an object that will contain a separate data structure that will be used to represent the board. Furthermore, each object will have its own key within the list this allows a search algorithm to quickly find the children to any given node within any of the values in the list. And if there are no children to a node (meaning that it would be considered a leaf node), the value for the key of the leaf node would be left blank.

Board

Given that AI needs to search through a large amount of data in order to make a decision, making the process as efficient as possible is essential to make the project viable. This therefore comes into question the type of data structure that will be used to represent the board of each node within the tree (including the current board). As previously mentioned in the analysis section, a solution to this problem would be to refrain from using a 2 dimensional array with characters but instead use 2 arrays that store integers equal to the length of the board, where each integer is a bitmap that will either represent whether or not there. As proved by working in the analysis section, the alternative to using a 2d array uses up less memory and thus should prove to be faster to search through.

Here is an example of a comparison showing a board where we show the presence of a token and the respective bitmap representation:

x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	0	0	x	x	x
x	x	x	0	0	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x

Key:

x = unoccupied

0 = occupied

Bitmap representation:

[0, 0, 0, 24, 24, 0, 0, 0]

As seen in the example, the bitmap is calculated by giving each level in the vertical axis a value that would be accurate with the binary counting system, i.e. (from top to down) 1, 2, 4, 8, 16 etc. Then if a square is found to be occupied, you would add that rows binary value to the total value. In the example above, the value for column 4 and 5 was 24 as the rows values were 8 and 16 (which added together give 24). The same method is also applied to represent the colour of the board (for the project I have considered white tokens to be true and black tokens to be False). Hence by combing these two arrays it is possible for the computer to process the board as the first array tells us if the square is occupied and then (if occupied) if it is a black or white token.

To work out the value of the each row simply substitute the row number (counting downwards) into the equation $2^{(N-1)}$ where N is equal to the row number. However, it is also necessary for the program to also be able to read that there is a token as well as being able to set it. In order to discern if there is a token in a specific square from the bitmap representation I can take advantage of the bitmap format to use shifts. By shifting the total value of a column to the left by the row number we are looking for minus 1, we can see that if the total value now is odd then the leftmost bit must be true which means that the cell it represents must also be occupied. This method is illustrated bellow:

0, 0, 0, 24, 24, 0, 0, 0

1, 2, 4, 8, 16, 32, 64, 128

We wish to know if the coordinates (4, 4) are occupied.

0, 0, 0, 1, 1, 0, 0, 0

1, 2, 4, 8, 16, 32, 64, 128

So by looking at column 4's bit stream we can clearly see that the 4th row is occupied.

1, 1, 0, 0, 0, 0, 0, 0

1 + 2 = 3

However, in order to prove this without looking, if we shift the digits to the left by the row number minus 1 we can see that the row we want now occupies the 1 bit slot. Therefore, if there is a token at our target cords then we simply must prove that the new total number is odd.

3 modulo 2 = 1

This can be simply solved by using taking the modulo 2 of the total number, if there is any remainder, then the new total number is odd and therefore the target cords must also be occupied

Key Algorithms

Tree

Tree generation

To generate the tree data structure, an iterative process is needed as the tree will be constantly changing throughout the course of the game. Furthermore, in order to keep the AI at a constant difficulty it is necessary to also to keep the tree at a constant search depth (meaning its vertical height it terms of nodes must be the same), to do this the function that controls the expansion of the tree must keep a timer to cancel the process when the maximum amount of iterations has been reached. Furthermore, there must also be another function that directly creates the children nodes for an individual parent node and adds it to the tree, this function therefore this function would be called multiple times as controlled by the previous function. Finally, a final function that supplies the expansion function with the data that shows the resultant change to each parent node after every possible legal move has been made, allowing the expansion function to generate its children. The following shows pseudocode that highlights these three functions.

```
DEF EXPAND_TREE(int_expansionLimit):
    int_expansionLimit ← int_expansionLimit + 1
    FOR EACH Node IN AllChildlessNodes:
        list_allFoundChildren ← find_children(Node)
        IF NO CHILDREN:
            PASS
        ELSE:
            CREATE_NEXT_LEVEL(list_allFoundChildren, Node)
        ENDIF
    ENDLOOP
    IF int_expansionLimit > SearchDepth:
        PASS
    ELSE:
        EXPAND_TREE(int_expansionLimit)
    ENDIF
```

```
DEF CREATE_NEXT_LEVEL(list_allFoundChanges, object_parentNode):
    IF No Changes:
        PASS
    ELSE:
        FOR EACH changed_coordinates IN list_allFoundChanges:
            CREATE NEXT NODE
            APPEND NODE TO PARENT VALUE
            CREATE NEW NODE KEY
        ENDLOOP
    ENDIF
DEF FIND_CHILDREN(object_searchBoard):
    List_foundChildren ← []
    Char_colour ←Current Player
    FOR x IN RANGE(1, LengthOfBoard+1):
        FOR y in RANGE(1, LengthOfBoard+1):
            IF MOVE(x, y) VALID:
                SIMULATE MOVE
                APPEND RESULT TO list_foundChildren
            ENDIF
        ENDLOOP
    ENDLOOP
    RETURN list_foundChildren
```

