|  |  |
| --- | --- |
|  |  |
| HSC SDD STAGE 1&2 |  |
|  |  |
|  | Jack Bashford3623819438177IS1101 2021Maze Game |

# Table of Contents

Contents

[HSC SDD STAGE 1&2 1](#_Toc89187814)

[Jack Bashford 1](#_Toc89187815)

[36238194 1](#_Toc89187816)

[38177 1](#_Toc89187817)

[IS1101 2021 1](#_Toc89187818)

[Maze Game 1](#_Toc89187819)

[Table of Contents 2](#_Toc89187820)

[Design Specifications 4](#_Toc89187821)

[User’s Perspective 4](#_Toc89187822)

[Developer’s Perspective 5](#_Toc89187823)

[User Section 6](#_Toc89187824)

[Overview of the program 6](#_Toc89187825)

[Screen Designs and Descriptions of Screen Designs from User’s Perspective 7](#_Toc89187826)

[Form 1: frmMain 7](#_Toc89187827)

[Form 2: frmInstructions 8](#_Toc89187828)

[Form 3: frmGame 9](#_Toc89187829)

[Form 4: frmGameOver 10](#_Toc89187830)

[Form 5: frmHighscores 11](#_Toc89187831)

[Developer Section 12](#_Toc89187832)

[Level 1 Data Flow Diagram 12](#_Toc89187833)

[Description of Data Flow Diagram 12](#_Toc89187834)

[Data Structures 13](#_Toc89187835)

[Arrays 13](#_Toc89187836)

[Arrays of Records 13](#_Toc89187837)

[Files 14](#_Toc89187838)

[Structure Chart 15](#_Toc89187839)

[MazeGame (main) 15](#_Toc89187840)

[SetupGame 16](#_Toc89187841)

[PlayGame 17](#_Toc89187842)

[ScoreGame 18](#_Toc89187843)

[Description of purpose of each module 19](#_Toc89187844)

[Function 1: validateName 22](#_Toc89187845)

[IPO Diagram 22](#_Toc89187846)

[Data Dictionary 22](#_Toc89187847)

[Pseudocode 23](#_Toc89187848)

[Function 2: generateSeed 24](#_Toc89187849)

[IPO Diagram 24](#_Toc89187850)

[Data Dictionary 24](#_Toc89187851)

[Pseudocode 25](#_Toc89187852)

[Subroutine: sortHighscores (and swap) 26](#_Toc89187853)

[IPO Diagram 26](#_Toc89187854)

[Data Dictionary 26](#_Toc89187855)

[Pseudocode 27](#_Toc89187856)

# Design Specifications

## User’s Perspective

* The game will have three primary difficulties, with varying maze sizes (standard sizes of 10x10, 20x20, and 30x30, with a random choice of those three available as well).
* A main screen with access to the highscores, instructions, and gameplay screens will be available to view upon the initial launch of the program.
* An ending screen with access to the highscores and main screens will be available to view upon the conclusion of the game. This will display the time taken through the maze, and ask whether the player would like to view highscores, play again, or exit the game.
* A highscores screen will be available to view from the main screen and the ending screen. This will display the fastest times through a maze of the same size if playing with any of the three standard sizes (10x10, 20x20, 30x30).
* An instructions screen will be available to view from the main screen. This will display the instructions for playing the game.
* A gameplay screen will be available to view from the main screen. This will display the maze, user's position, and elapsed time throughout the game.
* The game's timer starts as soon as the maze is created and the user is able to interact with the game, and the timer ends as soon as the player reaches the finish cell of the maze.
* The user can move their player through the maze with keyboard or mouse shortcuts. They will only be able to move into open cells, not cells blocked by walls or the edge of the maze.
* The user may choose to prematurely exit the maze, which will not allow the user to achieve a high score ranking position.
* At the conclusion of each game, the user will be taken to the ending screen, and from there they will be able to view highscores and/or start a new game.

## Developer’s Perspective

* The maze will be stored as a 2D array arrGameBoard, consisting of cells represented as integers.
  + Each cell will contain a binary number (0000-1111) which is the mask of the walls that are open on that cell (i.e. 0010 would mean the southern wall is open only).
* The maze will be generated using a randomised algorithm that is based upon a seed (this seed can be used to predictively randomise the Randomize() function in VB.NET).
  + This can be used to regenerate the same maze from a certain seed if the user wishes to re-play that maze. Note it will only work for mazes of the same dimensions, i.e. a smaller portion of a maze cannot be generated with a seed of a larger maze.
  + The algorithm (recursive backtracking) creates a simply connected maze (i.e. it is made from branching passages and contains no loops) which is guaranteed to have a solution.
* The player's position will be represented by a tuple playerPosition (Integer, Integer) of the x-y coordinates relative to the starting position / start cell. Each movement will change the relevant component direction's value within that tuple by a value of 1 or -1.
* The game screen will be drawn using the Windows GDI+ graphics engine; this will allow for a range of shapes to be drawn, including pixel-by-pixel drawings.
  + Each combination of walls and open sides will be drawn following the bitmask rules for each cell (this will only have to happen once at the start of the game, and will be a linear operation with a worst-case runtime complexity of O(4n) which is equivalent to O(n).
* When the game starts, the maze will be generated and drawn, the user will be drawn at the start cell, and the timer will begin as soon as the user is able to move.
  + The timer will stop as soon as the user enters the final / winning square.
* Upon each movement command, the game will either move the user in the requested direction if it's possible to do so, or nothing will occur as there is a wall or maze boundary.
* The game's highscores will be represented within the game as an array of records for each maze size (it will be fetched upon loading of the highscores form, to avoid unnecessary lag and system resource usage when starting playing the game).
  + The record recGameResult will be used to represent the result of a game, containing the player's name (String), the maze dimensions (Integer), the elapsed time in milliseconds (Integer), and the maze's seed (Integer). This is a 'temporary' record (i.e. a maximum of only one data item of this record type will exist at any point in time) as it's either discarded if not a highscore time, or its data is copied into a new recHighscore to be stored in the highscores.txt file and the recGameResult value is discarded. As such, it does not appear in any system models because it will be entirely contained within one subroutine (checkIfHighscore).
* The project as a whole will be constructed with the top-down development method, using the structured software development approach. This will ensure that the resulting program is sufficiently robust with no flaws upon usage, but is also clearly structured in a hierarchical manner within the source code.
  + The source code will contain both internal and intrinsic documentation to allow for maintenance and optimisation of the program or any of its submodules.
  + Each module will be extensively tested both in isolation and within the final program to ensure the quality of the resulting program.

