**Final Year Project Report**

**Full Unit – Final Report**

**AI MAZE GAME USING IMAGE PROCESSING WITH A\* PATHFINDING AND DFS ALGORITHM FOR MAZE GENERATION**

Jessada Khaoseengam

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science**

**Supervisor:** Matteo Sammartino



Department of Computer Science

Royal Holloway, University of London

December 07, 2023

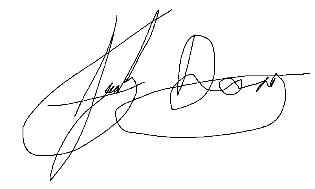
**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 4,940

Student Name: Jessada Khaoseengam

Date of Submission: 07/12/2023



Signature:

Table of Contents

[Abstract 1](#_Toc152808910)

[Chapter 1: Introduction 2](#_Toc152808911)

[1.1 Aims and Objectives of the Project 2](#_Toc152808912)

[1.2 Planning and Timescale 2](#_Toc152808913)

[Chapter 2: Survey of Related Information 5](#_Toc152808914)

[2.1 Methods of Maze Generations 5](#_Toc152808915)

[2.1.1 Prim’s Algorithm 5](#_Toc152808916)

[2.1.2 Kruskal’s Algorithm 5](#_Toc152808917)

[2.1.3 Depth-First Search Algorithm 5](#_Toc152808918)

[2.2 Choice of Programming Language 6](#_Toc152808919)

[2.3 User Interface 6](#_Toc152808920)

[2.4 Time Complexity 7](#_Toc152808921)

[Chapter 3: Software Engineering 8](#_Toc152808922)

[3.1 Testing 8](#_Toc152808923)

[3.1.1 User Interface 8](#_Toc152808924)

[3.1.2 Generate Maze 9](#_Toc152808925)

[3.1.3 User Input and Player Model 9](#_Toc152808926)

[3.2 Code Technical Explanation 10](#_Toc152808927)

[3.2.1 Maze Generation Class 10](#_Toc152808928)

[Chapter 4: Appendix 13](#_Toc152808929)

[4.1 Acronyms 13](#_Toc152808930)

[4.2 Glossary 13](#_Toc152808931)

[4.3 Diary 13](#_Toc152808932)

[Chapter 5: Bibliography 15](#_Toc152808933)

Abstract

In the era of rapid technological advancements, the fusion of image processing ([IP](#IP)) and artificial intelligence ([AI](#AI)) has opened new horizons for interactive gaming experiences. This project presents the development of a maze game that utilizes AI algorithms – the A\* algorithm for real-time pathfinding and Depth First Search ([DFS](#DFS)) algorithm for random maze generation. For current two-dimensional maze games, pre-generated stages are used as the main driver for gameplay which provides a limited content experience for the player and inhibits replayability. The player would then find themselves repeating the same stages, eventually finding the game mundane. A way to resolve this issue is to generate a series of random stages as they progress ensuring that no new level is the same[[3].](#References)To provide endless gameplay potential, a choice of three maze generation algorithms: Depth-first search, recursive division algorithm ([RDA](#rda)) and Prim’s can be considered to fulfill this objective. As the game is ran on a two-dimensional grid of limited size, DFS is the most suitable algorithm to pair with the A\* pathfinding algorithm as it can produce the maximal path lengths compared to the other algorithms[[1]](#References) it also works seamlessly with two-dimensional grid applications which is helpful when developing on an Python platform. With the use of DFS, procedurally generated mazes are truly random and can offer unique mazes for the player to solve.

The main objective of this project is to produce an engaging and easy to play experience where players can navigate through a series of randomly generated mazes whilst racing against the clock with the limited option for AI assistance when stuck on a stage. To achieve this, the application of the A\* algorithm is used to calculate the shortest path from start to finish upon command, in real time. Since the game will be operating on a finite graph, the A\* algorithm executes by managing two scoring mechanisms where one represents the score for reaching a node on a current route and another heuristic score for selecting the next node to visit[[2]](#References). This mode of operation ensures efficiency as with each iteration a path to the final node is established. This aspect is an essential driver for gameplay features. The player is tasked to complete as many stages as they can before the timer runs out. With each completed stage the timer is renewed, and the cycle of gameplay is repeated until the timer runs out. To allow greater playability and intrigue, a score of how many stages the player passed will be recorded where they aim to beat their highest score on their future plays encouraging further attempts to beat their high score[[5]](#References). The game also offers another feature allowing the player to create their own mazes on paper where the image processing feature of the game will map the maze, allowing the pathfinding algorithm to solve it.

The fundamental element in developing the image processing feature of the game is the use of OpenCV in conjunction with [VS Code](#VSCode) [IDE](#IDE) using [Python](#Python) for core game elements. IP is used to map the custom maze the player creates where image and edge detection will plot the walls of the maze and place start/finish nodes allowing the pathfinding algorithm to process. Since capturing an image may not always be perfect, IP techniques can manipulate the image to reduce noise such as Gaussian blur[[4]](#References) and Image histograms for wall detection to ensure the maze is suitable for mapping and applying the pathfinding algorithm. This aspect of the game will be the next focus after developing the base game.

# Introduction

## Aims and Objectives of the Project

The Project’s main purpose is to create an interactive maze puzzle game applying various techniques of artificial intelligence to enable assistive capability while producing an efficient maze generator to maximise user engagement.

The first step to accomplishing this goal is to establish a UI where the user can interact with the game providing visual feedback. It will consist of a start, solve, and quit buttons contained within a small window where the game will run. When selecting play the user will initiate the main game where they must reach the end waypoint objective whilst navigating through the maze. Upon completion they will progress onto the next stage with a slightly more complex maze structure. For every completion the maze generator will produce larger and more complex mazes. The user must locate the objective before time runs out which replenishes with each completed stage. If the timer runs out the score of every completed stage is recorded and the game will prompt the user to play again or return to the main menu.