Move Simulation

To simulate any given move in Othello there are 2 functions that need to be performed, first when a valid move has been selected all of the vectors from that coordinate to all other tokens of that colour that lie on either diagonal, horizontal or vertical lines from the placement need to be found and calculated if there is the line between them is made up of the opposite colour token/s. Then when this is calculated, another function must go along each of these line vectors and find the coordinates of all of the enemy tokens that need to be flipped to the opposite colour. The following shows some pseudocode of 2 functions a main function called simulate_move and another called retrieve_changes that is responsible for checking along each found vector.

```
DEF SIMULATE_MOVE(int_x, int_y, char_colour, object_board):
    Dict_foundChanges ←{w: [], b:[]}
    APPEND INT_x, AND INT_y TO Dict_foundChanges
    FOR x IN RANGE(int_x- 1, int_x +2):
        FOR y in RANGE(int_y- 1, int_y +2)
            CALCULATE VECTOR
            IF VECTOR EQUAL T0 [0,0]:
                PASS
            ELSE:
                If char_colour == “w” and GET_TOKEN(x, y) == “b”:
                Dict_foundChanges ← RETRIEVE_CHANGES
            ENDIF
        ENDLOOP
    ENDLOOP
DEF RETRIEVE_CHANGES(int_x, int_y, vector, char_colour, object_board, Dict_foundChanges):
    list_changedCoordinates ← []
    bool_finished ← False
    bool_isValid ← True
    WHILE NOT FINISHED:
        Char_newPos ←GET_TOKEN
        IF Char_newPos == char_colour:
            bool_Finished ←True
        ELSE IF Char_newPos == e OR Char_newPos == o:
            bool_finished ← True
            bool_isValid ← False
        ELSE:
            APPEND (int_x, int_y) TO list_changedCoordinates
        ENDIF
        int_x ← int_x + 1
        int_y ← int_y + 1
    ENDIF
    IF VALID:
        FOR EACH coord IN list_changedCoordinates:
            APPEND coord TO Dict_foundChanges
    ENDIF
    RETURN Dict_foundChanges
```

In order for the AI to make decision

Tree pruning

When the user makes a move or the AI makes a move, the Game Tree needs to be pruned of unused nodes that are now impossible to reach. To do this it is necessary to have a function that can iterate through the tree and eliminate all of these useless nodes from memory, but will also select the part of the tree that is to be saved before it starts the pruning process. This is done in the first function where the designation of the root node. The following is some pseudocode depicting a recursive function that will continue to spread down the tree until it has removed the last unused leaf nodes:

```
DEF PRUNE_TREE(object_node, bool_isFirstPrune, object_selectedChild):
    List_pruneItems ← []
    bool_parentFound ← False
    FOR EACH PARENT, CHILD IN TREE:
        IF PARENT == object_node
            bool_parentFound ← True
            list_pruneItems ← children
        ENDIF
    ENDLOOP
    IF bool_isFirstPrune:
        DELETE object_selectedChild FROM list_pruneItems
        ROOT NODE ← object_selectedChild
    ENDIF

    IF bool_parentFound :
        DELETE KEY FOR object_node
        FOR EACH Child IN list_pruneItems:
            PRUNE_TREE(Child, False)
        ENDFOR
    ENDIF
```

Search

Minimax/alpha beta pruning

When the AI has to make a decision it will be using the Minimax search algorithm, the process by which Minimax makes decision regarding different nodes in a given game tree has already been explained in the analysis section. Furthermore, to make the search more efficient I will also be using Alpha/beta pruning to shorten the length of the search by not having to check through moves that would not make sense from a mathematical standpoint. Again, this algorithm's functionality has already been explained in the analysis section. However, when it comes to its possible implementation, in order for this to function each node in the Game Tree must have some kind of arbitrary value that can be used in comparison. To find this value I have come up with a formula that can be used to calculate this:

$$Value = \frac{xt}{4} + \frac{4a}{t}$$

Where x = Amount of tokens belonging to AI, a = Total Strategic Value of Board and t = Turn Number.

The purpose of this function is for the AI to take into account strategic positioning over quantity of tokens in the early game, but for the opposite to be true in the late game. This is to ensure the AI does not give the other player an advantage during the mid-game where having a large amount of tokens is considered a disadvantage by many professional Othello player as it allows for the opponent to easily take large parts of the board without being stopped.

The definition of what exactly counts as a "strategic position" is highly debatable, however there are some squares on the board that are universally considered valuable. The greatest of these are the corner pieces as they cannot be taken from another player once it has been taken and it offers the greatest range of potential future tokens. For this reason I will establish the value of a corner piece as 10, and the value of the central pieces to be 1 as they can be considered to be the least valuable. The value for the rest of the tokens can be found by dividing up the board into layers (or square rings for better visualisation) and by peeling of each ring you can assign priority value to each corner of that ring as well as the border pieces. This method is what will be used to designate value out to the rest of the tokens on the board.

The following is pseudocode for this proposed strategic value evaluate as well as the functions for the Minimax search algorithm with alpha/beta pruning:

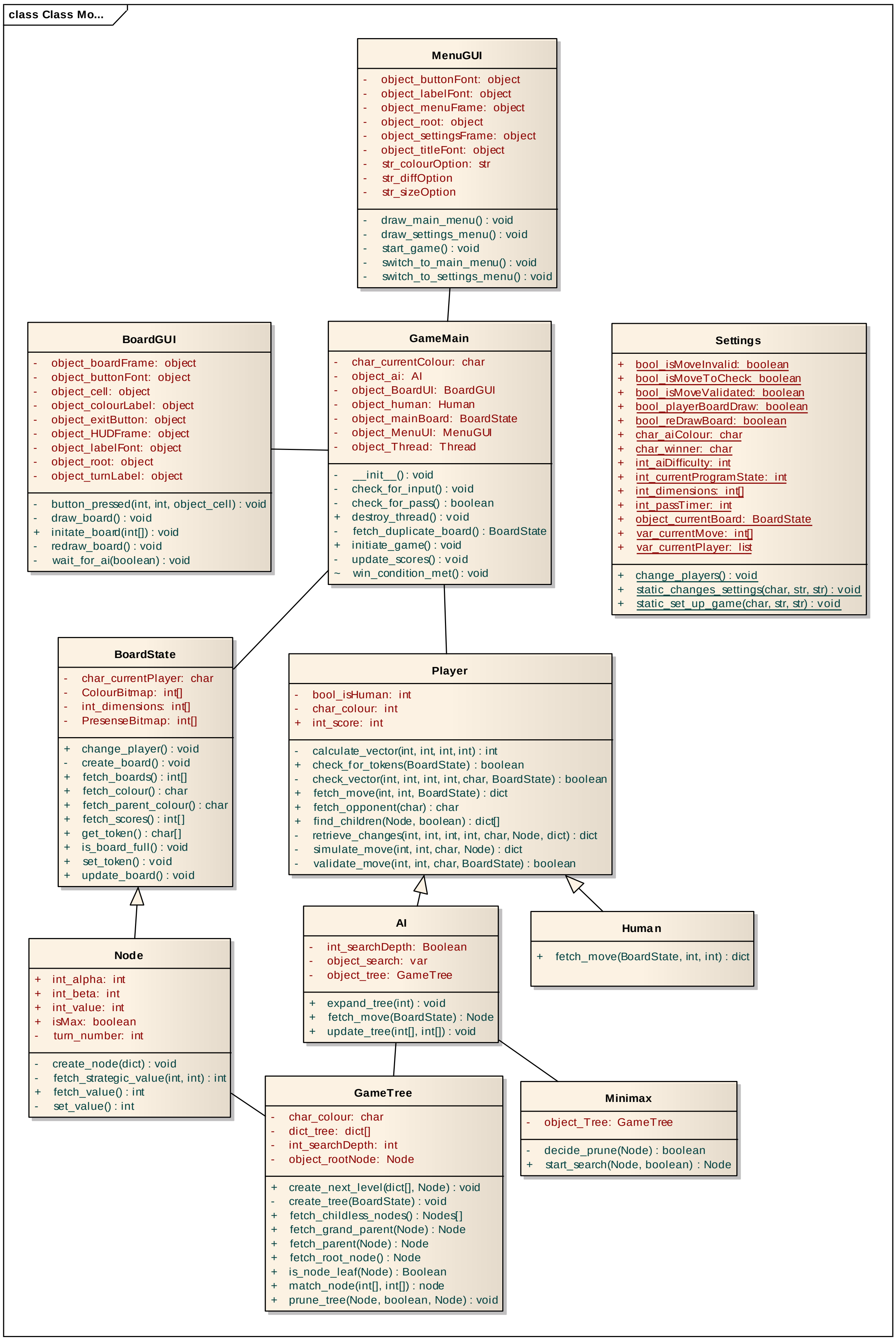
```
DEF FETCH_STRATEGIC_VALUE(int_x,  int_y):
    IF (BOARD_LENGTH/2)+1 > int_x > (BOARD_LENGTH/2)-1 and (BOARD_LENGTH/2)+1 > int_y
    >(BOARD_LENGTH/2)-1:
        RETURN 0
    ELSE IF int_x MOD BOARD_LENGTH ==  0 AND int_y MOD MOD BOARD_LENGTH ==  0:
        RETURN 10
    ENDIF
    FOR EACH INNER RING:
        IF COORD A CORNER PIECE:
            RETURN 4
        ELSE IF COORD LIES ON RING:
            RETURN 2
        ENDIF
    ENDLOOP
DEF START_SEARCH(object_node, bool_isFirstSearch):
    IF object_node IS LEAF:
        Return LEAF VALUE
    ELSE IF NODE IS MAX:
        FETCH ALL CHILDREN
        FOR EACH CHILD OF NODE:
            IF DECIDE_PRUNE(NODE):
                PASS
            ELSE:
                int_value ← START_SEARCH(CHILD, FALSE)
                IF int_value > ALPHA OF NODE:
                    ALPHA OF NODE ← int_value
                    object_selectedChild ← CHILD
                ENDIF
        ENDLOOP
        IF bool_isFirstSearch:
            RETURN object_selectedChild
        ELSE:
            RETURN ALPHA OF NODE
    ELSE:
        FETCH ALL CHILDREN
        FOR EACH CHILD OF NODE:
            IF DECIDE_PRUNE(NODE):
                PASS
            ELSE:
                int_value ← START_SEARCH(CHILD, FALSE)
                IF int_value < BETA OF NODE:
                    BETA OF NODE ← int_value
                    object_selectedChild ← CHILD
                ENDIF
        ENDLOOP

        IF bool_isFirstSearch:
            RETURN object_selectedChild
        ELSE:
            RETURN BETA OF NODE


DEF DECIDE_PRUNE(object_child):
    FETCH GRANDPARENT OF object_child
    IF NO GRANDPARENT:
        RETURN FALSE
    ELSE:
        IF ALPHA OF CHILD > BETA OF GRANDPARENT:
            RETURN TRUE
        ELSE IF BETA OF CHILD < ALPHA OF GRANDPARENT:
            RETURN TRUE
        ELSE:
            RETURN FALSE
        ENDIF
    ENDIF
```