# User Section

## Overview of the program

Maze Game is a single player game where the player is provided with a randomly generated, square-shaped maze containing a starting point and ending point. The objective is to navigate through the maze from the start to the end in the fastest possible time.

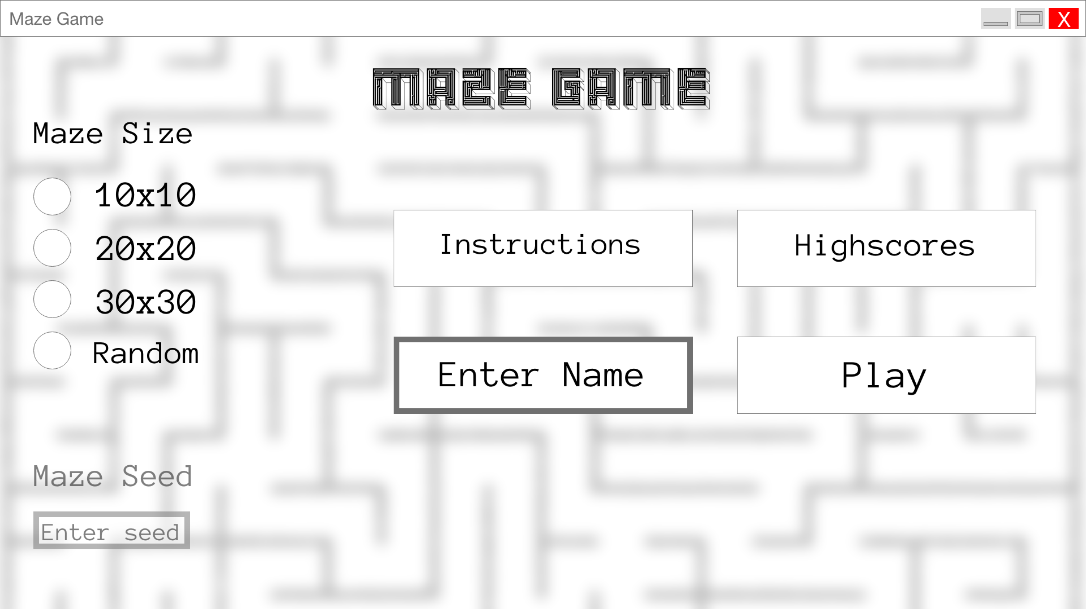
After entering their name and selecting their maze size (and optionally a set seed), the maze will be generated and the player will be able to begin playing the game.

The player's choices at each stage of the game are to move up, down, left, or right. The player is not able to move into a cell that is blocked from their current cell by a wall, nor are they allowed to move outside of the maze.

Once the player enters the maze and is able to move, their timer starts. The timer will end once they enter the final goal square of the maze. From there, they will be able to view the previous fastest times for that maze size in the highscores screen.

## Screen Designs and Descriptions of Screen Designs from User’s Perspective

### Form 1: frmMain

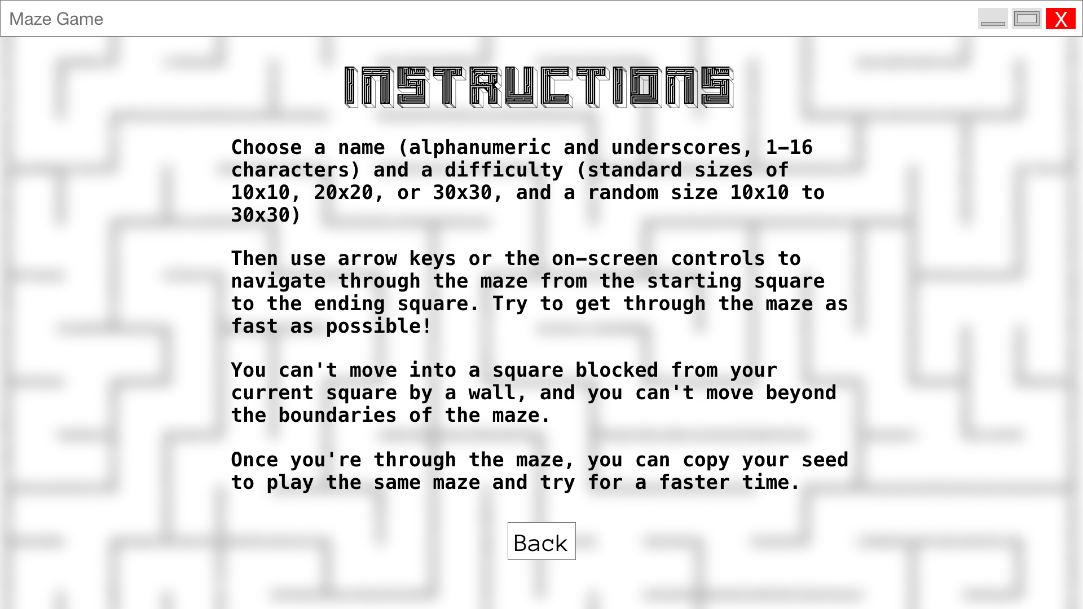


Different difficulties:

|  |  |
| --- | --- |
|  |  |
|  |  |

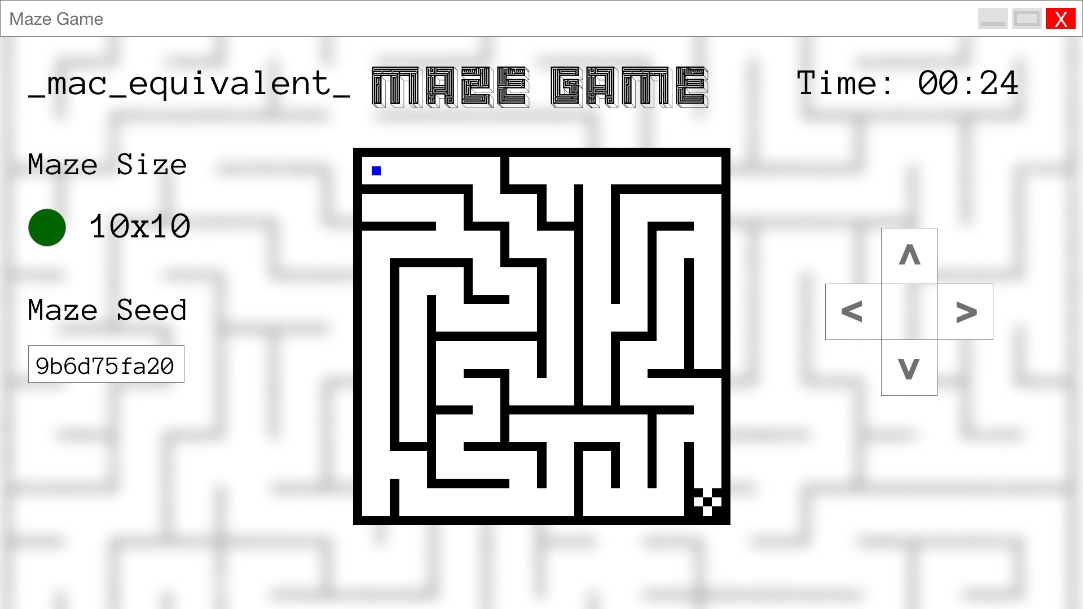
This form is initially shown to the player when the game starts. It allows them to select a difficulty (10x10, 20x20, 30x30, or a random selection of those), access the Instructions and Highscores screens (frmInstructions and frmHighscores respectively), and start the game (which will only occur if the player's name is valid [alphanumeric and underscores from 1-16 characters]).

### Form 2: frmInstructions



This form allows the player to view the game's instructions from within the game; in particular, it provides information about the rules/objective of the game, as well as how to use the seeding functionality in the game. It then allows the player to return to the main screen.

### Form 3: frmGame



Different difficulties:

|  |  |
| --- | --- |
|  |  |
|  | |

This form allows the player to actually play the game once the maze has been generated. It displays their name, the difficulty / size of the maze, the current elapsed time of their run through the maze, as well as the unique seed of this maze (which will allow the player to re-play this maze once they've finished this run). This form also has on-screen controls if the user doesn't/can't use the keyboard controls. The player will be automatically moved from this form to frmGameOver once they've completed the game.

### Form 4: frmGameOver

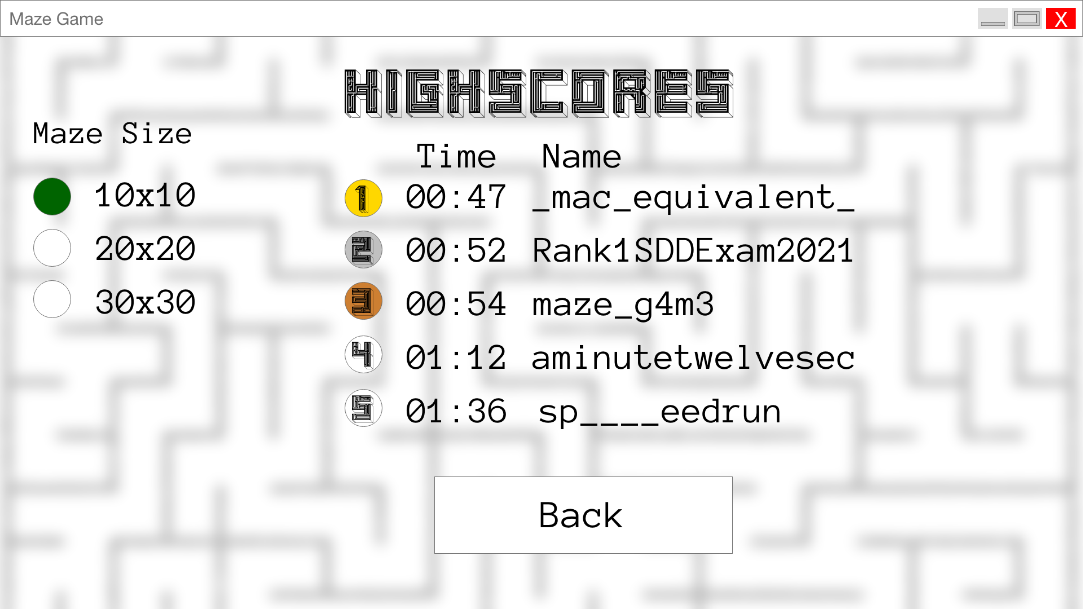


Different difficulties:

|  |  |
| --- | --- |
|  |  |
|  | |

This form is shown to the player once they've completed the game. It provides a breakdown of the game they just played, including their username, the game's size and difficulty, the time that they took to navigate the maze, and the seed that was used to generate the maze if the player would like to play that maze again. This form allows the player to access the highscores (updated after each round), as well as return to the main screen to re-configure a new round of the game.