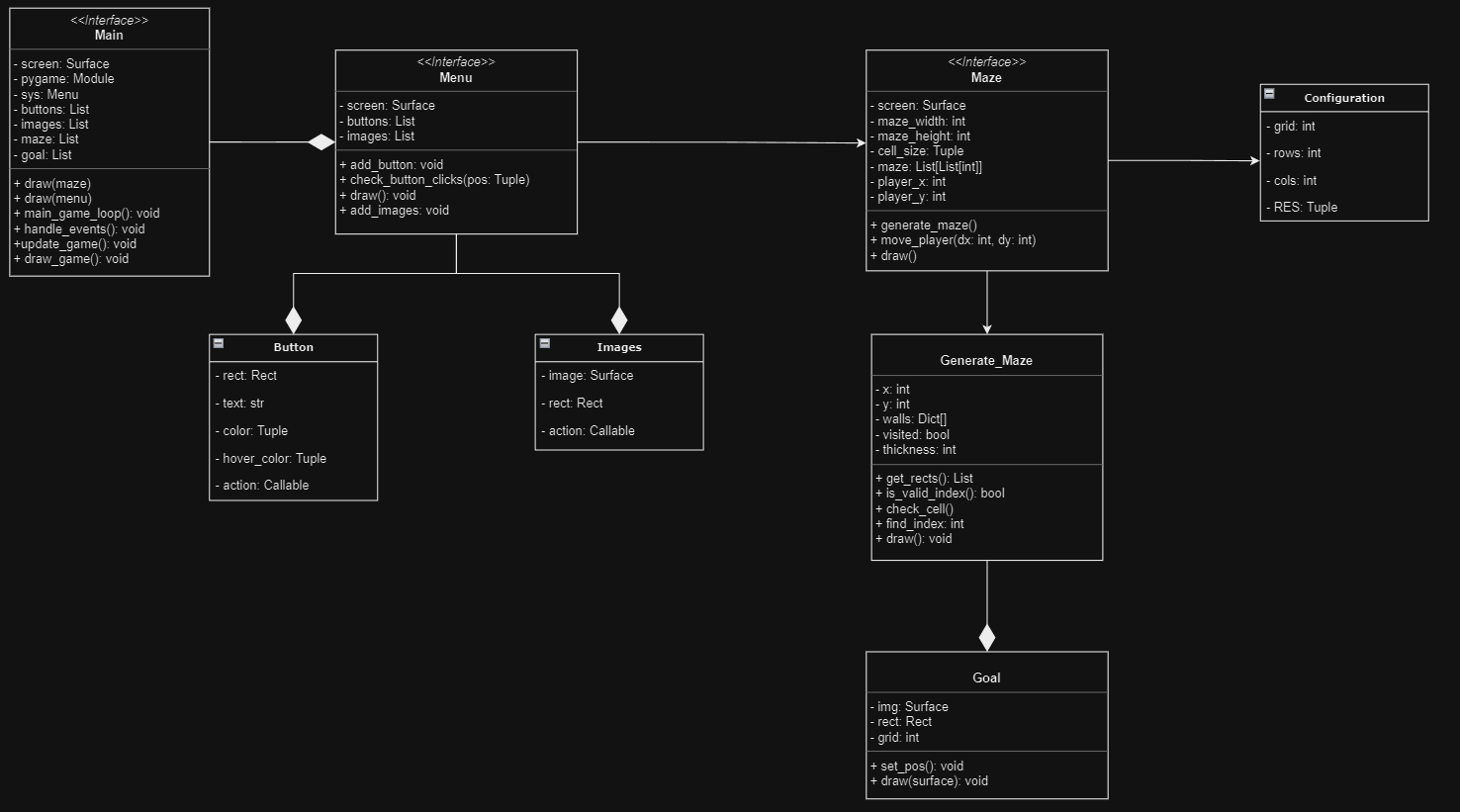
The solve button launches the IP aspects of the program where they are prompted to present a premade maze to their webcam/camera in which the pathfinding algorithm will display a path highlighting the solution from a starting point to another end point.

## Planning and Timescale

During term one I will primarily concentrate on establishing the foundations of the game and implementing the various AI algorithms. In term two I can then focus on adding the image processing feature and further cleanup code. Shown in [*figure 1*](#figure1)is a Unified Modeling Language ([UML](#UML)) diagram which blueprints the basic overview of the structure of the code that is developed.

The planning grid below provides structural organization showing me the different features of the project and helps me prioritize key elements. Starting with a UI I can build upon this foundational requirement and progressively add other modules from then on. Using created sketches, I can visualize the requirements needed to create the ideal user interface and identify any areas of improvement to maximize user experience.

In terms of technical requirements, creating software involving diagrams like flowcharts can provide me with visual representation to show the flow and structure of a certain component in development. By understanding the logic at a glance, it makes it easier to comprehend the complex processes I need to face. When dealing with algorithms, visually representing them helps in refining the logic and identifying any potential issues early in the development process.