Class diagram

The following main classes have been identified as being used thusly:


class Class Mo...
MenuGUI
- object_buttonFont: object
- object_labelFont: object
- object_menuFrame: object
- object_root: object
- object_settingsFrame: object
- object_titleFont: object
- str_colourOption: str
- str_diffOption
- str_sizeOption
- draw_main_menu() : void
- draw_settings_menu() : void
- start_game() : void
- switch_to_main_menu() : void
- switch_to_settings_menu() : void
BoardGUI
- object_boardFrame: object
- object_buttonFont: object
- object_cell: object
- object_colourLabel: object
- object_exitButton: object
- object_HUDFrame: object
- object_labelFont: object
- object_root: object
- object_turnLabel: object
- button_pressed(int, int, object_cell) : void
- draw_board() : void
+ initate_board(int[]) : void
- redraw_board() : void
- wait_for_ai(boolean) : void
GameMain
- char_currentColour: char
- object_ai: AI
- object_BoardUI: BoardGUI
- object_human: Human
- object_mainBoard: BoardState
- object_MenuUI: MenuGUI
- object_Thread: Thread
- __init__() : void
- check_for_input() : void
- check_for_pass() : boolean
+ destroy_thread() : void
- fetch_duplicate_board() : BoardState
+ initiate_game() : void
- update_scores() : void
~ win_condition_met() : void
Settings
+ bool_isMoveInvalid: boolean
+ bool_isMoveToCheck: boolean
+ bool_isMoveValidated: boolean
+ bool_playerBoardDraw: boolean
+ bool_reDrawBoard: boolean
+ char_aiColour: char
+ char_winner: char
+ int_aiDifficulty: int
+ int_currentProgramState: int
+ int_dimensions: int[]
+ int_passTimer: int
+ object_currentBoard: BoardState
+ var_currentMove: int[]
+ var_currentPlayer: list
+ change_players() : void
+ static_changes_settings(char, str, str) : void
+ static_set_up_game(char, str, str) : void
BoardState
- char_currentPlayer: char
- ColourBitmap: int[]
- int_dimensions: int[]
- PresenseBitmap: int[]
+ change_player() : void
- create_board() : void
+ fetch_boards() : int[]
+ fetch_colour() : char
+ fetch_parent_colour() : char
+ fetch_scores() : int[]
+ get_token() : char[]
+ is_board_full() : void
+ set_token() : void
+ update_board() : void
Player
- bool_isHuman: int
- char_colour: int
+ int_score: int
- calculate_vector(int, int, int, int) : int
+ check_for_tokens(BoardState) : boolean
- check_vector(int, int, int, int, char, BoardState) : boolean
+ fetch_move(int, int, BoardState) : dict
+ fetch_opponent(char) : char
+ find_children(Node, boolean) : dict[]
- retrieve_changes(int, int, int, int, char, Node, dict) : dict
- simulate_move(int, int, char, Node) : dict
- validate_move(int, int, char, BoardState) : boolean
Node
+ int_alpha: int
+ int_beta: int
+ int_value: int
+ isMax: boolean
- turn_number: int
- create_node(dict) : void
- fetch_strategic_value(int, int) : int
+ fetch_value() : int
- set_value() : int
AI
- int_searchDepth: Boolean
- object_search: var
- object_tree: GameTree
+ expand_tree(int) : void
+ fetch_move(BoardState) : Node
+ update_tree(int[], int[]) : void
Human
+ fetch_move(BoardState, int, int) : dict
GameTree
- char_colour: char
- dict_tree: dict[]
- int_searchDepth: int
- object_rootNode: Node
+ create_next_level(dict[], Node) : void
- create_tree(BoardState) : void
+ fetch_childless_nodes() : Nodes[]
+ fetch_grand_parent(Node) : Node
+ fetch_parent(Node) : Node
+ fetch_root_node() : Node
+ is_node_leaf(Node) : Boolean
+ match_node(int[], int[]) : node
+ prune_tree(Node, boolean, Node) : void
Minimax
- object_Tree: GameTree
- decide_prune(Node) : boolean
+ start_search(Node, boolean) : Node

What each class does:
GameMain
The main class “Game”, when instantiated, will form the main container for the game. When instantiated it will first instantiate the “MenuGUI” class and will wait the correct event made by the user calls the “initiate_game” method that will begin instantiating all of the other major classes, including: “BoardState” class, “BoardGUI” class and 2 “Player” sub classes called “Human” and “AI”. When all of the needed main objects have been created, GameMain will then create an object from an external class called “Thread” that is designed to allow for the support of multithreading in python. The target for this object will be the method “check_for_input” that will wait for an event in the program where the player makes a move and will manage the main board as well as the AI’s tree updates and the checks to see if the game has been won. The Method “check_for_pass” is designed to check if the current player cannot move and skip their turn if they cannot move. If they cannot move the method will also check to see if the next player can move to see if the game win condition has been met. The “update_scores” method is designed to update the scores of each player whenever a valid move has been made. Furthermore, the “fetch_duplicate_board” method is used to fetch a clone of the current board object so it can be used as a node or as a means of calculation or comparison for the player classes as well as the tree class. Lastly, the “win_condition_met” method will be called whenever the win condition event has been triggered and will display the winner (or a draw) as well as the final score for each player and will wait for the user to exit.

Player – base class for a player

The class “Player” is used as a base class for the “Player” and “AI” subclasses, it is used to hold an attribute (“colour”) that holds a character to represent the colour the player is using for the current game. There is also another attribute that is also used to store if the player is a human or AI (bool_isHuman) and there is the “int_score” attribute that holds that players last updated score. Furthermore, the player class also offers getters and setters for the move that each player makes, resulting in the key interactions necessary between the subclasses of “Player” and with the main class “Game”. The player class also holds the key simulation methods prototyped earlier, including: “fetch_children” (that finds all of the possible changes that can be made to any given parent node), “simulate_move” (that manages and finds all of the possible vectors that have a chance of altering the board and returning the changes made), “calculate vector” (a simple method that performs a simple coordinates to find the vector to go from one coordinate to another) and “retrieve_changes” (works directly with “simulate_board” as it searches each of the vectors it provides and returns any changed coordinates to the other function). Finally, the class also contains 2 key methods for validation, one of these is “check_for_tokens” which is a simple method that checks to see if any free spaces are present on the board to check for a possible win condition on the players turn, also there is the “validate_move” method that checks to see if a set of inputted coordinates class as a valid move to an inputted board.