### Form 5: frmHighscores



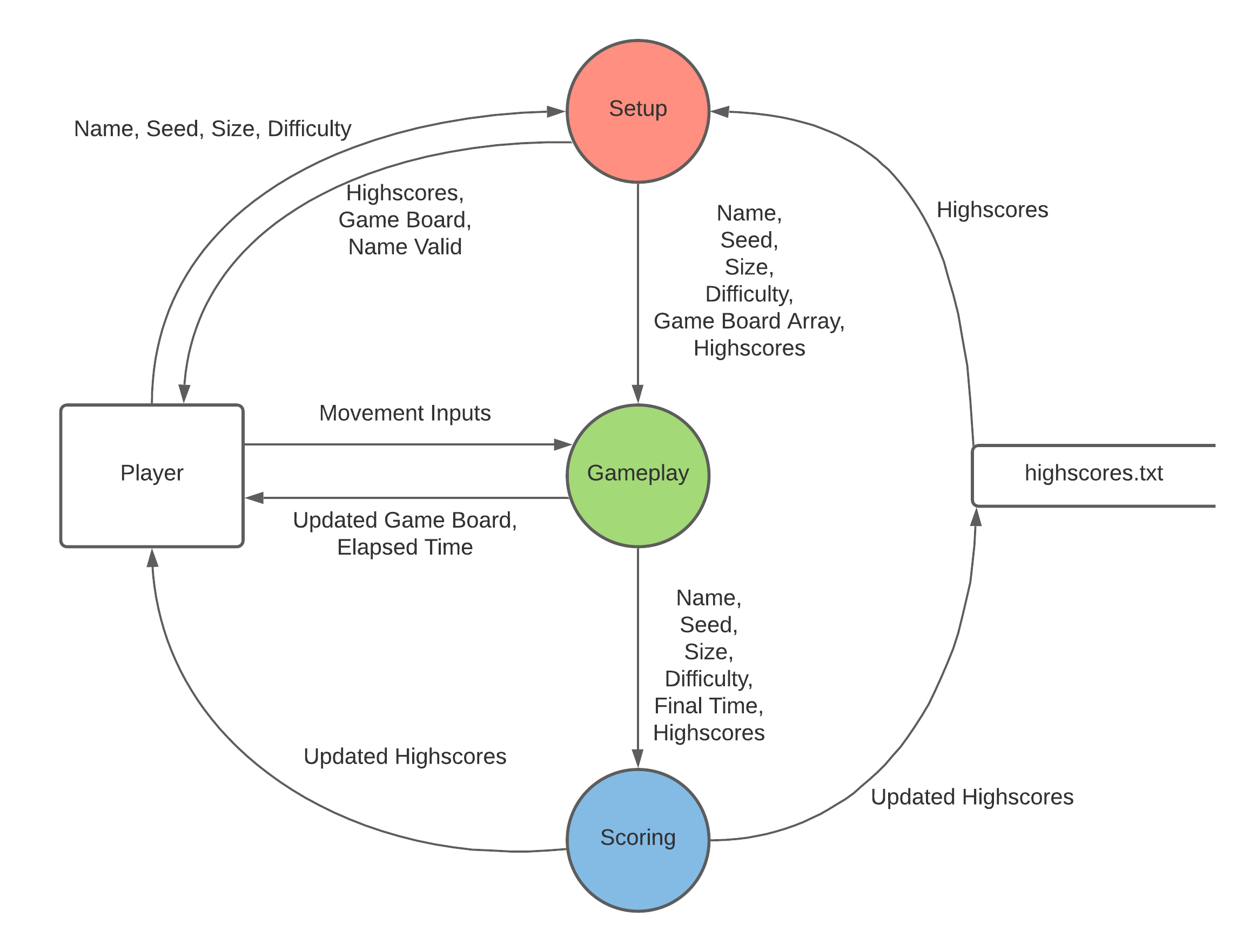
Different difficulties:

|  |  |
| --- | --- |
|  |  |

This form is shown to the player once they've completed the game. It provides a breakdown of the game they just played, including their username, the game's size and difficulty, the time that they took to navigate the maze, and the seed that was used to generate the maze if the player would like to play that maze again. This form allows the player to access the highscores (updated after each round), as well as return to the main screen to re-configure a new round of the game.

# Developer Section

## Level 1 Data Flow Diagram



Note: PDF copy attached in submission

### Description of Data Flow Diagram

The Setup process receives the player name, and the maze seed, size, and difficulty, from the external entity Player. It also reads the highscores from the sequential file highscores.txt. The player's name is validated, and if necessary the player is notified of an invalid name. The highscores are presented to the player also. Finally, the game board is created based on the maze seed, size, and difficulty, and then it is presented to the player once they begin the game. All of the information that the player provided, along with the new game board array, is then sent to the Gameplay process.

The Gameplay process begins once the player has been presented with the game board. It receives movement inputs and requests from the player, evaluating / playing these movements, then displaying the new and updated game board to the player. There is also a timer contained within this process, which sends the elapsed time back to the player at each step/frame of the game. Once the player completes the maze, their name and final time, along with the maze's seed, size, and difficulty, are all sent to the Scoring process.

The Scoring process begins once the player has completed the maze. It receives the player's name and time, and the maze seed, size, and difficulty from the Gameplay process. The highscores are then updated (if necessary) with the new player's highscores (this is done in-place to avoid a complete re-sort of the array), then written to highscores.txt and displayed to the player.

## Data Structures

### Arrays

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Name** | **Data Type** | **Dimension** | **Size/Indexes** | **Description** | **Scope** |
| arrGameBoard | Integer | 2 | Based on mazeSize – any of 10,10; 20,20; or 30,30; | Stores each cell of the maze and the walls each one is surrounded by in a binary bitmask fashion (1100 -> bottom and left sides of the cell are walls). This is procedurally generated at the start of the game (including dimensions, which are user input). | Global |
| arrDirections | Integer | 1 | 4 | Stores the binary mask of each of the four directions (up, right, down, left) and is randomly shuffled for the generation of the maze. | Local to recursePassage |

### Arrays of Records

**Record Name:** recHighScore

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| playerName | String | Stores the name of the player who achieved this highscore. |
| mazeSize | Signed 8-bit integer | Stores the maze size (side length, any of 10;20;30) of this highscore. |
| mazeSeed | Integer | Stores the seed used to generate the maze. |
| gameTime | Integer | Stores the total time that the player took to complete the maze (in milliseconds to avoid any rounding issues that may come with floating point). |

**Array Structure:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Name** | **Data Type** | **Dimension** | **Size/Indexes** | **Description** | **Scope** |
| arrHighscores | recHighscore | 2 | 3,5 | Stores each of the highscores (top 5 for each maze size) for the game. The first dimension indicates the difficulty in ascending order (i.e. arrHighscores(1) would be the highscores for 10x10), and the second dimension is the rank (i.e. index 1 is rank 1 / 1st, index 5 is rank 5 / 5th) | Global |

### Files

|  |  |  |  |
| --- | --- | --- | --- |
| **Filename** | **File Type** | **Structure** | **Example** |
| highscores.txt | Sequential (with sentinel value and field/record delimiters) | P1size,P1name,P1time,P1seed  P2size,P2name,P2time,P2seed  P3size,P3name,P3time,P3seed  …  PXsize,PXname,PXtime,PXseed  9999 | 10,alex,43520,73fa736784  10,AMG,49026,16d12216d9  20,angus,83491,989f1584ae  20,ben,88127,27c2264a92  20,calvin,89316,08a7e43e2e  30,chameen,103762,3dca79fb09  9999 |