***figure 1.*** *Initial UML plan*

**Term 1**

|  |  |
| --- | --- |
| Week 2 | Research required software and tools  Find the resources necessary to build base game |
| Week 3 | Research necessary algorithms  Understand how they work and how to apply to code |
| Week 4 | Implement foundational app structure  Start coding base game creating basic UI and simple stage |
| Week 5-6 | Generate basic maze stages for user playability  Use AI to generate more sophisticated stages |
| Week 7-8 | Apply pathfinding algorithm to maze and test  Start trialing AI and test playability and function for the player |
| Week 9 | Research Image processing features  Understand and learn OpenCV, its features and how it integrates in game |
| Week 10 | Optimize game  Fix any prominent issues and ensure smooth functionality |
| Week 11 | Work on presentation and interim report |

**Term 2**

|  |  |
| --- | --- |
| Week 1-2 | Develop image processing feature structure  Apply basic structure for IP to game and ensure compatibility with game |
| Week 3-4 | Implement and test mazes using image processing feature  Use webcam to test limitations of different mazes in various positions |
| Week 5-6 | Merge Image processing features to base game  Implement pathfinding code to mapped image |
| Week 7-8 | Test playability and bugfix |
| Week 9 | Refine [UI](#UI) and cleanup code |
| Week 10 | Evaluate game as whole  Ensure seamless gameplay and optimal performance |
| Week 11 | Final report preparation |

# Survey of Related Information

## Methods of Maze Generations

To produce truly random mazes, I had to decide between various generation algorithms to achieve the most optimum result in efficiency. I can choose between Kruskal’s, Prim’s, and Depth-first search and compare their capabilities to determine which one is the most effective and efficient in producing the ideal maze structure for the game. Conclusively I chose Depth-First Search as it is better suited to my requirements. Below explains the characteristics of each algorithm and compares each one.

### Prim’s Algorithm

Compared to other algorithms this one tends to generate a better maze quality as it creates more structured paths allowing easier navigation and less dead ends to add more complexity. This is done by marking an arbitrary cell and setting it as the initial cell. The walls of the cell are added to a list. Randomly choose a wall from the wall list and if only one of the two cells separated by this wall has been visited make, make the wall a passage and mark the unvisited cell as part of the maze. Then add the neighbouring walls of the cell to the list. Repeat the expansion step until the desired number of the cells is met. However, a downside to implementing this algorithm is the deterministic nature in its basic form which can lead to predictable maze creation. It also has an edge bias where it selects edges that lead to less explored regions of the maze. In comparison to the other selected algorithms, there is limited configurability which gives us less control over the maze characteristics. After reviewing this method of maze generation, I concluded that it doesn’t meet the requirements for my goals.

### Kruskal’s Algorithm

This algorithm uses a minimum spanning tree that can be modified to fit the criteria of random maze generation. It produces uniform path lengths which results in balanced structure and is not biased towards long corridor formations. This is done by randomly shuffling walls in a list removing the wall and merging the sets, making them a part of the same tree. With these simple steps it provides easy implementation. However, like Prim’s the algorithm tends to produce deterministic structures and edge bias towards horizontal and vertical passages due the way the edges are selected. This can result in mazes appearing more grid-like which can ruin the aesthetic of the maze. The algorithm involves maintaining a disjoint-set data structure which can consume additional memory, especially for larger scale mazes[[7].](#References).

### Depth-First Search Algorithm

I chose this algorithm primarily due to is capability of configurable randomness by tweaking the order in which neighbours are visited and choosing random starting points. By configuring the number of neighbours of the cells I can adjust the complexity of the structure of the maze which allows me to set different difficulties to the maze. It also features natural dead-end creation whilst remaining memory efficient. This is a significant factor as larger maze structures can be produced in comparison to Prim’s or Kruskal’s.

The way DFS can be implemented in the program is by starting with an initial grid of x cells and mark them as unvisited. Choose a random starting cell and then mark it as visited. Create a stack and push the start cell on the stack. Pop the cell from the stack and randomly shuffle the list of neighbouring unvisited cells. For each neighbour create a passage by removing the walls of each grid cell based of an x and y coordinate. Mark the neighbour as visited then push the neighbour onto the stack. If a cell has no unvisited neighbours backtrack by popping cells from the stack until a cell with unvisited neighbours is encountered. Repeat this main loop until the stack is empty and all cells have been visited. This method of backtracking ensures that each individual cell is visited and processed to form a wall of a maze.

## Choice of Programming Language

Since the program is a game, I needed to find the most suitable language and framework for 2D-based objects and fulfilment to my objectives. The options were C++, C#, Java, Python and Lua. Python with the Pygame library was the optimal choice as other options like C# using Unity are more focused on resource intensive 3D games which does not meet the requirements to achieve my goals. Python is known for its simplicity and readability. It provides clean and concise syntax which makes it easy to understand and learn its functionality. Python also provides rapid deployment capability which allows me to create prototypes and develop at a rapid pace. This is particularly favourable for small/medium sized projects like a maze game. With the use of extensive libraries and frameworks, development is a lot simpler especially with good documentation and community support which can help find solutions to common issues. I primarily chose Python as it is a scripting language. This grants the ability for quick iteration and testing. As I progress during development, making frequent changes and experimenting will not affect the critical structure of the program.

## User Interface

Employing a theme of futuristic sci-fi elements was the decisive factor between a fantasy or noir theme as it would complement a modern and simplistic interface allowing better user interaction. With this theme I can use a variety of colour to spark intrigue. I chose a dark background with neon features which allows buttons and text to stand out whilst being easy on the eyes. Aiming to maintain a minimalistic display the use of buttons is displayed central the screen with large boxes and display text. With sufficient spacing between the buttons, this can help prevent the user from any misinput. On any menu screen all keyboard interaction are disabled and the user can only navigate the buttons via mouse input. Since the programs visual structure relies on using only one screen, objects such as the menu and maze game must share this screen and be drawn depending on the situation. By disabling keyboard interactions which are associated with the game, menu interactions via mouse are isolated and control the menu objects independently mitigating any risk of accidently controlling the game in the background without notice.