Human – subclass of Player

The class “Human” is a subclass of player and is used to validate and return the changes made by the user’s inputted moves during the game. While the class does not have any unique methods or attributes it does override the player “fetch_move” method so it returns a dictionary as well as calling a validation function before calculating the changes made to the board.

AI – subclass of Player

The class “AI” is a subclass of player and is used to manage the GameTree object as well as to calculate the move made by the computer against the human player. It performs these two functions by using 3 core new methods. For tree management it uses the method “expand_tree” (covered earlier) that is designed to generate the tree until it has reached the maximum search depth and the “update_tree” method that is called whenever the opponent makes a move and prunes the tree as well as generates another layer once it knows which branch the player has selected. For making decisions, the subclass overrides the player method “fetch_move” so that it will create an instance of the Minimax class and will execute its search and will return the node that it has decided to move to as well as calling the other tree management methods when the search function has completed.

Minimax
The class “Minimax” is a class that is dedicated to being the container for the Minimax search algorithm as well as the alpha/beta algorithm to enhance the efficiency of the search. The object when created by the AI class will contain an attribute that hold a temporary copy of the GameTree so that it can be searched through. The class has 2 methods, the first “start_search” (covered early) contains the actual Minimax algorithm and functions as a recursive function that will flow through the tree and will then work its way up until it has found an optimal path to take. The second method “decide_prune” (also covered earlier) is the primary alpha/beta mechanism as it will decide for each node if it should be pruned, or in other words if it should be ignored and left out of the search. Details on the functionality of the program can be found in its explanation in the analysis and its pseudocode.

Board state

The class “BoardState”, when instantiated, is used to represent the current board of the game and to offer methods in order to interact with the board. Furthermore, in order to construct a game tree for the AI, it is necessary to have a linked list of child “BoardState” objects so that the program can represent all of the different possible board states that could span from any given current board. However, given that many different interactions of the board have to be held in memory and searched through, it is needed to make the storage of the board to be efficient so It does not risk slowing down the system. Currently, there are two different methods that can be used to store the board. One is to use a 2-dimensional array of characters – where a character would represent the state of the square i.e. empty, black etc – to represent the whole board. The alternative to this is to store the board on 2 different lists of 8 integers where the integers act as bitmaps, in that their binary representation can be used to show two different states on the board. Given that this method is binary in nature, 2 different lists are needed to represent 4 different board states (occupied, unoccupied, black, white). Furthermore, bitwise operates will be needed to apply masks and shift operations in order to interact with the board, although all of this will be abstracted to the level of this object thus the rest of the program can still deal with normal coordinates without having to worry about the details of how to handle the bitmap interactions. To do this, get/set methods such as “FetchToken” and “SetToken” would be used to encapsulate this within the “BoardState” object.

To demonstrate if either method is more efficient I am going to perform a simple calculation in which I will assume that python uses 16 bits to represent the value of an integer variable and python also uses ASCII encoding (8 bits) for its characters.

Total Bits used for integer storage = 16(bits per number) x 8(integers per list) x 2(lists) = 256(bits in total)
Total Bits used for string storage = 8(bits per letter) x 64(letters) = 512(bits in total)

So, as we can see here if these 2 assumptions are true (note I assumed the best-case scenario for the string storage) then using the integer method is clearly superior to the string method. However, even if we were to assume that python used 32 bits instead of 16 bits to represent its integers that would mean that the total space used would still be 512 and thus equal to the best-case scenario for the string method. Therefore, I conclude that for the interest of efficiency in our system it would make sense for me to use the integer method of storage of the board so that is what I will now choose to move on with.

Node

This subclass of "Boardstate" is used to represent the nodes within the tree data structure. It has some notable differences to its parent class. Firstly, it has the following new attributes: "char_colour" (used to store the colour of the player who would be playing on the same turn as this board), "int_alpha" (used to store the current alpha value used for alpha/beta pruning during the AI's Minimax search), "int_beta" (used to store the current beta value used for alpha/beta pruning during the AI's Minimax search), "isMax" (a Boolean used to tell if the node is a maximiser or a minimiser during the Minimax search) and "int_value" (a integer that stores the value of the current board for the AI and is used to make decisions during the Minimax search). Secondly, the subclass also has 3 additional methods to the base class. The first of these is called "create_node" and is used to map the dictionary of changes to the parent board onto it in order to create the

GameTree

The class "Tree" is used to store and manage the data structure that the AI will search through in order to make a decision. The data structure stored in this class is designed to mimic the functionality of a tree. In this tree the various nodes are represented by the "Node" subclass. The tree itself is stored as a list of dictionaries with a key value pair of a parent and its children, with this system the root node is distinguished from the rest by having a special attribute in the class that stores a reference directly to the current node that is the root node of the tree. The class contains 2 key methods, one of these called "create_next_level" is responsible for expanding the tree under a selected leaf node for the next turn when it receives a dictionary from the player classes that depict the changes to the board for each possible move that can be made. When receiving this dictionary it will call the "Node" subclass to instantiate the nodes by providing it with its parent and a dictionary with coordinates that show which coordinates on the board have been changed to white and vice versa. Another key method is "prune_tree", this is a recursive function that has the purpose of pruning the tree after a player has made a move so it can get rid of sections of the tree that no longer can be reached thus increasing future performance considerably. For the first iteration of the function the key that references the root node is found and the child picked to be the new root node it selected from the roots children and the attribute that stores the root node is altered to represent this selected the node. The rest of the children are appended into a list and the current key value pair is deleted. The function then calls itself for each child and repeats the following process for all other recursions, the key value pair for the parent is found and its children are appended to a list and the key value pair is deleted, the function is then recalled for each child stored in the list. By repeating this process the tree will be completely pruned of all unusable nodes.