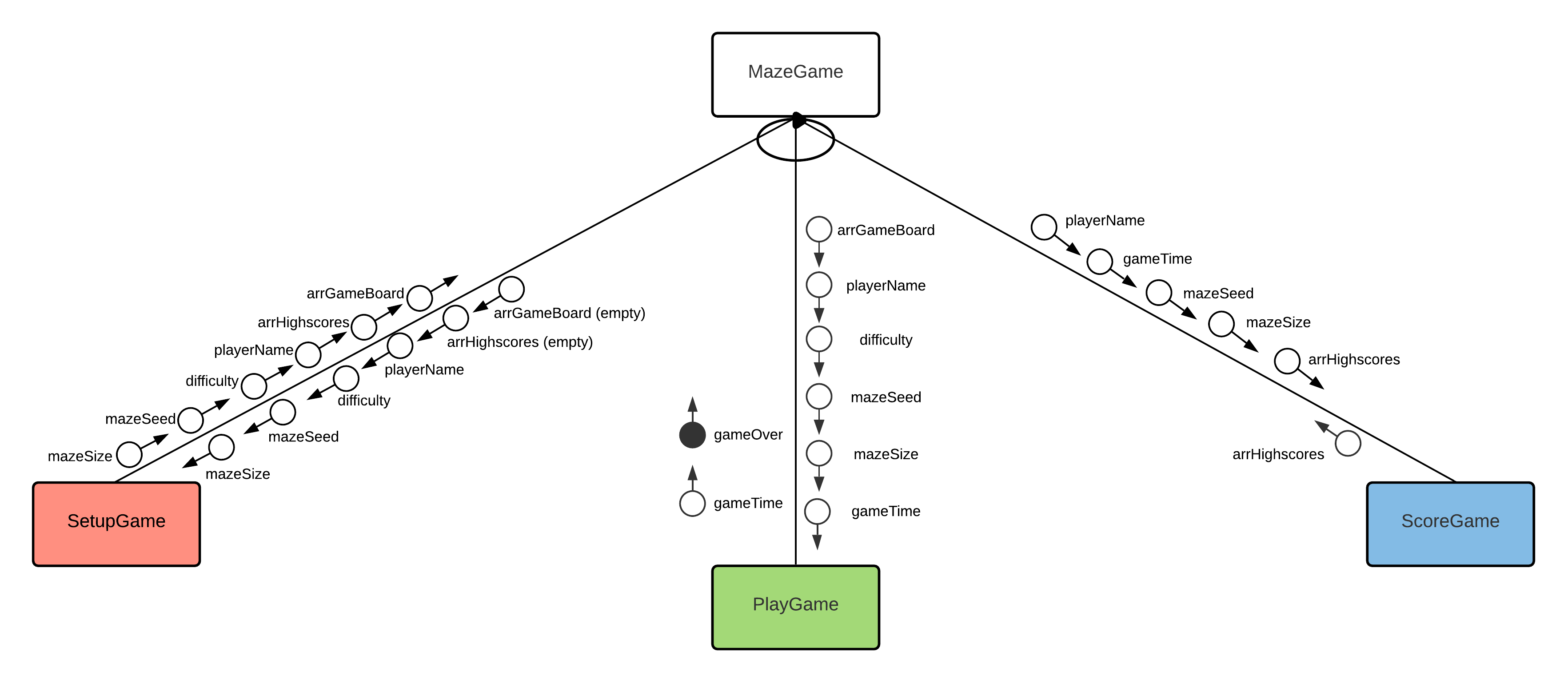
Note that for highscores.txt, the ‘PXtime’ field is stored in milliseconds to keep consistency with gameTime and to avoid losing precision for rounding to seconds. Also note that ‘X’ in PXsize etc. can be two digits if more than 9 people have achieved highscores.

Note: All files are located in the directory 'ApplicationPath\bin\debug\net5.0-windows\filename.ext'. This is to ensure access from within the application is not reliant on any absolute paths (as this is the file path accessible when using 'Open('filename.ext')' in VB.NET)

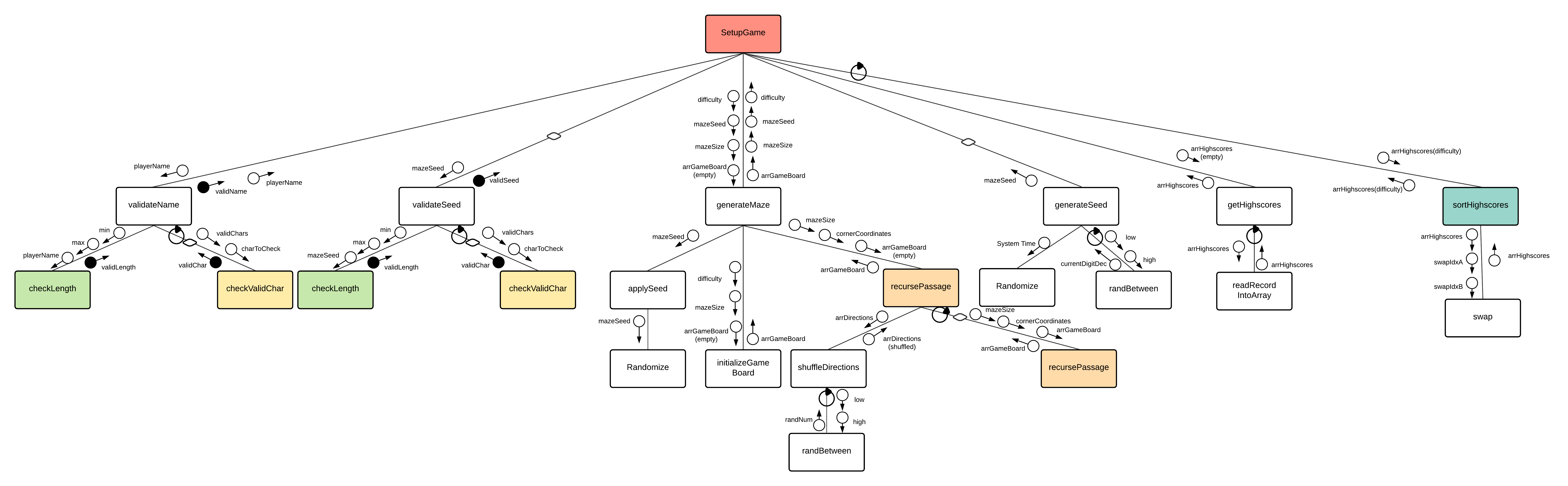
## Structure Chart

Note that PDF copies are included in the submission.