Currently as a placeholder the player sprite is shown as a white rectangle (shown in [*figure 4*](#figure4)*)* to assist with collision detection between the walls and the goal but will change to a more animated sprite in later development. Every button in the program has a border outline which lights up blue whenever the mouse cursor hovers within its vicinity. This can assist the user in navigating the menu and provides good user interface design as it provides responsive feedback to the user. A countdown timer is also displayed to indicate the remaining time left on the current stage to alert the user on the situation. The current stage indicator also serves the same purpose of informing the user on their progress. For better detail, Pygame can make use of animations which can controlled using event handler where I have the option to animate player movements, scene elements and backgrounds. Currently there is a dynamically shifting background upon first launch of the game to provide better aesthetic appeal.



***figure 2.*** *Current start menu*

## Time Complexity

To provide better context in the algorithm used in the random maze generation I need to understand its time complexity. Typically, in a DFS algorithm, the time complexity is dependent on the representation of the graph and the specific implementation details. In a standard adjacency list representation where each vertex maintains a list of its adjacent vertices, the time complexity is often expressed as

V in the context of the program refers to the number of grid cells and the initialization results in a time complexity of this is because each cell is visited exactly once during instantiation. In the worst case each cell is visited once, and each edge is examined once. In a grid-based maze where each cell has four potential neighbours, the number of edges (E) is proportional to the number of vertices (V). In the removal of walls between adjacent cells, this is done once for each visited cell and each wall removal takes constant time. In the worst case, each cell is visited once leading to a time complexity of [[8]](#References).

Its important to note that the constants hidden by the big-O notation can vary based on factors such as the specific grid size used which can change based on the difficulty of the maze and the efficiency of the list data structure used in implementation. The analysed time complexity breakdown provides me with a high-level understanding of the algorithm’s performance in terms of the number of cells and edges in the grid.

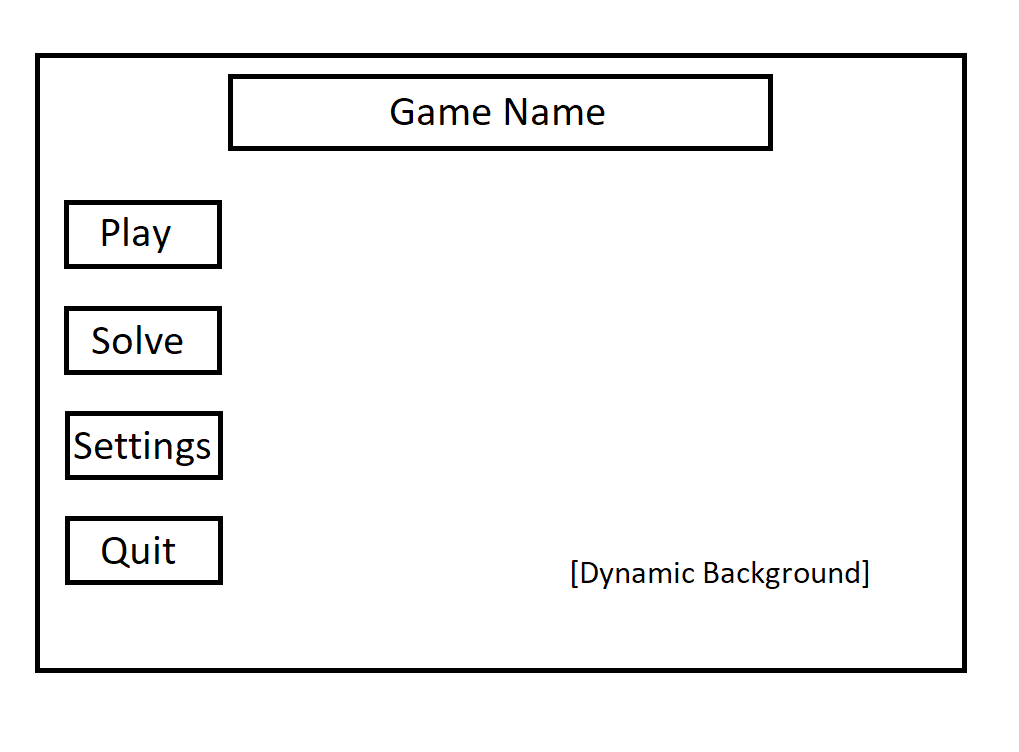
# Software Engineering

## Testing

I experienced technical difficulties when trying to implement unit testing. It was not fully compatible with OpenCV and the Pygame modules so as an alternative I produced my own testing process.