Settings

The static class "Settings" is used to hold important variables about the state of the program in memory and to also act as a platform to communicate between the thread and the main loop. It holds important control signals that govern the order of execution in the system such as the attribute “bool_isMoveValdated” that will not let the main loop update the main board display until the move made by the player has been validated and if valid, simulated. It also contains attributes that hold temporary important data such as var_currentMove which stores the current move made by the player and passes it to the thread to be analysed. As well as the attributes, the class also has 3 different static methods that are used to change the static variables. The first is “change_player” that simply switched the “var_currentPlayer” variable so now it stores the opposite player and the opposite colour to what it did before. The second is, “change_settings” which is called by the “MenuGUI” class when the user alters the main settings so this is called to update those respective variables. The final method “set_up_game” is simply used to change the attribute “int_currentProgramState” to 2 so that the main program knows that the system has transitioned from the menu to the game.

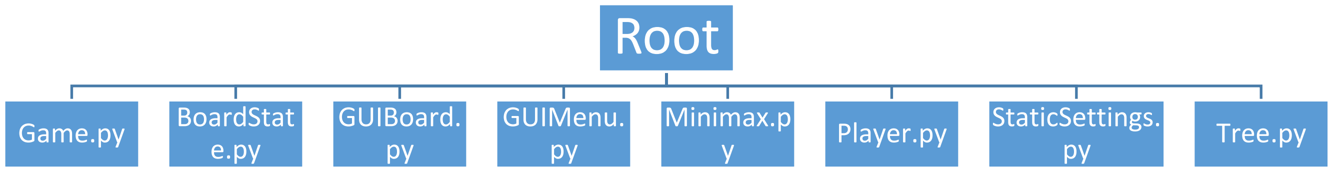
MenuGUI

The “MenuGUI” class acts as the container for the user interface during the pregame part of the system. When the game class is initialised this class will be instantiated and it will instansiate classes from the external library called “TKinter” which is a library dedicated to the generation and management of a graphical interface for python. The classes TKinter Attributes fall under a central “root” attribute that is divided between 2 frames that manage the main menu and the settings menu. Each of these will manage the other button and label attributes as well as 3 drop down selection boxes for the Graphical interface. When these have been created the class will hold the main loop that will listen for events being called by any of these buttons and will call the various other functions in the program as shown in the high level flowchart diagram.

BoardGUI

Similarly to the “MenuGUI” class, this class also acts as a container and manager for the graphical interface during the game phase of the program. When the program transitions to the second phase, the class will hold the main loop which will listen for events from any of the tkinter objects. The main loop centres around listening for the “button_pressed” event which is triggered when the player clicks on one of the board cells and passes the move to the settings class and sends a control signal to the thread to begin analysis of the players move, this method will then call “wait_for_ai” which will wait until the thread has completed processing the AI’s decision and will redraw the board accordingly.

File Structure



User Interface

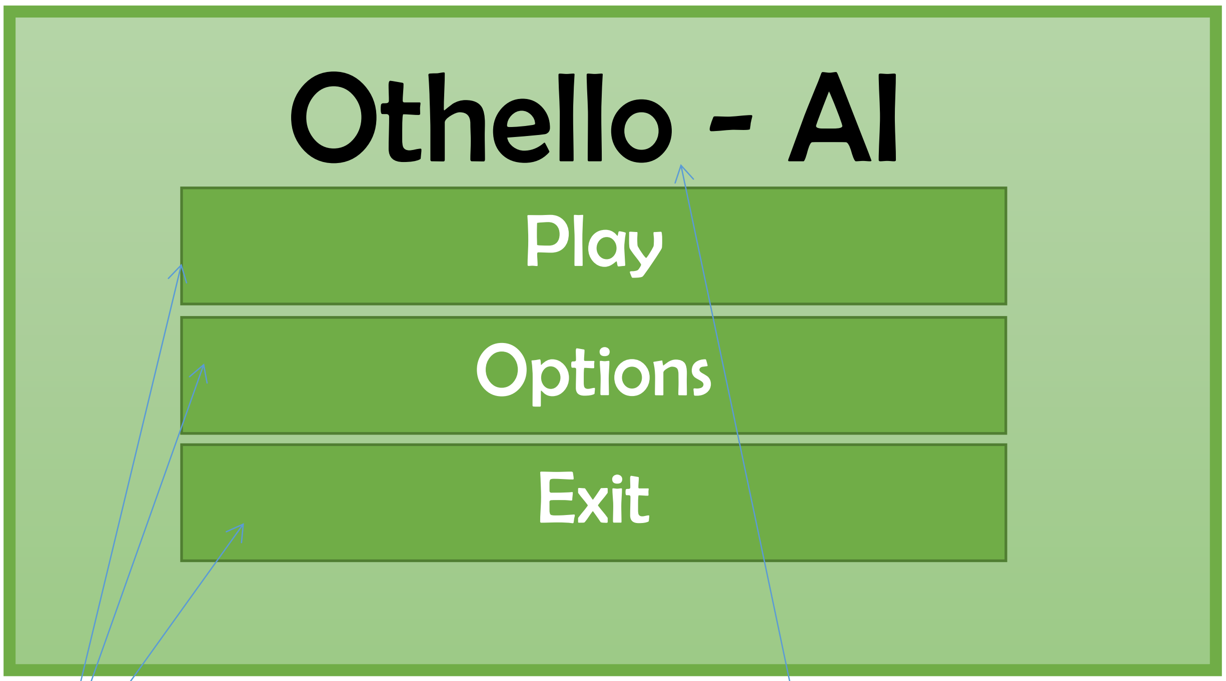
The UI has been designed around the needs of the users which have been summarised to be the following:

- The UI must be graphical, a text based interface cannot address the highly variant technical skill of the target audience.
- A very simplistic interface with the minimal amount of settings and widgets to prevent any form of confusion.
- A clearly labelled interface with each button having a label that identifies the purpose of each widget without ambiguity as to what they do.
- Simple English as young age ranges will be playing the game so avoiding complex language will greatly improve the success of the UI.
- Contrasting colours are important as they allow clear identification of buttons and other widgets used for user input.
- A clear and simple ruleset that can be customised to some degree before imitating games.