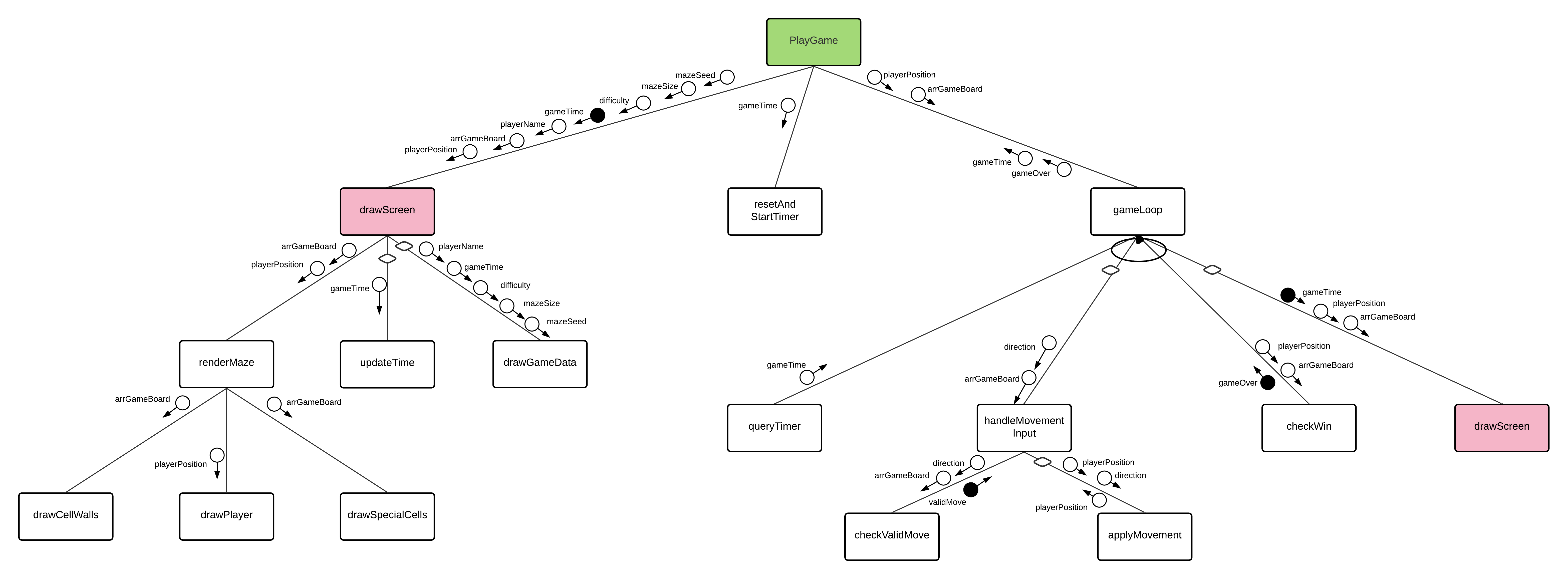
### MazeGame (main)