### User Interface

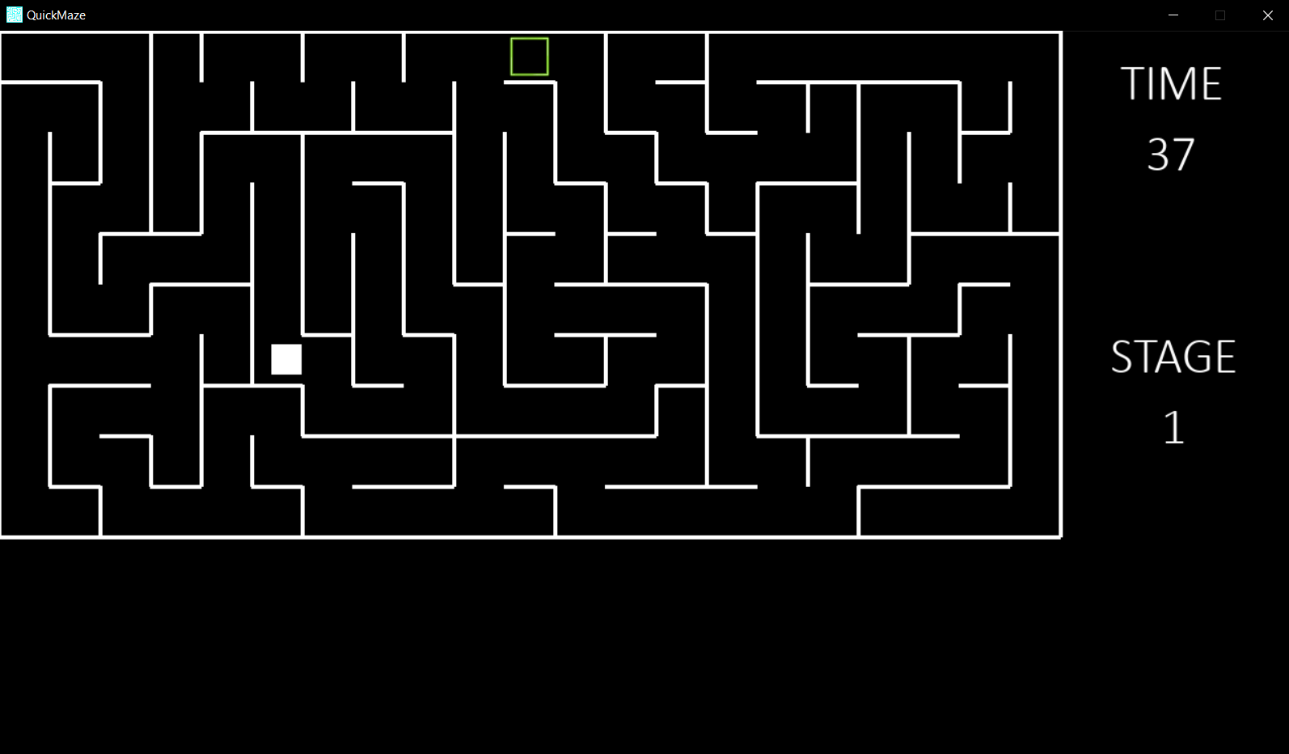
Initially, taking inspiration from the preexisting maze-like game “Pacman” I decided to adopt the theme of neon on black due to its visual appeal. I created an initial design plan for the UI which I used as a basis for design of the main menu screen leading towards the game itself. In [*figure 3*](#figure3) the initial design sketch is shown. The result produced is shown in [*figure 2*](#figure2)where I strayed slightly from the design plan by placing the buttons more central to the screen to scale appropriately for different resolution options which will be added in a future development iteration.



***figure 3.*** *Initial menu sketch*

To test the functionality of each button when pressed the button should update the screen to show the contents of each respective text. I also had to test the input reading of the program where only mouse input is enabled when presented with a menu screen and isolate the game controls only when play is pressed.

In [*figure 4*](#figure4), the in-game UI is shown. The maze structure is displayed on the top left of the screen to allow the screen to accommodate other elements of the game like time and stage in the empty spaces. Noticeably there is a large space shown below the maze where I will produce option buttons “Return to Menu” and “solve” in a future development iteration.



***figure 4.*** *In-game User Interface*

### Generate Maze

Testing the output quality of the DFS is a vital step to ensuring the success of the gameplay. The method firstly involves creating a grid instance of default size of 100. This should produce a relatively easy maze to navigate. I must ensure that all paths within the maze were accessible and that the walls are truly randomly generated. I created more than 10 instances of the same grid size and analysed the structure ensuring that the testing criteria is met. This process is repeated for larger grid sizes where I also analysed the output checking for any faults in wall placement and path obstructions. During my testing I have discovered that the generation algorithm performed optimally for values between 50 – 120 grid size. Exceeding beyond this range results in a very simplistic maze only generating at most 5 walls within the grid. Going under this range results in the maze producing an overwhelming number of walls compacted within the game screen which in turn scales the player model to a miniscule proportion making the game unplayable.