Here is the justification as to how the user interface design meets this selection criteria:

The user interface I have selected for this project is graphical. My justification for this is because of the large variance in the target age range. For this reason I have chosen to use the Berlin Sans FB font with black and white colouring in order to create a high contrast with the background making all of the labels extremely clear. Furthermore, I am also using large font sizes (25 ~ 40pts). All places where text has been required (for buttons, labels etc), simple English has been used and large and potentially complicated words have been replaced. For example the word “dimensions” was replaced with “size” in the settings menu. The settings menu itself was created with the purpose to provide the user with the ability to expand on the normal Othello ruleset by allowing minor customisation such as board size and the starting turn order.

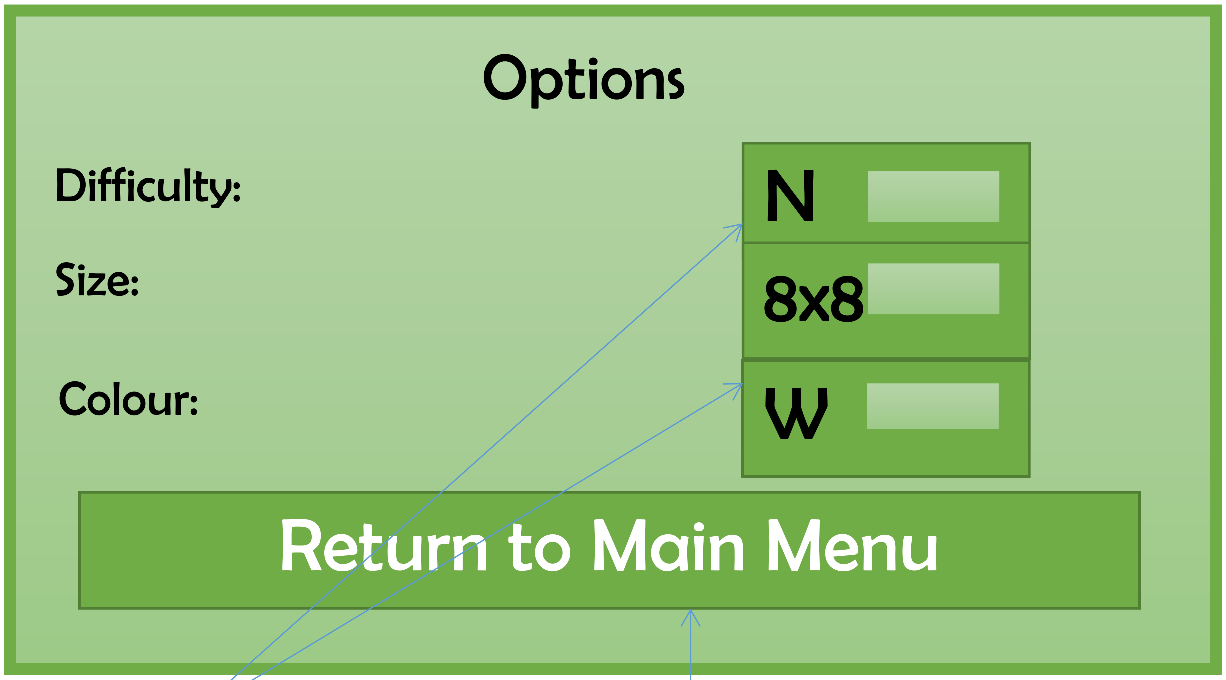
The following are some diagrams that depict the design for each of the 3 windows that will be in the program.



Large buttons used to create clear user inputs

The title has been coloured black instead of white to differentiate it from the button.