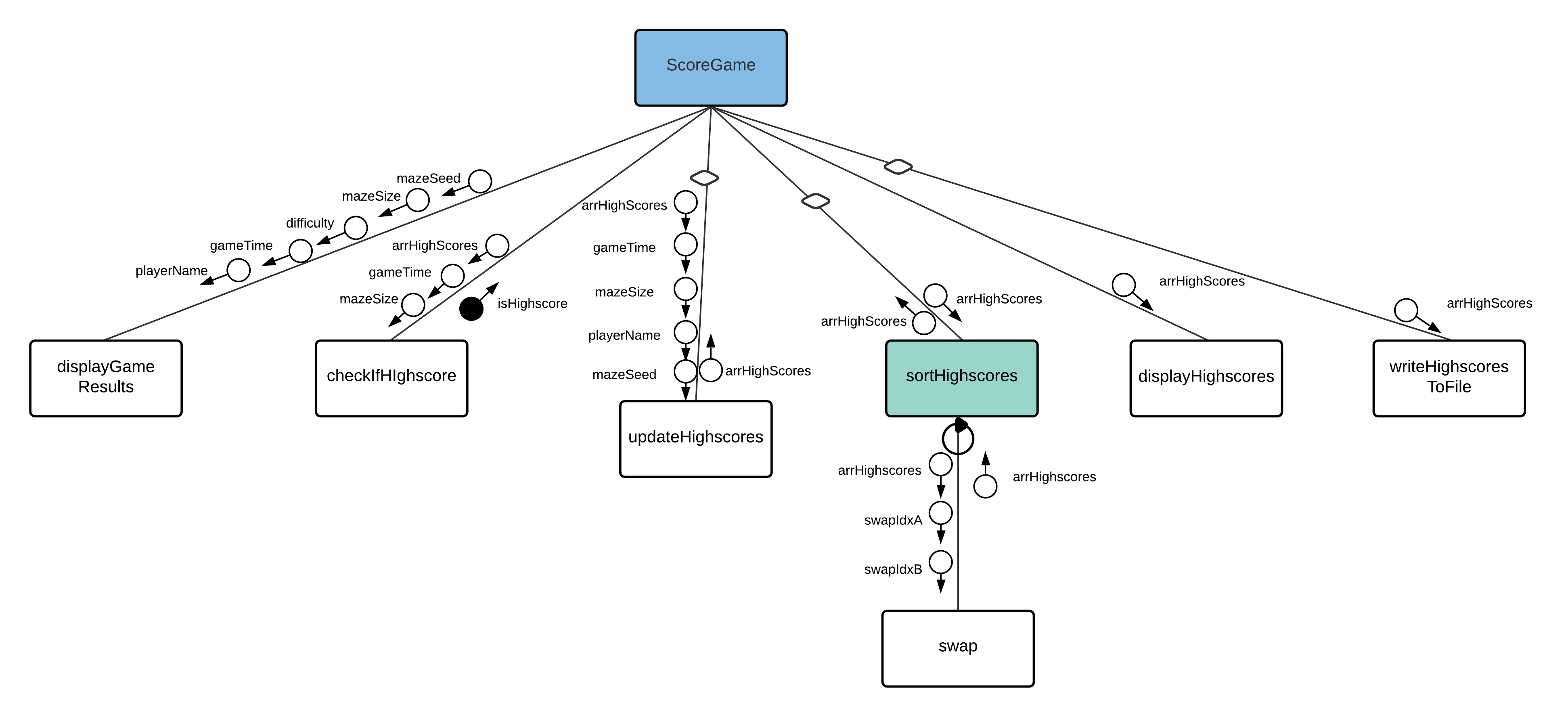
### SetupGame



### PlayGame



### ScoreGame



## Description of purpose of each module

|  |  |  |
| --- | --- | --- |
| Subroutine/function | Control Structures | Description of purpose |
| SetupGame | * Sequence * Binary selection | Runs the setup procedures for the game - validating player name and seed (if necessary), generating seed (if necessary), generating the maze, and fetching the highscores. |
| validateName | * Sequence * Pre-test repetition * Binary selection | Validates the player's entered name (must be 1-16 chars, and consist of alphanumeric and underscore characters). |
| checkLength | * Sequence * Binary selection | Validates that the given string is between the provided min and max values (inclusive). |
| checkValidChar | * Sequence * Binary selection | Validates that the given character is included in the provided valid character range. |
| validateSeed | * Sequence * Pre-test repetition * Binary selection | Validates the player's entered seed (must be 1-10 chars, and consist of valid hexadecimal characters). |
| generateMaze | * Sequence | Generates the maze the player will attempt with the generated/provided seed, and stores it in arrGameBoard. |
| applySeed | * Sequence | Applies the player's seed to the random number generation to ensure it is predictable and based entirely on the player's seed. |
| initializeGameBoard | * Sequence | Re-dimensions the arrGameBoard array to the required size as indicated in mazeSize. |
| recursePassage | * Sequence * Counted repetition * Binary selection | Recursively generates a passage through the maze based on the player's input seed, and stores it in arrGameBoard. |
| shuffleDirections | * Sequence * Pre-test repetition * Binary selection | Shuffles the array arrDirections to randomise the order in which the passages will be created (this is based on the seed and predictable - it's also the source of the randomisation in the recursive backtrack generation algorithm used in this program). |
| randBetween | * Sequence | Generates a random number in the given range of high and low (inclusive). |
| generateSeed | * Sequence * Counted repetition | Generates a random string of hexadecimal digits for a random seed. |
| getHighscores | * Sequence * Pre-test repetition | Reads the highscores from the highscores.txt file into the array of records arrHighscores. |
| readRecordIntoArray | * Sequence | Reads a raw record string from the highscores file, and parses it into a record before appending it to arrHighscores |
| sortHighscores | * Sequence * Pre-test repetition * Binary selection | Sorts arrHighscores into ascending order by game time for each of the three difficulties using the NESA bubble sort algorithm [O(n^2)]. |
| swap | * Sequence | Swaps the elements at the two given indices in the given array. |
| PlayGame | * Sequence | Runs the subroutines required in the gameplay - this is primarily graphical and involves drawing the maze and its elements, as well as the player/game data, handling the timer, and processing player movement data to efficiently redraw the required components of the screen when the player moves. This also checks when the player has won the game, and presents the frmGameOver form with all required information. |
| drawScreen | * Sequence * Binary selection | Handles the drawing of required elements onto the screen - both the maze and player (requiring graphics engine API interaction), and updating the game data (Windows Forms interaction). |
| renderMaze | * Sequence | Draws each of the three elements of the maze graphic (the cell walls, the colourings of the special cells, and the player) |
| drawCellWalls | * Sequence * Repetition * Binary selection | Loops through each cell, and draws the walls around it (but doesn't draw the walls if they've already been drawn for the adjacent cell). |
| drawPlayer | * Sequence | Draws the player at their current position. |
| drawSpecialCells | * Sequence | Draws the special cells in the maze (start and end cells) with different colourings. |
| updateTime | * Sequence | Updates the UI with the new game time (only called optionally during gameplay, not during the setup phase). |
| drawGameData | * Sequence | Updates the UI with the game information (name, initial time, difficulty, maze size, maze seed) once the gameplay form loads. Only called on setup (not during gameplay). |
| resetAndStartTimer | * Sequence | Resets then starts the game timer, and uses the gameTime variable as a handle to access the timer with. |
| gameLoop | * Sequence * Post-test repetition * Binary selection | Runs each 'tick' of the game (0.1 seconds, as this is fast enough to be able to handle every movement the player inputs due to human response time being >0.2 seconds on average) and receives player input, processes it, and updates the screen (after checking for a win). |
| queryTimer | * Sequence | Retrieves the current game time. |
| handleMovementInput | * Sequence * Multiway selection | Runs when a player movement input is detected; checks if the move is valid (doesn't collide with walls or maze boundaries) and if so, moves the player. |
| checkValidMove | * Sequence * Binary selection | Determines whether the player's desired movement will collide with cell walls or the maze boundaries. |
| applyMovement | * Sequence | Applies the desired movement (after validation) to the player's position. |
| checkWin | * Sequence * Binary selection | Checks whether the player is in the 'win' cell of the maze (and thus whether the game has ended). |
| ScoreGame | * Sequence * Binary selection | Handles the game ending (displaying player results as well as whether the player has achieved a high score), updating the arrHighscores and highscores.txt file as required. |
| displayGameResults | * Sequence | Shows the player a summary of their game, with their name, time, and the maze size, chosen difficulty, and maze seed. |
| checkIfHighscore | * Sequence * Binary selection | Checks if the player's time is faster than the slowest highscore for their difficulty (i.e. the player has achieved a highscore), passing back a Boolean control flag indicating this to only call the update and sort modules if necessary. |
| updateHighscores | * Sequence | Overwrites the last highscore in the arrHighscores array for the player's difficulty with the new player's game information. |
| displayHighscores | * Sequence | Displays the high scores to the player. |
| writeHighscoresToFile | * Sequence | Writes the highscores to the highscores.txt file if they have changed (i.e. if the player has achieved a highscore). |