### User Input and Player Model

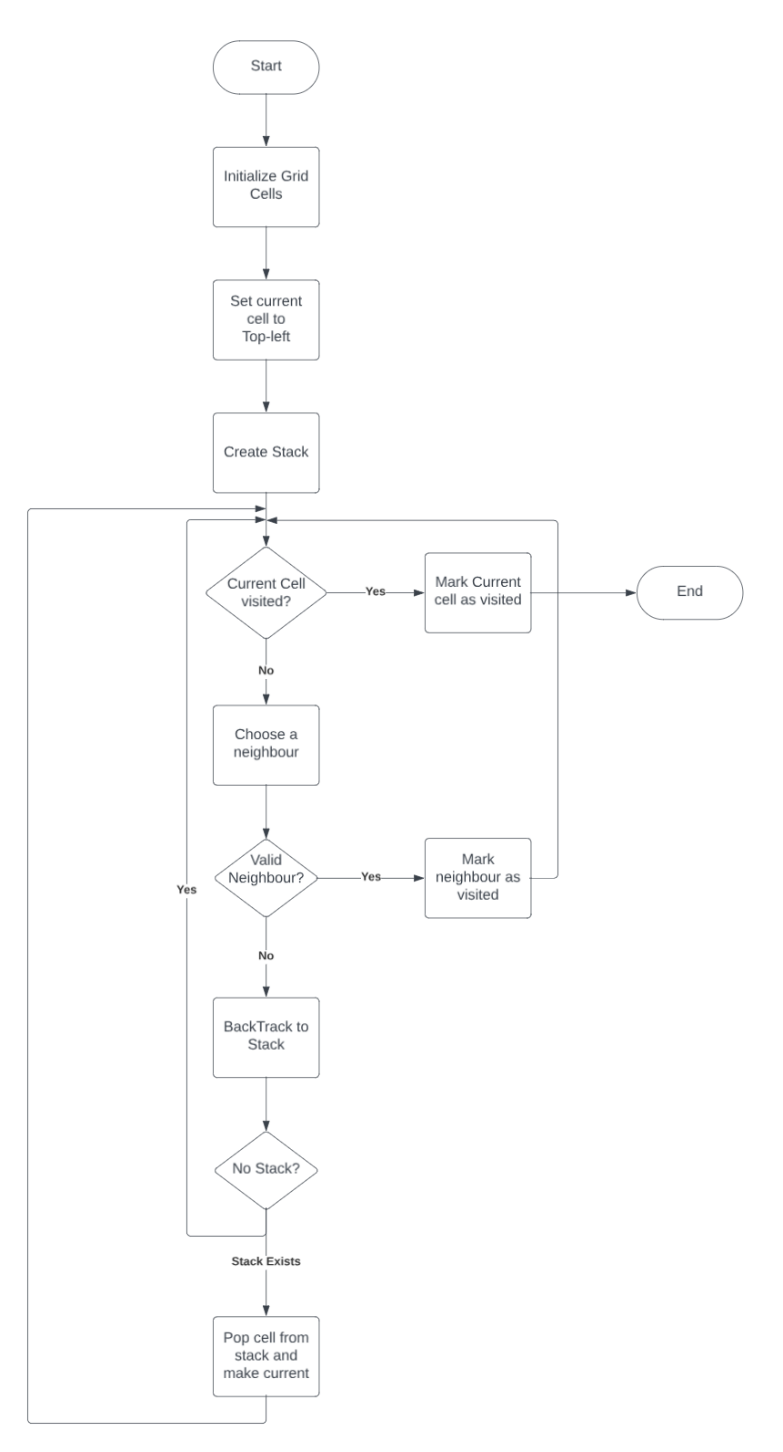
To ensure smooth player control and better user experience I must test all input functionality from the user in a menu state as well as the game state. During menu interaction, the player can interact with the game using only the mouse to navigate the program. I established certain criteria and made sure each aspect is suitably achieved. Regarding accessibility testing I ensured that the game can only be interacted with a mouse and keyboard and no other forms of alternative methods. During gameplay I stress tested the player movement within the maze to ensure the games performance can handle rapid player input. Since the Pygame event handler only has the keys ‘w’, ‘a’, ‘s’, ‘d’ mapped to player movement, there is no need for any form of error handling during gameplay. I also tested the collision detection between the player model and the walls of the maze to ensure that the player was moving within its set surroundings and not exceeding the limited map size. This was done by testing on different map configurations and attempting to ‘escape’ the maze.

## Code Technical Explanation

### Maze Generation Class

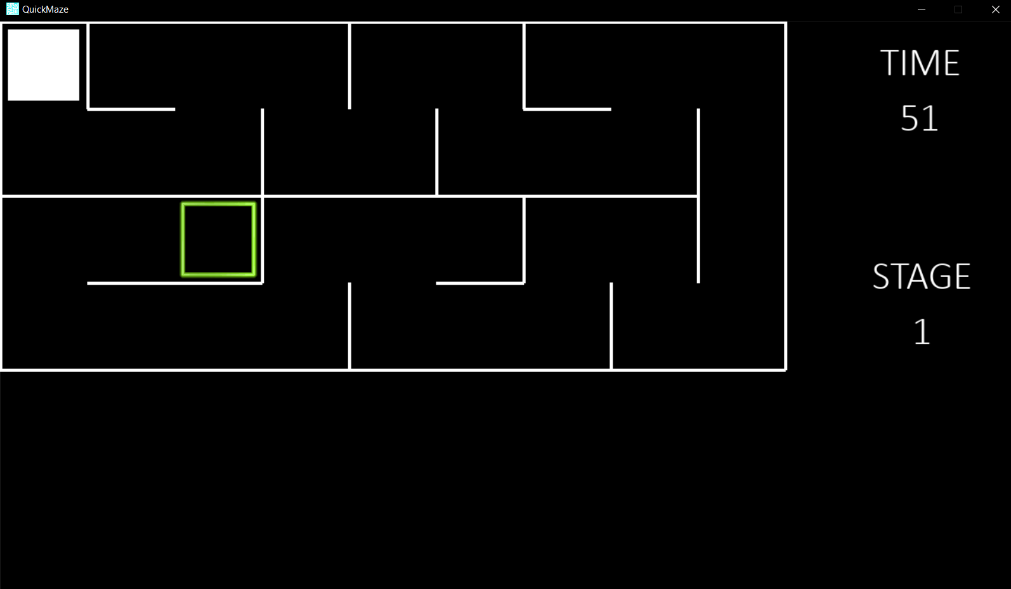
In [*figure 5*](#figure5)a flowchart is used to describe the main processes of the maze generation class. This class represents cells in a maze grid. It contains information about the cell’s position via ‘x’ and ‘y’ wall configuration. A “get\_rects” method is used to return a list of rectangles representing the walls of the cells which are also used to draw the maze. The generator then checks given coordinates to see if they comply within the valid range of the maze grid. If they fall within range True is returned and parsed to a find index method which computes the index of a cell based on a list. Then it checks the next value and finds any unvisited neighbouring cells of the current cell. It returns False if one does not exit or returns a random valid neighbour. Then when called a draw method is established used to create the visual structure of the maze.