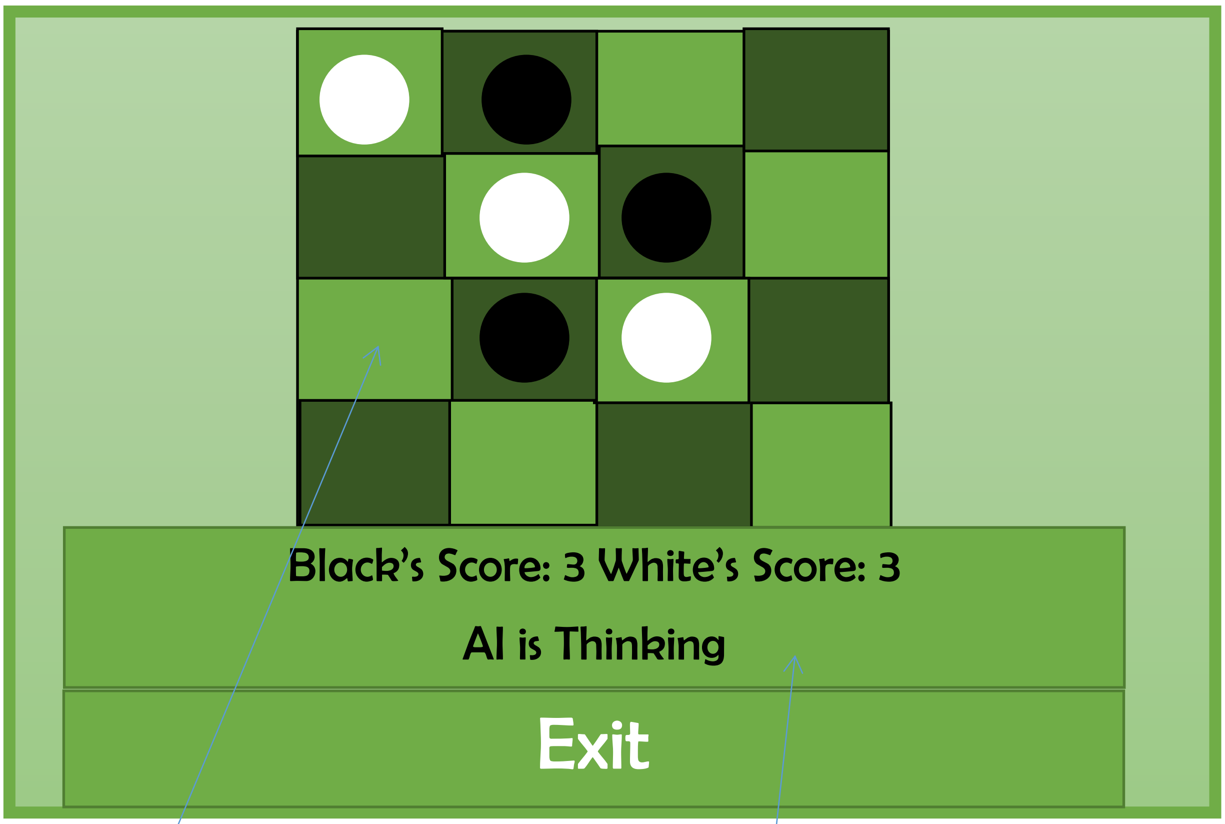
>Options Menu



Drop down boxes used so that the user can easily select from the different options for each setting available.

Return to main menu button allowing users to switch between the menus.

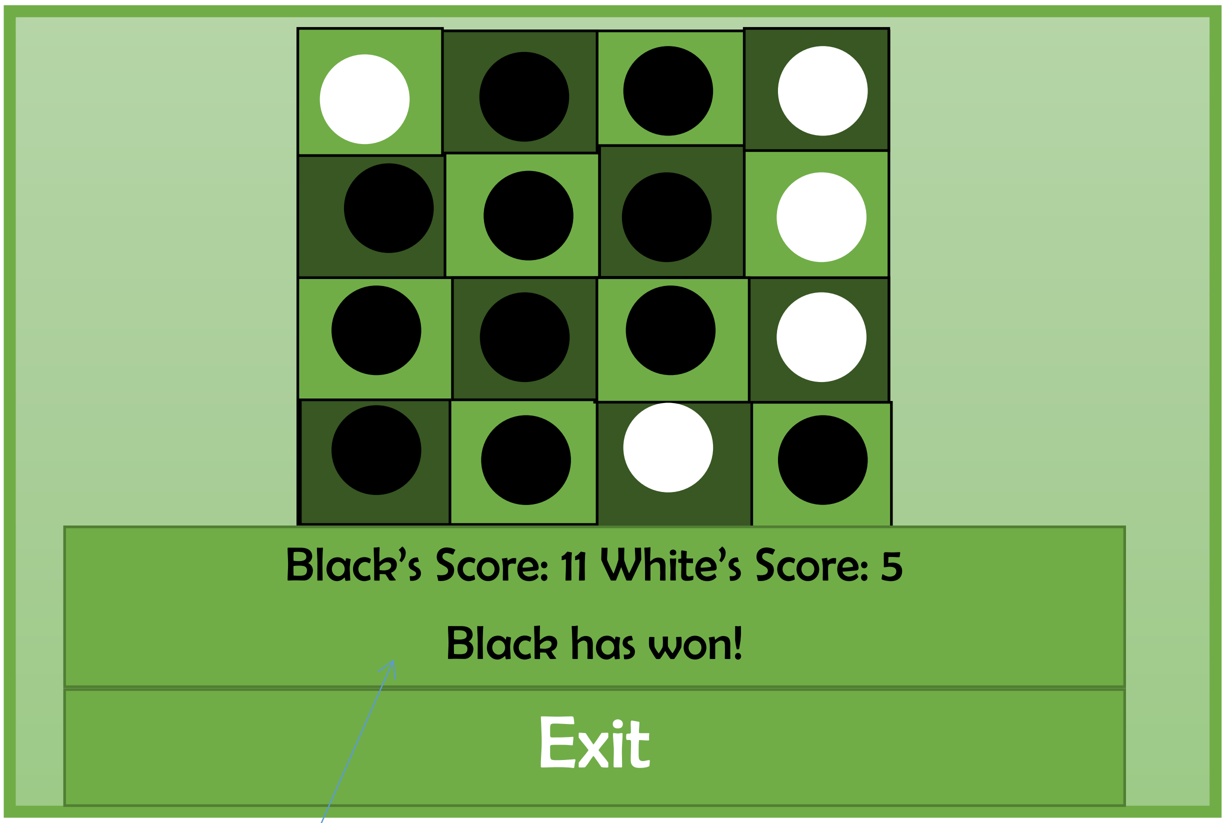
>Play Menu



The Board is designed to be directly intractable so the user has to click on the square that they want to place token at.

The black label is designed to show the current state of the game including the score for each player and who has the current turn.

>Game Finished



And when the game has been finished the main label will display the winner (or display a tie) when a game is finished along with the finished score for each player.