## Function 1: validateName

### IPO Diagram

|  |  |  |
| --- | --- | --- |
| **Input** | **Process** | **Output** |
| playerName | Set result to true.  Check if the name is empty (length 0) or length is more than 16 characters. If so, set result to false. |  |
|  | If the name is a valid length, check that it's composed only of letters, numbers, or underscores. If not, set result to false. |  |
|  | Return the result. | result (true or false) |

### Data Dictionary

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data Item** | **Data Type** | **Size (bytes)** | **Number of Characters** | **Number of Decimal Places** | **Description** | **Example** |
| playerName | String | Between 0-16 (1 byte per char, max 16 chars). | 1-16 inclusive (or empty) | N/A | Holds the player name requiring validation - can be from 1-16 characters (or empty) | "ThisIsAMazeGame\_" |
| result | Boolean | 1 bit, padded to 1 byte. | N/A | N/A | Holds the result of whether the player's name is valid or not. | True (represented as 1 within the computer system) |

### Pseudocode

BEGIN validateName(playerName as String) returns Boolean

Let result = True

'REM Name must be between 1-16 chars inclusive

IF playerName is empty OR playerName.Length > 16 THEN

result = False

ELSE

Let i = 1

'REM Name must only contain alphanumeric or underscore chars

WHILE i <= playerName.Length AND result = True

IF NOT checkValidChar(playerName(i), "A-Za-z0-9\_") THEN

result = False

ENDIF

ENDWHILE

ENDIF

RETURN result

END ValidateName

## Function 2: generateSeed

### IPO Diagram

|  |  |  |
| --- | --- | --- |
| **Input** | **Process** | **Output** |
| System time (accessed internally, not passed as parameter) | Initialise the system pseudorandom number generator with the system time to ensure a random string. |  |
|  | Generate a random integer from 0-15 inclusive, then convert it to a hexadecimal character. Append it to a random seed string, and stop when this string is 10 characters in length. |  |
|  | Return the generated random seed | mazeSeed (10-digit hex string) |

### Data Dictionary

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data Item** | **Data Type** | **Size (bytes)** | **Number of Characters** | **Number of Decimal Places** | **Description** | **Example** |
| mazeSeed | String | 10 (10 chars) | 10 | N/A | Stores the generated maze seed (10-digit hex string) | "1b2f78ac4e" |
| digitCounter | Signed 8-bit integer | 1 | N/A | 0 | Stores the number of digits currently in the generated hex string (to avoid repeated length accesses which are performance-costly) | 7 |
| currentDigitDec | Signed 8-bit integer | 1 | N/A | 0 | Stores the randomly generated digit that will go in the current position, in its decimal integer format (range 0-15). | 13 |
| currentDigitHex | Character | 1 | N/A | N/A | Stores the hexadecimal digit representation of the number stored in currentDigitDec | 'D' |

### Pseudocode

BEGIN generateSeed() returns String

Randomize(System Time) 'REM Initially randomise the inbuilt pseudorandom number generator

Let mazeSeed As String = ""

Let digitCounter As Integer = 0

Let currentDigitDec As Integer = 0

Let currentDigitHex As Char = ""

'REM Generate 10 digits of a hex string

WHILE digitCounter < 10

'REM The number is generated as an integer, then converted to a hex character

Let currentDigitDec = randBetween(0, 15)

Let currentDigitHex = Hex of currentDigitDec

Let mazeSeed = mazeSeed & currentDigitHex

Let digitCounter = digitCounter + 1

ENDWHILE

RETURN mazeSeed

END generateSeed

## Subroutine: sortHighscores (and swap)

### IPO Diagram

|  |  |  |
| --- | --- | --- |
| **Input** | **Process** | **Output** |
| Array of high score records for a certain difficulty (passed by reference, as a pointer) | Nested pre-test loops. Outer loop controls each pass of the sort algorithm, inner loop iterates over each pair of elements and compares them. |  |
|  | If the times of the two elements are not in ascending order, then swap the pair of elements (the entire records, not just the times). |  |
|  | Once no swaps have been done on a completed pass (checked after each iteration of the outer loop), the array is sorted. | Modified array of high score records for a certain difficulty (mutated by reference/pointer) |

### Data Dictionary

|  |  |  |
| --- | --- | --- |
| **Input** | **Process** | **Output** |
| Array of high score records for a certain difficulty (passed by reference, as a pointer) | Nested pre-test loops. Outer loop controls each pass of the sort algorithm, inner loop iterates over each pair of elements and compares them. |  |
|  | If the times of the two elements are not in ascending order, then swap the pair of elements (the entire records, not just the times). |  |
|  | Once no swaps have been done on a completed pass (checked after each iteration of the outer loop), the array is sorted. | Modified array of high score records for a certain difficulty (mutated by reference/pointer) |

Note that for the data item 'arr', it is an array of recHighScore records. Please refer to the section [Arrays of Records](#_Arrays_of_Records) for more information about this data structure.

### Pseudocode

BEGIN sortHighScores(arr As recHighScore() By Reference)

'REM Standard NESA bubble sort algorithm

Let last As Integer = number of items in arr

Let swapped As Boolean = True

Let i As Integer = 1

'REM Outer loop controlling each pass through the algorithm

WHILE swapped = True

Let swapped = False

Let i = 1

'REM Inner loop controlling the pairwise comparison and swaps of elements in the array

WHILE i < last

IF arr(i).elapsedTime > arr(i + 1).elapsedTime THEN

'REM Swap subroutine that swaps arr(i) and arr(i+1)

swap(arr, i, i + 1)

Let swapped = True

ENDIF

Let i = i + 1

ENDWHILE

Let last = last - 1

END WHILE

END sortHighScores

BEGIN swap(arr As Any() By Reference, swapIdxA As Integer, swapIdxB As Integer)

'REM Swap arr(swapIdxA) with arr(swapIdxB), mutating original array by reference

Let temp As Any = arr(swapIdxA)

Let arr(swapIdxA) = arr(swapIdxB)

Let arr(swapIdxB) = temp

END swap