A generate function is used to create a list of grid cells representing the entire maze. it uses a while loop and continues until all the cells are visited. Then to create paths of the maze a ‘remove’ function is used which takes two cells ‘current’ and ‘next’ and removes the walls between them based on their relative positions (top, bottom, left, right). This maze generation class performs optimally for grid sizes 50 – 120 with a 50 creating a larger complex maze structure and 120 with a simplistic structure with less walls. An instance of this class is created when maze() is called from the main program loop. When the Pygame event handler is initialized and the condition ‘play’ is set to true, then an instance object of this class is created. The visual aspect of the maze is drawn on the screen whenever the draw() method of the class is called. Examples of the range of difficulties of ‘easy’, ‘medium’, ‘hard’ are shown in [*figure 6*](#figure6)*,* [*figure 7*](#figure7)and [*figure 8*](#figure8)respectively. By simply changing the value of the grid cell variable every visual element involving the gameplay scales correctly including the maze, player, and target goal to allow progressively challenging environments with every stage completion.

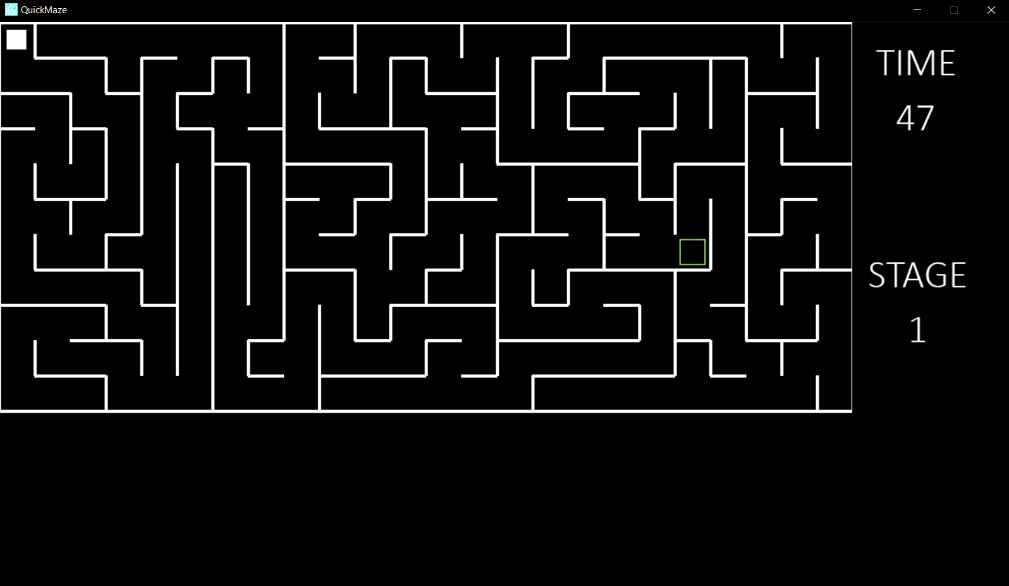


***figure 5.*** *Maze generation flowchart*

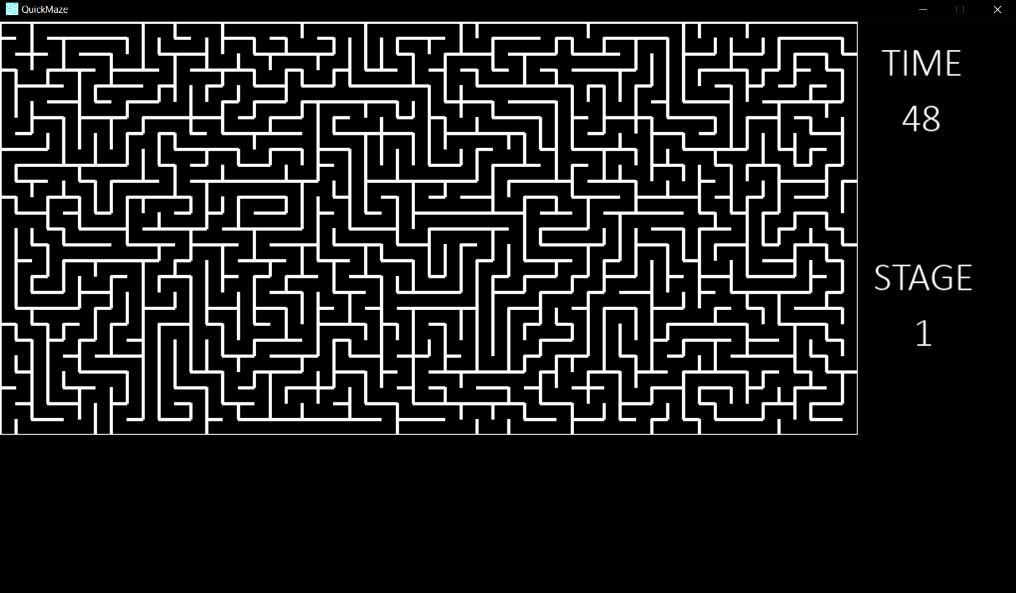
Using the flowchart shown in [*figure 5*](#figure5), I created a class function which complied with the design of the flowchart to assist with the implementation of the DFS algorithm used for the random generation in the game. Recursive backtracking is used in DFS where the algorithm recursively explores neighbouring cells and randomly shuffles the order in which the neighbours are considered. With each iteration a new path is carved b removing a wall between the current cell and the neighbour. The backtracking in this instance involves meeting the condition in which all neighbours have been visited or there are no unvisited neighbours[[6]](#References). If either case is met then backtrack by returning from the recursive calls, effectively undoing the path carving for the current cell. When a dead-end is reached (a cell with no unvisited neighbours), then algorithm backtracks to the previous cell in the call stack exploring other paths from that point. This process creates a maze with intricate pathways and dead ends.



***figure 6.*** *Easy difficulty*



***figure 7.*** *Medium difficulty*



***figure 8.*** *Hard difficulty*

# Appendix

## Acronyms

**AI** Artificial Intelligence

**IP** Image Processing

**UI** User Interface

**DFS** Depth First Search

**RDA** Recursive Division Algorithm

**IDE** Integrated development environment

**AR** Augmented Reality

**UML** Unified Modelling Language

## Glossary

**OpenCV** Open-source computer vision, machine learning library for vision-based tasks

**Python** High level programming language used in various fields

**VS Code** IDE with versatile features and python support via extensions

**GitLab** Web based version control platform to manage source code

**UML Model** A visual representation of system architecture and design

**Pygame** Library module use to handle graphics, sound and input

## Diary

October 20th

After researching the ideal platform and programming language to be used, I decided that python used on visual studio code would work suitably for the specification I had planned. By using the pygame module I can create a basic UI as a starting base point. Using the pygame module will allow me to create the start menu of the game and include buttons used to launch the game.

October 24th

After creating a new interface, I implemented some functionality to the game by adding a play and quit button and then started research into implementing the maze feature to the game. I must now think about the code structure and as to how to implement the actual game to the main menu when play is pressed.

October 29th

I completely restructured my code making it more object oriented. By creating object instances of menu elements and buttons I can just reuse the same screen and draw the necessary elements required making the program more efficient but at the cost of restructuring and reformatting the code to make it runnable

November 3rd

I improved the visual aesthetic of the main menu to fulfil the goal of user engagement at an early stage. I also amended the maze generation algorithm to scale correctly on the screen allowing more space for other UI elements like score, exit button and stage information indicators.

November 11th

I began developing player controls allowing navigation around the maze and establishing collider boxes which can enclose the player within the walls. Issues were found where when the player completes a stage, and the maze structure is regenerated the player sprite would spawn inside a wall where the collision boxes were activated restricting any player movement. This was fixed by also regenerating the player sprite in a random place ensuring collider boxes are within the playable area.

November 19th

I began developing the random maze generator by implementing the DFS algorithm. So far this has been the most challenging aspect of development as some instances of the algorithm would produce a solid grid rather than a maze structure which perplexed me suggesting that the walls were not randomly removed when visiting each new neighbour cell. However, the maze scales correctly on the screen and matches the collider boxes generated which allows the player to roam the maze.

November 26th

I fixed and cleaned up the maze generator which now produces truly random maze structures with each regeneration. I added a countdown timer, stage number and the player goal which Is randomly spawned within the maze. Now when the player collider touches the goal collider a new structure is generated, and all game elements are respawned as well as the timer incrementing by 10 seconds and stage statistic by 1.

# Bibliography

[1] A. Kozlova, J. A. Brown and E. Reading, "Examination of representational expression in maze generation algorithms," 2015 IEEE Conference on Computational Intelligence and Games (CIG), Tainan, Taiwan, 2015, pp. 532-533, <https://doi.org/10.1109/CIG.2015.7317902>

[2] E. R. Firmansyah, S. U. Masruroh and F. Fahrianto, "Comparative Analysis of A\* and Basic Theta\* Algorithm in Android-Based Pathfinding Games," 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Jakarta, Indonesia, 2016, pp. 275-280, <https://doi.org/10.1109/ICT4M.2016.063>

[3] P. H. Kim and R. Crawfis, "The quest for the perfect perfect-maze," 2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES), Louisville, KY, USA, 2015, pp. 65-72, <https://doi.org/10.1109/CGames.2015.7272964>

[4] R. S. Deepthi and S. Sankaraiah, "Implementation of mobile platform using Qt and OpenCV for image processing applications," 2011 IEEE Conference on Open Systems, Langkawi, Malaysia, 2011, pp. 284-289, <https://doi.org/10.1109/ICOS.2011.6079235>

[5] Buday R, Baranowski T, Thompson D. Fun and Games and Boredom. Games Health J. 2012 Aug;1(4):257-261. <https://doi.org/10.1089/g4h.2012.0026>

[6] D. Ghosh and U. Bhaduri, "A simple recursive backtracking algorithm for knight's tours puzzle on standard 8×8 chessboard," 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, India, 2017, pp. 1195-1200, <https://doi.org/10.1109/ICACCI.2017.8126004>

[7] Mhd. Furqan, Herman Mawengkang, Opim Salim Sitompul, A.P.U. Siahaan, Muhammad Donni Lesmana Siahaan, Wanayumini, Nazaruddin Nasution, Sriani. “A Review Of Prim And Genetic Algorithms In Finding And Determining Routes On Connected Weighted Graphs” 2018 International Journal of Civil Engineering and Technology (IJCIET), Medan Indonesia, <https://www.researchgate.net/profile/Mhd-Furqan/publication/342766283_Article_ID_IJCIET_09_09_170_Nazaruddin_Nasution_Sriani_A_Review_of_Prim_and_Genetic_Algorithms_in_Finding_and_Determining_Routes_on_Connected_Weighted_Graphs/links/5f05b0a1299bf188160a4abe/Article-ID-IJCIET-09-09-170-Nazaruddin-Nasution-Sriani-A-Review-of-Prim-and-Genetic-Algorithms-in-Finding-and-Determining-Routes-on-Connected-Weighted-Graphs.pdf>

[8] Paolo G. Franciosa, Giorgio Gambosi, Umberto Nanni, The incremental maintenance of a Depth First-Search tree in directed acyclic graphs, Information Processing Letters, Volume 61, Issue 2, 1997, Pages 113-120, ISSN 0020-0190, <https://doi.org/10.1016/S0020-0190(96)00202-5>